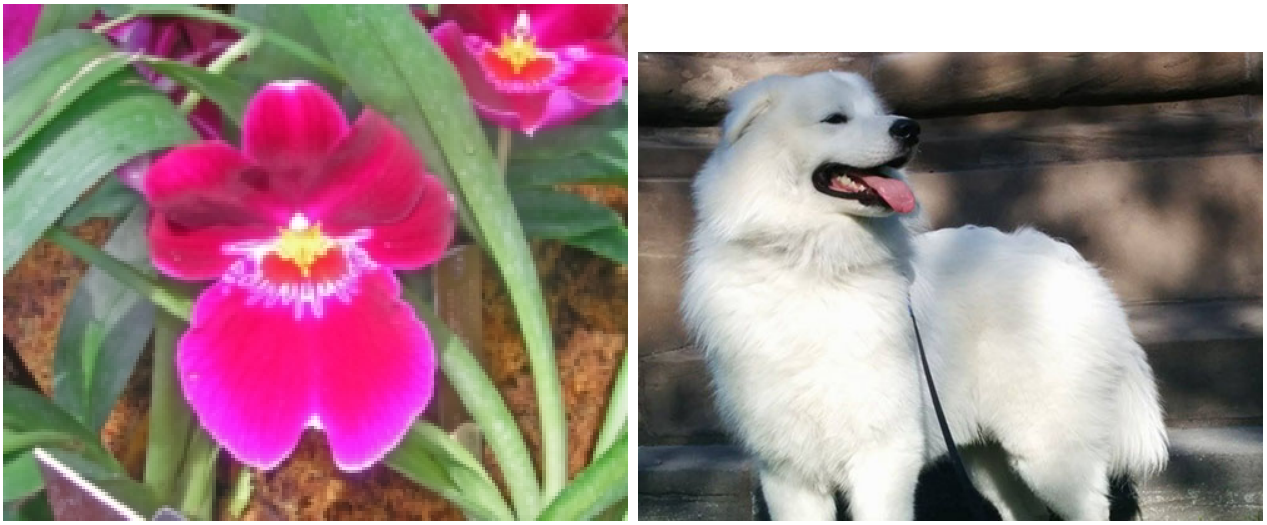
CSC320: Project 2

Painterly Renderings

Ramaneek Gill and Ryan D'Souza

February 22, 2015



(a) Default image provided for the assignment (b) My dog Blue

Figure 1: Raw Input Images

Part 1

Covering the Canvas: Blind Sampling
In part 1 I simply changed the for loop to a while loop that will loop until there is no more pixels left to paint:

```
while len(where(canvas < 0)[0]) != 0:
```

Part 2

Covering the Canvas: Systematic Sampling
In part 2 I added the following code to always randomly select an *unpainted* pixel:

```
#tuple of pixels not painted
empty_canvas_pixels = where(canvas < 0)
#choose a random non-painted pixel from the tuple
index = randint(0, len(empty_canvas_pixels[0]))
#get the position for the original canvas to paint on in array form
cntr = array([empty_canvas_pixels[1][index], empty_canvas_pixels[0][index]]) + 1
```

The output for part 1 and 2 is the same and is shown in Figure 2 below. Keep in mind with the code used in part two the images render much quicker, in the case of `"orchid.jpg"` the rendering required a couple hundred fewer strokes.

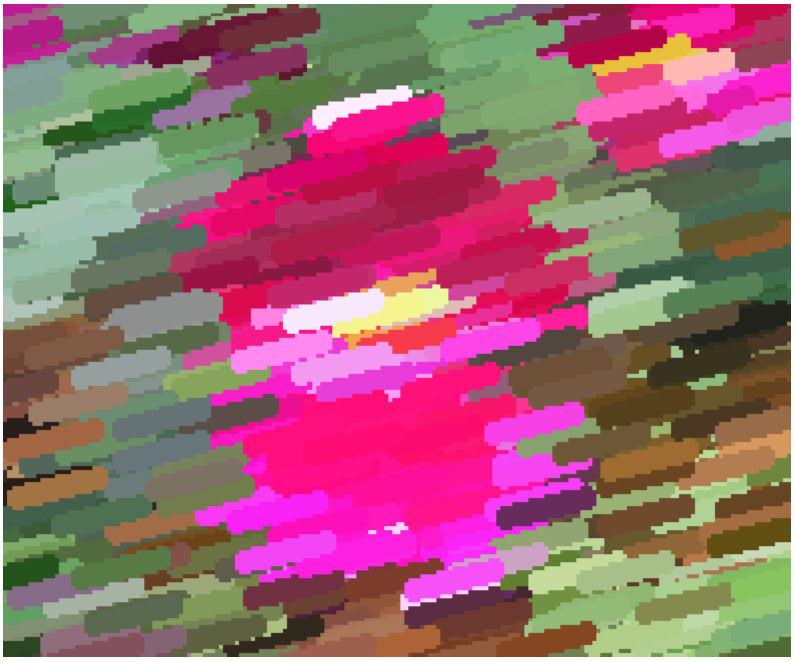


Figure 2: The output for part 1 and 2 is basically the same.
rad=3, halflen=10

Part 3

Computing Canny Edgels

In part 3 I ignored Litwinowicz's suggestion and decided to compute the canny edges of the red channel of the original image in order to better capture the petals of `orchid.jpg`. I set the Gaussian standard deviation to 2 and the high threshold to 20 and low to 7. For my other image I followed Litwinowicz's suggestion and used the same values for the canny edge detector as before. Figure 3 is the output image of the canny edge detection for both input images.

(a) The edges of orchid.jpg produced by `canny(imRGBmono, 2.0, 20, 7)`

(b) The edges of blue.jpg produced by `canny(imRGBmono, 2.0, 20, 7)`

Figure 3: Canny Edges of the Input Images

Part 4

Clipping Paint Strokes at Canny Edges

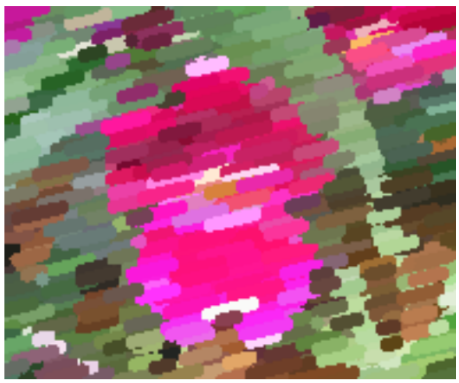
In part 4 the strokes were clipped if they hit an edge (any white pixel in Figure 2). This was done with the following code:

```
left = cntr - delta*i - 1
#while we're still less than or equal to halflen away from the center
#and the corresponding pixel is not an edel
while i <= length1: #and canny_im[left[1], left[0]] == 0:
    if canny_im[left[1], left[0]] != 0:
        #print "hit edge breaking"
        break
    canvas = paintStroke(canvas, x, y, cntr - delta * i, cntr, colour, rad)
    left = cntr - delta*i - 1
    if left[0] < 0 or left[1] < 0 or left[1] >= canny_im.shape[0] or left[0] >=
        canny_im.shape[1]: #then going out of bound
        break
    i += 1

i = 0
right = cntr + delta*i - 1
#now do it for the opposite direction
```

Part 4 continued on next page...

```
while i <= length2: #and canny_im[right[1], riimshowght[0]] == 0:
    if canny_im[right[1], right[0]] != 0:
        #print "hit edge breaking"
        break
    canvas = paintStroke(canvas, x, y, cntr, cntr + delta * i, colour, rad)
    right = cntr + delta*i -1
    if right[0] < 0 or right[1] < 0 or right[1] >= canny_im.shape[0] or right[0] >=
        canny_im.shape[1]: #then going out of bound
        break
    i += 1
```

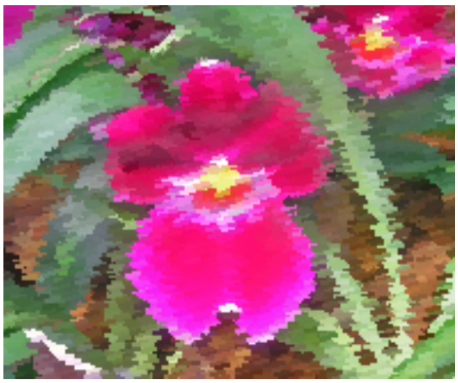
Refer to Figure 4 for the results with `rad=3` and `halflen=10`. Figure 5 is the output of the program with `rad=1` and `halflen=3` for `"orchid.jpg"` and `halflen=10` for "blue.jpg". This was done to show how a smaller halflen *and* rad value can highlight edges more clearly. Both these figures use the same edges as shown in Figure 3, refer to Part 3 on the details of how the images were produced.



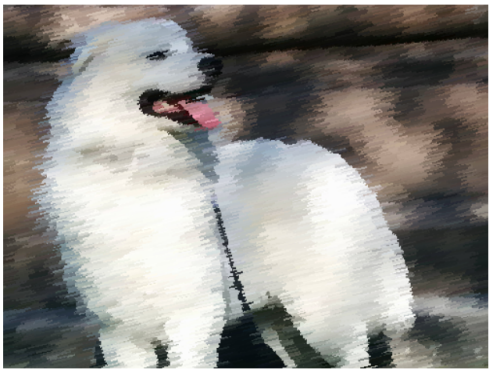
(a) The output of the program on the image "orchid.jpg" with clipping strokes at edges.
`rad=3, halflen=10`

(b) The output of the program on the image "blue.jpg" with clipping strokes at edges.
`rad=3, halflen=10`

Figure 4: Canny Edges of the Input Images



(a) The output of the program on the image "orchid.jpg" with clipping strokes at edges.
`rad=1, halflen=3`



(b) The output of the program on the image "blue.jpg" with clipping strokes at edges.
`rad=1, halflen=10`

Figure 5: Canny Edges of the Input Images

Part 5

Orienting the Paint Strokes

In part 5 I followed exactly what the handout described. The output images in Figure 6a and 6b were done with the Gaussian standard deviation set to 4 by `sigma=4`, the gradient threshold was set to 0.33 by `xx, yy = where(grad < 0.33)`. Angles below this threshold were all given a constant value determined by a random function. This threshold was determined by many tests and found to be an adequate threshold to show a ditributed theta array as shown in Figure 6b.

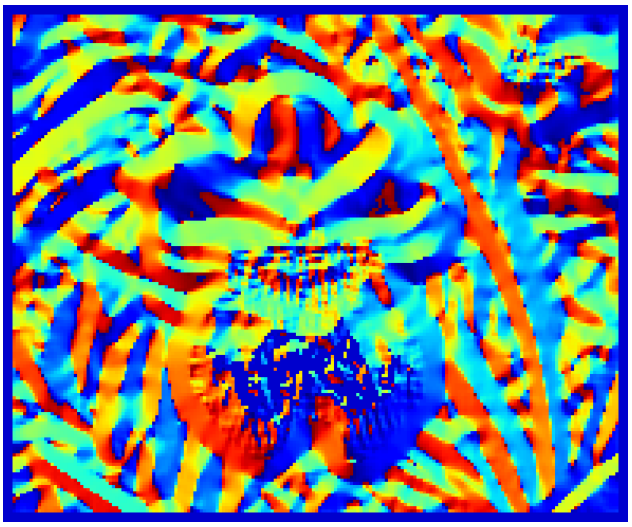
The main chunk of code used came from `canny.py`, here are the specific changes (note that `fy` multiplied by -1 in my code):

```
imin = imRGB_mono.copy() * 255.0
wsize = 5
sigma = 4

gausskernel = gaussFilter(sigma, window = wsize)
# fx is the filter for vertical gradient
# fy is the filter for horizontal gradient
# Please not the vertical direction is positive X
fx = createFilter([0, 1, 0,
               0, 0, 0,
               0, -1, 0])

fy = createFilter([ 0, 0, 0,
               1, 0, -1,
                0, 0, 0])

imout = conv(imin, gausskernel, 'valid')
# print "imout:", imout.shape
gradxx = conv(imout, fx, 'valid')
gradyy = conv(imout, fy, 'valid')

gradx = np.zeros(imRGB_mono.shape)
grady = np.zeros(imRGB_mono.shape)
padx = (imin.shape[0] - gradxx.shape[0]) / 2.0
pady = (imin.shape[1] - gradxx.shape[1]) / 2.0
gradx[padx:-padx, pady:-pady] = gradxx
grady[padx:-padx, pady:-pady] = gradyy

# Net gradient is the square root of sum of square of the horizontal
# and vertical gradients

grad = hypot(gradx, grady)
theta = arctan2(grady, gradx)
theta = 180 + (180 / pi) * theta
# Only significant magnitudes are considered. All others are removed
xx, yy = where(grad < 0.33)
theta[xx, yy] = math.degrees(2 * pi * np.random.rand(1,1)[0][0])
#grad[xx, yy] = 0 not needed

imshow(theta)
show()
#colorImSave("flipped_fy_part5_theta.png", theta)
```

Part 5 continued on next page...

```
normals = theta.copy() + 90 #add pi/2 to it for the normals
```

(a) The output of the program on the image "orchid.jpg" with clipping strokes at edges and stroke orientation. `rad=1, halflen=5`



(b) The theta array output of the program on the image "orchid.jpg". `sigma=4, threshold=0.33, default thetas = 2 * pi * np.random.rand(1,1)[0][0]`

Figure 6: Canny Edges of the Input Images

Part 6

Random Perturbations

In part 6 I perturbed each rgb channel's colour randomly from a range of $[-15/255, 15/255]$ seperately in order to allow for variance for black and white extreme values. I augmented this perturbation with a random intesntiy scaling from range $[-15\%, 15\%]$ for even more variance to allow for a more "hand painted" feel to the image. This value was clipped to the range [0,1] to allow the program to interpret these pixel intensities correctly. My last random perturbation was randomly orienting the paint strokes from a range of $[-15, 15]$ degrees to make the image look less "digitized", this gave the image an even better human look. Figure 7 shows these results. These images have the same values and design principles as set out in the previous parts of the report.

(a) The output of the program on the image "orchid.jpg" with clipping strokes at edges and stroke orientation and random perturbations.
`rad=1, halflen=2`

(b) The output of the program on the image "blue.jpg" with clipping strokes at edges and stroke orientation and random perturbations.
`rad=1, halflen=2`

Figure 7: Canny Edges of the Input Images

As a submission to the painterly rendering contest I modified the program even more to blur the rgb channels output canvas independantly to help dissolve the sharp edges of a brush stroke to make the entire image have a more natural feel because of the smoother flow/strokes.

I chose the picture "blue.jpg" carefully because of his all white coat I felt that rendering a painterly image of an object with a cery consistent colour would have been difficult to do because the only thing that stands out is his fur's texture. In order to do this well random perturbations were important to highlight the shadows and the variances in the dog's coat. Overall I am impressed with the result of my programs output on such a difficult input image.

Figure 8 and 9 is the result of running:

```
fi = zeros((sizeIm[0], sizeIm[1], 3))
#for dog image use sigma values 1, 1.2, 1.1 respectively for rgb channels
fi[:,:,0] = gaussian_filter(canvas[:,:,0], sigma = 0.4)
fi[:,:,1] = gaussian_filter(canvas[:,:,1], sigma = 0.6)
fi[:,:,2] = gaussian_filter(canvas[:,:,2], sigma = 0.5)
figure(2)
imshow(fi)
show()
```

Part 6 continued on next page...

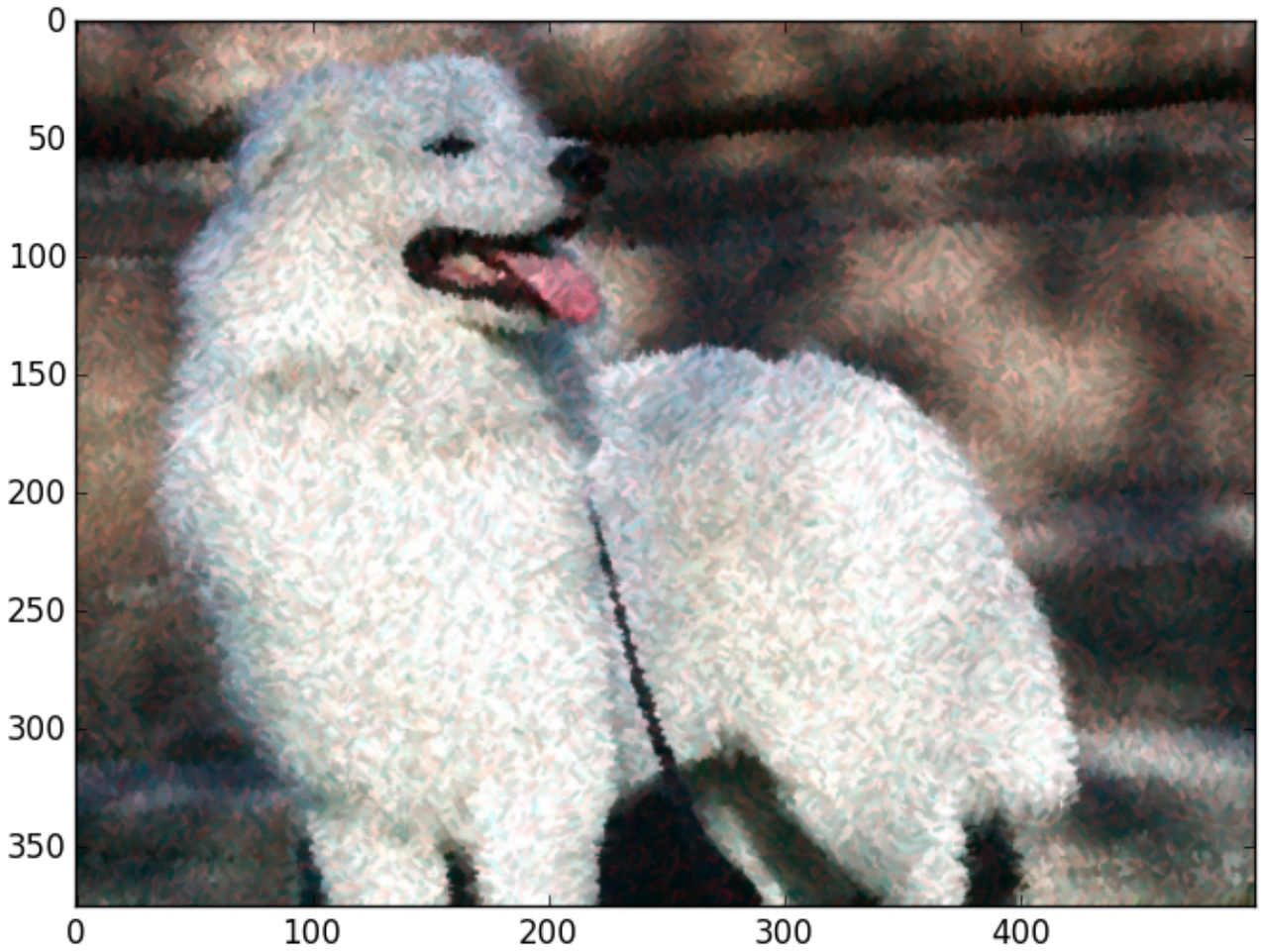


Figure 8: The final output for "blue.jpg"
rad=1, halflen=2

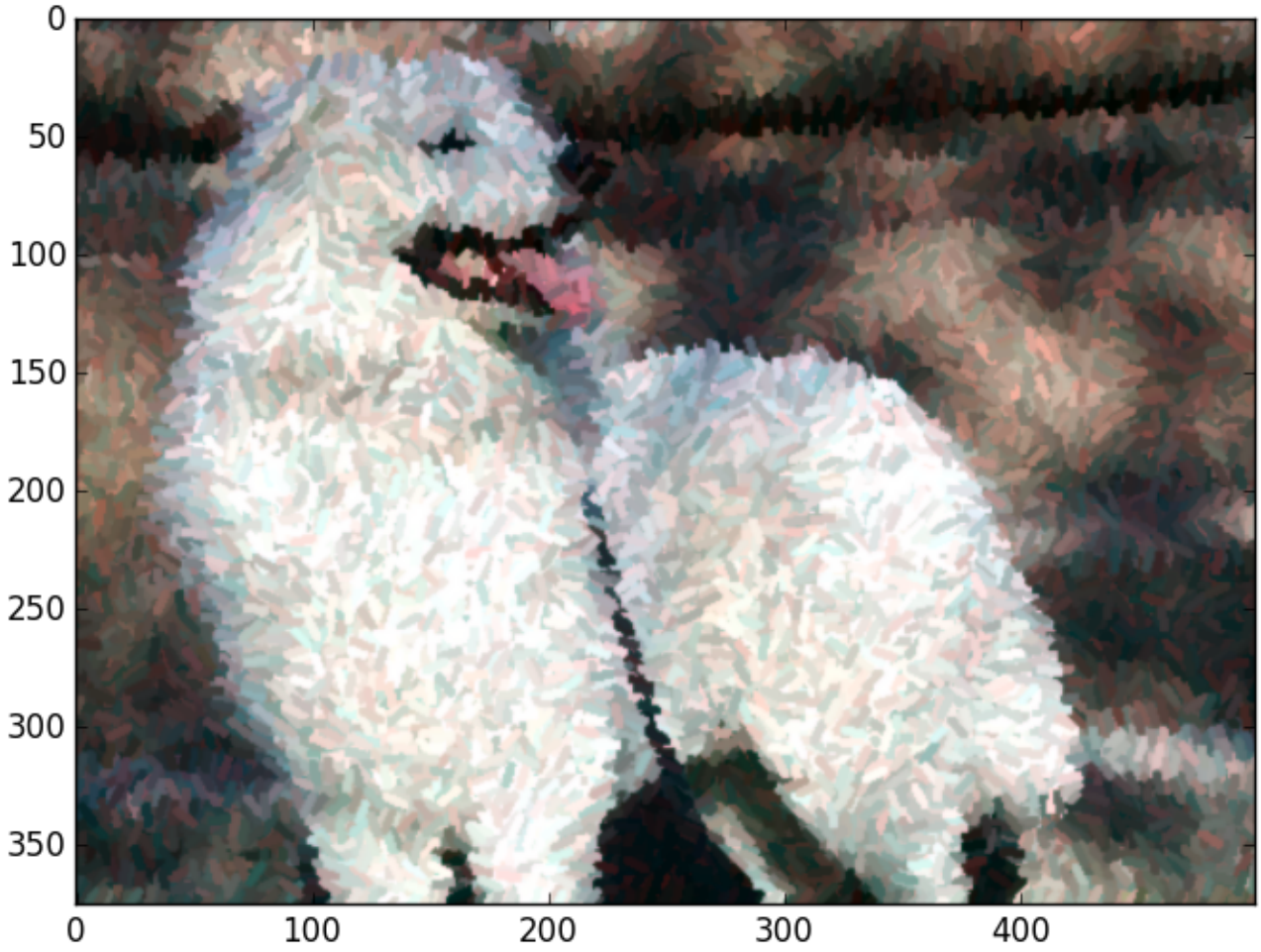


Figure 9: A more pastel like rendering of Figure 8 due to the increased rad and halflen value
rad=2, halflen=4

paintrend.py:

```
import os

os.chdir('C:/Users/Ramaneek/SkyDrive/Documents/University/Third Year/CSC320/project
    2/')

########################################################################
## Handout painting code.
########################################################################
from PIL import Image
from pylab import *
from canny import *
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook
import random
import time
import matplotlib.image as mpimg
import scipy as sci
import scipy.misc
from scipy.signal import convolve2d as conv


np.set_printoptions(threshold = np.nan)

def colorImSave(filename, array):
    imArray = sci.misc.imresize(array, 3., 'nearest')
    if (len(imArray.shape) == 2):
        sci.misc.imsave(filename, cm.jet(imArray))
    else:
        sci.misc.imsave(filename, imArray)

def markStroke(mrkd, p0, p1, rad, val):
    # Mark the pixels that will be painted by
    # a stroke from pixel p0 = (x0, y0) to pixel p1 = (x1, y1).
    # These pixels are set to val in the ny x nx double array mrkd.
    # The paintbrush is circular with radius rad>0

    sizeIm = mrkd.shape
    sizeIm = sizeIm[0:2];
    nx = sizeIm[1]
    ny = sizeIm[0]
    p0 = p0.flatten('F')
    p1 = p1.flatten('F')
    rad = max(rad,1)
    # Bounding box
    concat = np.vstack([p0,p1])
    bb0 = np.floor(np.amin(concat, axis=0))-rad
    bb1 = np.ceil(np.amax(concat, axis=0))+rad
    # Check for intersection of bounding box with image.
    intersect = 1
    if ((bb0[0] > nx) or (bb0[1] > ny) or (bb1[0] < 1) or (bb1[1] < 1)):
```

```
        intersect = 0
    if intersect:
        # Crop bounding box.
        bb0 = np.amax(np.vstack([np.array([bb0[0], 1]), np.array([bb0[1],1])]), axis=1)
        bb0 = np.amin(np.vstack([np.array([bb0[0], nx]), np.array([bb0[1],ny])]),
            axis=1)
        bb1 = np.amax(np.vstack([np.array([bb1[0], 1]), np.array([bb1[1],1])]), axis=1)
        bb1 = np.amin(np.vstack([np.array([bb1[0], nx]), np.array([bb1[1],ny])]),
            axis=1)
        # Compute distance d(j,i) to segment in bounding box
        tmp = bb1 - bb0 + 1
        szBB = [tmp[1], tmp[0]]
        q0 = p0 - bb0 + 1
        q1 = p1 - bb0 + 1
        t = q1 - q0
        nrmt = np.linalg.norm(t)
        [x,y] = np.meshgrid(np.array([i+1 for i in range(int(szBB[1]))]), np.array([i+1
            for i in range(int(szBB[0]))]))
        d = np.zeros(szBB)
        d.fill(float("inf"))

        if nrmt == 0:
            # Use distance to point q0
            d = np.sqrt( (x - q0[0])**2 +(y - q0[1])**2)
            idx = (d <= rad)
        else:
            # Use distance to segment q0, q1
            t = t/nrmt
            n = [t[1], -t[0]]
            tmp = t[0] * (x - q0[0]) + t[1] * (y - q0[1])
            idx = (tmp >= 0) & (tmp <= nrmt)
            if np.any(idx.flatten('F')):
                d[np.where(idx)] = abs(n[0] * (x[np.where(idx)] - q0[0]) + n[1] *
                    (y[np.where(idx)] - q0[1]))
            idx = (tmp < 0)
            if np.any(idx.flatten('F')):
                d[np.where(idx)] = np.sqrt( (x[np.where(idx)] - q0[0])**2
                    +(y[np.where(idx)] - q0[1])**2)
            idx = (tmp > nrmt)
            if np.any(idx.flatten('F')):
                d[np.where(idx)] = np.sqrt( (x[np.where(idx)] - q1[0])**2
                    +(y[np.where(idx)] - q1[1])**2)

            #Pixels within crop box to paint have distance <= rad
            idx = (d <= rad)
        #Mark the pixels
        if np.any(idx.flatten('F')):
            xy = (bb0[1]-1+y[np.where(idx)] + sizeIm[0] *
                (bb0[0]+x[np.where(idx)]-2)).astype(int)
            sz = mrkd.shape
            m = mrkd.flatten('F')
            m[xy-1] = val
            mrkd = m.reshape(mrkd.shape[0], mrkd.shape[1], order = 'F')
```

```
        '''
        row = 0
        col = 0
        for i in range(len(m)):
            col = i//sz[0]
            mrkd[row][col] = m[i]
            row += 1
            if row >= sz[0]:
                row = 0
        '''



    return mrkd

def paintStroke(canvas, x, y, p0, p1, colour, rad):
    # Paint a stroke from pixel p0 = (x0, y0) to pixel p1 = (x1, y1)
    # on the canvas (ny x nx x 3 double array).
    # The stroke has rgb values given by colour (a 3 x 1 vector, with
    # values in [0, 1]. The paintbrush is circular with radius rad>0
    sizeIm = canvas.shape
    sizeIm = sizeIm[0:2]
    idx = markStroke(np.zeros(sizeIm), p0, p1, rad, 1) > 0
    # Paint
    if np.any(idx.flatten('F')):
        canvas = np.reshape(canvas, (np.prod(sizeIm),3), "F")
        xy = y[idx] + sizeIm[0] * (x[idx]-1)
        canvas[xy-1,:] = np.tile(np.transpose(colour[:]), (len(xy), 1))
        canvas = np.reshape(canvas, sizeIm + (3,), "F")
    return canvas


if __name__ == "__main__":
    # Read image and convert it to double, and scale each R,G,B
    # channel to range [0,1].
    imRGB = array(Image.open('orchid.jpg'))
    imRGB = double(imRGB) / 255.0
    plt.clf()
    plt.axis('off')

    sizeIm = imRGB.shape
    sizeIm = sizeIm[0:2]
    # Set radius of paint brush and half length of drawn lines
    rad = 1
    halfLen = 10

    # Set up x, y coordinate images, and canvas.
    [x, y] = np.meshgrid(np.array([i+1 for i in range(int(sizeIm[1]))]), np.array([i+1
        for i in range(int(sizeIm[0]))]))
    canvas = np.zeros((sizeIm[0],sizeIm[1], 3))
    canvas.fill(-1) ## Initially mark the canvas with a value out of range.
    # Negative values will be used to denote pixels which are unpainted.
```

```
    # Random number seed
    np.random.seed(29645)

    # Orientation of paint brush strokes
    theta = 2 * pi * np.random.rand(1,1)[0][0]
    # Set vector from center to one end of the stroke.
    delta = np.array([cos(theta), sin(theta)])

    time.time()
    time.clock()
    k=0

##############################################################################
    gray()
    #imRGB_mono = np.zeros((sizeIm[0], sizeIm[1]))
    #imRGB_mono = imRGB[:,:,0] * 0.30 + imRGB[:,:,1] * 0.59 + imRGB[:,:,2] * 0.11
    #using canny edge detection on red filter
    imRGB_mono = np.zeros((sizeIm[0], sizeIm[1], 3))
    imRGB_mono = imRGB[:,:,0]

    #orchid
    high = 20; low = 7;

    #myimg
    #high = 15; low = 2;

    canny_im = np.zeros((sizeIm[0],sizeIm[1], 3))
    canny_im = canny(imRGB_mono, 2.0, high, low)

    imshow(canny_im)
    show()

### Part 5 code
    imin = imRGB_mono.copy() * 255.0
    wsize = 5
    sigma = 4

    gausskernel = gaussFilter(sigma, window = wsize)
    # fx is the filter for vertical gradient
    # fy is the filter for horizontal gradient
    # Please not the vertical direction is positive X
    fx = createFilter([0, 1, 0,
                   0, 0, 0,
                   0, -1, 0])

    fy = createFilter([ 0, 0, 0,
                   1, 0, -1,
                    0, 0, 0])

    imout = conv(imin, gausskernel, 'valid')
    # print "imout:", imout.shape
    gradxx = conv(imout, fx, 'valid')
```

```
    gradyy = conv(imout, fy, 'valid')

    gradx = np.zeros(imRGB_mono.shape)
    grady = np.zeros(imRGB_mono.shape)
    padx = (imin.shape[0] - gradxx.shape[0]) / 2.0
    pady = (imin.shape[1] - gradxx.shape[1]) / 2.0
    gradx[padx:-padx, pady:-pady] = gradxx
    grady[padx:-padx, pady:-pady] = gradyy

    # Net gradient is the square root of sum of square of the horizontal
    # and vertical gradients

    grad = hypot(gradx, grady)
    theta = arctan2(grady, gradx)
    theta = 180 + (180 / pi) * theta
    # Only significant magnitudes are considered. All others are removed
    xx, yy = where(grad < 0.33)
    theta[xx, yy] = math.degrees(2 * pi * np.random.rand(1,1)[0][0])
    #grad[xx, yy] = 0 not needed

    imshow(theta)
    show()
    #colorImSave("flipped_fy_part5_theta.png", theta)
    normals = theta.copy() + 90 #add pi/2 to it for the normals
    empty_canvas_pixels = where(canvas < 0)
################################################################################

    #run while there isn still a pixel left to paint
    while len(where(canvas < 0)[0]) != 0:
        #tuple of pixels not painted
        empty_canvas_pixels = where(canvas < 0)
        #choose a random non-painted pixel from the tuple
        index = randint(0, len(empty_canvas_pixels[0]))
        #get the position for the original canvas to paint on in array form
        cntr = array([empty_canvas_pixels[1][index], empty_canvas_pixels[0][index]]) + 1
        # Grab colour from image at center position of the stroke.
        colour = np.reshape(imRGB[cntr[1]-1, cntr[0]-1, :],(3,1))

        #preturb each r,g,b colour seperately
        colour[0] = colour[0] - randint(-15, 15)*1.0/255
        colour[0] = colour[0] - randint(-15, 15)*1.0/255
        colour[0] = colour[0] - randint(-15, 15)*1.0/255
        #scale colour from -15% to +15% and clamp to valid range [0,1]
        colour = colour * (randint(-15,15) + 100)/100
        colour = np.clip(colour, 0, 1)
        #preturb stroke orientation
        perturb_orientation = randint(-15, 15)
        #preturb stroke length, doing this for extra randomness for winning the bonus
            marks
        #pass

        # Add the stroke to the canvas
        nx, ny = (sizeIm[1], sizeIm[0])
```

```
    length1, length2 = (halfLen, halfLen)

    if canny_im[cntr[1]-1, cntr[0]-1] > 0:
        canvas = paintStroke(canvas, x, y, cntr, cntr, colour, rad)
    else:
        delta = np.array([
                    cos(math.radians(normals[cntr[1]-1][cntr[0]-1] +
                        perturb_orientation)),
                    sin(math.radians(normals[cntr[1]-1][cntr[0]-1] +
                        perturb_orientation))
                    ])

        i = 0
        left = cntr - delta*i - 1
        #while we're still less than or equal to halflen away from the center
        #and the corresponding pixel is not an edel
        while i <= length1: #and canny_im[left[1], left[0]] == 0:
            if canny_im[left[1], left[0]] != 0:
                #print "hit edge breaking"
                break
            canvas = paintStroke(canvas, x, y, cntr - delta * i, cntr, colour, rad)
            left = cntr - delta*i - 1
            if left[0] < 0 or left[1] < 0 or left[1] >= canny_im.shape[0] or left[0]
                >= canny_im.shape[1]: #then going out of bound
                break
            i += 1

        i = 0
        right = cntr + delta*i - 1
        #now do it for the opposite direction
        while i <= length2: #and canny_im[right[1], riimshowght[0]] == 0:
            if canny_im[right[1], right[0]] != 0:
                #print "hit edge breaking"
                break
            canvas = paintStroke(canvas, x, y, cntr, cntr + delta * i, colour, rad)
            right = cntr + delta*i -1
            if right[0] < 0 or right[1] < 0 or right[1] >= canny_im.shape[0] or
                right[0] >= canny_im.shape[1]: #then going out of bound
                break
            i += 1

        #canvas = paintStroke(canvas, x, y, cntr - delta * length2, cntr + delta *
            length1, colour, rad)
    #print imRGB[cntr[1]-1, cntr[0]-1, :], canvas[cntr[1]-1, cntr[0]-1, :]
    print 'stroke', k
    k += 1

print "done!"
time.time()

canvas[canvas < 0] = 0.0
plt.clf()
plt.axis('off')
```

```
    figure(1)
    plt.imshow(canvas)
    #show()

    ##FOR THE BONUS MARKS
    #going to blur each rgb channel a bit to mask some of the sharp edges
    #this makes the image look less digitized and more natural since the colors
    #blend in together
    fi = zeros((sizeIm[0], sizeIm[1], 3))
    #for dog image use sigma values 1, 1.2, 1.1 respectively for rgb channels
    fi[:,:,0] = gaussian_filter(canvas[:,:,0], sigma = 0.4)
    fi[:,:,1] = gaussian_filter(canvas[:,:,1], sigma = 0.6)
    fi[:,:,2] = gaussian_filter(canvas[:,:,2], sigma = 0.5)
    figure(2)
    imshow(fi)
    show()

    #plt.pause(3)
    #colorImSave('output.png', canvas)
```

‘