Lab 3: Perspective Transformations and Motion Tracking

Shane Kirkley
February 25, 2019

Table of Contents

1. Introduction

In this lab, we will investigate digital image processing for perspective correction and motion tracking. Part I will introduce how to create a linear model using regression. We will apply the discussed linear regression technique to change the apparent perspective of an image. In Part II, we will investigate motion tracking between two consecutive images. This will allow us to view the changes between two frames of video, and to predict an image based on the motion. We will examine two statistical minimization methods for motion estimation and discuss their differences.

2. Background

Digital images can be represented as two-dimensional arrays of integer intensity values. Color images have three intensity values for each pixel (Red, Green and Blue) while grayscale images have one value per pixel. Recognizing that images are simply matrices of intensity values allows us to use matrix arithmetic to manipulate images.

Recall that a system of linear equations can be written using matrices. Consider *m*-by-*n* matrix **A**, a column vector **x** of *n* elements, and column vector **b** of *m* elements. The matrix equation $\mathbf{Ax} = \mathbf{b}$ is equivalent to the system of linear equations

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n &= b_1 \\ &\vdots \\ a_{m,1}x_1 + a_{m,2}x_2 + \cdots + a_{m,n}x_n &= b_m \end{aligned}$$

If these equations are independent, then $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. If **A** is square, then $\mathbf{A}^{-1}$ is the inverse matrix. If **A** isn't square, $\mathbf{A}^{-1}$ is found using the pseudoinverse. We find the pseudoinverse of **A** with linearly independent columns as $\mathbf{A}^{-1} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T$. These observations on matrix arithmetic are key to modeling a system using linear regression, which we will examine in Part I of this lab.

Part II of this lab will make use of projective homogeneous coordinates. Recall that in projective geometry, a point in N-D space is described by an (N+1)-D vector. The last coordinate in the vector, *k*, is a multiplier of the preceding coordinates. As specified, we will represent 2-D (*x*, *y*) coordinates in an image with homogeneous coordinates of the form (*k***x*, *k***y*, *k*). For the problems examined in this lab, we will find that the use homogeneous coordinates makes for easier calculations since it enables the use of linear techniques instead of struggling with non-linear relationships.

3. Part I: Correcting Perspective Distortion

a. Linear Regression

Linear regression is an approach to modeling the relationship between one or more independent variables and a scalar response. We will use linear regression to perform perspective distortion correction, but first let's examine a simpler situation in which linear regression can be used to create an accurate prediction model.

Assume that a company is doing research on dieting. They believe electrode measurements made at 3 different positions on the brain will provide a way to predict how hungry a person feels. The subjective hunger a person feels is reported on a scale where negative numbers are not hungry at all, to positive number where they are very hungry. The gathered electrode data and reported subjective hunger is shown in Figure 1.

Electrode 1 (mV)	Electrode 2 (mV)	Electrode 3 (mV)	**Reported** Subjective Hunger
1	8	3	65.66
-46	-98	108	-1763.1
5	12	-9	195.2
63	345	-27	3625
23	78	45	716.9
-12	56	-8	339
1	34	78	-25.5
56	123	-5	1677.1

Figure 1 - Experimental Data

We want to find the coefficients of a linear model for this data that minimizes the mean squared error (MSE) between experimental data and predicted data. One method of minimizing this error is multiplying the response vector (subjective hunger) by the pseudoinverse of the regressor matrix (electrode data). We compute the pseudoinverse using function `pseudo_inverse`, which can be seen in script `linear_regression.py` in the appendix. We find the coefficients of our linear model as `[11.93027003  8.01762435 -3.98242241]`. These coefficients can now be used to find the predicted subjective hunger by computing the dot product of the electrode data matrix with our coefficients. The resulting data is shown in Figure 2.

Electrode 1 (mV)	Electrode 2 (mV)	Electrode 3 (mV)	**Predicted** Subjective Hunger
1	8	3	64.124
-46	-98	108	-1764.621
5	12	-9	191.705
63	345	-27	3625.213
23	78	45	720.562
-12	56	-8	337.683
1	34	78	-26.099
56	123	-5	1674.175

Figure 2 - Predicted Data with Linear Regression Model

The MSE provides a metric for how closely our model predicts data in comparison to known values. This is computed using function `MSE`, again found in script `linear_regression` in the appendix. We find the MSE between our model predictions and actual data to be `5.124`. Because the MSE is small in comparison to the range of subjective hunger, we conclude that this linear model is a reasonably good predictor of subjective hunger based on electrode data. More data points would result in either a more accurate linear model, or an increased MSE. A high MSE would indicate that a linear model is insufficient in predicting subjective hunger based on electrode data.

b. Perspective Correction

The method of linear regression described in the previous section can be used for perspective correction in digital images. Given a set of points in a distorted image known to lie on a plane, there exists a matrix, H, that maps the distorted image points to corrected image points. Figure 3 shows a distorted image on which we will experiment. First we select 20 points, half of which lie on a known line and the other half which lie on a different line. Importantly, these points all lie on the same plane - the windowed wall of the building. Then we choose 20 corresponding points in a new blank image for which the selected points will map to. The selected points and their corresponding mapped points are shown in the table following Figure 3..

Figure 3 - Distorted image. (PC_test_2.jpg)

Distorted Image Point (row, col)	Processed Image Point (row, col)
(339, 294)	(200, 90)
(334, 312)	(200, 180)
(329, 331)	(200, 270)
(322, 358)	(200, 360)
(314, 388)	(200, 450)
(304, 426)	(200, 540)
(291, 475)	(200, 630)
(275, 537)	(200, 720)
(250, 625)	(200, 810)

(214, 754)	(200, 900)
(455, 290)	(600, 90)
(460, 308)	(600, 180)
(465, 328)	(600, 270)
(471, 353)	(600, 360)
(479, 384)	(600, 450)
(488, 421)	(600, 540)
(499, 469)	(600, 630)
(515, 532)	(600, 720)
(539, 617)	(600, 810)
(573, 745)	(600, 900)

We use homogeneous coordinate vectors to represent points in the distorted and processed image. Consider the mapping of point (c, d) in the distorted image to point (a, b) in the processed image. We seek to find a matrix H such that

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & 1 \end{bmatrix} \begin{bmatrix} c \\ d \\ 1 \end{bmatrix}, \text{ where } \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} v_1/v_3 \\ v_2/v_3 \end{bmatrix}.$$

We solve for v_1/v_3 and v_2/v_3 to find expressions for a and b, then arrange the expressions in matrix form to get

$$\begin{bmatrix} c & d & 1 & 0 & 0 & 0 & -ac & -ad \\ 0 & 0 & 0 & c & d & 1 & -bc & -bd \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix}.$$

Here we note that for each additional point that we choose, the left-side matrix and right-side vector extend down by two rows. We see that solving for vector h becomes a linear regression problem much like the subjective hunger problem in the previous section. The dot product of the pseudoinverse of the left-side matrix with the right-side vector will yield the vector h, which we can reshape to the 3x3 matrix H by appending 1 to the bottom right element location.

Now that we have computed H, we can use H^{-1} to find a corresponding pixel in the distorted image for each pixel in the corrected image. Since the returned pixel (c, d) consists of floating point numbers, we use bilinear interpolation to estimate a corrected pixel value based on the surrounding pixels in the distorted image. If the point lies outside the bounds of original image, we simply leave the pixel black. Figure 4 shows the results of perspective correction on the distorted image from Figure 3. The code used to generate this image can be found in script `perspective_correction.py` in the appendix.

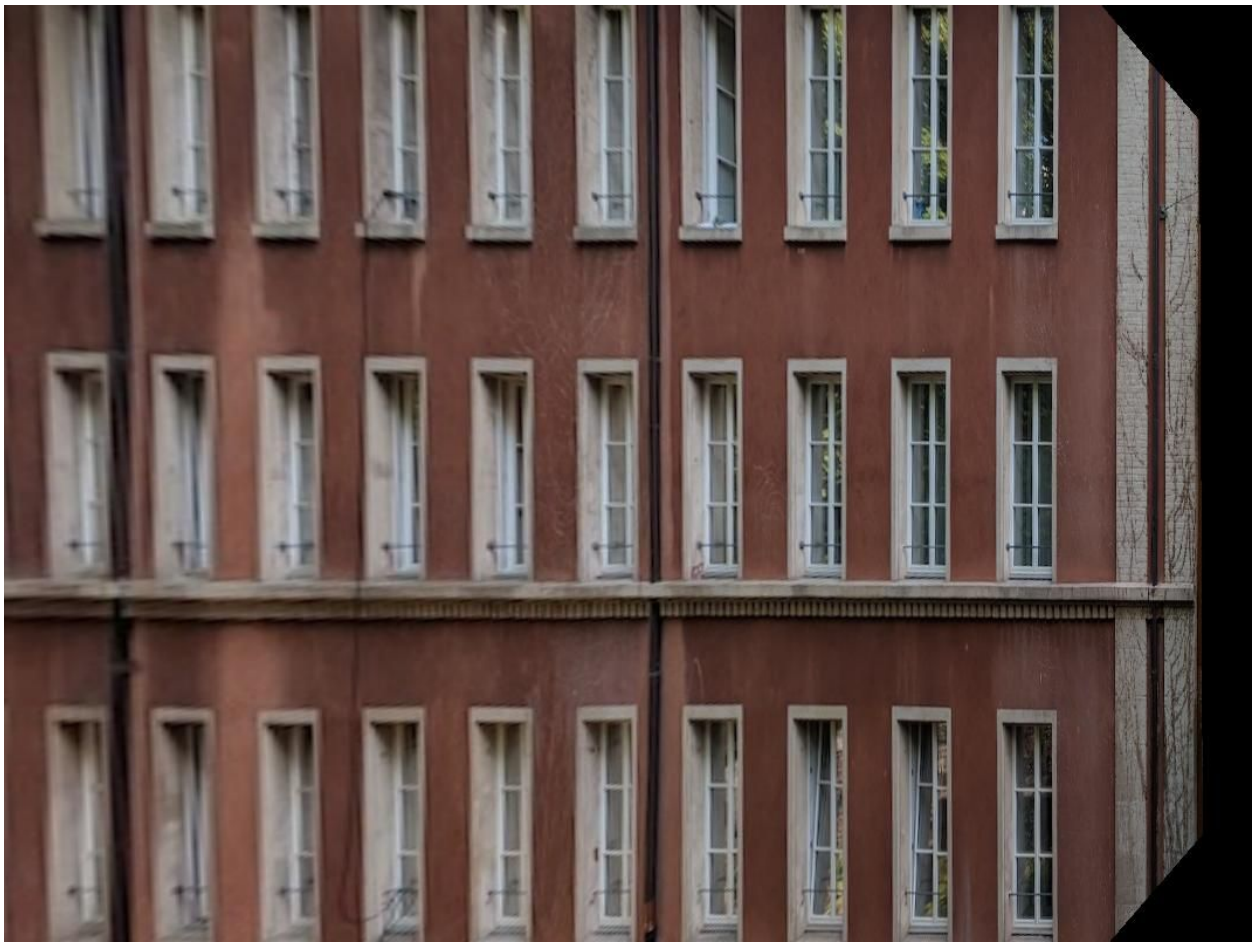

Figure 4 - Perspective corrected image.

The corrected image now shows a perpsective that is closer to directly facing the building,however we see that we have introduced a new kind of distortion - the windows appear thinner than we would expect from the original image. This is due to the spacing of points that we chose for the processed image. We have effectively zoomed in on the wall of the building, stretching it vertically and removing the sky and ground from the image. Additionally, we have lost some of the left side of the image, while introducing a black section to the right side of the image, where mapped points lie outside of the size of the original image. By selecting different points for the corrected image, we can adjust the distortion that we introduce to achieve different

results. Figure 5 shows the corrected image after offsetting both rows of selected points by 100 pixels (line 1: row 200 to 300, and line 2: row 600 to 500).

Figure 5 - Different perspective correction.

This image represents the building in a more satisfactory way - the windows appear at a width that we would expect based on the original image. However, we have introduced more black pixels to the right side of the image. As with the first image, we are only displaying 10 windows and have cut off the left side of the building due to our selection of points. We note that we have introduced more distortion on the bottom of the image, where we see that the angled window and bush are now heavily stretched out. These pixels are not part of the plane that we selected points on. From this result, we can infer that points not on the selected plane will always have some sort of introduced distortion, because our linear transformation only works for a known plane. We also see that windows on the right side of the image appear to be higher resolution than those on the left side of the image. The left side windows in the distorted image were farther away from the camera and thus the image contained less information about them, compared to the windows on the right which were closer to the camera. Thus increasing the size of these elements causes the bilinear interpolation method to introduce blurriness to the corrected image.

4. Part II: Motion Tracking

Video is formed by displaying temporally consecutive frames of image data at some frame rate. When objects move in a video, consecutive frames will have some changes in their image data due to the change in location of the moving objects. Consider two consecutive frames, *n* and *n+1*, in which some movement has occured. We can break up frame *n+1* into blocks, and search each block in frame *n* using a statistical method to find a best match, indicating where the block moved from. For now we can choose the statistical method to be the minimum Mean Square Error (MSE) between a block of data in frame *n+1* and a block of data in frame *n*. Once the MSE for each block in some search area from frame *n* has been calculated, we select the block with the lowest MSE as the origin of the frame *n+1* block. We then repeat this method for each block in frame *n+1*. The larger the pixel search area in frame *n*, the slower this technique will be.

Figure 6 on the following page shows two consecutive frames of image data, which we label *n* and *n+1*. The script `motion_prediction.py`, located in the appendix, generates a vector field graph of the calculated block movements from frame *n* to *n+1,* using lowest MSE. This graph is shown in Figure 7. We note that this graph has a few vectors that estimate motion where we can see there is actually little or none between frames. This is a byproduct of the MSE minimization technique. Our motion estimation function could be further refined by identifying large outlier vectors and omitting them from the results. This would of course increase complexity and thus increase the run time of our program. Figure 8 shows the image resulting from assigning each block with the closest match from frame *n* to *n+1*. We can see some small errors in this predicted image, especially in details that are perhaps smaller than our block size of 16 pixels.

Using the minimum Mean Absolute Error (MAE) between blocks instead of the MSE results in a slightly different motion estimation. Figures 9 and 10 show the vector field graph and resulting image prediction using the minimum MAE match. We note that the results are very similar to using the minimum MSE, but some small differences can be found. The metrics are inherently different. For some error *e* between two blocks, the MSE = e^2 and the MAE = $|e|$. For $|e| > 1$, we find $e^2 > |e|$, and for $|e| < 1$, we find $e^2 < |e|$. Thus we see that the two metrics can provide different results in the minimization function, although for our purpose the end result is very similar.

Figure 6 - Frame n (top) and Frame n+1 (bottom)

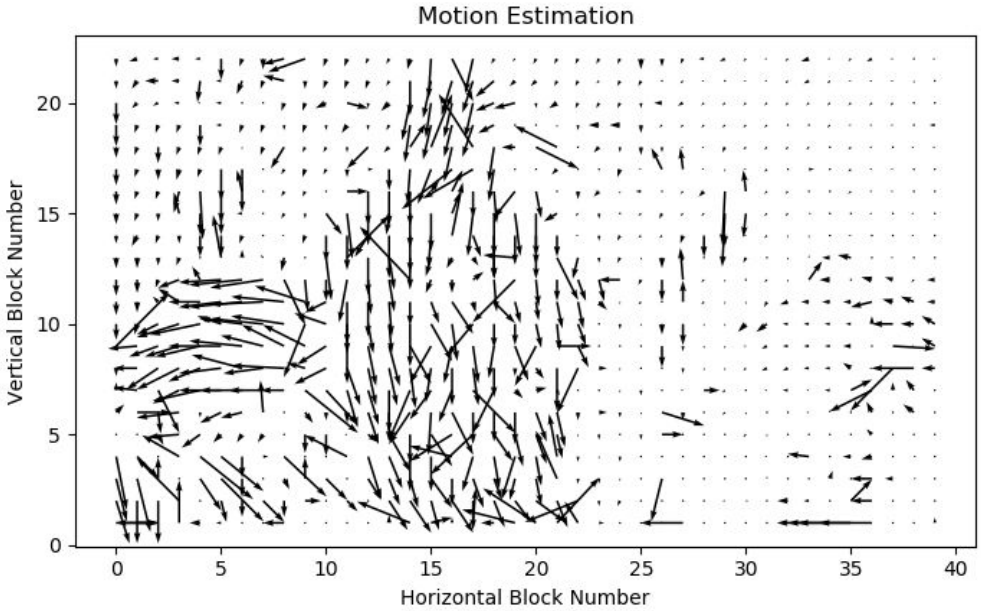

Figure 7 - Vector field graph of movement between frame's n and n+1 using minimum MSE. Vectors point from their position in frame n+1 to their estimated origin in frame n.

Figure 8 - Frame n+1 generated using motion estimation from n to n+1 with minimized MSE.

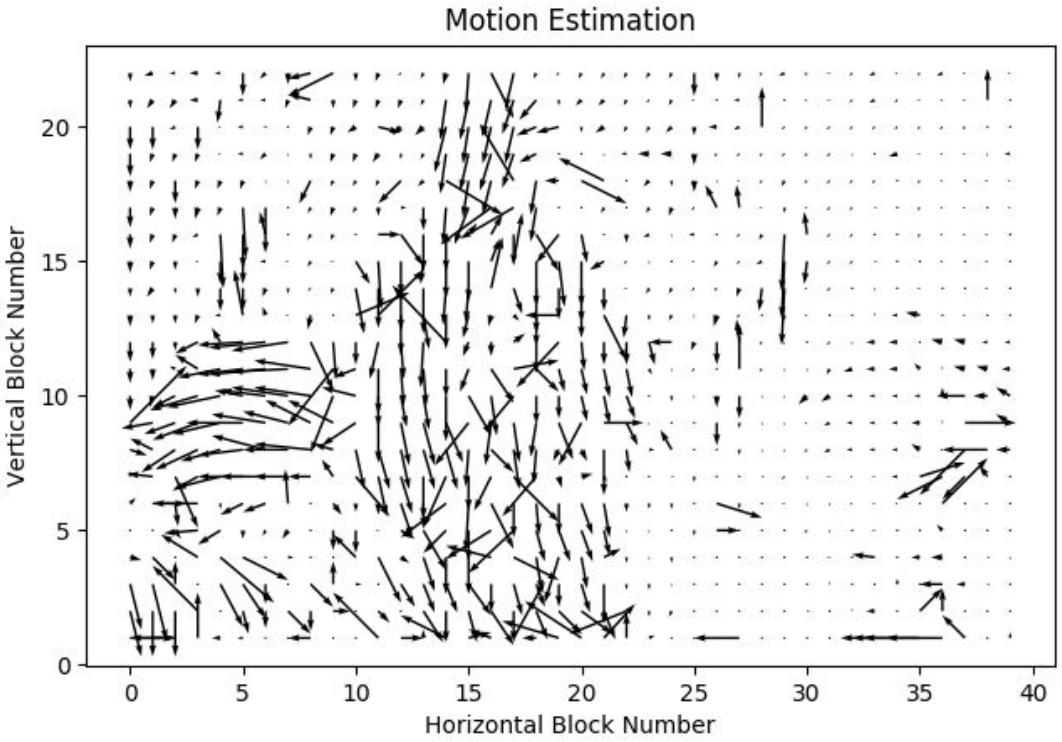

Figure 9 - Vector field graph of motion from frame n to n+1 using minimum MAE.

Figure 10 - Frame n+1 generated using motion estimation from n to n+1 with minimized MAE.

Conclusion

In this lab, we examined a linear method of perspective correction for digital images. We learned how to select points to create a more accurate model for perspective correction, and discovered the trade-off of this method in the unwanted distortion of parts of the image not on a known plane. We also investigated motion estimation techniques between two images, which gives insight into how motion tracking in video could be implemented. The difference between using two different common statistical error measurements for motion estimation was examined and shown in practice.

Appendix

linear_regression.py

```
import numpy as np
import sys
import math
from numpy.linalg import inv

# perform linear regression to predict the sujectively reported
# hunger based on the electrode data. Write code to calculate the
# psuedo-inverse yourself (don't use library call).

# psuedo inverse assuming linearly independent columns of A
def psuedo_inverse(A):
   return inv(A.T.dot(A)).dot(A.T)

# MSE returns the mean-squared error between sets A and B
def MSE(A, B):
   return ((A-B)**2).mean()

data = np.array([[1,     8,   3], \
                 [-46, -98, 108], \
                 [5,    12,  -9], \
                 [63,  345, -27], \
                 [23,   78,  45], \
                 [-12,  56,  -8], \
                 [1,    34,  78], \
                 [56,  123,  -5]])

hunger = np.array([65.66, -1763.1, 195.2, 3625, 716.9, 339, -25.5, 1677.1])

x = np.dot(psuedo_inverse(data), hunger)
print("Coeffficients:")
print(x)

predict = np.dot(data,x)
mse = MSE(hunger, predict)

print("actual hunger:")
print(hunger)
print("linear regression prediction:")
print(predict)
print("Mean Squared Error:")
print(mse)
```

perpesctive_correction.py

```
import numpy as np
import sys
import math
from numpy.linalg import inv
from PIL import Image
import bilinear_interp as bi

def psuedo_inverse(A):
   return inv(A.T.dot(A)).dot(A.T)

pilim = Image.open("PC_test_2.jpg")
imdat = np.asarray(pilim, np.float)  # put PIL image into numpy array of floats
new_imdat = np.zeros(imdat.shape)

# array of source image points (c, d)
source = np.array([[339, 294], [334, 312], [329, 331], [322, 358], [314, 388],\
                   [304, 426], [291, 475], [275, 537], [250, 625], [214, 754], \
                   [455, 290], [460, 308], [465, 328], [471, 353], [479, 384], \
                   [488, 421], [499, 469], [515, 532], [539, 617], [573, 745] ])

# array of points in new image (a, b)
target = np.array([[200, 90] , [200, 180], [200, 270], [200, 360], [200, 450],\
                   [200, 540], [200, 630], [200, 720], [200, 810], [200, 900], \
                   [600, 90] , [600, 180], [600, 270], [600, 360], [600, 450], \
                   [600, 540], [600, 630], [600, 720], [600, 810], [600, 900] ])

T = target.flatten()
V = np.zeros((40, 8))
""" V is hard coded for 20 points, could change to arbitrary size by
   appending rows instead of filling in place. """
# for each point fill in two rows of V (can this be done cleaner?)
for i in range(source.shape[0]):
   c = source[i,0]
   d = source[i,1]
   a = target[i,0]
   b = target[i,1]
   V[2*i] = [c, d, 1, 0, 0, 0, -1*a*c, -1*a*d]
   V[2*i+1] = [0, 0, 0, c, d, 1, -1*b*c, -1*b*d]

# find H by getting inverse of V, dot with T, append 1 and reshape to square.
h = np.dot(psuedo_inverse(V), T)
h = np.append(h, 1)
H = np.reshape(h, (3,3))

# loop through new image pixels
for a in range(new_imdat.shape[0]):
```

```
   for b in range(new_imdat.shape[1]):
       # find the c,d from input image corresponding to a,b output
       v = np.array([a, b, 1])
       cd = np.dot(inv(H), v)
       c = cd[0]/cd[2]
       d = cd[1]/cd[2]
       # use bilinear_interp to assign value of corresponding pixel
       new_imdat[a,b] = bi.bilinear_interp(d, c, imdat)

Image.fromarray(new_imdat.astype('uint8')).save("test1.jpg")
Image.fromarray(new_imdat.astype('uint8')).show()
```

motion_prediction.py

```
import numpy as np
import sys
import math
from numpy.linalg import inv
from PIL import Image
import matplotlib.pyplot as plt
import me_method as me

filename_out = "xipredict.jpg"

# N is block size
N = 16

# get frame n data
pilim = Image.open("xi01.jpg")
n_imdat_rgb = np.asarray(pilim, np.float)
n_imdat_bw = np.asarray(pilim.convert('L'), np.float)

# get frame n+1 data
pilim = Image.open("xi02.jpg")
n1_imdat_rgb = np.asarray(pilim, np.float)
n1_imdat_bw = np.asarray(pilim.convert('L'), np.float)

# create prediction image data array
predict_imdat = np.zeros(n1_imdat_rgb.shape)

U = []
V = []
X = []
Y = []

# for each block in frame n+1, find the matched block in frame n
for row in range(math.floor(n1_imdat_bw.shape[0]/N)):
   for col in range(math.floor(n1_imdat_bw.shape[1]/N)):
```

```
        # extract the block from frame n+1
        block = n1_imdat_bw[row*N:row*N+N, col*N:col*N+N]
        # get prediction block, mse, and offsets for best match
        pred_block, mse, row_off, col_off = me.motion_match(row*N, col*N, 20,
block, n_imdat_bw, n_imdat_rgb)
        # (X,Y) are coords in frame n (tail), (U,V) coord in frame n+1 (head)
        U = np.append(U, col_off)
        V = np.append(V, -1*row_off)
        X = np.append(X, col)
        Y = np.append(Y, n1_imdat_bw.shape[0]/N - row)
        predict_imdat[row*N:row*N+N, col*N:col*N+N] = pred_block

plt.quiver(X,Y,U,V)
plt.title("Motion Estimation")
plt.xlabel("Horizontal Block Number")
plt.ylabel("Vertical Block Number")
# plt.ion()
plt.show()
val = input('hit enter to continue')

Image.fromarray(predict_imdat.astype('uint8')).save(filename_out)
Image.fromarray(predict_imdat.astype('uint8')).show()
```