CPSC 425

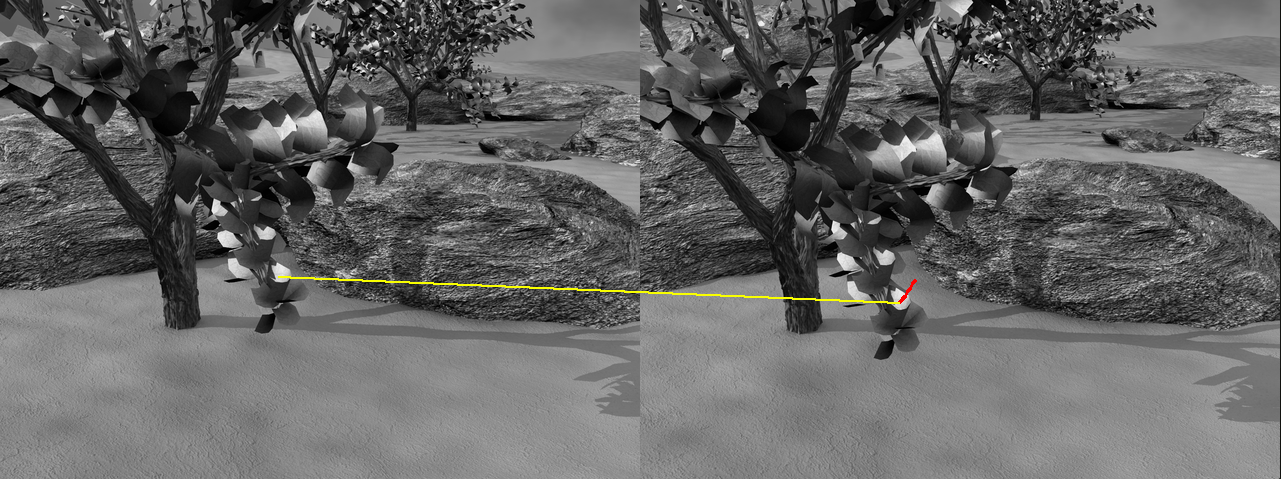
Assignment #6

Robert MacEachern

m9c7 – 42415091

**Question 5:**

I was tracking the point (278, 277) and I was able to track the point through each of the test frames when using a *window\_size=23* and a *sigma=2.25*. Below is the final result and the location of each tracked point:



tracked in frame 7 : ( 278 , 277 )

tracked in frame 8 : ( 275.38636926 , 280.895627459 )

tracked in frame 9 : ( 272.927413306 , 284.744041589 )

tracked in frame 10 : ( 270.465243956 , 288.630715491 )

tracked in frame 11 : ( 267.735982846 , 292.382763173 )

tracked in frame 12 : ( 265.048614234 , 296.130259667 )

tracked in frame 13 : ( 262.335993402 , 299.836526972 )

tracked in frame 14 : ( 259.214991968 , 303.213443582 )

**Question 6:**

One situation in which the Lucas-Kanade algorithm will fail is when we are trying to track a point that is close enough to the edge of the image that our window run off the edge. In this case, our partial derivative estimations would change drastically (or would be undefined) from one frame to the next as the number of ‘out of bounds’ pixels in the windows changes due to the motion in the scene. An example would be trying to track one of the leaves in the top left corner of our initial image, such as point (15, 15). My implementation actually only runs until the window reaches the edges and then crashes, and I don’t see how any Lucas-Kanade approach could handle this case.

Other situations in which I believe will give the Lucas-Kanade algorithm difficulties are: 1) the case of an image where the structure of the image is essentially random and 2) cases where the gradients are very small (ie. a mostly uniform surface). The reason these two cases are difficult for Lucas-Kanade is because of the algorithm’s dependence on the information in the intensity gradient to explain the intensity changes between frames. In both cases, the gradient simply does not give us enough information about the overall image to track the motion of a point properly.

from PIL import Image, ImageDraw

import matplotlib.pyplot as plt

import numpy as np

import math

from scipy import signal

import numpy.linalg as lin

# START OF FUNCTIONS CARRIED FORWARD FROM ASSIGNMENT 2

def boxfilter(n):

'''

Returns a box filter of size n by n (a numpy array)

Throws an error if n is even.

'''

# ensure that n is always odd

assert n % 2 == 1, "AssertionError: boxfilter dimension must be odd"

# create an empty n by n numpy array

filter = np.empty([n, n])

# fill each entry with 1/n^2

filter.fill(float(1) / (n \* n))

return filter

def gauss1d(sigma):

'''

Returns a 1D Gaussian filter with length ceil(sigma \* 6),

rounded up to the next odd integer.

Each value of the filter is computed with the Gaussian function, exp(- x^2 / (2\*sigma^2)),

and the values of the filter sum to 1.

'''

assert sigma > 0, 'gauss1d: sigma cannot be less than or equal to zero'

sigma = float(sigma)

# length should be 6 times sigma rounded up to the next odd integer

length = math.ceil(sigma \* 6)

# increment if length is even

if length % 2 == 0:

length = length + 1;

# construct initial 1D x values

maxx = int(length)/2

arange = np.arange(-maxx, maxx + 1)

# apply the formula to each value in our array

twoSigmaSqr = 2 \* sigma \* sigma

gaussFilter = np.exp(-arange \*\* 2 / twoSigmaSqr)

# normalize (ensure sum of matrix values is 1)

gaussFilter /= np.sum(gaussFilter)

return gaussFilter

def gauss2d(sigma):

'''

Returns a 2D Gaussian filter for a given sigma.

'''

a = gauss1d(sigma)[np.newaxis]

g = signal.convolve2d(a, a.T)

return g

# gauss = gauss1d(sigma)[np.newaxis]

# gaussTranspose = gauss1d(sigma)[np.newaxis].transpose()

# convolved = signal.convolve2d(gauss, gaussTranspose)

# return convolved

def gaussconvolve2d(image\_array, sigma):

'''

Applies 2D Gaussian convolution to the given image array.

'''

filtered\_array = signal.convolve2d(image\_array, gauss2d(sigma), 'same')

return filtered\_array

# END OF FUNCTIONS CARRIED FORWARD FROM ASSIGNMENT 2

# Define a function, boxconvolve2d, to convolve an image with a boxfilter of size n

# (used in Estimate\_Derivatives below).

def boxconvolve2d(image, n):

'''

Convolves an image with a boxfilter of size n

'''

filtered\_array = signal.convolve2d(image, boxfilter(n), 'same')

return filtered\_array

def Estimate\_Derivatives(im1, im2, sigma=1.5, n=3):

"""

Estimate spatial derivatives of im1 and temporal derivative from im1 to im2.

Smooth im1 with a 2D Gaussian of the given sigma. Use first central difference to

estimate derivatives.

Use point-wise difference between (boxfiltered) im2 and im1 to estimate temporal derivative

"""

# UNCOMMENT THE NEXT FOUR LINES WHEN YOU HAVE DEFINED THE FUNCTIONS ABOVE

im1\_smoothed = gaussconvolve2d(im1,sigma)

Ix, Iy = np.gradient(im1\_smoothed)

It = boxconvolve2d(im2, n) - boxconvolve2d(im1, n)

return Ix, Iy, It

def Optical\_Flow(im1, im2, x, y, window\_size, sigma=1.5, n=3):

'''

Calculates the optical flow between two images at a given point

using the Lucas-Kanade method.

im1 - the source image

im2 - the destination image

x,y - the location of the point to track in the source image

window\_size - the size of the window to track

sigma - the sigma value to use when estimating derivatives of source and destination images

n - the box filter size to use when estimating derivatives of source and destination images

'''

assert((window\_size % 2) == 1) , "Window size must be odd"

Ix, Iy, It = Estimate\_Derivatives(im1, im2, sigma, n)

half = np.floor(window\_size/2)

win\_Ix = Ix[y-half-1:y+half, x-half-1:x+half].T

win\_Iy = Iy[y-half-1:y+half, x-half-1:x+half].T

win\_It = -It[y-half-1:y+half, x-half-1:x+half].T

A = np.vstack((win\_Ix.flatten(), win\_Iy.flatten())).T

V = np.dot(np.linalg.pinv(A), win\_It.flatten())

return V[1], V[0]

def AppendImages(im1, im2):

"""Create a new image that appends two images side-by-side.

The arguments, im1 and im2, are PIL images of type RGB

"""

im1cols, im1rows = im1.size

im2cols, im2rows = im2.size

im3 = Image.new('RGB', (im1cols+im2cols, max(im1rows,im2rows)))

im3.paste(im1,(0,0))

im3.paste(im2,(im1cols,0))

return im3

def DisplayFlow(im1, im2, x, y, uarr, varr):

"""Display optical flow match on a new image with the two input frames placed side by side.

Arguments:

im1 1st image (in PIL 'RGB' format)

im2 2nd image (in PIL 'RGB' format)

x, y point coordinates in 1st image

u, v list of optical flow values to 2nd image

Displays and returns a newly created image (in PIL 'RGB' format)

"""

im3 = AppendImages(im1,im2)

offset = im1.size[0]

draw = ImageDraw.Draw(im3)

xinit = x+uarr[0]

yinit = y+varr[0]

for u,v,ind in zip(uarr[1:], varr[1:], range(1, len(uarr))):

draw.line((offset+xinit, yinit, offset+xinit+u, yinit+v),fill="red",width=2)

xinit += u

yinit += v

draw.line((x, y, offset+xinit, yinit), fill="yellow", width=2)

im3.show()

del draw

return im3

def HitContinue(Prompt='Hit any key to continue'):

#raw\_input(Prompt)

nothing = 1

##############################################################################

# Here's your assigned target point to track #

##############################################################################

# uncomment the next two lines if the leftmost digit of your student number is 4

x=278

y=277

##############################################################################

# Global "magic numbers" #

##############################################################################

# window size (for estimation of optical flow)

window\_size=23

# sigma of the 2D Gaussian (used in the estimation of Ix and Iy)

sigma=2.25

# size of the boxfilter (used in the estimation of It)

n = 3

##############################################################################

# basic testing (optical flow from frame 7 to 8 only) #

##############################################################################

# scale factor for display of optical flow (to make result more visible)

scale=10

PIL\_im1 = Image.open('frame07.png')

PIL\_im2 = Image.open('frame08.png')

im1 = np.asarray(PIL\_im1)

im2 = np.asarray(PIL\_im2)

dx, dy = Optical\_Flow(im1, im2, x, y, window\_size, sigma, n)

print 'Optical flow: [', dx, ',', dy, ']'

plt.imshow(im1, cmap='gray')

plt.hold(True)

plt.plot(x,y,'xr')

plt.plot(x+dx\*scale,y+dy\*scale, 'dy')

print 'Close figure window to continue...'

#plt.show()

uarr = [dx]

varr = [dy]

##############################################################################

# run the remainder of the image sequence #

##############################################################################

# UNCOMMENT THE CODE THAT FOLLOWS (ONCE BASIC TESTING IS COMPLETE/DEBUGGED)

print 'frame 7 to 8'

DisplayFlow(PIL\_im1, PIL\_im2, x, y, uarr, varr)

HitContinue()

print "tracked in frame", 7, ": (", x, ",", y, ")"

prev\_im = im2

xcurr = x+dx

ycurr = y+dy

offset = PIL\_im1.size[0]

print "tracked in frame", 8, ": (", xcurr, ",", ycurr, ")"

for i in range(8, 14):

im\_i = 'frame%0.2d.png'%(i+1)

#print 'frame', i, 'to', (i+1)

PIL\_im\_i = Image.open('%s'%im\_i)

numpy\_im\_i = np.asarray(PIL\_im\_i)

dx, dy = Optical\_Flow(prev\_im, numpy\_im\_i, xcurr, ycurr, window\_size, sigma, n)

xcurr += dx

ycurr += dy

print "tracked in frame", i+1, ": (", xcurr, ",", ycurr, ")"

prev\_im = numpy\_im\_i

uarr.append(dx)

varr.append(dy)

# redraw the (growing) figure

DisplayFlow(PIL\_im1, PIL\_im\_i, x, y, uarr, varr)

HitContinue()