PyTrx

Release 1.2.4

Penelope How

Apr 28, 2023

CONTENTS:

PyTrx (short for 'Python Tracking') is a Python object-oriented toolbox created for the purpose of calculating real-world measurements from oblique images and time-lapse image series. Its primary purpose is to obtain velocities, surface areas, and distances from imagery of glacial environments.

CHAPTER

ONE

INSTALLATION

1.1 PyTrx set-up

PyTrx has been coded with Python 3 and has been tested on Linux and Windows operating systems (it should also work on Apple operating systems too, it just hasn't been tested). PyTrx was originally written using a Linux operating system, so the inputted file path structures given in the example scripts may differ between operating systems and it is therefore advised to check file path structures before running these.

PyTrx can either be downloaded directly from the GitHub repository, or installed PyPI package manager (pip).

1.2 Installing PyTrx through pip

PyTrx is available through pip and can be installed with the following simple command:

```
pip install pytrx
```

Be warned that there are difficulties with the GDAL package on pip, meaning that gdal could not be declared explicitly as a PyTrx dependency in the pip package compiling. Please ensure that gdal is installed separately if installing PyTrx through pip. You should be able to create a new environment, install GDAL and the other dependencies with conda, and then install PyTrx with pip.

```
conda create --name pytrx python=3.7

conda install gdal opencv pillow scipy matplotlib spyder

pip install pytrx
```

If you still run into problems then we suggest creating a new conda environment from the .yml environment file provided in the PyTrx repository, as described here. This includes a fresh install of PyTrx.

```
conda env create --file environment.yml
```

To check that PyTrx is working, some simple unit tests can be run from the command line.

```
python -m unittest PyTrx.Area PyTrx.CamEnv PyTrx.DEM PyTrx.Line PyTrx.Velocity
```

Additionally, open a Python console or IDE such as Spyder, and try to import PyTrx, PyTrx's help guide, and one of PyTrx's modules as a test.

```
import PyTrx

help(PyTrx)

from PyTrx import Area
```

If PyTrx is working correctly, the unit tests should run successfully and the help statement should print PyTrx's metadata, including PyTrx's license, a brief description of the toolset, and its structure. If this does not work and throws up an error, it is likely that the package dependencies are invalid so reconfigure them and then try again.

1.3 Cloning PyTrx from GitHub

PyTrx can be cloned from PyTrx's GitHub repository, or using git.

```
git clone https://github.com/PennyHow/PyTrx.git
```

The repository can be installed in your python environment by navigating to the top level of the repository and using pip to install the local package.

```
python -m pip install -e .
```

Or you can use PyTrx in a python environment with the installed dependencies. We recommend installing PyTrx's dependencies with conda.

```
conda install gdal opencv pillow scipy matplotlib spyder
```

CHAPTER

TWO

GETTING STARTED

PyTrx comes with working examples to get started with. These scripts are available in the PyTrx repository here. These examples are for applications in glaciology, which can be adapted and used. We hope these are especially useful for beginners in coding.

2.1 Automated detection of supraglacial lakes

In this example, we will derive changes in surface area of supraglacial lakes captured from Kronebreen, Svalbard, for a small subset of the 2014 melt season. This example can be found in KR_autoarea.py.

We will automatically detect water on the glacier based on differences in pixel intensity and corrected for image distortion; using images from Kronebreen camera 3 and the associated camera environment.

First, we need to import the PyTrx packages that we are going to use.

```
from PyTrx.CamEnv import CamEnv
from PyTrx.Area import Area
from PyTrx.Velocity import Homography
import PyTrx.FileHandler as FileHandler
from PyTrx.Utilities import plotAreaPx, pltAreaXYZ
```

We load our camera environment and masks (for feature extraction and image registration), and set the paths to our input images and output folder.

```
# Define camera environment input file
camdata = '../Examples/camenv_data/camenvs/CameraEnvironmentData_KR3_2014.txt'

# Define feature detection and registration mask files
camamask = '../Examples/camenv_data/masks/KR3_2014_amask.jpg'
caminvmask = '../Examples/camenv_data/invmasks/KR3_2014_inv.jpg'

# Define image folder
camimgs = '../Examples/images/KR3_2014_subset/*.JPG'
```

Next, we create a CamEnv object using our previously defined camera environment text file which contains information about the camera location and pose, and file paths to our DEM, ground control point positions, camera calibration coefficients, and reference image.

```
# Create camera environment
cameraenvironment = CamEnv(camdata)
```

If certain camera environment parameters are unknown or guessed, then PyTrx's optimisation parameters can be used to refine the camera environment and improve the georectification. This refinement is conducted based on the ground control points.

In this case, the camera pose (yaw, pitch, roll - YPR) is unknown, so we will use the optimisation routine the refine the YPR values.

```
# Set camera optimisation parameters

optparams = 'YPR'       # Flag to denote which parameters to optimise:
                        # YPR=camera pose; INT=intrinsic camera model;
                        # EXT=extrinsic camera model; ALL=all camera
                        # parameters

optmethod = 'trf'       # Optimisation method: trf=Trust Region
                        # Reflective algorithm; dogbox=dogleg algorithm;
                        # lm=Levenberg-Marquardt algorithm

# Optimise camera
cameraenvironment.optimiseCamEnv(optparams, optmethod, show=True)
```

In order to make measurements from the images, we need to ensure that motion in the camera platform is corrected for (otherwise we will see jumps in the positions of our detected lakes when the camera platform moves).

We will use PyTrx's Homography object to track static features in the image and identify camera platform motion. We can subsequently use these movements to create a homography model and correct for this motion.

```
# Set homography parameters
# Homography tracking method - sparse or dense tracking
hgmethod='sparse'

# Pt seeding parameters (max. pts, quality, min. distance
hgseed = [50000, 0.1, 5.0]

# Tracking parameters (window size, backtracking threshold, min. num of pts)
hgtrack = [(25,25), 1.0, 4]


# Set up Homography object
homog = Homography(camimgs, cameraenvironment, caminvmask,
                   calibFlag=True, band='L', equal=True)

# Calculate homography
hg = homog.calcHomographies([hgmethod, hgseed, hgtrack])

# Compile homography matrices from output
homogmatrix = [item[0] for item in hg]
```

Now we have our homography model, we can look at detecting lakes in the images. As we want the lake features as polygons, we will use PyTrx's Area object to automatically identify these features. First, we will initialise the object with our images, camera environment object, homography model, and three flags denoting whether the images should be corrected for lens distortion, which pixel band should be used in the detection process (red, green, blue or grayscale), and whether the pixels in the images should be adjusted with histogram equalisation.

Lakes will be identified based on the difference in pixel intensities between the water and adjacent ice. The time-lapse images will also be enhanced to aid in identifying them.

```
# Set parameters to initialise Area object
# Detect with corrected or uncorrected images
calibFlag = True

# Pixel band to carry forward ('R', 'G', 'B' or 'L')
imband = 'R'

# Images with histogram equalisation or not
equal = True

# Set up Area object
lakes = Area(camimgs, cameraenvironment, homogmatrix, calibFlag, imband, equal)
```

We can set a number of detection parameters in our Area object to aid in the automated identification of lakes, including image enhancing, image masking, and setting athreshold for the number of detected polygons that will be retained.

```
# Set image enhancement parameters
diff = 'light'
phi = 50
theta = 20
lakes.setEnhance(diff, phi, theta)

# Set mask and image number with maximum area of interest
maxim = 0
lakes.setMax(camamask,maxim)

# Set polygon threshold (i.e. number of polygons kept)
threshold = 5
lakes.setThreshold(threshold)
```

Following this, we will use a pre-defined pixel value range to detect lakes from the images. In this case, pixel values between 1 and 8 will be classified as water. The calcAutoAreas function will then be executed to detect water through all the time-lapse images in our sequence.

```
# Set pixel colour range, from which extents will be distinguished
maxcol = 8
mincol = 1
lakes.setColourrange(maxcol, mincol)
```

The calcAutoAreas function will then be executed to detect water through all the time-lapse images in our sequence. The colour and verify flags can be toggled for defining the pixel colour range in each image and verifying each identified polygon manually, respectively.

```
# Calculate real areas
areas = lakes.calcAutoAreas(colour=False, verify=False)
```

Now we have our detected lakes, we can plot them in both the image plane (u,v) and real-world coordinates (x,y,z) to see how they look using the plotting functions in the Utilities module.

```
# Retrieve images and distortion parameters for plotting
imgset=lakes._imageSet
cameraMatrix=cameraenvironment.getCamMatrixCV2()
distortP=cameraenvironment.getDistortCoeffsCV2()
```

(continues on next page)

(continued from previous page)

```
# Retrieve DEM array for plotting
dem = cameraenvironment.getDEM()

# Retrieve uv and xyz coordinates of lakes
uvpts = [item[1][1] for item in areas]
xyzpts = [item[0][1] for item in areas]

# Show image extents and dems
for i in range(len(areas)):
    plotAreaPX(uvpts[i],
               imgset[i].getImageCorr(cameraMatrix, distortP),
               show=True, save=None)
    plotAreaXYZ(xyzpts[i], dem, show=True, save=None)
```

And finally, we can export our identified lakes as both text files and shapefiles using the writing functions in the FileHandler module (we suggest modifying the output file paths to your desired workspace).

```
# Get all image names for reference
imn = lakes.getImageNames()

# Get pixel and sq m lake areas
uvareas = [item[1][0] for item in areas]
xyzareas = [item[0][0] for item in areas]


# Write areas to text file
FileHandler.writeAreaFile(uvareas, xyzareas, imn, 'areas.csv')

# Write area coordinates to text file
FileHandler.writeAreaCoords(uvpts, xyzpts, imn,
                            'uvcoords.txt', 'xyzcoords.txt')

# Write lakes to shapefiles with WGS84 projection
proj = 32633
FileHandler.writeAreaSHP(xyzpts, imn, 'shpfiles', proj)
```

2.2 Manual detection of plume footprints

In this example, we will derive meltwater plume footprints from the front of Kronebreen, Svalbard, for a small subset of the 2014 melt season. This example can be found in KR_manualarea.py.

We will manually delineate meltwater plume footprints from corrected time-lapse images to derive surface areas at sea level. In this example, we will use images from Kronebreen camera 1 and the KR1 camera environment data.

First, we need to import the PyTrx packages that we are going to use.

```
from PyTrx.CamEnv import CamEnv
from PyTrx.Area import Area
from PyTrx.Velocity import Homography
import PyTrx.FileHandler as FileHandler
```

And then define the filepaths to our camera information (for creating our camera environment), our image mask (for identifying camera motion), and our time-lapse images.

```
# Define camera info filepath
camdata = '../Examples/camenv_data/camenvs/CameraEnvironmentData_KR1_2014.txt'

# Define image mask filepath
caminvmask = '../Examples/camenv_data/invmasks/KR1_2014_inv.jpg'

# Define folder path with time-lapse images
camimgs = '../Examples/images/KR1_2014_subset/*.JPG'
```

Next we need to create our camera environment using PyTrx's CamEnv object. As we do not know the camera pose (yaw, pitch, roll - YPR), we can estimate this using PyTrx's optimisation routines. The optimisation routine uses the difference between the u,v ground control points and the reprojected x,y,z ground control points to adjust and refine the camera model.

```
# Define camera environment
cameraenvironment = CamEnv(camdata)

# Optimise camera YPR
cameraenvironment.optimiseCamEnv('YPR')
```

To correct for motion in the camera platform, we will use PyTrx's Homography object (found in the Velocity module) to track static features and identify camera motion. From this motion, the Homography object creates a series of homography matrices (also known as a homography model) to co-register the images to one another.

```
# Set up Homography object
homog = Homography(camimgs, cameraenvironment,
                   caminvmask, calibFlag=True,
                   band='L', equal=True)

# Set homography parameters
hmethod='sparse'                #Method
hgmax=50000                     #Max number of seeding pts
hgqual=0.1                      #Seeding corner quality
hgmind=5.0                      #Min seeding pt distance
hgwinsize=(25,25)               #Tracking window size
hgback=1.0                      #Back-tracking threshold
hgminf=4                        #Min seeded pts to track

# Calculate homography
hg = homog.calcHomographies([hmethod, [hgmax, hgqual, hgmind], [hgwinsize, hgback,␣
↪hgminf]])

# Extract homography model
homogmatrix = [item[0] for item in hg]
```

Now we can initialise our Area object and manually delineate the plume footprints using the calcManualAreas function. This should bring up a pop-up window for each image, where you can click around each plume footprint and press 'enter' to move to the next.

```
# Set up Area object
plumes = Area(camimgs, cameraenvironment,
```

(continues on next page)

(continued from previous page)

```
                homogmatrix, calibFlag=True,
                imband='R', equal=True)

# Calculate real areas
areas = plumes.calcManualAreas()
```

We will save our manually-delineated plume footprints as area and coordinate text files using the export functions in the FileHandler module.

```
# Retrieve plume areas
uvareas = [item[1][0] for item in areas]
xyzareas = [item[0][0] for item in areas]

# Retrieve image names
imn=plumes.getImageNames()

# Write areas to text file
FileHandler.writeAreaFile(uvareas, xyzareas, imn, 'areas.csv')

# Retrieve coordinates of plume extents
xyzpts = [item[0][1] for item in areas]
uvpts = [item[1][1] for item in areas]

# Write coordinates to text file
FileHandler.writeAreaCoords(uvpts, xyzpts, imn,
                            'uvcoords.txt',
                            'xyzcoords.txt')
```

And we will also export the plume footprints as shapefiles, using the same projection as our inputted DEM. These shapefiles can be used in subsequent analysis and imported into GIS software for viewing.

```
# Define projection
proj = 32633

# Write to shapefile
FileHandler.writeAreaSHP(xyzpts, imn, 'shpfiles', proj)
```

And finally, we can plot the plume footprints onto the time-lapse images for viewing purposes. Here is an example to plot the footprints onto RGB versions of the images, using a workflow using opencv and matplotlib.

```
# Import packages
import glob,cv2
import matplotlib.image as mpimg
import matplotlib.pyplot as plt

# Get original images in directory
ims = sorted(glob.glob(camimgs))

# Get camera correction variables
cameraMatrix=cameraenvironment.getCamMatrixCV2()
distortP=cameraenvironment.getDistortCoeffsCV2()
newMat, roi = cv2.getOptimalNewCameraMatrix(cameraMatrix, distortP,
                                            (5184,3456),1,(5184,3456))
```

(continues on next page)

(continued from previous page)

```
# Get corresponding xy pixel areas and images
count=1
for p,i in zip(uvpts,ims):
    x=[]
    y=[]
    for ps in p[0]:
        x.append(ps[0])
        y.append(ps[1])

    # Read image and undistort
    im1=mpimg.imread(i)
    im1 = cv2.undistort(im1, cameraMatrix, distortP,
                        newCameraMatrix=newMat)

 # Plot image
 plt.figure(figsize=(20,10))
 plt.imshow(im1)
 plt.axis([0,5184,3456,0])
 plt.xticks([])
 plt.yticks([])

 # Plot pixel area
 plt.plot(x,y,'#fff544',linewidth=2)

 # Save image to file
 plt.savefig('plumeplotted' + str(count) + '.JPG', dpi=300)
 plt.show()
 count=count+1
```

2.3 Manual detection of glacier terminus profiles

Here, we will delineate glacier terminus profiles (as line features) from a small subset of time-lapse images from Tunabreen, Svalbard, during the 2014 melt season. This example can be found in TU_manualline.py.

We will manually delineate terminus profiles from corrected time-lapse images to derive a sequence of positions representing glacier retreat. In this example, we will use images from Tunabreen camera 1 and the associated camera environment data.

First, we need to import the PyTrx packages that we are going to use.

```
from PyTrx.CamEnv import CamEnv
from PyTrx.Line import Line
from PyTrx.Velocity import Homography
import PyTrx.FileHandler as FileHandler
from PyTrx.Utilities import plotLinePx, plotLineXYZ
```

And define the paths to our camera information, image mask (for tracking static points and correcting for camera platform motion), and time-lapse images.

```
# Define data input directories
camdata = '../Examples/camenv_data/camenvs/CameraEnvironmentData_TU1_2015.txt'
invmask = '../Examples/camenv_data/invmasks/TU1_2015_inv.jpg'
camimgs = '../Examples/images/TU1_2015_subset/*.JPG'
```

Firstly, we can initialise a CamEnv object which represents our camera environment, using our camera information .txt file.

```
# Create camera environment
cam = CamEnv(camdata)
```

In this example, the camera pose (yaw, pitch, roll - YPR) is unknown as it is difficult to measure this in the field. We can determine the YPR using PyTrx's optimisation routine.

```
# Define what parameters to optimise
optflag = 'YPR'

# Define optimisation method
optmethod = 'trf'

# Optimise camera environment
cam.optimiseCamEnv(optflag, optmethod, show=False)
```

To account for motion in the camera platform, we will track static features in the image (in the areas defined by our image mask) using PyTrx's Homography object. Here, we track selected corner features in the image to derive a homography matrix for each image pair.

```
# Set homography parameters
hmethod='sparse'                #Seeding method
hgwinsize=(25,25)               #Tracking window size
hgback=1.0                      #Back-tracking threshold
hgmax=50000                     #Max num of pts to seed
hgqual=0.1                      #Corner quality for seeding
hgmind=5.0                      #Min distance between seeded pts
hgminf=4                        #Min num seeded pts to track

# Set up Homography object
homog = Homography(camimgs, cam, invmask, calibFlag=True, band='L',
                   equal=True)

# Calculate homography
hg = homog.calcHomographies([hmethod, [hgmax, hgqual, hgmind],
                             [hgwinsize, hgback, hgminf]])

# Extract homography matrices
homogmatrix = [item[0] for item in hg]
```

Now we can manually delineate our terminus profiles from each time-lapse image using the Line object in PyTrx. First, we initialise the object, and then use the calcManualLines() function to start the manual delineations. For each image, an interactive window will open, where you can click points to trace the terminus, and press 'enter' when you are finished to prompt the next image to load.

```
# Set up line object
terminus = Line(camimgs, cam, homogmatrix)
```

(continues on next page)

(continued from previous page)

```
# Manually define terminus lines
lines = terminus.calcManualLines()
```

PyTrx's FileHandler module can be used to export all findings to file. Here, we will write out two files containing line lengths and coordinates, shapefiles for each line geometry, and information about the homography to file.

```
# Get image names
imn=terminus.getImageNames()

# Get uv and xyz lines
pxlines = [item[1][0] for item in lines]
xyzlines = [item[0][0] for item in lines]

# Write line data to .csv file
FileHandler.writeLineFile(pxlines, xyzlines, imn, 'lines.csv')

# Write line coordinates to txt file
FileHandler.writeLineCoords(pxcoords, xyzcoords, imn,
                        'uvcoord.txt', 'xyzcoords.txt')

# Get uv and xyz line coordinates
pxcoords = [item[1][1] for item in lines]
xyzcoords = [item[0][1] for item in lines]


# Write shapefiles from line data
projection=32633
FileHandler.writeLineSHP(xyzcoords, imn, 'shapefiles', projection)

# Write homography data to .csv file
FileHandler.writeHomogFile(hg, imn, 'homography.csv')
```

Lastly, we can view our delineated terminus profiles in both the image and the DEM space using the plotting function in PyTrx's FileHandler module.

```
# Get dem array
dem = cam.getDEM()

# Get image sequence as arrays
imgset=terminus._imageSet

# Retrieve image correction coefficients
cameraMatrix=cam.getCamMatrixCV2()
distortP=cam.getDistortCoeffsCV2()

# Plot uv lines on image
for i in range(len(pxcoords)):

   # Plot lines in image plane and as XYZ lines
    plotLinePX(pxcoords[i],
```

(continues on next page)

(continued from previous page)

```
                imgset[i].getImageCorr(cameraMatrix, distortP),
                show=True,
                save='uv_'+str(imn[i]))
    # Plot xyz lines on DEM
    plotLineXYZ(xyzcoords[i],
                dem,
                show=True,
                save='xyz_'+str(imn[i]))
```

2.4 Georectification of glacier calving event point locations

Here, we will georectify some pre-defined points that denote the locations of glacier calving events at Tunabreen, Svalbard, captured from high-frequency time-lapse images. One point represents a calving event identified in the image plane, which will be imported and georectified to x,y,z coordinates using the georectification functions in PyTrx. The x,y,z coordinates will then be plotting onto the DEM, and exported to shapefile.

This example can be found in TU_ptsgeorectify.py, using the Tunabreen camera 1 environment data file.

First, we need to import the PyTrx functions that we are going to use along with some other packages (for GIS, data manipulation and plotting), and define the file paths to our camera environment information and point data.

```
# Import PyTrx CamEnv functions
from PyTrx.CamEnv import CamEnv, setProjection, projectUV

# Import other packages to use
import matplotlib.pyplot as plt
import osgeo.ogr as ogr
import osgeo.osr as osr
import numpy as np

# Define camera environment file path
tu1camenv='../Examples/camenv_data/camenvs/CameraEnvironmentData_TU1_2015.txt'

# Define calving pt data file path
tu1calving = '../Examples/results/ptsgeorectify/TU1_calving_xy.csv'
```

Next, we will load our point data (i.e. calving event locations)

```
# Open file
f=open(tu1calving,'r')

# Read header line
header=f.readline()

# Create empty variables to populate
time=[]
region=[]
style=[]
tu1_xy=[]

# Read each line from file
```

(continues on next page)

(continued from previous page)

```
for line in f.readlines():

   # Split line into variables
   temp=line.split(',')

   # Extract variables
   time.append(float(temp[0].rstrip()))
   region.append(temp[1].rstrip())
   style.append(temp[2].rstrip())
   tu1_xy.append([float(temp[3].rstrip()), float(temp[4].rstrip())])

print(f'{len(tu1_xy)} locations for calving events detected')

# Change pt coordinate list to array
tu1_xy = np.array(tu1_xy)
```

Next, we will create a CamEnv object to hold all the information about our camera. We will initialise the object with our camera environment file, which includes paths to the camera calibration, ground control point positions, reference image and DEM, along with the position of our camera and its pose represented along three axes (yaw, pitch, roll - YPR).

```
# Define camera environment
tu1cam = CamEnv(tu1camenv)
```

Now we have our camera environment, we need to model how the three-dimensional world (represented by the DEM) is translated to the two-dimensional image plane (represented by our reference image). We will use the setProjection function in PyTrx's CamEnv module in order to do this.

```
# Get DEM from camera environment
demobj = tu1cam.getDEM()

# Get inverse projection variables through camera info
invprojvars = setProjection(demobj, tu1cam._camloc, tu1cam._camDirection,
                            tu1cam._radCorr, tu1cam._tanCorr, tu1cam._focLen,
                            tu1cam._camCen, tu1cam._refImage)
```

With our inverse projection model, we can translate the calving event locations defined in the image plane to x,y,z coordinates with the project UV function.

```
# Inverse project uv coodinates to xyz coordinates
tu1_xyz = projectUV(tu1_xy, invprojvars)
```

To view our reprojected x,y,z points, we can plot them using the plotting functionality in matplotlib. We will plot the points over our DEM.

```
# Retrieve DEM extent and elevation array
demextent = demobj.getExtent()
dem = demobj.getZ()

# Get camera position (xyz) for plotting
post = tu1cam._camloc

# Plot DEM and camera location
```

(continues on next page)

(continued from previous page)

```
fig,(ax1) = plt.subplots(1, figsize=(15,15))
fig.canvas.set_window_title('TU1 calving event locations')
ax1.locator_params(axis = 'x', nbins=8)
ax1.tick_params(axis='both', which='major', labelsize=0)
ax1.imshow(dem, origin='lower', extent=demextent, cmap='gray')
ax1.axis([demextent[0], demextent[1], demextent[2], demextent[3]])
cloc = ax1.scatter(post[0], post[1], c='g', s=10, label='Camera location')

# Plot calving locations on DEM
xr = [pt[0] for pt in tu1_xyz]
yr = [pt[1] for pt in tu1_xyz]
ax1.scatter(xr, yr, c='r',s=10)

# Save and show plot
plt.savefig('TU1_calving_xyz.JPG', dpi=300)
plt.show()
```

And finally we will export the inverse projected x,y,z point coordinates to a shapefile using the osgeo modules ogr and osr.

```
# Get ESRI shapefile driver
driver = ogr.GetDriverByName('ESRI Shapefile' )

# Create data source
shp = 'tu1_calving.shp'
ds = driver.CreateDataSource(shp)
if ds is None:
    print(f'Could not create file {shp}')

# Set WGS84 projection
proj = osr.SpatialReference()
proj.ImportFromEPSG(32633)

# Create layer in data source
layer = ds.CreateLayer('tu1_calving', proj, ogr.wkbPoint)


# Add ID and time attributes to layer
layer.CreateField(ogr.FieldDefn('id', ogr.OFTInteger))
layer.CreateField(ogr.FieldDefn('time', ogr.OFTReal))

# Add terminus region attribute
field_region = ogr.FieldDefn('region', ogr.OFTString)
field_region.SetWidth(8)
layer.CreateField(field_region)

# Add calving style attribute
field_style = ogr.FieldDefn('style', ogr.OFTString)
field_style.SetWidth(10)
layer.CreateField(field_style)
```

(continues on next page)

(continued from previous page)

```
# Create point features with data attributes in layer
for a,b,c,d in zip(tu1_xyz, time, region, style):
    count=1

    # Create feature
    feature = ogr.Feature(layer.GetLayerDefn())

    # Write feature attributes
    feature.SetField('id', count)
    feature.SetField('time', b)
    feature.SetField('region', c)
    feature.SetField('style', d)

    # Create feature geometry
    wkt = "POINT(%f %f)" %  (float(a[0]) , float(a[1]))
    point = ogr.CreateGeometryFromWkt(wkt)
    feature.SetGeometry(point)

    # Compile feature
    layer.CreateFeature(feature)

    # Close feature
    feature.Destroy()
    count=count+1

# Close layer
ds.Destroy()
```

2.5 Sparse feature-tracking to derive glacier flow

In this example, we will calculate glacier flow velocities from Kronebreen, Svalbard, using PyTrx's sparse feature-tracking method. The sparse feature-tracking method using corner feature detection to identify coherent features on the glacier surface, and then tracks them between image pairs using Optical Flow.

We will derive glacier velocities from a subset of time-lapse images from the 2014 melt season, which can be found in the PyTrx GitHub repository, using the Kronebreen camera 2 environment data file. This example can be found in KR_velocity1.py.

Let's firstly import the PyTrx modules we need.

```
from PyTrx.CamEnv import CamEnv
from PyTrx.Velocity import Velocity, Homography
from PyTrx.FileHandler import writeHomogFile, writeVeloFile, \
     writeVeloSHP, writeCalibFile
from PyTrx.Utilities import plotVeloPX, plotVeloXYZ, \
     interpolateHelper, plotInterpolate
```

And then define the file paths to our camera information, our time-lapse images, and the masks we will use to identify the regions of the image we want to use for deriving glacier flow velocities and tracking static features.

```
# Camera environment file path
camdata = '../Examples/camenv_data/camenvs/CameraEnvironmentData_KR2_2014.txt'

# Mask for velocity feature-tracking
camvmask = '../Examples/camenv_data/masks/KR2_2014_vmask.jpg'

# Inverse mask for image registration
caminvmask = '../Examples/camenv_data/invmasks/KR2_2014_inv.jpg'

# Time-lapse images
camimgs = '../Examples/images/KR2_2014_subset/*.JPG'
```

We will construct a CamEnv object using our camera environment file, which will hold all information about the translation of our images to x,y,z space (represented by our DEM). We will optimise our camera environment, using our pre-defined ground control points to refine the model and estimate the camera pose (i.e. yaw, pitch, roll - YPR)

```
# Define camera environment
cameraenvironment = CamEnv(camdata)

# Optimise camera environment to refine camera pose
cameraenvironment.optimiseCamEnv('YPR')
```

We can check our camera environment parameters using a reporter and various plotting functions.

```
# Report camera environment parameters
cameraenvironment.reportCamData()

# Show ground control points
cameraenvironment.showGCPs()

# Show image principal point
cameraenvironment.showPrincipalPoint()

# Show ground control point residuals
cameraenvironment.showResiduals()
```

Next we will calculate the homography model using PyTrx's Homography object. This represents correction for motion in the camera platform which, if uncorrected, can introduce false motion into our velocity measurements. We can account for this using our homography model to co-register our time-lapse images.

```
# Set homography parameters
hmethod='sparse'                #Method
hgwinsize=(25,25)               #Tracking window size
hgback=1.0                      #Back-tracking threshold
hgmax=50000                     #Maximum number of points to seed
hgqual=0.1                      #Corner quality for seeding
hgmind=5.0                      #Minimum distance between seeded points
hgminf=4                        #Minimum number of seeded points to track

# Set up Homography object
homog = Homography(camimgs, cameraenvironment, caminvmask, calibFlag=True,
                   band='L', equal=True)
```

(continues on next page)

(continued from previous page)

```
# Calculate homography
hgout = homog.calcHomographies([hmethod, [hgmax, hgqual, hgmind], [hgwinsize,
                                hgback, hgminf]])
```

Now we can look at measuring the flow of the glacier using the feature-tracking functionality in PyTrx's Velocity object. There are a number of parameters we can set to adjust our tracking conditions

```
# Set image conditions
calibration = True                # Correct images for distortion?
iband = 'L'                      # Image band to track with (R/G/B/L)
eq = True                         # Images with histogram equalisation?

# Set up Velocity object
velo=Velocity(camimgs, cameraenvironment, hgout, camvmask, calibFlag=True,
              band='L', equal=True)

# Set velocity parameters
vmethod = 'sparse'               # Method
vwinsize = (25,25)               # Tracking window size
bk = 1.0                         # Back-tracking threshold
mpt = 50000                      # Maximum number of points to seed
ql = 0.1                         # Corner quality for seeding
mdis = 5.0                       # Minimum distance between seeded points
mfeat = 4                        # Minimum number of seeded points to track

# Calculate glacier flow velocity
velocities = velo.calcVelocities([vmethod, [mpt, ql, mdis], [vwinsize, bk,
                                  mfeat]])
```

To export our results, we can write out our intrinsic camera matrix (which can be useful when you have optimised the intrinsic camera parameters of the camera environment) and calculated homography using the exporting functions in PyTrx's FileHandler module.

```
# Write out camera calibration info to .txt file
matrix, tancorr, radcorr = cameraenvironment.getCalibdata()
writeCalibFile(matrix, tancorr, radcorr, 'KR2_2014_1.txt')

# Write homography data to .csv file
imn = velo.getImageNames()
writeHomogFile(hgout, imn, 'homography.csv')
```

And then we can export our calculated velocities to .csv file and .shp shapefiles for plotting and further analysis

```
# Fetch uv and xyz velocities
xyzvel=[item[0][0] for item in velocities]
uvvel=[item[1][0] for item in velocities]

# Write out velocity data to .csv file
writeVeloFile(xyzvel, uvvel, hgout, imn, 'velo_output.csv')

# Fetch xyz pt coordinates and tracking errors
xyz0=[item[0][1] for item in velocities]
xyzerr=[item[0][3] for item in velocities]
```

(continues on next page)

(continued from previous page)

```
# Write points to shp file with EPSG:32633 projection
proj = 32633
writeVeloSHP(xyzvel, xyzerr, xyz0, imn, 'shpfiles', proj)
```

If we want to view the results, we can retrieve all of our tracked points (in both the images and x,y,z coordinates) and plot them over the top of our images and DEM.

```
# Get calibration coefficients for plotting corrected images
cameraMatrix=cameraenvironment.getCamMatrixCV2()
distortP=cameraenvironment.getDistortCoeffsCV2()

# Get images for overlaying uv pts
imgset=velo._imageSet

# Get DEM array for overlaying xyz pts
dem=cameraenvironment.getDEM()

# Get uv seeded and tracked point positions
uv0=[item[1][1] for item in velocities]
uv1corr=[item[1][3] for item in velocities]

# Get xyz seeded and tracked point positions
xyz0=[item[0][1] for item in velocities]
xyz1=[item[0][2] for item in velocities]

# Cycle through data from image pairs
for i in range(len(imn)-1):

    # Get image name and print
    print('\nVisualising data for ' + str(imn[i]))

    # Plot uv velocity points on image plane
    print('Plotting image plane output')
    plotVeloPX(uvvel[i], uv0[i], uv1corr[i],
               imgset[i].getImageCorr(cameraMatrix, distortP),
               show=True, save='uv_'+imn[i])


    # Plot xyz velocity points on dem
    print('Plotting XYZ output')
    plotVeloXYZ(xyzvel[i], xyz0[i], xyz1[i],
                dem, show=True, save='xyz_'+imn[i])


    # Plot interpolation map with linear interpolation
    print('Plotting interpolation map')
    grid, pointsextent = interpolateHelper(xyzvel[i], xyz0[i], xyz1[i], 'linear')
    plotInterpolate(grid, pointsextent, dem, show=True,
                    save='interp_'+imn[i])
```

Additionally, we can export our velocities as gridded ASCII files. These files are recognised by many mapping software, such as ArcGIS and QGIS, and can be imported to create raster surfaces.

```
# import numpy for grid operations
import numpy as np

# Cycle through velocity data from image pairs
for i in range(velo.getLength()-1):

    # Change all the nans to -999.999 and flip the y axis
    grid[np.isnan(grid)] = -999.999
    grid = np.flipud(grid)

    # Open new file with write permissions
    imn=velo._imageSet[i].getImageName()
    afile = open(imn + '_interpmap.txt','w')

 # Make a list for each raster header variable, with the label and value
 col = ['ncols', str(grid.shape[1])]
 row = ['nrows', str(grid.shape[0])]
 x = ['xllcorner', str(pointsextent[0])]
 y = ['yllcorner', str(pointsextent[2])]
 cell = ['cellsize', str(10.)]
 nd = ['NODATA_value', str(-999.999)]

 # Write each header line on a new line of the file
 header = [col,row,x,y,cell,nd]
 for i in header:
     afile.write(' '.join(i) + '\n')

 # Iterate through each row and column value
 for i in range(grid.shape[0]):
     for j in range(grid.shape[1]):

         # Write each data value to the row, separated by spaces
         afile.write(str(grid[i,j]) + ' ')

     # New line at end of row
     afile.write('\n')

 # Close file
 afile.close()
```

2.6 Dense feature-tracking to derive glacier flow

PyTrx's dense feature-tracking utilises traditional cross-correlation template matching to track a regular grid of points between an image pair. In this example, we calculate glacier flow velocities from Kronebreen, Svalbard, using a similar workflow to the sparse feature-tracking workflow shown previously.

This workflow can be found in KR_velocity2.py, and a merged workflow using both sparse and dense feature-tracking can be found in KR_velocity2.py. The main difference in the merged workflow is that velocities are processed using the stand-alone functions provided in PyTrx, rather than handled by PyTrx's class objects. This provides the user with a script that is more flexible and adaptable.

The main difference in the dense feature-tracking workflow (compared to the sparse workflow) is in the input variables

to the Velocity object's calcVelocities function. When the tracking method is set to 'dense' then the following variables can be defined – grid spacing, template and search window size, template matching method, threshold correlation, and minimum number of tracked points.

```
# Set up Velocity object
velo=Velocity(camimgs, cameraenvironment, hgout, camvmask, calibFlag=True,
              band='L', equal=True)

# Set velocity tracking parameters
vmethod = 'dense'                    # Method
vgrid = [50,50]                      # Dense matching grid distance
vtemplate = 10                       # Template size
vsearch = 50                         # Search window size
vmethod = 'cv2.TM_CCORR_NORMED'      # Method for template matching
vthres = 0.8                         # Threshold average template correlation
vminf = 5                            # Minimum number of tracked points

# Calculate dense velocities
velocities = velo.calcVelocities([vmethod, vgrid, [vmethod, vtemplate, vsearch,
                                  vthres, vminf]])
```

CHAPTER

THREE

PACKAGE GUIDE

Detailed documentation is included in the scripts that make up PyTrx. Each script contains classes and functions for handling each aspect needed for photogrammetric processing.

For beginners in programming, it is advised to look at the example applications provided in the GitHub repository and adapt them accordingly for your own use. For experienced programmers, get stuck in. Feel free to contact us if you run into major problems or have constructive comments that will help us further PyTrx and its capabilities. We will not respond to minor troubleshooting or unconstructive comments.

3.1 CamEnv

Handles the associated data with the camera environment. The GCPs class handles the Ground Control Points (GCPs) and their correspondence to the associated DEM and CamImage object. The CamCalib class handles information concerning the camera calibration, i.e. the intrinsic camera matrix and lens distortion coefficients. This class contains functionality for reading in calibration files from .txt and .mat formats.

The CamEnv compiles all the information about the camera environment from the GCPs and CamCalib classes, and also contains information about the camera object (pose and location). This is also where georectification functionality is held, with functions for projection and inverse projection. The class is initialised using a .txt file containing file path directories to all the associated data files.

3.2 DEM

Handles the DEM data. This currently supports .mat and .tif file types. The ExplicitRaster class represents a DEM as a numeric raster with explicit XY cell referencing in each grid cell. The class includes functions for densification, calculating viewsheds, and incorporates unbound functions that import a DEM file from .mat and .tif formats.

3.3 FileHandler

This module contains a set of functions for reading in data from files (such as image data and calibration information) and writing out data.

3.4 Images

Handles the image data, and the image sequence. The CamImage class holds information about a singular image and contains functionality for importing image data from file and passing specific image bands forward for subsequent processing. The ImageSequence class holds information about an image sequence, i.e. a collection of CamImage objects, from which specific images and image pairs can be called.

3.5 Velocity

Calculates velocities and homography. This can either be achieved through the Velocity class for processing velocities and homography through a series of images, or using the functions provided within the script for processing velocities and homography between an image pair.

3.6 Area

Automated and manual detection of surface areas from imagery (e.g. supraglacial lakes, meltwater plume surface extent). This can either be achieved through the Area class for defining areas of interest through a series of images, or using the functions provided within the script for defining areas of interest in a single image.

3.7 Line

Manual detection of line features from imagery (e.g. glacier terminus position). This can either be achieved through the Line class for defining line features through a series of images, or using the functions provided within the script for defining line features in a single image.

3.8 Utilities

This module contains a set of functions for plotting and interpolating data.

CHAPTER

FOUR

MODULES

4.1 Area module

4.2 CamEnv module

4.3 DEM module

The DEM module contains functionality for handling DEM data and implementing this data into the PyTrx.CamEnv.CamEnv object class.

DEM.**DEM_FromMat**(*matfile*)

Function for loading a DEM array from a Matlab (.mat) file containing separate X, Y, Z matrices

Parameters
matfile (*str*) – DEM .mat filepath

Returns
A DEM object

Return type
PyTrx.DEM.ExplicitRaster

DEM.**DEM_FromTiff**(*tiffFile*)

Function for loading a DEM array from a .tiff file containing raster-formatted data. The tiff data importing is handled by GDAL

Parameters
tiffFile (*str*) – DEM .tif filepath

Returns
A DEM object

Return type
PyTrx.DEM.ExplicitRaster

class DEM.**ExplicitRaster**(*X*, *Y*, *Z*, *nodata=nan*)

Bases: object

A class to represent a numeric Raster with explicit XY cell referencing in each grid cell

Variables

- **_data** (*arr*) – DEM data array
- **_nodata** (*float*) – Nodata value

- **_extents** (*list*) – DEM extent [X0, X1, Y0, Y1]

densify(*densefac=2*)

Function to densify the DEM array by a given densification factor. The array is multiplied by the given densification factor and then subsequently values are interpolated using the SciPy function RectBivariate-Spline. The densification factor is set to 2 by default, meaning that the size of the DEM array is doubled

Parameters
densefac (*int*) – Densification factor

Returns
Densified DEM

Return type
PyTrx.DEM.ExplicitRaster

getCols()

Return the number of columns in the DEM data array

getData(*dim=None*)

Return DEM data. XYZ dimensions can be individually called with the dim input variable (integer: 0, 1, or 2)

Parameters
dim (*int*) – Dimension to retrieve (0, 1, or 2), default to None

Returns
DEM dimension as array

Return type
arr

getExtent()

Return DEM extent

getNoData()

Return fill value for no data in DEM array

getRows()

Return the number of rows in the DEM data array

getShape()

Return the shape of the DEM data array

getZ()

Return height (Z) data of DEM

getZcoord(*x*, *y*)

Return height (Z) at a given XY coordinate in DEM

Parameters

- **x** (*int*) – X coordinate
- **y** (*int*) – Y coordinate

Returns
DEM Z value for given coordinate

Return type
int

reportDEM()

Self reporter for DEM class object. Returns the number of rows and columns in the array, how NaN values in the array are filled, and the data extent coordinates

subset(*cmin*, *cmax*, *rmin*, *rmax*)

Return a specified subset of the DEM array

Parameters

- **cmin** (*int*) – Column minimum extent
- **cmax** (*int*) – Column maximum extent
- **rmin** (*int*) – Row minimum extent
- **rmax** (*int*) – Row maximum extent

Returns
Subset of DEM

Return type
PyTrx.DEM.ExplicitRaster

class DEM.**TestDEM**(*methodName='runTest'*)

Bases: TestCase

test_ExplicitRaster()

test_getRows()

test_subset()

test_voxelviewshed()

DEM.**load_DEM**(*demfile*)

Function for loading DEM data from different file types, which is automatically detected. Recognised file types: .mat and .tif

Parameters
demfile (*str*) – DEM filepath

Returns
DEM object

Return type
PyTrx.DEM.ExplicitRaster

DEM.**voxelviewshed**(*dem*, *viewpoint*)

Calculate a viewshed over a DEM from a given viewpoint in the DEM scene. This function is based on the viewshed function (voxelviewshed.m) available in ImGRAFT. The ImGRAFT voxelviewshed.m script is available at: http://github.com/grinsted/ImGRAFT/blob/master/voxelviewshed.m

Parameters

- **dem** (*PyTrx.DEM.ExplicitRaster*) – DEM object
- **viewpoint** (*list*) – 3-element vector specifying the viewpoint

Returns
vis – Boolean visibility matrix (which is the same size as dem)

Return type
arr

4.4 FileHandler module

Th FileHandler module contains all the functions called by a PyTrx object to load and export data.

FileHandler.**checkMatrix**(*matrix*)

Function to support the calibrate function. Checks and converts the intrinsic matrix to the correct format for calibration with OpenCV

Parameters
matrix (*arr*) – Inputted matrix for checking

Returns
mat – Validated matrix

Return type
arr

FileHandler.**importAreaData**(*xyzfile*, *pxfile*)

Import xyz and px data from text files

Parameters

- **xyzfile** (*str*) – File directory to xyz coordinates
- **pxfile** (*str*) – File directory to uv coordinates

Returns
areas – Coordinates and areas of detected areas

Return type
list

FileHandler.**importAreaFile**(*fname*, *dimension*)

Import pixel polygon data from text file and compute pixel extents

Parameters

- **fname** (*str*) – Path to the text file containing the UV coordinate data
- **dimension** (*int*) – Integer denoting the number of dimensions in coordinates

Returns
areas – UV coordinates for polygons and pixel areas for polygons

Return type
list

FileHandler.**importLineData**(*xyzfile*, *pxfile*)

Import xyz and px data from text files

Parameters

- **xyzfile** (*str*) – File directory to xyz coordinates
- **pxfile** (*str*) – File directory to uv coordinates

Returns
lines – Coordinates and lengths of detected lines

Return type
list

FileHandler.**importLineFile**(*fname*, *dimension*)

Import XYZ line data from text file and compute line lengths

Parameters

- **fname** (*str*) – Path to the text file containing the XYZ coordinate data
- **dimension** (*int*) – Number of dimensions in point coordinates i.e. 2 or 3

Returns
lines – List containing line coordinates and lengths

Return type
list

FileHandler.**lineSearch**(*lineList*, *search*)

Function to supplement the readCalib function. Given an input parameter to search within the file, this will return the line numbers of the data

Parameters

- **lineList** (*list*) – List of strings within a file line
- **search** (*str*) – Target keyword to search for

Returns
datalines – Line numbers with keyword match

Return type
list

FileHandler.**readCalib**(*fileName*, *paramList*)

Function to find camera calibrations from a file given a list or Matlab file containing the required parameters. Returns the parameters as a dictionary object. Compatible file structures: 1) .txt file ("RadialDistortion [k1,k2,k3…k8], TangentialDistortion [p1,p2], IntrinsicMatrix [fx 0. 0.][s fy 0.][cx cy 1] End"); 2) .mat file (Camera calibration file output from the Matlab Camera Calibration App (available in the Computer Vision Systems toolbox)

Parameters

- **fileName** (*str*) – File directory for calibration file
- **paramList** (*list*) – List of strings denoting keywords to look for in calibration file

Returns
calib – Calibration parameters denoted by keywords

Return type
list

FileHandler.**readGCPs**(*fileName*)

Function to read ground control points from a .txt file. The data in the file is referenced to under a header line. Data is appended by skipping the header line and finding the world and image coordinates from each line

Parameters
fileName (*str*) – File path directory for GCP file

Returns

- **world** (*arr*) – GCPs in xyz coordinates
- **image** (*arr*) – GCPs in image coordinates

FileHandler.**readImg**(*path*, *band='L'*, *equal=True*)

Function to prepare an image by opening, equalising, converting to either grayscale or a specified band, then returning a copy

Parameters

- **path** (*str*) – Image file path directory
- **band** (*str*) – Desired band output - 'R': red band; 'B': blue band; 'G': green band; 'L': grayscale (default='L')
- **equal** (*bool*) – Flag to denote if histogram equalisation should be applied (default=True)

Returns
bw – Image array

Return type
arr

FileHandler.**readMask**(*img*, *writeMask=None*)

Function to create a mask for point seeding using PIL to rasterize polygon. The mask is manually defined by the user using the pyplot ginput function. This subsequently returns the manually defined area as a .jpg mask. The writeMask file path is used to either open the existing mask at that path or to write the generated mask to this path

Parameters

- **img** (*arr*) – Image to define mask in
- **writeMask** (*str, optional*) – File destination that mask output is written to, default to None

Returns
myMask – Array defining the image mask

Return type
arr

FileHandler.**readMatrixDistortion**(*path*)

Function to support the calibrate function. Returns the intrinsic matrix and distortion parameters required for calibration from a given file

Parameters
path (*str*) – Directory of calibration file

Returns

- **intrMat** (*arr*) – Intrinsic matrix as a 3x3 array, including focal length, principal point, and skew
- **tanDis** (*arr*) – Tangential distortion values (p1, p2)
- **radDis** (*arr*) – Radial distortion values (k1, k2… k6)

FileHandler.**returnData**(*lines*, *data*)

Function to supplement the importCalibration function. Given the line numbers of the parameter data (the ouput of the lineSearch function), this will return the data

Parameters

- **lines** (*list*) – Given line numbers to extract data from
- **data** (*list*) – Raw line data

Returns
D – Extracted data

Return type
arr

FileHandler.**writeAreaCoords**(*pxpts*, *xyzpts*, *imn*, *pxdestination*, *xyzdestination*)

Write UV and XYZ area coordinates to text files. These file types are compatible with the importing tools (importAreaPX, importAreaXYZ)

Parameters

- **xyzarea** (*list*) – XYZ areas
- **xyzpts** (*list*) – XYZ coordinates
- **imn** (*list*) – Image names
- **pxdestination** (*str*) – File directory where UV coordinates will be written to
- **xyzdestination** (*str*) – File directory where XYZ coordinates will be written to

FileHandler.**writeAreaFile**(*pxareas*, *xyzareas*, *imn*, *destination*)

Write UV and XYZ areas to csv file

Parameters

- **pxarea** (*list*) – Pixel extents
- **xyzarea** (*list*) – XYZ areas
- **imn** (*list*) – Image names
- **destination** (*str*) – File directory where csv file will be written to

FileHandler.**writeAreaSHP**(*xyzpts*, *imn*, *fileDirectory*, *projection=None*)

Write OGR real polygon areas (from ALL images) to file in a .shp file type that is compatible with ESRI mapping software

Parameters

- **xyzpts** (*list*) – XYZ coordinates for polygons
- **imn** (*list*) – Image name
- **fileDirectory** (*str*) – Destination that shapefiles will be written to
- **projection** (*int/str, optional*) – Coordinate projection that the shapefile will exist in. This can either be an ESPG number (expressed as an integer) or a well-known geographical coordinate system (expressed as a string). Well-known geographical coordinate systems are: 'WGS84', 'WGS72', NAD83' or 'EPSG:n'

FileHandler.**writeCalibFile**(*intrMat*, *tanDis*, *radDis*, *fname*)

Write camera calibration data to .txt file, including camera matrix, and radial and tangential distortion parameters

Parameters

- **intrMat** (*arr*) – Intrinsic camera matrix
- **tanDis** (*arr*) – Tangential distortion parameters
- **radDis** (*arr*) – Radial distortion parameters
- **fname** (*str*) – Directory to write file to

FileHandler.**writeHomogFile**(*homog*, *imn*, *fname*)

Function to write all homography data from a given timeLapse sequence to .csv file. Data is formatted as sequential columns containing the following information: Image pair 1 name, Image pair 2 name, Homography matrix (i.e. all values in the 3x3 matrix), Number of features tracked, X mean displacement, Y mean displacement, X standard deviation, Y standard deviation, Mean error magnitude, Mean homographic displacement, and Signal-to-noise ratio

Parameters

- **homog** (*list*) – Homography [mtx, im0pts, im1pts, ptserr, homogerr]
- **imn** (*list*) – List of image names
- **fname** (*str*) – Directory for file to be written to

FileHandler.**writeLineCoords**(*uvpts*, *xyzpts*, *imn*, *pxdestination*, *xyzdestination*)

Write UV and XYZ line coordinates to text file. These file types are compatible with the importing tools (importLinePX, importLineXYZ)

Parameters

- **uvpts** (*list*) – Pixel coordinates
- **xyzpts** (*list*) – XYZ coordinates
- **imn** (*list*) – Image names
- **pxdestination** (*str*) – File directory where UV coordinates will be written to
- **xyzdestination** (*str*) – File directory where XYZ coordinates will be written to

FileHandler.**writeLineFile**(*pxline*, *xyzline*, *imn*, *destination*)

Write UV and XYZ line lengths to csv file

Parameters

- **pxline** (*list*) – Pixel line lengths
- **xyzline** (*list*) – XYZ line lengths
- **imn** (*list*) – Image names
- **destination** (*str*) – File directory where output will be written to

FileHandler.**writeLineSHP**(*xyzpts*, *imn*, *fileDirectory*, *projection=None*)

Write OGR real line features (from ALL images) to file in a .shp file type that is compatible with ESRI mapping software

Parameters

- **xyzpts** (*list*) – XYZ coordinates for polygons
- **imn** (*list*) – Image name
- **fileDirectory** (*str*) – Destination that shapefiles will be written to
- **projection** (*int/str, optional*) – Coordinate projection that the shapefile will exist in. This can either be an ESPG number (expressed as an integer) or a well-known geographical coordinate system (expressed as a string). Well-known geographical coordinate systems are: 'WGS84', 'WGS72', NAD83' or 'EPSG:n'

FileHandler.**writeVeloFile**(*xyzvel*, *uvvel*, *homog*, *imn*, *fname*)

Function to write all velocity data from a given timeLapse sequence to .csv file. Data is formatted as sequential columns containing the following information: Image pair 1 name, Image pair 2 name, Average xyz velocity, Number of features tracked, Average pixel velocity, Homography residual mean error (RMS), and Signal-to-noise ratio

Parameters

- **xyzvel** (*list*) – XYZ velocities
- **uvvel** (*list*) – Pixel velocities
- **homog** (*list*) – Homography [mtx, im0pts, im1pts, ptserr, homogerr]
- **imn** (*list*) – List of image names
- **fname** (*str*) – Filename for output file. File destination can also specified

FileHandler.**writeVeloSHP**(*xyzvel*, *xyzerr*, *xyz0*, *imn*, *fileDirectory*, *projection=None*)

Write OGR real velocity points (from ALL images) to file in a .shp file type that is compatible with ESRI mapping software

Parameters

- **xyzvel** (*list*) – XYZ velocities
- **xyz0** (*list*) – XYZ pt0
- **imn** (*list*) – Image name
- **fileDirectory** (*str*) – Destination that shapefiles will be written to
- **projection** (*int/str, optional*) – Coordinate projection that the shapefile will exist in. This can either be an ESPG number (expressed as an integer) or a well-known geographical coordinate system (expressed as a string). Well-known geographical coordinate systems are: 'WGS84', 'WGS72', NAD83' or 'EPSG:n'

4.5 Images module

4.6 Line module

4.7 Utilities module

4.8 Velocity module

CHAPTER

FIVE

LINKS AND ACKNOWLEDGEMENTS

5.1 PyTrx citations

We are happy for others to use and adapt PyTrx for their own processing needs. If used, please cite the following key publication and Digital Object Identifier:

How et al. (2020) PyTrx: a Python-based monoscopic terrestrial photogrammetry toolset for glaciology. Frontiers in Earth Science 8:21, doi:10.3389/feart.2020.00021

PyTrx has been used in the following publications. In addition to the publication above, please cite any that are applicable where possible:

- How et al. (2019) Calving controlled by melt-undercutting: detailed mechanisms revealed through time-lapse observations. *Annals of Glaciology* 60 (78), 20-31, doi:10.1017/aog.2018.28
- How (2018) Dynamical change at tidewater glaciers examined using time-lapse photogrammetry. PhD thesis, University of Edinburgh, UK, https://hdl.handle.net/1842/31103
- How et al. (2017) Rapidly changing subglacial hydrological pathways at a tidewater glacier revealed through simultaneous observations of water pressure, supraglacial lakes, meltwater plumes and surface velocities. *The Cryosphere* 11, 2691-2710, doi:10.5194/tc-11-2691-2017
- Addison (2015) PyTrx: feature tracking software for automated production of glacier velocity. MSc thesis, University of Edinburgh, UK, https://hdl.handle.net/1842/11794

5.2 Permissions

The DEM of the Kongsfjorden area provided as an example dataset for PyTrx originates from the freely available DEM dataset provided by the Norwegian Polar Institute, data product 'S0 Terrengmodell - Delmodell_5m_2009_13822_33 (GeoTIFF)'. This data is licensed under the Creative Commons Attribution 4.0 International (CC BY 4.0) license:

Norwegian Polar Institute (2014). Terrengmodell Svalbard (S0 Terrengmodell) [Data set]. Norwegian Polar Institute. doi:10.21334/npolar.2014.dce53a47

The two DEMs distributed with PyTrx for the Kongsfjorden region are *KR_demsmooth.tif* and *KR_demzero.tif*, which have been modified and manipulated from the original NPI data. In both cases, the scene has been clipped to the area of interest, downgraded to 20 metre resolution, and smoothed using a linear interpolation method. The latter of these DEMs has been manipulated in order to better represent the terminus position of Kronebreen in 2014 (the time at which the images were taken) and project meltwater plumes to a flat, homogeneous surface at sea level.

The DEM of the Tempelfjorden area provided as an example dataset for PyTrx originates from ArcticDEM, Scene ID: WV01_20130714_1020010 (July 14, 2013). There is no license for the ArcticDEM data and it can be used and distributed freely. The DEM was created from DigitalGlobe, Inc., imagery and funded under National Science Foundation awards 1043681, 1559691, and 1542736.

The DEM distributed with PyTrx of the Tempelfjorden region is called *TU_demzero.tif*, which has been modified and manipulated from the original ArcticDEM data. The scene has been clipped to the area of interest, downgraded to 20 metre resolution, and all low-lying elevations (< 150 m) have been transformed to 0 m a.s.l. in order to project point locations and line profiles to a flat, homogeneous surface at sea level.

The example image sets and UAV-derived DEM from Qasigiannguit glacier are used courtesy of Asiaq Greenland Survey as part of Messerli et al. (In Review), through GlacioBasis Nuuk under the GEM (Greenland Ecosystem Monitoring) programme. The DEM file provided as *QAS_drone_dem.tif* was acquired from UAV surveying in September 2020 and downgraded to 20 metre resolution.

5.3 Acknowledgements

PyTrx was initially developed and released as part of the CRIOS (Calving Rates and Impact on Sea Level) project. Example image sets and camera data from Svalbard glaciers are provided by CRIOS. PyTrx's continued development and maintenance is funded by an ESA Living Planet Fellowship.

Parts of the georectification functions in the PyTrx toolbox were inspired and translated from ImGRAFT, a photogrammetry toolbox for Matlab (Messerli and Grinsted, 2015). Where possible, ImGRAFT has been credited for in the corresponding PyTrx scripts (primarily some passages in the CamEnv.py script) and cited in relevant PyTrx publications.

5.4 Links

There are other useful software available for terrestrial photogrammetry in glaciology:

- Pointcatcher: Matlab-based GUI toolbox for feature-tracking and georectification
- ImGRAFT: Matlab toolbox for feature-tracking and georectification
- EMT (Environmental Motion Tracking): GUI toolbox for feature-tracking and georectification
- CIAS: IDL gui for feature-tracking
- PRACTISE: Matlab toolbox for georectification

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX