CV1 : project #2 (total 10 points)
Restoration and Inpainting
Due October 27th, 2024, 11:59pm

1 Background

This project is based on Section 3.2.1, Section 3.2.2, and Section 3.2.3 of the textbook. You shall read the corresponding parts and understand the underlying logistics before writing your code.

1.1 Python Library

Please install the latest cv2, PIL, numpy, scipy, matplotlib, tqdm, torch (including torch-vision), and the cython (if you want) packages. You are also welcome to utilize any libraries of your choice, **but please report them in your report (for autograder)! Again, report any customized library in the report (do not go too crazy as this will add a significant burden to TAs).**

1.2 What to hand in?

Please submit both a formal report and the accompanying code. For the report, kindly provide a PDF version. You may opt to compose a new report or complete the designated sections within this document, as can be started by simply loading the tex file to Overleaf. Your score will be based on the quality of **your results**, **the analysis** (diagnostics of issues and comparisons) of these results in your report, and your **code implementation**.

Notice

1. There is no immediate feedback autograder to help with the debugging this time. The autograder will be run after the end of the homework!
2. Do not modify the function names in the given code, unless explicitly specified in this document.

1.3 Help

Make a diligent effort to independently address any encountered issues, and in cases where challenges exceed your capabilities, do not hesitate to seek assistance! Collaboration with your peers is permitted, but it is crucial that you refrain from directly examining or copying one another's code. Please be aware that you'll fail the course if our **code similarity checker**, which has found some prohibited behaviors for Project 1, detects these violations.

For details, please refer to https://yzhu.io/s/teaching/plagiarism/

2 Objective

This project serves as a preliminary exercise so that you can get familiar with Gibbs/MRF models, the fundamental principles of the Gibbs sampler, and the application of Partial Differential Equations (PDE) in the context of image restoration and inpainting. Three kinds of images are featured in the illustration presented below, all enclosed within the compressed file provided. The original image comprises distinct color bands, specifically Red, Green, and Blue. At the same time, the distorted counterpart is the result of superimposing a mask image onto the **Red band** of the original image. It is worth noting that two image sets are provided, one featuring a small font size, and the other, a big font size.

(a) Original Image (b) Mask Image (c) Distorted Image

In this experimental setting, pretend that you are provided solely with the distorted image denoted as **I** in subfigure (c) and the corresponding mask image **M** featured in subfigure (b). The primary objective of this experiment is to reconstruct the original image, represented as **O**, by filling in the masked pixels exclusively within the Red-band. It is essential to underscore that no restorative action is required for the Green and Blue bands. Given that information within the masked pixels has been irreversibly compromised, our task is to approximate the original image as **X**, which serves as an estimable substitute for **O**. To evaluate the efficacy of this restoration process, a per-pixel error assessment must be undertaken, explicitly concerning all the pixels concealed by the mask, comparing the reconstructed **X** with the ground truth image **O**.

As demonstrated below, to make the project more interesting, we offer three original images, and their corresponding distorted version. Note that all of the images are of the same size, and they are masked at the same place. **You should conduct experiments on all of them.**

(a) stone (b) sce (c) room

3 Method 1: Gibbs Sampler

3.1 Overview

Let Λ be the white pixels in the mask image **M** (distorted in I), and $\partial\Lambda$ the boundary condition (i.e., the undistorted pixels), which will stay unchanged. We compute

$$X_\Lambda | X_{\partial\Lambda} \sim p(X_\Lambda | X_{\partial\Lambda})$$

by sampling from a Gibbs or MRF model of the following form

$$p(X_\Lambda | X_{\partial\Lambda}) = \frac{1}{Z} \exp\{-\beta \sum_{(x,y)\in\Lambda} E(\nabla_x X(x,y)) + E(\nabla_y X(x,y))\},$$

where E() is a potential energy. We need to try two choices of functions:

- L_1 norm: $E(\nabla_x X(x,y)) = |\nabla_x X(x,y)|$
- L_2 norm: $E(\nabla_x X(x,y)) = (\nabla_x X(x,y))^2$.

From this Gibbs model, we can derive the local characteristics, given

$$X_s \sim p(X_s | X_{\partial s}), \forall s \in \Lambda.$$

By drawing from this 1D probability (using the inverse CDF method mentioned in Project 1), we can iteratively compute the values of the distorted pixels. Visiting all distorted pixels once is called 1-sweep (in whatever order, it does not matter). You need to experiment with an annealing scheme by slowly increasing the parameter β from 0.1 to 1 (or more).

3.2 Instructions

1. Please read through the main file and understand the whole logic.
2. Please implement the “conv()” function, which calculates the x direction and y direction gradients. Please be careful that the function’s return should hold the same shape as the input, and the boundary condition selected is **periodic boundary condition**.
3. Please implement the “gibbs_sampler()” function designed to execute Gibbs sampling for an individual pixel. (Hints: Consider optimizations to enhance computational efficiency by avoiding superfluous calculations.)
4. Implement the main function and tune the parameters for better effect.

3.3 Implementation

The implementation of this part of the project consists of three $TODO$ locations, corresponding to "conv()", "gibbs_sampler()" and the place of experimenting with the value of β. In the following few points, I will discuss the logic behind each impementation.

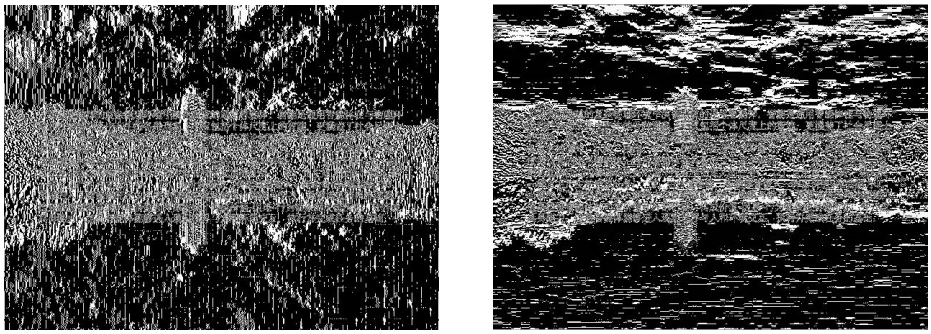
3.3.1 conv()

Below is the code used for calculation of the x direction and y direction gradients.

```
filtered_image = signal.convolve2d(image, filter, boundary='wrap')
```

Here, $scipy.signal.convolve2d()$ method is used to perform 2-d convolution onto our image. The condition on the boundary "wrap" accounts for the requirement of the convolution should respect the periodic (cyclic) boundary condition.

The code itself also contains if statement, but those are mainly just for the purpose of testing and debugging; by visualizing the x and y direction gradients, I was adjusting the implementation of this function as well as others. The resulting x direction and y direction gradients for the images are shown below:



(a) Horizontal direction gradient. (b) Vertical direction gradient.

Figure 3: Directional gradients of the image of the Beijing University lake.

As could be seen from the gradient images, both of the gradients are able to pick up on the text-disturbance on the image, but the edges of text are simultaneously accompanied with the edges of the image itself, and so the small-text edges are actually not that well-discernible to a naked eye.

3.3.2 gibbs_sampler()

This function is the heart of the whole algorithm. Before starting the implementation, I had to spend quite some time understanding the main flow of the code, the necessity of some calculations and the general logic of the sampling process. As such, the implementation of the "energy_list" variable calculation turned out to be the following:

```
# TODO: calculate the energy
# prep: calculate original values for grads around all
original pixels for easier calculation
nabla_x = conv(img, filter =
np.array([[-1,1]]).astype(np.float64)) # forward
difference for both
nabla_y = conv(img, filter = np.array([[-1],
[1]]).astype(np.float64))
for value in range(256): ## for any value of pixels
0~256
    # assume that the pixel being inspected becomes
    this intensity
    img[loc[0], loc[1]] = value
    # because the change of this pixel only affects
    the value of neighbouring 4 pixels, we need to
    only
    # calculate for them. the rest of the pixels can
    be represented by energy - difference between the
    altered
    # value for pixels then and now
    # 1. calculate the changed values for grads around
```

```
this pixel; x: left and right pixels, y: top and
bottom pixels
# since we're calculating the grads with forward
diff -> loc pos pixel change only influences the
prev pixel and itself
left = (loc[0] - 1, loc[1]) if loc[0] - 1 > 0
else (loc[0] + img_width, loc[1])
right = (loc[0] + 1, loc[1]) if loc[0] + 1 <
img_width
else (loc[0] - img_width, loc[1])
top = (loc[0], loc[1] - 1) if loc[1] - 1 > 0
else (loc[0], loc[1] + img_height)
bottom = (loc[0], loc[1] + 1) if loc[1] + 1 <
img_height
else (loc[0], loc[1] - img_height)
new_left = value - img[left[0], left[1]]
new_loc_x = img[right[0], right[1]] - value
new_bottom = value - img[bottom[0], bottom[1]]
new_loc_y = img[bottom[0], bottom[1]] - value
# 2. substitute the original values for energy at
this locations and add the new ones
to update energy aspect
energy += np.abs(-nabla_x[left[0], left[1]] -
nabla_x[loc[0], loc[1]] - nabla_y[bottom[0],
bottom[1]] - nabla_y[loc[0], loc[1]] +
            (new_left + new_loc_x + new_bottom +
            new_loc_y))
energy_list[value] = energy
```

The code and the corresponding comments already showcase a lot of details about the implementation. Firstly, a variable of energy – the total value over all pixels for the disturbed image, calculated ahead of time, is given to us in this function.

At the same time, being provided that the calculation requires some sort of optimization (indeed, calculating the gradients for each value of the energy list would be too taxing for the process), this energy definitely needs to be used. And so, considering that upon the try-out of different values for the pixel at the given location, the only pixels but the one at location whose gradient values are also affected are the left and bottom ones.

The reason for the right and top ones not being affected is the choice of propagation for the gradient calculation: in the original code the filters used both do forward propagation, which means the gradient calculation result at the current pixel is only influenced by itself and the next pixel; if we choose to backward propagate, conversely right and top pixels relative to the target one will get influenced.

With this, to get the new energy of this image with the location pixel swapped out for our try-values, we just subtract the old values and add the new ones. Like this, we finish getting the energy corresponding to this specific value at a pixel after the change of the pixel value, and put it into the array.

3.3.3 Inverse CDF and Sampling

As our energy list is computed for each value, or more like for all possible values of the distribution, then the inverse CDF function can be simply calculated as a cumulative sum. A random number within bounds (0, 1) is then called, and a pixel that best matches the distribution is chosen to substitute the disturbed one. The code implementation looks like this:

```
# TODO
```

```
# we sample from the conditional probability the same
way we did from 1/p^3 in the last paper
# so before the sampling, we just need to have the
distro for all possible pixel values for this location
(energy list) to sample from it
cdf = np.cumsum(probs)
random_value = random.random()
new_pixel_value = np.searchsorted(cdf,random_value)
img[loc[0], loc[1]] = new_pixel_value
```

3.4 Results and Analysis

The implementation of this method was, frankly, quite taxing both in the process and in the computation part. For speed up, I used cython as recommended – but even that ended up giving me on average 5 minutes per sweep.

On top of that, the results were also confusing at first. When i first finished the implementation – having completed the analysis on the gradients and more other checks, the convergence rate, or the rate of disappearance of the red text was almost nonexistent. Indeed, there was an issue with part of code that was not making an update to the distort correctly after the first batch. Here is an example, where sweep 1 and 35 contain more or less percentage of red text, but the positioning is slightly different at times:

(a) Sweep 1. (b) Sweep 35.

Figure 4: Comparison of sweep 1 and 35 with the incomplete implementation.

Unfortunately, I did not manage to figure out where the issue of the error is. It seems like the function either falsely updates one of it's values or performs the calculation faulty in some way. I hope that the chain of thought of this implementation explained above was enough to showcase the implementation.

(a) Sweep 1. (b) Sweep 35.

Figure 5: Restored image sequence by sweep, total of # sweeps.

4 Method 2: PDE

4.1 Overview

For L_2-norm energy functions, you can minimize the energy by the heat-diffusion equation.

4.2 Instructions

1. Implement the “pde()” function, which performs pde update for a specific pixel.
2. Implement the main function and tune the parameters for better effect.
3. Please run more sweeps to see the marvelous effect of this method.

4.3 Implementation

The implementation of this part follows the concepts introduced in chapter 3.2.3 on Textures. Here, we also have three $TODO$ locations to fill in, corresponding to the function call for calculating the PDE for one single pixel location, "pde()" itself and the update of β.

4.3.1 Function call

Similarly as in the first method, we also need to only address the pixels in the red band that have the pixel value of 255, pointing to the red text on the image in the mask. We fill in that code and call the "pde()" function correspondingly.

```
if mask[i,j,2] == 255:
distort[:, :, 2] = pde(distort[:, :, 2], [i, j], beta, f)
```

4.3.2 pde()

The logic behind the implementation of this is the following: having that the accumulation of the energy in the image pixels equals to the Laplacian over that image by the heat-diffusion equation, then we need to estimate the location pixel with the Laplacian equation in order to get the needed direction of change.

After that, we use the value of β again to decide how much the update is going to affect the pixel, perform the update and clip the negative values to area of 0 255. The image returned is the reconstructed result, and by repeating this process over multiple sweeps, we get the fully reconstructed image.

```
original_pixel = img[loc[0], loc[1]]
# TODO
# do pde only for this specific pixel, the rest of the pixels
dont really change
laplacian_value = (
    img[loc[0]+1, loc[1]] + img[loc[0]-1, loc[1]] +
    img[loc[0], loc[1]+1] + img[loc[0], loc[1]-1] -
    4 * img[loc[0], loc[1]]
)
f.write(f'laplacian value is: {laplacian_value}\n')
img[loc[0], loc[1]] += beta*laplacian_value
f.write(f'previous value at loc: {original_pixel}, new estimated
    value is: {img[loc[0], loc[1]]}\n')
img[loc[0], loc[1]] = np.clip(img[loc[0], loc[1]], 0, 255)
```

4.3.3 Results and Analysis

Similar to the first method, here we also need to try different values of β before finding the correct one. But unlike the first method, this one is actually a lot more sensitive to smaller changes. As such, the initial value of 1 with an upgrade of 0.1 does not get rid of the letters, just changing their colour to the green specter instead; as such, I decided to make the value smaller.

The corresponding experiments and their results will be shown below; through them, I got to the more stable value of convergence over 100 sweeps: $beta = 0.001$ with an upgrade of 0.0065 each sweep. This value makes sure the image converges and fully gets rid of the visible text, while also not going too far that the result becomes messy.

Now, the pixel error calculated with MSE method over the sweeps has the following graph:

The result of the graph is as expected: by the final sweeps, the MSE goes closer and closer to 0, and the images merge into the same one.

5 Speed Up (Optional)

This is not required in this project, and there is no extra bonus for implementing it. If you have time, you can have a try.

1. If your implementation is done by using Torch, it'll be much faster to change to numpy.
2. Use the **cython** library! Please first read the basic tutorial, and understand some basic logistics of **cython**. Please be cautious that the **"pyximport" method** is the **preferred approach** as opposed to the **"setup.py" method**. The latter may generate different files on different operating systems, and it's important to be aware that the autograder utilizes the Linux operating system. Now, you are almost ready to speed up your Gibbs sampling, please read the working with numpy section of the document. You have done a great job, and please speed up your Gibbs sampling process with Cython!

6 Hand-in

1. Plot the per pixel error $\frac{1}{|\Lambda|}\sum(X(x,y)-O(x,y))^2$ over the number of sweeps t in for both methods.
2. Show the restored image sequence corresponding to the number of sweeps.
3. **(Bonus, Optional)** Where are the original images from? Try to find them in PKU, take nice photos of them at a similar angle, report their names, and submit your photos.

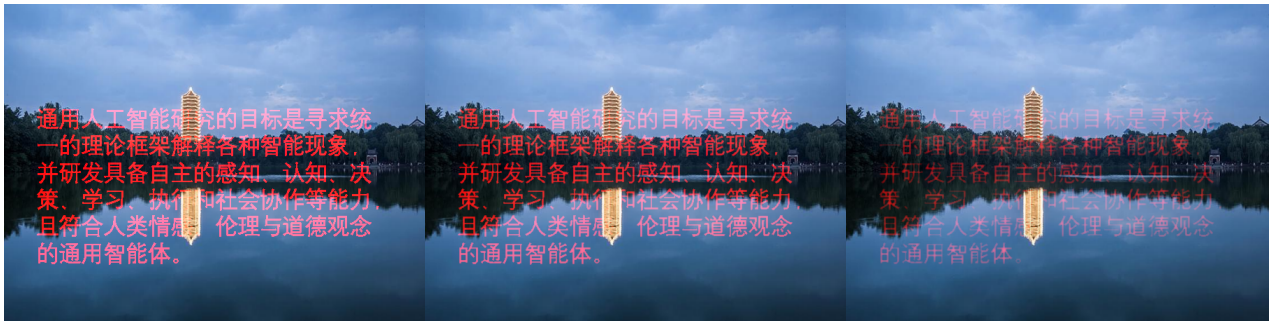
(a) Sweep 10. (b) Sweep 20. (c) Sweep 30.

(d) Sweep 40. (e) Sweep 50. (f) Sweep 60.

(g) Sweep 70. (h) Sweep 80. (i) Sweep 90.

(j) Sweep 90.

Figure 6: Over sweep 100 for $beta = 0.01$, update 0.01.



(a) Sweep 10. (b) Sweep 20. (c) Sweep 30.

(d) Sweep 40. (e) Sweep 50. (f) Sweep 60.

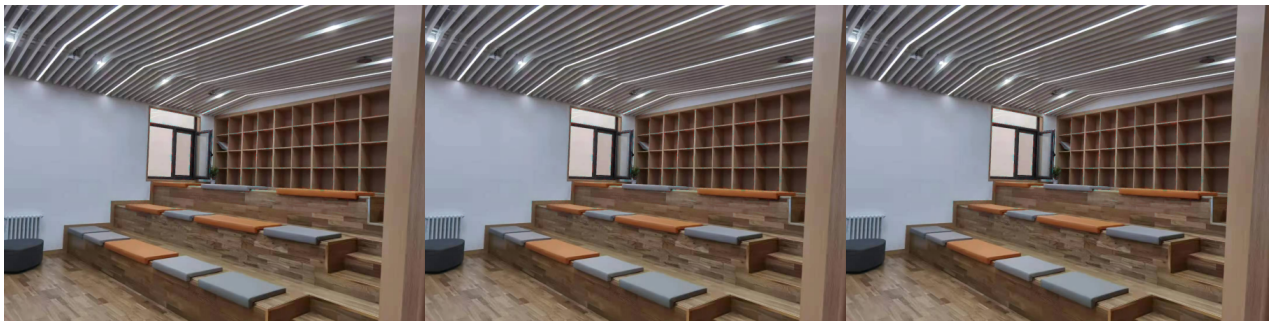
(g) Sweep 70. (h) Sweep 80. (i) Sweep 90.

(j) Sweep 90.

Figure 7: Over sweep 100 for optimal $beta = 0.001$, update 0.0065.

(a) Sweep 10. (b) Sweep 20. (c) Sweep 30.

(d) Sweep 40. (e) Sweep 50. (f) Sweep 60.



(g) Sweep 70. (h) Sweep 80. (i) Sweep 90.

(j) Sweep 90.

Figure 8: Over sweep 100 for optimal $beta = 0.001$, update 0.0065. Room with small letters. For this case, could do even more sweeps for better result.

(a) Sweep 10. (b) Sweep 20. (c) Sweep 30.

(d) Sweep 40. (e) Sweep 50. (f) Sweep 60.

(g) Sweep 70. (h) Sweep 80. (i) Sweep 90.

(j) Sweep 90.

Figure 9: Over sweep 100 for optimal $beta = 0.001$, update 0.0065. Stone with small letters. For this case, could do even more sweeps for better result.

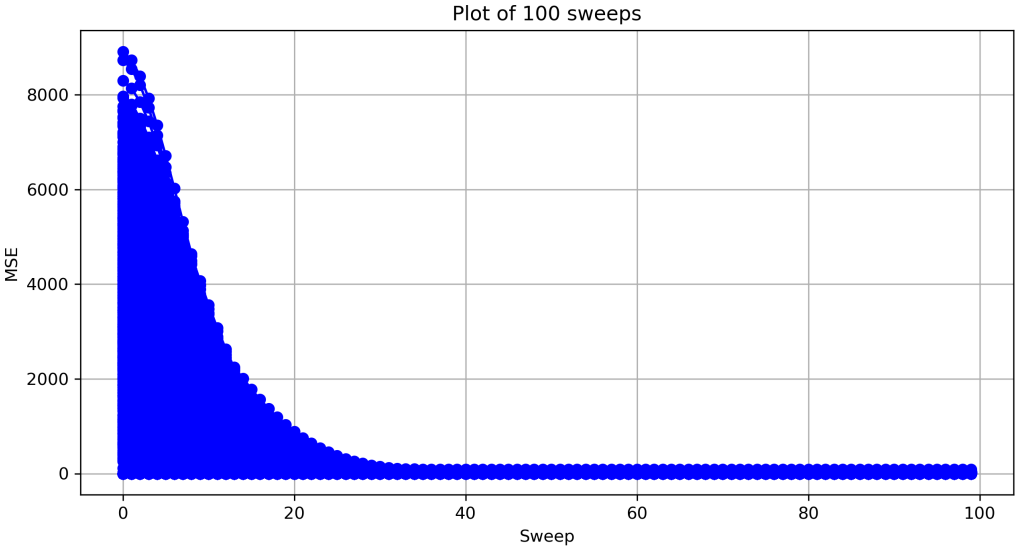


Figure 10: Plot of the Method 2 PDE pixel error over sweeps.