```
import numpy as np
import cv2
import scipy.io
import os
from numpy.linalg import norm
from matplotlib import pyplot as plt
from numpy.linalg import det
from numpy.linalg import inv
from scipy.linalg import rq
from numpy.linalg import svd
import matplotlib.pyplot as plt
import numpy as np
import math
import random
import sys
from scipy import ndimage, spatial
from tqdm.notebook import tqdm, trange

import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
from torch.autograd import Variable
import torchvision
from torchvision import datasets, models, transforms
from torch.utils.data import Dataset, DataLoader, ConcatDataset
from skimage import io, transform,data
from torchvision import transforms, utils
import numpy as np
import math
import glob
import matplotlib.pyplot as plt
import time
import os
import copy
import sklearn.svm
import cv2
from matplotlib import pyplot as plt
import numpy as np
from os.path import exists
import pandas as pd
import PIL
import random
from google.colab import drive
from sklearn.metrics.cluster import completeness_score
from sklearn.cluster import KMeans
from tqdm import tqdm, tqdm_notebook
from functools import partial
from torchsummary import summary
from torchvision.datasets import ImageFolder
from torch.utils.data.sampler import SubsetRandomSampler
import h5py as h5

#cuda_output = !ldconfig -p|grep cudart.so|sed -e 's/.*\.\([0-9]*\)\.\([0-9]*\)$/cu\1\2/'
#accelerator = cuda_output[0] if exists('/dev/nvidia0') else 'cpu'

#print("Accelerator type = ",accelerator)
#print("Pytorch verision: ", torch.__version__)
```

```
from google.colab import drive

# This will prompt for authorization.
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

```
#!cp "/content/drive/My Drive/cv2_gpu/cv2.cpython-37m-x86_64-linux-gnu.so" .
```

```
cv2.__version__
```

```
'4.5.3-pre'
```

```
def warpnImages_mod(len_H_left,len_H_right,scale_factor=16,offset=0):
    #img1-centre,img2-left,img3-right
    f=h5.File('drive/MyDrive/all_images_bgr_sift_443.h5','r')
    img = f['data'][0]
    f.close()
    h, w = img.shape[:2]
    h = round(h/scale_factor)
    w = round(w/scale_factor)

    pts_left = []
    pts_right = []

    pts_centre = np.float32([[0, 0], [0, h], [w, h], [w, 0]]).reshape(-1, 1, 2)

    for j in range(offset,len_H_left):
      pts = np.float32([[0, 0], [0, h], [w, h], [w, 0]]).reshape(-1, 1, 2)
      pts_left.append(pts)

    for j in range(offset,len_H_right):
      pts = np.float32([[0, 0], [0, h], [w, h], [w, 0]]).reshape(-1, 1, 2)
      pts_right.append(pts)

    pts_left_transformed=[]
    pts_right_transformed=[]

    H_scale = np.eye(3)
    H_scale[0,0] = H_scale[1,1] = 1/scale_factor
    H_scale[0,1] = H_scale[1,0] = 1
    H_scale[0,2] = H_scale[1,2] = scale_factor
    H_scale[2,0] = H_scale[2,1] = 1/scale_factor
    #H_scale[0,0] = H_scale[1,1] = 1/scale_factor


    for j,pts in enumerate(pts_left):
      if j==0:
        f=h5.File('drive/MyDrive/H_left_sift_220.h5','r')
        H_trans = f['data'][j+offset]
        f.close()
        #H_trans = H_left[j]
      else:
```

```
      f=h5.File('drive/MyDrive/H_left_sift_220.h5','r')
      H_trans = H_trans@f['data'][j+offset]
      f.close()
      #H_trans = H_trans@H_left[j]
    #H_trans[0,2] = (1/scale_factor) * H_trans[0,2]
    #H_trans[1,2] = (1/scale_factor) * H_trans[1,2]
    #H_trans[2,0] = (scale_factor) * H_trans[2,0]

    if scale_factor>1:
      pts_ = cv2.perspectiveTransform(pts, H_trans@np.linalg.inv(H_scale))
    else:
      pts_ = cv2.perspectiveTransform(pts, H_trans)

    pts_left_transformed.append(pts_)

  for j,pts in enumerate(pts_right):
    if j==0:
      f=h5.File('drive/MyDrive/H_right_sift_222.h5','r')
      H_trans = f['data'][j+offset]
      f.close()
      #H_trans = H_right[j]
    else:
      f=h5.File('drive/MyDrive/H_right_sift_222.h5','r')
      H_trans = H_trans@f['data'][j+offset]
      f.close()
      #H_trans = H_trans@H_right[j]
    #H_trans[0,2] = (1/scale_factor) * H_trans[0,2]
    #H_trans[1,2] = (1/scale_factor) * H_trans[1,2]
    #H_trans[2,0] = (scale_factor) * H_trans[2,0]

    if scale_factor>1:
      pts_ = cv2.perspectiveTransform(pts, H_trans@np.linalg.inv(H_scale))
    else:
      pts_ = cv2.perspectiveTransform(pts, H_trans)

    pts_right_transformed.append(pts_)


  print('Step1:Done')


  #pts = np.concatenate((pts1, pts2_), axis=0)

  pts_concat = np.concatenate((pts_centre,np.concatenate(np.array(pts_left_transformed),axis=0),np.concatenate(np.array(pts_right_transformed),axis=0)), axis=0)

  [xmin, ymin] = np.int32(pts_concat.min(axis=0).ravel() - 0.5)
  [xmax, ymax] = np.int32(pts_concat.max(axis=0).ravel() + 0.5)
  t = [-xmin, -ymin]
  Ht = np.array([[1, 0, t[0]], [0, 1, t[1]], [0, 0, 1]])  # translate
  #Ht = Ht*scale_factor

  print('Step2:Done')


  return xmax,xmin,ymax,ymin,t,h,w,Ht
```

```
def final_steps_right_union_gpu_mod(warp_img_init_prev,len_H_right,xmax,xmin,ymax,ymin,t,h,w,Ht,scale_factor=16,is_gray=True):


  from tqdm import tqdm
  tqdm = partial(tqdm, position=0, leave=True)
  H_scale = np.eye(3)
  H_scale[0,0] = H_scale[1,1] = 1/scale_factor
  H_scale[0,1] = H_scale[1,0] = 1
  H_scale[0,2] = H_scale[1,2] = scale_factor
  H_scale[2,0] = H_scale[2,1] = 1/scale_factor
  #H_scale[0,0] = H_scale[1,1] = 1/scale_factor

  for j in tqdm(range(len_H_right)):
    #print(j)
    f=h5.File('drive/MyDrive/H_right_sift_222.h5','r')
    H = f['data'][j]
    f.close()
    if scale_factor>1:
      H =  H@np.linalg.inv(H_scale)
    if j==0:
      H_trans = Ht@H
    else:
      H_trans = H_trans@H

    f=h5.File('drive/MyDrive/all_images_bgr_sift_443.h5','r')
    input_img_orig = f['data'][(len_H_right)+j+2]
    f.close()
    del f
    src = cv2.cuda_GpuMat()
    src.upload( np.uint8(input_img_orig))
    if scale_factor>1:
      dst = cv2.cuda.resize(src,None,fx=(1/scale_factor),fy = (1/scale_factor),interpolation = cv2.INTER_CUBIC)
    else:
      dst = src
    #input_img = dst.download()
    if is_gray==True:
      dst = cv2.cuda.cvtColor(dst, cv2.COLOR_BGR2GRAY)
    #print('input image accesssed')
    input_img = dst.download()
    #input_img = images_right[j+1]
    #result = np.zeros((ymax-ymin,xmax-xmin,3),dtype='uint8')

    src = cv2.cuda_GpuMat()
    src.upload( np.uint8(input_img))

    #dst = cv2.cuda_GpuMat()
    #dst.upload(result)

    #print('Step 42: Done')


    dst = cv2.cuda.warpPerspective(src, M = H_trans, dsize = (xmax-xmin, ymax-ymin) )



    #cv2.warpPerspective(src = np.uint8(input_img), M = H_trans, dsize = (xmax-xmin, ymax-ymin),dst=result)
    del input_img
    result = dst.download()
```

```
      warp_img_init_curr = result

      del result
      #print('Step 44: Done')

      if is_gray==True:
        inds = warp_img_init_prev[:, :] == 0
      else:
        inds = warp_img_init_prev[:, :, 0] == 0
        inds &= warp_img_init_prev[:, :, 1] == 0
        inds &= warp_img_init_prev[:, :, 2] == 0
      #print('Step 45: Done')


      warp_img_init_prev[inds] = warp_img_init_curr[inds]
      #print('Step 46: Done')

      plt.clf()
      plt.imshow(warp_img_init_prev,cmap='gray')
      plt.show()
      plt.imshow(warp_img_init_curr,cmap='gray')
      plt.show()


      del inds,warp_img_init_curr

    return warp_img_init_prev
```

```
#%%file mprun_demo31.py
import numpy as np
import cv2
import h5py as h5
import tqdm

def final_steps_left_union(len_H_left,xmax,xmin,ymax,ymin,t,h,w,Ht,scale_factor=16):


    for j in range(len_H_left):
      print(j)
      f=h5.File('drive/MyDrive/H_left_sift_220.h5','r')
      H = f['data'][j]
      f.close()
      if j==0:
        H_trans = Ht.dot(H)
      else:
        H_trans = H_trans.dot(H)

      f=h5.File('drive/MyDrive/all_images_bgr_sift_443.h5','r')
      input_img_orig = f['data'][j+1]
      f.close()
      del f
      input_img = cv2.resize(input_img_orig,None,fx=(1/scale_factor),fy = (1/scale_factor),interpolation = cv2.INTER_CUBIC)
      #input_img = cv2.cvtColor(input_img, cv2.COLOR_BGR2GRAY)
      #print('input image accesssed')

      #input_img = images_left[j+1]
      result = np.zeros((ymax-ymin,xmax-xmin,3),dtype='uint8')
      #print('output init done')

      cv2.warpPerspective(src = np.uint8(input_img), M = H_trans, dsize = (xmax-xmin, ymax-ymin),dst=result)
      del input_img
      warp_img_init_curr = result

      if j==0:
        f=h5.File('drive/MyDrive/all_images_bgr_sift_443.h5','r')
        first_img_orig = f['data'][0]
        f.close()
        del f
        first_img = cv2.resize(first_img_orig,None,fx=(1/scale_factor),fy = (1/scale_factor),interpolation = cv2.INTER_CUBIC)
        #first_img = cv2.cvtColor(first_img, cv2.COLOR_BGR2GRAY)
        result[t[1]:h+t[1], t[0]:w+t[0]] = first_img
        warp_img_init_prev = result
        continue
      #inds = warp_img_init_prev[:, :] == 0
      del result

      inds = warp_img_init_prev[:, :, 0] == 0
      inds &= warp_img_init_prev[:, :, 1] == 0
      inds &= warp_img_init_prev[:, :, 2] == 0

      #black_pixels = np.where((warp_img_init_prev[:, :, 0] == 0) & (warp_img_init_prev[:, :, 1] == 0) & (warp_img_init_prev[:, :, 2] == 0))

      warp_img_init_prev[inds] = warp_img_init_curr[inds]

      del inds,warp_img_init_curr



    print('Step31:Done')

    return warp_img_init_prev
```

```
#%%file mprun_demo31.py
import numpy as np
import cv2
import h5py as h5
import tqdm

def final_steps_left_union_gpu_mod(len_H_left,xmax,xmin,ymax,ymin,t,h,w,Ht,warp_img_init_prev ,scale_factor=16,is_gray=True,offset=0,H_trans=np.eye(3)):
    from tqdm import tqdm
    tqdm = partial(tqdm, position=0, leave=True)
    H_scale = np.eye(3)
    H_scale[0,0] = H_scale[1,1] = 1/scale_factor
    H_scale[0,1] = H_scale[1,0] = 1
    H_scale[0,2] = H_scale[1,2] = scale_factor
    #H_scale[2,0] = H_scale[2,1] = 1/scale_factor
    #H_scale[0,0] = H_scale[1,1] = 1/scale_factor

    for j in tqdm(range(offset,len_H_left)):
      #print(j)
      f=h5.File('drive/MyDrive/H_left_sift_220.h5','r')
      H = f['data'][j]
      f.close()
      if scale_factor>1:
        H =  H@np.linalg.inv(H_scale)
```

```
      if j==0:
        H_trans = Ht.dot(H)
      else:
        H_trans = H_trans.dot(H)

      f=h5.File('drive/MyDrive/all_images_bgr_sift_443.h5','r')
      input_img_orig = f['data'][j+1]
      f.close()
      del f
      src = cv2.cuda_GpuMat()
      src.upload( np.uint8(input_img_orig))
      if scale_factor>1:
        dst = cv2.cuda.resize(src,None,fx=(1/scale_factor),fy = (1/scale_factor),interpolation = cv2.INTER_CUBIC)
      else:
        dst = src
      #input_img = dst.download()

      if is_gray==True:
        dst = cv2.cuda.cvtColor(dst, cv2.COLOR_BGR2GRAY)
      #print('input image accesssed')
      input_img = dst.download()

      #input_img = images_left[j+1]
      #result = np.zeros((ymax-ymin,xmax-xmin,3),dtype='uint8')
      #print('output init done')
      src = cv2.cuda_GpuMat()
      src.upload( np.uint8(input_img))



      #print('Step 42: Done')
      #if is_gray==False:
      #  result = np.zeros((ymax-ymin,xmax-xmin,3),dtype='uint8')
      #else:
      #  result = np.zeros((ymax-ymin,xmax-xmin),dtype='uint8')

      #dst = cv2.cuda_GpuMat()
      #dst.upload(result)


      dst = cv2.cuda.warpPerspective(src, M = H_trans, dsize = (xmax-xmin, ymax-ymin) )





      #cv2.warpPerspective(src = np.uint8(input_img), M = H_trans, dsize = (xmax-xmin, ymax-ymin),dst=result)
      del input_img
      result = dst.download()
      warp_img_init_curr = result
      #print('Step 43: Done')

      if j==0:
        f=h5.File('drive/MyDrive/all_images_bgr_sift_443.h5','r')
        first_img_orig = f['data'][0]
        f.close()
        del f
        src = cv2.cuda_GpuMat()
        src.upload(np.uint8(first_img_orig))
        if scale_factor>1:
          dst = cv2.cuda.resize(src,None,fx=(1/scale_factor),fy = (1/scale_factor),interpolation = cv2.INTER_CUBIC)
        else:
          dst = src
        #first_img = dst.download()
        #first_img = cv2.resize(first_img_orig,None,fx=(1/scale_factor),fy = (1/scale_factor),interpolation = cv2.INTER_CUBIC)
        if is_gray==True:
          dst = cv2.cuda.cvtColor(dst, cv2.COLOR_BGR2GRAY)
        first_img = dst.download()
        result[t[1]:h+t[1], t[0]:w+t[0]] = first_img
        warp_img_init_prev = result
        continue
      del result
      #print('Step 44: Done')

      if is_gray==True:
        inds = warp_img_init_prev[:, :] == 0
      else:
        inds = warp_img_init_prev[:, :, 0] == 0
        inds &= warp_img_init_prev[:, :, 1] == 0
        inds &= warp_img_init_prev[:, :, 2] == 0
      #print('Step 45: Done')

      #black_pixels = np.where((warp_img_init_prev[:, :, 0] == 0) & (warp_img_init_prev[:, :, 1] == 0) & (warp_img_init_prev[:, :, 2] == 0))

      plt.clf()
      plt.imshow(warp_img_init_prev,cmap='gray')
      plt.show()
      plt.imshow(warp_img_init_curr,cmap='gray')
      plt.show()

      warp_img_init_prev[inds] = warp_img_init_curr[inds]
      #print('Step 46: Done')

      plt.clf()
      plt.imshow(warp_img_init_prev,cmap='gray')
      plt.show()
      plt.imshow(warp_img_init_curr,cmap='gray')
      plt.show()


      del inds,warp_img_init_curr



    print('Step31:Done')

    return warp_img_init_prev
```

```
f=h5.File('drive/MyDrive/H_left_sift_220.h5','r')
H_trans = f['data'][0]
f.close()
```

```
print(H_trans.shape)
```

```
(3, 3)
```

```
scale_factor=16
H_scale = np.eye(3)
#H_scale[0,1] = H_scale[1,0] = 1
#H_scale[0,2] = H_scale[1,2] = scale_factor
#H_scale[2,0] = H_scale[2,1] = 1/scale_factor
H_scale[0,0] = H_scale[1,1] = scale_factor
```

```
print(H_trans)
```

```
[[ 1.21655743e+00  5.73600495e-02 -2.24433882e+02]
 [ 5.55579072e-02  1.19496478e+00  7.28613496e+01]
 [ 8.30986639e-05  2.27364524e-05  1.00000000e+00]]
```

```
print(H_trans@np.linalg.inv(H_scale))
```

```
[[ 7.60348396e-02  3.58500309e-03 -2.24433882e+02]
 [ 3.47236920e-03  7.46852986e-02  7.28613496e+01]
 [ 5.19366649e-06  1.42102828e-06  1.00000000e+00]]
```

```
def warpnImages(len_H_left,len_H_right,scale_factor=16,offset=0):
    #img1-centre,img2-left,img3-right
    f=h5.File('drive/MyDrive/all_images_bgr_sift_300.h5','r')
    img = f['data'][0]
    f.close()
    h, w = img.shape[:2]
    h = round(h/scale_factor)
    w = round(w/scale_factor)

    pts_left = []
    pts_right = []

    pts_centre = np.float32([[0, 0], [0, h], [w, h], [w, 0]]).reshape(-1, 1, 2)

    for j in range(offset,len_H_left):
      pts = np.float32([[0, 0], [0, h], [w, h], [w, 0]]).reshape(-1, 1, 2)
      pts_left.append(pts)

    for j in range(offset,len_H_right):
      pts = np.float32([[0, 0], [0, h], [w, h], [w, 0]]).reshape(-1, 1, 2)
      pts_right.append(pts)

    pts_left_transformed=[]
    pts_right_transformed=[]

    H_scale = np.eye(3)
    #H_scale[0,0] = H_scale[1,1] = 1/scale_factor
    #H_scale[0,1] = H_scale[1,0] = 1
    #H_scale[0,2] = H_scale[1,2] = scale_factor
    #H_scale[2,0] = H_scale[2,1] = 1/scale_factor
    H_scale[0,0] = H_scale[1,1] = 1/scale_factor


    for j,pts in enumerate(pts_left):
      if j==0:
        f=h5.File('drive/MyDrive/H_left_sift_120.h5','r')
        H_trans = f['data'][j+offset]
        f.close()
        #H_trans = H_left[j]
      else:
        f=h5.File('drive/MyDrive/H_left_sift_120.h5','r')
        H_trans = H_trans@f['data'][j+offset]
        f.close()
        #H_trans = H_trans@H_left[j]
      #H_trans[0,2] = (1/scale_factor) * H_trans[0,2]
      #H_trans[1,2] = (1/scale_factor) * H_trans[1,2]
      #H_trans[2,0] = (scale_factor) * H_trans[2,0]

      if scale_factor>1:
        pts_ = cv2.perspectiveTransform(pts, H_scale@H_trans@np.linalg.inv(H_scale))
      else:
        pts_ = cv2.perspectiveTransform(pts, H_trans)

      pts_left_transformed.append(pts_)

    for j,pts in enumerate(pts_right):
      if j==0:
        f=h5.File('drive/MyDrive/H_right_sift_130.h5','r')
        H_trans = f['data'][j+offset]
        f.close()
        #H_trans = H_right[j]
      else:
        f=h5.File('drive/MyDrive/H_right_sift_130.h5','r')
        H_trans = H_trans@f['data'][j+offset]
        f.close()
        #H_trans = H_trans@H_right[j]
      #H_trans[0,2] = (1/scale_factor) * H_trans[0,2]
      #H_trans[1,2] = (1/scale_factor) * H_trans[1,2]
      #H_trans[2,0] = (scale_factor) * H_trans[2,0]

      if scale_factor>1:
        pts_ = cv2.perspectiveTransform(pts, H_scale@H_trans@np.linalg.inv(H_scale))
      else:
        pts_ = cv2.perspectiveTransform(pts, H_trans)

      pts_right_transformed.append(pts_)


    print('Step1:Done')


    #pts = np.concatenate((pts1, pts2_), axis=0)

    pts_concat = np.concatenate((pts_centre,np.concatenate(np.array(pts_left_transformed),axis=0),np.concatenate(np.array(pts_right_transformed),axis=0)), axis=0)

    [xmin, ymin] = np.int32(pts_concat.min(axis=0).ravel() - 0.5)
    [xmax, ymax] = np.int32(pts_concat.max(axis=0).ravel() + 0.5)
    t = [-xmin, -ymin]
    Ht = np.array([[1, 0, t[0]], [0, 1, t[1]], [0, 0, 1]])  # translate
    #Ht = Ht*scale_factor

    print('Step2:Done')


    return xmax,xmin,ymax,ymin,t,h,w,Ht
```

```python
#%%file mprun_demo31.py
import numpy as np
import cv2
import h5py as h5
import tqdm

def final_steps_left_union_gpu(len_H_left,xmax,xmin,ymax,ymin,t,h,w,Ht,warp_img_init_prev ,scale_factor=16,is_gray=True,offset=0,H_trans=np.eye(3)):
    from tqdm import tqdm
    tqdm = partial(tqdm, position=0, leave=True)
    H_scale = np.eye(3)
    #H_scale[0,0] = H_scale[1,1] = 1/scale_factor
    #H_scale[0,1] = H_scale[1,0] = 1
    #H_scale[0,2] = H_scale[1,2] = scale_factor
    #H_scale[2,0] = H_scale[2,1] = 1/scale_factor
    H_scale[0,0] = H_scale[1,1] = 1/scale_factor

    for j in tqdm(range(offset,len_H_left)):
      #print(j)
      f=h5.File('drive/MyDrive/H_left_sift_120.h5','r')
      H = f['data'][j]
      f.close()
      if scale_factor>1:
        H =  H_scale@H@np.linalg.inv(H_scale)
      if j==0:
        H_trans = Ht.dot(H)
      else:
        H_trans = H_trans.dot(H)

      f=h5.File('drive/MyDrive/all_images_bgr_sift_300.h5','r')
      input_img_orig = f['data'][j+1]
      f.close()
      del f
      src = cv2.cuda_GpuMat()
      src.upload( np.uint8(input_img_orig))
      if scale_factor>1:
        dst = cv2.cuda.resize(src,None,fx=(1/scale_factor),fy = (1/scale_factor),interpolation = cv2.INTER_CUBIC)
      else:
        dst = src
      #input_img = dst.download()

      if is_gray==True:
        dst = cv2.cuda.cvtColor(dst, cv2.COLOR_BGR2GRAY)
      #print('input image accesssed')
      input_img = dst.download()

      #input_img = images_left[j+1]
      #result = np.zeros((ymax-ymin,xmax-xmin,3),dtype='uint8')
      #print('output init done')
      src = cv2.cuda_GpuMat()
      src.upload( np.uint8(input_img))


      '''
      #print('Step 42: Done')
      if is_gray==False:
        #result = np.zeros((ymax-ymin,xmax-xmin,3),dtype='uint8')
        result = lil_matrix((ymax-ymin,xmax-xmin,3))
      else:
        #result = np.zeros((ymax-ymin,xmax-xmin),dtype='uint8')
        result = lil_matrix((ymax-ymin,xmax-xmin))

      dst = cv2.cuda_GpuMat()
      dst.upload(result.toarray())

      '''


      dst = cv2.cuda.warpPerspective(src, M = H_trans, dsize = (xmax-xmin, ymax-ymin) )





      #cv2.warpPerspective(src = np.uint8(input_img), M = H_trans, dsize = (xmax-xmin, ymax-ymin),dst=result)
      del input_img
      result = dst.download()
      warp_img_init_curr = result
      #print('Step 43: Done')

      if j==0:
        f=h5.File('drive/MyDrive/all_images_bgr_sift_300.h5','r')
        first_img_orig = f['data'][0]
        f.close()
        del f
        src = cv2.cuda_GpuMat()
        src.upload(np.uint8(first_img_orig))
        if scale_factor>1:
          dst = cv2.cuda.resize(src,None,fx=(1/scale_factor),fy = (1/scale_factor),interpolation = cv2.INTER_CUBIC)
        else:
          dst = src
        #first_img = dst.download()
        #first_img = cv2.resize(first_img_orig,None,fx=(1/scale_factor),fy = (1/scale_factor),interpolation = cv2.INTER_CUBIC)
        if is_gray==True:
          dst = cv2.cuda.cvtColor(dst, cv2.COLOR_BGR2GRAY)
        first_img = dst.download()
        result[t[1]:h+t[1], t[0]:w+t[0]] = first_img
        warp_img_init_prev = result
        continue
      del result
      #print('Step 44: Done')

      if is_gray==True:
        inds = warp_img_init_prev[:, :] == 0
      else:
        inds = warp_img_init_prev[:, :, 0] == 0
        inds &= warp_img_init_prev[:, :, 1] == 0
        inds &= warp_img_init_prev[:, :, 2] == 0
      #print('Step 45: Done')

      #black_pixels = np.where((warp_img_init_prev[:, :, 0] == 0) & (warp_img_init_prev[:, :, 1] == 0) & (warp_img_init_prev[:, :, 2] == 0))
      '''
      plt.clf()
      plt.imshow(warp_img_init_prev,cmap='gray')
      plt.show()
      plt.imshow(warp_img_init_curr,cmap='gray')
      plt.show()
```

```
      plt.show()
      '''
      warp_img_init_prev[inds] = warp_img_init_curr[inds]
      #print('Step 46: Done')
      '''
      plt.clf()
      plt.imshow(warp_img_init_prev,cmap='gray')
      plt.show()
      plt.imshow(warp_img_init_curr,cmap='gray')
      plt.show()
      '''

      del inds,warp_img_init_curr



    print('Step31:Done')

    return warp_img_init_prev
```

```
def final_steps_right_union_gpu(warp_img_init_prev,len_H_right,xmax,xmin,ymax,ymin,t,h,w,Ht,scale_factor=16,is_gray=True):


    from tqdm import tqdm
    tqdm = partial(tqdm, position=0, leave=True)
    H_scale = np.eye(3)
    #H_scale[0,0] = H_scale[1,1] = 1/scale_factor
    #H_scale[0,1] = H_scale[1,0] = 1
    #H_scale[0,2] = H_scale[1,2] = scale_factor
    #H_scale[2,0] = H_scale[2,1] = 1/scale_factor
    H_scale[0,0] = H_scale[1,1] = 1/scale_factor

    for j in tqdm(range(len_H_right)):
      #print(j)
      f=h5.File('drive/MyDrive/H_right_sift_130.h5','r')
      H = f['data'][j]
      f.close()
      if scale_factor>1:
        H =  H_scale@H@np.linalg.inv(H_scale)
      if j==0:
        H_trans = Ht@H
      else:
        H_trans = H_trans@H

      f=h5.File('drive/MyDrive/all_images_bgr_sift_300.h5','r')
      input_img_orig = f['data'][(len_H_right)+j+2]
      f.close()
      del f
      src = cv2.cuda_GpuMat()
      src.upload( np.uint8(input_img_orig))
      if scale_factor>1:
        dst = cv2.cuda.resize(src,None,fx=(1/scale_factor),fy = (1/scale_factor),interpolation = cv2.INTER_CUBIC)
      else:
        dst = src
      #input_img = dst.download()
      if is_gray==True:
        dst = cv2.cuda.cvtColor(dst, cv2.COLOR_BGR2GRAY)
      #print('input image accesssed')
      input_img = dst.download()
      #input_img = images_right[j+1]
      #result = np.zeros((ymax-ymin,xmax-xmin,3),dtype='uint8')

      src = cv2.cuda_GpuMat()
      src.upload( np.uint8(input_img))

      #dst = cv2.cuda_GpuMat()
      #dst.upload(result)

      #print('Step 42: Done')


      dst = cv2.cuda.warpPerspective(src, M = H_trans, dsize = (xmax-xmin, ymax-ymin) )



      #cv2.warpPerspective(src = np.uint8(input_img), M = H_trans, dsize = (xmax-xmin, ymax-ymin),dst=result)
      del input_img
      result = dst.download()

      warp_img_init_curr = result

      del result
      #print('Step 44: Done')

      if is_gray==True:
        inds = warp_img_init_prev[:, :] == 0
      else:
        inds = warp_img_init_prev[:, :, 0] == 0
        inds &= warp_img_init_prev[:, :, 1] == 0
        inds &= warp_img_init_prev[:, :, 2] == 0
      #print('Step 45: Done')


      warp_img_init_prev[inds] = warp_img_init_curr[inds]
      #print('Step 46: Done')
      '''
      plt.clf()
      plt.imshow(warp_img_init_prev,cmap='gray')
      plt.show()
      plt.imshow(warp_img_init_curr,cmap='gray')
      plt.show()
      '''

      del inds,warp_img_init_curr

    return warp_img_init_prev
```

BA Optmizer

```
def calculate_rcenter_affine(src_list, dst_list):
    log('get_recenter_affine():')
    src = [[], [], [], []]      # current camera locations
```

```
    dst = [[], [], [], []]      # original camera locations
    for i in range(len(src_list)):
        src_ned = src_list[i]
        src[0].append(src_ned[0])
        src[1].append(src_ned[1])
        src[2].append(src_ned[2])
        src[3].append(1.0)
        dst_ned = dst_list[i]
        dst[0].append(dst_ned[0])
        dst[1].append(dst_ned[1])
        dst[2].append(dst_ned[2])
        dst[3].append(1.0)
        # print("{} <-- {}".format(dst_ned, src_ned))
    A = transformations.superimposition_matrix(src, dst, scale=True)
    log("A:\n", A)
    return A

# transform a point list given an affine transform matrix
def transform_points( A, pts_list ):
    src = [[], [], [], []]
    for p in pts_list:
        src[0].append(p[0])
        src[1].append(p[1])
        src[2].append(p[2])
        src[3].append(1.0)
    dst = A.dot( np.array(src) )
    result = []
    for i in range(len(pts_list)):
        result.append( [ float(dst[0][i]),
                         float(dst[1][i]),
                         float(dst[2][i]) ] )
    return result

# This is a python class that optimizes the estimate camera and 3d
# point fits by minimizing the mean reprojection error.
class Optimizer():
    def __init__(self, root):
        self.root = root
        self.camera_map_fwd = {}
        self.camera_map_rev = {}
        self.feat_map_fwd = {}
        self.feat_map_rev = {}
        self.last_mre = None
        self.graph = None
        #self.graph_counter = 0
        #self.optimize_calib = 'global' # global camera optimization
        self.optimize_calib = 'none' # no camera calibration optimization
        #self.ftol = 1e-2              # stop condition - extra coarse
        self.ftol = 1e-3              # stop condition - quicker
        #self.ftol = 1e-4              # stop condition - better
        self.min_chain_len = 2        # use whatever matches are defind upstream
        self.with_bounds = True
        #self.cam_method = 'rvec_tvec'
        self.cam_method = 'ned_quat'
        if self.cam_method == 'rvec_tvec':
            self.ncp = 6           # 3 tvec values, 3 rvec values
        elif self.cam_method == 'ned_quat':
            self.ncp = 7            # 3 ned values, 4 quat values
        self.cam2body = np.array( [[0, 0, 1],
                                   [1, 0, 0],
                                   [0, 1, 0]], dtype=float )
        self.body2cam = np.linalg.inv(self.cam2body)

    # plot range
    def my_plot_range(self, data, stats=False):
        if stats:
            avg = np.mean(data)
            std = np.std(data)
            min = math.floor((avg-3*std) / 10) * 10
            max = math.ceil((avg+3*std) / 10) * 10
        else:
            min = math.floor(np.amin(data) / 10) * 10
            max = math.ceil(np.amax(data) / 10) * 10
        return min, max

    #  input rvec, tvec, and return
    # corresponding ypr and ned values
    def rvectvec2yprned(self, rvec, tvec):
        Rned2cam, jac = cv2.Rodrigues(rvec)
        Rned2body = self.cam2body.dot(Rned2cam)
        Rbody2ned = np.matrix(Rned2body).T
        ypr = transformations.euler_from_matrix(Rbody2ned, 'rzyx')
        pos = -np.matrix(Rned2cam).T * np.matrix(tvec).T
        ned = np.squeeze(np.asarray(pos.T[0]))
        return ypr, ned

    def nedquat2rvectvec(self, ned, quat):
        body2ned = transformations.quaternion_matrix(np.array(quat))[:3,:3]
        ned2body = body2ned.T
        R = self.body2cam.dot( ned2body )
        rvec, jac = cv2.Rodrigues(R)
        tvec = -np.matrix(R) * np.matrix(ned).T
        return rvec, tvec

    # compute the sparsity matrix (dependency relationships between
    # observations and parameters the optimizer can manipulate.)
    # Because of the extreme number of parameters and observations, a
    # sparse matrix is required to run in finite time for all but the
    # smallest data sets.
    def bundle_adjustment_sparsity(self, n_cameras, n_points,
                                   camera_indices, point_indices):
        m = camera_indices.size * 2
        n = n_cameras * self.ncp + n_points * 3
        if self.optimize_calib == 'global':
            n += 8  # three K params (fx == fy) + five distortion params
        A = lil_matrix((m, n), dtype=int)
        log('sparsity matrix is %d x %d' % (m, n))

        i = np.arange(camera_indices.size)
        for s in range(self.ncp):
            A[2 * i, camera_indices * self.ncp + s] = 1
            A[2 * i + 1, camera_indices * self.ncp + s] = 1

        for s in range(3):
            A[2 * i, n_cameras * self.ncp + point_indices * 3 + s] = 1
            A[2 * i + 1, n_cameras * self.ncp + point_indices * 3 + s] = 1

        if self.optimize_calib == 'global':
```

```
            for s in range(0,3): # K
                A[2 * i, n_cameras * self.ncp + n_points * 3 + s] = 1
                A[2 * i + 1, n_cameras * self.ncp + n_points * 3 + s] = 1
            for s in range(3,8): # dist coeffs
                A[2 * i, n_cameras * self.ncp + n_points * 3 + s] = 1
                A[2 * i + 1, n_cameras * self.ncp + n_points * 3 + s] = 1

        log('A-matrix non-zero elements:', A.nnz)
        return A

    # compute an array of residuals (one for each observation)
    # params contains camera parameters, 2-D coordinates, and
    # camera calibration parameters.
    def residuals(self, params, n_cameras, n_points, by_camera_point_indices, by_camera_points_2d):
        error = None
        # extract the parameters
        camera_params = params[:n_cameras * self.ncp].reshape((n_cameras, self.ncp))

        points_3d = params[n_cameras * self.ncp:n_cameras * self.ncp + n_points * 3].reshape((n_points, 3))

        if self.optimize_calib == 'global':
            # assemble K and distCoeffs from the optimizer param list
            camera_calib = params[n_cameras * self.ncp + n_points * 3:]
            K = np.identity(3)
            K[0,0] = camera_calib[0]
            K[1,1] = camera_calib[0]
            K[0,2] = camera_calib[1]
            K[1,2] = camera_calib[2]
            distCoeffs = camera_calib[3:]
        else:
            # use a fixed K and distCoeffs
            K = self.K
            distCoeffs = self.distCoeffs

        #fixme: global calibration optimization, but force distortion
        #paramters to stay fixed to those originally given
        #distCoeffs = self.distCoeffs

        sum = 0
        # cams_3d = np.zeros((n_cameras, 3)) # for plotting
        by_cam = []             # for debugging data set problems
        for i, cam in enumerate(camera_params):
            if len(by_camera_point_indices[i]) == 0:
                continue
            if self.cam_method == 'rvec_tvec':
                rvec = cam[:3]
                tvec = cam[3:6]
            elif self.cam_method == 'ned_quat':
                ned = cam[:3]
                quat = cam[3:7]
                rvec, tvec = self.nedquat2rvectvec(ned, quat)
                #print(i, ned, quat)
            # ypr, ned = self.rvectvec2yprned(rvec, tvec)
            # cams_3d[i] = ned # for plotting
            proj_points, jac = cv2.projectPoints(points_3d[by_camera_point_indices[i]], rvec, tvec, K, distCoeffs)
            sum += len(proj_points.ravel())
            cam_error = (by_camera_points_2d[i] - proj_points).ravel()
            by_cam.append( [np.mean(np.abs(cam_error)),
                            np.amax(np.abs(cam_error)),
                            self.camera_map_fwd[i] ] )
            if error is None:
                error = cam_error
            else:
                error = np.append(error, cam_error)

        mre = np.mean(np.abs(error))
        std = np.std(error)

        # debug
        count_std = 0
        count_bad = 0
        for e in error.tolist():
            if e > mre + 3 * std:
                count_std += 1
            if e > 10000:
                count_bad += 1
        # print( 'std: %.2f %d/%d > 3*std (max: %.2f)' % (std, count_std, error.shape[0], np.amax(error)) )
        # by_cam = sorted(by_cam, key=lambda fields: fields[0], reverse=True)
        # for line in by_cam:
        #     if line[0] > mre + 2*std:
        #         print("  %s -- mean: %.3f max: %.3f" % (line[2], line[0], line[1]))

        # provide some runtime feedback for the operator
        if self.last_mre is None or 1.0 - mre/self.last_mre > 0.001:
            # mre has improved by more than 0.1%
            self.last_mre = mre
            log('mre: %.3f std: %.3f max: %.2f' % (mre, np.std(error), np.amax(np.abs(error))) )
            if self.optimize_calib == 'global':
                log("K:\n", K)
                log("distCoeffs: %.3f %.3f %.3f %.3f %.3f" %
                    (distCoeffs[0], distCoeffs[1], distCoeffs[2],
                     distCoeffs[3], distCoeffs[4]))
            # if not self.graph is None:
            #     points = points_3d
            #     #points = cams_3d
            #     self.graph.set_offsets(points[:,[1,0]])
            #     self.graph.set_array(-points[:,2])
            #     xmin, xmax = self.my_plot_range(points[:,1])
            #     ymin, ymax = self.my_plot_range(points[:,0])
            #     plt.xlim(xmin, xmax)
            #     plt.ylim(ymin, ymax)
            #     cmin, cmax = self.my_plot_range(-points[:,2], stats=True)
            #     plt.clim(cmin, cmax)
            #     plt.gcf().set_size_inches(16,9,forward=True)
            #     plt.draw()
            #     if False:
            #         # animate the optimizer progress as a movie
            #         # ex: ffmpeg -f image2 -r 2 -s 1280x720 -i optimizer-%03d.png -vcodec libx264 -crf 25  -pix_fmt yuv420p optimizer.mp4
            #         plt_name = 'optimizer-%03d.png' % self.graph_counter
            #         out_file = os.path.join(self.root, plt_name)
            #         plt.savefig(out_file, dpi=80)
            #         self.graph_counter += 1
            #     plt.pause(0.01)
        return error

    # assemble the structures and remapping indices required for
    # optimizing a group of images/features
    def assemble_initialization(self, proj, groups, group_index, matches_list, optimized=False,
```

```
            cam_calib=False):
        log('Setting up optimizer data structures...')
        if cam_calib:
            self.optimize_calib = 'global' # global camera optimization
        else:
            self.optimize_calib = 'none' # no camera calibration optimization

        # if placed_images == None:
        #     placed_images = []
        #     # if no placed images specified, mark them all as placed
        #     for i in range(len(proj.image_list)):
        #         placed_images.append(i)
        placed_images = set()
        for name in groups[group_index]:
            i = proj.findIndexByName(name)
            placed_images.add(i)
        log('Number of placed images:', len(placed_images))

        # construct the camera index remapping
        self.camera_map_fwd = {}
        self.camera_map_rev = {}
        for i, index in enumerate(placed_images):
            self.camera_map_fwd[i] = index
            self.camera_map_rev[index] = i
        #print(self.camera_map_fwd)
        #print(self.camera_map_rev)

        # initialize the feature index remapping
        self.feat_map_fwd = {}
        self.feat_map_rev = {}

        self.K = camera.get_K(optimized)
        self.distCoeffs = np.array(camera.get_dist_coeffs(optimized))

        # assemble the initial camera estimates
        self.n_cameras = len(placed_images)
        self.camera_params = np.empty(self.n_cameras * self.ncp)
        for cam_idx, global_index in enumerate(placed_images):
            image = proj.image_list[global_index]
            if self.cam_method == 'rvec_tvec':
                rvec, tvec = image.get_proj(optimized)
                self.camera_params[cam_idx*self.ncp:cam_idx*self.ncp+self.ncp] = np.append(rvec, tvec)
            elif self.cam_method == 'ned_quat':
                ned, ypr, quat = image.get_camera_pose(optimized)
                self.camera_params[cam_idx*self.ncp:cam_idx*self.ncp+self.ncp] = np.append(ned, quat)

        # count number of 3d points and observations
        self.n_points = 0
        n_observations = 0
        for i, match in enumerate(matches_list):
            # count the number of referenced observations
            if match[1] == group_index: # used by the current group
                count = 0
                for m in match[2:]:
                    if m[0] in placed_images:
                        count += 1
                if count >= self.min_chain_len:
                    n_observations += count
                    self.n_points += 1

        # assemble 3d point estimates and build indexing maps
        self.points_3d = np.empty(self.n_points * 3)
        point_idx = 0
        feat_used = 0
        for i, match in enumerate(matches_list):
            if match[1] == group_index: # used by the current group
                count = 0
                for m in match[2:]:
                    if m[0] in placed_images:
                        count += 1
                if count >= self.min_chain_len:
                    self.feat_map_fwd[i] = feat_used
                    self.feat_map_rev[feat_used] = i
                    feat_used += 1
                    ned = np.array(match[0])
                    if np.any(np.isnan(ned)):
                        print(i, ned)
                    self.points_3d[point_idx] = ned[0]
                    self.points_3d[point_idx+1] = ned[1]
                    self.points_3d[point_idx+2] = ned[2]
                    point_idx += 3

        # assemble observations (image index, feature index, u, v)
        self.by_camera_point_indices = [ [] for i in range(self.n_cameras) ]
        self.by_camera_points_2d = [ [] for i in range(self.n_cameras) ]
        #print('by_camera:', by_camera)
        #points_2d = np.empty((n_observations, 2))
        #obs_idx = 0
        for i, match in enumerate(matches_list):
            if match[1] == group_index: # used by the current group
                count = 0
                for m in match[2:]:
                    if m[0] in placed_images:
                        count += 1
                if count >= self.min_chain_len:
                    for m in match[2:]:
                        if m[0] in placed_images:
                            cam_index = self.camera_map_rev[m[0]]
                            feat_index = self.feat_map_fwd[i]
                            kp = m[1] # orig/distorted
                            #kp = proj.image_list[m[0]].uv_list[m[1]] # undistorted
                            self.by_camera_point_indices[cam_index].append(feat_index)
                            self.by_camera_points_2d[cam_index].append(kp)

        # convert to numpy native structures
        for i in range(self.n_cameras):
            size = len(self.by_camera_point_indices[i])
            self.by_camera_point_indices[i] = np.array(self.by_camera_point_indices[i])
            self.by_camera_points_2d[i] = np.asarray([self.by_camera_points_2d[i]]).reshape(size, 1, 2)

        # generate the camera and point indices (for mapping the
        # sparse jacobian entries which define which observations
        # depend on which parameters.)
        self.camera_indices = np.empty(n_observations, dtype=int)
        self.point_indices = np.empty(n_observations, dtype=int)
        obs_idx = 0
        for i in range(self.n_cameras):
            for j in range(len(self.by_camera_point_indices[i])):
```

```
                self.camera_indices[obs_idx] = i
                self.point_indices[obs_idx] = self.by_camera_point_indices[i][j]
                obs_idx += 1
        log("num observations:", obs_idx)

    # assemble the structures and remapping indices required for
    # optimizing a group of images/features, call the optimizer, and
    # save the result.
    def process_start(self):
        if self.optimize_calib == 'global':
            x0 = np.hstack((self.camera_params.ravel(), self.points_3d.ravel(),
                            self.K[0,0], self.K[0,2], self.K[1,2],
                            self.distCoeffs))
        else:
            x0 = np.hstack((self.camera_params.ravel(), self.points_3d.ravel()))
        f0 = self.fun(x0, self.n_cameras, self.n_points,
                      self.by_camera_point_indices, self.by_camera_points_2d)
        mre_start = np.mean(np.abs(f0))

        A = self.bundle_adjustment_sparsity(self.n_cameras, self.n_points,
                                            self.camera_indices,
                                            self.point_indices)

        if self.with_bounds:
            # quick test of bounds ... allow camera parameters to go free,
            # but limit 3d points =to +/- 100m of initial guess
            lower = []
            upper = []
            tol = 100.0
            for i in range(self.n_cameras):
                # unlimit the camera params
                for j in range(self.ncp):
                    if self.cam_method == 'ned_quat' and j < 3:
                        # bound the position of the camera to +/- 3
                        # meters of reported position
                        lower.append( self.camera_params[i*self.ncp + j] - 3 )
                        upper.append( self.camera_params[i*self.ncp + j] + 3 )
                    else:
                        lower.append( -np.inf )
                        upper.append( np.inf )
            for i in range(self.n_points * 3):
                #lower.append( points_3d[i] - tol )
                #upper.append( points_3d[i] + tol )
                # let point locations float without constraint
                lower.append( -np.inf )
                upper.append( np.inf )
            if self.optimize_calib == 'global':
                #tol = 0.0000001
                tol = 0.2
                # bound focal length
                lower.append(self.K[0,0]*(1-tol))
                upper.append(self.K[0,0]*(1+tol))
                #lower.append(self.K[0,0]*0.9)
                #upper.append(self.K[0,0]*1.1)
                cu = self.K[0,2]
                cv = self.K[1,2]
                lower.append(cu*(1-tol))
                upper.append(cu*(1+tol))
                lower.append(cv*(1-tol))
                upper.append(cv*(1+tol))
                # unlimit radial distortion params, limit tangential
                # params (5 parameters)
                lower.append( -np.inf )
                upper.append( np.inf )
                lower.append( -np.inf )
                upper.append( np.inf )
                lower.append( -tol )
                upper.append( tol )
                lower.append( -tol )
                upper.append( tol )
                lower.append( -np.inf )
                upper.append( np.inf )
            bounds = [lower, upper]
        else:
            bounds = (-np.inf, np.inf)
        # plt.figure(figsize=(16,9))
        # plt.ion()
        # mypts = self.points_3d.reshape((self.n_points, 3))
        # self.graph = plt.scatter(mypts[:,1], mypts[:,0], 100, -mypts[:,2], cmap=cm.jet)
        # plt.colorbar()
        # plt.draw()
        # plt.pause(0.01)

        t0 = time.time()
        # bounds=bounds,
        res = least_squares(self.fun, x0,
                            jac_sparsity=A,
                            verbose=2,
                            method='trf',
                            loss='linear',
                            ftol=self.ftol,
                            x_scale='jac',
                            bounds=bounds,
                            args=(self.n_cameras, self.n_points,
                                  self.by_camera_point_indices,
                                  self.by_camera_points_2d))
        t1 = time.time()
        log("Optimization took %.1f seconds" % (t1 - t0))
        # print(res['x'])
        log("res:", res)

        self.camera_params = res.x[:self.n_cameras * self.ncp].reshape((self.n_cameras, self.ncp))
        self.points_3d = res.x[self.n_cameras * self.ncp:self.n_cameras * self.ncp + self.n_points * 3].reshape((self.n_points, 3))
        if self.optimize_calib == 'global':
            camera_calib = res.x[self.n_cameras * self.ncp + self.n_points * 3:]
            fx = camera_calib[0]
            fy = camera_calib[0]
            cu = camera_calib[1]
            cv = camera_calib[2]
            distCoeffs_opt = camera_calib[3:]
        else:
            fx = self.K[0,0]
            fy = self.K[1,1]
            cu = self.K[0,2]
            cv = self.K[1,2]
            distCoeffs_opt = self.distCoeffs

        mre_final = np.mean(np.abs(res.fun))
```

```
        iterations = res.njev
        time_sec = t1 - t0

        log("Starting mean reprojection error: %.2f" % mre_start)
        log("Final mean reprojection error: %.2f" % mre_final)
        log("Iterations:", iterations)
        log("Elapsed time = %.1f sec" % time_sec)
        if self.optimize_calib == 'global':
            log("Final camera calib:\n", camera_calib)

        # final plot
        # plt.plot(res.fun)
        # plt.ioff()
        # plt.show()

        return ( self.camera_params, self.points_3d,
                 self.camera_map_fwd, self.feat_map_rev,
                 fx, fy, cu, cv, distCoeffs_opt )

    def optmizeed_poses_camera(self, proj):
        log('Updated the optimized camera poses.')

        # mark all the optimized poses as invalid
        for image in proj.image_list:
            opt_cam_node = image.node.getChild('camera_pose_opt', True)
            opt_cam_node.setBool('valid', False)

        for i, cam in enumerate(self.camera_params):
            image_index = self.camera_map_fwd[i]
            image = proj.image_list[image_index]
            ned_orig, ypr_orig, quat_orig = image.get_camera_pose()
            # print('optimized cam:', cam)
            if self.cam_method == 'rvec_tvec':
                rvec = cam[0:3]
                tvec = cam[3:6]
                Rned2cam, jac = cv2.Rodrigues(rvec)
                cam2body = image.get_cam2body()
                Rned2body = cam2body.dot(Rned2cam)
                Rbody2ned = np.matrix(Rned2body).T
                (yaw_rad, pitch_rad, roll_rad) = transformations.euler_from_matrix(Rbody2ned, 'rzyx')
                #print "orig ypr =", image.camera_pose['ypr']
                #print "new ypr =", [yaw/d2r, pitch/d2r, roll/d2r]
                pos = -np.matrix(Rned2cam).T * np.matrix(tvec).T
                ned = pos.T[0].tolist()[0]
            elif self.cam_method == 'ned_quat':
                ned = cam[0:3]
                quat = cam[3:7]
                (yaw_rad, pitch_rad, roll_rad) = transformations.euler_from_quaternion(quat, "rzyx")
            log(image.name, ned_orig, '->', ned, 'dist:', np.linalg.norm(np.array(ned_orig) - np.array(ned)))
            image.set_camera_pose( ned, yaw_rad*r2d, pitch_rad*r2d, roll_rad*r2d, opt=True )
            image.placed = True
        proj.save_images_info()

    # compare original camera locations with optimized camera
    # locations and derive a transform matrix to 'best fit' the new
    # camera locations over the original ... trusting the original
    # group gps solution as our best absolute truth for positioning
    # the system in world coordinates.  (each separately optimized
    # group needs a separate/unique fit)
    def re_project_optm(self, proj, matches, groups, group_index):
        matches_opt = list(matches) # shallow copy
        group = groups[group_index]
        log('refitting group size:', len(group))
        src_list = []
        dst_list = []
        # only consider images that are in the current   group
        for name in group:
            image = proj.findImageByName(name)
            ned, ypr, quat = image.get_camera_pose(opt=True)
            src_list.append(ned)
            ned, ypr, quat = image.get_camera_pose()
            dst_list.append(ned)
        A = get_recenter_affine(src_list, dst_list)

        # extract the rotation matrix (R) from the affine transform
        scale, shear, angles, trans, persp = transformations.decompose_matrix(A)
        log('  scale:', scale)
        log('  shear:', shear)
        log('  angles:', angles)
        log('  translate:', trans)
        log('  perspective:', persp)
        R = transformations.euler_matrix(*angles)
        log("R:\n{}".format(R))

        # fixme (just group):

        # update the optimized camera locations based on best fit
        camera_list = []
        # load optimized poses
        for image in proj.image_list:
            if image.name in group:
                ned, ypr, quat = image.get_camera_pose(opt=True)
            else:
                # this is just fodder to match size/index of the lists
                ned, ypr, quat = image.get_camera_pose()
            camera_list.append( ned )

        # refit
        new_cams = transform_points(A, camera_list)

        # update position
        for i, image in enumerate(proj.image_list):
            if not image.name in group:
                continue
            ned, [y, p, r], quat = image.get_camera_pose(opt=True)
            image.set_camera_pose(new_cams[i], y, p, r, opt=True)
        proj.save_images_info()

        if True:
            # update optimized pose orientation.
            dist_report = []
            for i, image in enumerate(proj.image_list):
                if not image.name in group:
                    continue
                ned_orig, ypr_orig, quat_orig = image.get_camera_pose()
                ned, ypr, quat = image.get_camera_pose(opt=True)
                Rbody2ned = image.get_body2ned(opt=True)
```

```
                # update the orientation with the same transform to keep
                # everything in proper consistent alignment

                newRbody2ned = R[:3,:3].dot(Rbody2ned)
                (yaw, pitch, roll) = transformations.euler_from_matrix(newRbody2ned, 'rzyx')
                image.set_camera_pose(new_cams[i], yaw*r2d, pitch*r2d, roll*r2d,
                                      opt=True)
                dist = np.linalg.norm( np.array(ned_orig) - np.array(new_cams[i]))
                qlog("image:", image.name)
                qlog("  orig pos:", ned_orig)
                qlog("  fit pos:", new_cams[i])
                qlog("  dist moved:", dist)
                dist_report.append( (dist, image.name) )
            proj.save_images_info()

            dist_report = sorted(dist_report,
                                 key=lambda fields: fields[0],
                                 reverse=False)
            log("Image movement sorted lowest to highest:")
            for report in dist_report:
                log(report[1], "dist:", report[0])

        # tranform the optimized point locations using the same best
        # fit transform for the camera locations.
        new_feats = transform_points(A, self.points_3d)

        # update any of the transformed feature locations that have
        # membership in the currently processing group back to the
        # master match structure.  Note we process groups in order of
        # little to big so if a match is in more than one group it
        # follows the larger group.
        for i, feat in enumerate(new_feats):
            match_index = self.feat_map_rev[i]
            match = matches_opt[match_index]
            in_group = False
            for m in match[2:]:
                if proj.image_list[m[0]].name in group:
                    in_group = True
                    break
            if in_group:
                #print(" before:", match)
                match[0] = feat
                #print(" after:", match)
```

```
def get_K(optimized=False):
    """
    Form the camera calibration matrix K using 5 parameters of
    Finite Projective Camera model.  (Note skew parameter is 0)
    See Eqn (6.10) in:
    R.I. Hartley & A. Zisserman, Multiview Geometry in Computer Vision,
    Cambridge University Press, 2004.
    """
    tmp = []
    if optimized and camera_node.hasChild('K_opt'):
        for i in range(9):
            tmp.append( camera_node.getFloatEnum('K_opt', i) )
    else:
        for i in range(9):
            tmp.append( camera_node.getFloatEnum('K', i) )
    K = np.copy(np.array(tmp)).reshape(3,3)
    return K

def set_K(fx, fy, cu, cv, optimized=False):
    K = np.identity(3)
    K[0,0] = fx
    K[1,1] = fy
    K[0,2] = cu
    K[1,2] = cv
    # store as linear python list
    tmp = K.ravel().tolist()
    if optimized:
        camera_node.setLen('K_opt', 9)
        for i in range(9):
            camera_node.setFloatEnum('K_opt', i, tmp[i])
    else:
        camera_node.setLen('K', 9)
        for i in range(9):
            camera_node.setFloatEnum('K', i, tmp[i])

# dist_coeffs = array[5] = k1, k2, p1, p2, k3
def get_dist_coeffs(optimized=False):
    tmp = []
    if optimized and camera_node.hasChild('dist_coeffs_opt'):
        for i in range(5):
            tmp.append( camera_node.getFloatEnum('dist_coeffs_opt', i) )
    else:
        for i in range(5):
            tmp.append( camera_node.getFloatEnum('dist_coeffs', i) )
    return np.array(tmp)

def set_dist_coeffs(dist_coeffs, optimized=False):
    if optimized:
        camera_node.setLen('dist_coeffs_opt', 5)
        for i in range(5):
            camera_node.setFloatEnum('dist_coeffs_opt', i, dist_coeffs[i])
    else:
        camera_node.setLen('dist_coeffs', 5)
        for i in range(5):
            camera_node.setFloatEnum('dist_coeffs', i, dist_coeffs[i])

def set_image_params(width_px, height_px):
    camera_node.setInt('width_px', width_px)
    camera_node.setInt('height_px', height_px)

def get_image_params():
    return ( camera_node.getInt('width_px'),
             camera_node.getInt('height_px') )

def set_mount_params(yaw_deg, pitch_deg, roll_deg):
    mount_node = camera_node.getChild('mount', True)
    mount_node.setFloat('yaw_deg', yaw_deg)
    mount_node.setFloat('pitch_deg', pitch_deg)
    mount_node.setFloat('roll_deg', roll_deg)
    #camera_node.pretty_print()

def get_mount_params():
    mount_node = camera_node.getChild('mount', True)
    return [ mount_node.getFloat('yaw_deg'),
```

```python
            mount_node.getFloat('pitch_deg'),
            mount_node.getFloat('roll_deg') ]

def get_body2cam():
    yaw_deg, pitch_deg, roll_deg = get_mount_params()
    body2cam = transformations.quaternion_from_euler(yaw_deg * d2r,
                                                     pitch_deg * d2r,
                                                     roll_deg * d2r,
                                                     "rzyx")
    return body2cam
```

```python
def translation_matrix(direction):

    M = numpy.identity(4)
    M[:3, 3] = direction[:3]
    return M


def translation_from_matrix(matrix):

    return numpy.array(matrix, copy=False)[:3, 3].copy()


def reflection_matrix(point, normal):

    normal = unit_vector(normal[:3])
    M = numpy.identity(4)
    M[:3, :3] -= 2.0 * numpy.outer(normal, normal)
    M[:3, 3] = (2.0 * numpy.dot(point[:3], normal)) * normal
    return M


def reflection_from_matrix(matrix):

    M = numpy.array(matrix, dtype=numpy.float64, copy=False)
    # normal: unit eigenvector corresponding to eigenvalue -1
    w, V = numpy.linalg.eig(M[:3, :3])
    i = numpy.where(abs(numpy.real(w) + 1.0) < 1e-8)[0]
    if not len(i):
        raise ValueError("no unit eigenvector corresponding to eigenvalue -1")
    normal = numpy.real(V[:, i[0]]).squeeze()
    # point: any unit eigenvector corresponding to eigenvalue 1
    w, V = numpy.linalg.eig(M)
    i = numpy.where(abs(numpy.real(w) - 1.0) < 1e-8)[0]
    if not len(i):
        raise ValueError("no unit eigenvector corresponding to eigenvalue 1")
    point = numpy.real(V[:, i[-1]]).squeeze()
    point /= point[3]
    return point, normal


def rotation_matrix(angle, direction, point=None):

    sina = math.sin(angle)
    cosa = math.cos(angle)
    direction = unit_vector(direction[:3])
    # rotation matrix around unit vector
    R = numpy.diag([cosa, cosa, cosa])
    R += numpy.outer(direction, direction) * (1.0 - cosa)
    direction *= sina
    R += numpy.array([[ 0.0,         -direction[2],  direction[1]],
                      [ direction[2], 0.0,          -direction[0]],
                      [-direction[1], direction[0],  0.0]])
    M = numpy.identity(4)
    M[:3, :3] = R
    if point is not None:
        # rotation not around origin
        point = numpy.array(point[:3], dtype=numpy.float64, copy=False)
        M[:3, 3] = point - numpy.dot(R, point)
    return M


def rotation_from_matrix(matrix):

    R = numpy.array(matrix, dtype=numpy.float64, copy=False)
    R33 = R[:3, :3]
    # direction: unit eigenvector of R33 corresponding to eigenvalue of 1
    w, W = numpy.linalg.eig(R33.T)
    i = numpy.where(abs(numpy.real(w) - 1.0) < 1e-8)[0]
    if not len(i):
        raise ValueError("no unit eigenvector corresponding to eigenvalue 1")
    direction = numpy.real(W[:, i[-1]]).squeeze()
    # point: unit eigenvector of R33 corresponding to eigenvalue of 1
    w, Q = numpy.linalg.eig(R)
    i = numpy.where(abs(numpy.real(w) - 1.0) < 1e-8)[0]
    if not len(i):
        raise ValueError("no unit eigenvector corresponding to eigenvalue 1")
    point = numpy.real(Q[:, i[-1]]).squeeze()
    point /= point[3]
    # rotation angle depending on direction
    cosa = (numpy.trace(R33) - 1.0) / 2.0
    if abs(direction[2]) > 1e-8:
        sina = (R[1, 0] + (cosa-1.0)*direction[0]*direction[1]) / direction[2]
    elif abs(direction[1]) > 1e-8:
        sina = (R[0, 2] + (cosa-1.0)*direction[0]*direction[2]) / direction[1]
    else:
        sina = (R[2, 1] + (cosa-1.0)*direction[1]*direction[2]) / direction[0]
    angle = math.atan2(sina, cosa)
    return angle, direction, point


def scale_matrix(factor, origin=None, direction=None):

    if direction is None:
        # uniform scaling
        M = numpy.diag([factor, factor, factor, 1.0])
        if origin is not None:
            M[:3, 3] = origin[:3]
            M[:3, 3] *= 1.0 - factor
    else:
        # nonuniform scaling
        direction = unit_vector(direction[:3])
        factor = 1.0 - factor
        M = numpy.identity(4)
        M[:3, :3] -= factor * numpy.outer(direction, direction)
        if origin is not None:
            M[:3, 3] = (factor * numpy.dot(origin[:3], direction)) * direction
```

```
    return M


def scale_from_matrix(matrix):

    M = numpy.array(matrix, dtype=numpy.float64, copy=False)
    M33 = M[:3, :3]
    factor = numpy.trace(M33) - 2.0
    try:
        # direction: unit eigenvector corresponding to eigenvalue factor
        w, V = numpy.linalg.eig(M33)
        i = numpy.where(abs(numpy.real(w) - factor) < 1e-8)[0][0]
        direction = numpy.real(V[:, i]).squeeze()
        direction /= vector_norm(direction)
    except IndexError:
        # uniform scaling
        factor = (factor + 2.0) / 3.0
        direction = None
    # origin: any eigenvector corresponding to eigenvalue 1
    w, V = numpy.linalg.eig(M)
    i = numpy.where(abs(numpy.real(w) - 1.0) < 1e-8)[0]
    if not len(i):
        raise ValueError("no eigenvector corresponding to eigenvalue 1")
    origin = numpy.real(V[:, i[-1]]).squeeze()
    origin /= origin[3]
    return factor, origin, direction


def projection_matrix(point, normal, direction=None,
                      perspective=None, pseudo=False):

    M = numpy.identity(4)
    point = numpy.array(point[:3], dtype=numpy.float64, copy=False)
    normal = unit_vector(normal[:3])
    if perspective is not None:
        # perspective projection
        perspective = numpy.array(perspective[:3], dtype=numpy.float64,
                                  copy=False)
        M[0, 0] = M[1, 1] = M[2, 2] = numpy.dot(perspective-point, normal)
        M[:3, :3] -= numpy.outer(perspective, normal)
        if pseudo:
            # preserve relative depth
            M[:3, :3] -= numpy.outer(normal, normal)
            M[:3, 3] = numpy.dot(point, normal) * (perspective+normal)
        else:
            M[:3, 3] = numpy.dot(point, normal) * perspective
        M[3, :3] = -normal
        M[3, 3] = numpy.dot(perspective, normal)
    elif direction is not None:
        # parallel projection
        direction = numpy.array(direction[:3], dtype=numpy.float64, copy=False)
        scale = numpy.dot(direction, normal)
        M[:3, :3] -= numpy.outer(direction, normal) / scale
        M[:3, 3] = direction * (numpy.dot(point, normal) / scale)
    else:
        # orthogonal projection
        M[:3, :3] -= numpy.outer(normal, normal)
        M[:3, 3] = numpy.dot(point, normal) * normal
    return M


def projection_from_matrix(matrix, pseudo=False):

    M = numpy.array(matrix, dtype=numpy.float64, copy=False)
    M33 = M[:3, :3]
    w, V = numpy.linalg.eig(M)
    i = numpy.where(abs(numpy.real(w) - 1.0) < 1e-8)[0]
    if not pseudo and len(i):
        # point: any eigenvector corresponding to eigenvalue 1
        point = numpy.real(V[:, i[-1]]).squeeze()
        point /= point[3]
        # direction: unit eigenvector corresponding to eigenvalue 0
        w, V = numpy.linalg.eig(M33)
        i = numpy.where(abs(numpy.real(w)) < 1e-8)[0]
        if not len(i):
            raise ValueError("no eigenvector corresponding to eigenvalue 0")
        direction = numpy.real(V[:, i[0]]).squeeze()
        direction /= vector_norm(direction)
        # normal: unit eigenvector of M33.T corresponding to eigenvalue 0
        w, V = numpy.linalg.eig(M33.T)
        i = numpy.where(abs(numpy.real(w)) < 1e-8)[0]
        if len(i):
            # parallel projection
            normal = numpy.real(V[:, i[0]]).squeeze()
            normal /= vector_norm(normal)
            return point, normal, direction, None, False
        else:
            # orthogonal projection, where normal equals direction vector
            return point, direction, None, None, False
    else:
        # perspective projection
        i = numpy.where(abs(numpy.real(w)) > 1e-8)[0]
        if not len(i):
            raise ValueError(
                "no eigenvector not corresponding to eigenvalue 0")
        point = numpy.real(V[:, i[-1]]).squeeze()
        point /= point[3]
        normal = - M[3, :3]
        perspective = M[:3, 3] / numpy.dot(point[:3], normal)
        if pseudo:
            perspective -= normal
        return point, normal, None, perspective, pseudo


def clip_matrix(left, right, bottom, top, near, far, perspective=False):

    if left >= right or bottom >= top or near >= far:
        raise ValueError("invalid frustum")
    if perspective:
        if near <= _EPS:
            raise ValueError("invalid frustum: near <= 0")
        t = 2.0 * near
        M = [[t/(left-right), 0.0, (right+left)/(right-left), 0.0],
             [0.0, t/(bottom-top), (top+bottom)/(top-bottom), 0.0],
             [0.0, 0.0, (far+near)/(near-far), t*far/(far-near)],
             [0.0, 0.0, -1.0, 0.0]]
    else:
        M = [[2.0/(right-left), 0.0, 0.0, (right+left)/(left-right)],
```

```
            [0.0, 2.0/(top-bottom), 0.0, (top+bottom)/(bottom-top)],
            [0.0, 0.0, 2.0/(far-near), (far+near)/(near-far)],
            [0.0, 0.0, 0.0, 1.0]]
    return numpy.array(M)


def shear_matrix(angle, direction, point, normal):

    normal = unit_vector(normal[:3])
    direction = unit_vector(direction[:3])
    if abs(numpy.dot(normal, direction)) > 1e-6:
        raise ValueError("direction and normal vectors are not orthogonal")
    angle = math.tan(angle)
    M = numpy.identity(4)
    M[:3, :3] += angle * numpy.outer(direction, normal)
    M[:3, 3] = -angle * numpy.dot(point[:3], normal) * direction
    return M


def shear_from_matrix(matrix):

    M = numpy.array(matrix, dtype=numpy.float64, copy=False)
    M33 = M[:3, :3]
    # normal: cross independent eigenvectors corresponding to the eigenvalue 1
    w, V = numpy.linalg.eig(M33)
    i = numpy.where(abs(numpy.real(w) - 1.0) < 1e-4)[0]
    if len(i) < 2:
        raise ValueError("no two linear independent eigenvectors found %s" % w)
    V = numpy.real(V[:, i]).squeeze().T
    lenorm = -1.0
    for i0, i1 in ((0, 1), (0, 2), (1, 2)):
        n = numpy.cross(V[i0], V[i1])
        w = vector_norm(n)
        if w > lenorm:
            lenorm = w
            normal = n
    normal /= lenorm
    # direction and angle
    direction = numpy.dot(M33 - numpy.identity(3), normal)
    angle = vector_norm(direction)
    direction /= angle
    angle = math.atan(angle)
    # point: eigenvector corresponding to eigenvalue 1
    w, V = numpy.linalg.eig(M)
    i = numpy.where(abs(numpy.real(w) - 1.0) < 1e-8)[0]
    if not len(i):
        raise ValueError("no eigenvector corresponding to eigenvalue 1")
    point = numpy.real(V[:, i[-1]]).squeeze()
    point /= point[3]
    return angle, direction, point, normal


def decompose_matrix(matrix):

    M = numpy.array(matrix, dtype=numpy.float64, copy=True).T
    if abs(M[3, 3]) < _EPS:
        raise ValueError("M[3, 3] is zero")
    M /= M[3, 3]
    P = M.copy()
    P[:, 3] = 0.0, 0.0, 0.0, 1.0
    if not numpy.linalg.det(P):
        raise ValueError("matrix is singular")

    scale = numpy.zeros((3, ))
    shear = [0.0, 0.0, 0.0]
    angles = [0.0, 0.0, 0.0]

    if any(abs(M[:3, 3]) > _EPS):
        perspective = numpy.dot(M[:, 3], numpy.linalg.inv(P.T))
        M[:, 3] = 0.0, 0.0, 0.0, 1.0
    else:
        perspective = numpy.array([0.0, 0.0, 0.0, 1.0])

    translate = M[3, :3].copy()
    M[3, :3] = 0.0

    row = M[:3, :3].copy()
    scale[0] = vector_norm(row[0])
    row[0] /= scale[0]
    shear[0] = numpy.dot(row[0], row[1])
    row[1] -= row[0] * shear[0]
    scale[1] = vector_norm(row[1])
    row[1] /= scale[1]
    shear[0] /= scale[1]
    shear[1] = numpy.dot(row[0], row[2])
    row[2] -= row[0] * shear[1]
    shear[2] = numpy.dot(row[1], row[2])
    row[2] -= row[1] * shear[2]
    scale[2] = vector_norm(row[2])
    row[2] /= scale[2]
    shear[1:] /= scale[2]

    if numpy.dot(row[0], numpy.cross(row[1], row[2])) < 0:
        numpy.negative(scale, scale)
        numpy.negative(row, row)

    angles[1] = math.asin(-row[0, 2])
    if math.cos(angles[1]):
        angles[0] = math.atan2(row[1, 2], row[2, 2])
        angles[2] = math.atan2(row[0, 1], row[0, 0])
    else:
        #angles[0] = math.atan2(row[1, 0], row[1, 1])
        angles[0] = math.atan2(-row[2, 1], row[1, 1])
        angles[2] = 0.0

    return scale, shear, angles, translate, perspective


def compose_matrix(scale=None, shear=None, angles=None, translate=None,
                   perspective=None):

    M = numpy.identity(4)
    if perspective is not None:
        P = numpy.identity(4)
        P[3, :] = perspective[:4]
        M = numpy.dot(M, P)
    if translate is not None:
        T = numpy.identity(4)
```

```python
        T[:3, 3] = translate[:3]
        M = numpy.dot(M, T)
    if angles is not None:
        R = euler_matrix(angles[0], angles[1], angles[2], 'sxyz')
        M = numpy.dot(M, R)
    if shear is not None:
        Z = numpy.identity(4)
        Z[1, 2] = shear[2]
        Z[0, 2] = shear[1]
        Z[0, 1] = shear[0]
        M = numpy.dot(M, Z)
    if scale is not None:
        S = numpy.identity(4)
        S[0, 0] = scale[0]
        S[1, 1] = scale[1]
        S[2, 2] = scale[2]
        M = numpy.dot(M, S)
    M /= M[3, 3]
    return M


def orthogonalization_matrix(lengths, angles):

    a, b, c = lengths
    angles = numpy.radians(angles)
    sina, sinb, _ = numpy.sin(angles)
    cosa, cosb, cosg = numpy.cos(angles)
    co = (cosa * cosb - cosg) / (sina * sinb)
    return numpy.array([
        [ a*sinb*math.sqrt(1.0-co*co),  0.0,    0.0, 0.0],
        [-a*sinb*co,                    b*sina, 0.0, 0.0],
        [ a*cosb,                       b*cosa, c,   0.0],
        [ 0.0,                          0.0,    0.0, 1.0]])


def affine_matrix_from_points(v0, v1, shear=True, scale=True, usesvd=True):

    v0 = numpy.array(v0, dtype=numpy.float64, copy=True)
    v1 = numpy.array(v1, dtype=numpy.float64, copy=True)

    #print( "v0.shape = %s" % str(v0.shape))
    #print( "v1.shape = %s" % str(v1.shape))
    #print( "v0.shape[1] = %s" % str(v0.shape[1]))
    ndims = v0.shape[0]
    if ndims < 2 or v0.shape[1] < ndims or v0.shape != v1.shape:
        raise ValueError("input arrays are of wrong shape or type")

    # move centroids to origin
    t0 = -numpy.mean(v0, axis=1)
    M0 = numpy.identity(ndims+1)
    M0[:ndims, ndims] = t0
    v0 += t0.reshape(ndims, 1)
    t1 = -numpy.mean(v1, axis=1)
    M1 = numpy.identity(ndims+1)
    M1[:ndims, ndims] = t1
    v1 += t1.reshape(ndims, 1)

    if shear:
        # Affine transformation
        A = numpy.concatenate((v0, v1), axis=0)
        u, s, vh = numpy.linalg.svd(A.T)
        vh = vh[:ndims].T
        B = vh[:ndims]
        C = vh[ndims:2*ndims]
        t = numpy.dot(C, numpy.linalg.pinv(B))
        t = numpy.concatenate((t, numpy.zeros((ndims, 1))), axis=1)
        M = numpy.vstack((t, ((0.0,)*ndims) + (1.0,)))
    elif usesvd or ndims != 3:
        # Rigid transformation via SVD of covariance matrix
        u, s, vh = numpy.linalg.svd(numpy.dot(v1, v0.T))
        # rotation matrix from SVD orthonormal bases
        R = numpy.dot(u, vh)
        if numpy.linalg.det(R) < 0.0:
            # R does not constitute right handed system
            R -= numpy.outer(u[:, ndims-1], vh[ndims-1, :]*2.0)
            s[-1] *= -1.0
        # homogeneous transformation matrix
        M = numpy.identity(ndims+1)
        M[:ndims, :ndims] = R
    else:
        # Rigid transformation matrix via quaternion
        # compute symmetric matrix N
        xx, yy, zz = numpy.sum(v0 * v1, axis=1)
        xy, yz, zx = numpy.sum(v0 * numpy.roll(v1, -1, axis=0), axis=1)
        xz, yx, zy = numpy.sum(v0 * numpy.roll(v1, -2, axis=0), axis=1)
        N = [[xx+yy+zz, 0.0,      0.0,      0.0],
             [yz-zy,    xx-yy-zz, 0.0,      0.0],
             [zx-xz,    xy+yx,    yy-xx-zz, 0.0],
             [xy-yx,    zx+xz,    yz+zy,    zz-xx-yy]]
        # quaternion: eigenvector corresponding to most positive eigenvalue
        w, V = numpy.linalg.eigh(N)
        q = V[:, numpy.argmax(w)]
        q /= vector_norm(q)  # unit quaternion
        # homogeneous transformation matrix
        M = quaternion_matrix(q)

    if scale and not shear:
        # Affine transformation; scale is ratio of RMS deviations from centroid
        v0 *= v0
        v1 *= v1
        M[:ndims, :ndims] *= math.sqrt(numpy.sum(v1) / numpy.sum(v0))

    # move centroids back
    M = numpy.dot(numpy.linalg.inv(M1), numpy.dot(M, M0))
    M /= M[ndims, ndims]
    return M


def affine_matrix_from_points_weighted(v0, v1, weights, shear=True, scale=True, usesvd=True):

    v0 = numpy.array(v0, dtype=numpy.float64, copy=True)
    v1 = numpy.array(v1, dtype=numpy.float64, copy=True)

    #print( "v0.shape = %s" % str(v0.shape))
    #print( "v1.shape = %s" % str(v1.shape))
    #print( "v0.shape[1] = %s" % str(v0.shape[1]))
    ndims = v0.shape[0]
    if ndims < 2 or v0.shape[1] < ndims or v0.shape != v1.shape:
```

```
    if ndims < 2 or v0.shape[1] < ndims or v0.shape != v1.shape:
        raise ValueError("input arrays are of wrong shape or type")

    # move centroids to (weighted) origin
    t0_sum = numpy.zeros(ndims)
    w_sum = 0.0
    for i in range(v0.shape[1]):
        t0_sum += (v0[:,i] * weights[i])
        w_sum += weights[i]
    #t0 = -numpy.mean(v0, axis=1)
    #print("t0 orig =", t0)
    t0 = -t0_sum / w_sum
    #print("t0 weighted =", t0)
    M0 = numpy.identity(ndims+1)
    M0[:ndims, ndims] = t0
    v0 += t0.reshape(ndims, 1)

    t1_sum = numpy.zeros(ndims)
    w_sum = 0.0
    for i in range(v1.shape[1]):
        t1_sum += (v1[:,i] * weights[i])
        w_sum += weights[i]
    #t1 = -numpy.mean(v1, axis=1)
    #print("t1 orig =", t1)
    t1 = -t1_sum / w_sum
    #print("t1 weighted =", t1)
    M1 = numpy.identity(ndims+1)
    M1[:ndims, ndims] = t1
    v1 += t1.reshape(ndims, 1)

    if shear:
        # Affine transformation
        A = numpy.concatenate((v0, v1), axis=0)
        u, s, vh = numpy.linalg.svd(A.T)
        vh = vh[:ndims].T
        B = vh[:ndims]
        C = vh[ndims:2*ndims]
        t = numpy.dot(C, numpy.linalg.pinv(B))
        t = numpy.concatenate((t, numpy.zeros((ndims, 1))), axis=1)
        M = numpy.vstack((t, ((0.0,)*ndims) + (1.0,)))
    elif usesvd or ndims != 3:
        # Rigid transformation via SVD of covariance matrix
        #print( "numpy.dot()", numpy.dot(v1, v0.T) )
        #print( "v0.shape = %s" % str(v0.shape))
        #print( "v1.shape = %s" % str(v1.shape))
        dotsum = numpy.zeros( (ndims, ndims) )
        #print("dotsum shape =", dotsum.shape)
        for k in range(v0.shape[1]):
            for i in range(ndims):
                for j in range(ndims):
                    dotsum[j,i] += ( v0[i,k] * v1[j,k] ) * weights[k]
        #print( "sum M =", dotsum)

        #u, s, vh = numpy.linalg.svd(numpy.dot(v1, v0.T))
        u, s, vh = numpy.linalg.svd(dotsum)

        # rotation matrix from SVD orthonormal bases
        R = numpy.dot(u, vh)
        if numpy.linalg.det(R) < 0.0:
            # R does not constitute right handed system
            R -= numpy.outer(u[:, ndims-1], vh[ndims-1, :]*2.0)
            s[-1] *= -1.0
        # homogeneous transformation matrix
        M = numpy.identity(ndims+1)
        M[:ndims, :ndims] = R
    else:
        print("NOT WEIGHTED YET")
        # Rigid transformation matrix via quaternion
        # compute symmetric matrix N
        xx, yy, zz = numpy.sum(v0 * v1, axis=1)
        xy, yz, zx = numpy.sum(v0 * numpy.roll(v1, -1, axis=0), axis=1)
        xz, yx, zy = numpy.sum(v0 * numpy.roll(v1, -2, axis=0), axis=1)
        N = [[xx+yy+zz, 0.0,      0.0,      0.0],
             [yz-zy,    xx-yy-zz, 0.0,      0.0],
             [zx-xz,    xy+yx,    yy-xx-zz, 0.0],
             [xy-yx,    zx+xz,    yz+zy,    zz-xx-yy]]
        # quaternion: eigenvector corresponding to most positive eigenvalue
        w, V = numpy.linalg.eigh(N)
        q = V[:, numpy.argmax(w)]
        q /= vector_norm(q)  # unit quaternion
        # homogeneous transformation matrix
        M = quaternion_matrix(q)

    if scale and not shear:
        # Affine transformation; scale is ratio of RMS deviations from centroid
        v0 *= v0
        v1 *= v1
        M[:ndims, :ndims] *= math.sqrt(numpy.sum(v1) / numpy.sum(v0))

    # move centroids back
    M = numpy.dot(numpy.linalg.inv(M1), numpy.dot(M, M0))
    M /= M[ndims, ndims]
    return M


def superimposition_matrix(v0, v1, scale=False, usesvd=True):


    v0 = numpy.array(v0, dtype=numpy.float64, copy=False)[:3]
    v1 = numpy.array(v1, dtype=numpy.float64, copy=False)[:3]
    return affine_matrix_from_points(v0, v1, shear=False,
                                     scale=scale, usesvd=usesvd)


def euler_matrix(ai, aj, ak, axes='sxyz'):

    try:
        firstaxis, parity, repetition, frame = _AXES2TUPLE[axes]
    except (AttributeError, KeyError):
        _TUPLE2AXES[axes]  # validation
        firstaxis, parity, repetition, frame = axes

    i = firstaxis
    j = _NEXT_AXIS[i+parity]
    k = _NEXT_AXIS[i-parity+1]

    if frame:
        ai, ak = ak, ai
    if parity:
```

```
    if parity:
        ai, aj, ak = -ai, -aj, -ak

    si, sj, sk = math.sin(ai), math.sin(aj), math.sin(ak)
    ci, cj, ck = math.cos(ai), math.cos(aj), math.cos(ak)
    cc, cs = ci*ck, ci*sk
    sc, ss = si*ck, si*sk

    M = numpy.identity(4)
    if repetition:
        M[i, i] = cj
        M[i, j] = sj*si
        M[i, k] = sj*ci
        M[j, i] = sj*sk
        M[j, j] = -cj*ss+cc
        M[j, k] = -cj*cs-sc
        M[k, i] = -sj*ck
        M[k, j] = cj*sc+cs
        M[k, k] = cj*cc-ss
    else:
        M[i, i] = cj*ck
        M[i, j] = sj*sc-cs
        M[i, k] = sj*cc+ss
        M[j, i] = cj*sk
        M[j, j] = sj*ss+cc
        M[j, k] = sj*cs-sc
        M[k, i] = -sj
        M[k, j] = cj*si
        M[k, k] = cj*ci
    return M
```

SuperPoint + SuperGlue

```
!git clone https://github.com/magicleap/SuperPointPretrainedNetwork.git
```

```
weights_path = 'SuperPointPretrainedNetwork/superpoint_v1.pth'

cuda = 'True'
```

```
def to_kpts(pts, size=1):
  return [cv2.KeyPoint(pt[0], pt[1], size) for pt in pts]
```

```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F

torch.cuda.empty_cache()

class SuperPointNet(nn.Module):
    def __init__(self):
        super(SuperPointNet, self).__init__()
        self.relu = nn.ReLU(inplace=True)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        c1, c2, c3, c4, c5, d1 = 64, 64, 128, 128, 256, 256
        # Shared Encoder.
        self.conv1a = nn.Conv2d(1, c1, kernel_size=3, stride=1, padding=1)
        self.conv1b = nn.Conv2d(c1, c1, kernel_size=3, stride=1, padding=1)
        self.conv2a = nn.Conv2d(c1, c2, kernel_size=3, stride=1, padding=1)
        self.conv2b = nn.Conv2d(c2, c2, kernel_size=3, stride=1, padding=1)
        self.conv3a = nn.Conv2d(c2, c3, kernel_size=3, stride=1, padding=1)
        self.conv3b = nn.Conv2d(c3, c3, kernel_size=3, stride=1, padding=1)
        self.conv4a = nn.Conv2d(c3, c4, kernel_size=3, stride=1, padding=1)
        self.conv4b = nn.Conv2d(c4, c4, kernel_size=3, stride=1, padding=1)
        # Detector Head.
        self.convPa = nn.Conv2d(c4, c5, kernel_size=3, stride=1, padding=1)
        self.convPb = nn.Conv2d(c5, 65, kernel_size=1, stride=1, padding=0)
        # Descriptor Head.
        self.convDa = nn.Conv2d(c4, c5, kernel_size=3, stride=1, padding=1)
        self.convDb = nn.Conv2d(c5, d1, kernel_size=1, stride=1, padding=0)

    def forward(self, x):

        # Shared Encoder.
        x = self.relu(self.conv1a(x))
        x = self.relu(self.conv1b(x))
        x = self.pool(x)
        x = self.relu(self.conv2a(x))
        x = self.relu(self.conv2b(x))
        x = self.pool(x)
        x = self.relu(self.conv3a(x))
        x = self.relu(self.conv3b(x))
        x = self.pool(x)
        x = self.relu(self.conv4a(x))
        x = self.relu(self.conv4b(x))
        # Detector Head.
        cPa = self.relu(self.convPa(x))
        semi = self.convPb(cPa)
        # Descriptor Head.
        cDa = self.relu(self.convDa(x))
        desc = self.convDb(cDa)
        dn = torch.norm(desc, p=2, dim=1) # Compute the norm.
        desc = desc.div(torch.unsqueeze(dn, 1)) # Divide by norm to normalize.
        return semi, desc


class SuperPointFrontend(object):
    def __init__(self, weights_path, nms_dist, conf_thresh, nn_thresh,cuda=True):
        self.name = 'SuperPoint'
        self.cuda = cuda
        self.nms_dist = nms_dist
        self.conf_thresh = conf_thresh
        self.nn_thresh = nn_thresh # L2 descriptor distance for good match.
        self.cell = 8 # Size of each output cell. Keep this fixed.
        self.border_remove = 4 # Remove points this close to the border.

        # Load the network in inference mode.
        self.net = SuperPointNet()
        if cuda:
          # Train on GPU, deploy on GPU.
```

```
            self.net.load_state_dict(torch.load(weights_path))
            self.net = self.net.cuda()
        else:
          # Train on GPU, deploy on CPU.
            self.net.load_state_dict(torch.load(weights_path, map_location=lambda storage, loc: storage))
        self.net.eval()

    def nms_fast(self, in_corners, H, W, dist_thresh):

        grid = np.zeros((H, W)).astype(int) # Track NMS data.
        inds = np.zeros((H, W)).astype(int) # Store indices of points.
        # Sort by confidence and round to nearest int.
        inds1 = np.argsort(-in_corners[2,:])
        corners = in_corners[:,inds1]
        rcorners = corners[:2,:].round().astype(int) # Rounded corners.
        # Check for edge case of 0 or 1 corners.
        if rcorners.shape[1] == 0:
            return np.zeros((3,0)).astype(int), np.zeros(0).astype(int)
        if rcorners.shape[1] == 1:
            out = np.vstack((rcorners, in_corners[2])).reshape(3,1)
            return out, np.zeros((1)).astype(int)
        # Initialize the grid.
        for i, rc in enumerate(rcorners.T):
            grid[rcorners[1,i], rcorners[0,i]] = 1
            inds[rcorners[1,i], rcorners[0,i]] = i
        # Pad the border of the grid, so that we can NMS points near the border.
        pad = dist_thresh
        grid = np.pad(grid, ((pad,pad), (pad,pad)), mode='constant')
        # Iterate through points, highest to lowest conf, suppress neighborhood.
        count = 0
        for i, rc in enumerate(rcorners.T):
          # Account for top and left padding.
            pt = (rc[0]+pad, rc[1]+pad)
            if grid[pt[1], pt[0]] == 1: # If not yet suppressed.
                grid[pt[1]-pad:pt[1]+pad+1, pt[0]-pad:pt[0]+pad+1] = 0
                grid[pt[1], pt[0]] = -1
                count += 1
        # Get all surviving -1's and return sorted array of remaining corners.
        keepy, keepx = np.where(grid==-1)
        keepy, keepx = keepy - pad, keepx - pad
        inds_keep = inds[keepy, keepx]
        out = corners[:, inds_keep]
        values = out[-1, :]
        inds2 = np.argsort(-values)
        out = out[:, inds2]
        out_inds = inds1[inds_keep[inds2]]
        return out, out_inds

    def run(self, img):
        assert img.ndim == 2 #Image must be grayscale.
        assert img.dtype == np.float32 #Image must be float32.
        H, W = img.shape[0], img.shape[1]
        inp = img.copy()
        inp = (inp.reshape(1, H, W))
        inp = torch.from_numpy(inp)
        inp = torch.autograd.Variable(inp).view(1, 1, H, W)
        if self.cuda:
            inp = inp.cuda()
        # Forward pass of network.
        outs = self.net.forward(inp)
        semi, coarse_desc = outs[0], outs[1]
        # Convert pytorch -> numpy.
        semi = semi.data.cpu().numpy().squeeze()

        # --- Process points.
        dense = np.exp(semi) # Softmax.
        dense = dense / (np.sum(dense, axis=0)+.00001) # Should sum to 1.
        nodust = dense[:-1, :, :]
        # Reshape to get full resolution heatmap.
        Hc = int(H / self.cell)
        Wc = int(W / self.cell)
        nodust = np.transpose(nodust, [1, 2, 0])
        heatmap = np.reshape(nodust, [Hc, Wc, self.cell, self.cell])
        heatmap = np.transpose(heatmap, [0, 2, 1, 3])
        heatmap = np.reshape(heatmap, [Hc*self.cell, Wc*self.cell])
        prob_map = heatmap/np.sum(np.sum(heatmap))

        return heatmap, coarse_desc


    def key_pt_sampling(self, img, heat_map, coarse_desc, sampled):

        H, W = img.shape[0], img.shape[1]

        xs, ys = np.where(heat_map >= self.conf_thresh) # Confidence threshold.
        if len(xs) == 0:
            return np.zeros((3, 0)), None, None
        print("number of pts selected :", len(xs))


        pts = np.zeros((3, len(xs))) # Populate point data sized 3xN.
        pts[0, :] = ys
        pts[1, :] = xs
        pts[2, :] = heat_map[xs, ys]
        pts, _ = self.nms_fast(pts, H, W, dist_thresh=self.nms_dist) # Apply NMS.
        inds = np.argsort(pts[2,:])
        pts = pts[:,inds[::-1]] # Sort by confidence.
        bord = self.border_remove
        toremoveW = np.logical_or(pts[0, :] < bord, pts[0, :] >= (W-bord))
        toremoveH = np.logical_or(pts[1, :] < bord, pts[1, :] >= (H-bord))
        toremove = np.logical_or(toremoveW, toremoveH)
        pts = pts[:, ~toremove]
        pts = pts[:,0:sampled] #we take 2000 keypoints with highest probability from heatmap for our benchmark

        # --- Process descriptor.
        D = coarse_desc.shape[1]
        if pts.shape[1] == 0:
            desc = np.zeros((D, 0))
        else:
          # Interpolate into descriptor map using 2D point locations.
            samp_pts = torch.from_numpy(pts[:2, :].copy())
            samp_pts[0, :] = (samp_pts[0, :] / (float(W)/2.)) - 1.
            samp_pts[1, :] = (samp_pts[1, :] / (float(H)/2.)) - 1.
            samp_pts = samp_pts.transpose(0, 1).contiguous()
            samp_pts = samp_pts.view(1, 1, -1, 2)
            samp_pts = samp_pts.float()
            if self.cuda:
                samp_pts = samp_pts.cuda()
```

```
            desc = nn.functional.grid_sample(coarse_desc, samp_pts)
            desc = desc.data.cpu().numpy().reshape(D, -1)
            desc /= np.linalg.norm(desc, axis=0)[np.newaxis, :]


        return pts, desc
```

```
print('Loading pre-trained network.')
# This class runs the SuperPoint network and processes its outputs.
fe = SuperPointFrontend(weights_path=weights_path,nms_dist = 3,conf_thresh = 0.01,nn_thresh=0.5)
print('Successfully loaded pre-trained network.')
```

```
keypoints_all_left_superpoint = []
descriptors_all_left_superpoint = []
points_all_left_superpoint=[]

keypoints_all_right_superpoint = []
descriptors_all_right_superpoint = []
points_all_right_superpoint=[]

tqdm = partial(tqdm, position=0, leave=True)

for lfpth in tqdm(images_left):
  heatmap1, coarse_desc1 = fe.run(lfpth)
  pts_1, desc_1 = fe.key_pt_sampling(lfpth, heatmap1, coarse_desc1, 80000) #Getting keypoints and descriptors for 1st image

  keypoints_all_left_superpoint.append(to_kpts(pts_1.T))
  descriptors_all_left_superpoint.append(desc_1.T)
  points_all_left_superpoint.append(pts_1.T)


for rfpth in tqdm(images_right):
  heatmap1, coarse_desc1 = fe.run(rfpth)
  pts_1, desc_1 = fe.key_pt_sampling(rfpth, heatmap1, coarse_desc1, 80000) #Getting keypoints and descriptors for 1st image

  keypoints_all_right_superpoint.append(to_kpts(pts_1.T))
  descriptors_all_right_superpoint.append(desc_1.T)
  points_all_right_superpoint.append(pts_1.T)
```

```
!git clone https://github.com/magicleap/SuperGluePretrainedNetwork.git
```

SuperGlue

```
from models.matching import Matching
from models.utils import (compute_pose_error, compute_epipolar_error,
                          estimate_pose, make_matching_plot,
                          error_colormap, AverageTimer, pose_auc, read_image,
                          rotate_intrinsics, rotate_pose_inplane,
                          scale_intrinsics)

def add_superglue(inp0,inp1):
      # Perform the matching.
    pred = matching({'image0': inp0, 'image1': inp1})
    pred = {k: v[0].cpu().numpy() for k, v in pred.items()}
    kpts0, kpts1 = pred['keypoints0'], pred['keypoints1']
    matches, conf = pred['matches0'], pred['matching_scores0']
    timer.update('matcher')

    # Write the matches to disk.
    out_matches = {'keypoints0': kpts0, 'keypoints1': kpts1,
                   'matches': matches, 'match_confidence': conf}
    np.savez(str(matches_path), **out_matches)

        # Keep the matching keypoints.
    valid = matches > -1
    mkpts0 = kpts0[valid]
    mkpts1 = kpts1[matches[valid]]
    mconf = conf[valid]
```

NN + Lowe'sRatio + RANSAC

```
def compute_homography_fast(matched_pts1, matched_pts2,thresh=4):
    #matched_pts1 = cv2.KeyPoint_convert(matched_kp1)
    #matched_pts2 = cv2.KeyPoint_convert(matched_kp2)

    # Estimate the homography between the matches using RANSAC
    H, inliers = cv2.findHomography(matched_pts1,
                                    matched_pts2,
                                    cv2.RANSAC, ransacReprojThreshold =thresh, maxIters=3000)
    inliers = inliers.flatten()
    return H, inliers
```

```
def compute_homography_fast_other(matched_pts1, matched_pts2):
    #matched_pts1 = cv2.KeyPoint_convert(matched_kp1)
    #matched_pts2 = cv2.KeyPoint_convert(matched_kp2)

    # Estimate the homography between the matches using RANSAC
    H, inliers = cv2.findHomography(matched_pts1,
                                    matched_pts2,
                                    0)
    inliers = inliers.flatten()
    return H, inliers
```

```
def get_Hmatrix(imgs,keypts,pts,descripts,ratio=0.8,thresh=4,use_lowe=True,disp=False,no_ransac=False,binary=False):
  lff1 = descripts[0]
  lff = descripts[1]

  if use_lowe==False:
    #FLANN_INDEX_KDTREE = 2
    #index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
    #search_params = dict(checks=50)
    #flann = cv2.FlannBasedMatcher(index_params, search_params)
    #flann = cv2.BFMatcher()
    if binary==True:
      bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)

    else:
```

```
    bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)
    lff1 = np.float32(descripts[0])
    lff = np.float32(descripts[1])


  #matches_lf1_lf = flann.knnMatch(lff1, lff, k=2)
  matches_4 = bf.knnMatch(lff1, lff,k=2)
  matches_lf1_lf = []


  print("\nNumber of matches",len(matches_4))
  '''
  matches_4 = []
  ratio = ratio
  # loop over the raw matches
  for m in matches_lf1_lf:
    # ensure the distance is within a certain ratio of each
    # other (i.e. Lowe's ratio test)
    #if len(m) == 2 and m[0].distance < m[1].distance * ratio:
        #matches_1.append((m[0].trainIdx, m[0].queryIdx))
    matches_4.append(m[0])
  '''
  print("Number of matches After Lowe's Ratio",len(matches_4))
else:
  FLANN_INDEX_KDTREE = 2
  index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
  search_params = dict(checks=50)
  flann = cv2.FlannBasedMatcher(index_params, search_params)
  if binary==True:
    bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
    lff1 = np.float32(descripts[0])
    lff = np.float32(descripts[1])
  else:
    bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)
    lff1 = np.float32(descripts[0])
    lff = np.float32(descripts[1])


  matches_lf1_lf = flann.knnMatch(lff1, lff, k=2)
  #matches_lf1_lf = bf.knnMatch(lff1, lff,k=2)


  print("\nNumber of matches",len(matches_lf1_lf))
  matches_4 = []
  ratio = ratio
  # loop over the raw matches
  for m in matches_lf1_lf:
    # ensure the distance is within a certain ratio of each
    # other (i.e. Lowe's ratio test)
    if len(m) == 2 and m[0].distance < m[1].distance * ratio:
        #matches_1.append((m[0].trainIdx, m[0].queryIdx))
      matches_4.append(m[0])

  print("Number of matches After Lowe's Ratio",len(matches_4))



matches_idx = np.array([m.queryIdx for m in matches_4])
imm1_pts = np.array([keypts[0][idx].pt for idx in matches_idx])
matches_idx = np.array([m.trainIdx for m in matches_4])
imm2_pts = np.array([keypts[1][idx].pt for idx in matches_idx])
'''
# Estimate homography 1
#Compute H1
# Estimate homography 1
#Compute H1
imm1_pts=np.empty((len(matches_4),2))
imm2_pts=np.empty((len(matches_4),2))
for i in range(0,len(matches_4)):
  m = matches_4[i]
  (a_x, a_y) = keypts[0][m.queryIdx].pt
  (b_x, b_y) = keypts[1][m.trainIdx].pt
  imm1_pts[i]=(a_x, a_y)
  imm2_pts[i]=(b_x, b_y)
H=compute_Homography(imm1_pts,imm2_pts)
#Robustly estimate Homography 1 using RANSAC
Hn, best_inliers=RANSAC_alg(keypts[0] ,keypts[1], matches_4,  nRANSAC=1000, RANSACthresh=6)
'''

if no_ransac==True:
  Hn,inliers = compute_homography_fast_other(imm1_pts,imm2_pts)
else:
  Hn,inliers = compute_homography_fast(imm1_pts,imm2_pts,thresh)

inlier_matchset = np.array(matches_4)[inliers.astype(bool)].tolist()
print("Number of Robust matches",len(inlier_matchset))
print("\n")

if len(inlier_matchset)<25:
  matches_4 = []
  ratio = 0.5
  # loop over the raw matches
  for m in matches_lf1_lf:
    # ensure the distance is within a certain ratio of each
    # other (i.e. Lowe's ratio test)
    if len(m) == 2 and m[0].distance < m[1].distance * ratio:
        #matches_1.append((m[0].trainIdx, m[0].queryIdx))
        matches_4.append(m[0])
  print("Number of matches After Lowe's Ratio New",len(matches_4))

  matches_idx = np.array([m.queryIdx for m in matches_4])
  imm1_pts = np.array([keypts[0][idx].pt for idx in matches_idx])
  matches_idx = np.array([m.trainIdx for m in matches_4])
  imm2_pts = np.array([keypts[1][idx].pt for idx in matches_idx])
  Hn,inliers = compute_homography_fast(imm1_pts,imm2_pts)
  inlier_matchset = np.array(matches_4)[inliers.astype(bool)].tolist()
  print("Number of Robust matches New",len(inlier_matchset))
  print("\n")

#H=compute_Homography(imm1_pts,imm2_pts)
#Robustly estimate Homography 1 using RANSAC
#Hn=RANSAC_alg(keypts[0] ,keypts[1], matches_4,  nRANSAC=1500, RANSACthresh=6)

#global inlier_matchset

if disp==True:
  dispimg1=cv2.drawMatches(imgs[0], keypts[0], imgs[1], keypts[1], inlier_matchset, None,flags=2)
  displayplot(dispimg1,'Robust Matching between Reference Image and Right Image ')
```

```
  return Hn/Hn[2,2], len(matches_lf1_lf), len(inlier_matchset)
```

```
H_left_superpoint = []
H_right_superpoint = []

num_matches_superpoint = []
num_good_matches_superpoint = []

for j in tqdm(range(len(images_left))):
  if j==len(images_left)-1:
    break

  H_a,matches,gd_matches = get_Hmatrix(images_left_bgr[j:j+2][::-1],keypoints_all_left_superpoint[j:j+2][::-1],points_all_left_superpoint[j:j+2][::-1],descriptors_all_left_su
  H_left_superpoint.append(H_a)
  num_matches_superpoint.append(matches)
  num_good_matches_superpoint.append(gd_matches)

for j in tqdm(range(len(images_right))):
  if j==len(images_right)-1:
    break

  H_a,matches,gd_matches = get_Hmatrix(images_right_bgr[j:j+2][::-1],keypoints_all_right_superpoint[j:j+2][::-1],points_all_right_superpoint[j:j+2][::-1],descriptors_all_righ
  H_right_superpoint.append(H_a)
  num_matches_superpoint.append(matches)
  num_good_matches_superpoint.append(gd_matches)
```

Continue Stitching

```
from tqdm import tqdm
tqdm = partial(tqdm, position=0, leave=True)
```

```
xmax,xmin,ymax,ymin,t,h,w,Ht = warpnImages(100-1,100-1,scale_factor=2**3,offset=00)
```

```
Step1:Done
Step2:Done
```

```
print(ymax-ymin,xmax-xmin)
```

```
5500 2232
```

```
print(ymax-ymin,xmax-xmin)
```

```
1369 940
```

```
print(ymax-ymin,xmax-xmin)
```

```
5737957 370990
```

```
print(ymax-ymin,xmax-xmin)
```

```
261829 24888
```

```
print(ymax-ymin,xmax-xmin)
```

```
7463438 382989
```

```
warp_imgs_left = final_steps_left_union_gpu(100-1,xmax,xmin,ymax,ymin,t,h,w,Ht,1,scale_factor=2**3,is_gray=True,offset=0)
```

```
100%|██████████| 99/99 [00:58<00:00,  1.69it/s]Step31:Done
```

```
for j in range(1000,443,100):

  xmax,xmin,ymax,ymin,t,h,w,Ht = warpnImages(j-1,j-1,scale_factor=1,offset=j)
  warp_imgs_left = final_steps_left_union_gpu(j-1,xmax,xmin,ymax,ymin,t,h,w,Ht,1,scale_factor=1,is_gray=True,offset=j)
  warp_imgs_all = final_steps_right_union_gpu(warp_imgs_left,j-1,xmax,xmin,ymax,ymin,t,h,w,Ht,scale_factor=1,is_gray=True,offset=j)
```

```
fig,ax =plt.subplots()
fig.set_size_inches(20,20)
ax.imshow(warp_imgs_left,cmap='gray')
ax.set_title('300-Images Mosaic-SIFT-Modified2')
```

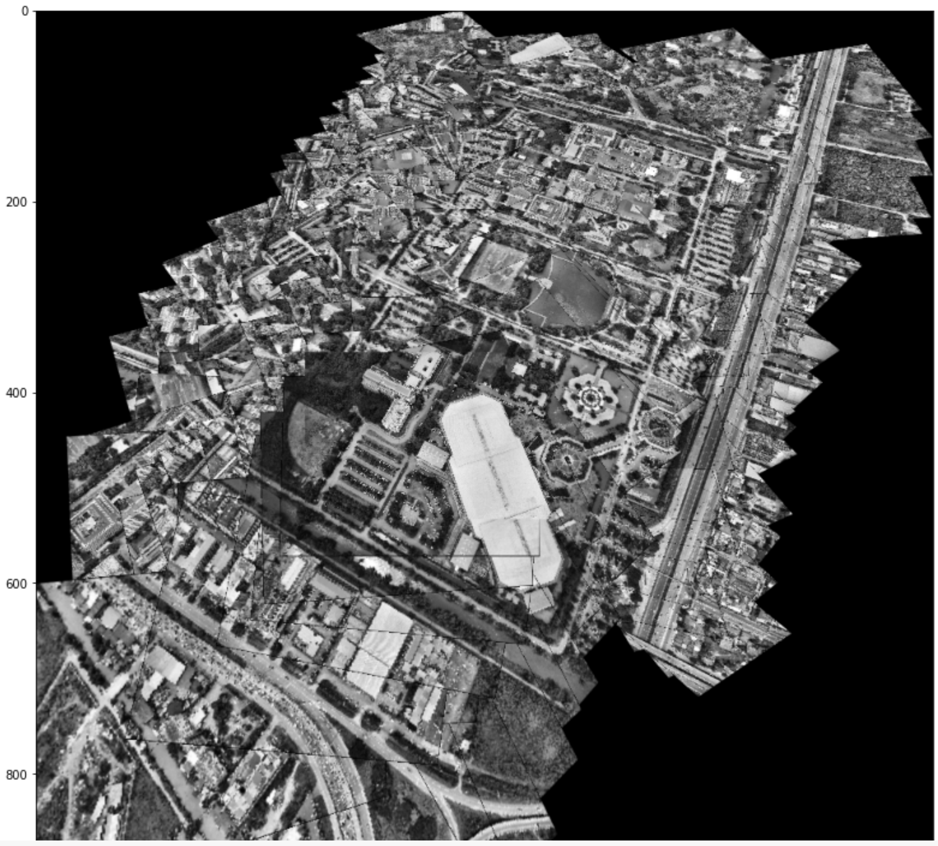
```
warp_imgs_all = final_steps_right_union_gpu(warp_imgs_left,100-1,xmax,xmin,ymax,ymin,t,h,w,Ht,scale_factor=2**3,is_gray=True)
```

```
100%|██████████| 99/99 [00:49<00:00,  1.99it/s]
```

Mosaic After BA (Still there seems to be black lines around contours)

```
fig,ax =plt.subplots()
fig.set_size_inches(20,20)
ax.imshow(warp_imgs_all,cmap='gray')
#ax.set_title('300-Images Mosaic-SIFT-Modified2')
```

<matplotlib.image.AxesImage at 0x7f43b04f9610>




```
f=h5.File('drive/MyDrive/all_images_bgr_sift.h5','r')
input_img_orig = f['data'][10]
f.close()
```

```
plt.imshow(input_img_orig)
```

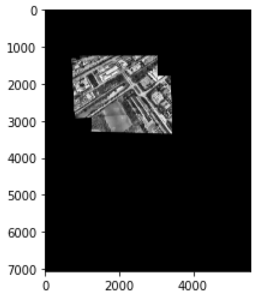
```
plt.imshow(input_img_orig)
```

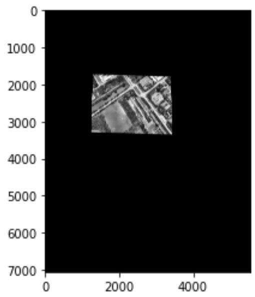
```
xmax,xmin,ymax,ymin,t,h,w,Ht = warpnImages(10,10,scale_factor=16,offset=00)
```

```
Step1:Done
Step2:Done
```

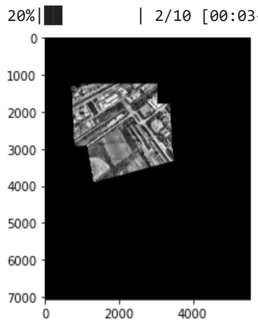
```
warp_imgs_left,H_trans = final_steps_left_union_gpu(10,xmax,xmin,ymax,ymin,t,h,w,Ht,1,scale_factor=1,is_gray=True,offset=0)
```

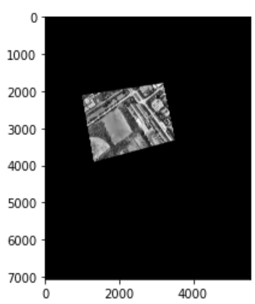
10%|█ | 1/10 [00:00<00:04, 1.95it/s]



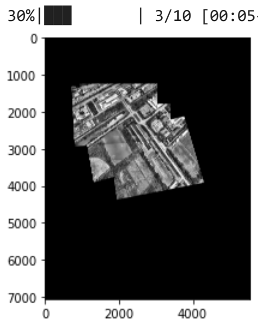


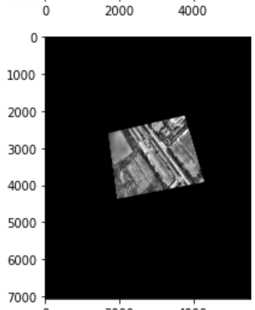
20%|██ | 2/10 [00:03<00:09, 1.16s/it]



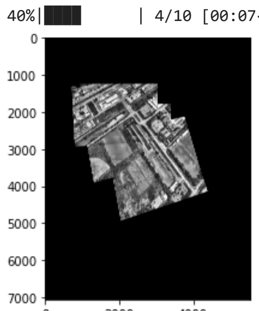


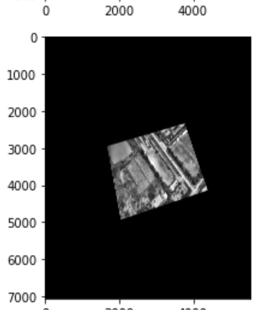
30%|███ | 3/10 [00:05<00:10, 1.47s/it]



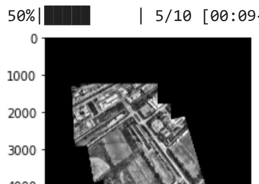


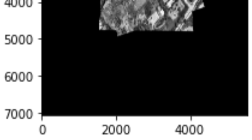
40%|████ | 4/10 [00:07<00:10, 1.69s/it]

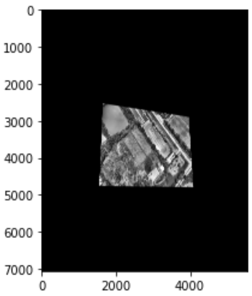




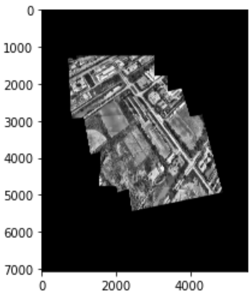
50%|█████ | 5/10 [00:09<00:09, 1.88s/it]

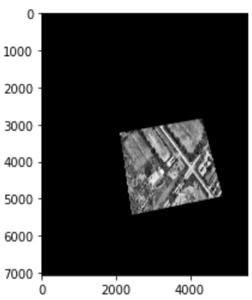




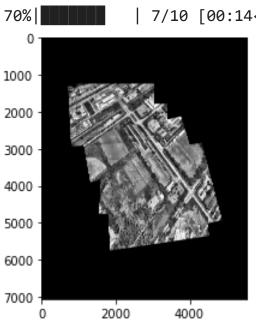


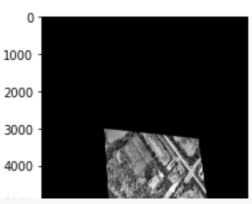

```
60%|██████    | 6/10 [00:12<00:07,  1.98s/it]
```






```
70%|███████   | 7/10 [00:14<00:06,  2.05s/it]
```





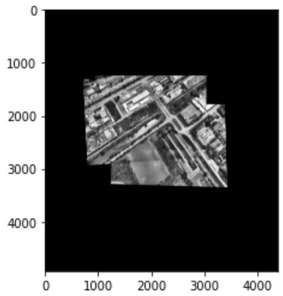

```
xmax,xmin,ymax,ymin,t,h,w,Ht = warpnImages(5,5,scale_factor=1,offset=00)
```

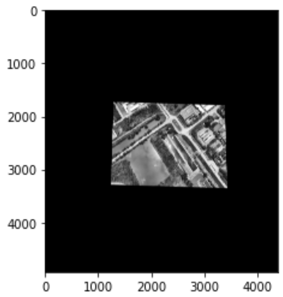
```
Step1:Done
Step2:Done
```

```
80%|████████  | 8/10 [00:16<00:04,  2.10s/it]
```

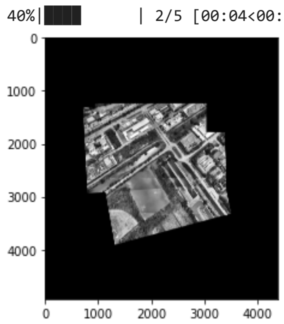
```
warp_imgs_left2,H_trans = final_steps_left_union_gpu(5,xmax,xmin,ymax,ymin,t,h,w,Ht,1,scale_factor=1,is_gray=True,offset=0)
```

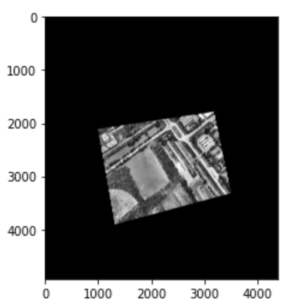
```
20%|██        | 1/5 [00:01<00:05,  1.30s/it]
```



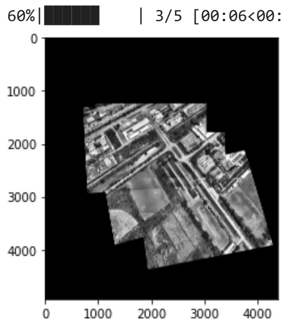


```
40%|████      | 2/5 [00:04<00:05,  1.90s/it]
```





```
60%|██████    | 3/5 [00:06<00:03,  1.95s/it]
```



```
xmax,xmin,ymax,ymin,t,h,w,Ht = warpnImages(10,10,scale_factor=1,offset=5)
```

```
Step1:Done
Step2:Done
```

```
warp_imgs_left2,H_trans = final_steps_left_union_gpu(10,xmax,xmin,ymax,ymin,t,h,w,Ht,warp_img_init_prev=warp_imgs_left2,scale_factor=1,is_gray=True,offset=5,H_trans=H_trans)
```

```
fig,ax =plt.subplots()
fig.set_size_inches(20,20)
ax.imshow(warp_imgs_left,cmap='gray')
ax.set_title('300-Images Mosaic-SIFT-Modified2')
```

```
xmax,xmin,ymax,ymin,t,h,w,Ht = warpnImages(10,10,scale_factor=1,offset=00)
```

```
Step1:Done
Step2:Done
```

```
print(xmax-xmin, ymax-ymin)
```

```
5557 7060
```

```
print(xmax-xmin, ymax-ymin)
```

```
5557 7060
```

```
warp_imgs_left,H_trans = final_steps_left_union_gpu(10,xmax,xmin,ymax,ymin,t,h,w,Ht,scale_factor=1,is_gray=True,offset=0)
```

```
100%|██████████| 10/10 [00:01<00:00,  9.21it/s]Step31:Done
```

```
fig,ax =plt.subplots()
fig.set_size_inches(20,20)
ax.imshow(warp_imgs_left,cmap='gray')
ax.set_title('300-Images Mosaic-SIFT-Modified2')
```

```
print(H_trans)
```

```
[[ 1.65387354e+00 -1.79429296e-01  2.83591525e+03]
 [ 3.33226951e-01  1.46379734e+00  4.52182521e+03]
 [ 7.59172052e-05 -1.26960152e-04  1.10845051e+00]]
```

```
xmax,xmin,ymax,ymin,t,h,w,Ht = warpnImages(20,20,scale_factor=1,offset=11)
```

```
Step1:Done
Step2:Done
```

```
warp_imgs_left2,H_trans2 = final_steps_left_union_gpu(20,xmax,xmin,ymax,ymin,t,h,w,Ht,warp_imgs_left,scale_factor=1,is_gray=True,offset=11,H_trans=H_trans)
```

```
warp_imgs_all = final_steps_right_union_gpu(warp_imgs_left,10,xmax,xmin,ymax,ymin,t,h,w,Ht,scale_factor=1,is_gray=True)
```

```
100%|██████████| 100/100 [03:30<00:00,  2.10s/it]
```

```
fig,ax =plt.subplots()
fig.set_size_inches(20,20)
ax.imshow(warp_imgs_all,cmap='gray')
ax.set_title('300-Images Mosaic-SIFT-Modified2')
```

```
Text(0.5, 1.0, '300-Images Mosaic-SIFT-Modified2')
```