Georgia Tech

Computational Photography

Final Project

Rhythm Muhtadyuzzaman Syed

CS6475 - Fall 2019

Inference Based Blur Removal from Images

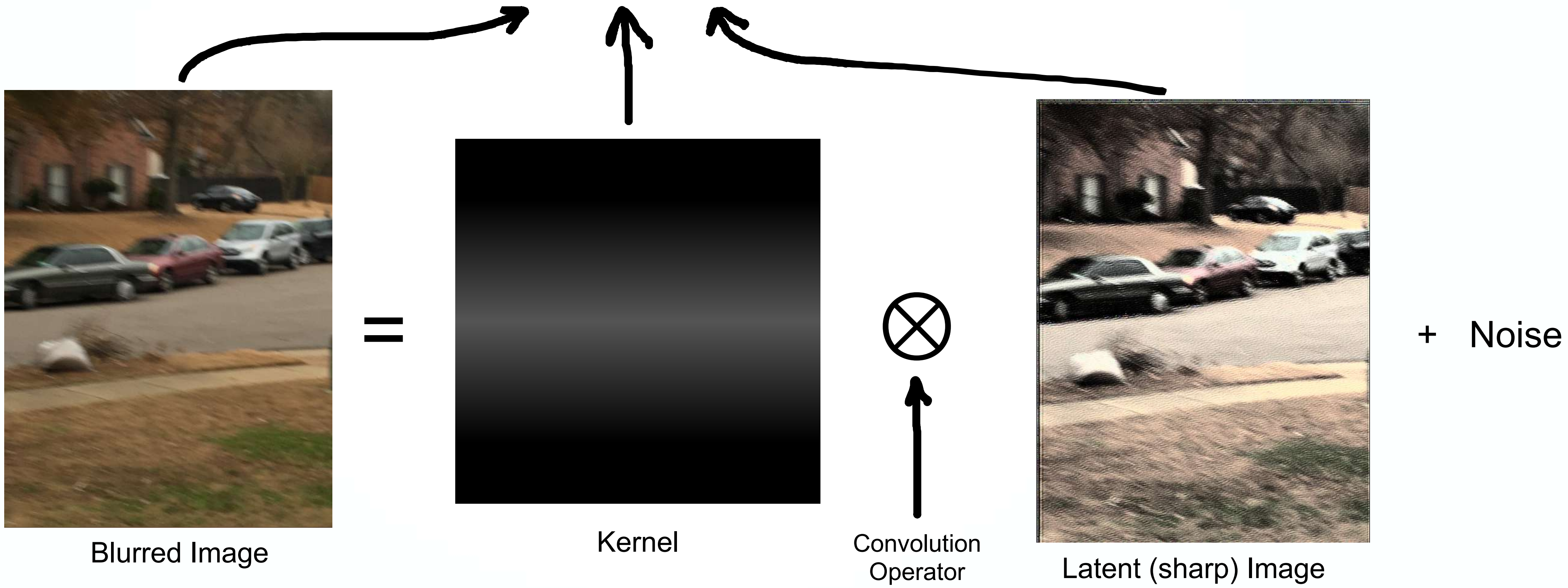
In today's age, photographs are often taken in nonideal conditions, whether due to unsatisfactory features in the scene or simply due to the cameraman's inability to hold the camera still when taking the photograph. For the latter case, blurring and camera shake are prevalent for any photography enthusiast. This project aims to tackle this issue by introducing and implementing a pipeline that removes blur and camera shake from everyday photographs.

The Goal of Your Project

Original Project Scope:

The scope of this implementation is to introduce an image pipeline that eliminates motion blur and camera shake from an image without knowing the deconvolution blur kernel beforehand. Given an input blurred image **B**, the image formation model can be constructed as the following:

$$B = K \otimes L + N$$



The Goal of Your Project

This is a very challenging problem in that both the kernel and the latent (sharp) image is unknown in the estimation process. Given these two random variables, there can be an infinite number of solutions to solve for, for the output sharp image. To solve this, we can use **natural image statistics**. Given an input blurred image by the user, we can analyze the image in the gradient domain and solve for a simple **Bayesian** equation that will return the maximum marginal probabilities for both the kernel and the latent image. First, the blurred image is passed through an image preprocessing block to remove noise and linearize the pixel intensities. Then using the image statistics, a variational Bayesian inference is conducted. Finally, using the inferred kernel, the blurred image is deconvolved to retrieve the latent image.

The Goal of Your Project

What motivated you to do this project?

This project is unique in that it combines domain knowledge in both the computational photography space as well as the data science space. Being a strong believer in data science and the benefits that it can bring to almost any problem, I was intrigued by the computation process of this specific implementation. Before this project, I never had the opportunity to implement Bayesian statistics to a real-life problem so this was a challenge I was excited to tackle.

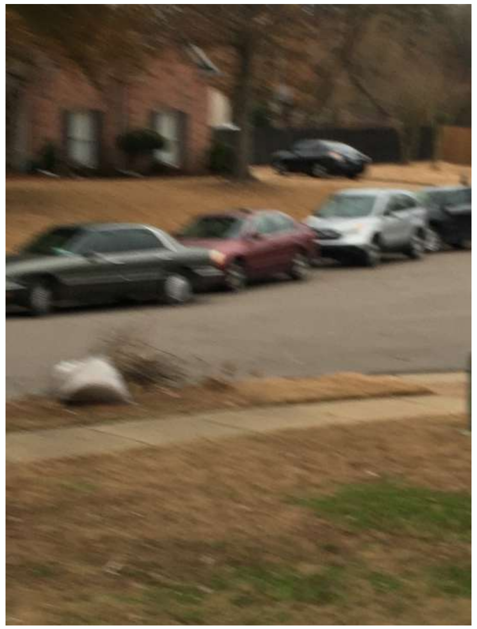
In my previous experience with the course work and projects, I was familiar with convolution and the idea of blurring an image using a gaussian kernel. This project implements image deconvolution, which is an inverse process of convolution. I was interested in learning this step for a better and more intuitive understanding of the applications of the method in a larger scope.

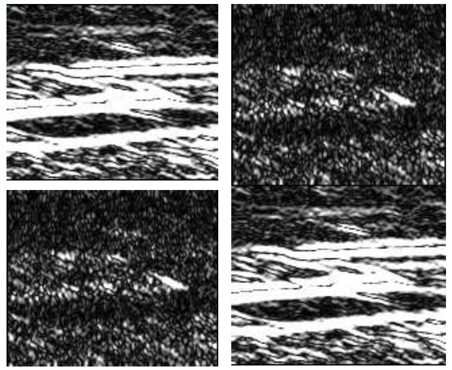
Scope Changes

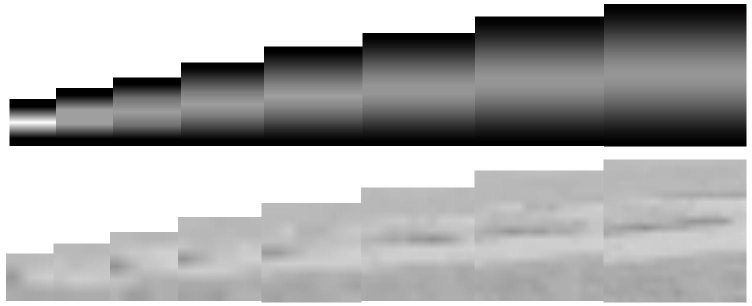
- Did you run into issues that required you to change project scope from your proposal? Give a detailed explanation of what changed

Yes. There was a major issue related to the variational Bayesian inference for this project that was initially mentioned in the proposal. First, the probability theory was completely new to me and had to be learned from scratch starting with Bayes Theorem. Second, a proper framework needed to be found so that the inference network can be constructed and run for convergence. In my attempts, I have tried to use BayesPy as the framework because it uses Variational Bayes as the main engine for the inference as compared to other frameworks like PyMC3, which uses Markov Chain Monte Carlo methods. Because the inference is a major part of the project, if this fails and the kernel output is incorrect, the kernel estimation can distort the latent image, giving bad results as the output. I will be detailing the failed attempts and *interesting* results shortly. To account for this failed implementation, I have included results with output of reasonable quality using other methods of kernel estimation. I have also extended the scope of this project to handle videos along with images, which will be detailed as well.

Showcase & High-Level View


Input: Blurry Image


Gradient Analysis


Multi-Scale Approach

User Patch Selection

Variational Bayesian Inference

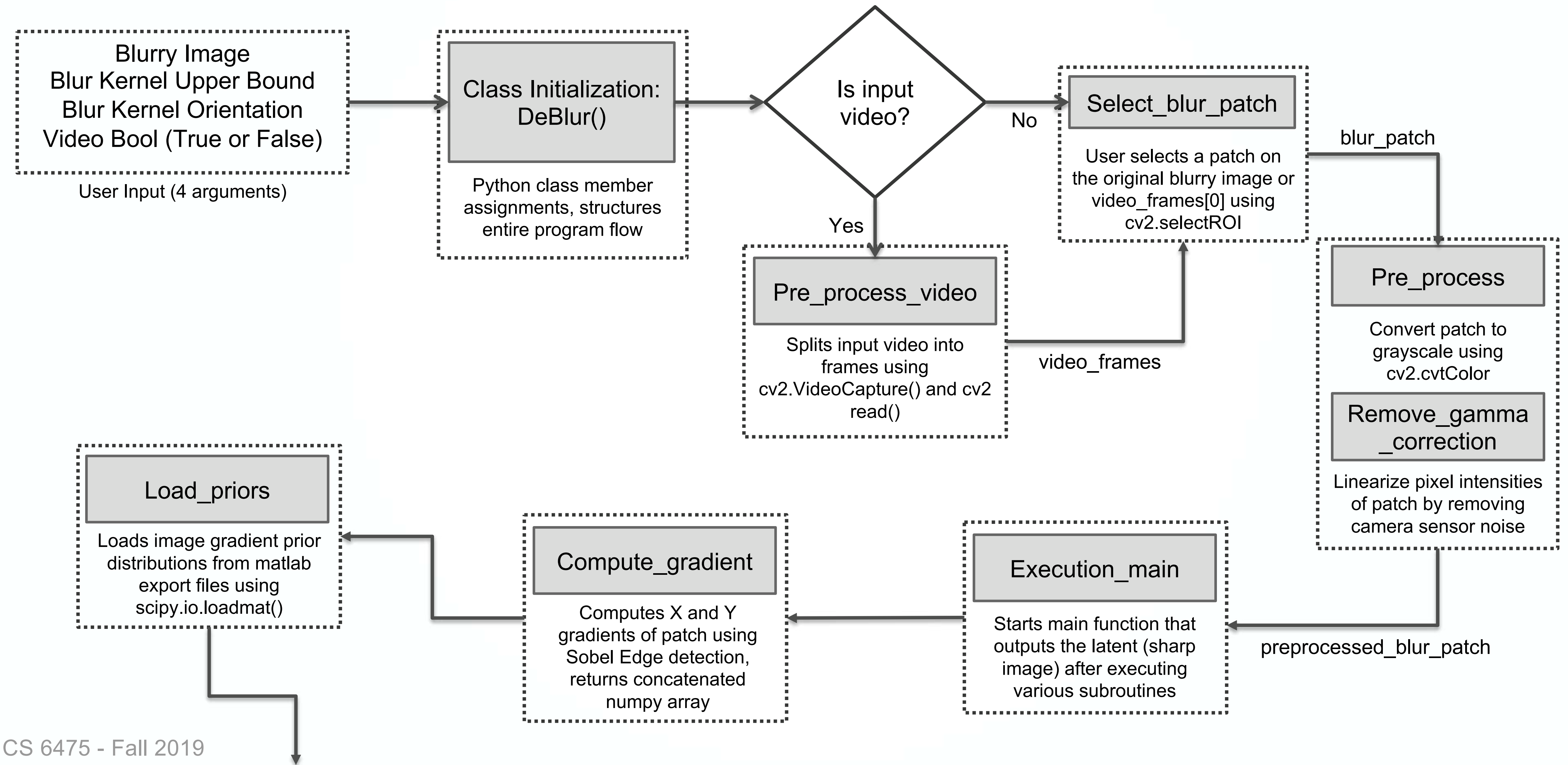
$$p(\mathbf{K}, \nabla \mathbf{L}_p | \nabla \mathbf{P}) \propto p(\nabla \mathbf{P} | \mathbf{K}, \nabla \mathbf{L}_p) p(\nabla \mathbf{L}_p) p(\mathbf{K})$$

Richardson Lucy Deconvolution

+ Post Processing

Output: Sharper Image

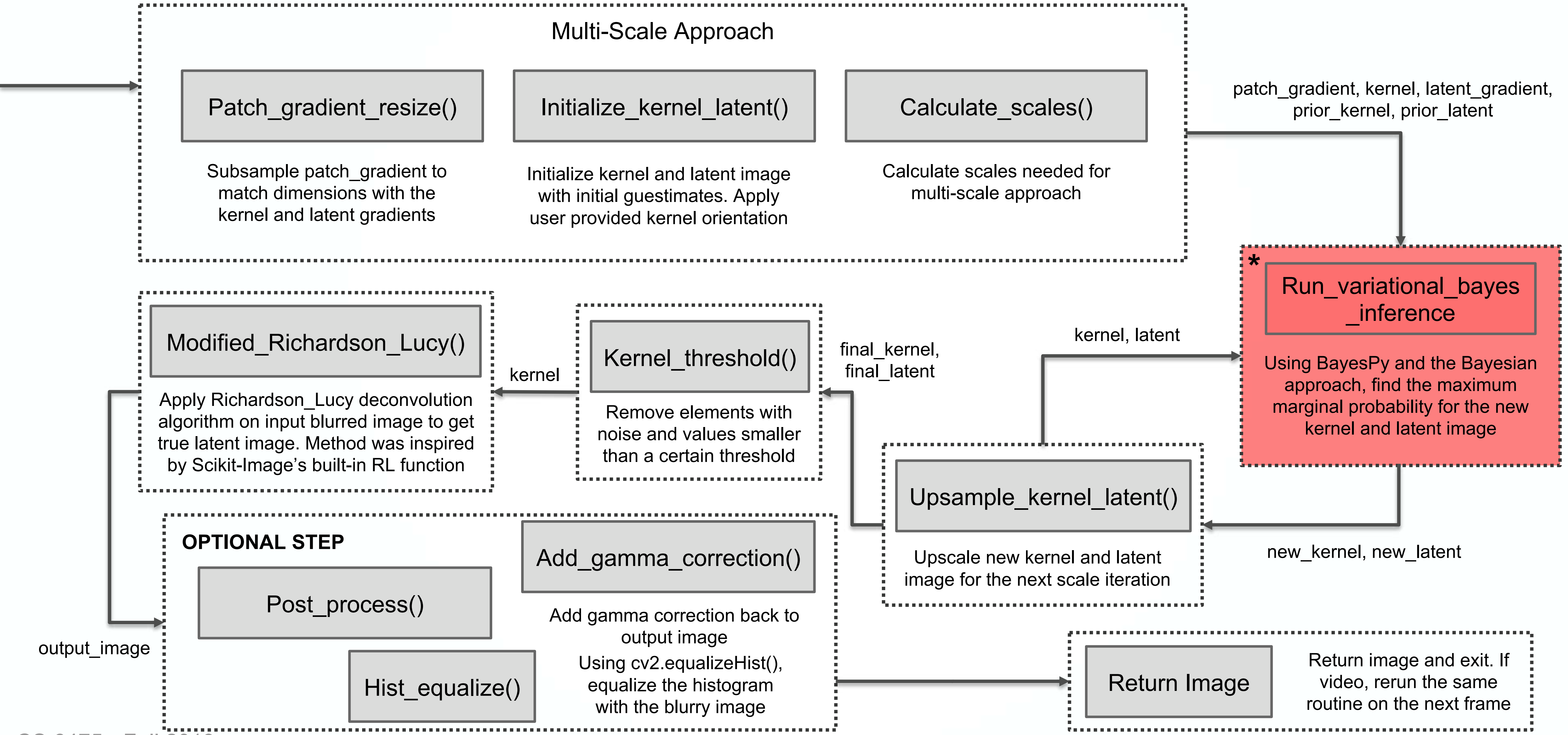
Project Pipeline



Blurry Image
Blur Kernel Upper Bound
Blur Kernel Orientation
Video Bool (True or False)
User Input (4 arguments)
Class Initialization: DeBlur()
Python class member assignments, structures entire program flow
Is input video?
No
Yes
Select_blur_patch
User selects a patch on the original blurry image or video_frames[0] using cv2.selectROI
blur_patch
Pre_process_video
Splits input video into frames using cv2.VideoCapture() and cv2 read()
video_frames
Pre_process
Convert patch to grayscale using cv2.cvtColor
Remove_gamma _correction
Linearize pixel intensities of patch by removing camera sensor noise
preprocessed_blur_patch
Execution_main
Starts main function that outputs the latent (sharp image) after executing various subroutines
Compute_gradient
Computes X and Y gradients of patch using Sobel Edge detection, returns concatenated numpy array
Load_priors
Loads image gradient prior distributions from matlab export files using scipy.io.loadmat()

Project Pipeline

* Semi-Completed, but probabilities are not accurate. Tweaks need to be made to the Bayesian Model



Subsample patch_gradient to match dimensions with the kernel and latent gradients

Initialize kernel and latent image with initial guestimates. Apply user provided kernel orientation

Calculate scales needed for multi-scale approach

Using BayesPy and the Bayesian approach, find the maximum marginal probability for the new kernel and latent image

Apply Richardson_Lucy deconvolution algorithm on input blurred image to get true latent image. Method was inspired by Scikit-Image's built-in RL function

Remove elements with noise and values smaller than a certain threshold

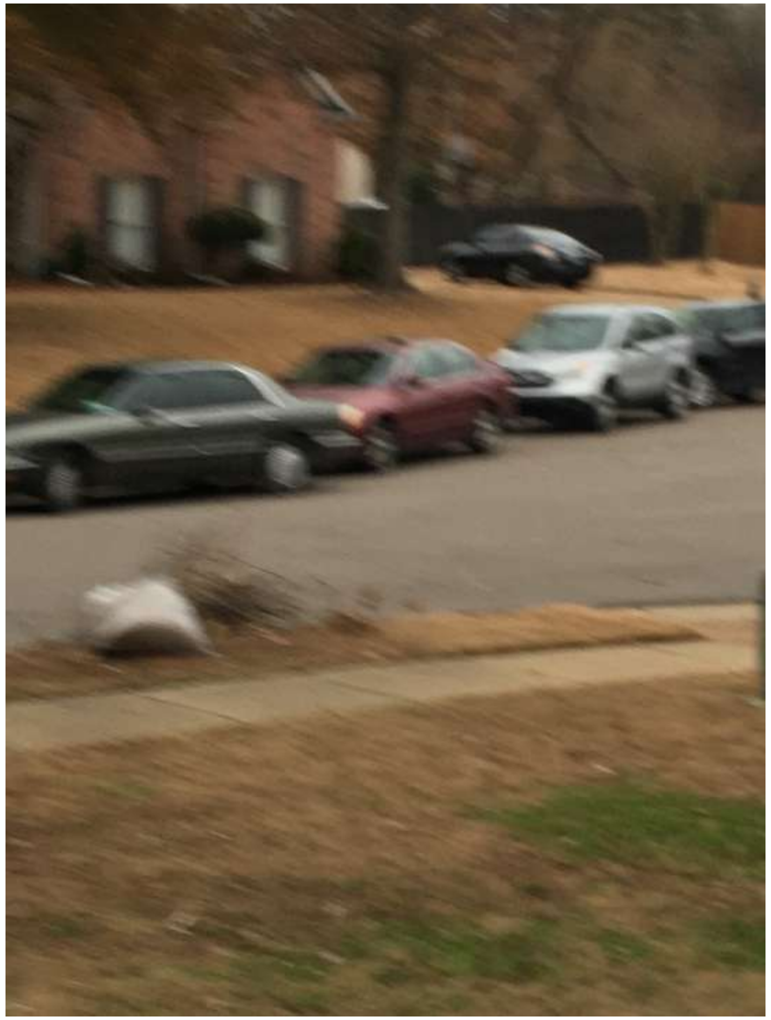
Upscale new kernel and latent image for the next scale iteration

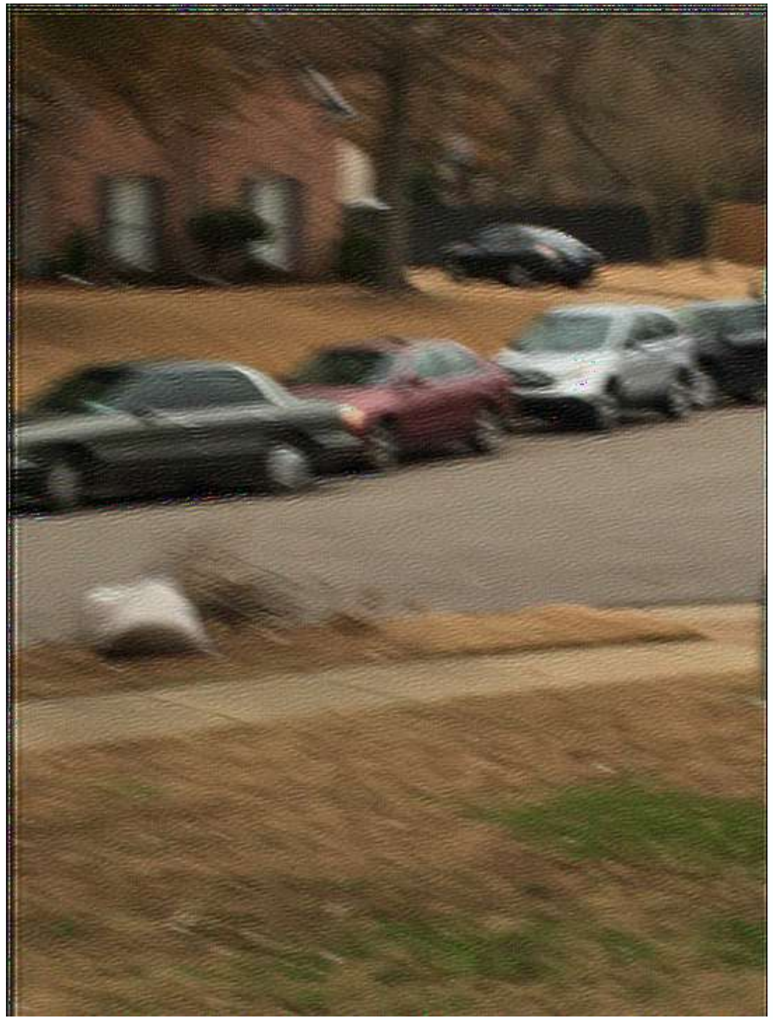
Add gamma correction back to output image

Using cv2.equalizeHist(), equalize the histogram with the blurry image

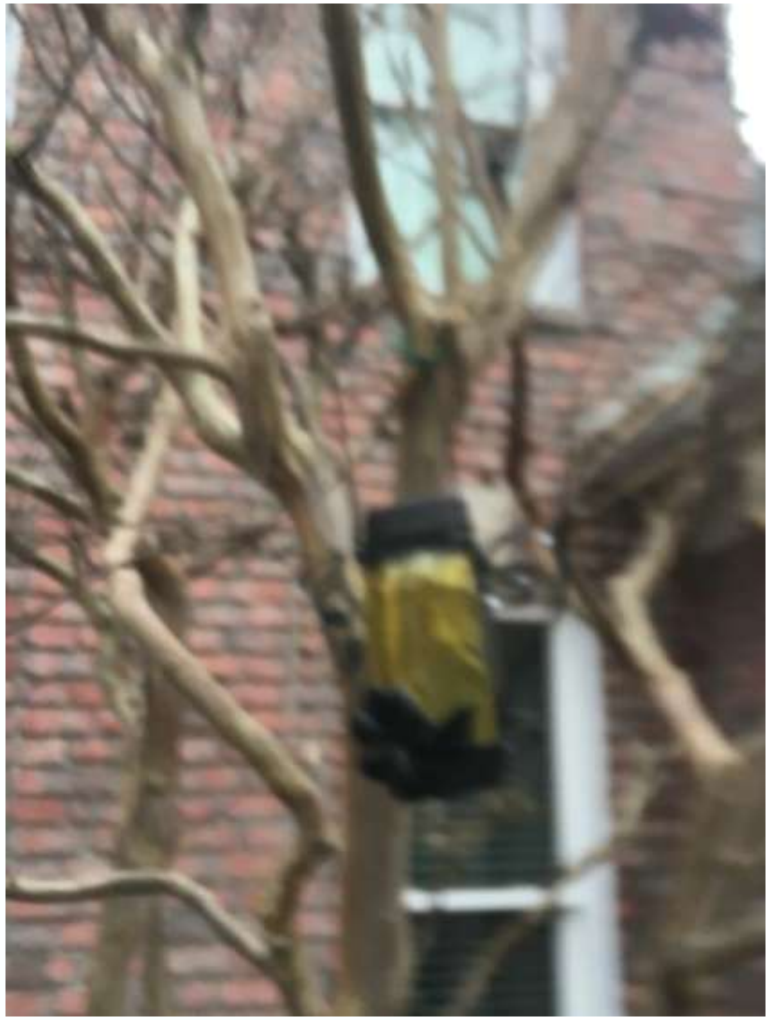
Return image and exit. If video, rerun the same routine on the next frame

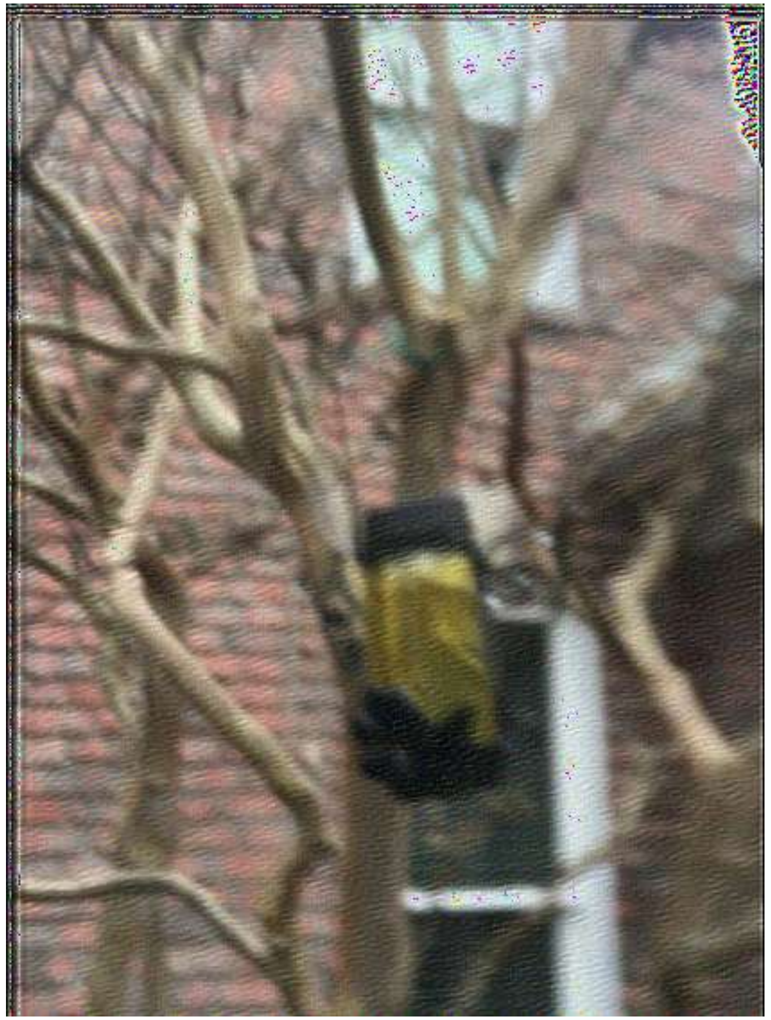
Demonstration: Result Set 1


Input: Blurry Image

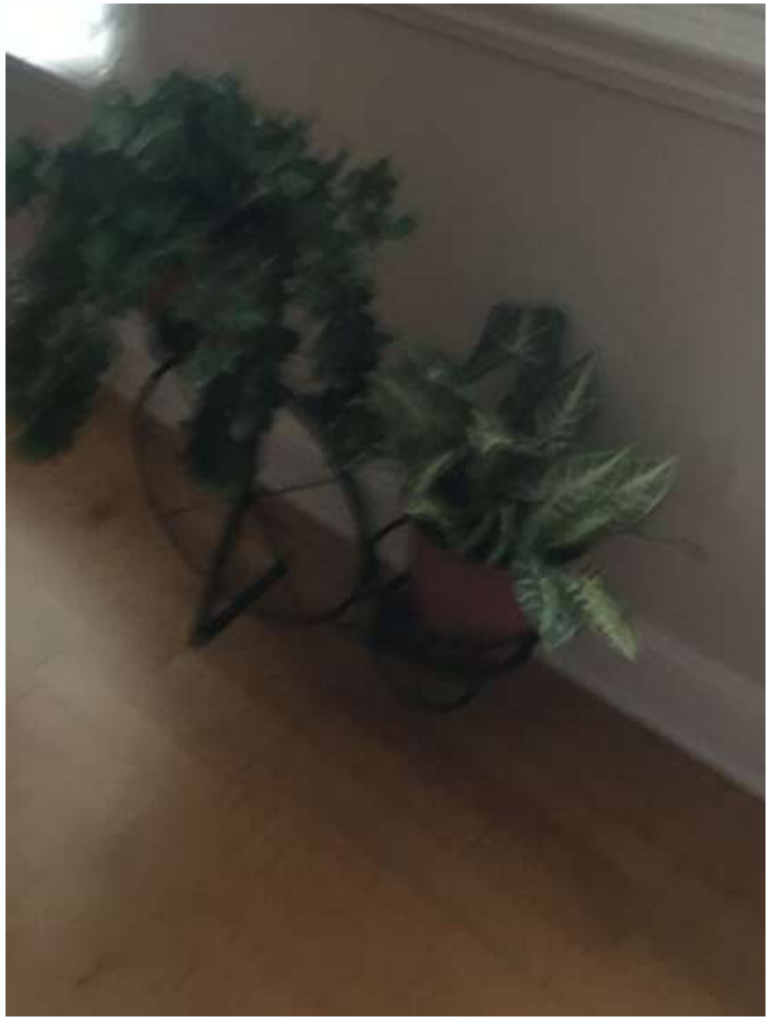

Output: Latent (Sharp) Image

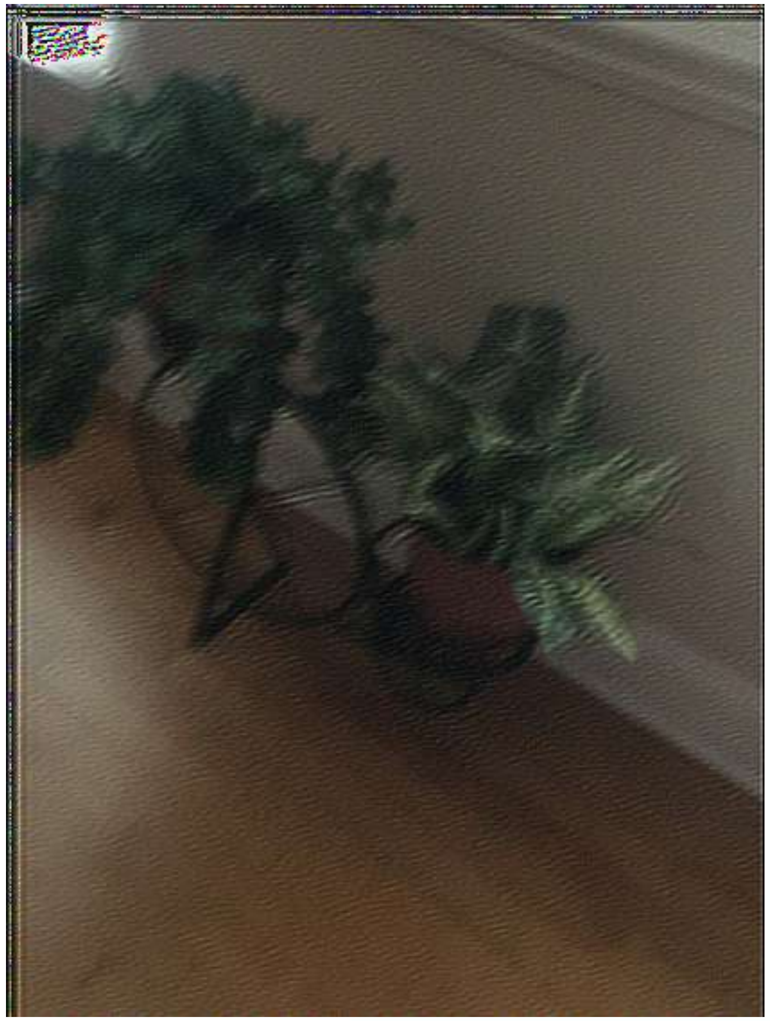
Demonstration: Result Set 2


Input: Blurry Image

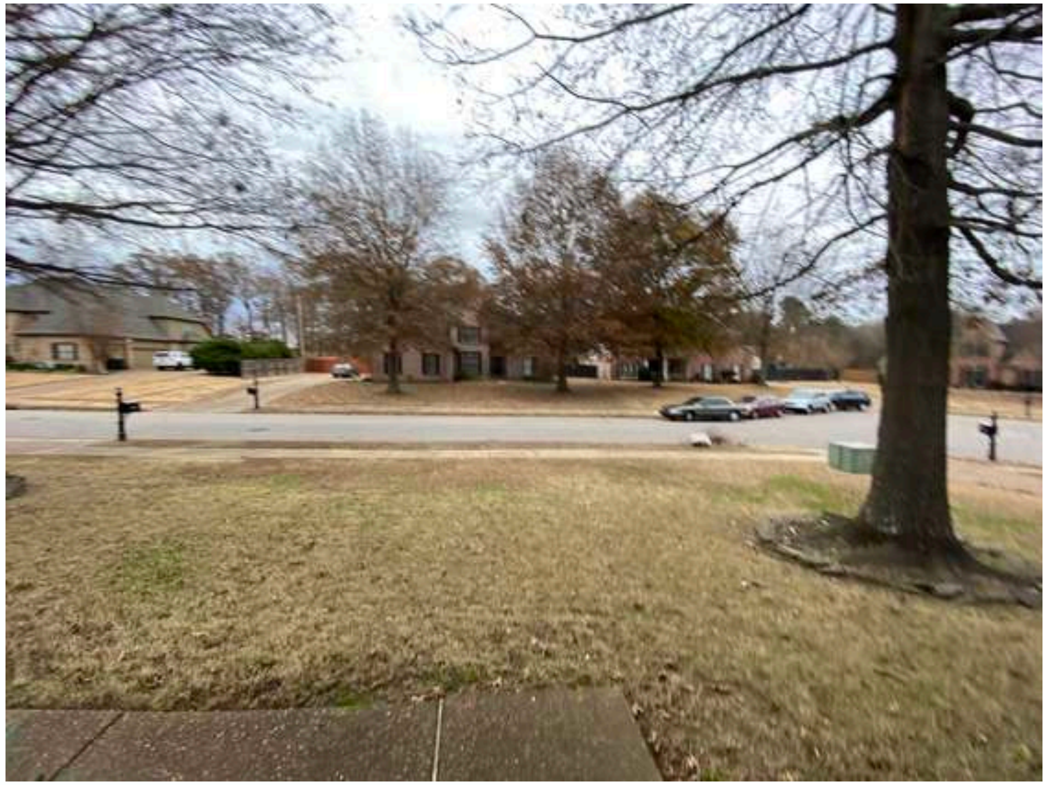

Output: Latent (Sharp) Image

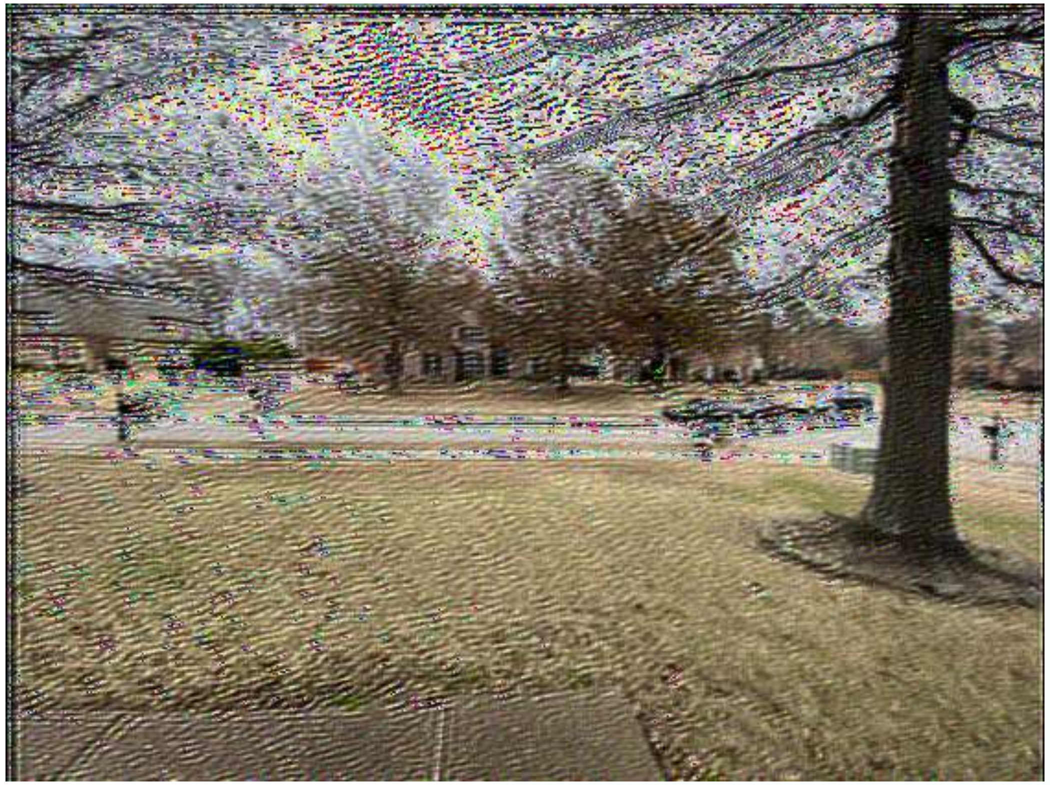
Demonstration: Result Set 3


Input: Blurry Image

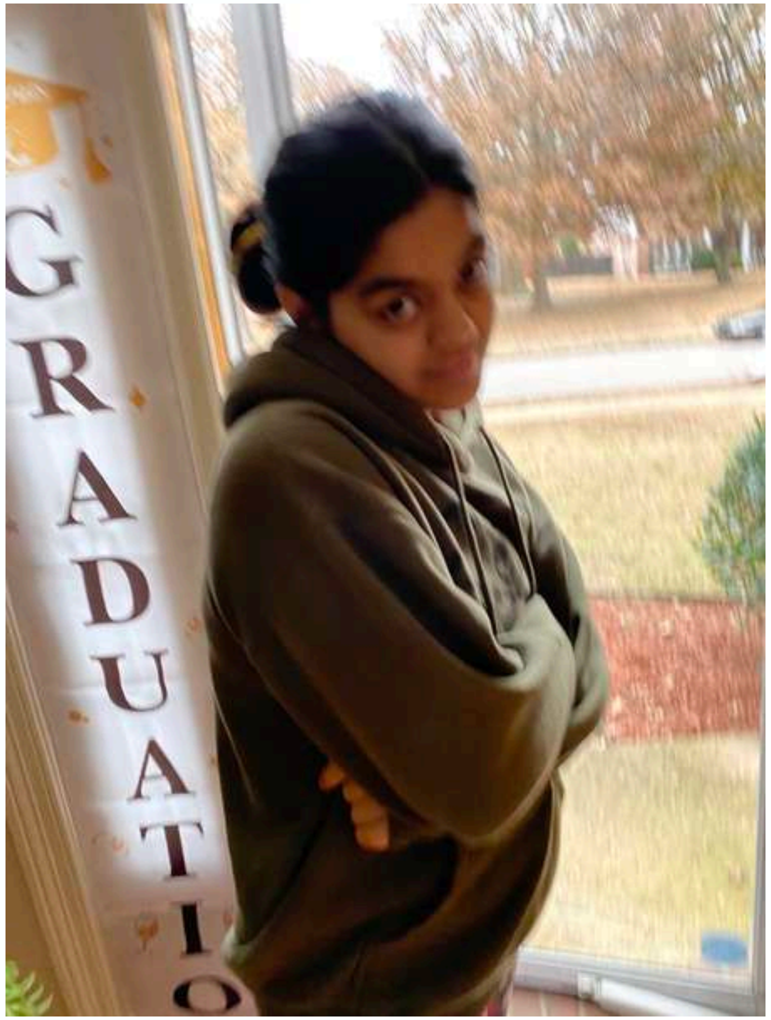

Output: Latent (Sharp) Image

Demonstration: Result Set 4

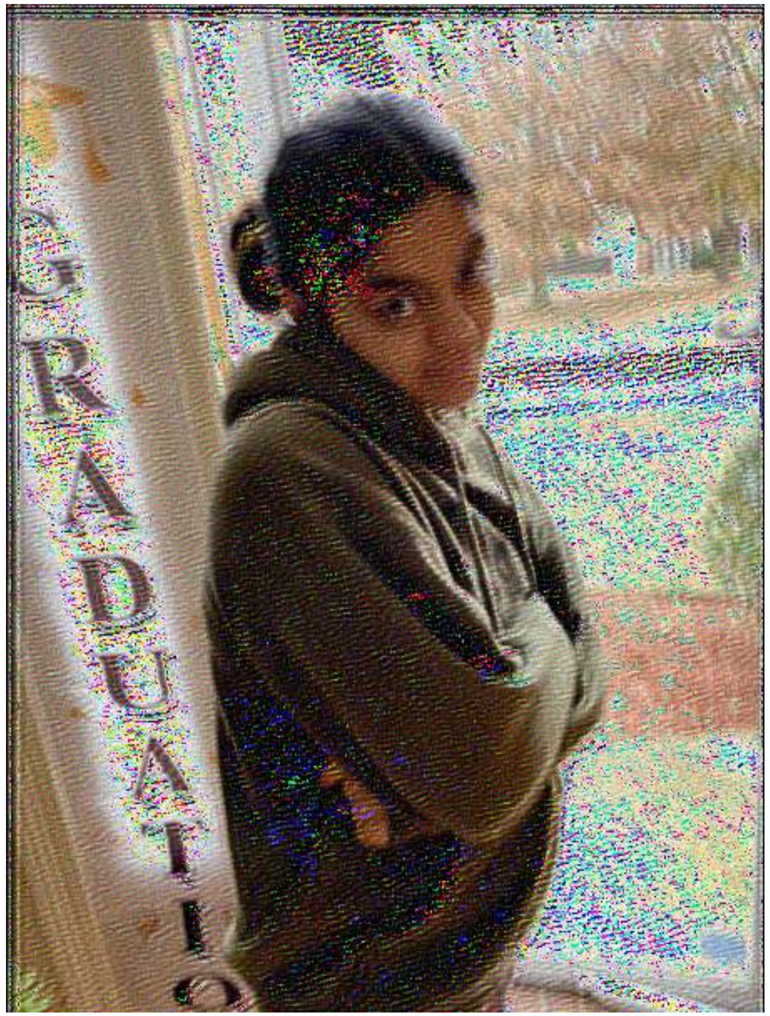

Input: Blurry Image


Output: Latent (Sharp) Image

Demonstration: Result Set 5

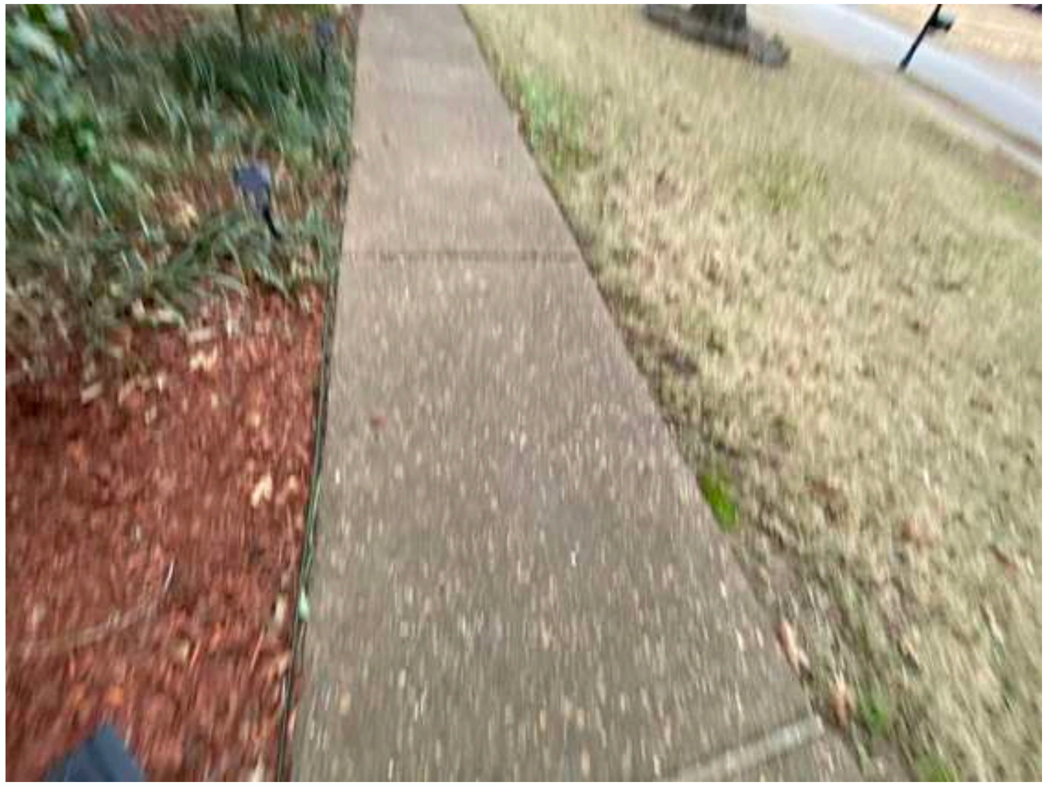


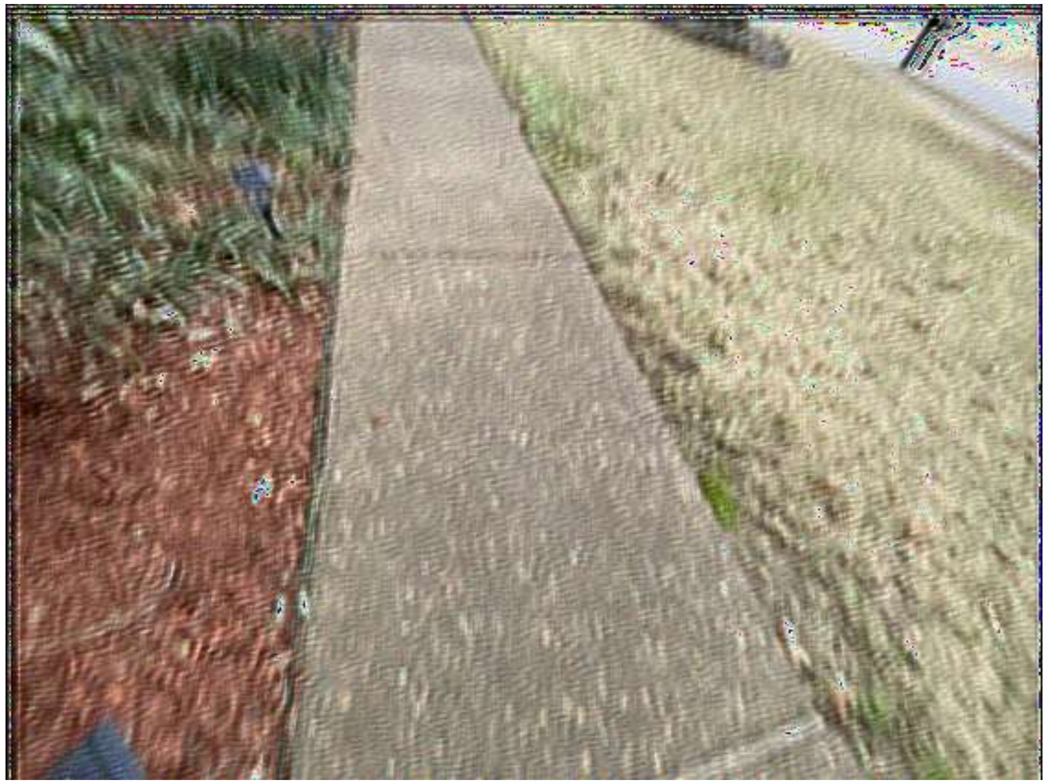
Input: Blurry Image



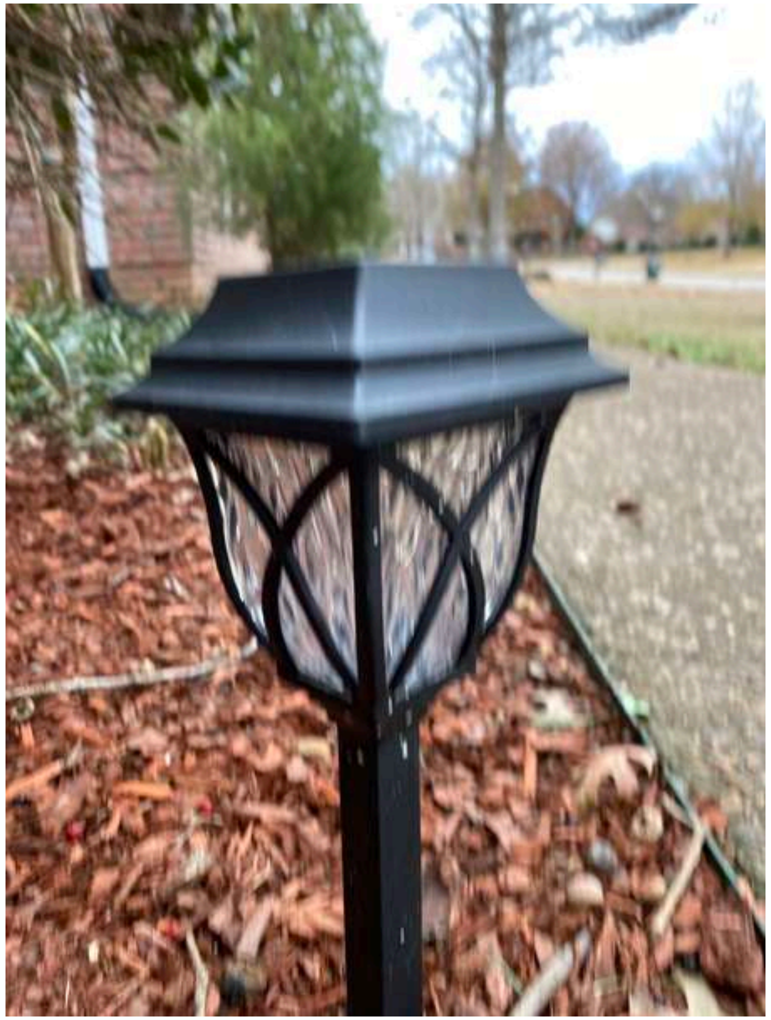
Output: Latent (Sharp) Image

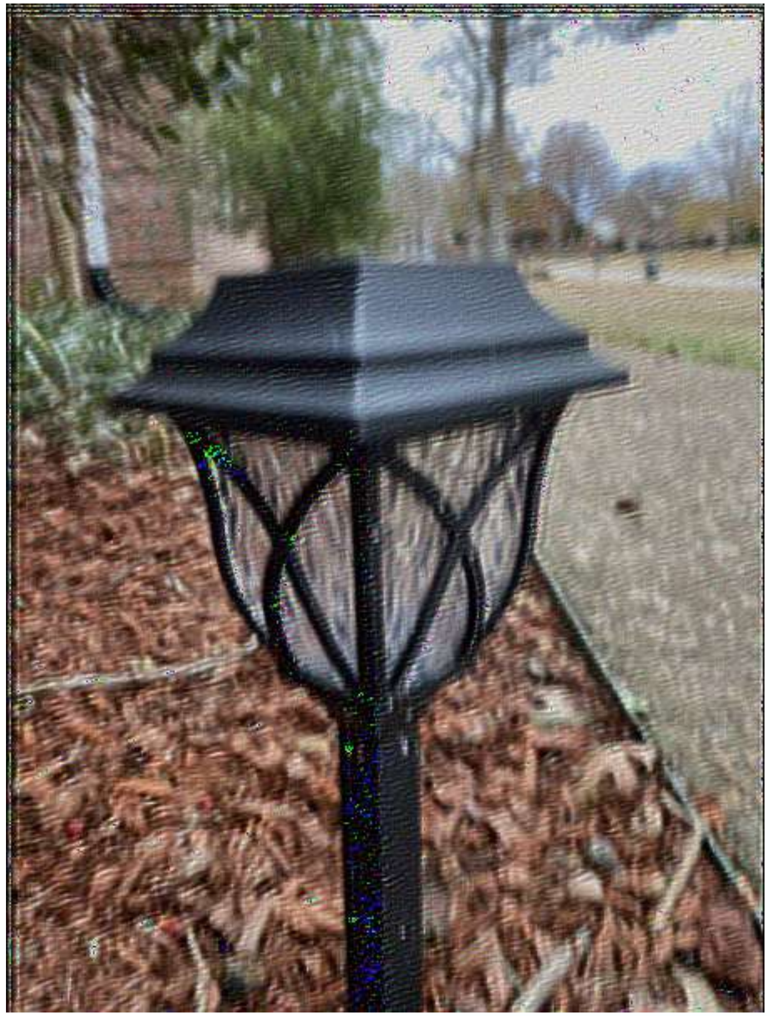
Demonstration: Result Set 6


Input: Blurry Image

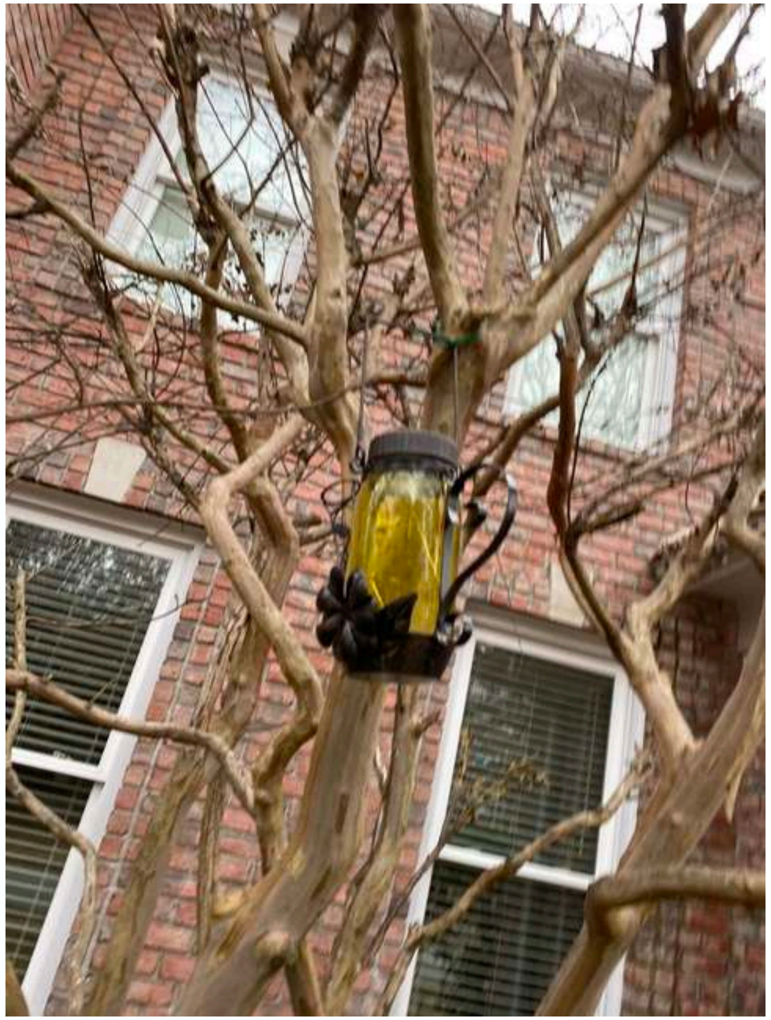

Output: Latent (Sharp) Image

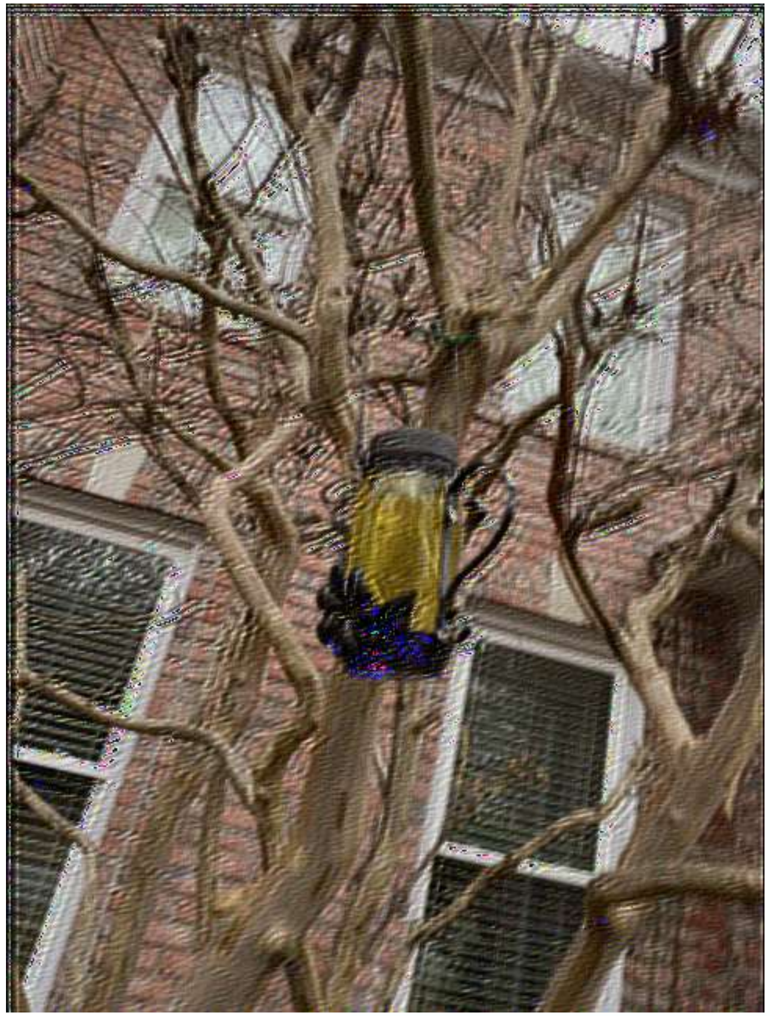
Demonstration: Result Set 7


Input: Blurry Image

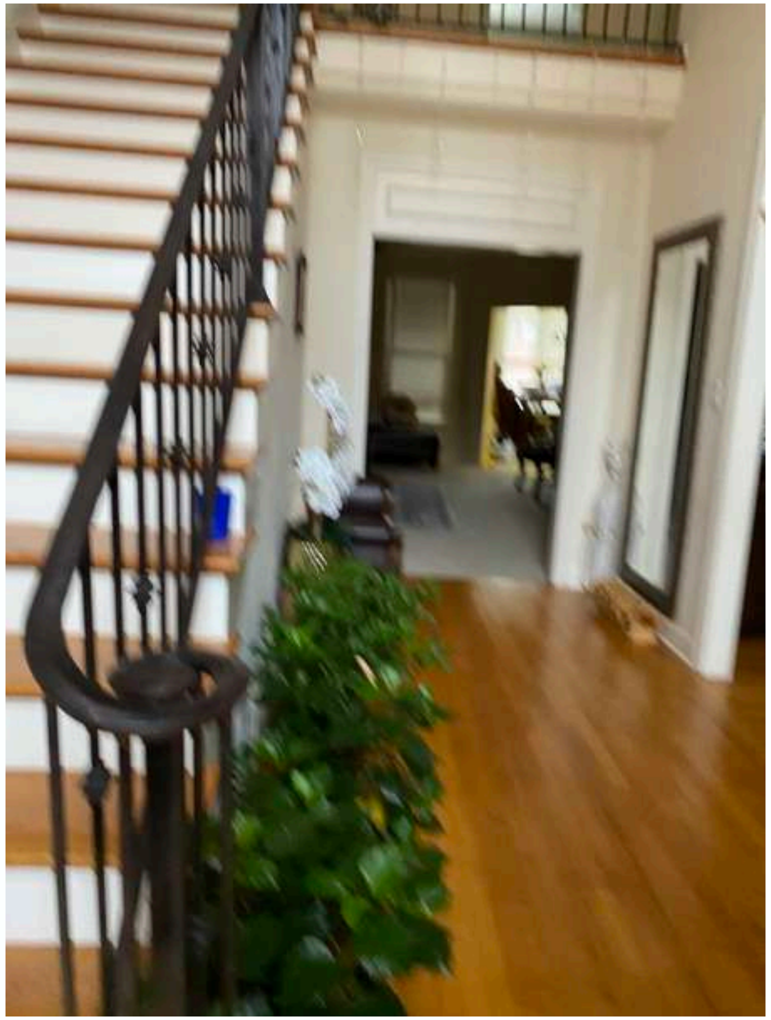

Output: Latent (Sharp) Image

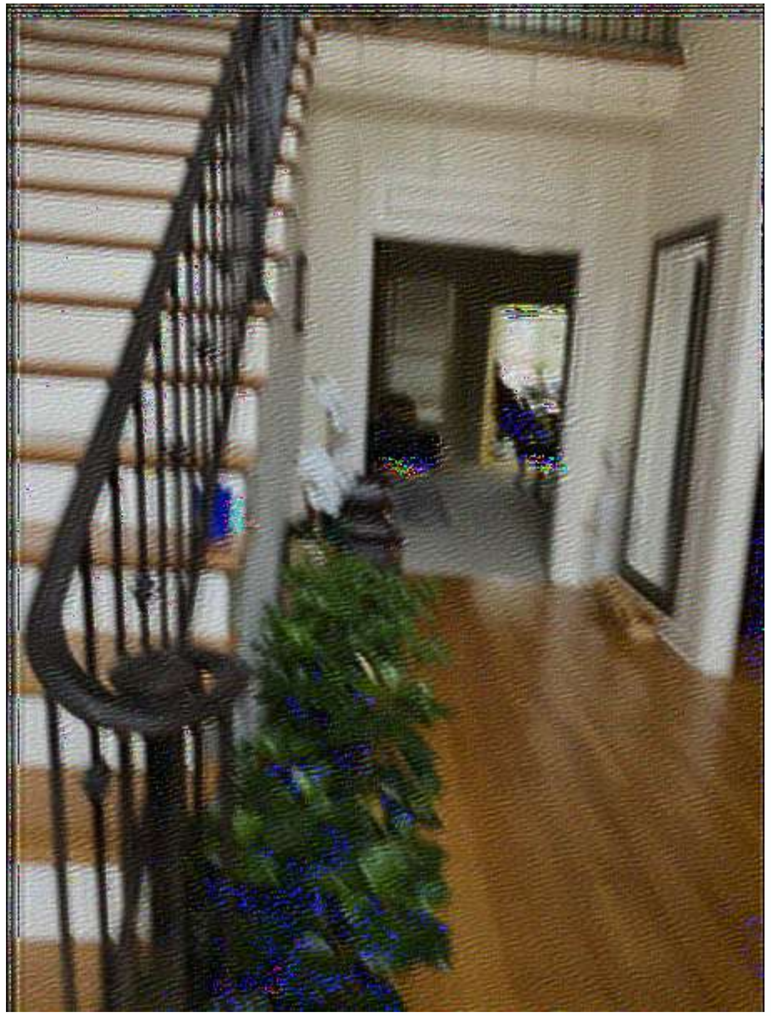
Demonstration: Result Set 8


Input: Blurry Image

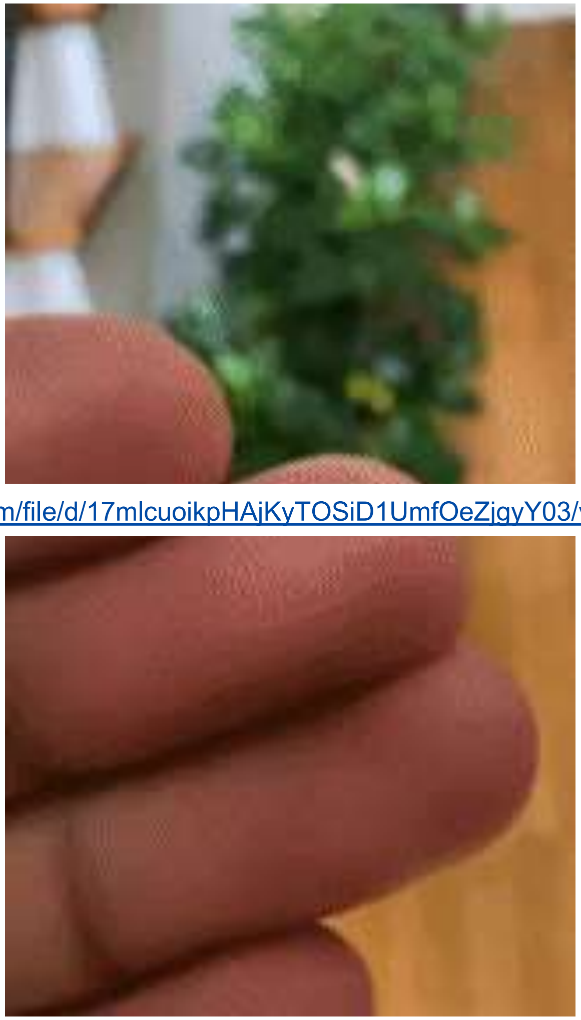

Output: Latent (Sharp) Image

Demonstration: Result Set 9


Input: Blurry Image

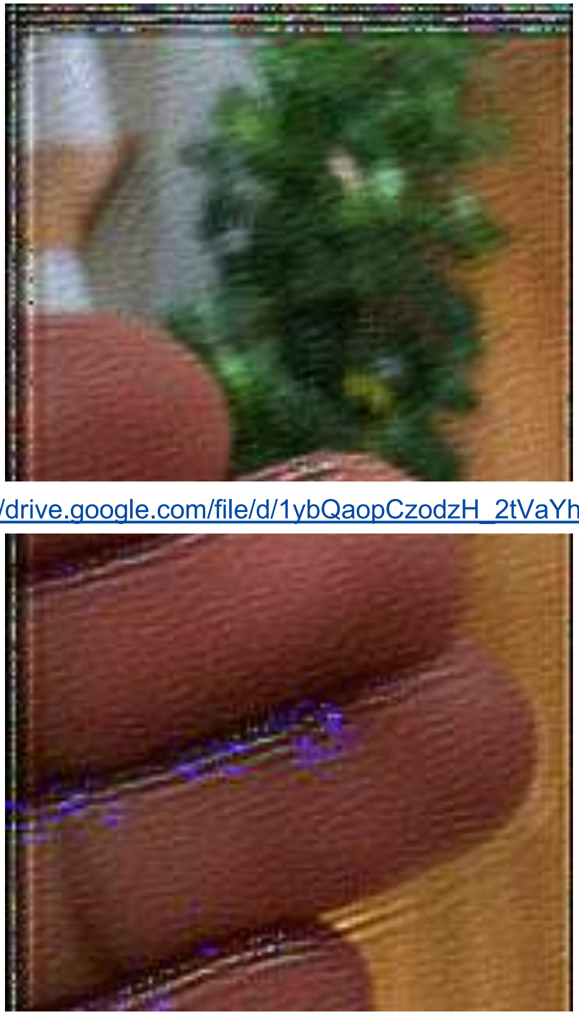

Output: Latent (Sharp) Image

Demonstration: Result Set 10 - Video



https://drive.google.com/file/d/17mlcuoikpHAjKyTOSiD1UmfOeZjgyY03/view

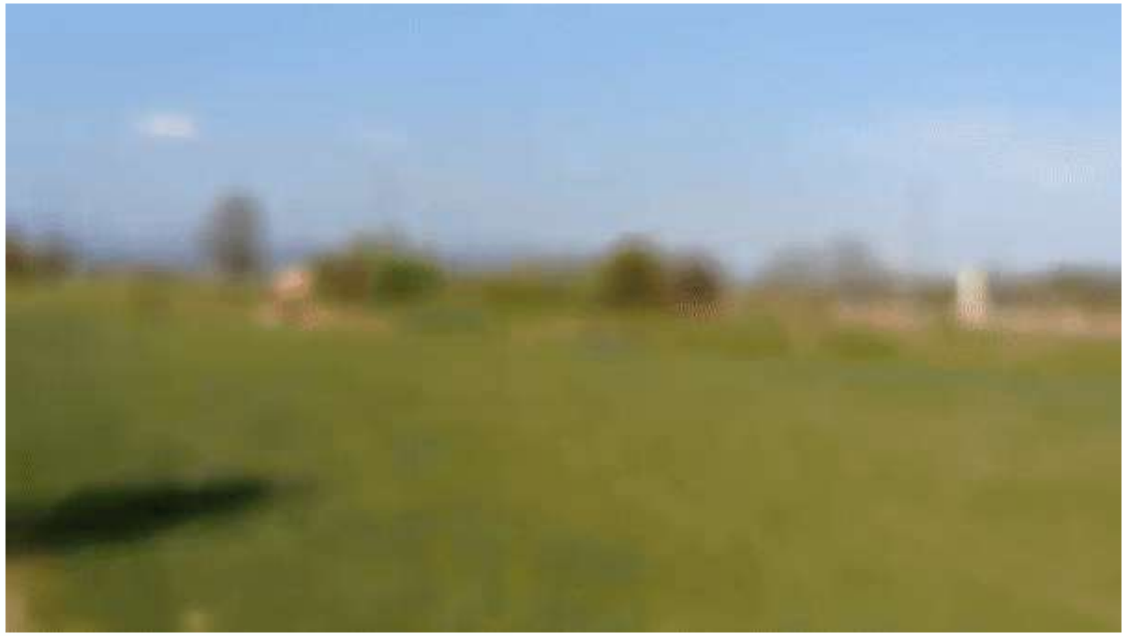
Input: Blurry Video

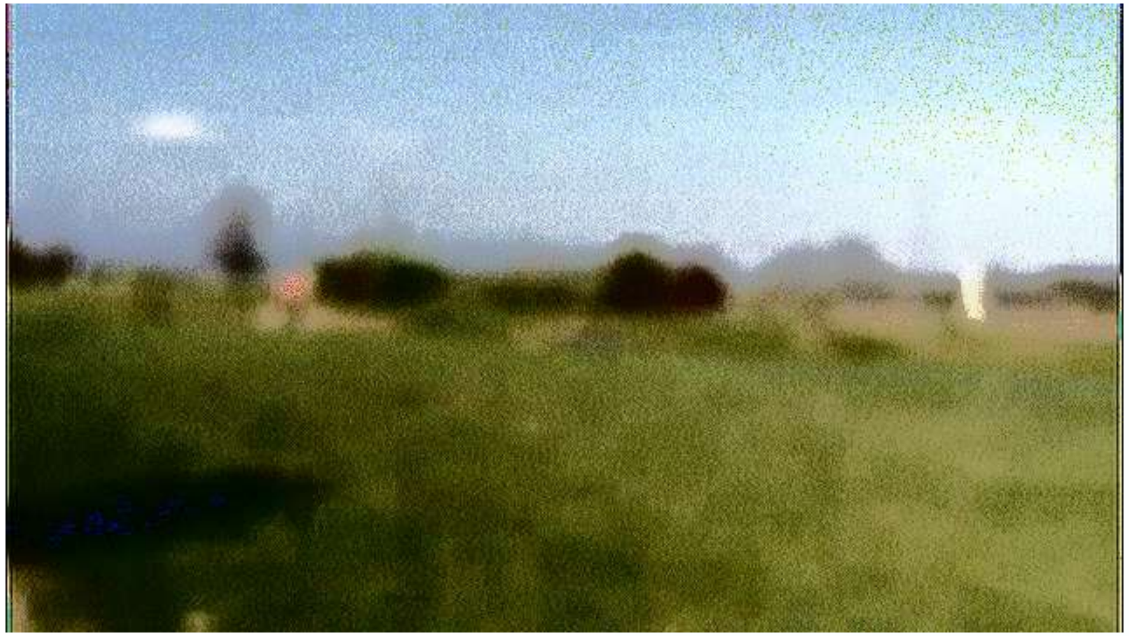


https://drive.google.com/file/d/1ybQaopCzodzH_2tVaYh0qrGI75WALjsB/view

Output: Latent (Sharp) Image

Demonstration: Result Set 11 - Video


Input: Blurry Video


Output: Latent (Sharp) Image

https://drive.google.com/file/d/1oaB6ugm_fBo9_g-RyKmv3ppVuCqpp9VQ/view

https://drive.google.com/file/d/1WuMNA9rcZ-qzB-CSyVBL0K1UnE5WpF69/view

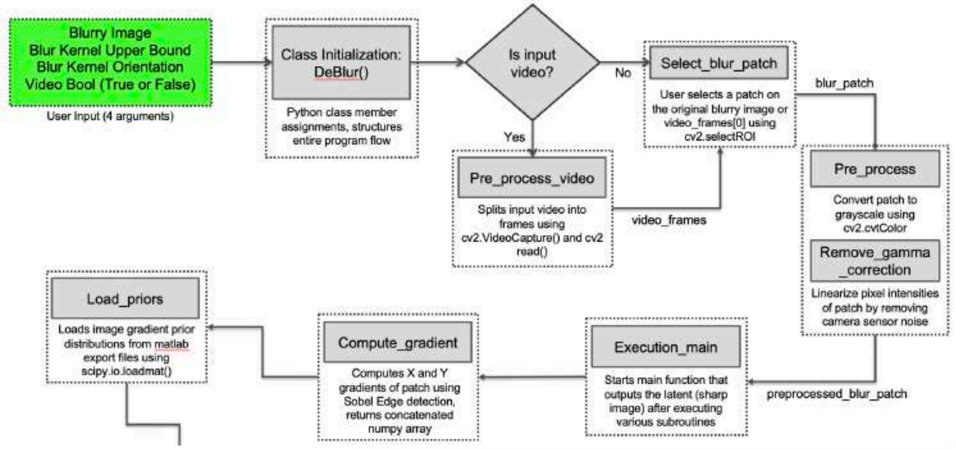
Project Development

The next several slides will detail a descriptive breakdown of the project from both the narrative standpoint and a low-level functional code description. The purpose of this is to get an intuitive understanding of the program flow of the project. Along the way, there will be discussion on both successes and failures on certain aspects of the code as well as potential pitfalls that may exist that output certain results.

On the top right corner of the slide, the project pipeline will be shown. The current step that is being discussed will be highlighted in green.

For clarity, the majority of this project is based on the paper, *Removing Camera Shake from a Single Photograph* by Fergus et al., however some deviations are taken. This will be discussed further.

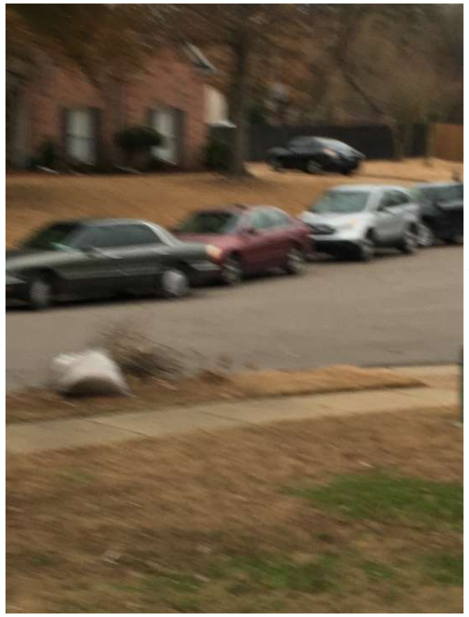
Let's begin!


Blurry Image
Blur Kernel Upper Bound
Blur Kernel Orientation
Video Bool (True or False)
User Input (4 arguments)
Class Initialization: DeBlur()
Python class member assignments, structures entire program flow
Is input video?
No
Yes
Select_blur_patch
User selects a patch on the original blurry image or video_frames[0] using cv2.selectROI
blur_patch
Pre_process_video
Splits input video into frames using cv2.VideoCapture() and cv2 read()
video_frames
Pre_process
Convert patch to grayscale using cv2.cvtColor
Remove_gamma_correction
Linearize pixel intensities of patch by removing camera sensor noise
preprocessed_blur_patch
Execution_main
Starts main function that outputs the latent (sharp image) after executing various subroutines
Compute_gradient
Computes X and Y gradients of patch using Sobel Edge detection, returns concatenated numpy array
Load_priors
Loads image gradient prior distributions from matlab export files using scipy.io.loadmat()

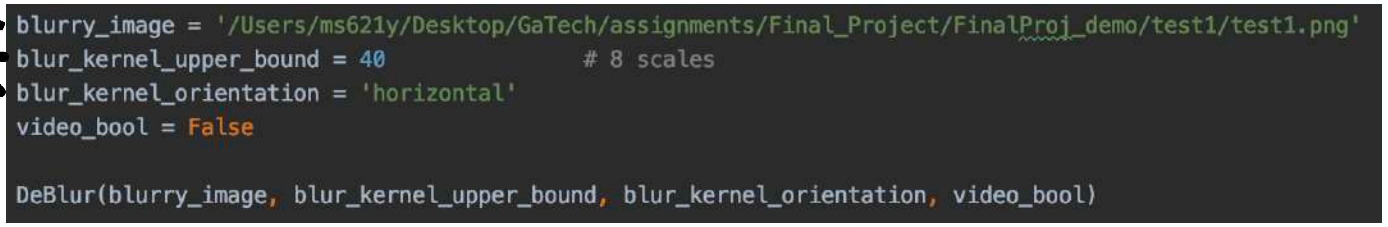
🟩 **Pipeline Progress**

Project Development: Inputs

To begin the project development breakdown, first we need to understand the inputs of this system.



The first argument is the **blurry image** which will be processed for a sharper image.

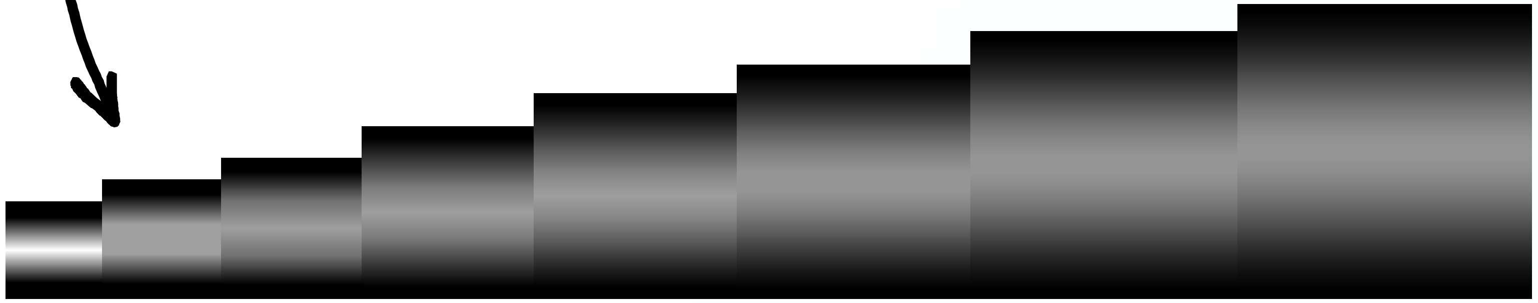


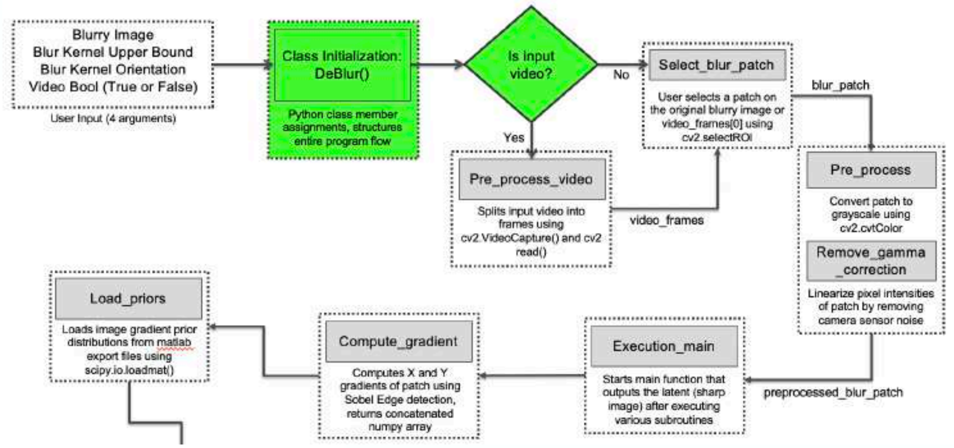
```
blurry_image = '/Users/ms621y/Desktop/GaTech/assignments/Final_Project/FinalProj_demo/test1/test1.png'
blur_kernel_upper_bound = 40              # 8 scales
blur_kernel_orientation = 'horizontal'
video_bool = False

DeBlur(blurry_image, blur_kernel_upper_bound, blur_kernel_orientation, video_bool)
```

The fourth argument determines whether or not the input is a video for further logic handling.

The second argument is the kernel upper bound in pixels which will determine how many times we will **scale up** the kernel in the multi-scale approach. Using the third argument, we will initialize the kernel using an initial guess of the orientation of the kernel. This being a **horizontal** kernel in this case.

This is a visualization of the iterative multi-scale approach where the estimated blur kernel is being scaled up.





Blurry Image
Blur Kernel Upper Bound
Blur Kernel Orientation
Video Bool (True or False)
User Input (4 arguments)
Class Initialization: DeBlur()
Python class member assignments, structures entire program flow
Is input video?
No
Yes
Select_blur_patch
User selects a patch on the original blurry image or video_frames[0] using cv2.selectROI
blur_patch
Pre_process_video
Splits input video into frames using cv2.VideoCapture() and cv2 read()
video_frames
Pre_process
Convert patch to grayscale using cv2.cvtColor
Remove_gamma_correction
Linearize pixel intensities of patch by removing camera sensor noise
preprocessed_blur_patch
Execution_main
Starts main function that outputs the latent (sharp image) after executing various subroutines
Compute_gradient
Computes X and Y gradients of patch using Sobel Edge detection, returns concatenated numpy array
Load_priors
Loads image gradient prior distributions from matlab export files using scipy.io.loadmat()

🟩 **Pipeline Progress**

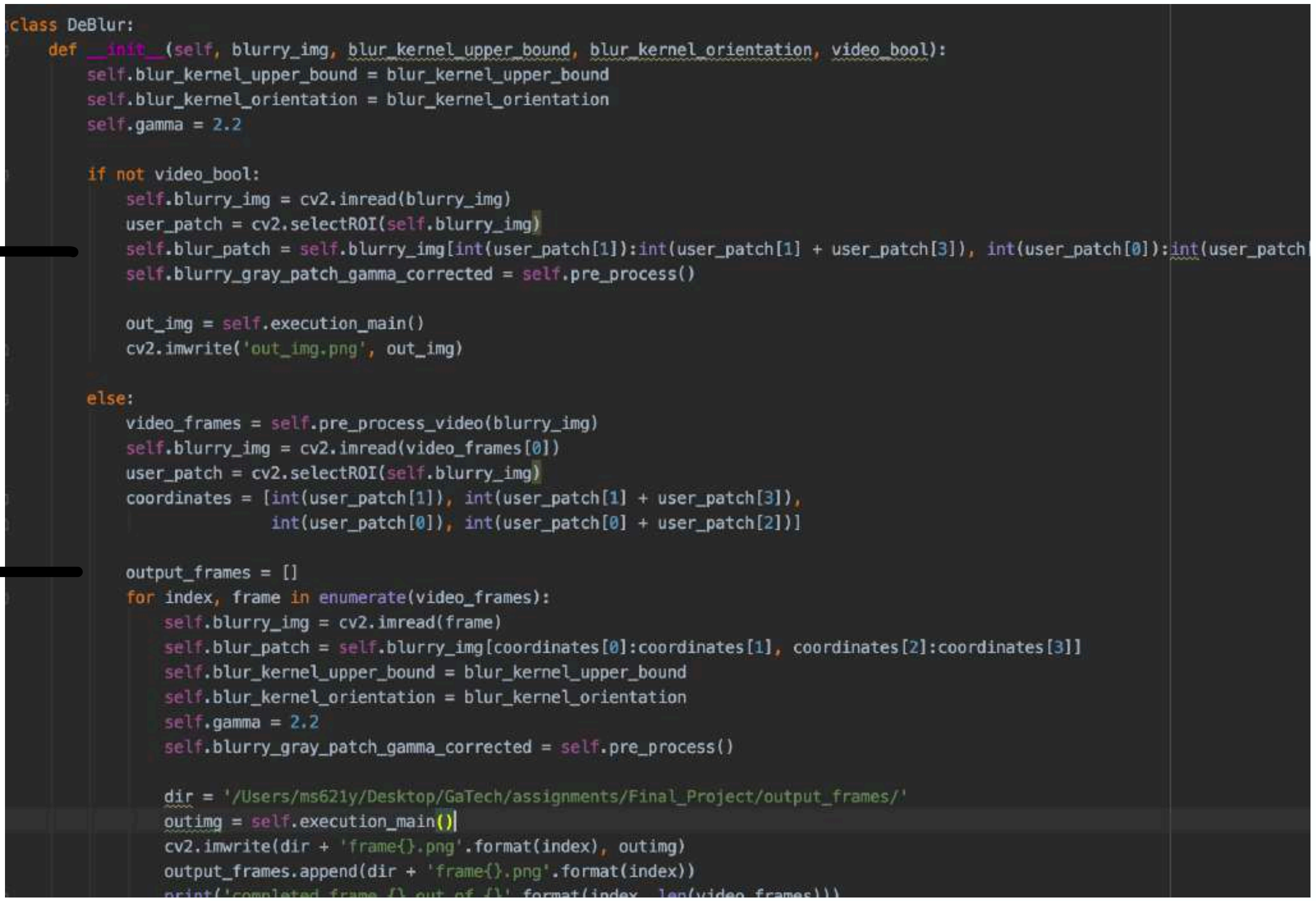
Project Development: Initialization

This step involves Pythonic syntax and initialization of the DeBlur() class. The input arguments from the previous step is assigned to the class member variables for later use by other subroutines. The packages being used here include OpenCV, Numpy, and Scipy.

Along with this, a decision is taken based on the video_bool flag.

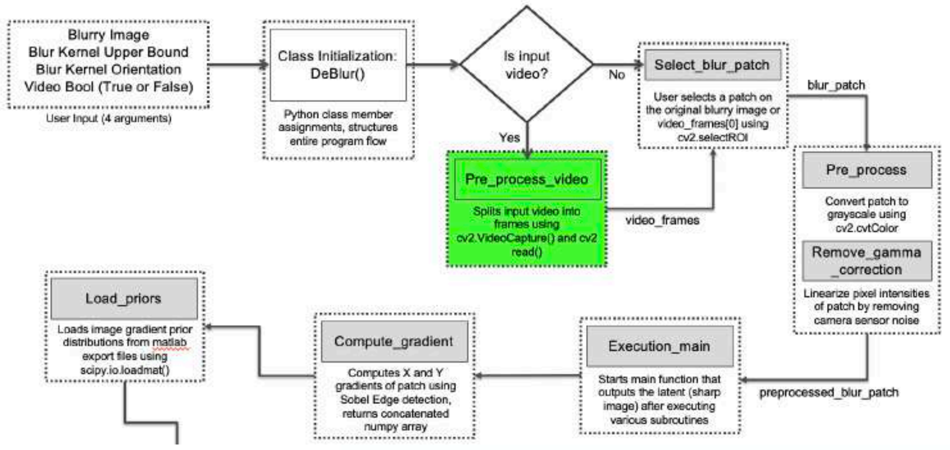
If False, the input image will be read by cv2.imread() and then passed along to cv2.selectROI where the user can proceed to select a blur patch.

If True, the input video will be passed along to the function pre_process_video() where the video will be broken down into its individual frames. Once the frames are received, we will continue to process the frames in the same way as we would with a single image, but instead, iterate for all of the frames.



```python
class DeBlur:
    def __init__(self, blurry_img, blur_kernel_upper_bound, blur_kernel_orientation, video_bool):
        self.blur_kernel_upper_bound = blur_kernel_upper_bound
        self.blur_kernel_orientation = blur_kernel_orientation
        self.gamma = 2.2

        if not video_bool:
            self.blurry_img = cv2.imread(blurry_img)
            user_patch = cv2.selectROI(self.blurry_img)
            self.blur_patch = self.blurry_img[int(user_patch[1]):int(user_patch[1] + user_patch[3]), int(user_patch[0]):int(user_patch
            self.blurry_gray_patch_gamma_corrected = self.pre_process()

            out_img = self.execution_main()
            cv2.imwrite('out_img.png', out_img)

        else:
            video_frames = self.pre_process_video(blurry_img)
            self.blurry_img = cv2.imread(video_frames[0])
            user_patch = cv2.selectROI(self.blurry_img)
            coordinates = [int(user_patch[1]), int(user_patch[1] + user_patch[3]),
                           int(user_patch[0]), int(user_patch[0] + user_patch[2])]

            output_frames = []
            for index, frame in enumerate(video_frames):
                self.blurry_img = cv2.imread(frame)
                self.blur_patch = self.blurry_img[coordinates[0]:coordinates[1], coordinates[2]:coordinates[3]]
                self.blur_kernel_upper_bound = blur_kernel_upper_bound
                self.blur_kernel_orientation = blur_kernel_orientation
                self.gamma = 2.2
                self.blurry_gray_patch_gamma_corrected = self.pre_process()

                dir = '/Users/ms621y/Desktop/GaTech/assignments/Final_Project/output_frames/'
                outimg = self.execution_main()
                cv2.imwrite(dir + 'frame{}.png'.format(index), outimg)
                output_frames.append(dir + 'frame{}.png'.format(index))
                print('completed frame {} out of {}'.format(index, len(video_frames)))
```

Project Development: Video Processing



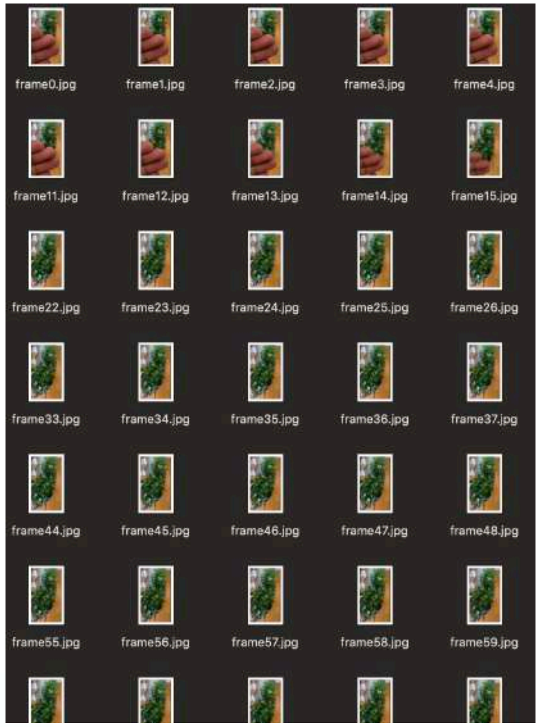
Blurry Image
Blur Kernel Upper Bound
Blur Kernel Orientation
Video Bool (True or False)
User Input (4 arguments)
Class Initialization: DeBlur()
Python class member assignments, structures entire program flow
Is input video?
No
Yes
Select_blur_patch
User selects a patch on the original blurry image or video_frames[0] using cv2.selectROI
blur_patch
Pre_process_video
Splits input video into frames using cv2.VideoCapture() and cv2 read()
video_frames
Pre_process
Convert patch to grayscale using cv2.cvtColor
Remove_gamma_correction
Linearize pixel intensities of patch by removing camera sensor noise
preprocessed_blur_patch
Execution_main
Starts main function that outputs the latent (sharp image) after executing various subroutines
Compute_gradient
Computes X and Y gradients of patch using Sobel Edge detection, returns concatenated numpy array
Load_priors
Loads image gradient prior distributions from matlab export files using scipy.io.loadmat()

🟩 **Pipeline Progress**

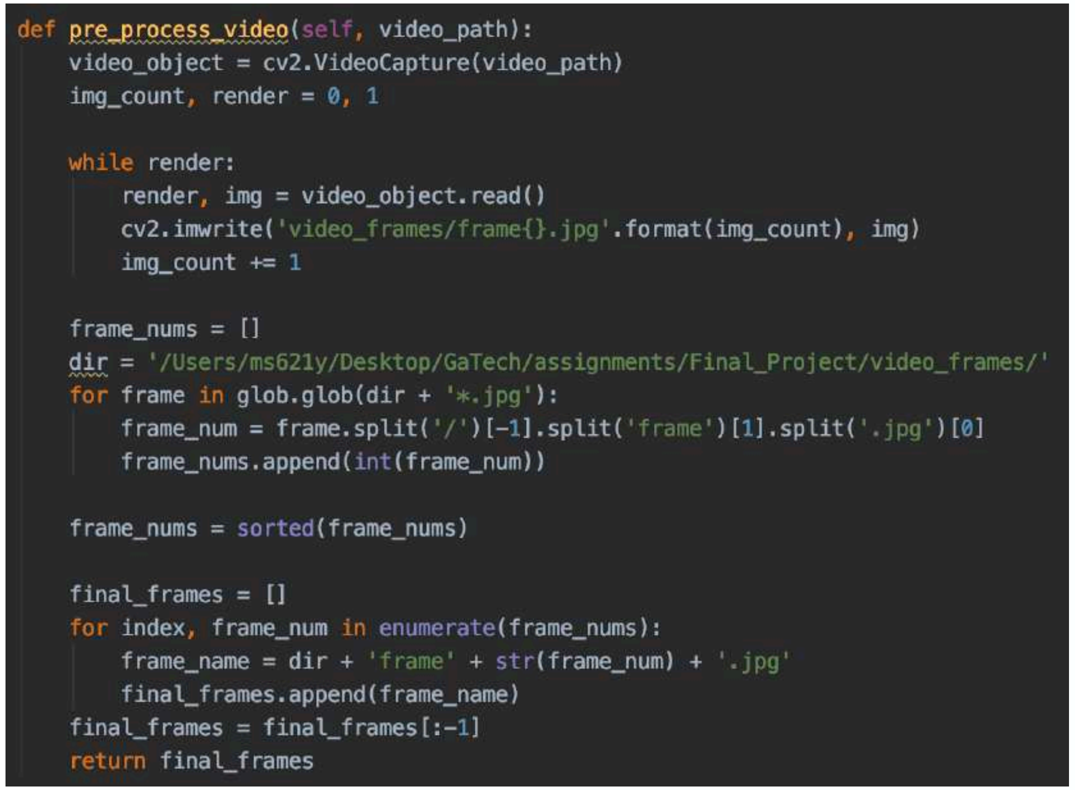
If video_bool is True =>

We will begin to process the input blurry video and break it down into individual frames using OpenCV, specifically the functions cv2.VideoCapture and cv2 read(). For each frame, the output will be **final_frames**, which is a list of numpy arrays that represent the image data.

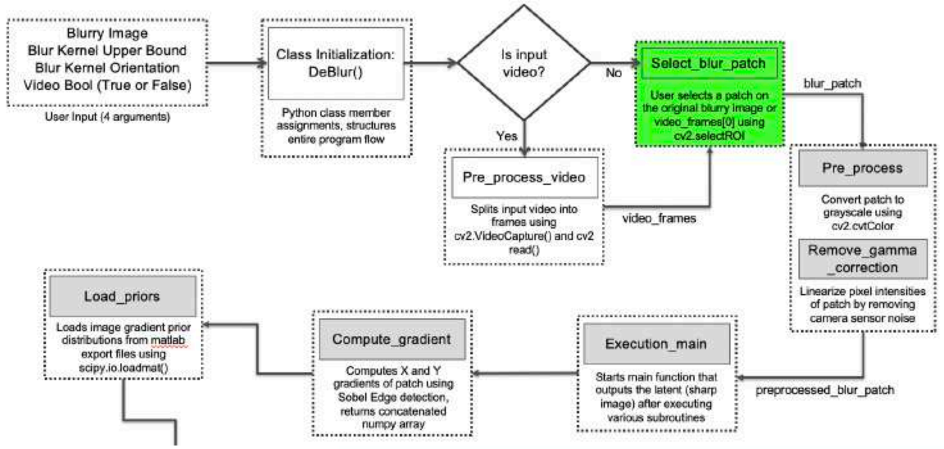
Input Video



Output frames



```python
def pre_process_video(self, video_path):
    video_object = cv2.VideoCapture(video_path)
    img_count, render = 0, 1

    while render:
        render, img = video_object.read()
        cv2.imwrite('video_frames/frame{}.jpg'.format(img_count), img)
        img_count += 1

    frame_nums = []
    dir = '/Users/ms621y/Desktop/GaTech/assignments/Final_Project/video_frames/'
    for frame in glob.glob(dir + '*.jpg'):
        frame_num = frame.split('/')[-1].split('frame')[1].split('.jpg')[0]
        frame_nums.append(int(frame_num))

    frame_nums = sorted(frame_nums)

    final_frames = []
    for index, frame_num in enumerate(frame_nums):
        frame_name = dir + 'frame' + str(frame_num) + '.jpg'
        final_frames.append(frame_name)
    final_frames = final_frames[:-1]
    return final_frames
```

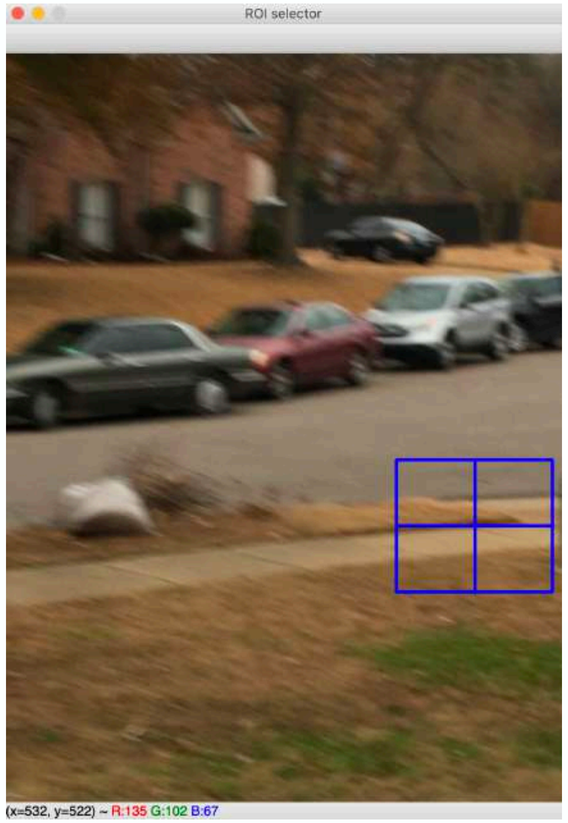

Blurry Image
Blur Kernel Upper Bound
Blur Kernel Orientation
Video Bool (True or False)
User Input (4 arguments)
Class Initialization: DeBlur()
Python class member assignments, structures entire program flow
Is input video?
No
Yes
Select_blur_patch
User selects a patch on the original blurry image or video_frames[0] using cv2.selectROI
blur_patch
Pre_process_video
Splits input video into frames using cv2.VideoCapture() and cv2 read()
video_frames
Pre_process
Convert patch to grayscale using cv2.cvtColor
Remove_gamma_correction
Linearize pixel intensities of patch by removing camera sensor noise
preprocessed_blur_patch
Execution_main
Starts main function that outputs the latent (sharp image) after executing various subroutines
Compute_gradient
Computes X and Y gradients of patch using Sobel Edge detection, returns concatenated numpy array
Load_priors
Loads image gradient prior distributions from matlab export files using scipy.io.loadmat()

🟩 **Pipeline Progress**

Project Development: Blur Patch Selection

Lets continue the case where the input is a single blurry image.

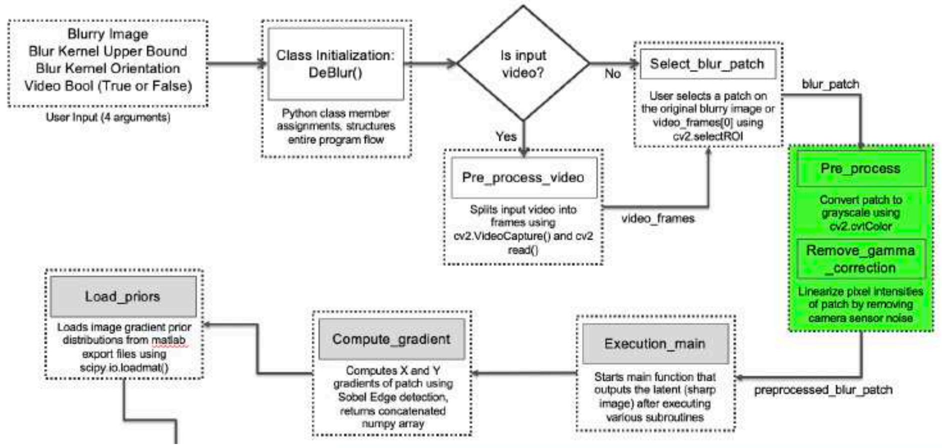
Once read by cv2.imread(), we will then present the user with the ability to select a patch on the blurry image using cv2.selectROI().



```
self.blurry_img = cv2.imread(blurry_img)
user_patch = cv2.selectROI(self.blurry_img)
self.blur_patch = self.blurry_img[int(user_patch[1]):int(user_patch[1] + user_patch[3]), int(user_patch
self.blurry_gray_patch_gamma_corrected = self.pre_process()

out_img = self.execution_main()
cv2.imwrite('out_img.png', out_img)
```

Once selected, the coordinates of the selected region will be used to crop the blurry image and then passed along to the image pre-processing step.

For clarity, the main reason for the blur patch selection is to enhance the performance of the system, especially when performing the variational Bayesian inference. The inference runs extremely slowly when processing on the entire image, hence the need for a smaller set of pixels to work on. When selecting the patch, the user should select pixels with good edge structures and a good mix of mid-range pixel intensities that represent the entire image as a whole. The accuracy of the estimated blur kernel depends **highly** on the quality of the patch selection so it is very important to use good intuition when choosing the patch.

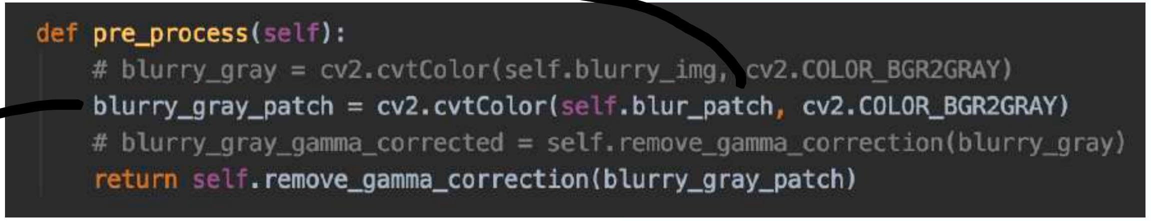

Blurry Image
Blur Kernel Upper Bound
Blur Kernel Orientation
Video Bool (True or False)
User Input (4 arguments)
Class Initialization: DeBlur()
Python class member assignments, structures entire program flow
Is input video?
No
Yes
Select_blur_patch
User selects a patch on the original blurry image or video_frames[0] using cv2.selectROI
blur_patch
Pre_process_video
Splits input video into frames using cv2.VideoCapture() and cv2 read()
video_frames
Pre_process
Convert patch to grayscale using cv2.cvtColor
Remove_gamma_correction
Linearize pixel intensities of patch by removing camera sensor noise
preprocessed_blur_patch
Execution_main
Starts main function that outputs the latent (sharp image) after executing various subroutines
Compute_gradient
Computes X and Y gradients of patch using Sobel Edge detection, returns concatenated numpy array
Load_priors
Loads image gradient prior distributions from matlab export files using scipy.io.loadmat()

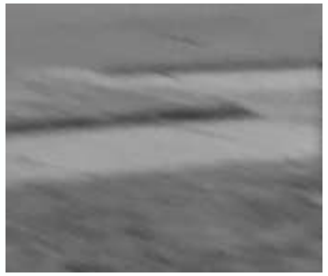
Project Development: Image Pre-Processing

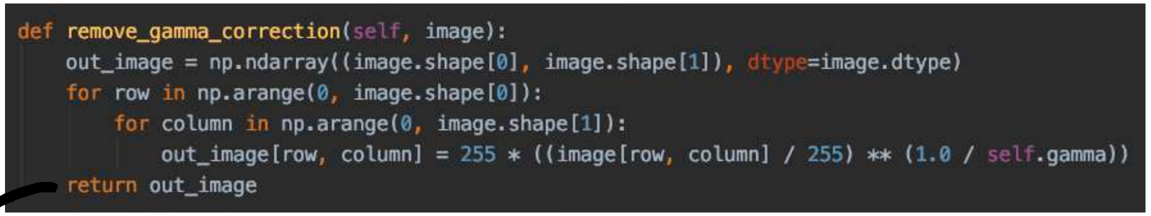
🟩 **Pipeline Progress**

We will now pre-preprocess the blur patch for easier processing for the latter stages of the system. The selected blur patch is grayscaled using cv2.cvtColor and then gamma correction is removed.

Selected blur patch


```python
def pre_process(self):
    # blurry_gray = cv2.cvtColor(self.blurry_img, cv2.COLOR_BGR2GRAY)
    blurry_gray_patch = cv2.cvtColor(self.blur_patch, cv2.COLOR_BGR2GRAY)
    # blurry_gray_gamma_corrected = self.remove_gamma_correction(blurry_gray)
    return self.remove_gamma_correction(blurry_gray_patch)
```


Grayscaled

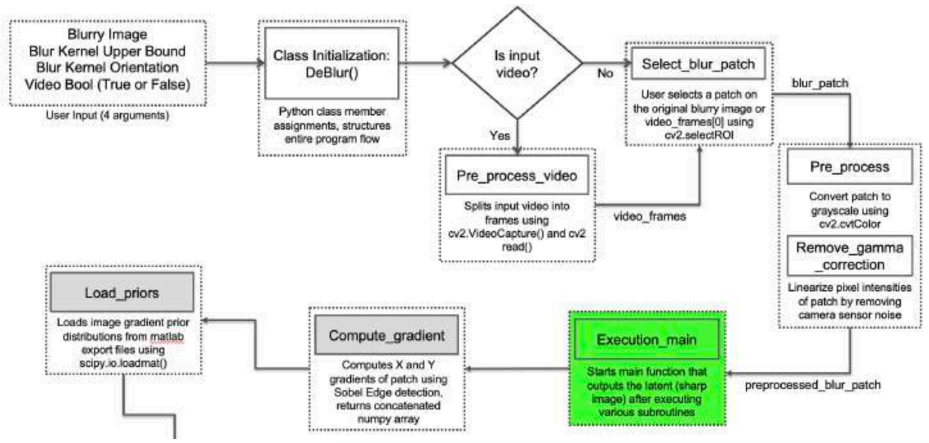

```python
def remove_gamma_correction(self, image):
    out_image = np.ndarray((image.shape[0], image.shape[1]), dtype=image.dtype)
    for row in np.arange(0, image.shape[0]):
        for column in np.arange(0, image.shape[1]):
            out_image[row, column] = 255 * ((image[row, column] / 255) ** (1.0 / self.gamma))
    return out_image
```

Removed gamma correction

$$\text{Pixel value} = (\text{CCD sensor value})^{1/\gamma}$$

This equation represents encoding using gamma correction. For removal, the inverse is calculated in order to linearize the pixel intensities by removing the camera sensor noise.

Project Development: Execution_Main()

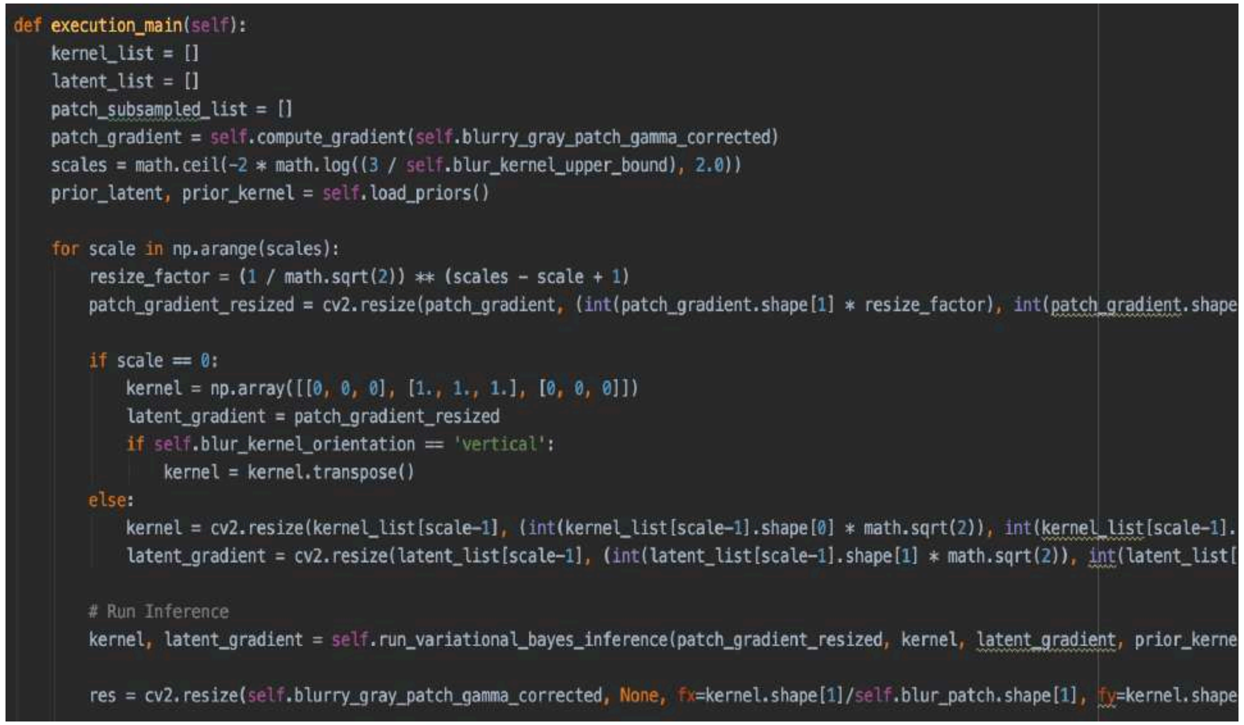


Blurry Image
Blur Kernel Upper Bound
Blur Kernel Orientation
Video Bool (True or False)
User Input (4 arguments)
Class Initialization: DeBlur()
Python class member assignments, structures entire program flow
Is input video?
No
Yes
Select_blur_patch
User selects a patch on the original blurry image or video_frames[0] using cv2.selectROI
blur_patch
Pre_process_video
Splits input video into frames using cv2.VideoCapture() and cv2 read()
video_frames
Pre_process
Convert patch to grayscale using cv2.cvtColor
Remove_gamma_correction
Linearize pixel intensities of patch by removing camera sensor noise
preprocessed_blur_patch
Execution_main
Starts main function that outputs the latent (sharp image) after executing various subroutines
Compute_gradient
Computes X and Y gradients of patch using Sobel Edge detection, returns concatenated numpy array
Load_priors
Loads image gradient prior distributions from matlab export files using scipy.io.loadmat()

🟩 **Pipeline Progress**

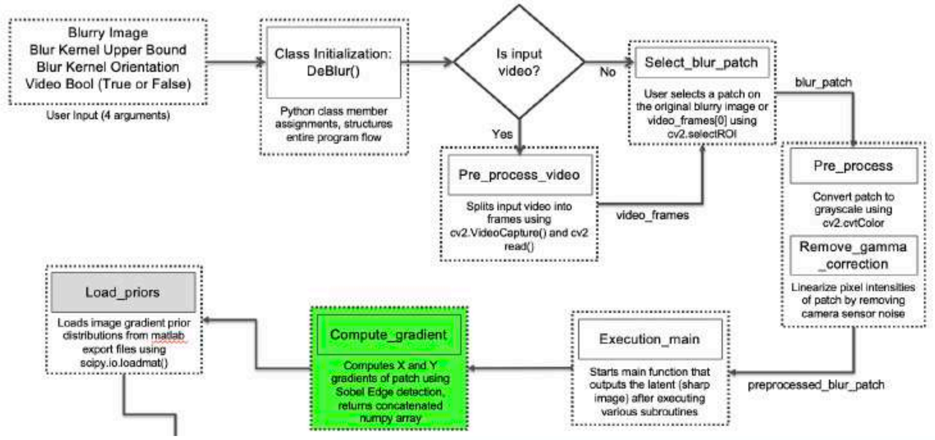
Now we enter the main function that will run the multi-scale iterative approach to determine the best kernel for the output latent (sharp) image. Once the gradients are found and the number of scales are determined, we will initialize the kernel, latent_gradient, and patch_gradient to be then run through the variational Bayesian inference. Once the new kernel and latent gradients are found, we will run through the inference again using these new values until we meet the upper bound of the scales. Afterwards, we will run through a deconvolution algorithm called Richardson-Lucy and then output the final results.

Lets dive deeper into the details of this function.



```
def execution_main(self):
    kernel_list = []
    latent_list = []
    patch_subsampled_list = []
    patch_gradient = self.compute_gradient(self.blurry_gray_patch_gamma_corrected)
    scales = math.ceil(-2 * math.log((3 / self.blur_kernel_upper_bound), 2.0))
    prior_latent, prior_kernel = self.load_priors()

    for scale in np.arange(scales):
        resize_factor = (1 / math.sqrt(2)) ** (scales - scale + 1)
        patch_gradient_resized = cv2.resize(patch_gradient, (int(patch_gradient.shape[1] * resize_factor), int(patch_gradient.shape

        if scale == 0:
            kernel = np.array([[0, 0, 0], [1., 1., 1.], [0, 0, 0]])
            latent_gradient = patch_gradient_resized
            if self.blur_kernel_orientation == 'vertical':
                kernel = kernel.transpose()
        else:
            kernel = cv2.resize(kernel_list[scale-1], (int(kernel_list[scale-1].shape[0] * math.sqrt(2)), int(kernel_list[scale-1].
            latent_gradient = cv2.resize(latent_list[scale-1], (int(latent_list[scale-1].shape[1] * math.sqrt(2)), int(latent_list[

        # Run Inference
        kernel, latent_gradient = self.run_variational_bayes_inference(patch_gradient_resized, kernel, latent_gradient, prior_kerne

        res = cv2.resize(self.blurry_gray_patch_gamma_corrected, None, fx=kernel.shape[1]/self.blur_patch.shape[1], fy=kernel.shape
```

Project Development: Compute Gradients

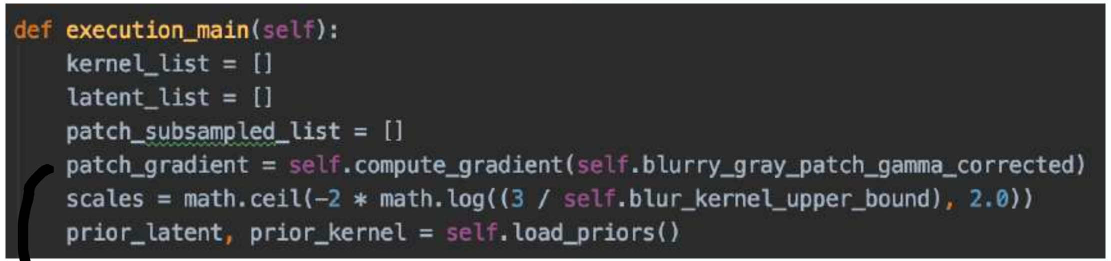


Blurry Image
Blur Kernel Upper Bound
Blur Kernel Orientation
Video Bool (True or False)
User Input (4 arguments)
Class Initialization: DeBlur()
Python class member assignments, structures entire program flow
Is input video?
No
Yes
Select_blur_patch
User selects a patch on the original blurry image or video_frames[0] using cv2.selectROI
blur_patch
Pre_process_video
Splits input video into frames using cv2.VideoCapture() and cv2 read()
video_frames
Pre_process
Convert patch to grayscale using cv2.cvtColor
Remove_gamma_correction
Linearize pixel intensities of patch by removing camera sensor noise
preprocessed_blur_patch
Execution_main
Starts main function that outputs the latent (sharp image) after executing various subroutines
Compute_gradient
Computes X and Y gradients of patch using Sobel Edge detection, returns concatenated numpy array
Load_priors
Loads image gradient prior distributions from matlab export files using scipy.io.loadmat()

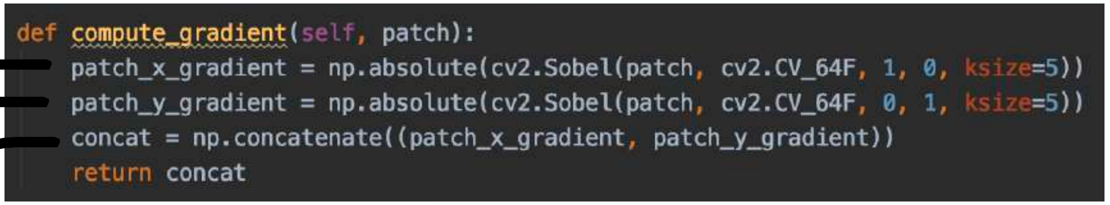
🟩 **Pipeline Progress**

In order to perform the Bayesian inference, we must first calculate the gradients of the blur patch. Because the convolution operator is a linear operation, we can perform the optimization of the inference in the gradient domain, hence the need for computation.

In the execution_main() function, we execute compute_gradient(), which utilizes the cv2.Sobel method for Sobel Edge Detection. This is a proven method of gradient calculation as it performs edge detection in both X and Y direction as needed. Finally, the output is a concatenated numpy array of both gradients so that input can be fed easier into the inference algorithm.

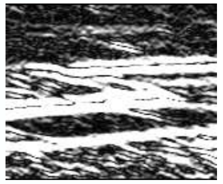


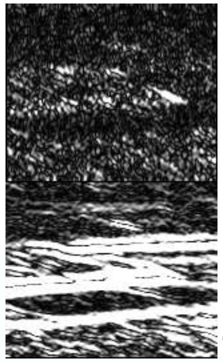
```
def execution_main(self):
    kernel_list = []
    latent_list = []
    patch_subsampled_list = []
    patch_gradient = self.compute_gradient(self.blurry_gray_patch_gamma_corrected)
    scales = math.ceil(-2 * math.log((3 / self.blur_kernel_upper_bound), 2.0))
    prior_latent, prior_kernel = self.load_priors()
```



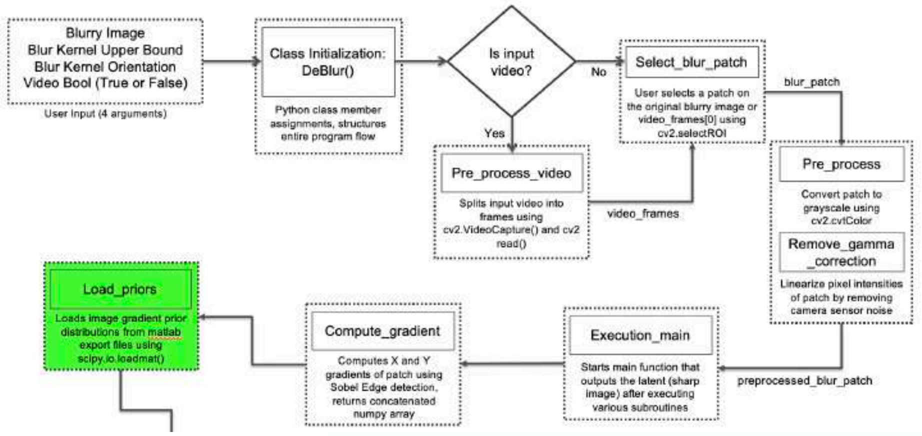
```
def compute_gradient(self, patch):
    patch_x_gradient = np.absolute(cv2.Sobel(patch, cv2.CV_64F, 1, 0, ksize=5))
    patch_y_gradient = np.absolute(cv2.Sobel(patch, cv2.CV_64F, 0, 1, ksize=5))
    concat = np.concatenate((patch_x_gradient, patch_y_gradient))
    return concat
```

Gradient_X


Gradient_Y


Concatenation

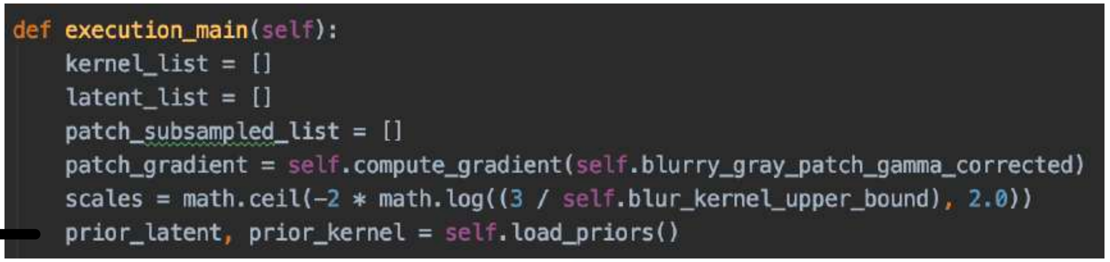
Project Development: Load Priors


Blurry Image
Blur Kernel Upper Bound
Blur Kernel Orientation
Video Bool (True or False)
User Input (4 arguments)
Class Initialization: DeBlur()
Python class member assignments, structures entire program flow
Is input video?
No
Yes
Select_blur_patch
User selects a patch on the original blurry image or video_frames[0] using cv2.selectROI
blur_patch
Pre_process_video
Splits input video into frames using cv2.VideoCapture() and cv2 read()
video_frames
Pre_process
Convert patch to grayscale using cv2.cvtColor
Remove_gamma_correction
Linearize pixel intensities of patch by removing camera sensor noise
preprocessed_blur_patch
Execution_main
Starts main function that outputs the latent (sharp image) after executing various subroutines
Compute_gradient
Computes X and Y gradients of patch using Sobel Edge detection, returns concatenated numpy array
Load_priors
Loads image gradient prior distributions from matlab export files using scipy.io.loadmat()

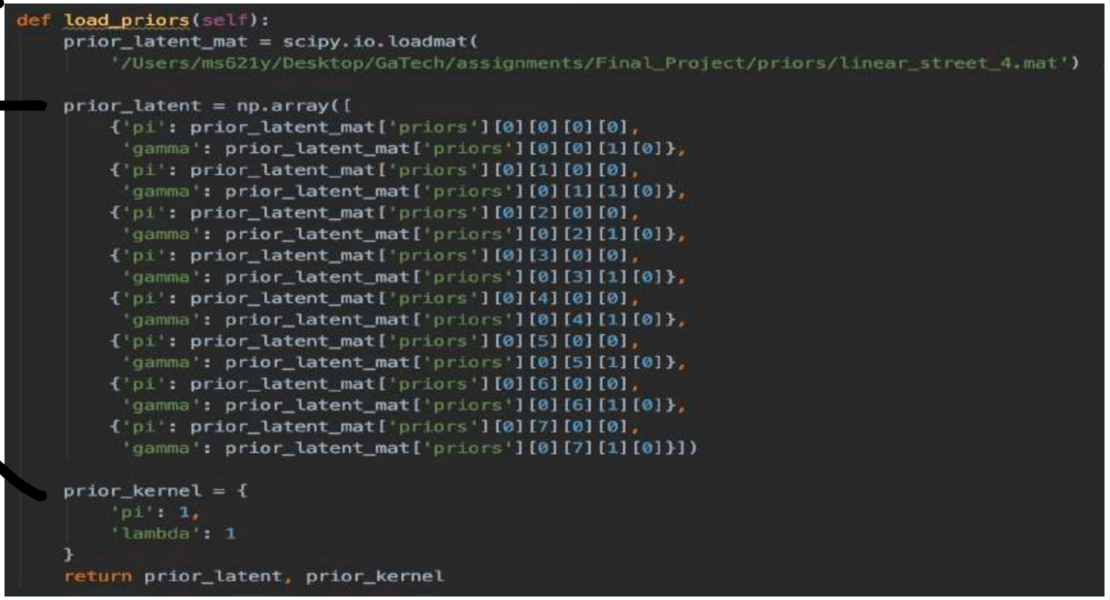
🟩 **Pipeline Progress**

Next, we load the priors of the Bayesian probability theorem from the authors of the paper. Below is the image formation model we are trying to solve using Bayesian Inferencing. The priors for the latent patch, given by Lp, contains a prior distribution on image gradients that the authors in the paper used for all of their experiments. In order to replicate my results as closely as possible, I aimed to use their priors as well. The .mat file was given by the authors on their project website: https://cs.nyu.edu/~fergus/research/deblur.html.

Using scipy.io.loadmat(), the .mat files of the priors were loaded and then stored into numpy data structures separated by **pi** and **gamma** using a dictionary. No priors were used for the kernel.



```
def execution_main(self):
    kernel_list = []
    latent_list = []
    patch_subsampled_list = []
    patch_gradient = self.compute_gradient(self.blurry_gray_patch_gamma_corrected)
    scales = math.ceil(-2 * math.log((3 / self.blur_kernel_upper_bound), 2.0))
    prior_latent, prior_kernel = self.load_priors()
```



```
def load_priors(self):
    prior_latent_mat = scipy.io.loadmat(
        '/Users/ms621y/Desktop/GaTech/assignments/Final_Project/priors/linear_street_4.mat')

    prior_latent = np.array([
        {'pi': prior_latent_mat['priors'][0][0][0][0],
         'gamma': prior_latent_mat['priors'][0][0][1][0]},
        {'pi': prior_latent_mat['priors'][0][1][0][0],
         'gamma': prior_latent_mat['priors'][0][1][1][0]},
        {'pi': prior_latent_mat['priors'][0][2][0][0],
         'gamma': prior_latent_mat['priors'][0][2][1][0]},
        {'pi': prior_latent_mat['priors'][0][3][0][0],
         'gamma': prior_latent_mat['priors'][0][3][1][0]},
        {'pi': prior_latent_mat['priors'][0][4][0][0],
         'gamma': prior_latent_mat['priors'][0][4][1][0]},
        {'pi': prior_latent_mat['priors'][0][5][0][0],
         'gamma': prior_latent_mat['priors'][0][5][1][0]},
        {'pi': prior_latent_mat['priors'][0][6][0][0],
         'gamma': prior_latent_mat['priors'][0][6][1][0]},
        {'pi': prior_latent_mat['priors'][0][7][0][0],
         'gamma': prior_latent_mat['priors'][0][7][1][0]}])

    prior_kernel = {
        'pi': 1,
        'lambda': 1
    }
    return prior_latent, prior_kernel
```

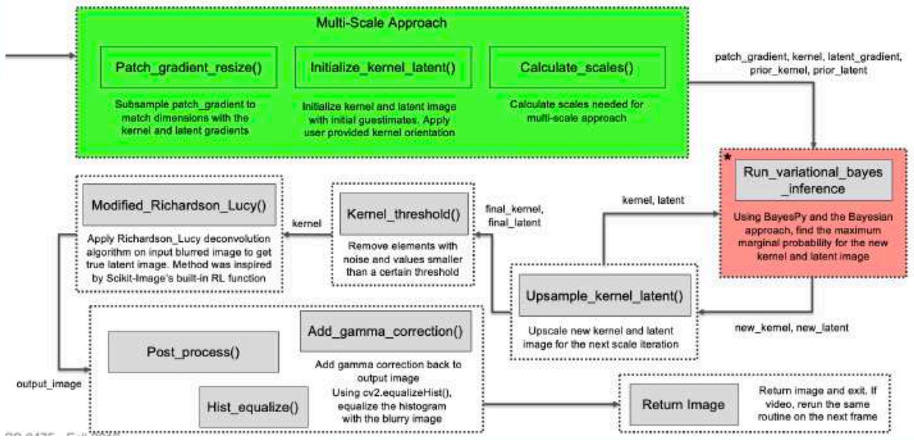
$$p(\mathbf{K}, \nabla\mathbf{L}_p | \nabla\mathbf{P}) \propto p(\nabla\mathbf{P} | \mathbf{K}, \nabla\mathbf{L}_p) p(\nabla\mathbf{L}_p) p(\mathbf{K})$$

Image Formation Model

Prior distribution for latent image

Prior distribution for kernel

Project Development: Multi-Scale Approach


Multi-Scale Approach
Patch_gradient_resize()
Subsample patch_gradient to match dimensions with the kernel and latent gradients
Initialize_kernel_latent()
Initialize kernel and latent image with initial guestimates. Apply user provided kernel orientation
Calculate_scales()
Calculate scales needed for multi-scale approach
patch_gradient, kernel, latent_gradient, prior_kernel, prior_latent
Run_variational_bayes_inference
Using BayesPy and the Bayesian approach, find the maximum marginal probability for the new kernel and latent image
kernel, latent
new_kernel, new_latent
Upsample_kernel_latent()
Upscale new kernel and latent image for the next scale iteration
final_kernel, final_latent
Kernel_threshold()
Remove elements with noise and values smaller than a certain threshold
kernel
Modified_Richardson_Lucy()
Apply Richardson_Lucy deconvolution algorithm on input blurred image to get true latent image. Method was inspired by Scikit-Image's built-in RL function
output_image
Post_process()
Add_gamma_correction()
Add gamma correction back to output image
Hist_equalize()
Using cv2.equalizeHist(), equalize the histogram with the blurry image
Return Image
Return image and exit. If video, rerun the same routine on the next frame

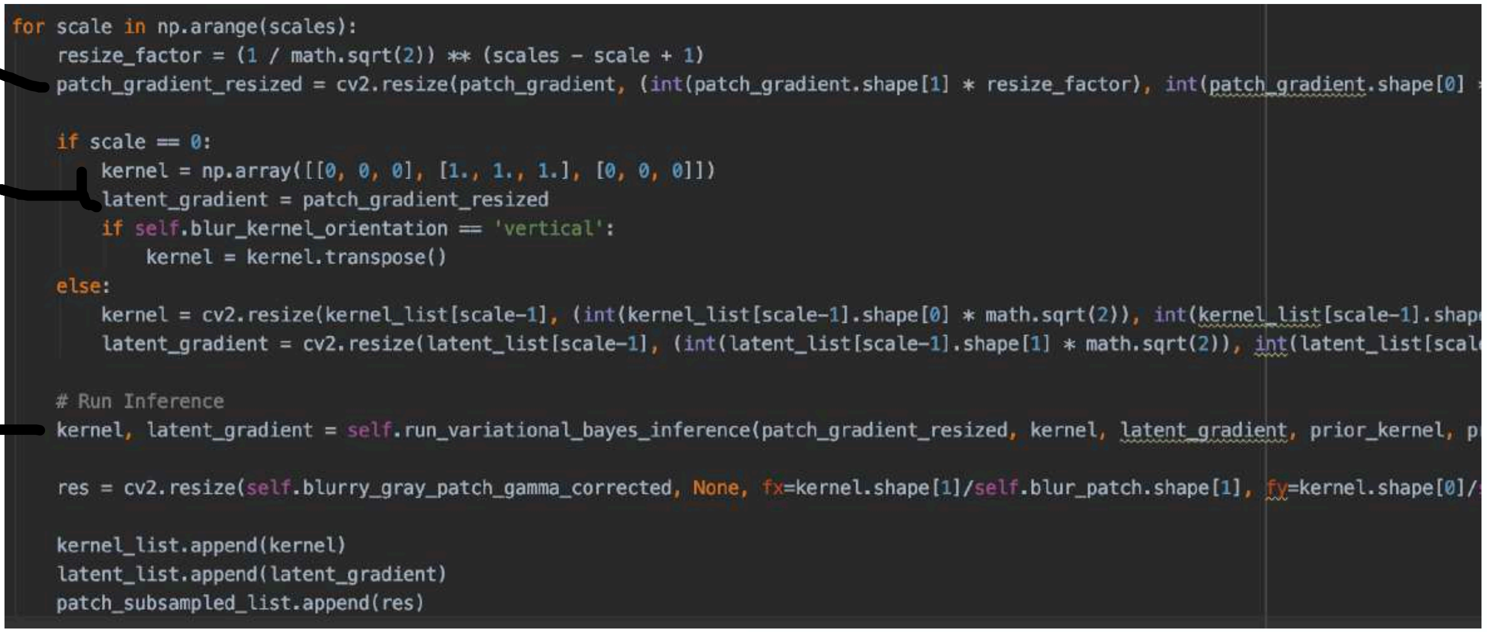
🟩 **Pipeline Progress**

Now, we take a Multi-Scale Approach to determine the kernel for the output image.

First, we subsample the patch_gradients to match the dimensions of the current kernel using cv2.resize() and bilinear interpolation.

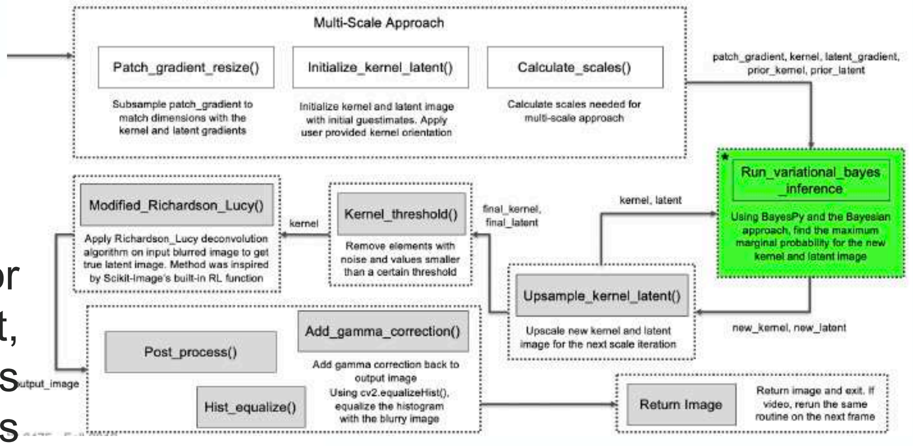
Next, we initialize the kernel to a simple horizontal bar kernel. If the user provided an initial guestimate of a vertical kernel in the input parameters, we do a simple transpose.

Afterwards, using the five paramters: patch_gradient, kernel, latent_gradient, prior_kernel and prior_latent, we execute the function **run_variational_bayes_inference()**.



```
for scale in np.arange(scales):
    resize_factor = (1 / math.sqrt(2)) ** (scales - scale + 1)
    patch_gradient_resized = cv2.resize(patch_gradient, (int(patch_gradient.shape[1] * resize_factor), int(patch_gradient.shape[0]

    if scale == 0:
        kernel = np.array([[0, 0, 0], [1., 1., 1.], [0, 0, 0]])
        latent_gradient = patch_gradient_resized
        if self.blur_kernel_orientation == 'vertical':
            kernel = kernel.transpose()
    else:
        kernel = cv2.resize(kernel_list[scale-1], (int(kernel_list[scale-1].shape[0] * math.sqrt(2)), int(kernel_list[scale-1].shap
        latent_gradient = cv2.resize(latent_list[scale-1], (int(latent_list[scale-1].shape[1] * math.sqrt(2)), int(latent_list[scal

    # Run Inference
    kernel, latent_gradient = self.run_variational_bayes_inference(patch_gradient_resized, kernel, latent_gradient, prior_kernel, p

    res = cv2.resize(self.blurry_gray_patch_gamma_corrected, None, fx=kernel.shape[1]/self.blur_patch.shape[1], fy=kernel.shape[0]/

    kernel_list.append(kernel)
    latent_list.append(latent_gradient)
    patch_subsampled_list.append(res)
```

The purpose of the multi-scale approach is to find the highest resolution kernel possible using an iterative approach to the inference.


Multi-Scale Approach
Patch_gradient_resize()
Subsample patch_gradient to match dimensions with the kernel and latent gradients
Initialize_kernel_latent()
Initialize kernel and latent image with initial guestimates. Apply user provided kernel orientation
Calculate_scales()
Calculate scales needed for multi-scale approach
patch_gradient, kernel, latent_gradient, prior_kernel, prior_latent
Run_variational_bayes_inference
Using BayesPy and the Bayesian approach, find the maximum marginal probability for the new kernel and latent image
kernel, latent
new_kernel, new_latent
Upsample_kernel_latent()
Upscale new kernel and latent image for the next scale iteration
final_kernel, final_latent
Kernel_threshold()
Remove elements with noise and values smaller than a certain threshold
kernel
Modified_Richardson_Lucy()
Apply Richardson_Lucy deconvolution algorithm on input blurred image to get true latent image. Method was inspired by Scikit-Image's built-in RL function
output_image
Post_process()
Add_gamma_correction()
Add gamma correction back to output image
Hist_equalize()
Using cv2.equalizeHist(), equalize the histogram with the blurry image
Return Image
Return image and exit. If video, rerun the same routine on the next frame

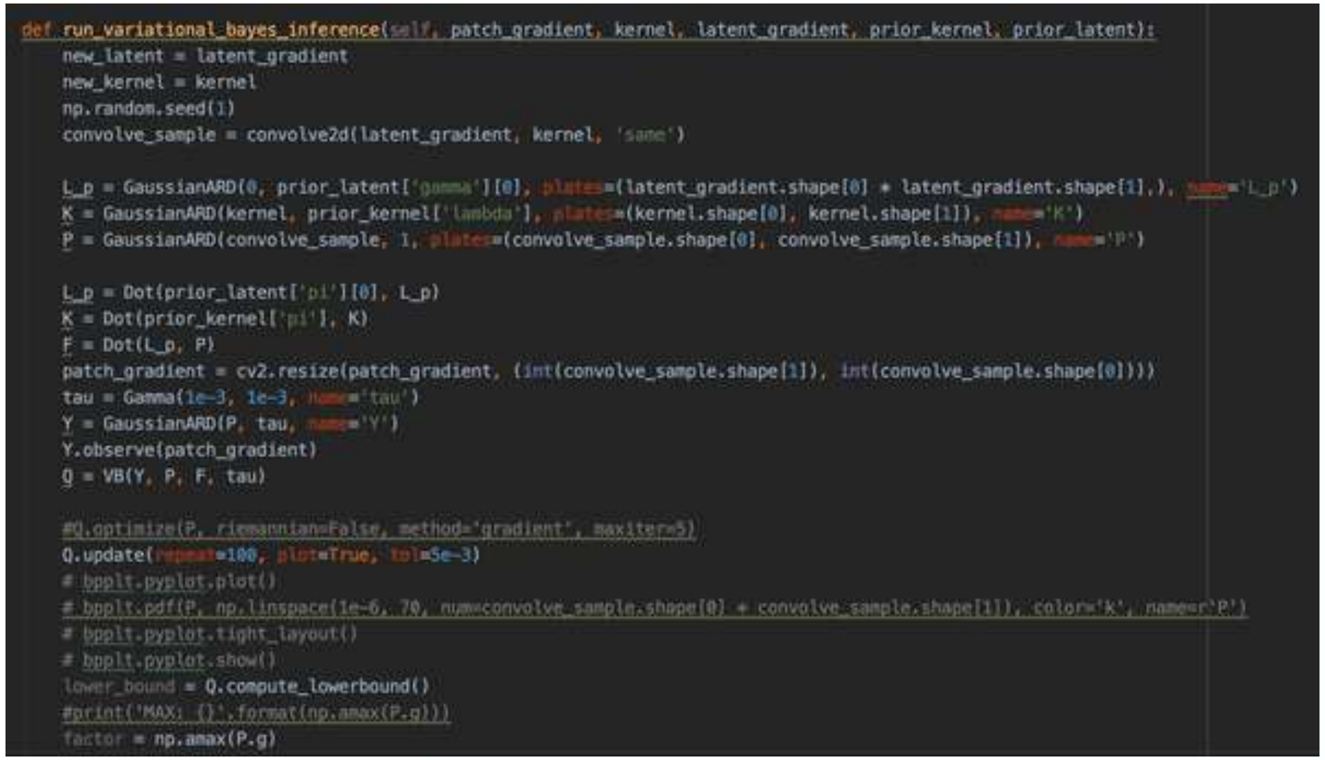
🟩 **Pipeline Progress**

Project Development: Bayesian Inference

Bayes Theorem is a powerful tool in probabilistic theory that allows one to solve for random variables while monitoring for convergence. In this implementation attempt, I attempted to implement a package called **BayesPy**, that uses a Variational Bayes engine for convergence calculation. Other packages I've looked into such as PyMC3, instead, uses Markov Chain Monte Carlo which is something the authors in this paper mentioned was not effective when solving for the blur kernel.

For the image formation model, three things needed to be constructed in order to estimate the posterior distribution: the prior model for the latent_gradient, the prior model for the kernel, and the likelihood model using the convolution pixels of the current kernel and latent gradients.

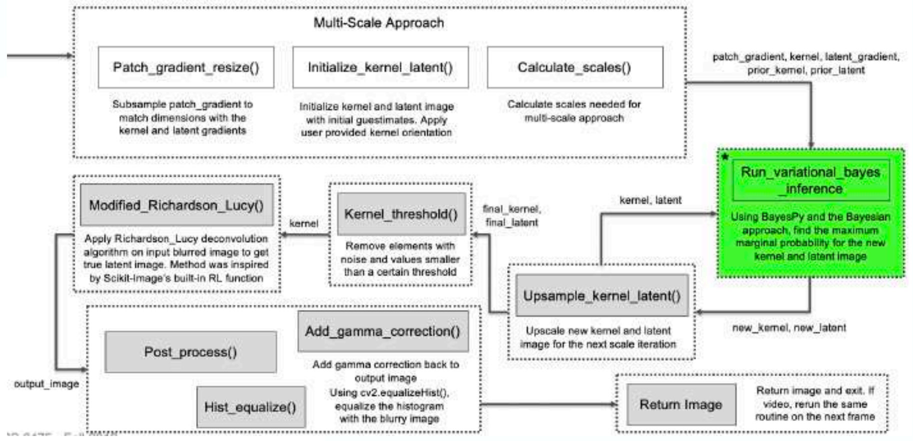
As can be seen, many methods were explored in constructing the node network in BayesPy such as implementing Gaussian, Gamma, and Exponential models. Once convergence is found after a certain threshold, which the authors found to be 5e-3, we can then compute the maximum marginal probability of the outputs to find the new kernel and latent gradients. Using this information, we run the inference again using a scaled up version of the output until all of the scales are completed.



```
def run_variational_bayes_inference(self, patch_gradient, kernel, latent_gradient, prior_kernel, prior_latent):
    new_latent = latent_gradient
    new_kernel = kernel
    np.random.seed(1)
    convolve_sample = convolve2d(latent_gradient, kernel, 'same')

    L_p = GaussianARD(0, prior_latent['gamma'][0], plates=(latent_gradient.shape[0] * latent_gradient.shape[1],), name='L_p')
    K = GaussianARD(kernel, prior_kernel['lambda'], plates=(kernel.shape[0], kernel.shape[1]), name='K')
    P = GaussianARD(convolve_sample, 1, plates=(convolve_sample.shape[0], convolve_sample.shape[1]), name='P')

    L_p = Dot(prior_latent['pi'][0], L_p)
    K = Dot(prior_kernel['pi'], K)
    F = Dot(L_p, P)
    patch_gradient = cv2.resize(patch_gradient, (int(convolve_sample.shape[1]), int(convolve_sample.shape[0])))
    tau = Gamma(1e-3, 1e-3, name='tau')
    Y = GaussianARD(P, tau, name='Y')
    Y.observe(patch_gradient)
    Q = VB(Y, P, F, tau)

    #Q.optimize(P, riemannian=False, method='gradient', maxiter=5)
    Q.update(repeat=100, plot=True, tol=5e-3)
    # bpplt.pyplot.plot()
    # bpplt.pdf(P, np.linspace(1e-6, 70, num=convolve_sample.shape[0] * convolve_sample.shape[1]), color='k', name=r'P')
    # bpplt.pyplot.tight_layout()
    # bpplt.pyplot.show()
    lower_bound = Q.compute_lowerbound()
    #print('MAX: {}'.format(np.amax(P.g)))
    factor = np.amax(P.g)
```

Failed Attempt at implementing Variational Bayesian Inference using BayesPy

Project Development: Bayesian Inference

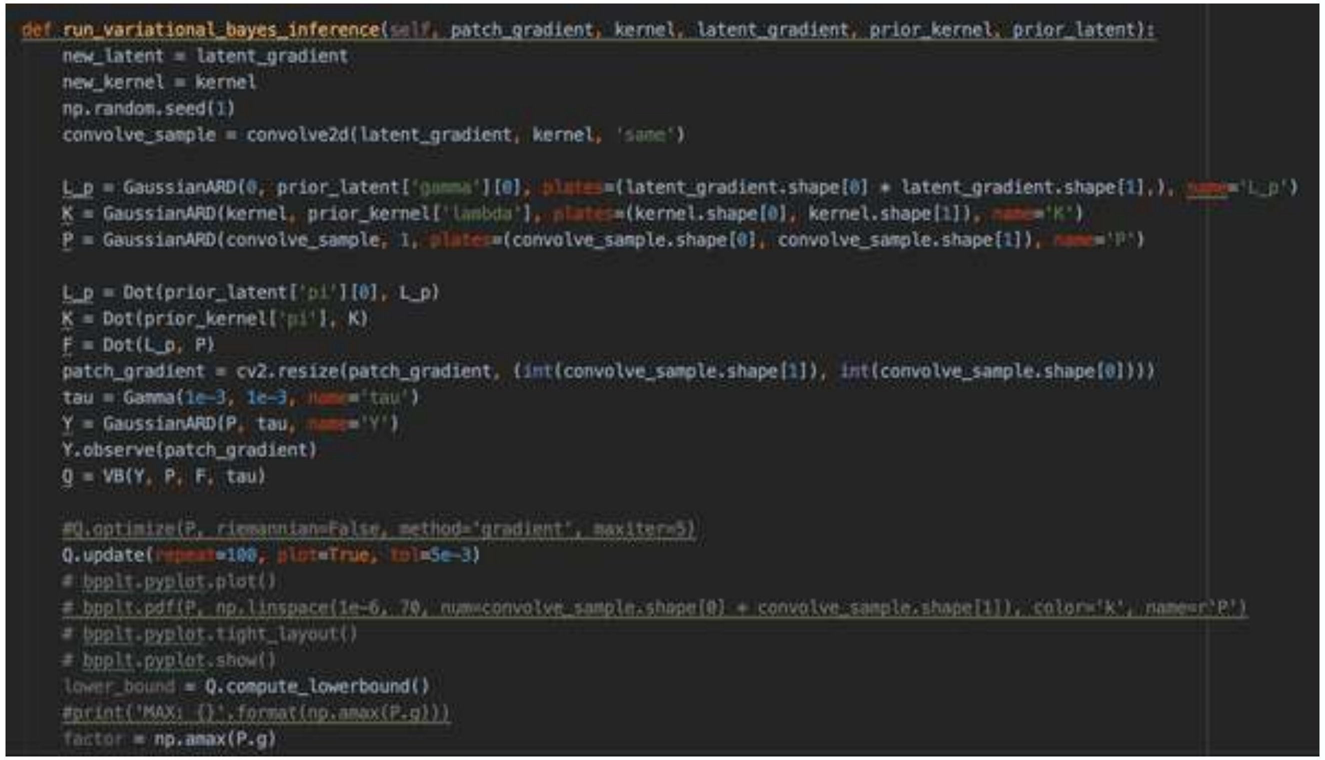


Multi-Scale Approach
Patch_gradient_resize()
Subsample patch_gradient to match dimensions with the kernel and latent gradients
Initialize_kernel_latent()
Initialize kernel and latent image with initial guestimates. Apply user provided kernel orientation
Calculate_scales()
Calculate scales needed for multi-scale approach
patch_gradient, kernel, latent_gradient, prior_kernel, prior_latent
Run_variational_bayes _inference
Using BayesPy and the Bayesian approach, find the maximum marginal probability for the new kernel and latent image
kernel, latent
new_kernel, new_latent
Upsample_kernel_latent()
Upscale new kernel and latent image for the next scale iteration
final_kernel, final_latent
Kernel_threshold()
Remove elements with noise and values smaller than a certain threshold
kernel
Modified_Richardson_Lucy()
Apply Richardson_Lucy deconvolution algorithm on input blurred image to get true latent image. Method was inspired by Scikit-Image's built-in RL function
output_image
Post_process()
Add_gamma_correction()
Add gamma correction back to output image
Hist_equalize()
Using cv2.equalizeHist(), equalize the histogram with the blurry image
Return Image
Return image and exit. If video, rerun the same routine on the next frame

🟩 **Pipeline Progress**

Reflecting on this part of the project, I spent a lot of time trying to implement Bayesian inference in determining the output kernel. What attracted me to this project a lot was the use case of data science into applications such as photography. However, because I am still very new to data science and machine learning, the probability theory was something I had to review from scratch, starting with what Bayes Theorem actually is! After reading the research paper I chose, I did not understand any of the formulas presented or any of the probability vocabulary used, such as posterior, prior, and convergence. Overall, it was a huge learning curve, no pun intended!

Some of the results in the output included completely black images with pixel values equal to zero. Others included distorted outputs which resulted in *very cool looking* images. These images will be showcased in the following section. In order to alleviate this failed function, I attempted to calculate output kernels using other methods such as trial and error and multiplying by a scalar number found during the inference. This resulted in the images shown previously.



```
def run_variational_bayes_inference(self, patch_gradient, kernel, latent_gradient, prior_kernel, prior_latent):
    new_latent = latent_gradient
    new_kernel = kernel
    np.random.seed(1)
    convolve_sample = convolve2d(latent_gradient, kernel, 'same')

    L_p = GaussianARD(0, prior_latent['gamma'][0], plates=(latent_gradient.shape[0] * latent_gradient.shape[1],), name='L_p')
    K = GaussianARD(kernel, prior_kernel['lambda'], plates=(kernel.shape[0], kernel.shape[1]), name='K')
    P = GaussianARD(convolve_sample, 1, plates=(convolve_sample.shape[0], convolve_sample.shape[1]), name='P')

    L_p = Dot(prior_latent['pi'][0], L_p)
    K = Dot(prior_kernel['pi'], K)
    F = Dot(L_p, P)
    patch_gradient = cv2.resize(patch_gradient, (int(convolve_sample.shape[1]), int(convolve_sample.shape[0])))
    tau = Gamma(1e-3, 1e-3, name='tau')
    Y = GaussianARD(P, tau, name='Y')
    Y.observe(patch_gradient)
    Q = VB(Y, P, F, tau)

    #Q.optimize(P, riemannian=False, method='gradient', maxiter=5)
    Q.update(repeat=100, plot=True, tol=5e-3)
    # bpplt.pyplot.plot()
    # bpplt.pdf(P, np.linspace(1e-6, 70, num=convolve_sample.shape[0] * convolve_sample.shape[1]), color='k', name=r'P')
    # bpplt.pyplot.tight_layout()
    # bpplt.pyplot.show()
    lower_bound = Q.compute_lowerbound()
    #print('MAX: {}'.format(np.amax(P.g)))
    factor = np.amax(P.g)
```

Failed Attempt at implementing Variational Bayesian Inference using BayesPy

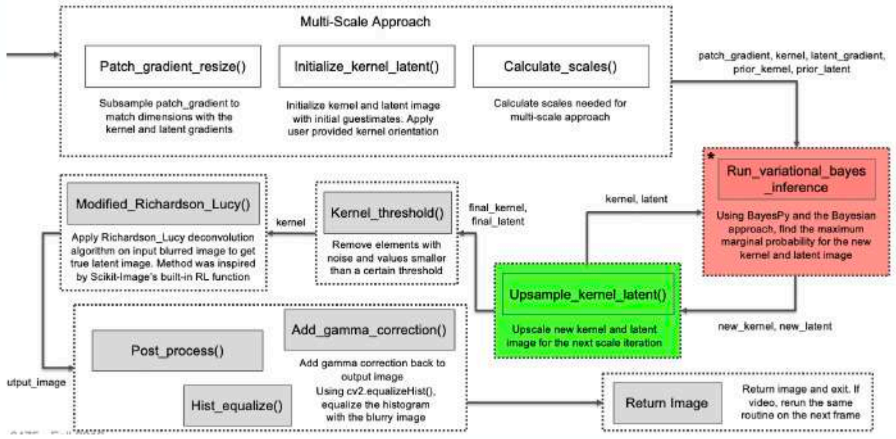
Project Development: Upsample Results

After the Bayesian Inference is completed, the outputs result in a new kernel and new latent image estimate.

For the purposes of the multi-scale approach, these results are then upscaled and then fed back into the Bayesian inference so that it can infer on a higher resolution set of data.

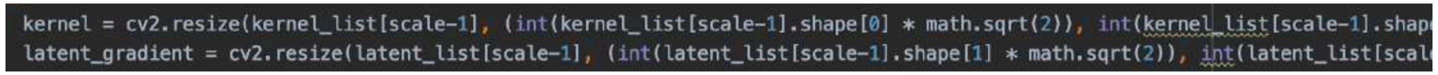
Following this method creates a pyramid for each step along the number of scales until the kernel and latent image resolution meets or crosses the upper bound specified by the user.

In the code, we run a simple iterative loop and store each layer of the pyramid in a list for processing in the next stage of the system.

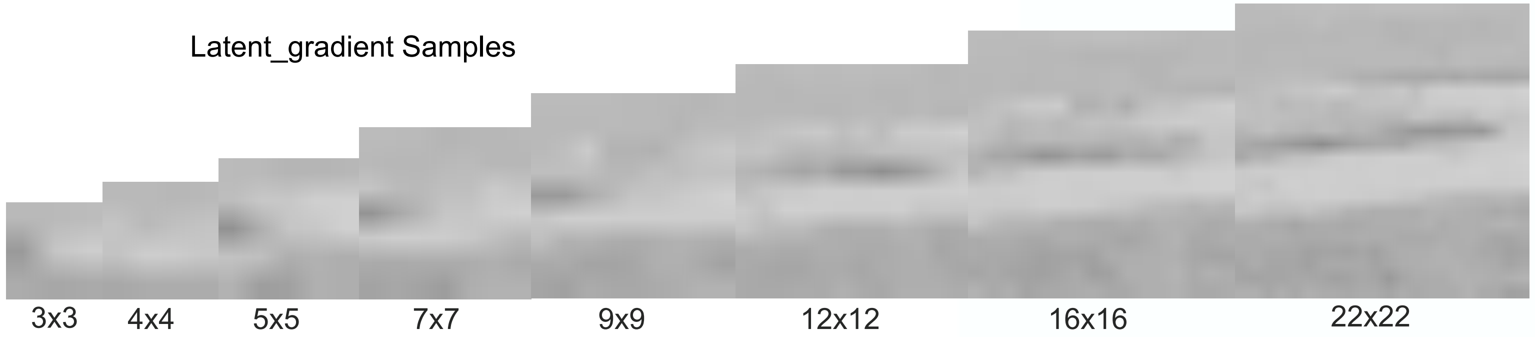


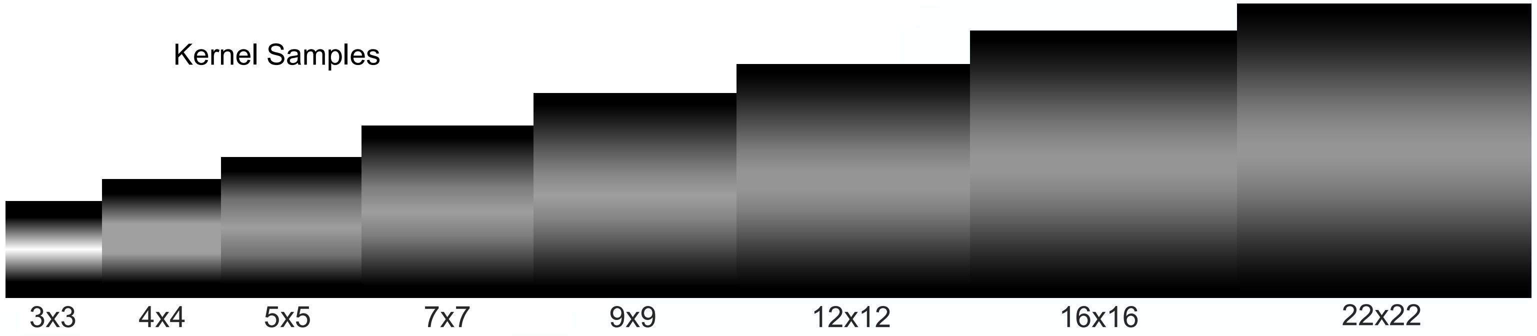
Multi-Scale Approach
Patch_gradient_resize()
Subsample patch_gradient to match dimensions with the kernel and latent gradients
Initialize_kernel_latent()
Initialize kernel and latent image with initial guestimates. Apply user provided kernel orientation
Calculate_scales()
Calculate scales needed for multi-scale approach
patch_gradient, kernel, latent_gradient, prior_kernel, prior_latent
Run_variational_bayes_inference
Using BayesPy and the Bayesian approach, find the maximum marginal probability for the new kernel and latent image
kernel, latent
new_kernel, new_latent
Upsample_kernel_latent()
Upscale new kernel and latent image for the next scale iteration
final_kernel, final_latent
Kernel_threshold()
Remove elements with noise and values smaller than a certain threshold
kernel
Modified_Richardson_Lucy()
Apply Richardson_Lucy deconvolution algorithm on input blurred image to get true latent image. Method was inspired by Scikit-Image's built-in RL function
utput_image
Post_process()
Add_gamma_correction()
Add gamma correction back to output image
Hist_equalize()
Using cv2.equalizeHist(), equalize the histogram with the blurry image
Return Image
Return image and exit. If video, rerun the same routine on the next frame

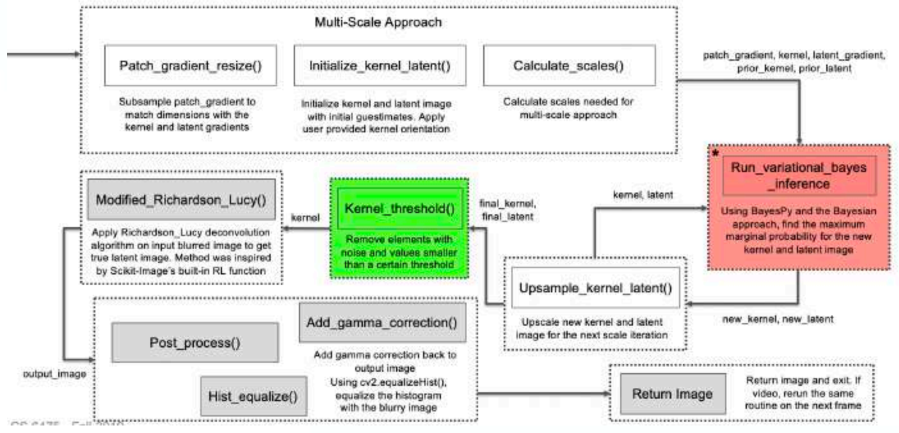
🟩 **Pipeline Progress**



```
kernel = cv2.resize(kernel_list[scale-1], (int(kernel_list[scale-1].shape[0] * math.sqrt(2)), int(kernel_list[scale-1].shap
latent_gradient = cv2.resize(latent_list[scale-1], (int(latent_list[scale-1].shape[1] * math.sqrt(2)), int(latent_list[scal
```






Multi-Scale Approach
Patch_gradient_resize()
Subsample patch_gradient to match dimensions with the kernel and latent gradients
Initialize_kernel_latent()
Initialize kernel and latent image with initial guestimates. Apply user provided kernel orientation
Calculate_scales()
Calculate scales needed for multi-scale approach
patch_gradient, kernel, latent_gradient, prior_kernel, prior_latent
Run_variational_bayes_inference
Using BayesPy and the Bayesian approach, find the maximum marginal probability for the new kernel and latent image
kernel, latent
new_kernel, new_latent
Upsample_kernel_latent()
Upscale new kernel and latent image for the next scale iteration
final_kernel, final_latent
Kernel_threshold()
Remove elements with noise and values smaller than a certain threshold
kernel
Modified_Richardson_Lucy()
Apply Richardson_Lucy deconvolution algorithm on input blurred image to get true latent image. Method was inspired by Scikit-Image's built-in RL function
output_image
Post_process()
Add_gamma_correction()
Add gamma correction back to output image
Hist_equalize()
Using cv2.equalizeHist(), equalize the histogram with the blurry image
Return Image
Return image and exit. If video, rerun the same routine on the next frame

Project Development: Kernel Threshold

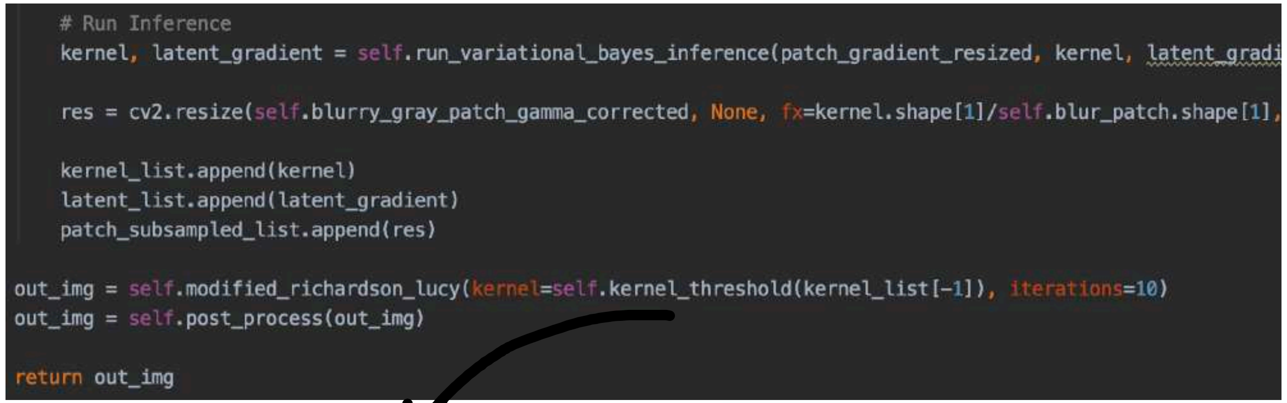
🟩 **Pipeline Progress**

After the multi-scaling, now we have a finalized kernel estimation.

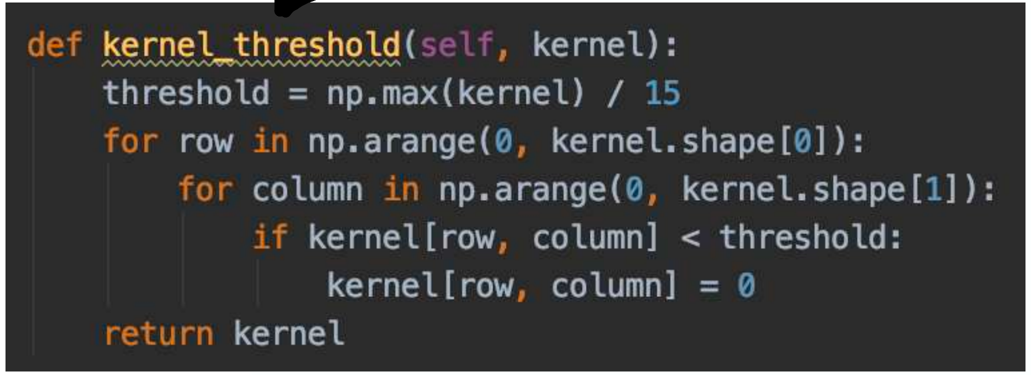
We run the kernel_threshold() function that passes along the most recent kernel prediction.

The function implements a denoising method that removes lower energy pixel values that may have been added due to noise and other factors. We take the largest magnitude pixel value in the array using numpy.max() and divide it by 15. The authors used the number 15 in their experiments and in my attempts to replicate as closely as possible, I used the same number as well. Changing this magic number may improve some of my own results with my own images.

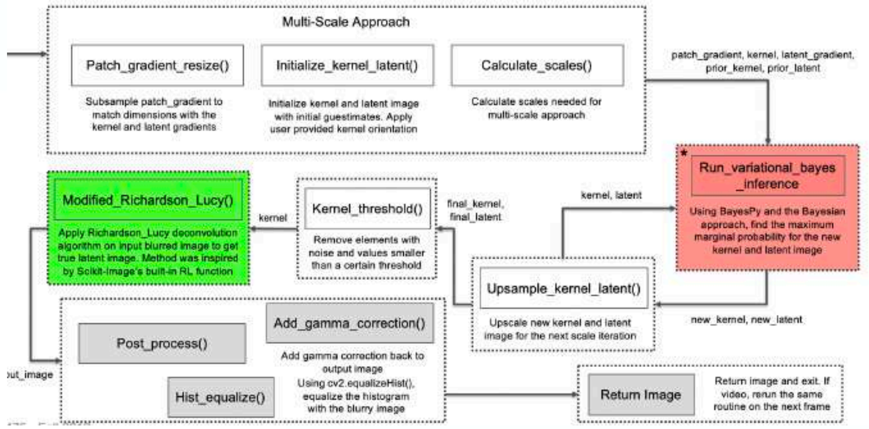
After applying the threshold, the kernel is returned.



```
    # Run Inference
    kernel, latent_gradient = self.run_variational_bayes_inference(patch_gradient_resized, kernel, latent_gradi

    res = cv2.resize(self.blurry_gray_patch_gamma_corrected, None, fx=kernel.shape[1]/self.blur_patch.shape[1],

    kernel_list.append(kernel)
    latent_list.append(latent_gradient)
    patch_subsampled_list.append(res)

out_img = self.modified_richardson_lucy(kernel=self.kernel_threshold(kernel_list[-1]), iterations=10)
out_img = self.post_process(out_img)

return out_img
```



```
def kernel_threshold(self, kernel):
    threshold = np.max(kernel) / 15
    for row in np.arange(0, kernel.shape[0]):
        for column in np.arange(0, kernel.shape[1]):
            if kernel[row, column] < threshold:
                kernel[row, column] = 0
    return kernel
```


Multi-Scale Approach
Patch_gradient_resize()
Subsample patch_gradient to match dimensions with the kernel and latent gradients
Initialize_kernel_latent()
Initialize kernel and latent image with initial guestimates. Apply user provided kernel orientation
Calculate_scales()
Calculate scales needed for multi-scale approach
patch_gradient, kernel, latent_gradient, prior_kernel, prior_latent
*
Run_variational_bayes_inference
Using BayesPy and the Bayesian approach, find the maximum marginal probability for the new kernel and latent image
kernel, latent
new_kernel, new_latent
Upsample_kernel_latent()
Upscale new kernel and latent image for the next scale iteration
final_kernel, final_latent
Kernel_threshold()
Remove elements with noise and values smaller than a certain threshold
kernel
Modified_Richardson_Lucy()
Apply Richardson_Lucy deconvolution algorithm on input blurred image to get true latent image. Method was inspired by Scikit-Image's built-in RL function
out_image
Post_process()
Add_gamma_correction()
Add gamma correction back to output image
Hist_equalize()
Using cv2.equalizeHist(), equalize the histogram with the blurry image
Return Image
Return image and exit. If video, rerun the same routine on the next frame

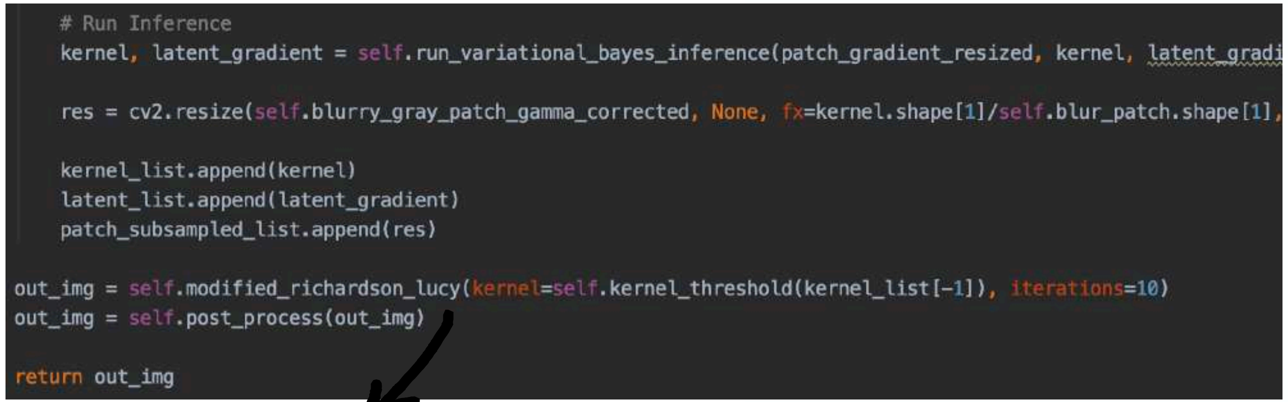
🟩 **Pipeline Progress**

Project Development: Richardson Lucy

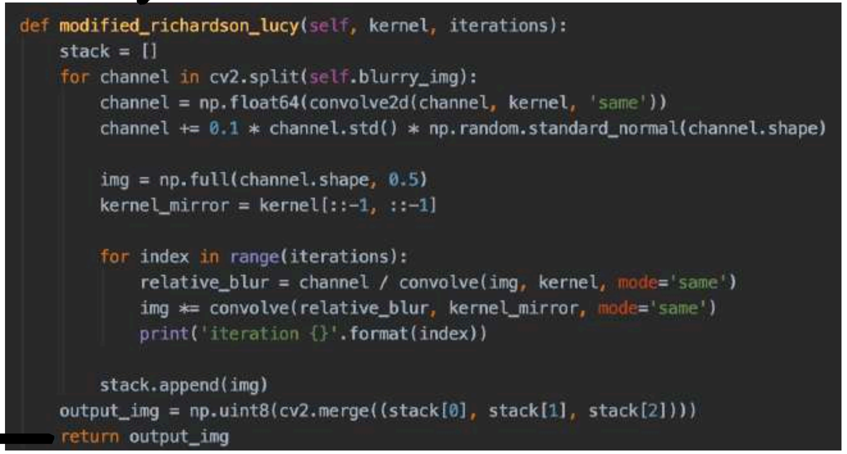
The next step is to take the newly formed kernel and apply it to the original blurry image. The method we will use to do this is the Richardson Lucy Deconvolution algorithm. It runs the same way a typical convolution algorithm works but instead in an inverse fashion to reconstruct the original sharp pixels.

The algorithm and code was inspired by the RL function found in the Scikit Image library. The code was extended to support multi channel images and an input blur kernel instead of using a static point spread function.

For the code, first the image is split into channels, then for each channel, a noise function is applied. A mirror of the kernel is created and a relative blur is created by dividing by the convolution of the image and the kernel. This value is then convolved with the mirrored kernel to get the output image.

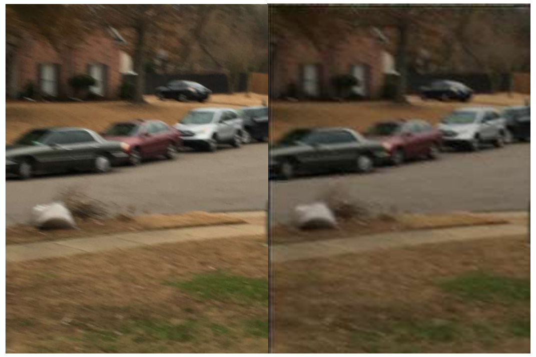


```
        # Run Inference
        kernel, latent_gradient = self.run_variational_bayes_inference(patch_gradient_resized, kernel, latent_gradi

        res = cv2.resize(self.blurry_gray_patch_gamma_corrected, None, fx=kernel.shape[1]/self.blur_patch.shape[1],

        kernel_list.append(kernel)
        latent_list.append(latent_gradient)
        patch_subsampled_list.append(res)

    out_img = self.modified_richardson_lucy(kernel=self.kernel_threshold(kernel_list[-1]), iterations=10)
    out_img = self.post_process(out_img)

    return out_img
```



```
def modified_richardson_lucy(self, kernel, iterations):
    stack = []
    for channel in cv2.split(self.blurry_img):
        channel = np.float64(convolve2d(channel, kernel, 'same'))
        channel += 0.1 * channel.std() * np.random.standard_normal(channel.shape)

        img = np.full(channel.shape, 0.5)
        kernel_mirror = kernel[::-1, ::-1]

        for index in range(iterations):
            relative_blur = channel / convolve(img, kernel, mode='same')
            img *= convolve(relative_blur, kernel_mirror, mode='same')
            print('iteration {}'.format(index))

        stack.append(img)
    output_img = np.uint8(cv2.merge((stack[0], stack[1], stack[2])))
    return output_img
```

Blurry Image



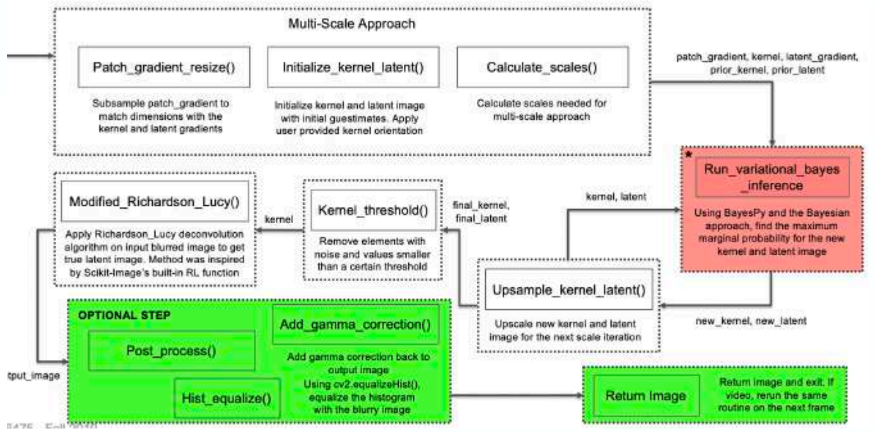
Output image after deconvolution

Project Development: Optional Post Process

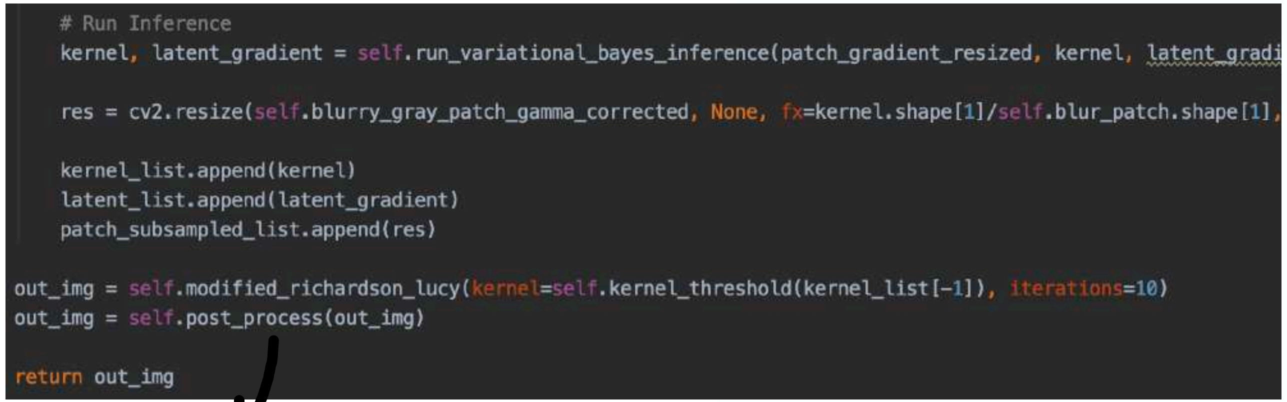
The final step is an optional image post-processing step where the output deconvoluted image is passed onto the function post_process(). The authors of the paper included this step into their method in order to add gamma correction back into the image. They also included histogram equalization to match the histogram distribution of the output image with the input blurry image.

Based on my observations and experiments, this step often created a ”washed out” effect on the output image which led to unsatisfactory results. The following section will outline the results after the post-processing step. Based on this, I opted to return the result directly after the deconvolution step. The image shown previously in the demonstration showcase these images.

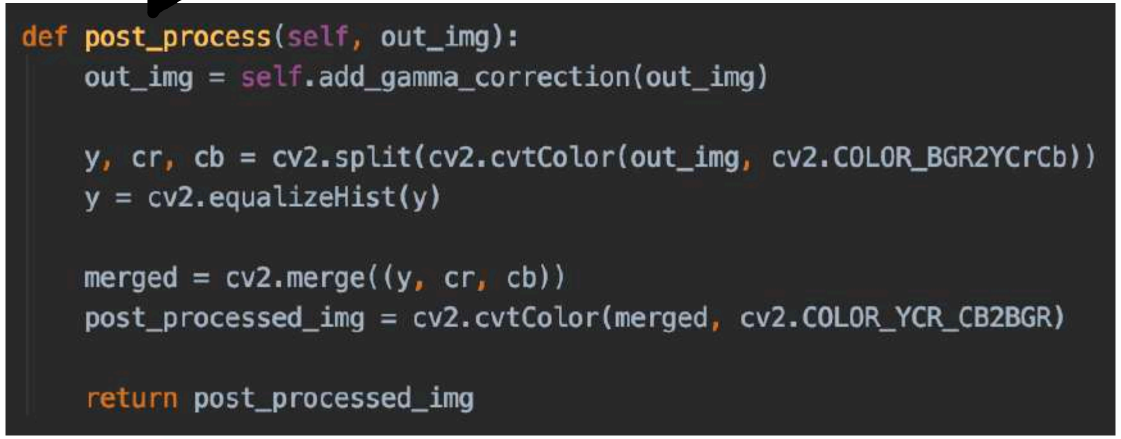
The functions used include cv2.split to split the image into channels, cv2.merge to merge the channels back together, cv2.equalizeHist to perform histogram equalization, and cv2.cvtColor to convert the color scheme from YCRCB to BGR. Afterwards, the image is returned. If the input is video, the system is looped again for the next few frames in the video sequence.


Multi-Scale Approach
Patch_gradient_resize()
Subsample patch_gradient to match dimensions with the kernel and latent gradients
Initialize_kernel_latent()
Initialize kernel and latent image with initial guestimates. Apply user provided kernel orientation
Calculate_scales()
Calculate scales needed for multi-scale approach
patch_gradient, kernel, latent_gradient, prior_kernel, prior_latent
Run_variational_bayes_inference
Using BayesPy and the Bayesian approach, find the maximum marginal probability for the new kernel and latent image
kernel, latent
new_kernel, new_latent
Upsample_kernel_latent()
Upscale new kernel and latent image for the next scale iteration
final_kernel, final_latent
Kernel_threshold()
Remove elements with noise and values smaller than a certain threshold
kernel
Modified_Richardson_Lucy()
Apply Richardson_Lucy deconvolution algorithm on input blurred image to get true latent image. Method was inspired by Scikit-Image's built-in RL function
tput_image
OPTIONAL STEP
Post_process()
Add_gamma_correction()
Add gamma correction back to output image
Hist_equalize()
Using cv2.equalizeHist(), equalize the histogram with the blurry image
Return Image
Return image and exit. If video, rerun the same routine on the next frame

🟩 **Pipeline Progress**

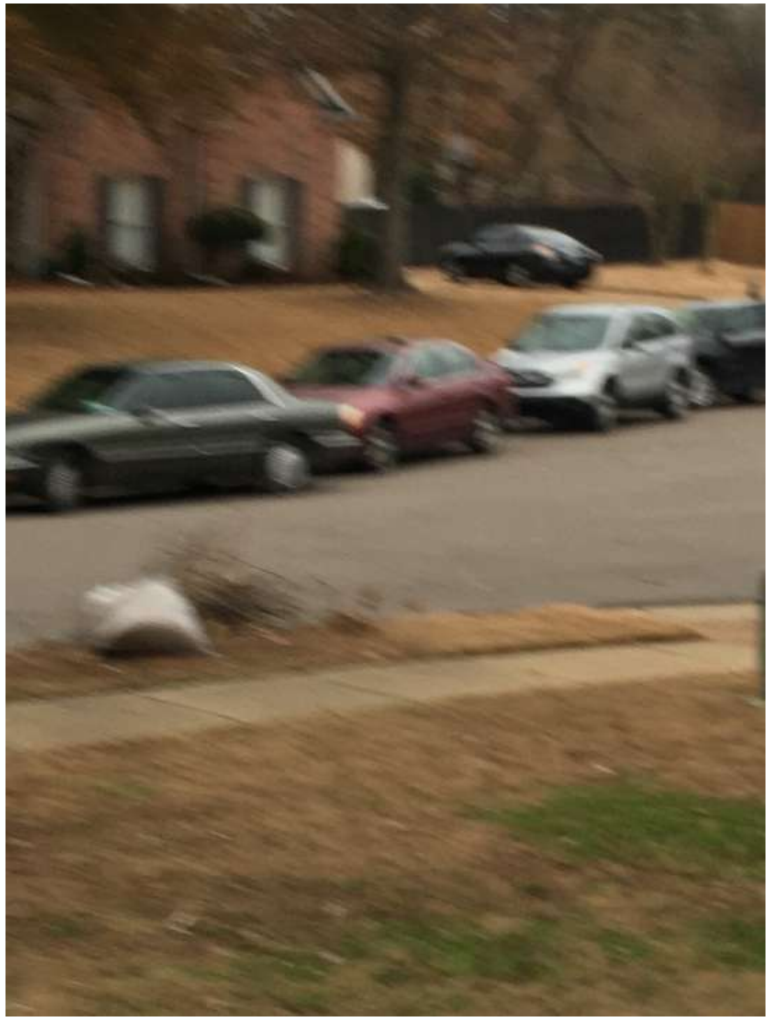


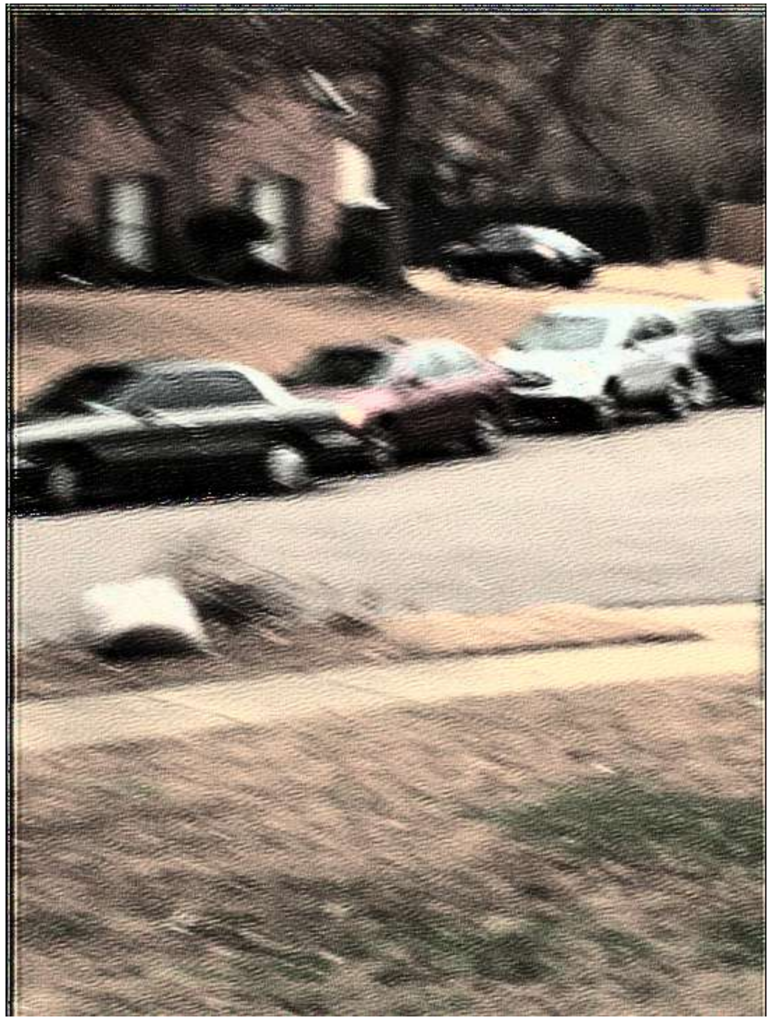
```
    # Run Inference
    kernel, latent_gradient = self.run_variational_bayes_inference(patch_gradient_resized, kernel, latent_gradi

    res = cv2.resize(self.blurry_gray_patch_gamma_corrected, None, fx=kernel.shape[1]/self.blur_patch.shape[1],

    kernel_list.append(kernel)
    latent_list.append(latent_gradient)
    patch_subsampled_list.append(res)

out_img = self.modified_richardson_lucy(kernel=self.kernel_threshold(kernel_list[-1]), iterations=10)
out_img = self.post_process(out_img)

return out_img
```

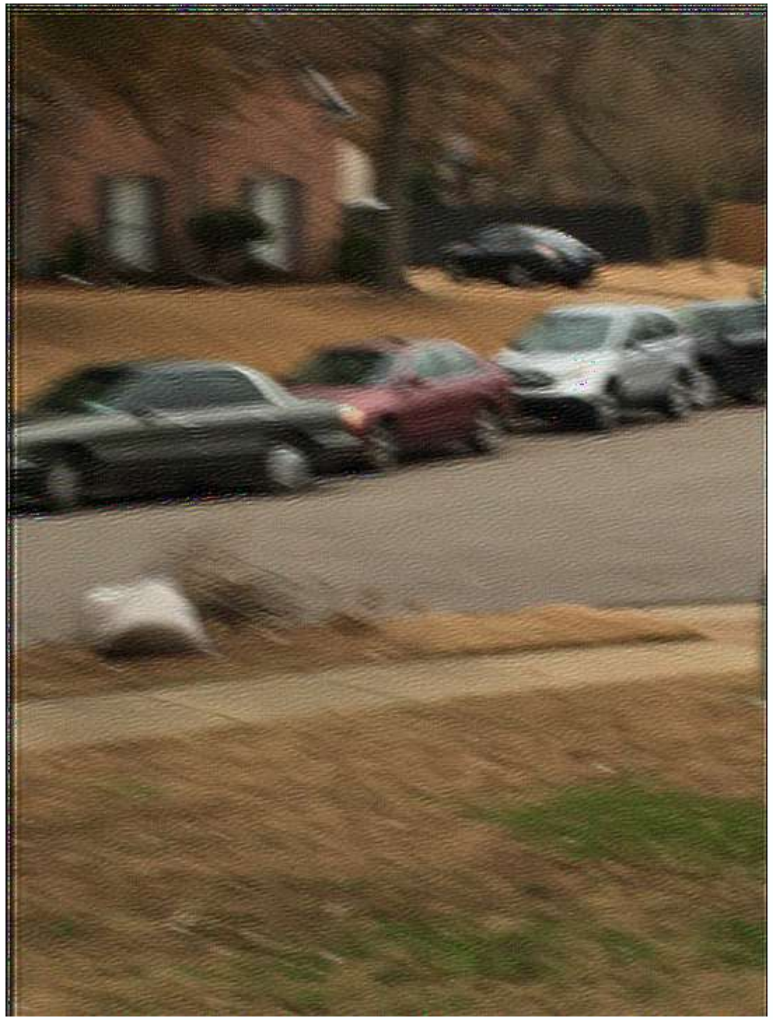


```
def post_process(self, out_img):
    out_img = self.add_gamma_correction(out_img)

    y, cr, cb = cv2.split(cv2.cvtColor(out_img, cv2.COLOR_BGR2YCrCb))
    y = cv2.equalizeHist(y)

    merged = cv2.merge((y, cr, cb))
    post_processed_img = cv2.cvtColor(merged, cv2.COLOR_YCR_CB2BGR)

    return post_processed_img
```

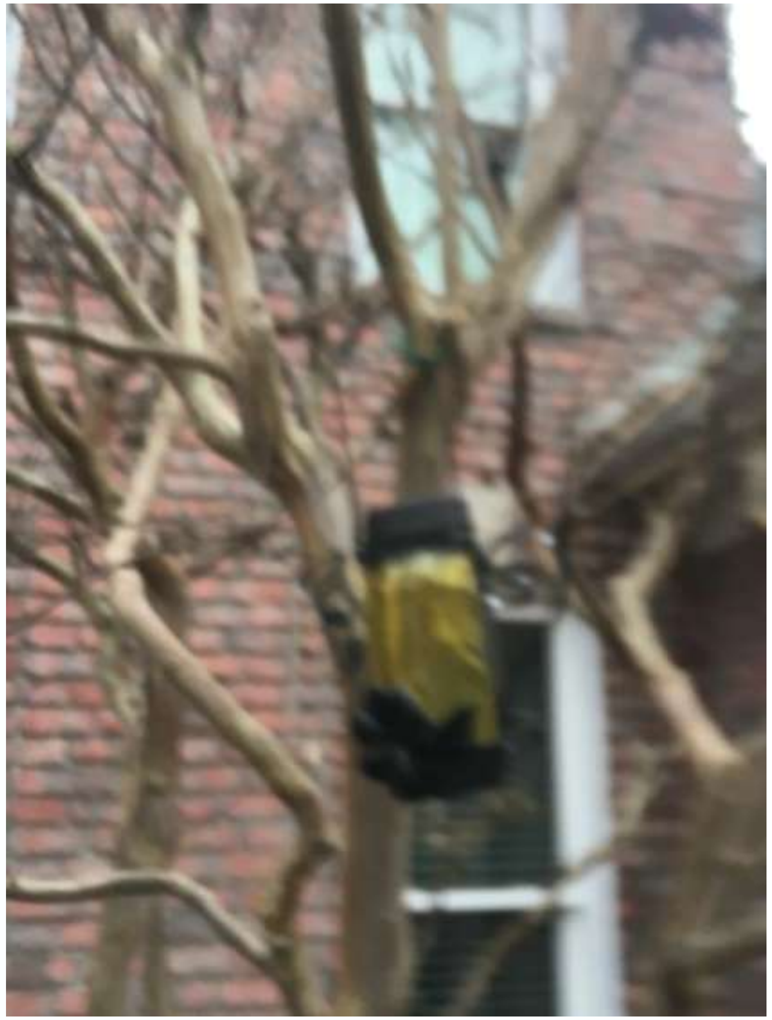
Demonstration: Post Processing Set 1

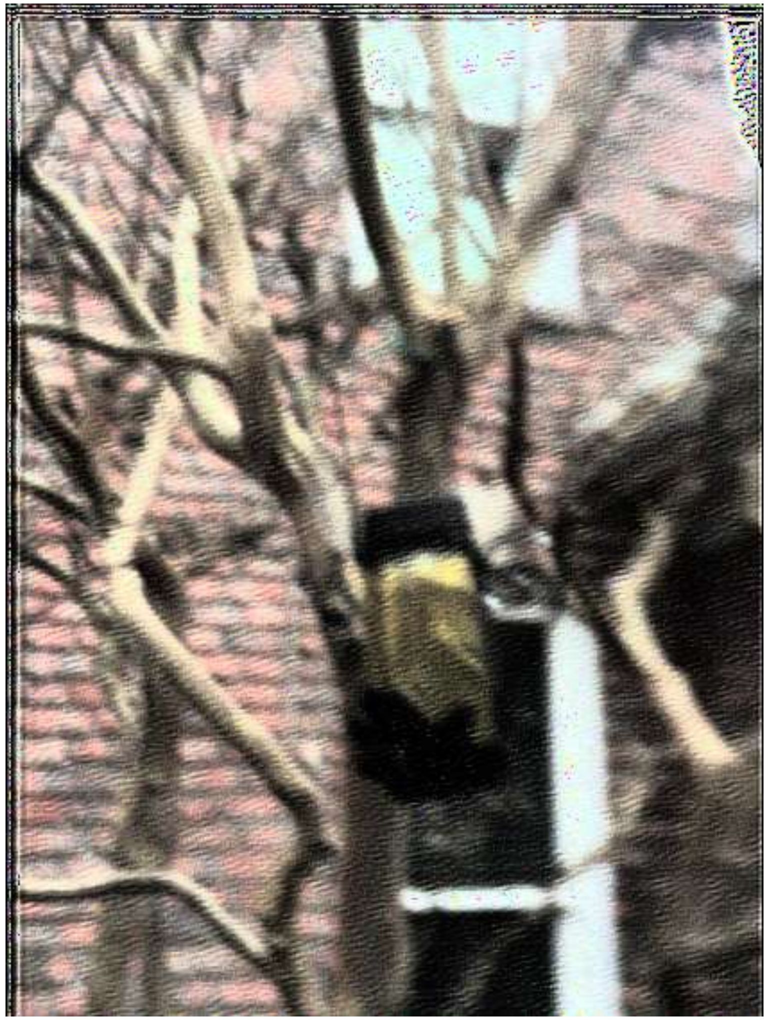

Input Blurry Image

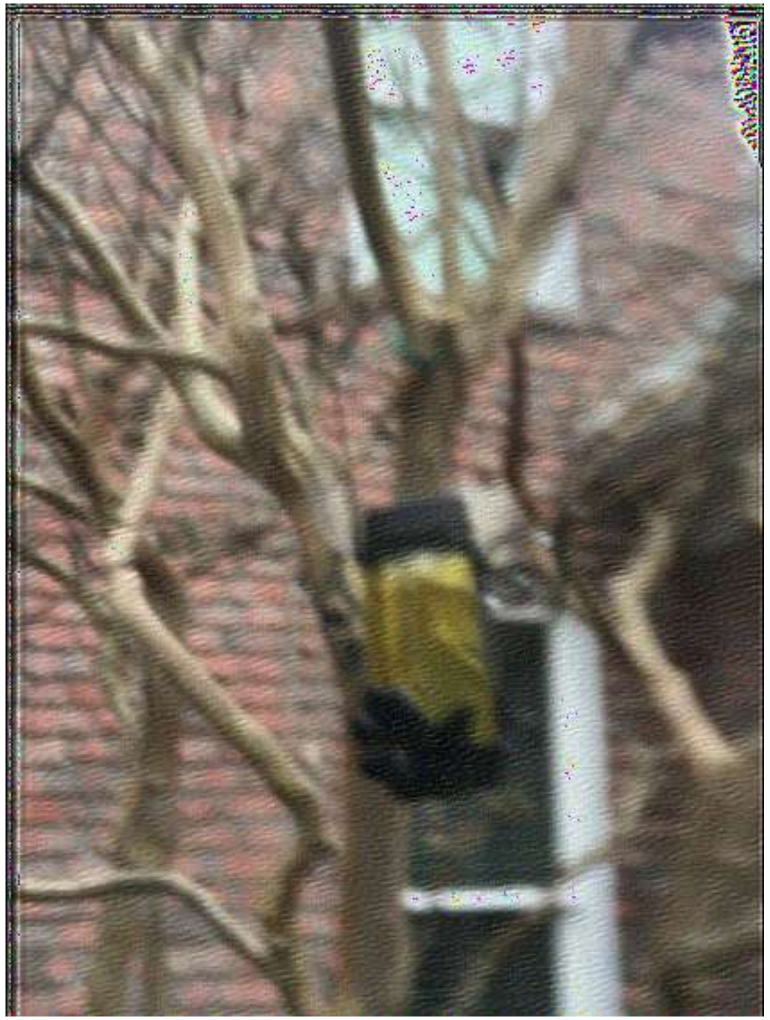

Post Processed Image


Output Latent Image

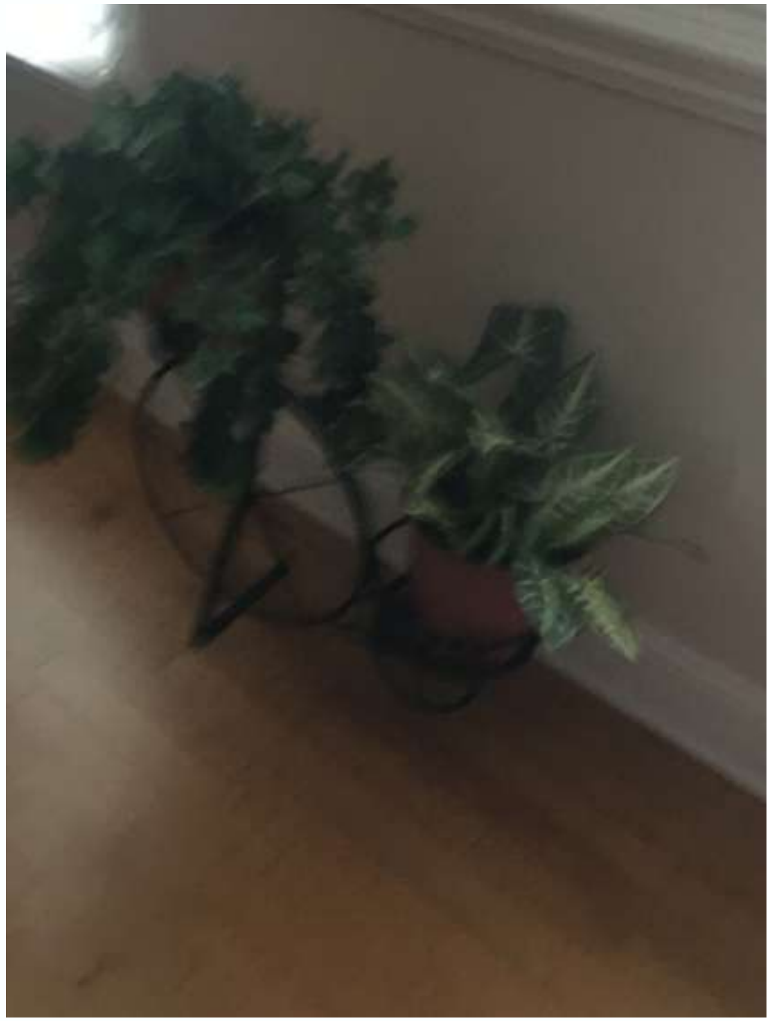
Demonstration: Post Processing Set 2

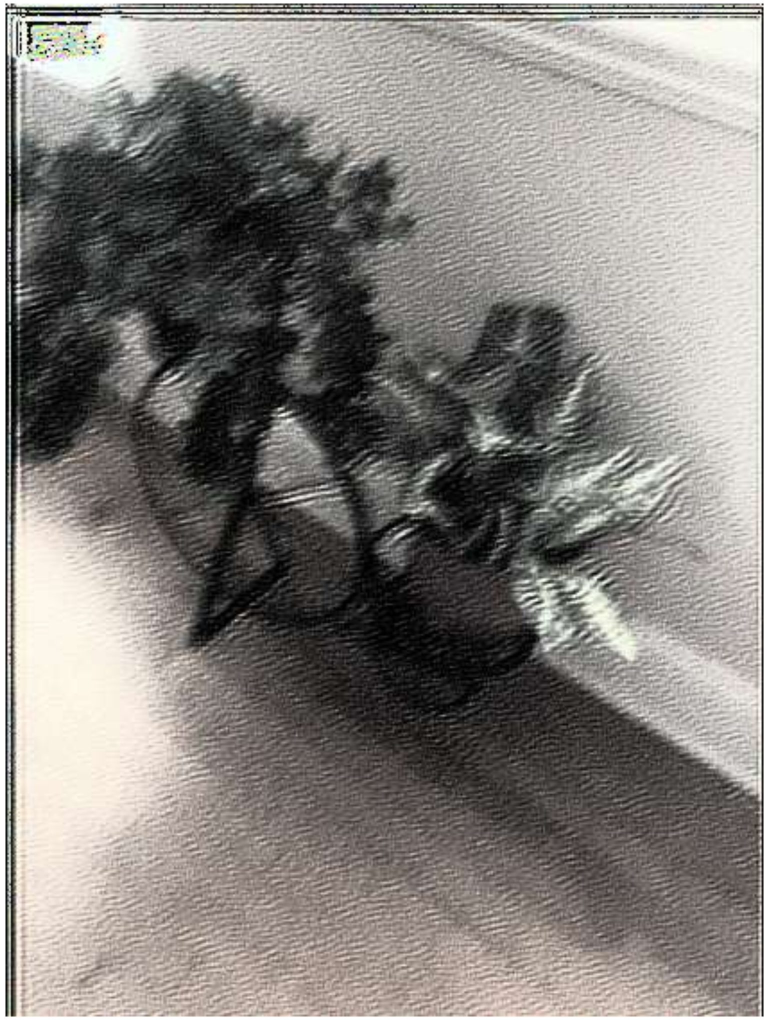

Input Blurry Image

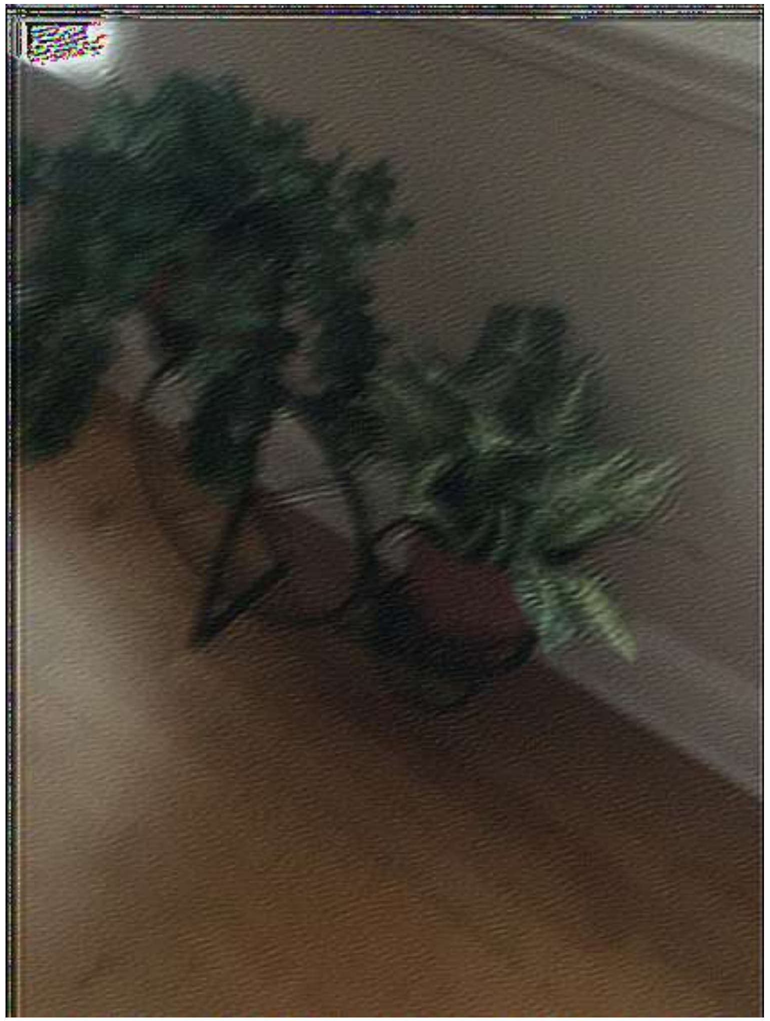

Post Processed Image


Output Latent Image

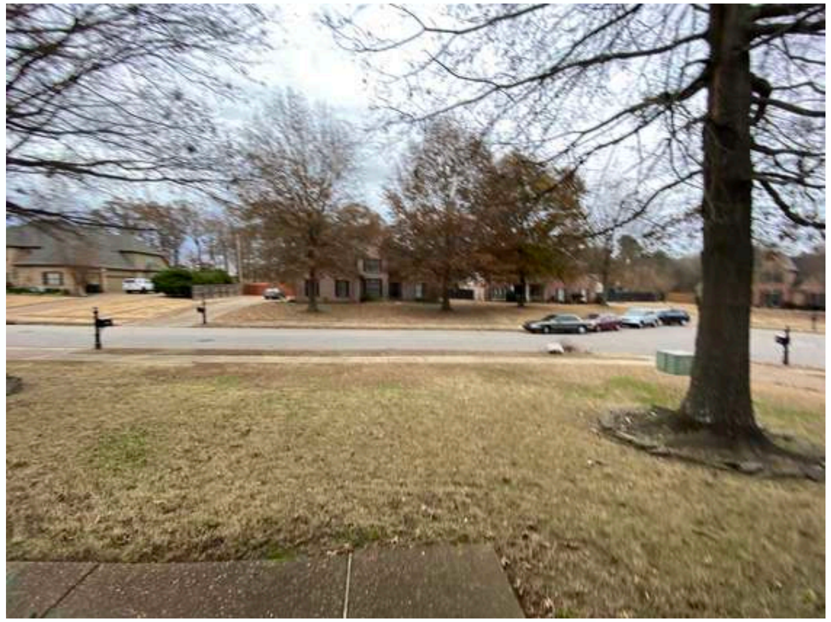
Demonstration: Post Processing Set 3

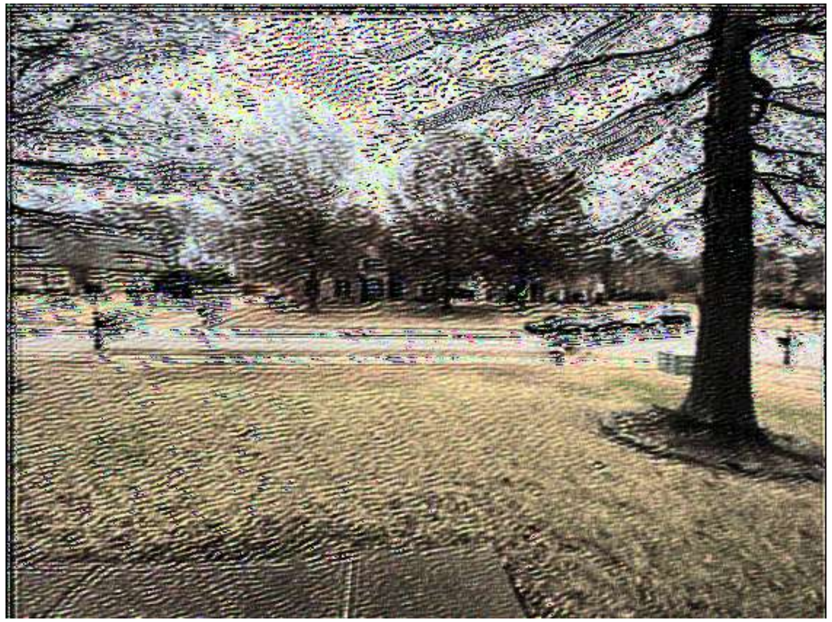

Input Blurry Image

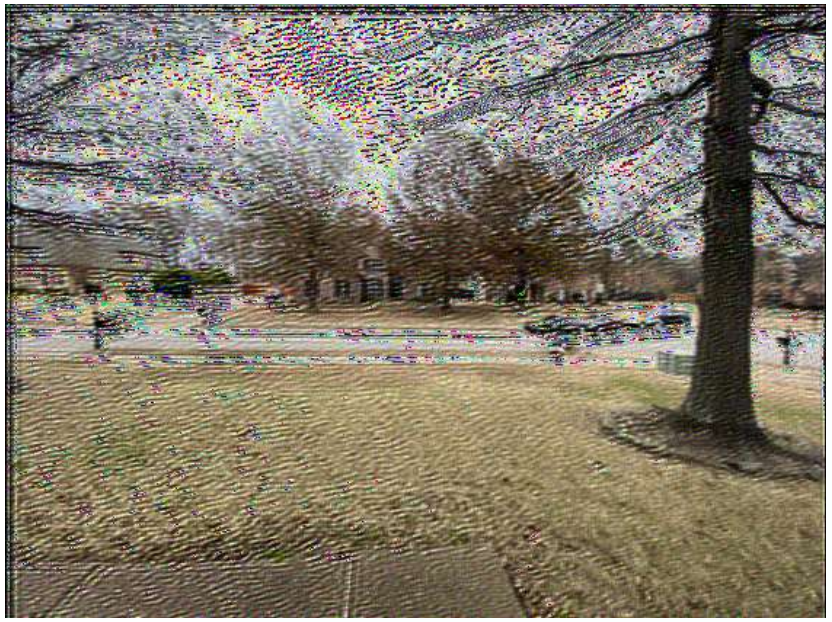

Post Processed Image


Output Latent Image

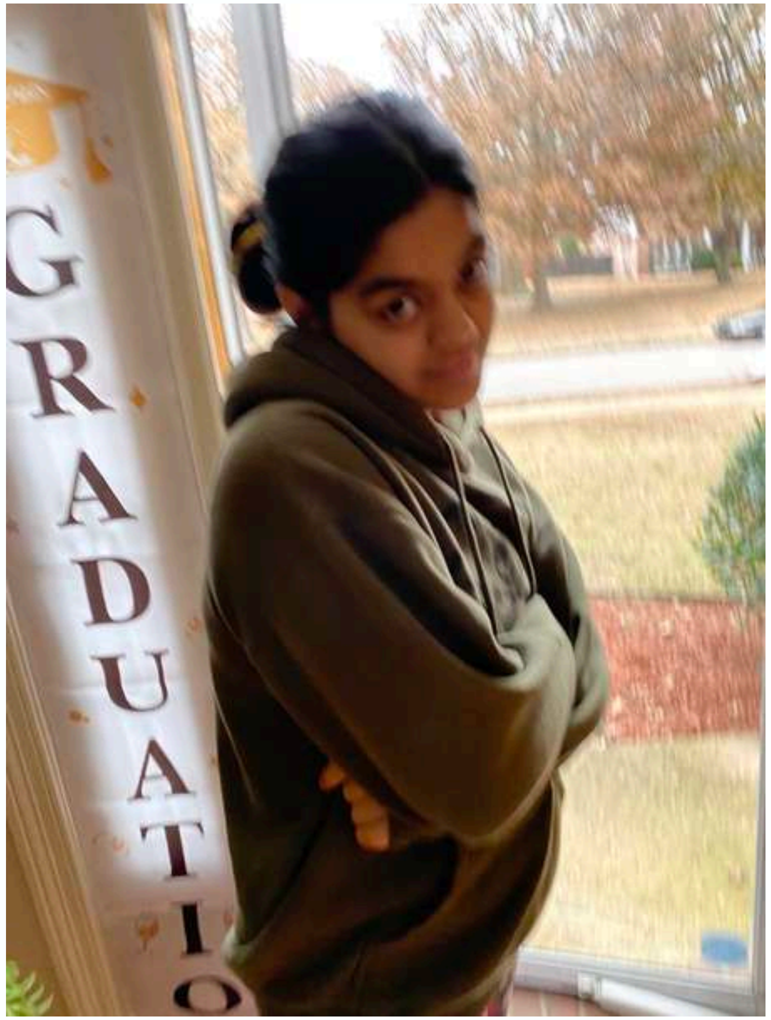
Demonstration: Post Processing Set 4


Input Blurry Image

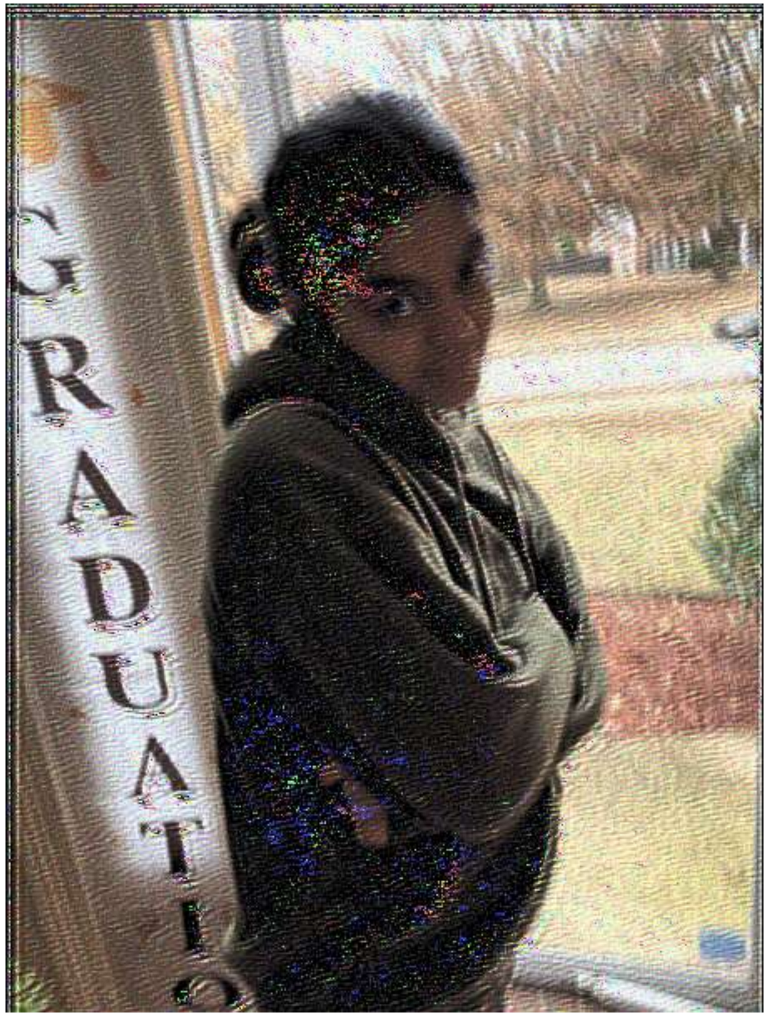

Post Processed Image


Output Latent Image

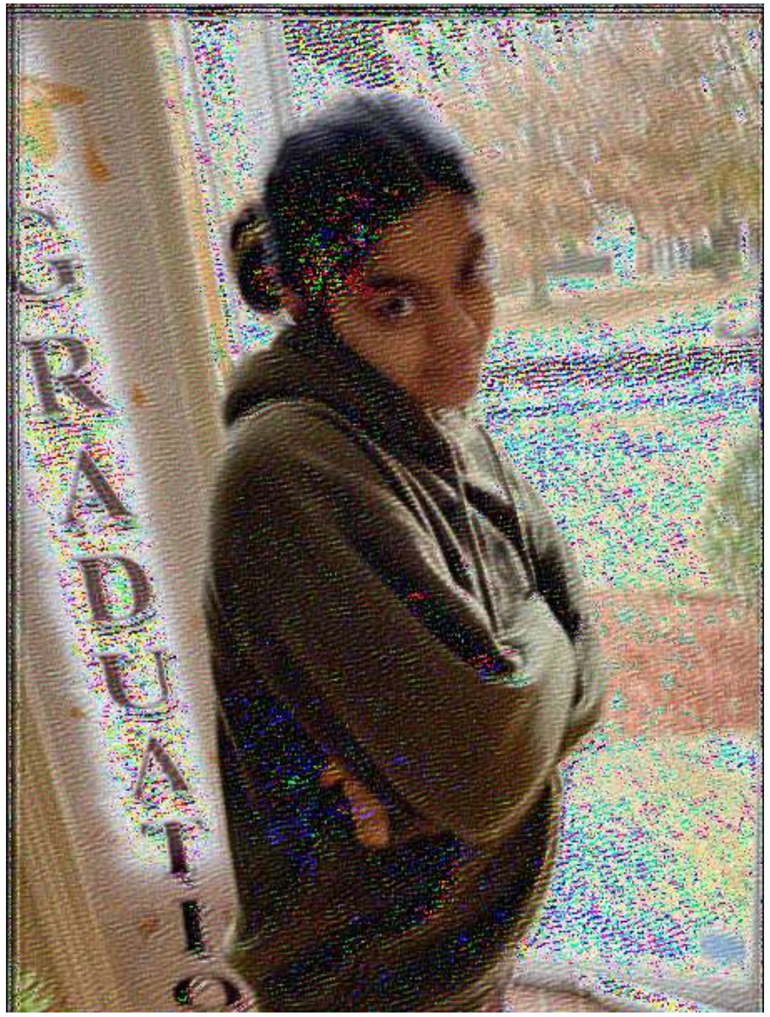
Demonstration: Post Processing Set 5



Input Blurry Image

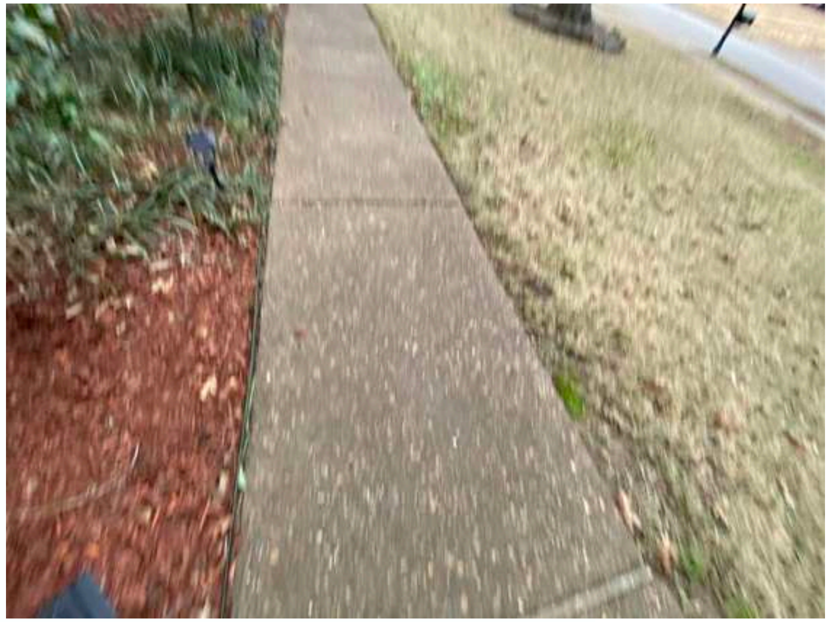


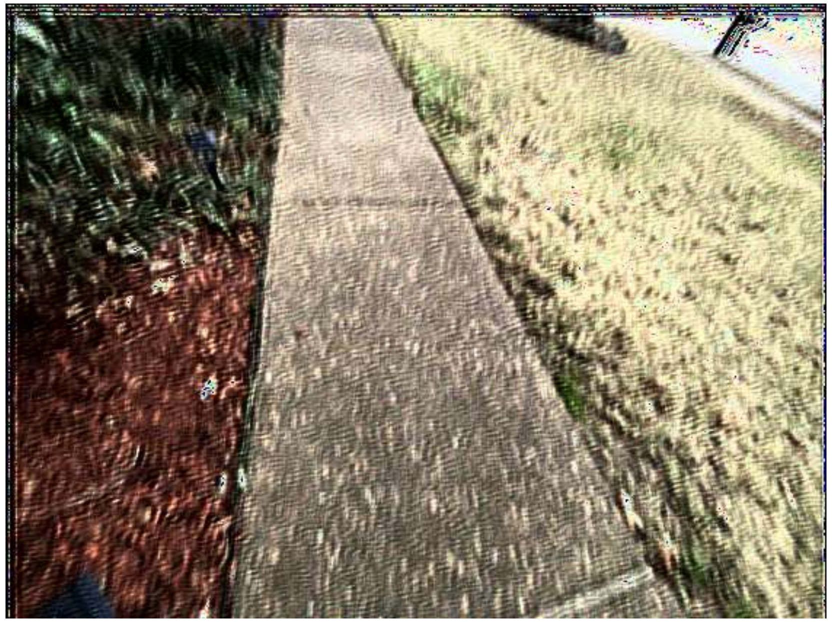
Post Processed Image

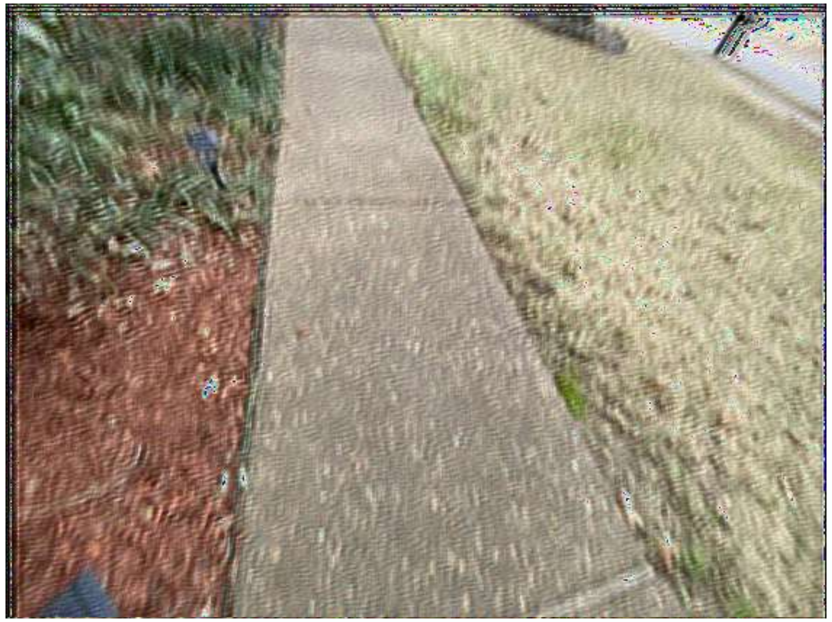


Output Latent Image

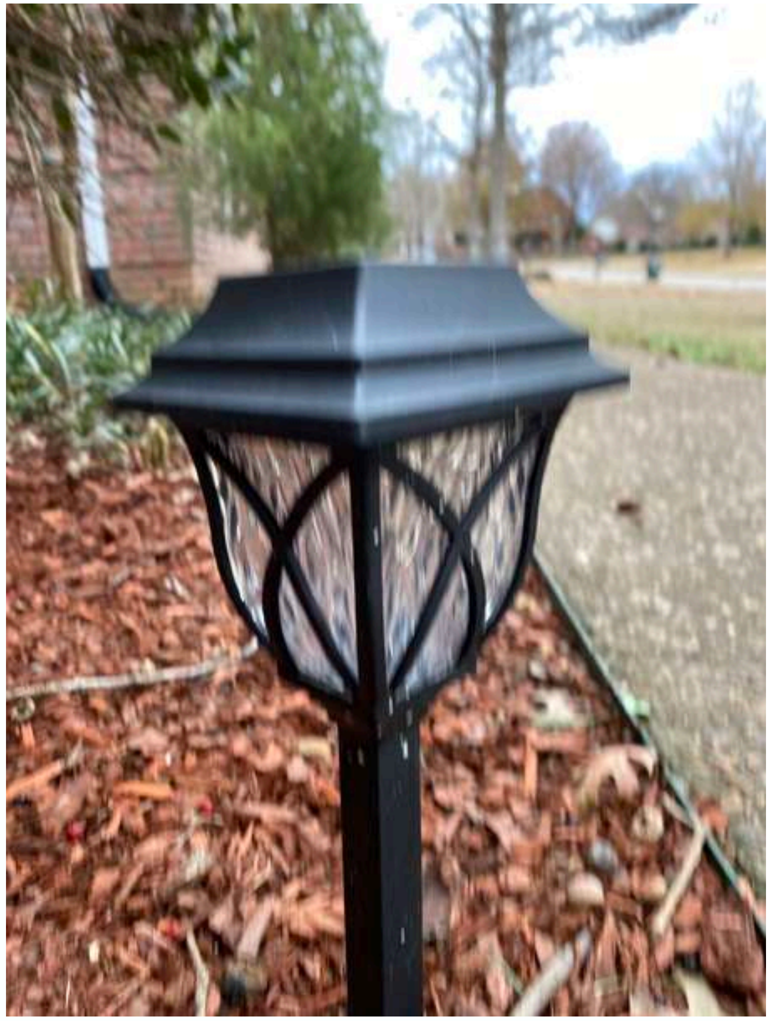
Demonstration: Post Processing Set 6

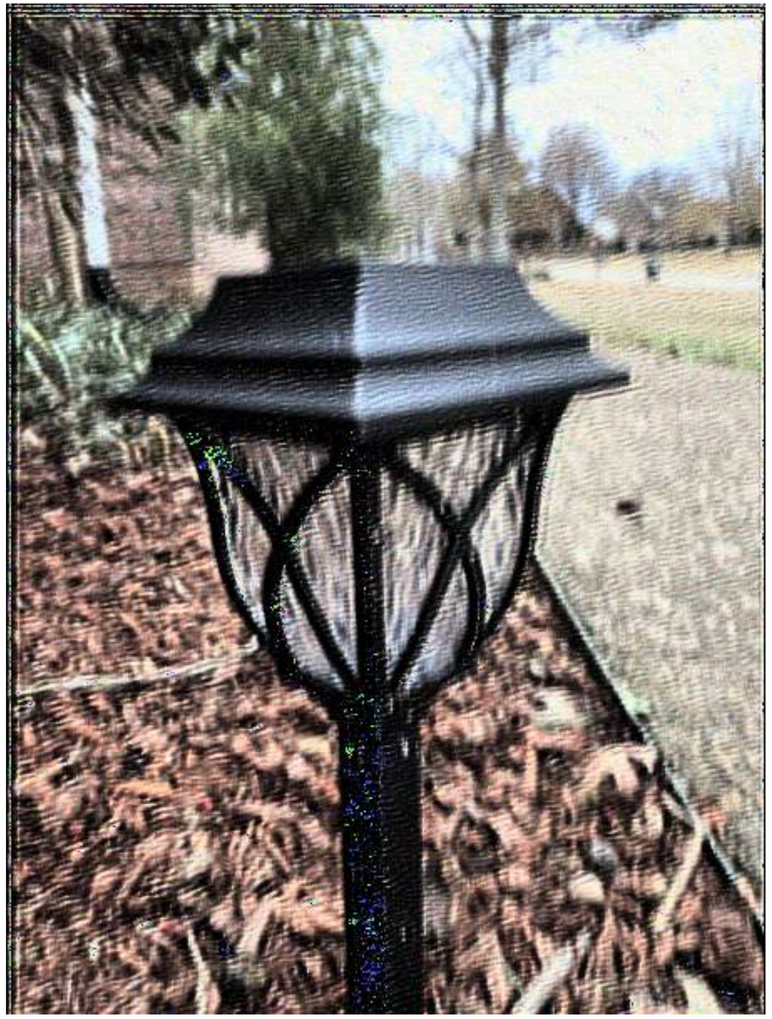

Input Blurry Image

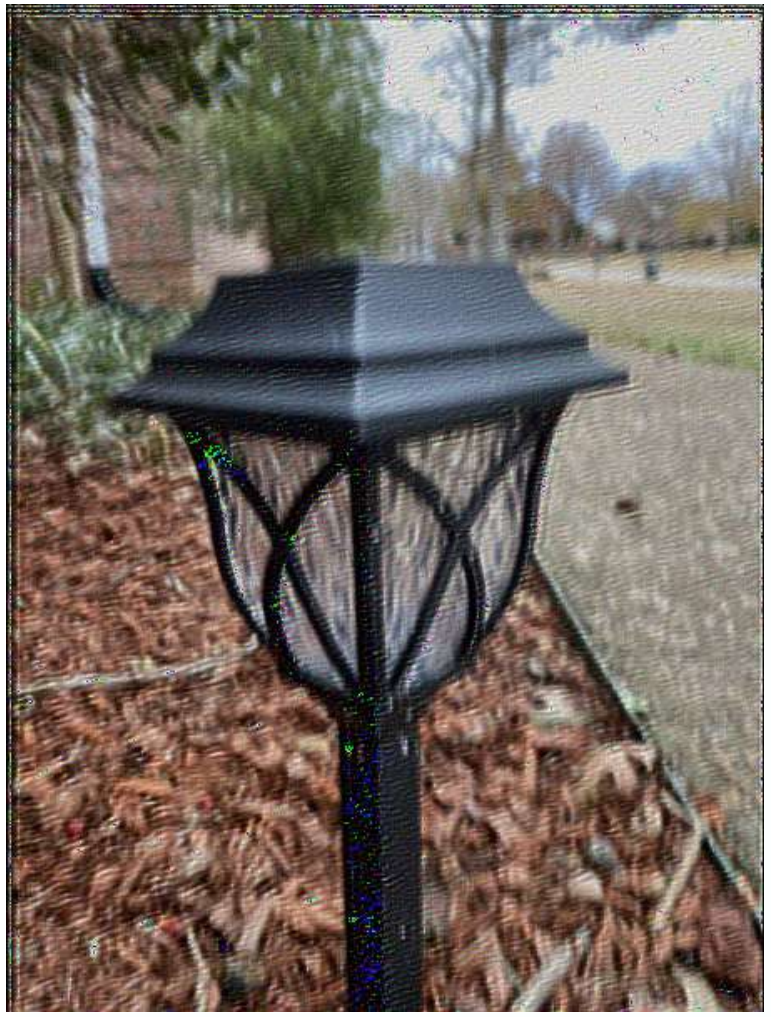

Post Processed Image


Output Latent Image

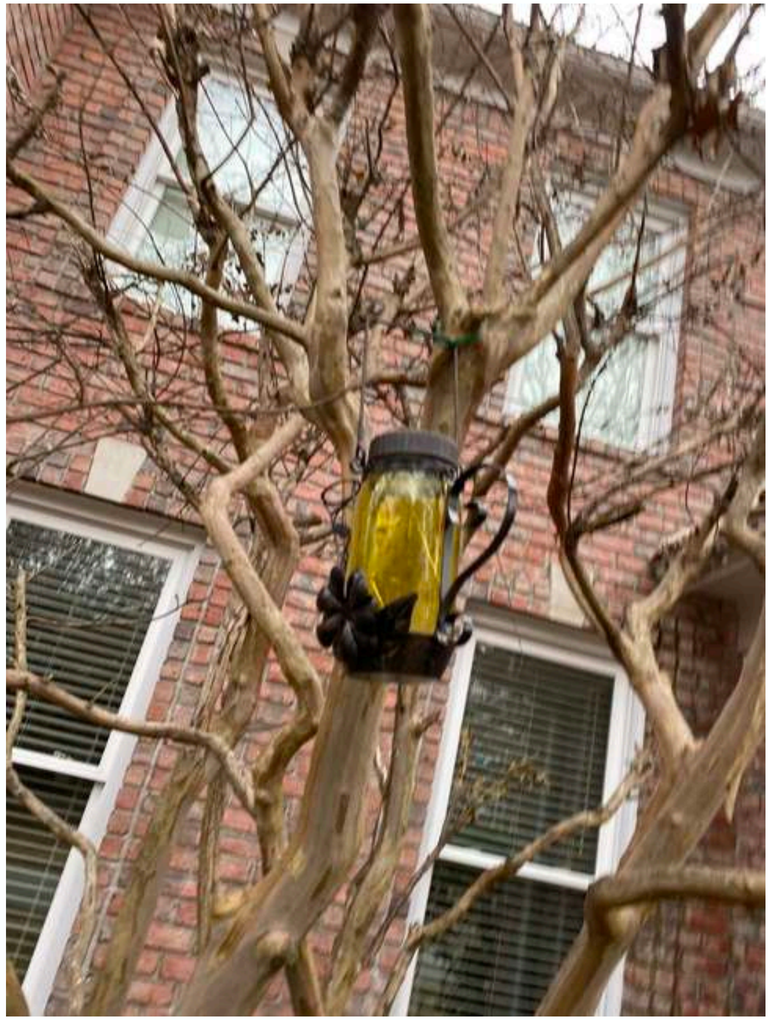
Demonstration: Post Processing Set 7

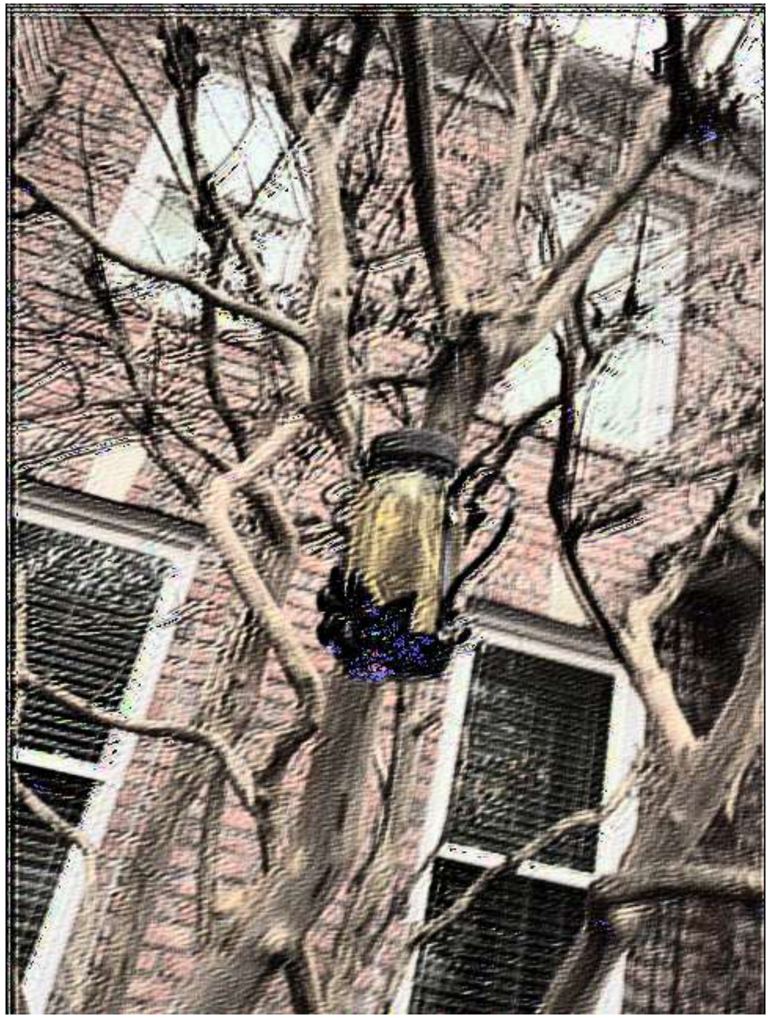

Input Blurry Image

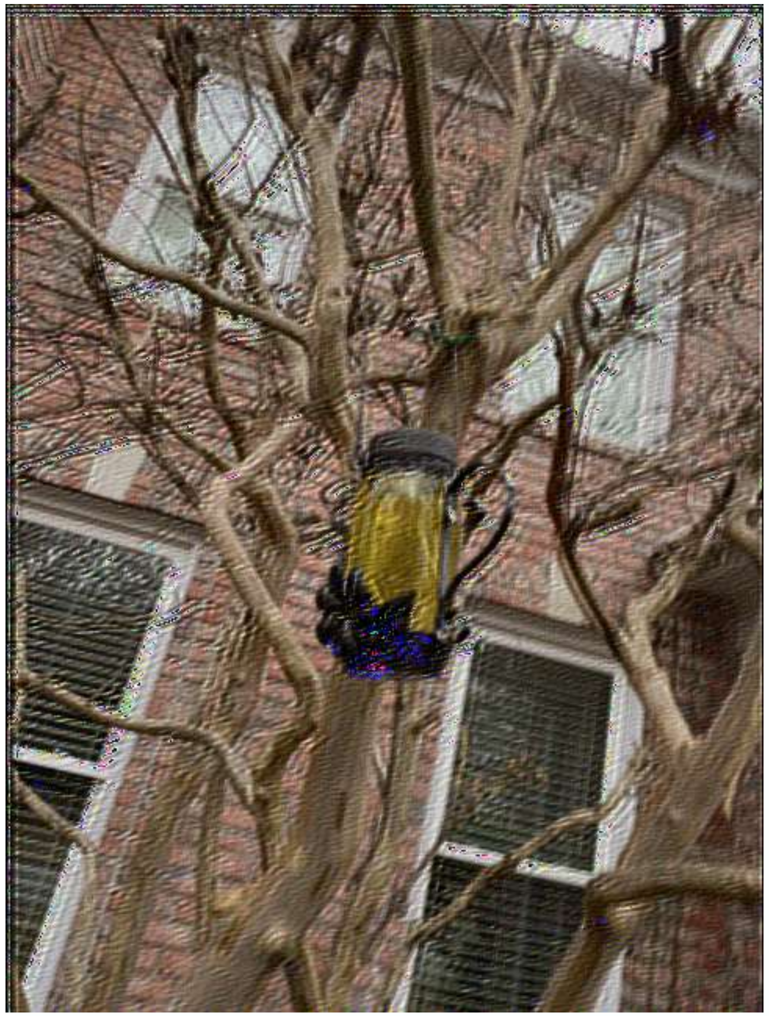

Post Processed Image


Output Latent Image

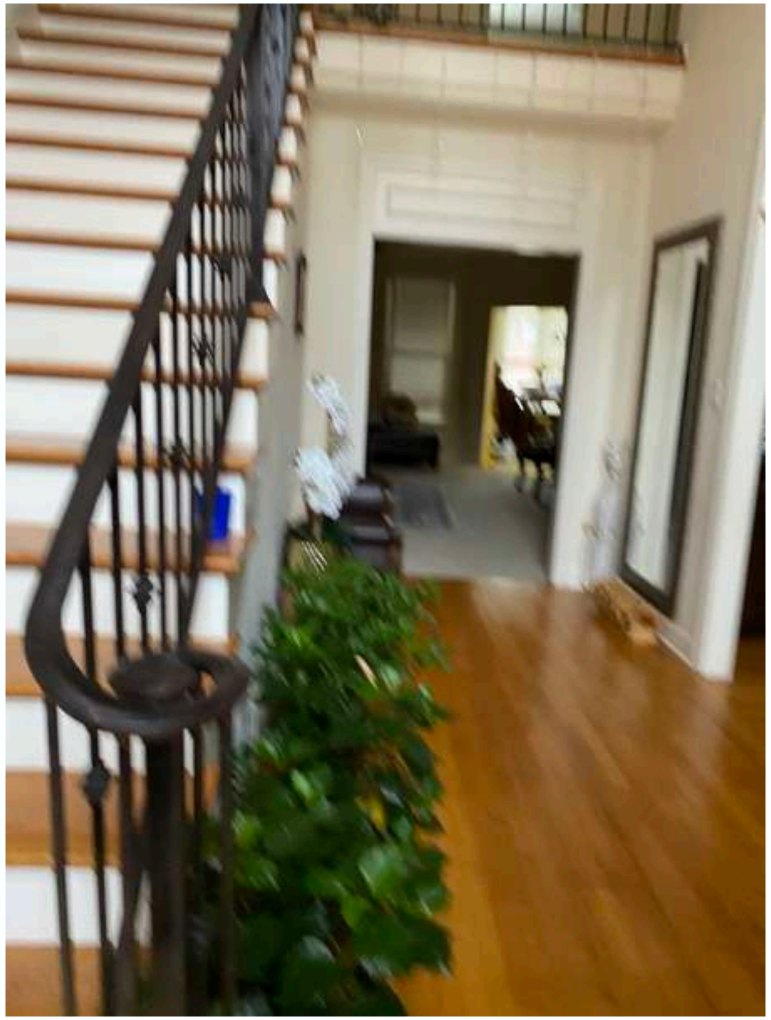
Demonstration: Post Processing Set 8

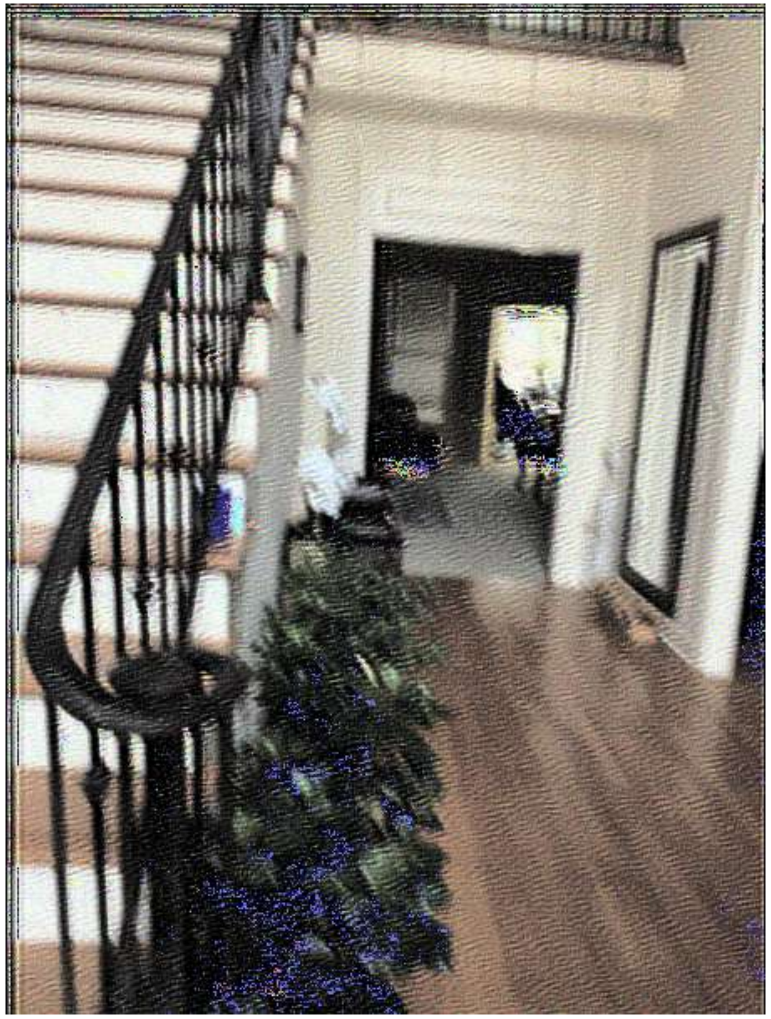

Input Blurry Image

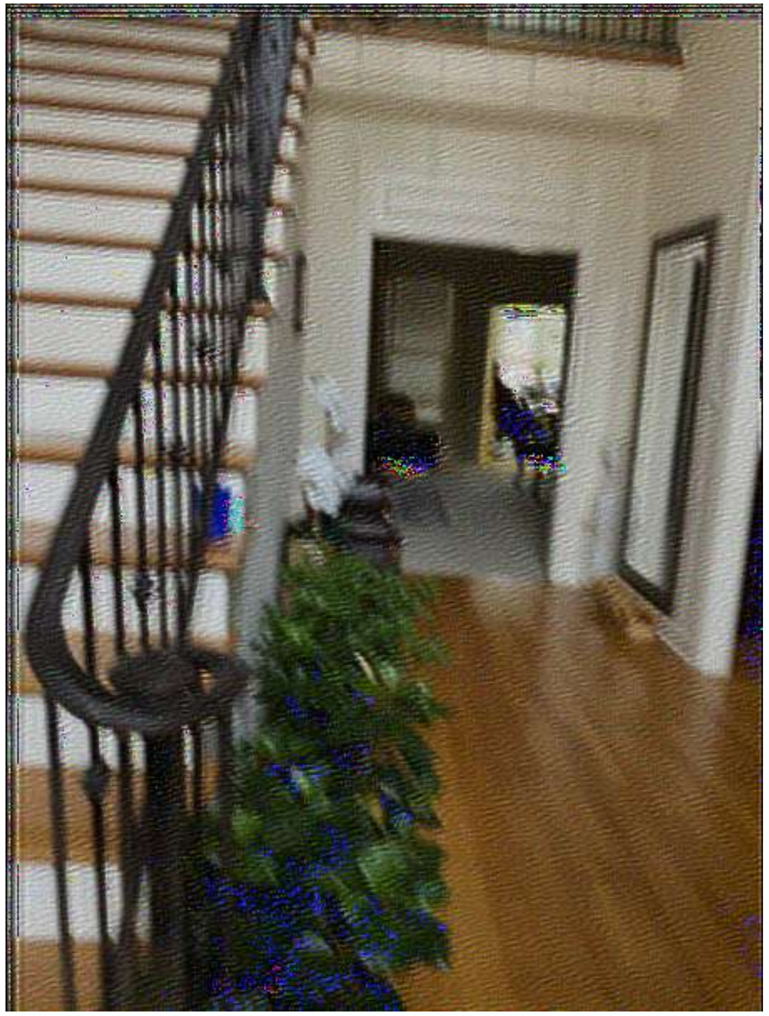

Post Processed Image


Output Latent Image

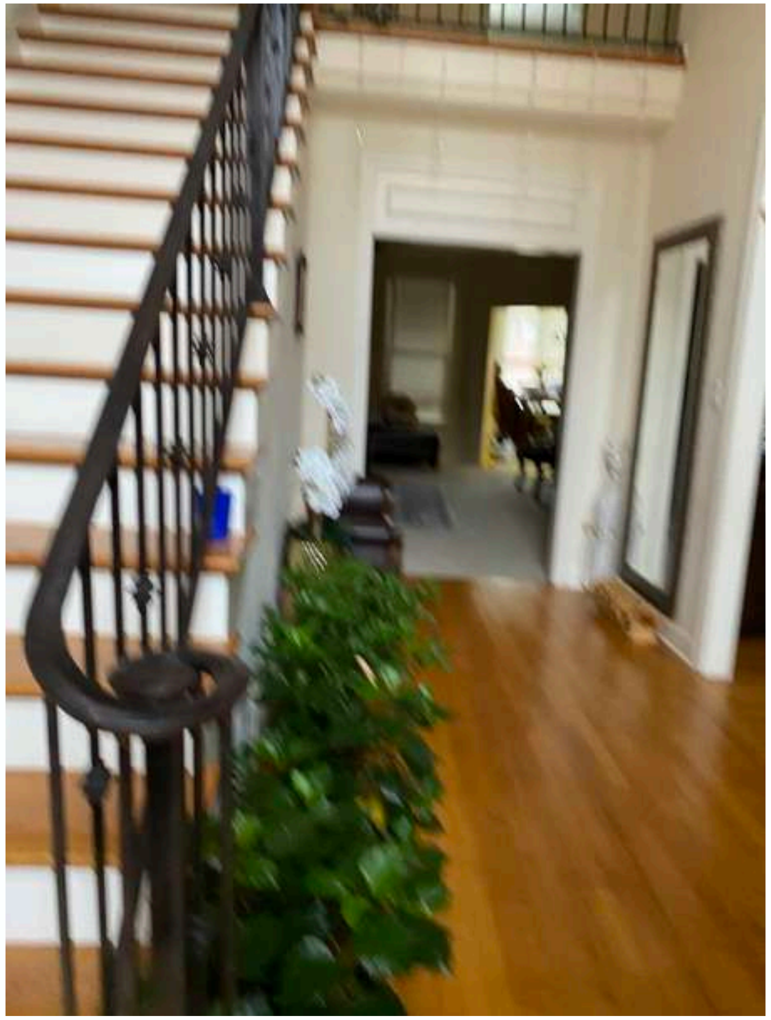
Demonstration: Post Processing Set 9

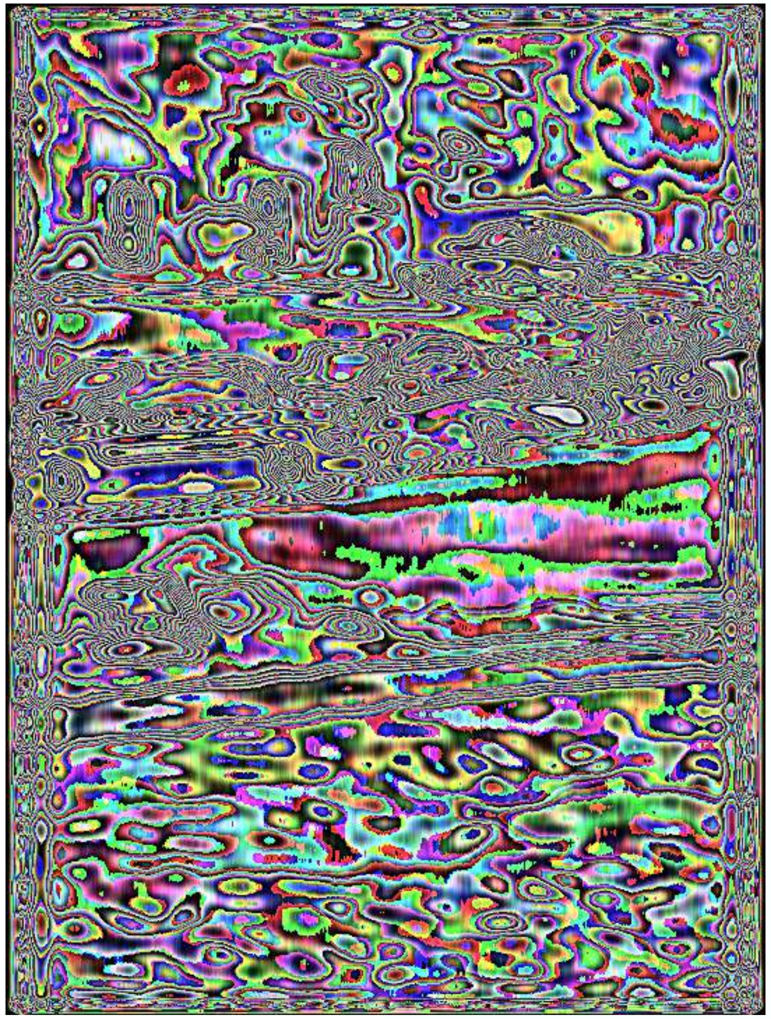

Input Blurry Image


Post Processed Image

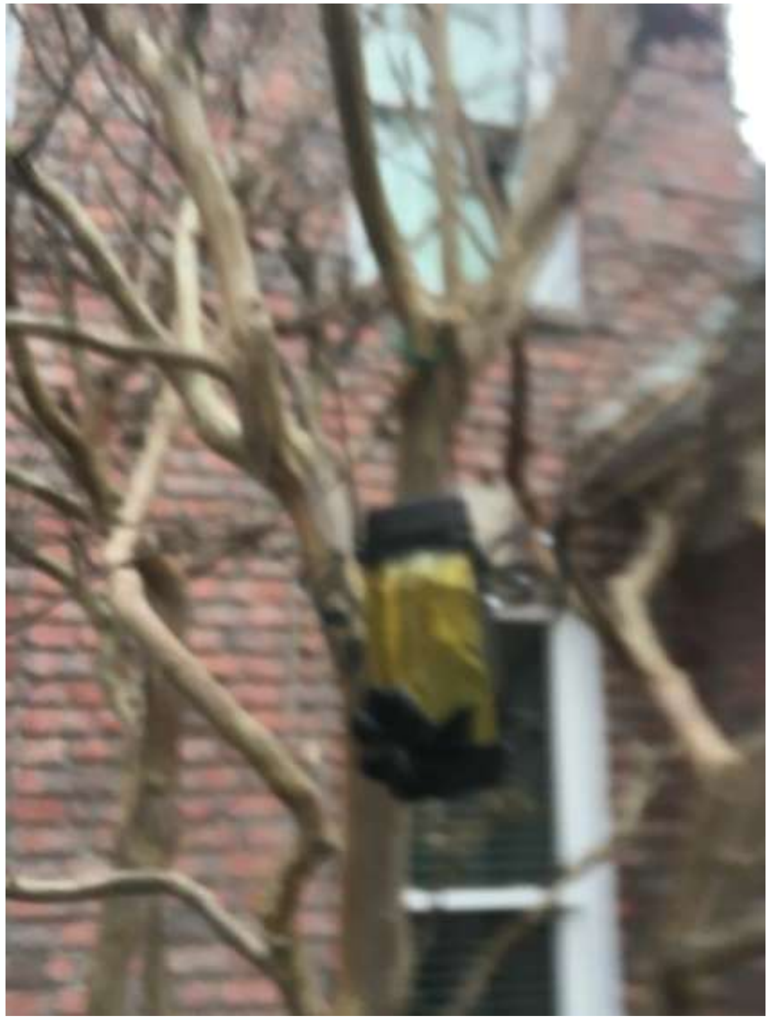

Output Latent Image

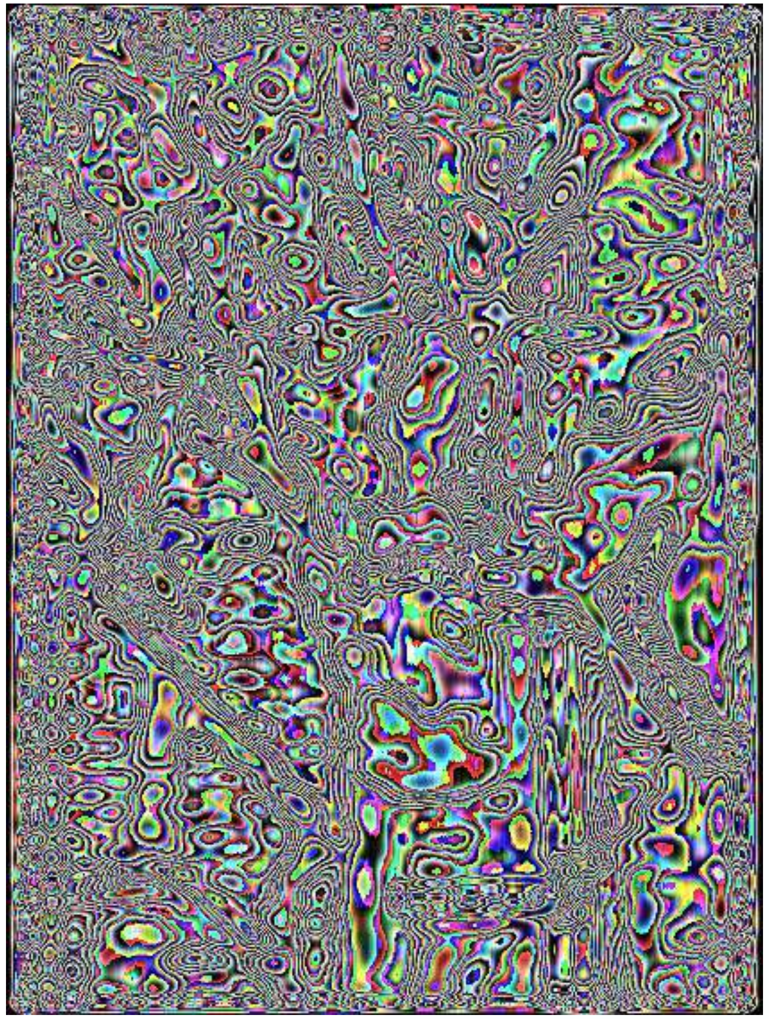
Intermediate Results of Bayes Inference: Set 1


Input Blurry Image

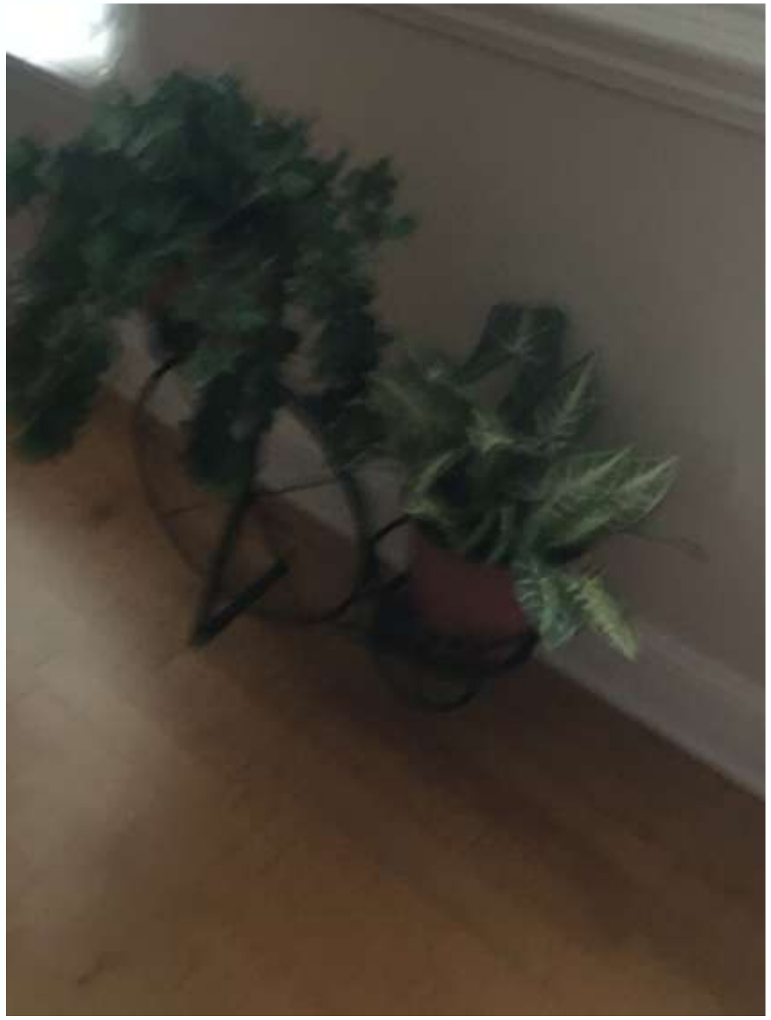

Output Latent Image

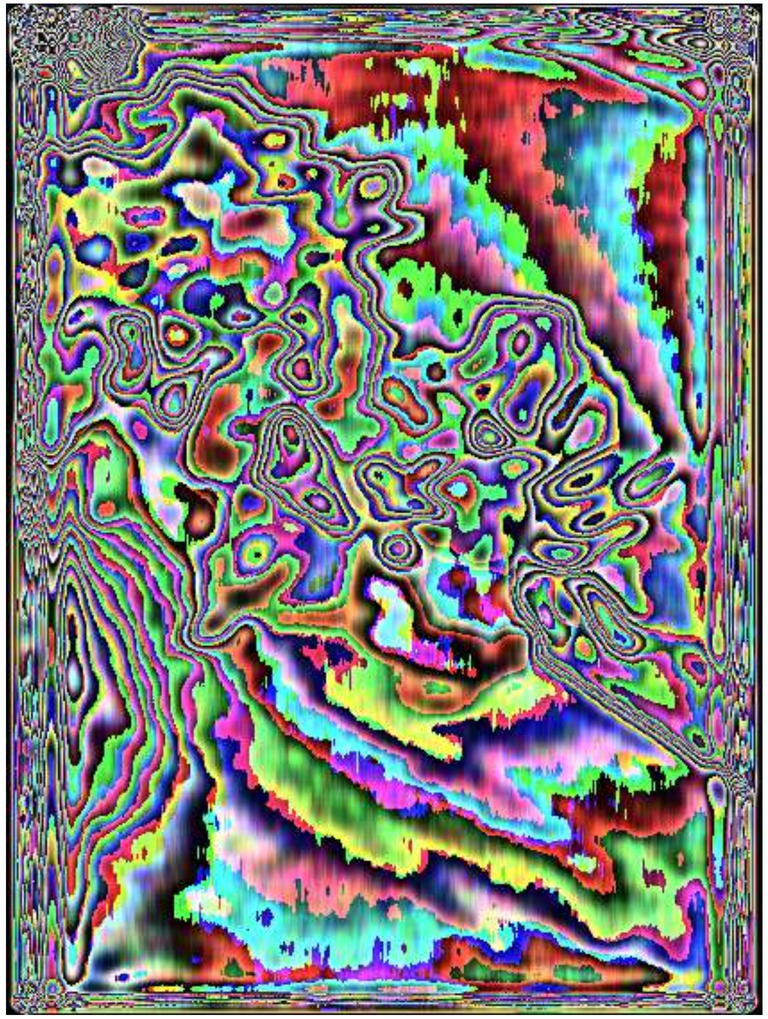
Intermediate Results of Bayes Inference: Set 2


Input Blurry Image

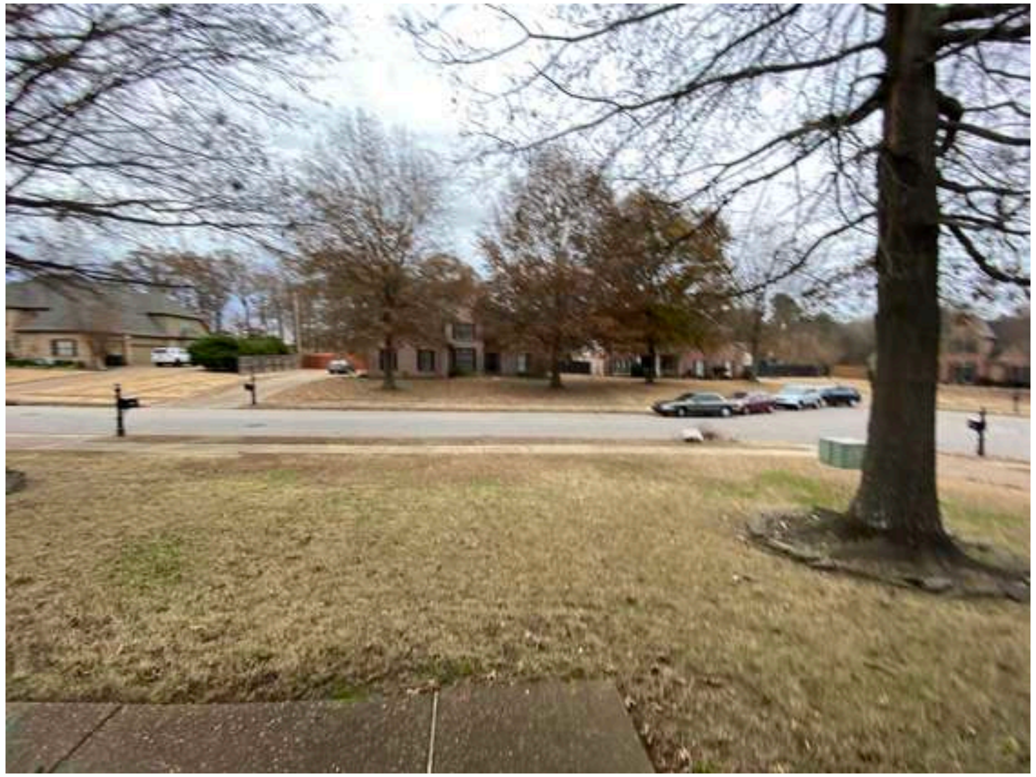

Output Latent Image

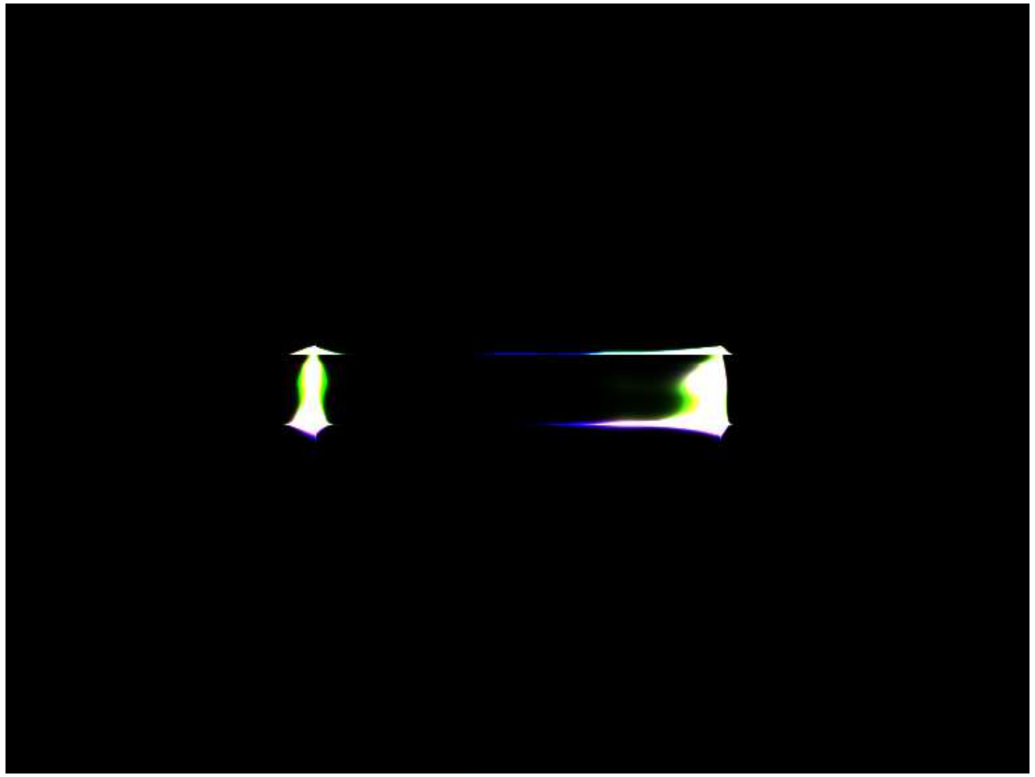
Intermediate Results of Bayes Inference: Set 3


Input Blurry Image


Output Latent Image

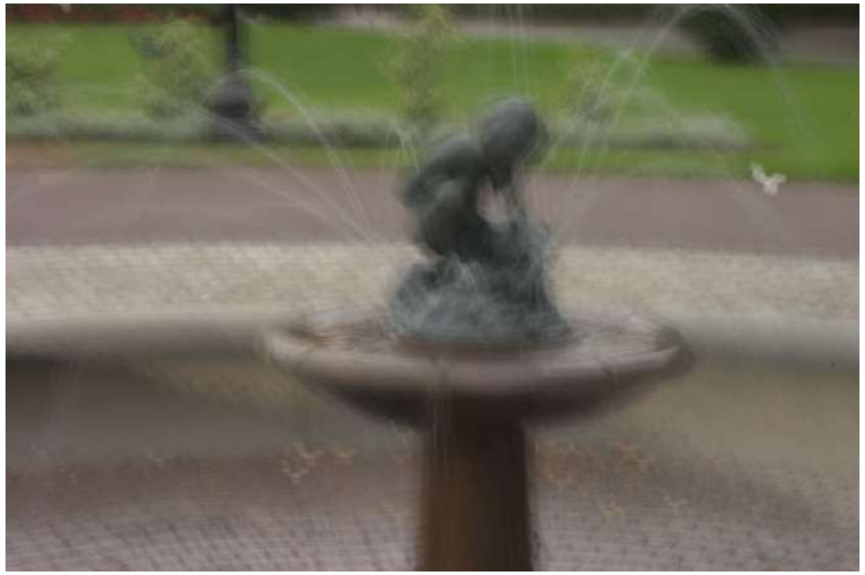
Intermediate Results of Bayes Inference: Set 4


Input Blurry Image

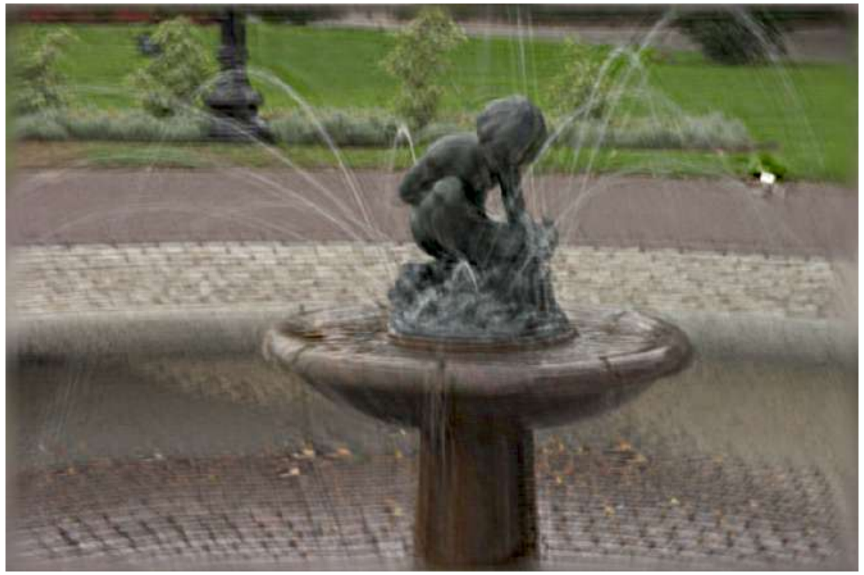

Output Latent Image

Comparison of Results from Author's Paper: Set 1

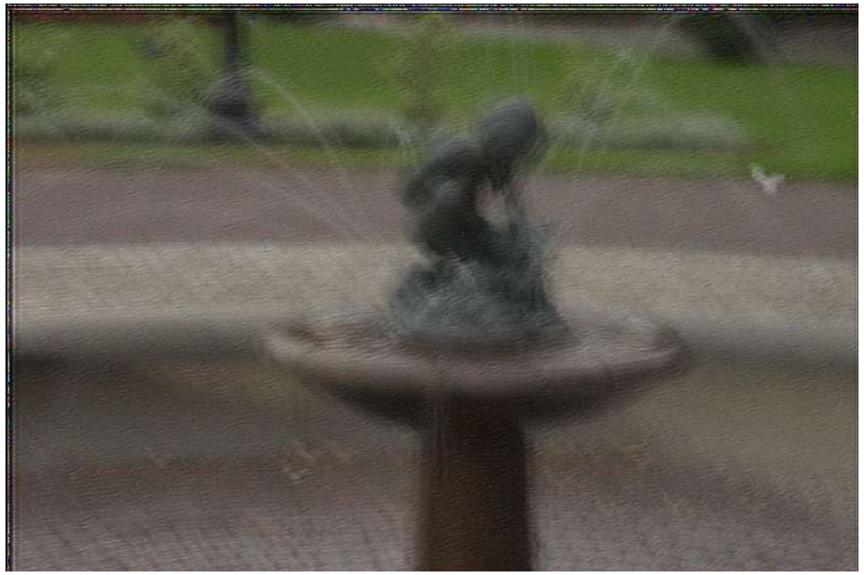
Input Blurry Image



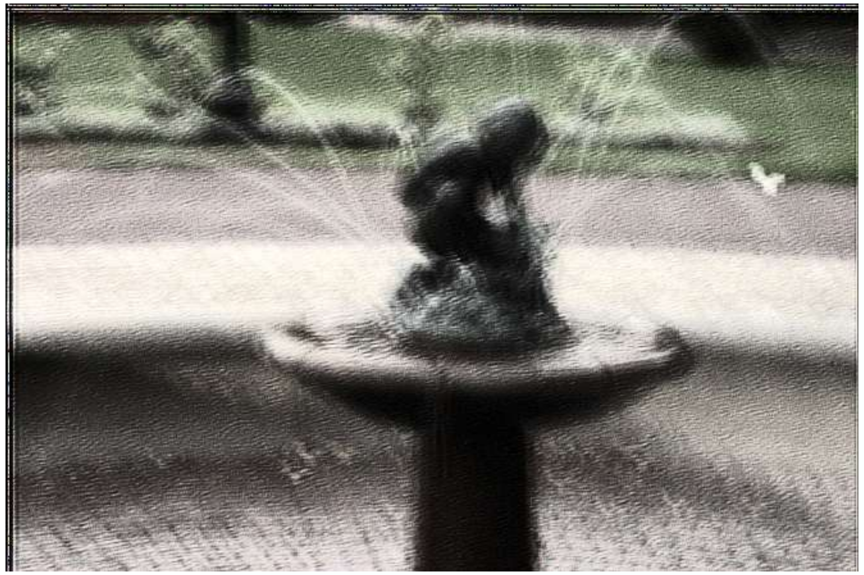
Output Latent Image (Paper)



Output Latent Image (Original)

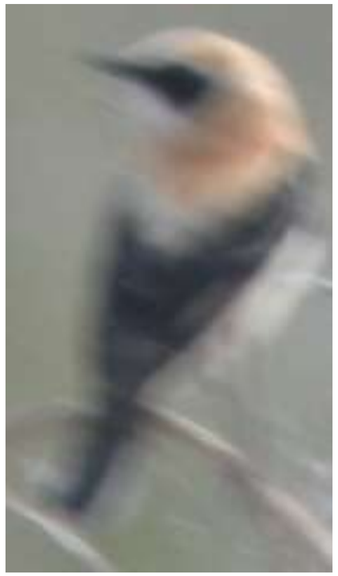


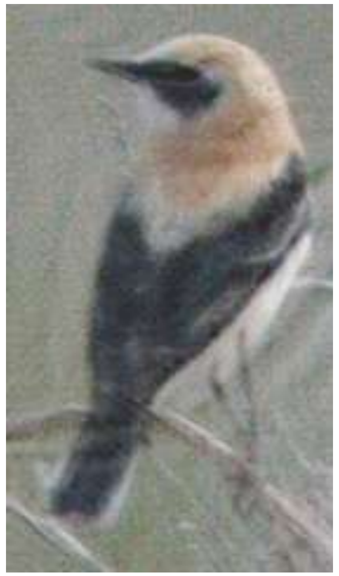
Post Processed Latent (Original)



Comparison of Results from Author’s Paper: Set 2

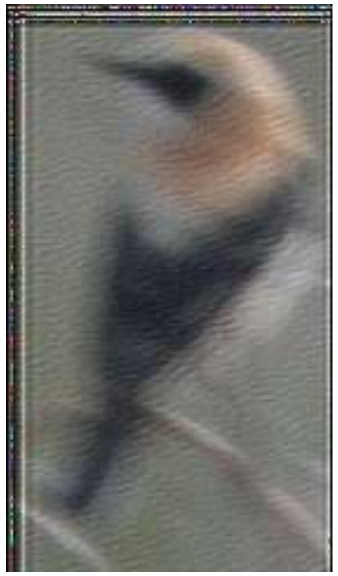
Input Blurry Image

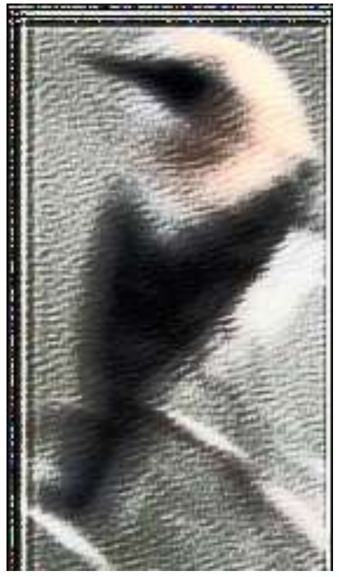




Output Latent Image (Paper)

Output Latent Image (Original)

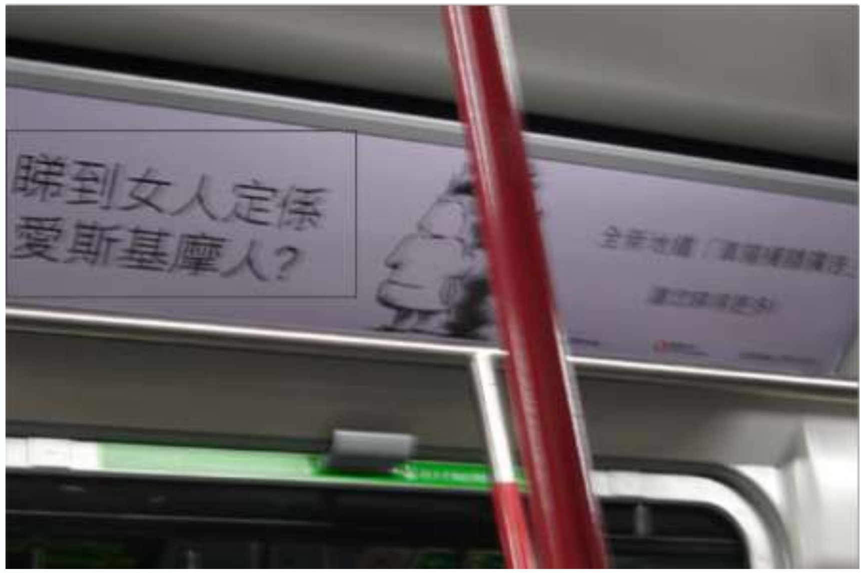


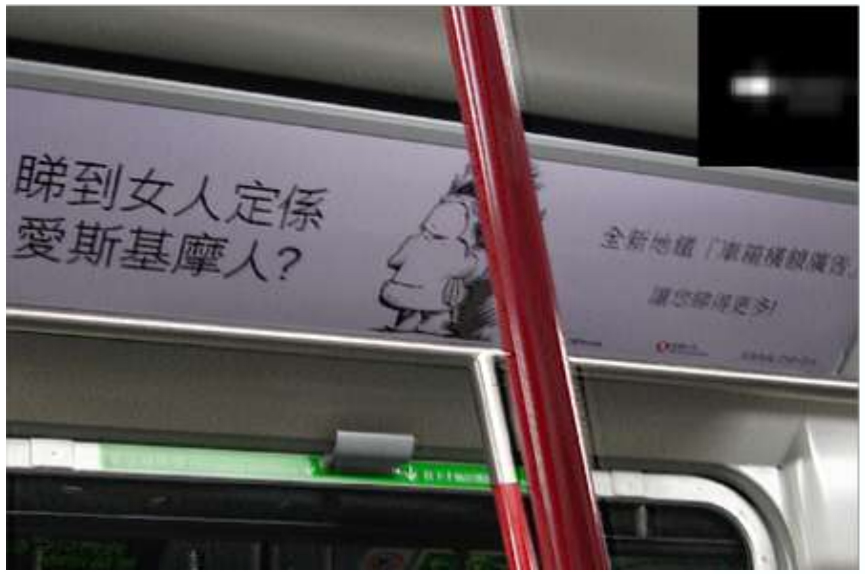


Post Processed Latent (Original)

Comparison of Results from Author’s Paper: Set 3

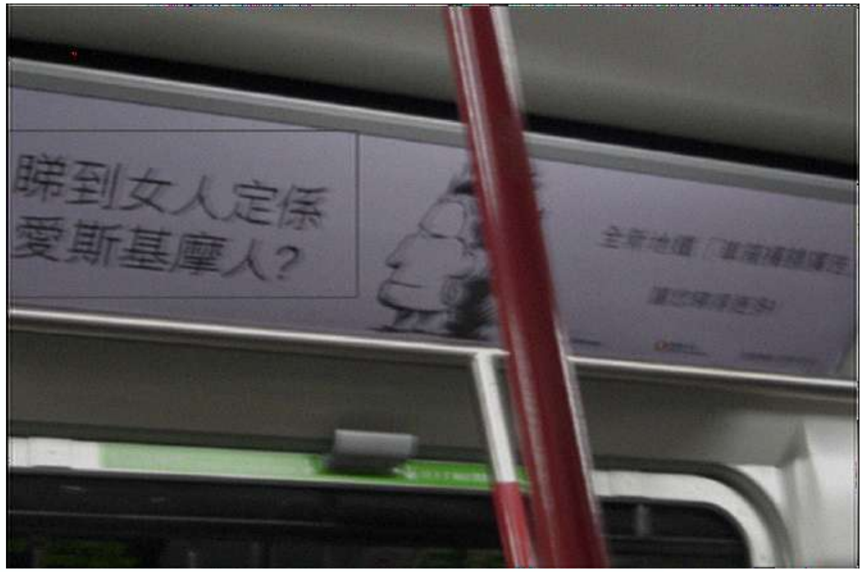
Input Blurry Image

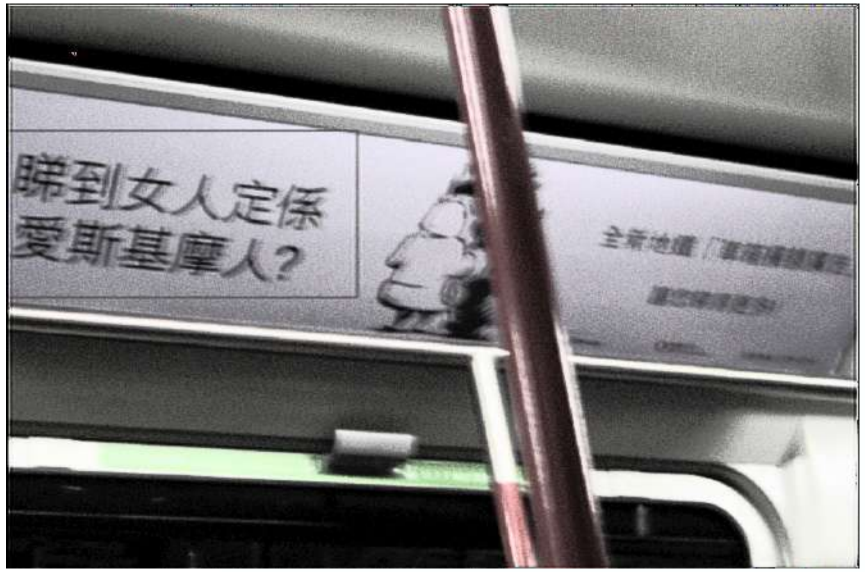




Output Latent Image
(Paper)

Output Latent Image
(Original)





Post Processed Latent
(Original)

Project Retrospective

This project offered me a great learning experience in the domains of computational photography and data science. Being able to leverage natural image statistics to enhance image quality such as motion blur, is an extremely relevant topic to everyday photographers. Although throughout the pipeline creation I did face some issues related to the Bayesian inference implementation using BayesPy, I learned about probability theory and statistics with application towards photography. This gave me a well-rounded understanding about how the material in this class can be applied to problems in other fields. If I had more time to complete the project, the main thing I would do differently is take a different approach to implementing the Bayesian Inferencing. Instead of using BayesPy, which I spent a lot of time on trying to understand by reading the package documentation and example on the internet, I would perhaps try to implement the Bayesian network from scratch. This would offer me a more intuitive understanding of the data science backend instead of trying to depend on a package that would handle the computation for me as a black box. I believe this is one of the main reasons for my failure of the implementation. There was not enough documentation available for the specific use case I was trying to implement and not enough examples. As a new user in data science and probability theory, it was difficult trying to learn all of this in this short amount of time.

Other than this issue, the rest of the pipeline was implemented successfully. Some improvements I could make are edge detection towards the output latent image and HDR enhancement. This would allow for a more detailed output where features can be more easily detected by the eye of the user.

Functional Code Retrospective

As explained in the Project Development section, some of the major libraries and packages that were used in this implementation include: Numpy, Scipy, OpenCV, Python built-in Math module, Glob, OS, and BayesPy. The BayesPy package is a new package added to the system that focuses on implementing a Bayesian network of nodes for variational Bayesian inference. It includes a variational Bayesian engine that will take in a constructed model and input data and then run for convergence on a certain threshold value. This package was preferably over other data science python package such as PyStan and PyMC3 mainly because it used variational Bayesian inferencing as the authors from the papers used, while the other papers did not. The authors of the paper also did not implement their own Bayesian network. They modified another existing research paper that implemented it in Matlab, which made it harder for me to understand why and how they went about the implementation process.

The main algorithms for this project include Bayesian Inferencing which stems from probability theory and Bayes Theorem, Multi-Scale Approach which utilizes a user input parameter to determining how many times a kernel is scaled up in resolution, and the Richardson-Lucy Deconvolution Algorithm, which takes an input point spread function and a blurry image, and returns a newly constructed sharp image.

Using these algorithms and packages, I was able to implement the pipeline described earlier.

Appendix: Your Code

Code Language: Python

List of code files:

- deblur.py

Prior File:

- **priors_author/linear_street_4.mat**

Credits or Thanks

The CS6475 staff for a wonderful semester!

Resources

- https://cs.nyu.edu/~fergus/papers/deblur_fergus.pdf : Author's paper that implementation is based on
- https://cs.nyu.edu/~fergus/research/deblur.html : Author's website that link to priors and source images
- https://www.eandbsoftware.org/how-bayesian-inference-works/ : Tutorial on Bayesian Inferencing
- https://www.youtube.com/watch?v=K_AMJQeVpVw : Huawei P10 - camera test – blurry video
- https://towardsdatascience.com/an-introduction-to-bayesian-inference-in-pystan-c27078e58d53 : Introduction to Bayesian Inference
- https://www.cs.unc.edu/~lazebnik/research/fall08/lec05_deblurring.pdf : UNC Lecture on Image Deblurring
- https://buildmedia.readthedocs.org/media/pdf/bayespy/latest/bayespy.pdf : BayesPy Documentation
- http://www.bayespy.org/user_guide/user_guide.html : BayesPy Official Website
- https://scikit-image.org/docs/dev/auto_examples/filters/plot_deconvolution.html : Scikit Image implementation on Deconvolution
- https://www.cambridgeincolour.com/tutorials/gamma-correction.htm : Gamma Correction Explanation