ACM/CS/EE 158, Spring 2017
Project Report
Inpainting from Global Image Statistics

Chloe (Ching-Yun) Hsu
Instructor – Venkat Chandrasekaran

May 30, 2017

1 Introduction

Inpainting is the task of restoring holes in images, based on background information. It is not a well-defined problem, but for simple images humans can naturally judge what looks "reasonable" according to surroundings of the hole. Inpainting is essential in image processing in a lot of use cases. For example, when removing objects from pictures, we need inpainting to fill in the holes in backgrounds. Red eye correction in photos is another example of inpainting.

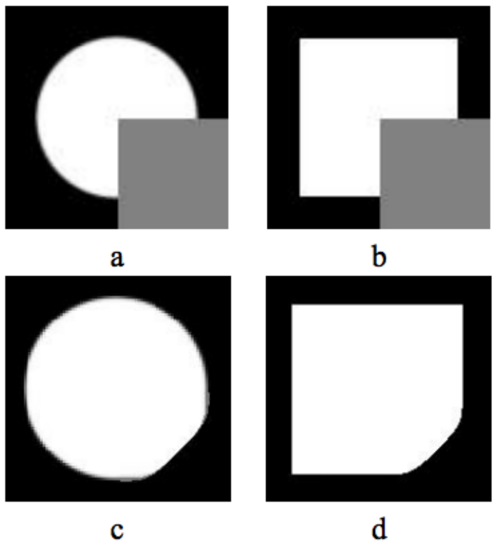
There are a lot of existing approaches to image inpainting. Pandya and Limbasya [1] surveyed some methods, including structural inpainting, texture inpainting, diffusion based inpainting, and exemplar-based inpainting.

My interest in image inpainting was initially motivated by patching missing regions of paintings and artworks. Compared to natural images, drawings and paintings contain a lot more geometric shapes and less noise, so the nature of the inpainting problem is quite different. For instance, I have not experimented, but I imaging diffusion-based inpainting would not work very well on drawings and paintings, since the shapes and lines would possibly get smeared by diffusion.

In this project, I implemented the statistical model by Levin, Zomet, and Weiss [2]. The reason for choosing this model is because the paper discusses by filling in holes in digitally generated images. A lot of other inpainting papers are mainly concerned with filling in holes in natural images, so they are not as suitable as Levin, Zomet, and Weiss's method.

Levin et al.'s inpainting approach is distinguished as global inpainting, rather than local inpainting. Most approaches before Levin, Zomet, and Weiss were local in the following sense: local inpainting completes the holes identically as along as the holes have identical boundary, even when the rest of the image is vastly different. Local inpainting approaches often require that the completed picture is "smooth" (e.g. low total variation or low curvature), and run an optimization for the best completion given the requirements.

In global inpainting, we hope to take global information into account when completing the image. Here's an example from Levin et al.'s paper [2] to illustrate why global inpainting provides valuable insights that are missing in local inpainting.



In this example, the holes in image (a) and (b) have identical boundaries. More specifically, the gradients and gray levels in the immediate boundary of the hole, are identical. Therefore, local inpainting methods would give the same hole fillings to (a) and (b), as shown in the completed images (c) and (d). Even though the filled holes appear to be smooth, any human observer can sense that there is something wrong with the pictures. Ideally, a global inpainting method should be able to tell apart a circle from a square, and fill in the two holes differently in (a) and (b).

This project adopts Levin et al.'s statistical global inpainting model: a probability distribution over images that is in the exponential family. Once I constructed the probability distribution according to Levin et al.'s model, I used simulated annealing to optimize the likelihood function.

2 Global Image Statistics

Since we would like the completed image to transition smoothly from the boundaries, it is a natural choice to consider gradients (of gray levels). The two image statistics in the model are 1) gradient magnitude, and 2) pairwise gradient angle.

Gradient magnitude tells us how sharply the gray scale colors transition. In digital images, gradient magnitude tends to be much smaller than natural images due to less noise.

Pairwise gradient angle is the angle between gradients at two neighboring pixels, which is motivated by the shape of the image. Consider a circle and a square. In a square, most of the gradient angle would be 0 since the edges are flat, and at the sharp corners the gradient angle would be close to 90 degrees. In contrast, a circle would have a non-zero gradient angle along the boundary (what the angle precisely is depends on the radius and curvature of the circle).

Here's a coarse plot of gradient magnitude and angle from my two examples.

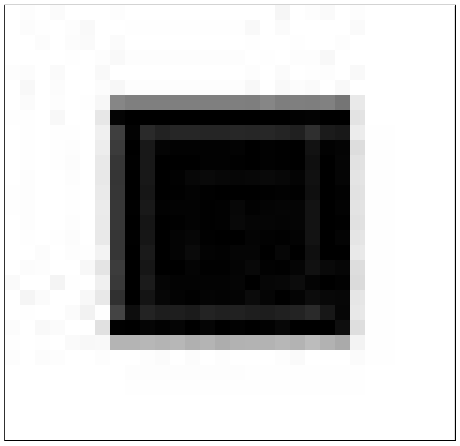


Figure 1: A 30x30 b/w square.

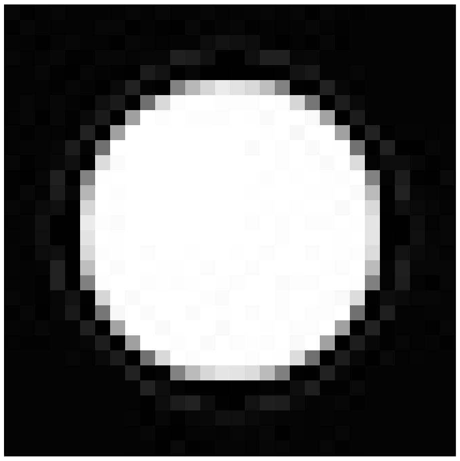


Figure 2: A 30x30 b/w circle.

The gradient magnitude histograms for square and circle:

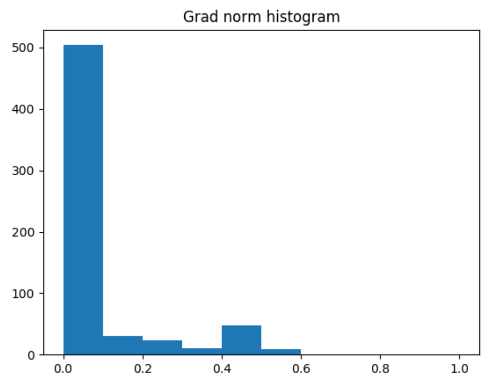


Figure 3: Gradient magnitude of square.

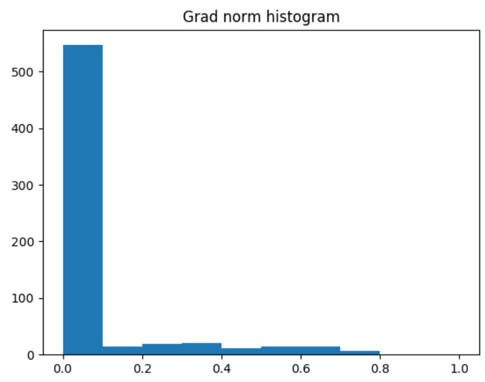


Figure 4: Gradient magnitude of circle.

The pairwise gradient angle histograms for square and circle:

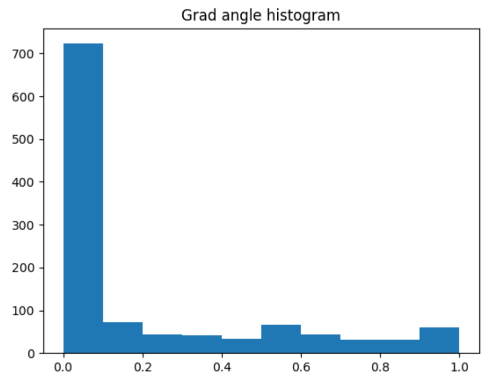


Figure 5: Gradient angle of square.

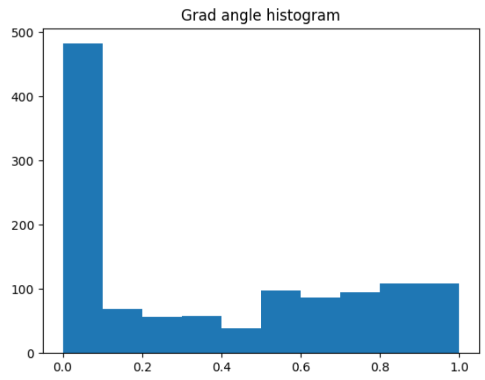


Figure 6: Gradient angle of circle.

More formally, gradient magnitude and pairwise gradient angle are defined as follows.

Let $I(x, y)$ be the gray level of the pixel located at position (x, y), then $g(x, y) = \nabla I(x, y)$ is the gradient and the square norm $|g|$ is the gradient magnitude. However, since this is working in a discrete setting, we need to specify how the gradient is discretized. I defined the discrete gradient as the average difference of two neighboring pixels

$$g(x, y) = (g_x, g_y) = \left(\frac{I(x+1, y) - I(x-1, y)}{2}, \frac{I(x, y+1) - I(x, y-1)}{2} \right)$$

On the edge of the image where there is only one side neighbor, then I just take that difference. And the feature is the square norm

$$F_1(x, y) = \|g(x, y)\|_2$$

The pairwise gradient angle is

$$F_2(x_1, y_1, x_2, y_2) = \cos^{-1}\left(\frac{g_1^T g_2}{|g_1||g_2|} \right),$$

where

$$g_1 = g(x_1, y_1), g_2 = g(x_2, y_2)$$

One question to address is how to define the pairwise gradient angle when $|g_1| = 0$ or $|g_2| = 0$. I took the lazy approach to define it as 0, but this is artificial and there is likely a more clever way to define the gradient angle at zero gradients.

The empirical probability distribution $\hat{P}$ of the two global image statistics are collected from the entire image. For pairwise gradient angle, the angles are collected from all pairs of neighbors. When producing the histogram, I found that it practically works well to bin the gradient magnitude and angle into intervals of 0.05.

3 Exponential Distribution

With an exponential distribution model on the above two features, the probability of an image is given by

$$P(I; \Phi_1, \Phi_2) \propto \exp\left[\sum_{(x,y)} \Phi_1(F_1(x, y)) + \sum_{(x_1,y_1)\sim(x_2,y_2)} \Phi_2(F_2(x_1, y_1, x_2, y_2)) \right]$$

To estimate Φ_1 and Φ_2 from the global image, the following estimation is adopted so that the predicted marginals of features match the empirical marginals in data.

$$\Phi_1(|g|) = \hat{P}(|g|)$$

$$\Phi_2(\theta) = \hat{P}(\theta)$$

Thus, our goal is to maximize the above expression of $P(I; \Phi_1, \Phi_2)$ in the hole.

4 Simulated Annealing

If we iterate over all possible hole configurations and brute force the maximization of $P(I; \Phi_1, \Phi_2)$, it would take too long since there are $|C|^N$ possible configurations where C is the candidate set of grayscale color levels and N is the size of the hole.

Instead of brute force, I used simulated annealing to approximate the best configuration to reduce computation time. The algorithm works as follows.

Define $V = \sum_{(x,y)} \Phi_1(F_1(x, y)) + \sum_{(x_1,y_1)\sim(x_2,y_2)} \Phi_2(F_2(x_1, y_1, x_2, y_2))$ to be the potential of a configuration. Let h be the function $h(a) = \min(a, 1)$.

Step 0: Initialize the hole with random grayscale colors sampled from a neighborhood.

Step n+1:

- Choose a pixel in the hole and change it to a new random grayscale color sampled from the neighborhood.

- Choose a uniform random number U from [0,1].

- Compute the new potential V'.

- Set the temperature to be $T = C/\log(n)$

- If $U < h(\exp[(V - V')/T])$, accept the new configuration and change that pixel. Otherwise, keep the old configuration.

In my implementation, I used an existing python simulated annealing package **simanneal**:

https://github.com/perrygeo/simanneal

5 Results

The results are from simulated annealing with initial temperature 10000K, and the candidate grayscale color levels are sampled from a neighborhood around the hole with 3 times the size of the hole.

In practice, I found it much better assign higher weights to Φ_1 on the boundary of the hole, so that the inpainted filling blends better with the surroundings. Otherwise, it's very easy to obtain an undesired filling of all white or all black pixels (since zero gradient has the highest probability among gradient magnitudes). In the following results, the pixels on the boundary are weighted by the size of the hole, and inner pixels are weighted by 1.

The algorithm works well to fill holes with 10-20 pixels, but simulated annealing very soon becomes too slow as the hole grows larger. For the 7x7 hole in Figure 17-18, simulated annealing takes $>$ 10 minutes to run, and the result is not very desirable.

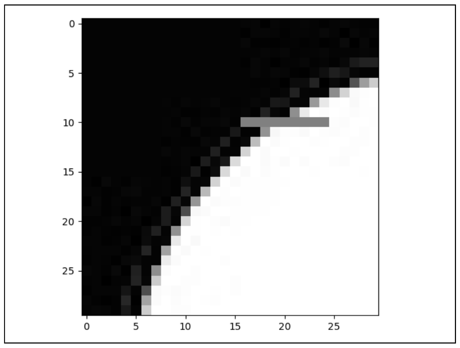


Figure 7: A 30x30 arc with 1x10 hole.

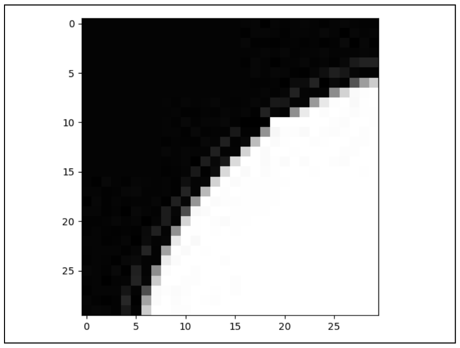


Figure 8: Filled hole after 10000 iterations.

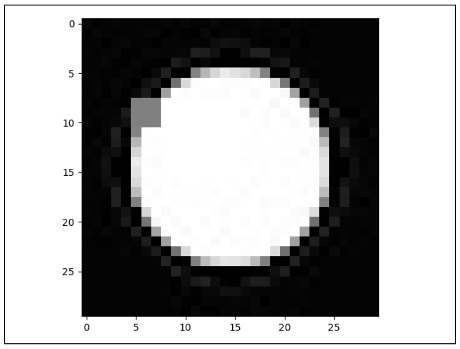


Figure 9: A 30x30 circle with 3x3 hole.

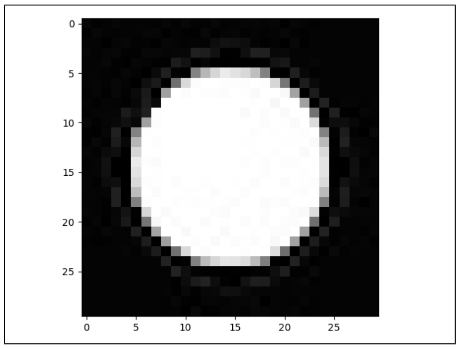


Figure 10: Filled hole after 50000 iterations.

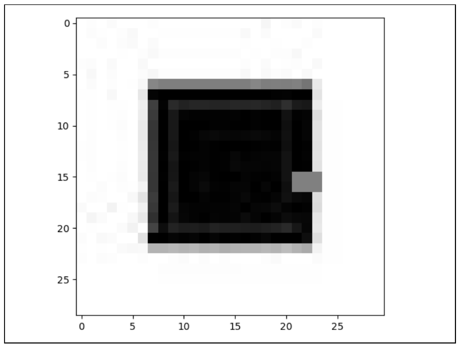


Figure 11: A 30x30 square with 3x3 hole.

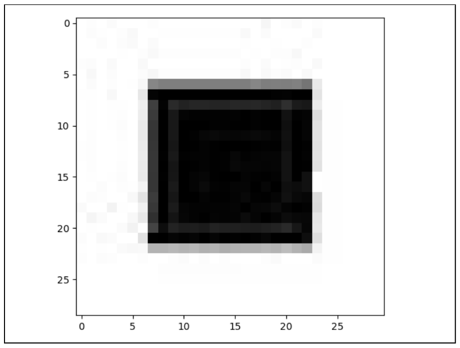


Figure 12: Filled hole after 50000 iterations.

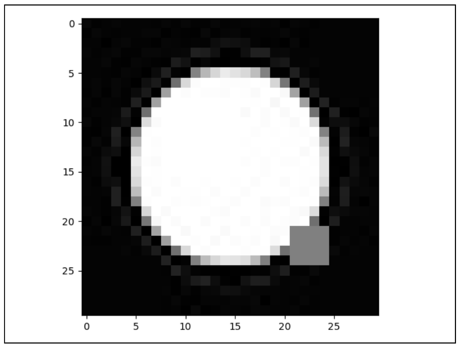


Figure 13: A 30x30 circle with 4x4 hole.

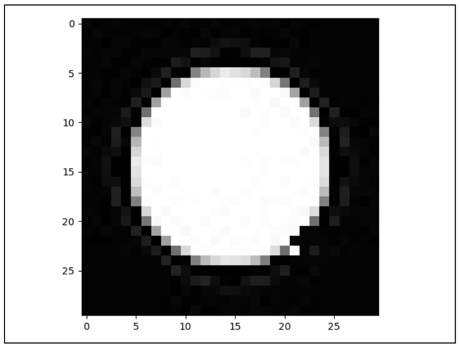


Figure 14: Filled hole after 50000 iterations.

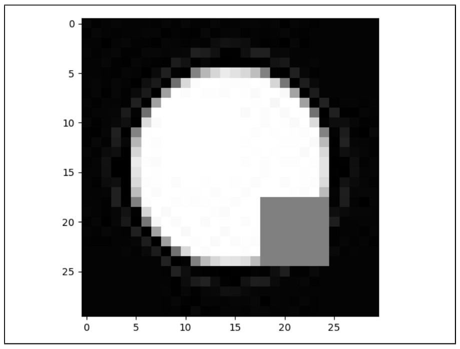


Figure 15: A 30x30 circle with 7x7 hole.

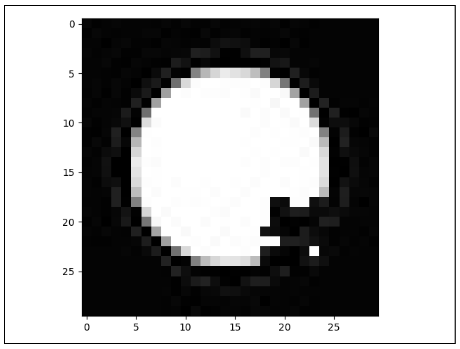


Figure 16: Filled hole after 100000 iterations.

6 Discussion

When the hole is small and we have enough time in simulated annealing to find a close-to-optimal configuration, the inpainting looks reasonable to human eyes. However, simulated annealing is not fast enough to handle more than 20 pixels.

One observation we can make is that this choice of the exponential model can be seen as a Markov Random Field on the gradient field. There is a well-known optimization algorithm called **Max-product Belief Propagation** [3]. On simply connected graphs, this algorithm can find the optimal configuration, and on general graphs this algorithm approximates the optimal configuration [3].

The next step of this project would be to implement max-product belief propagation and see if this inpainting model can handle larger holes.

It would also be interesting to experiment with other image statistics as features, such as log intensity, power spectrum, and wavelets [4].

References

[1] N. Pandya and B. Limbasiya, “A survey on image inpainting techniques,” *International Journal of Current Engineering and Technology*, vol. 3, no. 5, pp. 1828–1831, 2013.

[2] A. Levin, A. Zomet, and Y. Weiss, “Learning how to inpaint from global image statistics.,” in *ICCV*, vol. 1, pp. 305–312, 2003.

[3] Y. Weiss and W. T. Freeman, “On the optimality of solutions of the max-product belief-propagation algorithm in arbitrary graphs,” *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 736–744, 2001.

[4] T. Pouli, D. W. Cunningham, and E. Reinhard, “A survey of image statistics relevant to computer graphics,” in *Computer Graphics Forum*, vol. 30, pp. 1761–1788, Wiley Online Library, 2011.

Source Code

```
import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
import sys
import numpy as np
from skimage import data, io, filters
from math import sqrt, log, acos, pi, floor, exp
from simanneal import Annealer
from random import random, randint
from numpy.linalg import norm

NORM_SCALE = 10
ANGLE_SCALE = 10

def gradient_angle(g1, g2):
    tmp = np.dot(g1.transpose(), g2)
    if norm(g1) * norm(g2) > 0:
        tmp /= norm(g1) * norm(g2)
        # Sometimes the floating point numerical errors causes tmp to be
        # slightly larger than 1.0 or smaller than 1.0
        if tmp > 1.0:
            angle = 0
        elif tmp < -1.0:
            angle = 1.0
        else:
            # Normalize the angle to be between 0 and 1 for convenience
            angle = acos(tmp) / pi
    else:
        angle = 0
    return angle

class GradientInpainter(Annealer):

    """ Use simulated annealer to approximate optimal grayscale levels. """

    def move(self):
        x = randint(x1, x2-1)
        y = randint(y1, y2-1)
        image[x][y] = color[randint(0, len(color)-1)]
        self.state[0][x-x1][y-y1] = image[x][y]

    def energy(self):
        e = 0
        for i in range(x1-2, x2+2):
            for j in range(y1-2, y2+2):
                # Weight the boundary pixels higher so that the hole filling
                # blends well with surroundings
                if i == x1-1 or i == x2 or j == y1-1 or j == y2:
                    w = (y2-y1) * (x2-x1)
                else:
                    w = 1
                gx = (image[i+1][j] - image[i-1][j])/2
                gy = (image[i][j+1] - image[i][j-1])/2

                grad[i][j] = np.array([gx, gy])
                ll = hist_gradnorm[int(floor(norm(grad[i][j]) * NORM_SCALE))]
                e += log(ll) * w

        for i in range(x1-1, x2):
            for j in range(y1-1, y2):
                w = 1
                g1 = grad[i][j]
                if i+1 < x2:
                    g2 = grad[i+1][j]
                    angle = gradient_angle(g1, g2)
                    ll = hist_gradangle[int(floor(angle * ANGLE_SCALE))]
                    e += log(ll) * w
                if j+1 < y2:
                    g2 = grad[i][j+1]
                    angle = gradient_angle(g1, g2)
                    ll = hist_gradangle[int(floor(angle * ANGLE_SCALE))]
                    e += log(ll) * w

        return -e

def main():
    if len(sys.argv) != 8:
        print 'Please input filename, hole position, iterations, and Tmax as command
    line arguments.'
        print 'Example: python inpaint.py circle.png 10 10 20 20 100000 10000'
        exit()

    global image, x1, y1, x2, y2, grad, hist_gradnorm, hist_gradangle, color

    filename = str(sys.argv[1])
    image = io.imread(filename, as_grey=True)

    x1 = int(sys.argv[2])
    y1 = int(sys.argv[3])
    x2 = int(sys.argv[4])
    y2 = int(sys.argv[5])
    iterations = int(sys.argv[6])
    max_temp = int(sys.argv[7])
    (n1, n2) = image.shape

    for i in range(x1, x2):
        for j in range(y1, y2):
            image[i][j] = 0.5

    io.imshow(image)
    io.show()
    color = []
    grad = np.empty(image.shape, np.ndarray)
    gradnorm = []
    gradangle = []

    indices = np.ndindex(n1, n2)
    indices = filter(lambda (i,j): i not in range(x1, x2) and j not in range(y1,y2),
     indices)

    for i in range(x1 - (x2-x1), x2 + (x2-x1)):
        for j in range(y1 - (y2-y1), y2 + (y2-y1)):
            if (i,j) in indices:
                color.append(round(image[i][j],2))

    for (i,j) in indices:
        if i in [0, x2]:
            gx = image[i+1][j] - image[i][j]
        elif i in [x1-1, n1-1]:
            gx = image[i][j] - image[i-1][j]
        else:
            gx = (image[i+1][j] - image[i-1][j])/2

        if j in [0, y2]:
            gy = image[i][j+1] - image[i][j]
        elif j in [y1-1, n2-1]:
            gy = image[i][j] - image[i][j-1]
        else:
            gy = (image[i][j+1] - image[i][j-1])/2

        grad[i][j] = np.array([gx, gy])

        gradnorm.append(sqrt(gx**2 + gy**2))

    for (i,j) in indices:
        if (i+1,j) in indices:
            g1 = grad[i][j]
            g2 = grad[i+1][j]
            angle = gradient_angle(g1, g2)
            gradangle.append(angle)
        if (i,j+1) in indices:
            g1 = grad[i][j]
            g2 = grad[i][j+1]
            angle = gradient_angle(g1, g2)
            gradangle.append(angle)

    bins_gradnorm = np.multiply(np.arange(NORM_SCALE+1), 1.0/NORM_SCALE)
    (hist_gradnorm, bin_edges) = np.histogram(gradnorm, bins=bins_gradnorm, density=
    True)
    # print hist
    # print bin_edges

    # Modify the 0 entries to slightly positive so that we can take log
    for i in range(hist_gradnorm.size):
        hist_gradnorm[i] /= NORM_SCALE
        if hist_gradnorm[i] == 0:
            hist_gradnorm[i] = 1e-12
    hist_gradnorm = np.append(hist_gradnorm, hist_gradnorm[NORM_SCALE-1])
    bins_gradangle = np.multiply(np.arange(ANGLE_SCALE+1), 1.0/ANGLE_SCALE)
    (hist_gradangle, bin_edges) = np.histogram(gradnorm, bins=bins_gradangle,
    density=True)
    # print hist
    # print bin_edges

    # Modify the 0 entries to slightly positive so that we can take log
    for i in range(hist_gradangle.size):
        hist_gradangle[i] /= ANGLE_SCALE
        if hist_gradangle[i] == 0:
            hist_gradangle[i] = 1e-12
    hist_gradangle = np.append(hist_gradangle, hist_gradangle[ANGLE_SCALE-1])

    # plt.hist(gradnorm, bins=bins_gradnorm)
    # plt.title('Grad norm histogram')
    # plt.show()

    # plt.hist(gradangle, bins=bins_gradangle)
    # plt.title('Grad angle histogram')
    # plt.show()

    initial_state = np.empty((x2-x1, y2-y1), float)
    for i in range(x1, x2):
        for j in range(y1, y2):
            image[i][j] = color[randint(0,len(color)-1)]
            initial_state[i-x1][j-y1] = image[i][j]

    annealer = GradientInpainter([initial_state])
    annealer.steps = iterations
    annealer.Tmax = max_temp
    state, energy = annealer.anneal()
    print energy

    for i in range(x1, x2):
        for j in range(y1, y2):
            image[i][j] = state[0][i-x1][j-y1]

    io.imshow(state[0])
    io.show()

    io.imshow(image)
    io.show()

if __name__ == '__main__':
    main()
```

inpaint.py