CSE 252A Computer Vision I Fall 2018 - Assignment 4

Instructor: David Kriegman

Assignment Published On: Tuesday, November 27, 2018

Due On: Friday, December 7, 2018 11:59 pm

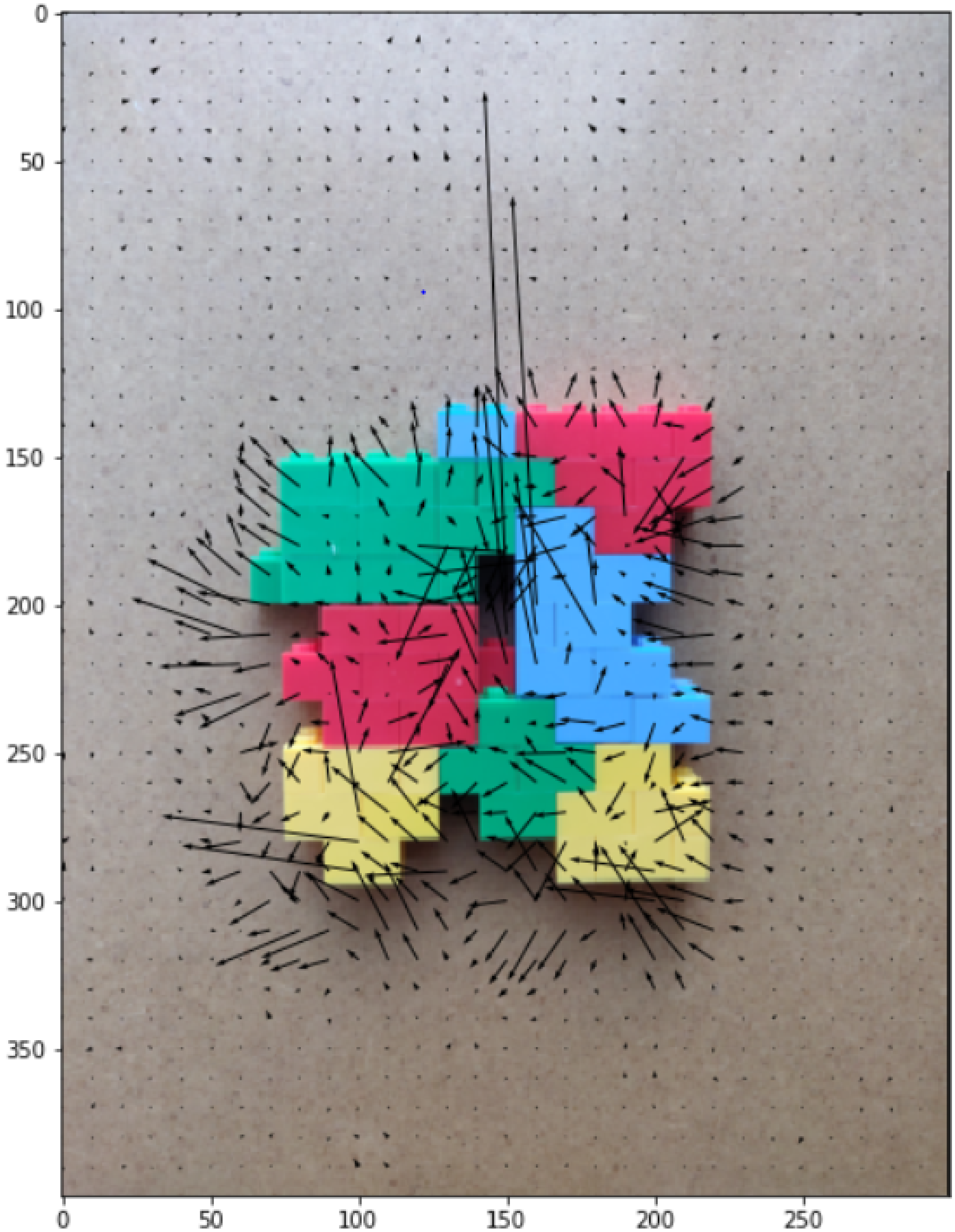
Instructions

- Review the academic integrity and collaboration policies on the course website.
- This assignment must be completed individually.
- Programming aspects of this assignment must be completed using Python in this notebook.
- If you want to modify the skeleton code, you can do so. This has been provided just to provide you with a framework for the solution.
- You may use python packages for basic linear algebra (you can use numpy or scipy for basic operations), but you may not use packages that directly solve the problem.
- If you are unsure about using a specific package or function, then ask the instructor and teaching assistants for clarification.
- You must submit this notebook exported as a pdf. You must also submit this notebook as .ipynb file.
- You must submit both files (.pdf and .ipynb) on Gradescope. You must mark each problem on Gradescope in the pdf.
- **Late policy** - 10% per day late penalty after due date up to 3 days.

Problem 1: Optical Flow [10 pts]

In this problem, the single scale Lucas-Kanade method for estimating optical flow will be implemented, and the data needed for this problem can be found in the folder 'optical_flow_images'.

An example optical flow output is shown below - this is not a solution, just an example output.

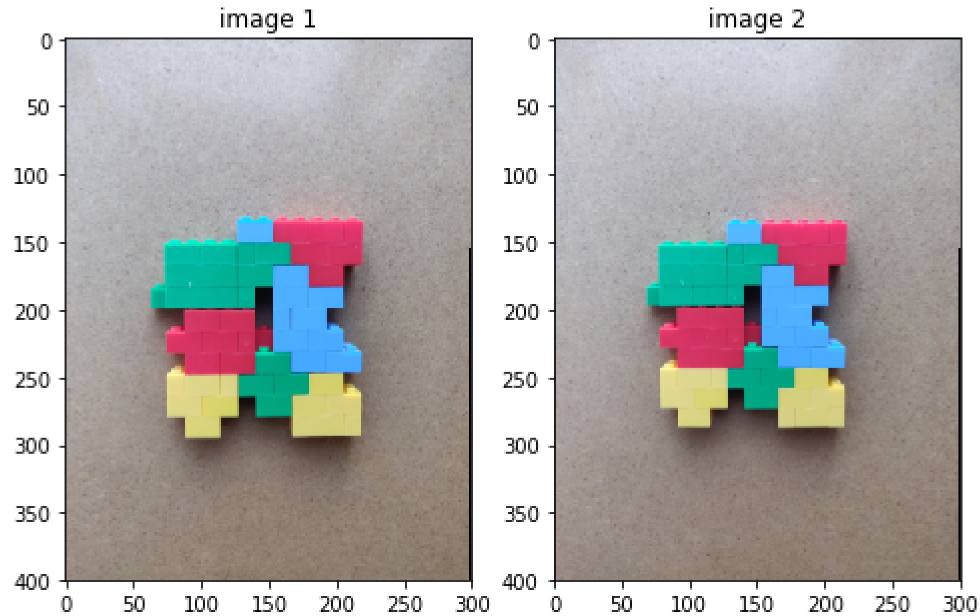


Part 1: Lucas-Kanade implementation [5 pts]

Implement the Lucas-Kanade method for estimating optical flow. The function 'LucasKanade' needs to be completed.

In [39]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import convolve2d as conv2

def grayscale(img):
    '''
    Converts RGB image to Grayscale
    '''
    gray=np.zeros((img.shape[0],img.shape[1]))
    gray=img[:,:,0]*0.2989+img[:,:,1]*0.5870+img[:,:,2]*0.1140
    return gray

def plot_optical_flow(img,U,V):
    '''
    Plots optical flow given U,V and one of the images
    '''

    # Change t if required, affects the number of arrows
    # t should be between 1 and min(U.shape[0],U.shape[1])
    t= 10

    # Subsample U and V to get visually pleasing output
    U1 = U[::t,::t]
    V1 = V[::t,::t]

    # Create meshgrid of subsampled coordinates
    r, c = img.shape[0],img.shape[1]
    cols,rows = np.meshgrid(np.linspace(0,c-1,c), np.linspace(0,r-1,r))
    cols = cols[::t,::t]
    rows = rows[::t,::t]

    # Plot optical flow
    plt.figure(figsize=(10,10))
    plt.imshow(img)
    plt.quiver(cols,rows,U1,V1)
    plt.show()

images=[]
for i in range(1,5):
    images.append(plt.imread('optical_flow_images/im'+str(i)+'.png'))
```

In [65]:
```
fig=plt.figure(figsize=(8, 8))
for idx, val in enumerate([0,3]):
    ax1 = fig.add_subplot(1, 2, idx + 1)
    title = "image %d"%(idx + 1)
    ax1.title.set_text(title)
    plt.imshow(images[val])
plt.show()
```

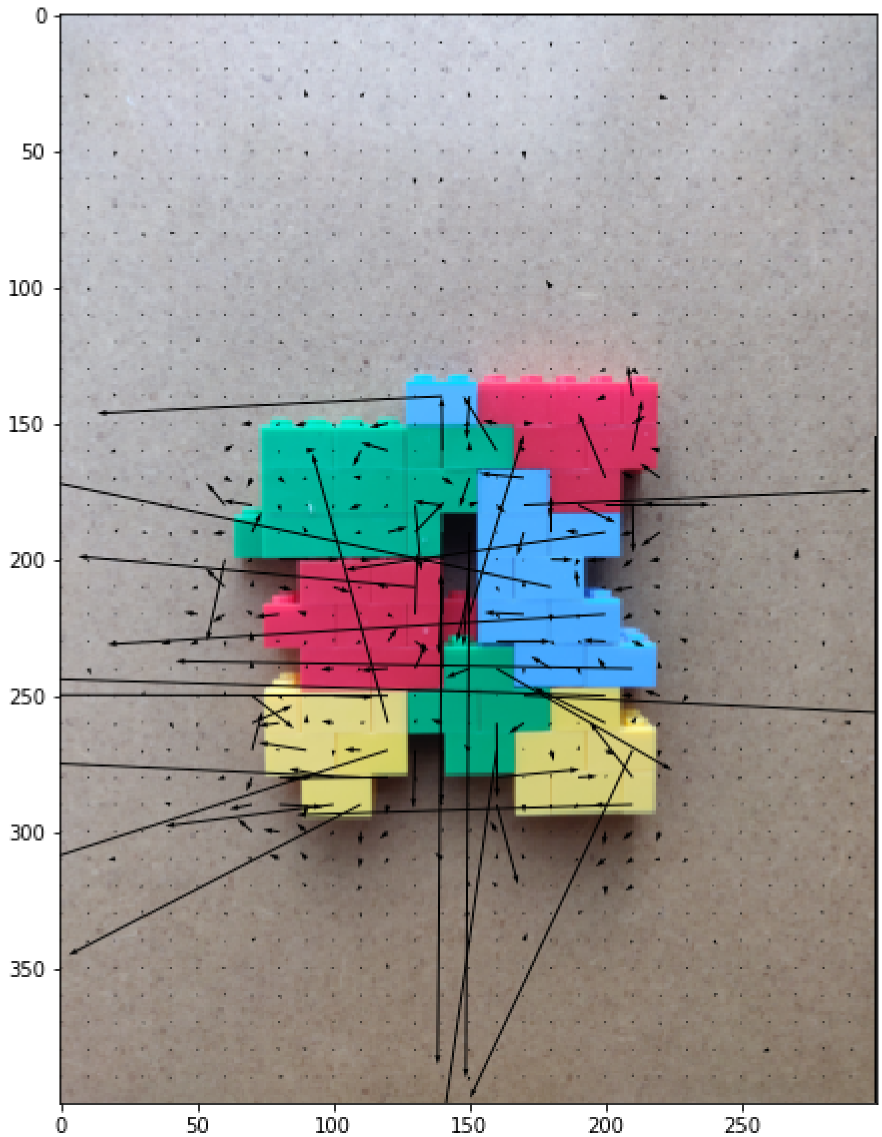


In [41]:
```
def LucasKanade(im1,im2,window):
    '''
    Inputs: the two images and window size
    Return U,V
    '''
    U = np.zeros(im1.shape)
    V = np.zeros(im1.shape)
    '''
    Your code here
    '''
    winRowHalf = int((window - 1) / 2)
    winColHalf = int((window - 1) / 2)
    #Iy, Ix = np.gradient(im1)
    dx = np.array([[ -1,  0,  1],[ -1,  0,  1],[-1,0,1]])
    dy = dx.T
    dx = np.flip(dx, axis = 0)
    dx = np.flip(dx, axis = 1)
    dy = np.flip(dy, axis = 0)
    dy = np.flip(dy, axis = 1)

    Ix = conv2(im1, dx, mode='same')
    Iy = conv2(im1, dy, mode='same')
    It = im2 - im1
    window = np.ones([window, window])
    Ix_2 = findC(Ix * Ix, window) # Ix^2
    Iy_2 = findC(Iy * Iy, window) # Iy^2
    Ixy_2 = findC(Ix * Iy, window) #IxIy
    Ixt = findC(Ix * It, window) # IxIt
    Iyt = findC(Iy * It, window) # IyIt

    for i in xrange(winRowHalf, im1.shape[0]-winRowHalf,):
        for j in xrange(winColHalf, im1.shape[1]-winColHalf):
            A = np.array([[Ix_2[i][j], Ixy_2[i][j]],[Ixy_2[i][j], Iy_2[i
][j]]])
            b = -np.array([[Ixt[i][j]], [Iyt[i][j]]])
            A_p = np.matmul(np.linalg.inv(np.matmul(A.T, A)), A.T)
            U[i][j] = np.matmul(A_p, b)[0]
            V[i][j] = np.matmul(A_p, b)[1]
    return U, V


def findC(img, window):
    winRowHalf = int((window.shape[0]- 1) / 2)
    winColHalf = int((window.shape[1] - 1) / 2)
    Row = img.shape[0]
    Col = img.shape[1]
    res = np.zeros((Row, Col))
    for i in xrange(winRowHalf, Row - winRowHalf):
        for j in xrange(winColHalf, Col - winColHalf):
            res[i][j] = (img[i - winRowHalf : i + winRowHalf+1, j - winR
owHalf : j + winColHalf+1] * window).sum()

    return res
```

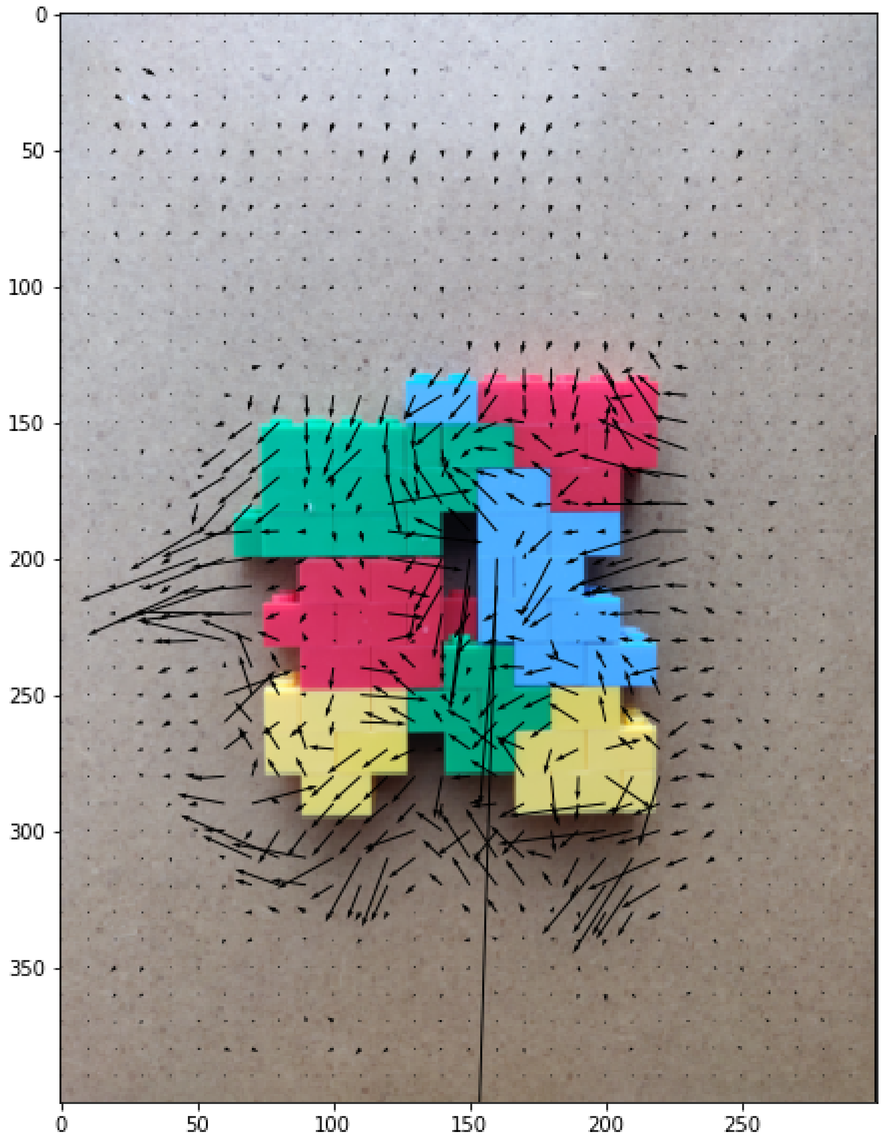
Part 2: Window size [2 pts]

Plot optical flow for the pair of images im1 and im2 for at least 3 different window sizes which leads to observable difference in the results. Comment on the effect of window size on results and justify.

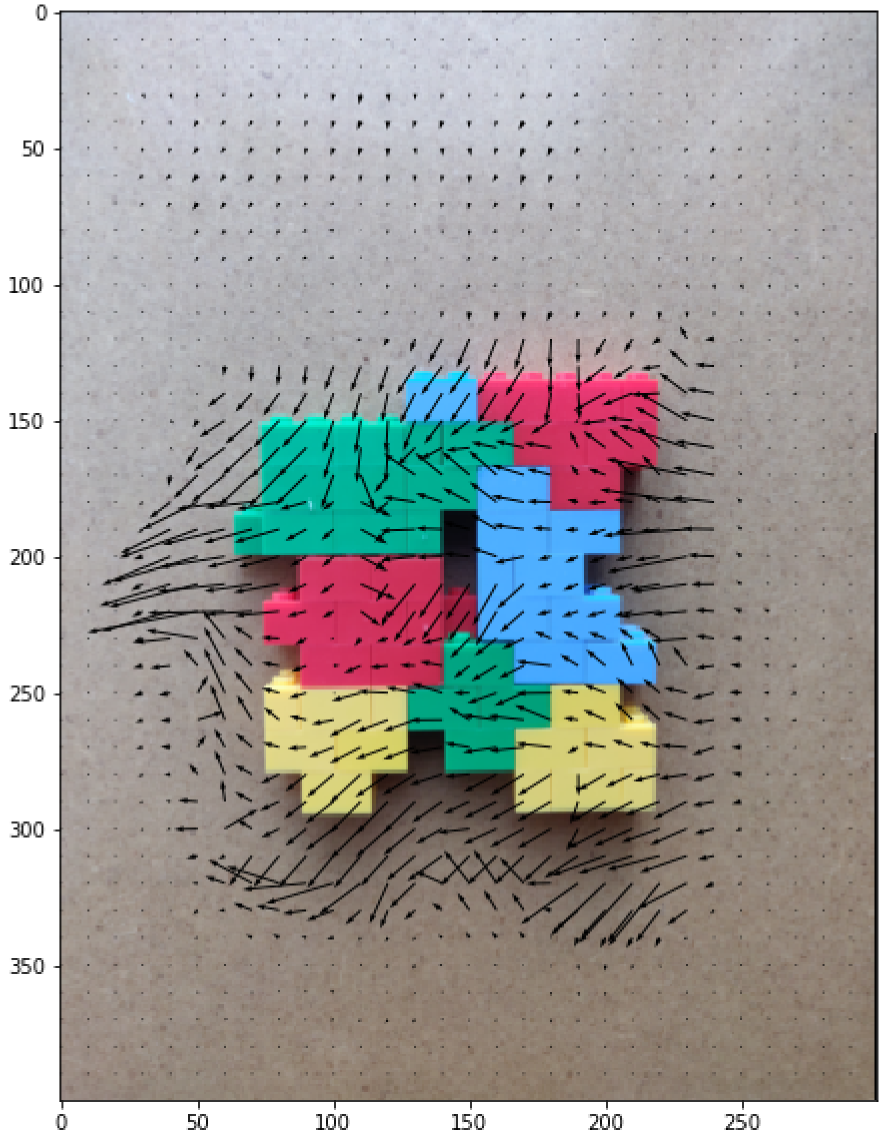
In [42]:
```
# Example code, change as required
window=5
U,V=LucasKanade(grayscale(images[0]),grayscale(images[1]),window)
plot_optical_flow(images[0],U,V)
```



In [47]:
```
# Example code, change as required
window=25
U,V=LucasKanade(grayscale(images[0]),grayscale(images[1]),window)
plot_optical_flow(images[0],U,V)
```



In [46]:
```
# Example code, change as required
window=45
U,V=LucasKanade(grayscale(images[0]),grayscale(images[1]),window)
plot_optical_flow(images[0],U,V)
```

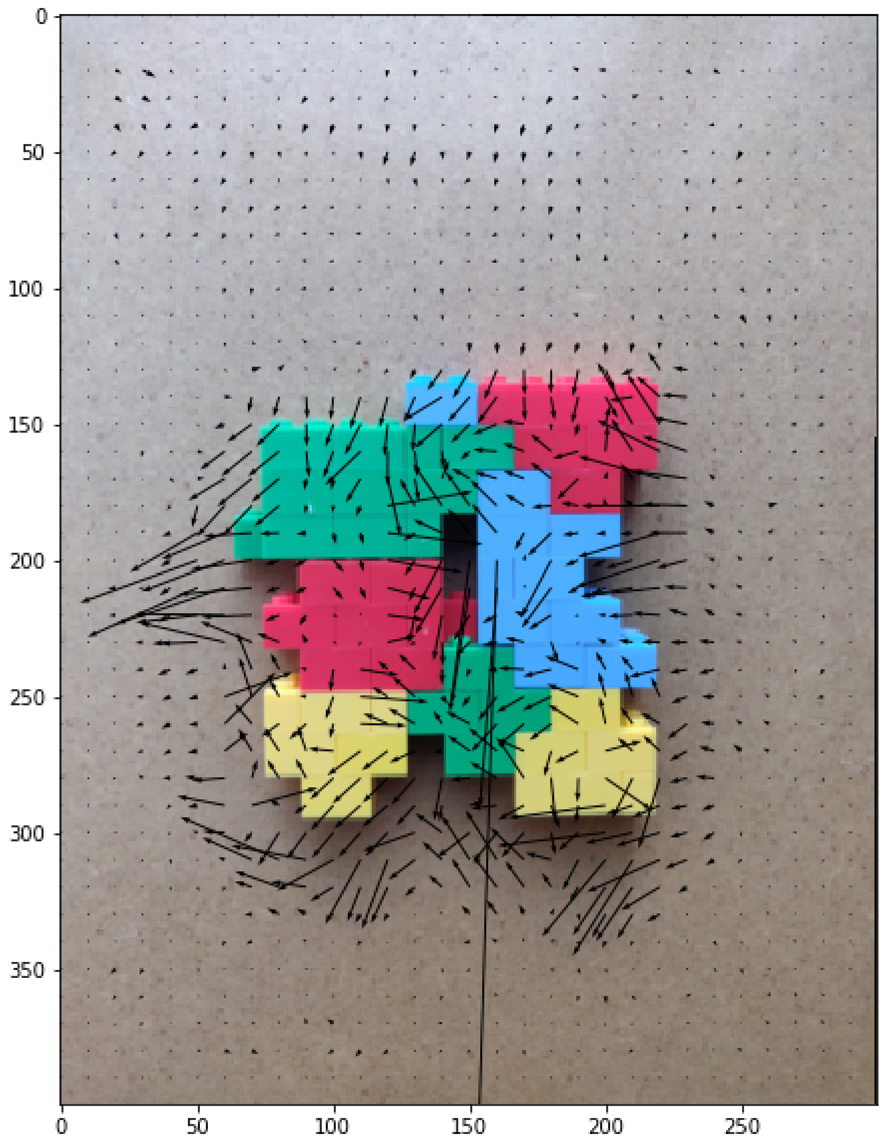


When the window size is small, the length of arrows are long and not consistent with the actual movement. When the window size is as big as 25, we can see the number of arrows presented increase and the length of arrows decreases. In addition, these arrows are consistent with the actual movement. But if the widnow size is too big, the movement of the background is also presented, which can be annoying if we only concern about the motion of the object. Therefore, we should choose the appropriate window size.

Part 3: All pairs [3 pts]

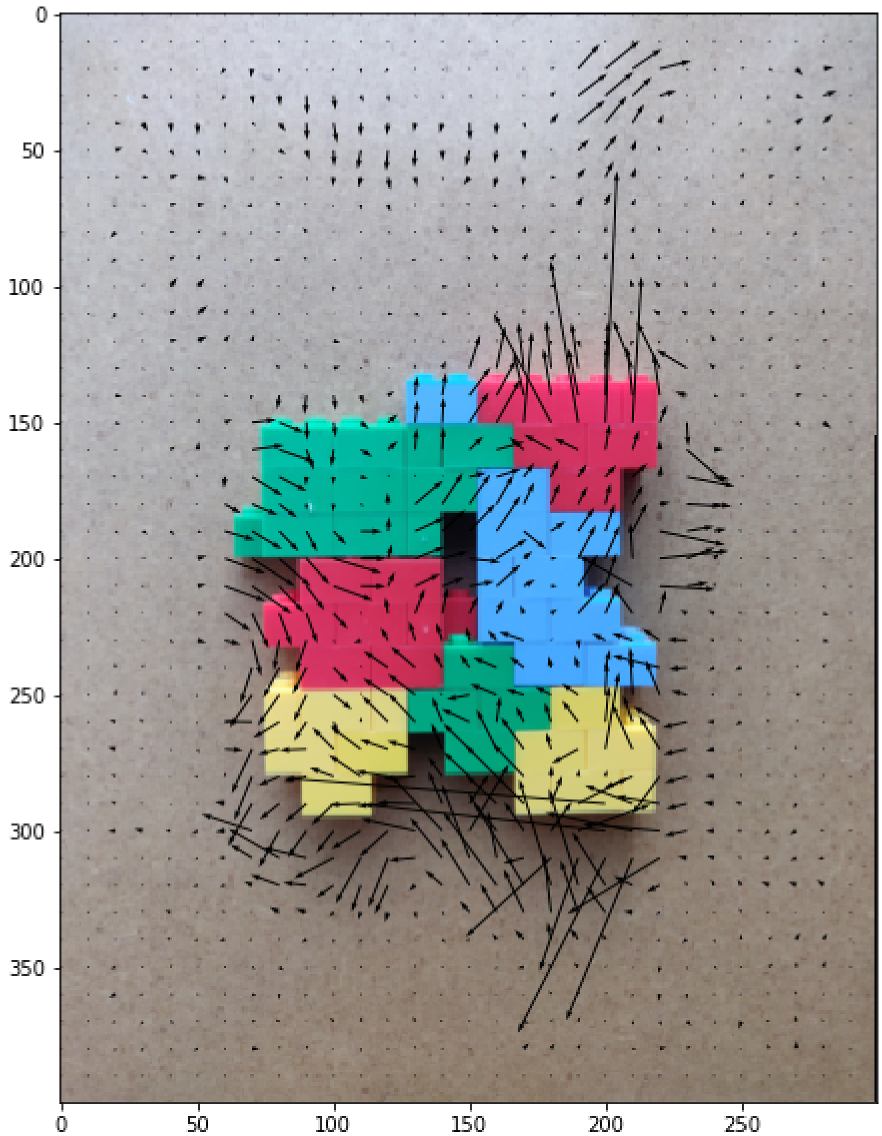
Find optical flow for the pairs (im1,im2), (im1,im3), (im1,im4) using a good window size. Does the optical flow result seem consistent with visual inspection? Comment on the type of motion indicated by results and visual inspection and explain why they might be consistent or inconsistent.

In [59]:
```
# Your code here
## pairs(im1, im2)
window = 25
U,V=LucasKanade(grayscale(images[0]),grayscale(images[1]),window)
plot_optical_flow(images[0],U,V)
```



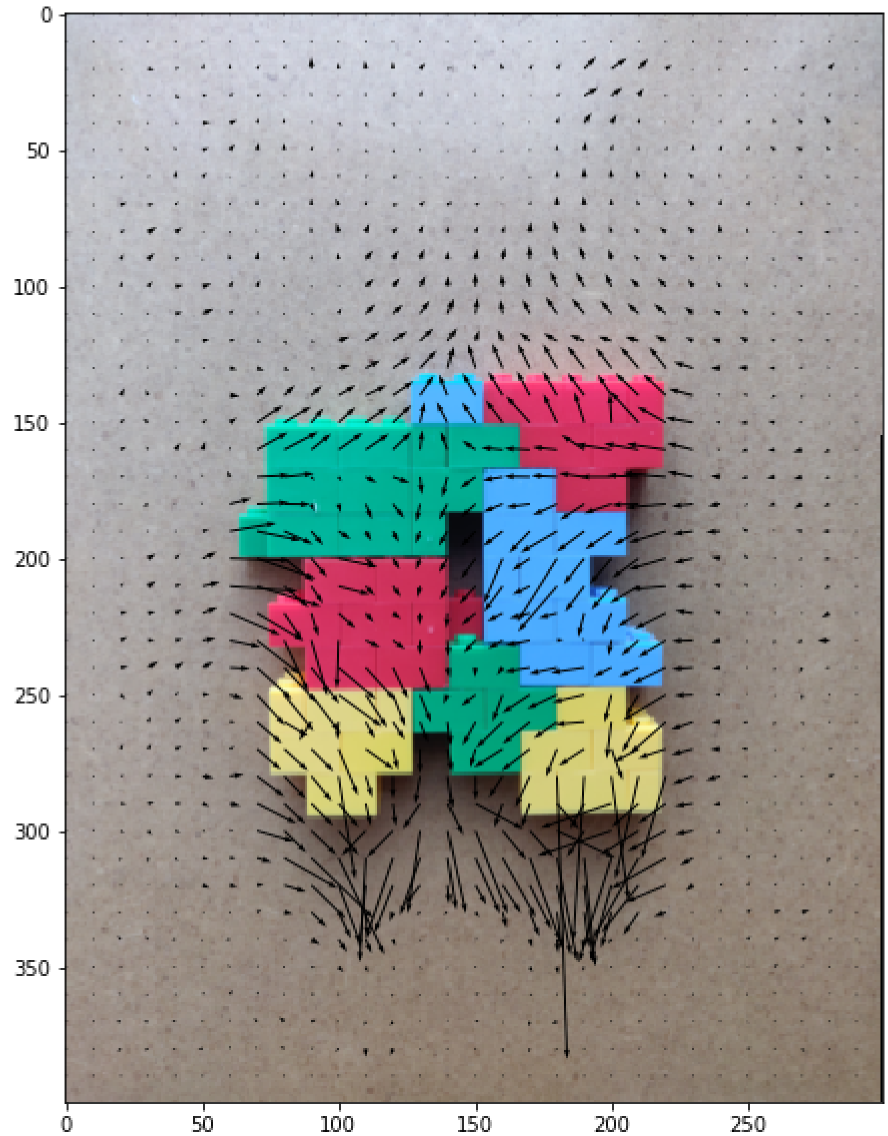
Answer: According to the optical flow result, the blocks are moving to the left and moving down, which is showing translation. The visual inspection shows the blocks are moving to the left. In addtion, a small number of arrows are trying to go up. The reason might be the optical flow result demonstrates more details and is better than human eyes.

In [63]:
```
## pairs(im1, im3)
window = 25
U,V=LucasKanade(grayscale(images[0]),grayscale(images[2]),window)
plot_optical_flow(images[0],U,V)
```



Answer: According to the optical flow result, the blocks contradicted themselves. Some are showing clockwise movement, while some are showing counterclockwise movement. I cannot expect the movement from the optical flows. The visual inspection shows the blocks are rotating clockwise. The reason might be the choices of the points.

In [64]:
```
## pairs(im1, im4)
window = 25
U,V=LucasKanade(grayscale(images[0]),grayscale(images[3]),window)
plot_optical_flow(images[0],U,V)
```



Answers: According the optical flow result, the object is shrinking. The visual inspection, given im1 and im4, shows the object is zooming out. So the optical flow result is consistent wit the visual inspection. But there are several arrows on the top and the bottom of the object. The result is that it takes the background into consideration.

Problem 2: Machine Learning [12 pts]

In this problem, you will implement several machine learning solutions for computer vision problems.

Part 1: Initial setup [1 pts]

Follow the directions on https://www.tensorflow.org/install/ (https://www.tensorflow.org/install/) to install Tensorflow on your computer. If you are using the Anaconda distribution for python, you can check out https://www.anaconda.com/blog/developer-blog/tensorflow-in-anaconda/ (https://www.anaconda.com/blog/developer-blog/tensorflow-in-anaconda/).

Note: You will not need GPU support for this assignment so don't worry if you don't have one. Furthermore, installing with GPU support is often more difficult to configure so it is suggested that you install the CPU only version.

Run the tensorflow hello world snippet below to verify your instalation.

Download the MNIST data from http://yann.lecun.com/exdb/mnist/ (http://yann.lecun.com/exdb/mnist/).

Download the 4 zipped files, extract them into one folder, and change the variable 'path' in the code below. (Code taken from https://gist.github.com/akesling/5358964 (https://gist.github.com/akesling/5358964))

Plot one random example image corresponding to each label from training data.

In [3]:
```
import tensorflow as tf
hello = tf.constant('Hello, TensorFlow!')
sess = tf.Session()
print(sess.run(hello))
```

```
/Users/huangzhisheng/anaconda2/lib/python2.7/site-packages/h5py/__init_
_.py:36: FutureWarning: Conversion of the second argument of issubdtype
from `float` to `np.floating` is deprecated. In future, it will be trea
ted as `np.float64 == np.dtype(float).type`.
  from ._conv import register_converters as _register_converters

Hello, TensorFlow!
```

In [6]:
```
import os
import struct

# Change path as required
#path = "./mnist_data/"
path = "/Users/huangzhisheng/Desktop/FA18_ucsd/CSE252/HW4/HW4/mnist_dat
a"

def read(dataset = "training", datatype='images'):
    """
    Python function for importing the MNIST data set.  It returns an ite
rator
    of 2-tuples with the first element being the label and the second el
ement
    being a numpy.uint8 2D array of pixel data for the given image.
    """

    if dataset is "training":
        fname_img = os.path.join(path, 'train-images-idx3-ubyte')
        fname_lbl = os.path.join(path, 'train-labels-idx1-ubyte')
    elif dataset is "testing":
        fname_img = os.path.join(path, 't10k-images-idx3-ubyte')
        fname_lbl = os.path.join(path, 't10k-labels-idx1-ubyte')

    # Load everything in some numpy arrays
    with open(fname_lbl, 'rb') as flbl:
        magic, num = struct.unpack(">II", flbl.read(8))
        lbl = np.fromfile(flbl, dtype=np.int8)

    with open(fname_img, 'rb') as fimg:
        magic, num, rows, cols = struct.unpack(">IIII", fimg.read(16))
        img = np.fromfile(fimg, dtype=np.uint8).reshape(len(lbl), rows,
cols)

    if(datatype=='images'):
        get_data = lambda idx: img[idx]
    elif(datatype=='labels'):
        get_data = lambda idx: lbl[idx]

    # Create an iterator which returns each image in turn
    for i in range(len(lbl)):
        yield get_data(i)

trainData=np.array(list(read('training','images')))
trainLabels=np.array(list(read('training','labels')))
testData=np.array(list(read('testing','images')))
testLabels=np.array(list(read('testing','labels')))
```

Some helper functions are given below.

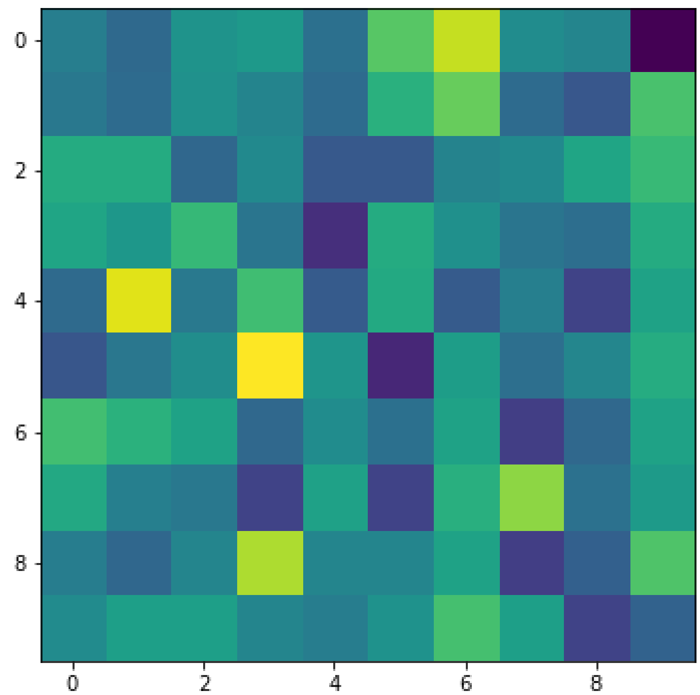
In [8]:
```
# a generator for batches of data
# yields data (batchsize, 3, 32, 32) and labels (batchsize)
# if shuffle, it will load batches in a random order
def DataBatch(data, label, batchsize, shuffle=True):
    n = data.shape[0]
    if shuffle:
        index = np.random.permutation(n)
    else:
        index = np.arange(n)
    for i in range(int(np.ceil(n/batchsize))):
        inds = index[i*batchsize : min(n,(i+1)*batchsize)]
        yield data[inds], label[inds]

# tests the accuracy of a classifier
def test(testData, testLabels, classifier):
    batchsize=50
    correct=0.
    for data,label in DataBatch(testData,testLabels,batchsize,shuffle=Fa
lse):
        prediction = classifier(data)
        correct += np.sum(prediction==label)
    return correct/testData.shape[0]*100

# a sample classifier
# given an input it outputs a random class
class RandomClassifier():
    def __init__(self, classes=10):
        self.classes=classes
    def __call__(self, x):
        return np.random.randint(self.classes, size=x.shape[0])

randomClassifier = RandomClassifier()
print('Random classifier accuracy: %f' %
      test(testData, testLabels, randomClassifier))
```

```
Random classifier accuracy: 9.930000
```

Part 2: Confusion Matrix [2 pts]

Here you will implement a function that computes the confusion matrix for a classifier. The matrix (M) should be nxn where n is the number of classes. Entry M[i,j] should contain the fraction of images of class i that was classified as class j.

In [36]:
```
# Using the tqdm module to visualize run time is suggested
# from tqdm import tqdm

# It would be a good idea to return the accuracy, along with the confusi
on
# matrix, since both can be calculated in one iteration over test data,
 to
# save time
def Confusion(testData, testLabels, classifier):
    '''
    Your code here
    '''
    M = np.zeros((10, 10))
    batchsize = 50
    for data, label in DataBatch(testData, testLabels, batchsize):
        prediction = classifier(data)
        M[label, prediction] += 1
    M /= M.sum(axis = 1)[:, np.newaxis]
    return M

def VisualizeConfusion(M):
    plt.figure(figsize=(14, 6))
    plt.imshow(M)
    plt.show()
    print(np.round(M,2))

M = Confusion(testData, testLabels, randomClassifier)
VisualizeConfusion(M)
```




```
[[0.1  0.09 0.1  0.1  0.1  0.11 0.12 0.1  0.1  0.08]
 [0.1  0.09 0.1  0.1  0.09 0.11 0.11 0.09 0.09 0.11]
 [0.11 0.11 0.09 0.1  0.09 0.09 0.1  0.1  0.1  0.11]
 [0.1  0.1  0.11 0.1  0.08 0.11 0.1  0.1  0.09 0.11]
 [0.09 0.12 0.1  0.11 0.09 0.11 0.09 0.1  0.09 0.1 ]
 [0.09 0.1  0.1  0.12 0.1  0.08 0.1  0.09 0.1  0.11]
 [0.11 0.11 0.1  0.09 0.1  0.1  0.1  0.09 0.09 0.1 ]
 [0.11 0.1  0.1  0.09 0.1  0.09 0.11 0.12 0.1  0.1 ]
 [0.1  0.09 0.1  0.12 0.1  0.1  0.1  0.09 0.09 0.11]
 [0.1  0.1  0.1  0.1  0.1  0.1  0.11 0.1  0.09 0.09]]
```

Part 3: K-Nearest Neighbors (KNN) [4 pts]

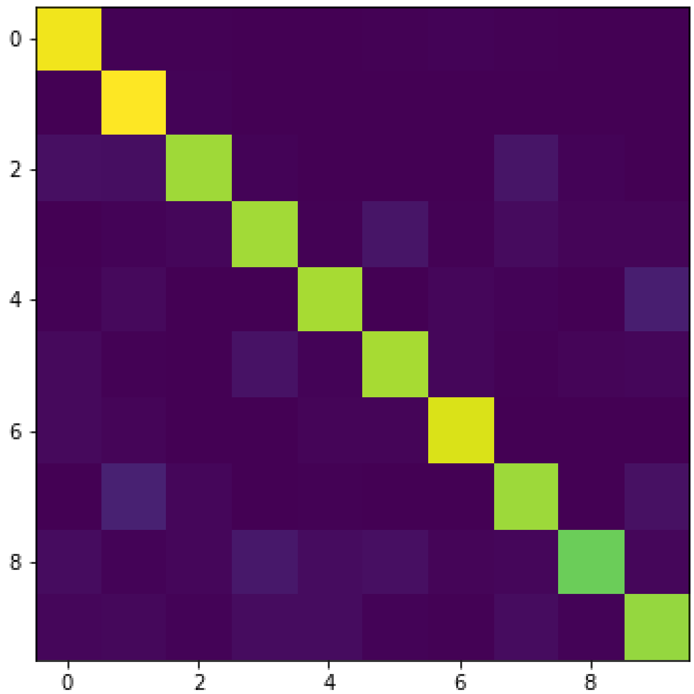
- Here you will implement a simple knn classifier. The distance metric is Euclidean in pixel space. k refers to the number of neighbors involved in voting on the class, and should be 3. You are allowed to use sklearn.neighbors.KNeighborsClassifier.
- Display confusion matrix and accuracy for your KNN classifier trained on the entire train set. (should be ~97 %)
- After evaluating the classifier on the testset, based on the confusion matrix, mention the number that the number '4' is most often predicted to be, other than '4'.

In [47]:
```
from sklearn.neighbors import KNeighborsClassifier
class KNNClassifer():
    def __init__(self, k=3):
        # k is the number of neighbors involved in voting
        '''
        your code here
        '''
        self.k = k

    def train(self, trainData, trainLabels):
        '''
        your code here
        '''
        self.X_train = np.reshape(trainData, (trainData.shape[0], -1))
        self.y_train = trainLabels
        self.knn = KNeighborsClassifier(n_neighbors = self.k) # use exis
ting function for k-means clustering
        self.knn.fit(self.X_train, self.y_train) # generate the k-mean c
lassifier

    def __call__(self, x):
        # this method should take a batch of images
        # and return a batch of predictions
        '''
        your code here
        '''
        X = np.reshape(x, (x.shape[0], -1))
        prediction = self.knn.predict(X)
        return prediction


# test your classifier with only the first 100 training examples (use th
is
# while debugging)
# note you should get ~ 65 % accuracy
knnClassiferX = KNNClassifer()
knnClassiferX.train(trainData[:100], trainLabels[:100])
print ('KNN classifier accuracy: %f'%test(testData, testLabels, knnClass
iferX))
```

```
KNN classifier accuracy: 64.760000
```

In [48]:
```
# test your classifier with all the training examples (This may take a w
hile)
knnClassifer = KNNClassifer()
knnClassifer.train(trainData, trainLabels)
print ('KNN classifier accuracy: %f'%test(testData, testLabels, knnClass
ifer))
# display confusion matrix for your KNN classifier with all the training
 examples
M = Confusion(testData, testLabels, knnClassifer)
VisualizeConfusion(M)
```

```
KNN classifier accuracy: 97.050000
```




```
[[0.97 0.   0.   0.   0.   0.   0.01 0.   0.   0.  ]
 [0.   0.99 0.01 0.   0.   0.   0.   0.   0.   0.  ]
 [0.04 0.04 0.85 0.01 0.   0.   0.   0.06 0.01 0.  ]
 [0.   0.01 0.02 0.85 0.   0.06 0.   0.03 0.01 0.01]
 [0.   0.03 0.   0.   0.86 0.   0.02 0.01 0.   0.08]
 [0.03 0.   0.   0.05 0.01 0.86 0.02 0.   0.01 0.02]
 [0.02 0.01 0.   0.   0.01 0.01 0.93 0.   0.   0.  ]
 [0.   0.09 0.02 0.   0.   0.   0.   0.85 0.   0.04]
 [0.03 0.01 0.02 0.06 0.03 0.04 0.01 0.02 0.77 0.02]
 [0.02 0.02 0.01 0.03 0.03 0.01 0.   0.03 0.01 0.83]]
```

Answer: Based on the confusion matrix, the number that the number '4' is most often predicted to be, other than '4' is '9'.

Part 4: Principal Component Analysis (PCA) K-Nearest Neighbors (KNN) [5 pts]

Here you will implement a simple KNN classifer in PCA space (for k=3 and 25 principal components). You should implement PCA yourself using svd (you may not use sklearn.decomposition.PCA or any other package that directly implements PCA transformations

Is the testing time for PCA KNN classifier more or less than that for KNN classifier? Comment on why it differs if it does.

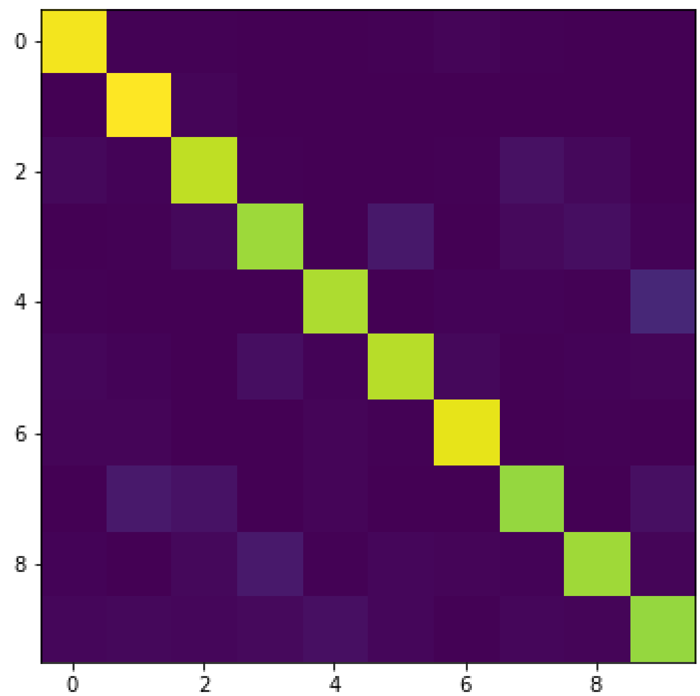
In [56]:

```
class PCAKNNClassifer():
    def __init__(self, components=25, k=3):
        # components = number of principal components
        # k is the number of neighbors involved in voting
        '''
        your code here
        '''
        self.components = components
        self.k = k

    def train(self, trainData, trainLabels):
        '''
        your code here
        '''
        self.X_train = np.reshape(trainData, (trainData.shape[0], -1))
        self.y_train = trainLabels
        covX_train = np.cov(self.X_train.T)
        U, S, V = np.linalg.svd(covX_train)
        self.ppca = V[:self.components + 1].T
        self.X_trainPCA = np.dot(self.X_train, self.ppca)
        self.knn = KNeighborsClassifier(n_neighbors = self.k, weights =
'uniform') # use existing function for k-means clustering
        self.knn.fit(self.X_trainPCA, self.y_train) # generate the k-mea
n classifier

    def __call__(self, x):
        # this method should take a batch of images
        # and return a batch of predictions
        '''
        your code here
        '''
        X = np.reshape(x, (x.shape[0],-1))
        X_pca = np.dot(X, self.ppca)
        prediction = self.knn.predict(X_pca)
        return prediction


# test your classifier with only the first 100 training examples (use th
is
# while debugging)
pcaknnClassiferX = PCAKNNClassifer()
pcaknnClassiferX.train(trainData[:100], trainLabels[:100])
print ('KNN classifier accuracy: %f'%test(testData, testLabels, pcaknnCl
assiferX))
```

```
KNN classifier accuracy: 65.940000
```

In [57]:
```
# test your classifier with all the training examples (This may take a w
hile)
pcaknnClassifer = PCAKNNClassifer()
pcaknnClassifer.train(trainData, trainLabels)
print ('KNN classifier accuracy: %f'%test(testData, testLabels, pcaknnCl
assifer))
# display confusion matrix for your PCA KNN classifier with all the trai
ning examples
M = Confusion(testData, testLabels, pcaknnClassifer)
VisualizeConfusion(M)
```

```
KNN classifier accuracy: 97.400000
```




```
[[0.97 0.   0.   0.   0.   0.   0.01 0.   0.   0.  ]
 [0.   0.99 0.01 0.   0.   0.   0.   0.   0.   0.  ]
 [0.02 0.01 0.89 0.   0.   0.   0.   0.05 0.02 0.  ]
 [0.   0.   0.02 0.84 0.   0.06 0.   0.03 0.04 0.01]
 [0.   0.   0.   0.   0.87 0.   0.01 0.01 0.   0.11]
 [0.02 0.01 0.   0.04 0.01 0.88 0.02 0.   0.01 0.01]
 [0.01 0.01 0.   0.   0.01 0.   0.95 0.   0.   0.  ]
 [0.   0.07 0.05 0.   0.01 0.   0.   0.83 0.   0.04]
 [0.01 0.   0.02 0.07 0.   0.02 0.01 0.01 0.84 0.01]
 [0.02 0.02 0.02 0.03 0.04 0.02 0.   0.02 0.01 0.83]]
```

Answer: The testing time for PCA KNN classifier is less than that for KNN classifier. PCA KNN classifier did not the full training data, instead it reduces the demensions, further decreasing the number of multiplication-accumulation operations. So it run fast.

Problem 3: Deep learning [12 pts]

Below is some helper code to train your deep networks. You can look at [https://www.tensorflow.org/get_started/mnist/beginners (https://www.tensorflow.org/get_started/mnist/beginners)](https://www.tensorflow.org/get_started/mnist/beginners) for reference.

In [32]:
```
# base class for your Tensorflow networks. It implements the training lo
op
# (train) and prediction(__call__)  for you.
# You will need to implement the __init__ function to define the network
s
# structures in the following problems.

class TFClassifier():
    def __init__(self):
        pass

    def train(self, trainData, trainLabels, epochs=1, batchsize=50):
        self.prediction = tf.argmax(self.y,1)
        self.cross_entropy = tf.reduce_mean(tf.nn.sparse_softmax_cross_e
ntropy_with_logits(labels=self.y_, logits=self.y))
        self.train_step = tf.train.AdamOptimizer(1e-4).minimize(self.cro
ss_entropy)
        self.correct_prediction = tf.equal(self.prediction, self.y_)
        self.accuracy = tf.reduce_mean(tf.cast(self.correct_prediction,
tf.float32))
        self.sess.run(tf.global_variables_initializer())

        for epoch in range(epochs):
            for i, (data,label) in enumerate(DataBatch(trainData, trainL
abels, batchsize, shuffle=True)):
                data=np.expand_dims(data,-1)
                _, acc = self.sess.run([self.train_step, self.accuracy],
 feed_dict={self.x: data, self.y_: label})

            print ('Epoch:%d Accuracy: %f'%(epoch+1, test(testData, test
Labels, self)))

    def __call__(self, x):
        return self.sess.run(self.prediction, feed_dict={self.x: np.expa
nd_dims(x,-1)})

    def get_first_layer_weights(self):
        return self.sess.run(self.weights[0])

# helper function to get weight variable
def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.01)
    return tf.Variable(initial)

# helper function to get bias variable
def bias_variable(shape):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)

# example linear classifier
class LinearClassifier(TFClassifier):
    def __init__(self, classes=10):
        self.sess = tf.Session()

        self.x = tf.placeholder(tf.float32, shape=[None,28,28,1]) # inpu
t batch of images
```

```
        self.y_ = tf.placeholder(tf.int64, shape=[None]) # input labels

        # model variables
        self.weights = [weight_variable([28*28,classes])]
        self.biases = [bias_variable([classes])]

        # linear operation
        self.y = tf.matmul(tf.reshape(self.x,(-1,28*28*1)),self.weights[
0]) + self.biases[0]


# test the example linear classifier (note you should get around 90% acc
uracy
# for 10 epochs and batchsize 50)
linearClassifier = LinearClassifier()
linearClassifier.train(trainData, trainLabels, epochs=10)
weights = linearClassifier.get_first_layer_weights()
# weights = np.reshape(weights, (28, 28, -1))
# print(weights.shape)
```

```
Epoch:1 Accuracy: 88.470000
Epoch:2 Accuracy: 88.910000
Epoch:3 Accuracy: 88.600000
Epoch:4 Accuracy: 89.650000
Epoch:5 Accuracy: 89.550000
Epoch:6 Accuracy: 89.570000
Epoch:7 Accuracy: 89.640000
Epoch:8 Accuracy: 90.290000
Epoch:9 Accuracy: 88.350000
Epoch:10 Accuracy: 88.790000
```

In [30]:
```
print(weights.shape)
```

```
(784, 10)
```

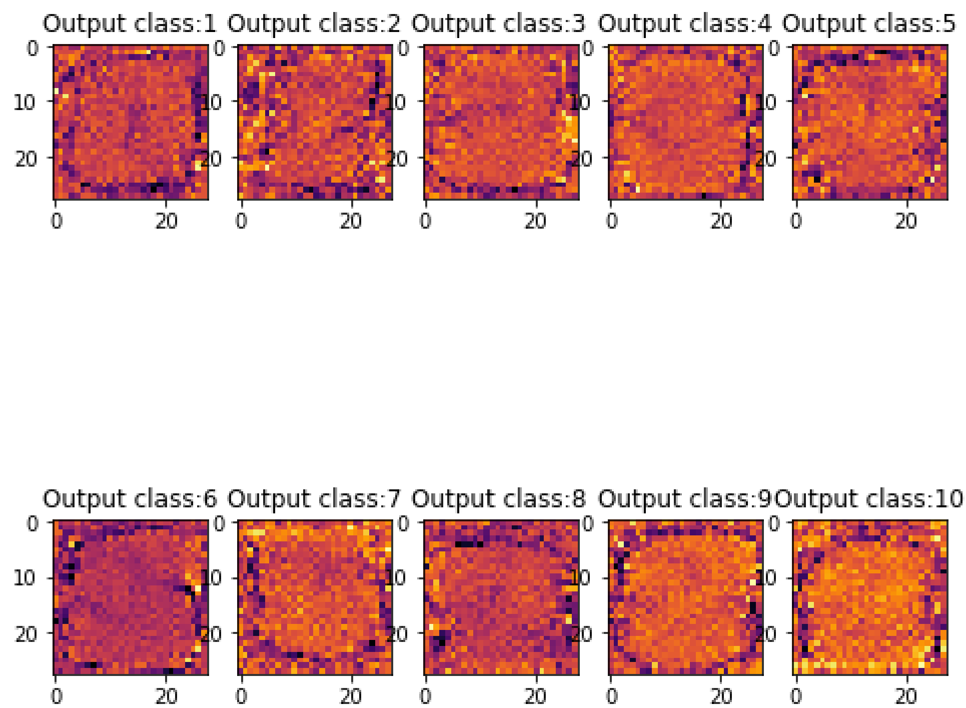
Part 1: Single Layer Perceptron [2 pts]

The simple linear classifier implemented in the cell already performs quite well. Plot the filter weights corresponding to each output class (weights, not biases) as images. (Normalize weights to lie between 0 and 1 and use color maps like 'inferno' or 'plasma' for good results). Comment on what the weights look like and why that may be so.

In [33]:

```
# M /= M.sum(axis = 1)[:, np.newaxis]
import matplotlib.pyplot as plt
max_list = []
min_list = []
norm_weights = np.ones(weights.shape)
for idx in xrange(10):
    max_list.append(weights[:, idx].max())
    min_list.append(weights[:, idx].min())
    norm_weights[:,idx] = (weights[:,idx] - min_list[idx])/(max_list[idx
] - min_list[idx])

norm_weights = np.reshape(norm_weights, (28, 28, -1))
print(norm_weights.shape)
fig=plt.figure(figsize=(8, 8))

for idx in xrange(10):
    ax1 = fig.add_subplot(2, 5, idx + 1)
    title = "Output class:%d"%(idx + 1)
    ax1.title.set_text(title)
    plt.imshow(norm_weights[:,:,idx],'inferno')
#plt.colorbar()
plt.show()
```

```
(28, 28, 10)
```



Answer: Based on the results, there is a circle-like shape with lower weight values. All the weight maps seem too blurry. But they could look like somewhat blurry digits from 0 to 9, but now they are too blurry. All these weight maps are different from each other. Each represents a special pattern for one digit.

Part 2: Multi Layer Perceptron (MLP) [5 pts]

Here you will implement an MLP. The MLP shoud consist of 2 layers (matrix multiplication and bias offset) that map to the following feature dimensions:

- 28x28 -> hidden (100)
- hidden -> classes
- The hidden layer should be followed with a ReLU nonlinearity. The final layer should not have a nonlinearity applied as we desire the raw logits output.
- The final output of the computation graph should be stored in self.y as that will be used in the training.

Display the confusion matrix and accuracy after training. Note: You should get ~ 97 % accuracy for 10 epochs and batch size 50.

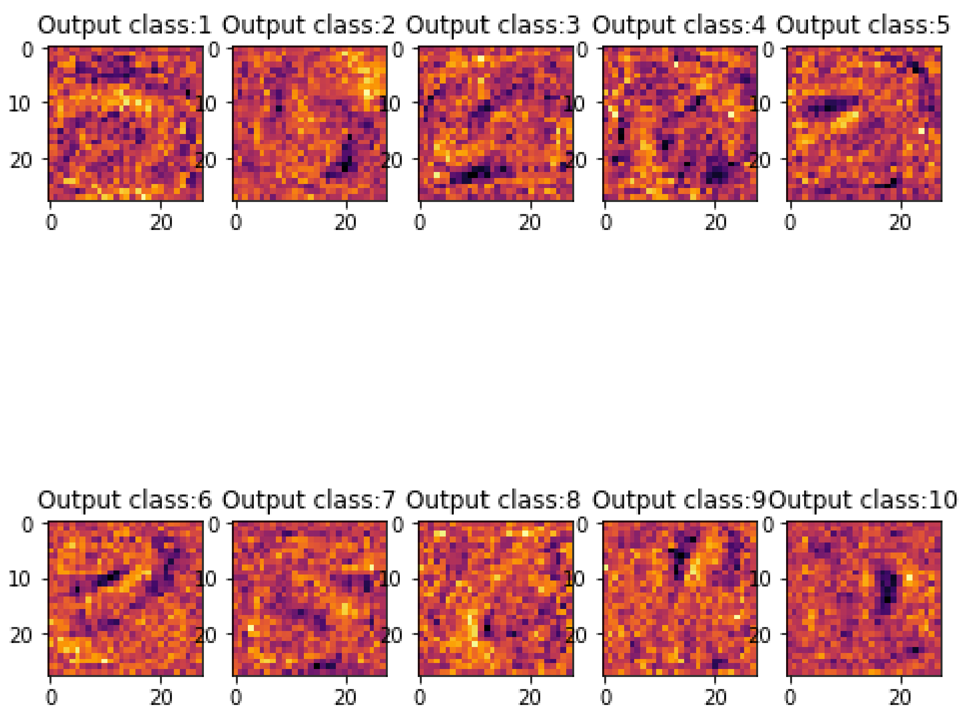
Plot the filter weights corresponding to the mapping from the inputs to the first 10 hidden layer outputs (out of 100). Do the weights look similar to the weights plotted in the previous problem? Why or why not?

In [20]:
```
class MLPClassifer(TFClassifier):
    def __init__(self, classes=10, hidden=100):
        '''
        your code here
        '''
        self.sess = tf.Session()

        self.x = tf.placeholder(tf.float32, shape=[None,28,28,1]) # inpu
t batch of images
        self.y_ = tf.placeholder(tf.int64, shape=[None]) # input labels
        # model variables
        self.weights = [weight_variable([28*28, hidden]),
                        weight_variable([hidden, classes])]
        self.biases = [ bias_variable([hidden]),
                        bias_variable([classes])]
        # linear operation
        # Hidden layer with RELU activation
        layer1 = tf.matmul(tf.reshape(self.x,(-1, 28*28*1)), self.weight
s[0]) + self.biases[0]
        layer1 = tf.nn.relu(layer1)
        # output layer
        self.y = tf.matmul(layer1, self.weights[1]) + self.biases[1]

mlpClassifier = MLPClassifer()
mlpClassifier.train(trainData, trainLabels, epochs=10)
```

```
Epoch:1 Accuracy: 95.900000
Epoch:2 Accuracy: 96.890000
Epoch:3 Accuracy: 96.630000
Epoch:4 Accuracy: 97.430000
Epoch:5 Accuracy: 97.720000
Epoch:6 Accuracy: 97.530000
Epoch:7 Accuracy: 97.360000
Epoch:8 Accuracy: 97.420000
Epoch:9 Accuracy: 97.300000
Epoch:10 Accuracy: 97.290000
```

In [34]:
```
weights = mlpClassifier.get_first_layer_weights()
weights = np.reshape(weights, (28, 28, -1))
#print(weights.shape)
idx = range(10)
max_list = []
min_list = []
norm_weights = np.ones(weights.shape)
for idx in xrange(10):
    max_list.append(weights[:, :, idx].max())
    min_list.append(weights[:, :, idx].min())
    norm_weights[:,:,idx] = (weights[:,:,idx] - min_list[idx])/(max_list
[idx] - min_list[idx])

print(norm_weights.shape)
fig=plt.figure(figsize=(8, 8))

for idx in xrange(10):
    ax1 = fig.add_subplot(2, 5, idx + 1)
    title = "Output class:%d"%(idx + 1)
    ax1.title.set_text(title)
    plt.imshow(norm_weights[:,:,idx], 'inferno')
plt.show()
```

```
(28, 28, 100)
```



Answer: These weight maps did not look like image class templates. Some of them look like pseudo-digits, others seem componets of digits. The first layer is one of the layers. The weight maps contain several features from different digits.

Part 3: Convolutional Neural Network (CNN) [5 pts]

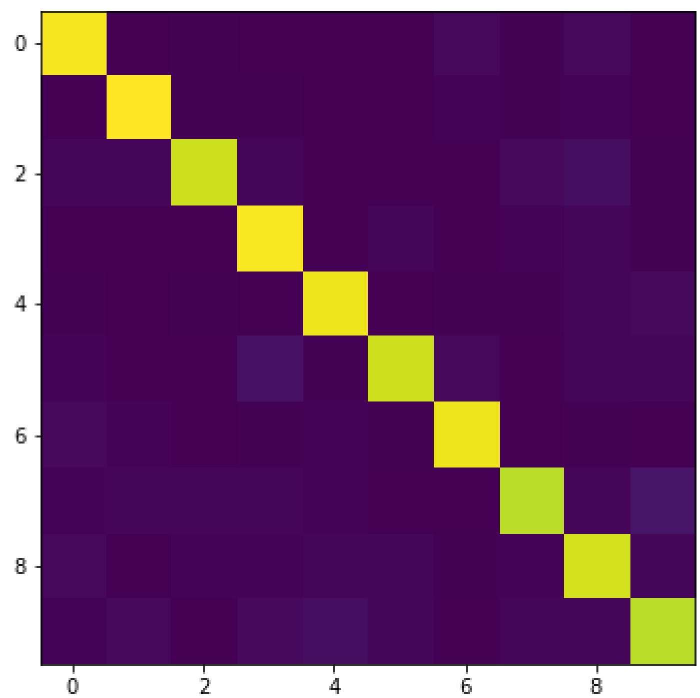
Here you will implement a CNN with the following architecture:

- n=5
- ReLU(Conv(kernel_size=4x4, stride=2, output_features=n))
- ReLU(Conv(kernel_size=4x4, stride=2, output_features=n*2))
- ReLU(Conv(kernel_size=4x4, stride=2, output_features=n*4))
- Linear(output_features=classes)

Display the confusion matrix and accuracy after training. You should get around ~ 98 % accuracy for 10 epochs and batch size 50.

In [37]:
```
def conv2d(x, W, stride=2):
    return tf.nn.conv2d(x, W, strides=[1, stride, stride, 1], padding='S
AME')

class CNNClassifer(TFClassifier):
    def __init__(self, classes=10, n=5):
        '''
        your code here
        '''
        self.sess = tf.Session()

        self.x = tf.placeholder(tf.float32, shape=[None,28,28,1]) # inpu
t batch of images
        self.y_ = tf.placeholder(tf.int64, shape=[None]) # input labels

        x_input = tf.reshape(self.x, [-1, 28, 28, 1])
        # model variables
        w_conv1 = weight_variable([4, 4, 1, n])
        b_conv1 = bias_variable([n])
        h1 = tf.nn.relu(conv2d(x_input, w_conv1) + b_conv1)

        w_conv2 = weight_variable([4, 4, n, n * 2])
        b_conv2 = bias_variable([n * 2])
        h2 = tf.nn.relu(conv2d(h1, w_conv2) + b_conv2)

        w_conv3 = weight_variable([4, 4, n * 2, n * 4])
        b_conv3 = bias_variable([n * 4])
        h3 = tf.nn.relu(conv2d(h2, w_conv3) + b_conv3)

        w_fc1 = weight_variable([4 * 4 * n * 4, 10])
        b_fc1 = bias_variable([10])
        h_c1 = tf.reshape(h3, [-1, 4 * 4 * n*4])

        self.y = tf.matmul(h_c1, w_fc1) + b_fc1


cnnClassifer = CNNClassifer()
cnnClassifer.train(trainData, trainLabels, epochs=10)
# display confusion matrix for your PCA KNN classifier with all the trai
ning examples
M = Confusion(testData, testLabels, cnnClassifer)
VisualizeConfusion(M)
```

```
Epoch:1 Accuracy: 92.470000
Epoch:2 Accuracy: 94.990000
Epoch:3 Accuracy: 96.430000
Epoch:4 Accuracy: 96.810000
Epoch:5 Accuracy: 97.440000
Epoch:6 Accuracy: 97.400000
Epoch:7 Accuracy: 97.630000
Epoch:8 Accuracy: 97.790000
Epoch:9 Accuracy: 97.940000
Epoch:10 Accuracy: 98.080000
```




```
[[0.95 0.   0.   0.   0.   0.   0.02 0.   0.02 0.  ]
 [0.   0.97 0.   0.   0.   0.   0.01 0.   0.01 0.  ]
 [0.02 0.01 0.89 0.01 0.   0.   0.   0.02 0.04 0.  ]
 [0.   0.   0.   0.96 0.   0.01 0.   0.01 0.01 0.  ]
 [0.   0.   0.   0.   0.94 0.   0.   0.   0.01 0.02]
 [0.01 0.   0.   0.05 0.   0.89 0.02 0.   0.01 0.01]
 [0.02 0.01 0.   0.   0.01 0.   0.94 0.   0.   0.  ]
 [0.01 0.01 0.02 0.02 0.01 0.   0.   0.87 0.01 0.06]
 [0.02 0.   0.01 0.01 0.01 0.01 0.   0.01 0.9  0.02]
 [0.01 0.02 0.   0.03 0.03 0.02 0.   0.02 0.01 0.86]]
```

- Note that the MLP/ConvNet approaches lead to an accuracy a little higher than the K-NN approach.
- In general, Neural net approaches lead to significant increase in accuracy, but in this case since the problem is not too hard, the increase in accuracy is not very high.
- However, this is still quite significant considering the fact that the ConvNets we've used are relatively simple while the accuracy achieved using K-NN is with a search over 60,000 training images for every test image.
- You can look at the performance of various machine learning methods on this problem at http://yann.lecun.com/exdb/mnist/ (http://yann.lecun.com/exdb/mnist/)
- You can learn more about neural nets/ tensorflow at https://www.tensorflow.org/tutorials/ (https://www.tensorflow.org/tutorials/)
- You can play with a demo of neural network created by Daniel Smilkov and Shan Carter at https://playground.tensorflow.org/ (https://playground.tensorflow.org/)