hw3_q2

April 20, 2023

[ ]:
```
import numpy as np
from skimage import io
import scipy.io
import matplotlib.pyplot as plt
```

[ ]:
```
### metrics
def ssd(A, B):
    return np.sum(np.power(A-B, 2), (0,1))

def mad(A,B):
    return np.max(np.abs(A-B), (0,1))

def sad(A,B):
    return np.sum(np.abs(A,B), (0,1))
```

[ ]:
```
### subroutines for algo
def blockmeasure(A, B, measure):
    # computes a measure between each block
    if measure == 'ssd':
        out = ssd(A,B)
    elif measure == 'mad':
        out = mad(A,B)
    elif measure == 'sad':
        out = sad(A,B)
    else:
        raise Exception('Unknown metric: '+str(measure))
    return out
```

[ ]:
```
def assemblepatches(frame1, i, j, frame2, qp, blocksize):
    # Setup blocks to measure distance between
    # A is the reference matrix at i,j from frame1
    # B is matrix of size blocksize centered at locations specified  by qp from␣
↪frame2
    dim0 = blocksize[0]
    dim1 = blocksize[1]
    dim2 = qp.shape[1]
    A = np.zeros((*dim0, *dim1, dim2))
```

```
    B = np.zeros((*dim0, *dim1, dim2))
    b0 = int(blocksize[0]//2)
    b0_ = int(blocksize[0]-b0)
    b1 = int(blocksize[1]//2)
    b1_ = int(blocksize[1]-b1)
    for k in range(dim2):
        x=i; y=j;
        # print(x-b0, x+b0, y-b1, y+b1, A.shape)
        A[:,:,k] = frame1[x-b0:x+b0_, y-b1:y+b1_]
        x=qp[0,k]; y=qp[1,k];
        # print(x-b0, x+b0, y-b1, y+b1)
        B[:,:,k] = frame2[x-b0:x+b0_, y-b1:y+b1_]
    return A, B
```

[ ]:
```
def logsearch4point(frame1, i, j, frame2, p, q, blocksize, searchwindow,␣
 ↪metric):
    # i,j is the base point we're computing metrics for
    # p,q is the new center around which we're searching as part of our␣
 ↪iterative log search
    if searchwindow[0]//4!=1:
        searchpoints = np.zeros((2,5), dtype=int)
        searchpoints[0,0] = -searchwindow[0]//4
        searchpoints[0,4] = +searchwindow[0]//4
        searchpoints[1,1] = -searchwindow[0]//4
        searchpoints[1,3] = +searchwindow[0]//4
        stop = False
    else:
        x,y = np.meshgrid([-1,0,1],[-1,0,1])
        searchpoints = np.vstack((x.ravel(), y.ravel()))
        stop = True
    qp = np.array([[p],[q]], dtype=int) + searchpoints

    met = np.ones(qp.shape[1])*1e16
    valid1 = np.logical_and(np.logical_and(qp[0,:]>0, qp[0,:]<frame1.shape[0]),␣
 ↪\
                            np.logical_and(qp[1,:]>0, qp[1,:]<frame1.shape[1]) )
    valid2 = np.logical_and(np.logical_and(qp[0,:]-blocksize[0]>0, qp[0,:
 ↪]+blocksize[0]<frame1.shape[0]), \
                            np.logical_and(qp[1,:]-blocksize[1]>0, qp[1,:
 ↪]+blocksize[1]<frame1.shape[1]))
    valid = np.logical_and(valid1, valid2)

    A,B = assemblepatches(frame1, i, j, frame2, qp[:, valid], blocksize)
    met[valid] = blockmeasure(A,B, metric)

    # find min among queried points and call logsearch again with that as the␣
 ↪new centre
```

```python
    k = np.argmin(met)
    if not stop:
        est = logsearch4point(frame1, i, j, frame2, qp[0,k], qp[1,k],␣
↪blocksize, searchwindow//2, metric)
        return est
    else:
        return qp[:,k]
```

[ ]:
```python
def logsearchalgo(frame1, frame2, blocksize, searchwindow, metric):
    estimates = np.zeros((frame1.shape[0]-2*int(blocksize[0]//2), \
                          frame1.shape[1]-2*int(blocksize[1]//2), \
                          2))
    for i in np.arange(blocksize[0]//2, frame1.shape[0]-blocksize[0]//2):
        for j in np.arange(blocksize[1]//2, frame1.shape[1]-blocksize[1]//2):
            estimates[i-blocksize[0]//2,j-blocksize[1]//2,:]= np.array([i,j]) -␣
↪logsearch4point(frame1, i, j, frame2, i, j, blocksize, searchwindow, metric)
    return estimates
```

[ ]:
```python
### Main
# loads frames and runs full-search algo on them
# also displays results and computes difference

# hyper-params
blocksize = np.array([16, 16]).reshape(2,1) # 16 16
searchwindow = np.array([48, 48]).reshape(2,1) # 48 48
qn = 'q2'
run = 1

# load frames from middlebury dataset
frame1 = io.imread('q1/other-data-gray/Walking/frame10.png')
frame2 = io.imread('q1/other-data-gray/Walking/frame11.png')
```

[ ]:
```python
metrics = ["sad","ssd","mad"]
for metric in metrics:
    # call algo and save results
    dispest = logsearchalgo(frame1, frame2, blocksize, searchwindow, metric)
    dispest = np.pad(dispest, ((8,8),(8,8),(0,0)), 'constant',␣
↪constant_values=0)
    savefilename = qn+'/myresult_'+metric+'_'+str(run)+'.mat'
    scipy.io.savemat(savefilename, dict(result=dispest))

    print(frame1.shape)
    print(dispest.shape)

    # loadfilename = qn+'/myresult_'+metric+'_'+str(run)+'.mat'
    # loadobj = scipy.io.loadmat(loadfilename)
    # dispest = loadobj['result']
```

```
(480, 640)
(480, 640, 2)
(480, 640)
(480, 640, 2)
(480, 640)
(480, 640, 2)
```

[ ]:
```
# Plot motion estimates
def plotestimates(metric, step, qn='q2', fsz=(15,7)):

    # load frames from middlebury dataset
    frame1 = io.imread('q1/other-data-gray/Walking/frame10.png')
    frame2 = io.imread('q1/other-data-gray/Walking/frame11.png')
    run=1

    steph = step
    stepv = step
    lcrop = 8
    rcrop = 8
    tcrop = 8
    bcrop = 8
    x,y = np.meshgrid(np.arange(frame1.shape[1]), np.arange(frame1.shape[0]))
    x_ = x[lcrop:-rcrop:stepv, tcrop:-bcrop:steph]
    y_ = y[lcrop:-rcrop:stepv, tcrop:-bcrop:steph]

    # create figure
    plt.figure(figsize=fsz)
    plt.suptitle('Metric:'+metric+' Step (horz):'+str(steph)+' Step (vert):
 ↪'+str(stepv))

    # Load estimates
    loadfilename = qn+'/myresult_'+metric+'_'+str(run)+'.mat'
    dispest = scipy.io.loadmat(loadfilename)
    dispest = dispest['result']

    u = dispest[lcrop:-rcrop:stepv, tcrop:-bcrop:steph, 1]
    v = dispest[lcrop:-rcrop:stepv, tcrop:-bcrop:steph, 0]
    plt.subplot(1,2,1)
    plt.imshow(frame1)
    plt.quiver(x_, y_, u, v, color='r')
    plt.title('overlaid estimates')
    plt.savefig(qn+'/
 ↪'+qn+'_'+metric+'_step_'+str(steph)+'_'+str(stepv)+'_overlaid_estimates.png')

    # Compare with ground truth values from middlebury datasetls
    refest = scipy.io.loadmat('q1/grove2_flo10.mat')
    refest = refest['gt']
```

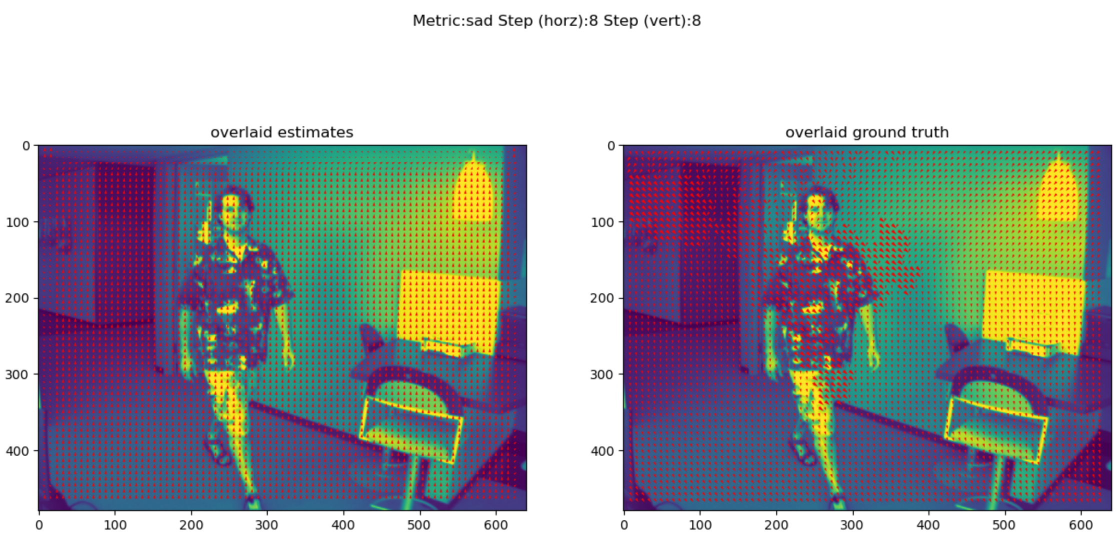
```
    u_ = refest[lcrop:-rcrop:stepv, tcrop:-bcrop:steph, 1]
    v_ = refest[lcrop:-rcrop:stepv, tcrop:-bcrop:steph, 0]
    plt.subplot(1,2,2)
    plt.imshow(frame1)
    plt.quiver(x_, y_, u_, v_, color='r')
    plt.title('overlaid ground truth')
    plt.savefig(qn+'/
↪'+qn+'_'+metric+'_step_'+str(steph)+'_'+str(stepv)+'_overlaid_truth.png')

    rmsediff = np.sum(np.power(dispest[lcrop:-rcrop,tcrop:-bcrop,:
↪]-refest[lcrop:-rcrop,tcrop:-bcrop,:], 2), axis=2)
    maxerr = np.max(rmsediff)
    print(maxerr)
```

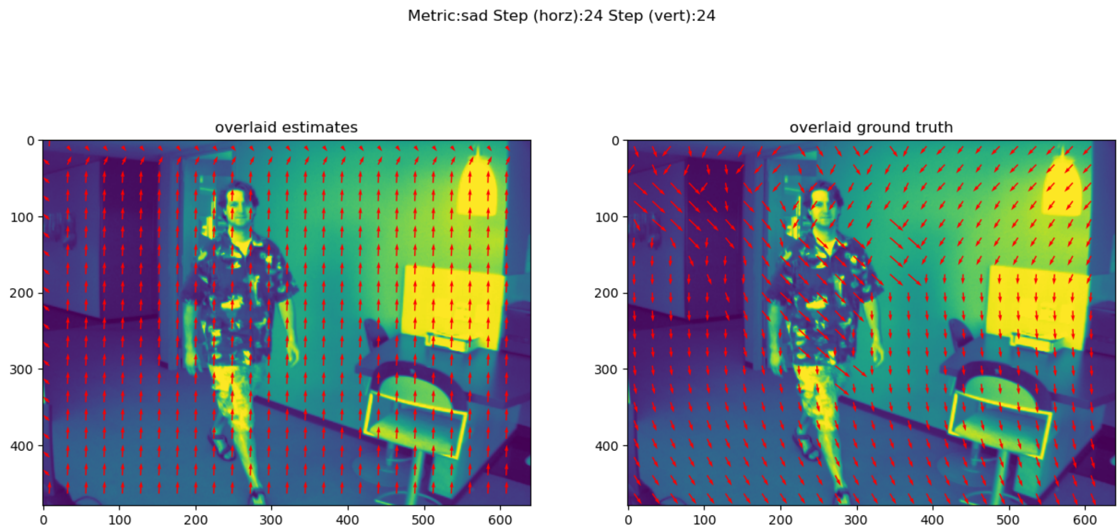
1 Results

Log search estimates are computed for almost the whole frame (excluding the edges). These images show estimates sampled at lower density as specified by the 'step' parameter for ease of viewing.

[ ]:
```
metric='sad'; step=8; plotestimates(metric, step);
metric='sad'; step=24; plotestimates(metric, step);
```

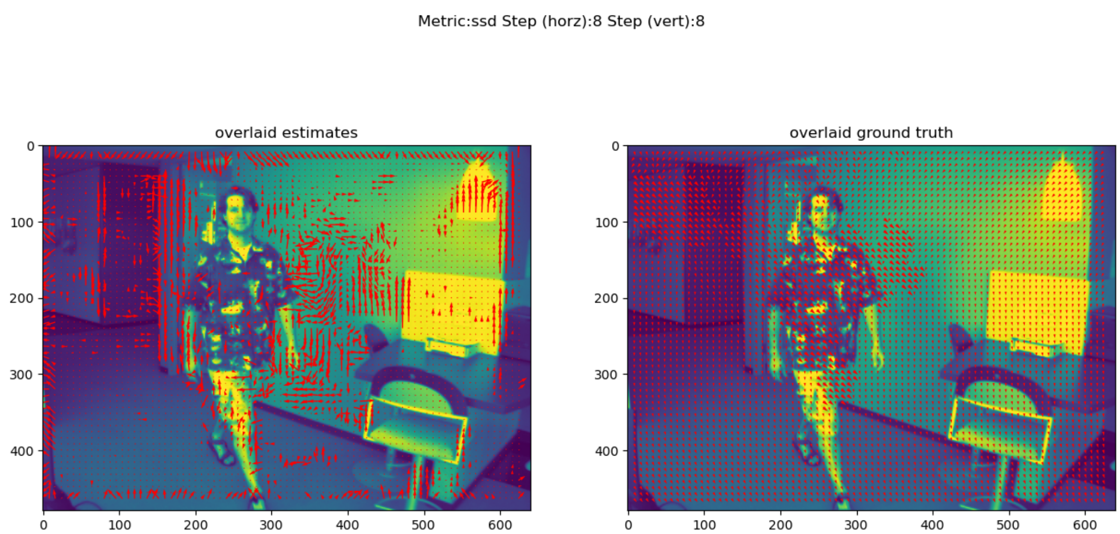
```
650.6022678947145
650.6022678947145
```

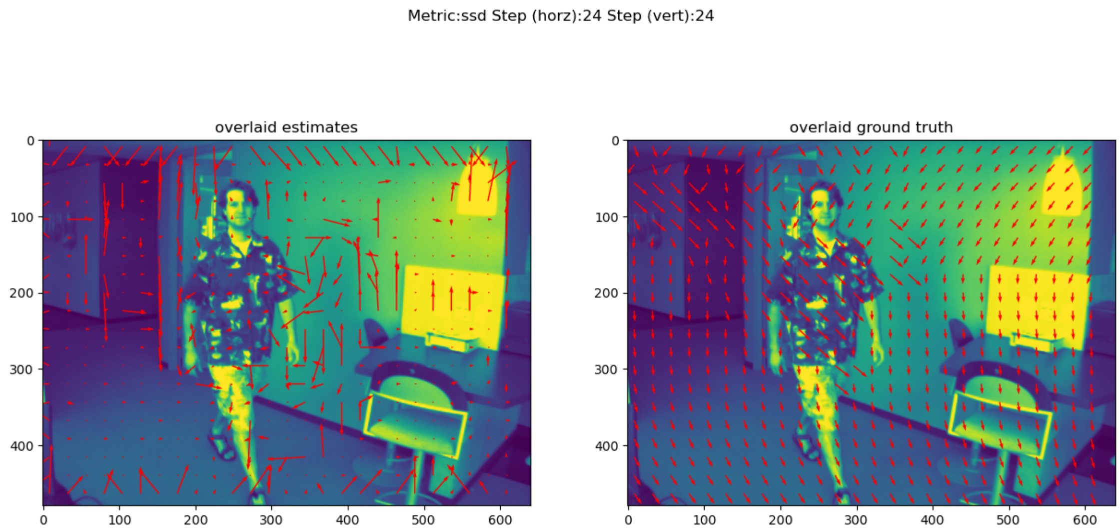




[ ]:
```python
metric='ssd'; step=8; plotestimates(metric, step);
metric='ssd'; step=24; plotestimates(metric, step);
```

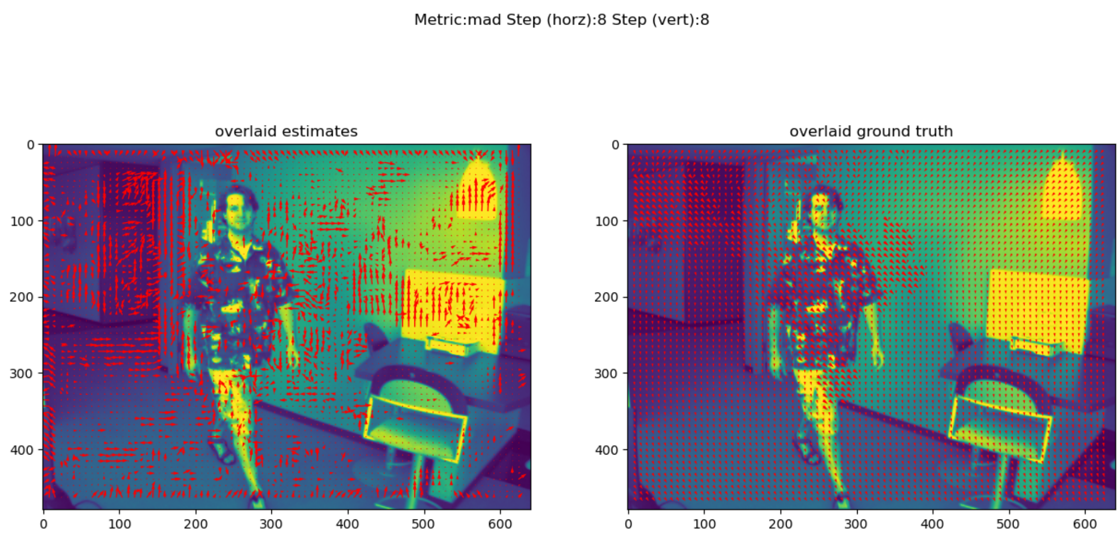
```
674.2907162501747
674.2907162501747
```



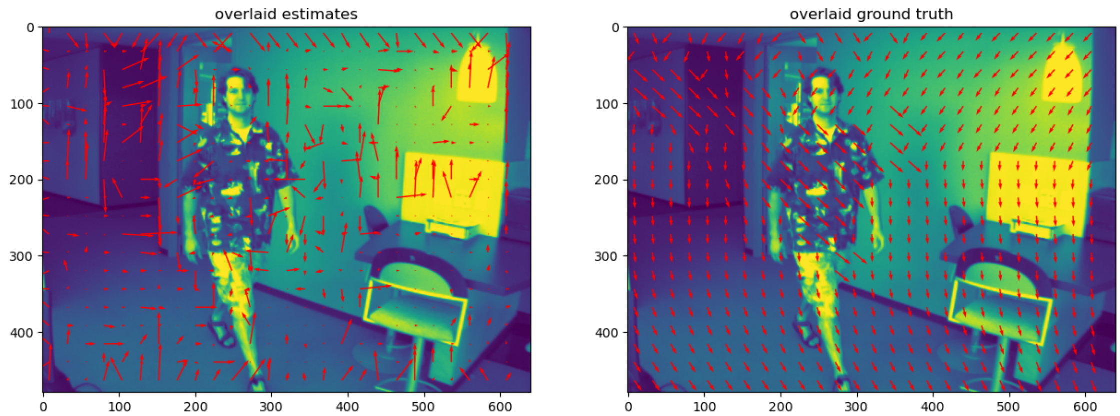


[ ]:
```
metric='mad'; step=8; plotestimates(metric, step);
metric='mad'; step=24; plotestimates(metric, step);
```

```
666.5153583524561
666.5153583524561
```



Metric:mad Step (horz):24 Step (vert):24



2 Comparison of computational complexity

2.1 Full Search Algo

SETUP: For a full search algorithm we need to perform searches for every point in the image. At every point we have searchwindow[0]*searchwindow[1] places to perform block matching at. And block matching is in itself an expensive operation since it is performed between two 16*16 matrices. FS algo computations can be halved smartly since the metrics we use (sad, mad, ssd) are all symmetric operations.

COMPUTATIONAL COMPLEXITY: Therefore, the FS algo roughly has operations of O(M*N*S0*S1*B0*B1) where M,N is the size of the image; S0, S1 are the search window dimensions along dim 0 and dim 1 respectively; B0, B1 are the block dimensions along dim 0 and dim 1 respectively.

RUN TIME: For the chosen frame, the FS Algo ran for nearly 5-10 minutes.

2.2 Log Search Algo

SETUP: The Log search algo also needs to evaluated at every point in the image. At each point, we have an initial search list of 5 points where we compute distance between the blocks centered at these points and the original reference block. We iteratively perform a similar comparison at 5 more points until search space becomes 3x3.

COMPUTATIONAL COMPLEXITY: We can estimate that we get to this termination state in O(log(max(searchwindow[0], searchwindow[1]))) steps and each step we perform only 5 comparisons. If we assume that the searchwindow is square for simplicity, the overall computational complexity is O(M*N*5*log(searchwindow)*B0*B1) following the same convention as in the previous section. The difference between 5*log(searchwindow) and searchwindow^2 makes a huge difference in terms of computational complexity.

RUN TIME: For the chosen frame, the Log search algo ran for just over 1 minute.