```
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_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_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_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_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
```

```
#%%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_220.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_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')
        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())




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




      #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()
```

```
      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_222.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_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
```

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
```

```
                # 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)
```

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

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

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

```
print
```

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

```
     44133 17883
```

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

```
     5500 2232
```

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

```
     5737957 370990
```

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

```
     261829 24888
```

```
from scipy.sparse import lil_matrix
```

```
A = np.zeros((ymax-ymin,xmax-xmin))
```

```
del A
```

```
B = lil_matrix((ymax-ymin,xmax-xmin))
```

```
del B
```

```
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=1,is_gray=True,offset=0)
```

```
       1%|          | 1/99 [00:05<08:34,  5.25s/it]
```

```
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,10-1,xmax,xmin,ymax,ymin,t,h,w,Ht,scale_factor=1,is_gray=True)
```

```
     100%|██████████| 9/9 [00:06<00:00,  1.35it/s]
```

```
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')
```

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

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

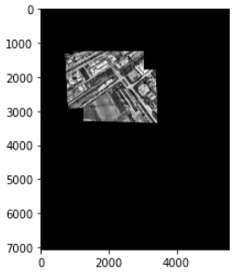
```
plt.imshow(input_img_orig)
```

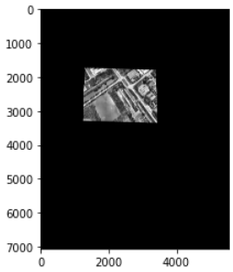
```
xmax,xmin,ymax,ymin,t,h,w,Ht = warpnImages(10,10,scale_factor=16,offset=00)
```

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

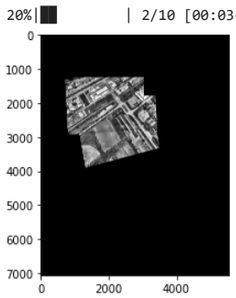
```
warp_imgs_left,H_trans = final_steps_left_union_gpu(10,xmax,xmin,ymax,ymin,t,h,w,Ht,1,scale_factor=1,is_gray=True,offset=0)
```

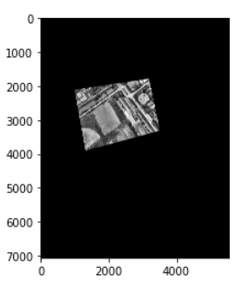
10%|▉ | 1/10 [00:00<00:04, 1.95it/s]



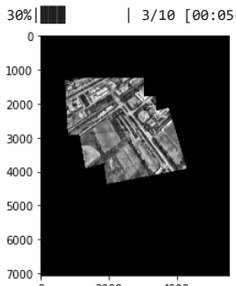


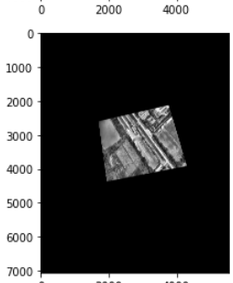
20%|██ | 2/10 [00:03<00:09, 1.16s/it]



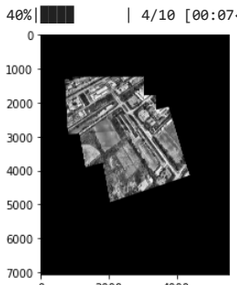


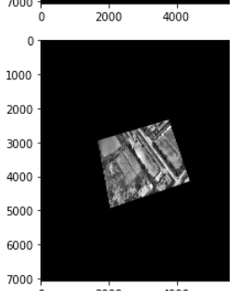
30%|███ | 3/10 [00:05<00:10, 1.47s/it]



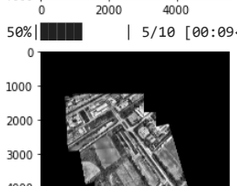


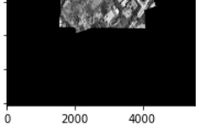
40%|████ | 4/10 [00:07<00:10, 1.69s/it]

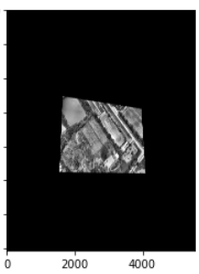




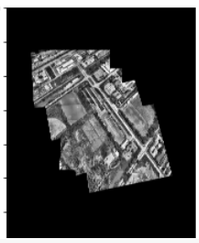
50%|█████ | 5/10 [00:09<00:09, 1.88s/it]







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



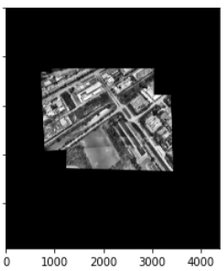

```
xmax,xmin,ymax,ymin,t,h,w,Ht = warpnImages(5,5,scale_factor=1,offset=00)
```

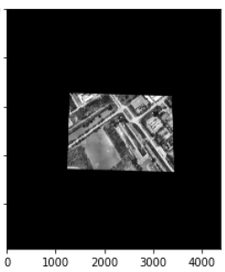
```
Step1:Done
Step2:Done
```



```
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]
```

```
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
```

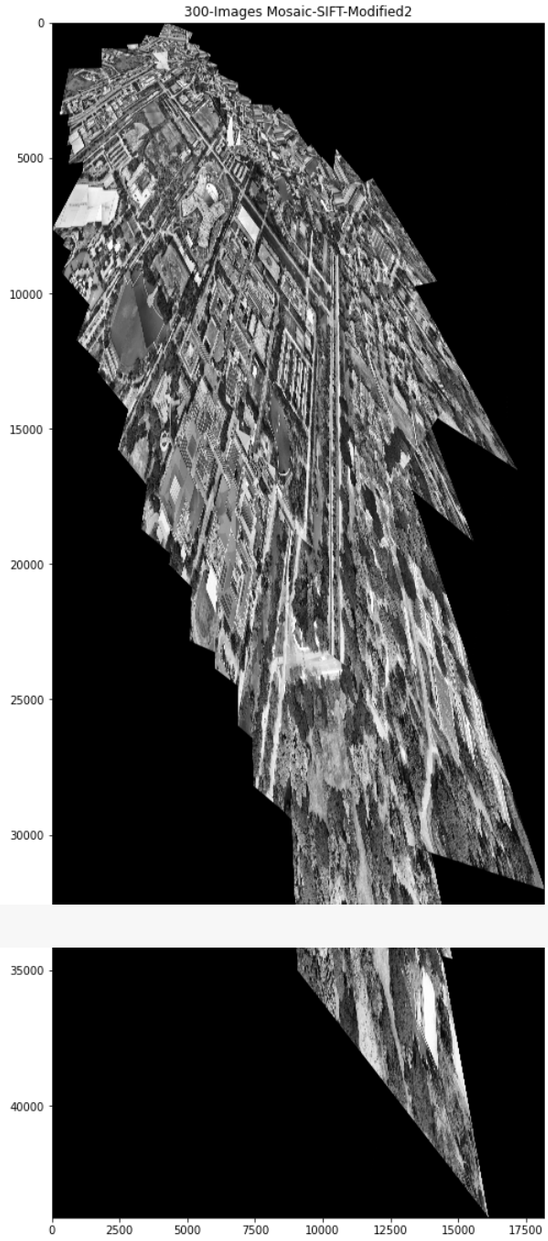
```
warp_imgs_left2,H_trans2 = final_steps_left_union_gpu(20,xmax,xmin,ymax,ymin,t,h,w,Ht,warp_imgs_left,scale_factor=1,is_gray=True,offset=11,H_trans=H_trans)
```

```
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')
```



✓ 0s completed at 11:12 AM