An Examination of “Image Deblurring Using Inertial Measurement Sensors”

Kevin O’Brien
Carnegie Mellon University
kobrien@andrew.cmu.edu

Abstract

For my final project, I set out to replicate the algorithm presented in “Image Deblurring Using Inertial Measurement Sensors” by Joshi et al [3]. I was initially attracted to this paper because it presents what feels like a very intuitive solution to the image deblurring problem: if we know the motion of the camera, we should be able to use that information to inform our deblurring optimization. I also saw this as a good opportunity to get a taste of IMUs, as this was my first time using this sensor modality. In this report, I show several intermediate and final results detailing my attempts to solve the camera shake correction problem. I will explain the work I did, some results I obtained, and the challenges I faced.

1. Introduction

Blurry images are a nuisance for anyone who has ever picked up a camera. Take as an example a photographer on vacation. She takes the perfect image of one of the hoodoos in Bryce Canyon National Park, but only when she gets back from her trip and closely examines her photos, she realizes that her image is blurry, and the definition on the jagged edges of the hoodoo is gone. She must have been weary from the hike when capturing her photo, and been shaking slightly while capturing the image. Now, her beautiful vacation photos are ruined, no longer Instagram-worthy.

The field of image deconvolution is an active research area that aims to solve problems of this nature. Essentially, deconvolution takes what we know about image filtering and blurring, and aims to reverse the process. To start, we will briefly discuss convolution, then the formulation of its inverse. The problem can be defined as follows: given a sharp image **I**, we can blur it with some blur kernel **K** to get a new blurry image, **B**. The blurring operation is performed using convolution, but we will see later that this can also be represented by a pure linear operation.

$$\mathbf{B} = \mathbf{K} * \mathbf{I} + N \tag{1}$$

where N is zero-mean noise that is independent of the image signal.

In practice, the kernel **K** is often a Gaussian filter or something similar that assigns to each pixel the local average of its neighboring pixels. Especially in the case of the Gaussian filter, this is usually used to de-noise the image, as taking the averages of pixel values will work to lessen the effect of noise in the image.

When we are just given a blurry image, without its associated sharp image, we can simply try to reverse this process to recover the initial sharp image:

$$\mathbf{I} = \mathbf{K}^{-1} * \mathbf{B} \tag{2}$$

Although seemingly trivial at a high level, this problem is not an easy one to solve in practice. Simply taking the inverse of a kernel and applying it to a blurry image will tend to produce very poor results. Most blur kernels tend to have many zero or close-to-zero values. So, inverting this will lead to divisions by zero, which will greatly amplify high-frequency noise in the signal, leaving us with an image that is unrecognizable from the expected result.

To remedy this, deconvolution algorithms introduce another term in the formulation of the sharp image, converting the problem from closed-form to an optimization. The minimization we want to enforce is that the difference between the sharp image convolved with the blur kernel and the blurry image is close to zero, while also ensuring that the gradients in the resulting sharp image are small. This small gradient assumption derives from the fact that natural images tend to be mostly smooth, so we want our solution to mimic that constraint. This formulation can be written as follows:

$$\min_{\mathbf{I}} \|\mathbf{B} - \mathbf{K} * \mathbf{I}\|^2 + \|\nabla \mathbf{I}\|^n \tag{3}$$

where n defines which L-norm we want to minimize the gradients over.

The above equation actually only solves part of the problem. It assumes that we already know the blur kernel to deconvolve with. However, in the case of camera shake removal, we do not know this blur kernel. So, at a high level, this now becomes a joint optimization over two variables, **I** as well as the unknown blur kernel **B**.

2. Theory from the Paper

2.1. Blur Function Formulation

The first thing to note here is that in the introduction, all of the ideas and equations presented dealt with the concept of spatially-invariant blur (ie, one blur kernel applied to the entire image). However, due to the fact that the depth and rotation of the captured image changes with camera motion, this paper's novel contribution is deriving a spatially-varying blur function.

To formulate the blur function, the authors present the following idea. Suppose we are capturing an image over some timespan t=[0,T], and the camera is moving during that time. At any time t, we know that the expression for a 3D scene point projected to a 2D pixel point is:

$$[u, v, 1]^T = \mathbf{P}[X, Y, Z, 1]^T \tag{4}$$

where u, v define 2D pixel coordinates, X, Y, Z define 3D world coordinates, and **P** is the 3x4 camera projection matrix that relates these two quantities. **P** can be further decomposed into the camera intrinsics **K** and the extrinsics $\mathbf{E}_t$. The extrinsic matrix consists of 6 degrees of freedom (3 rotation, 3 translation) describing the camera's position relative to some canonical frame. If we consider the camera's position at t=0 to be the canonical frame, we can clearly infer that the extrinsic matrix will be a function of time, as the camera's position with respect to that first frame will change during camera shake.

This change in camera extrinsic parameters at each timestep manifests itself as different sets of projected pixels onto the image plane. So, at each time t, we will have a slightly different distribution of scene points on our image plane depending on where in the world the camera was at that point. Effectively, we're trying to use the image points to reason about the camera motion. Luckily, we are essentially just relating T (number of timesteps) planes to each other, which is easily doable using homographies. We can parametrize a homography at a certain time t as a function of scene depth as follows:

$$H_t(d) = \mathbf{K}(R_t + \frac{1}{d}T_t N^T)\mathbf{K}^{-1} \tag{5}$$

where d is scene depth, R_t and T_t are the rotation and translation from the canonical frame, respectively, and N is the normal vector orthogonal to the image plane. Using this homography, we can get the pixel coordinates u,v at time t as follows:

$$[u_t, v_t, 1]^T = H_t(d)[u_0, v_0, 1]^T \tag{6}$$

Using this homography, we can then obtain the pixel values at those points u_t,v_t using the equation

$$\mathbf{I}_t(u_t, v_t) = \mathbf{I}(H_t(d)[u_0, v_0, 1]^T) \tag{7}$$

which can be reformulated as

$$I_t = \mathbf{A}_t(d)I_0 \tag{8}$$

In the above formulation, I_t and I_0 are vectorized versions of the images at time t and time 0, respectively. The matrix $\mathbf{A}_t(d)$ encodes the transformation due to homography as a NxN matrix, where N is the number of pixels in an image (height x width). $\mathbf{A}_t$ is a sparse matrix with four nonzero entries per row, with the four nonzero elements being the bilinear interpolation weights associated with the neighboring pixels. For example, if a warped pixel coordinate for some arbitrary (u_0,v_0), defined by Equation (6), comes out to be (13.6,17.9), then the four nonzero entries in the (u_0,v_0) row of the matrix would be in the columns for (13,17), (13,18), (14,17), and (14,18).

Once we have this formulation for the blur matrix at time t, we can simply take the sum of all of these blur matrices over the exposure time to formulate the final blur function, and subsequently, the final blurry image.

$$\mathbf{A}(d) = \int \mathbf{A}_t(d)dt \tag{9}$$

$$B = \mathbf{A}(d)I_0 + N \tag{10}$$

So we now have the ability to define a blur matrix **A** given the known spatial transformations between successive sensor samples during the capture process. Now that we know the blur function, this becomes a non-blind deconvolution problem. We aim to optimize over the following function:

$$\min_I \|B - \mathbf{A}(d)I\|^2 + \lambda \|\nabla I\|^{0.8} \tag{11}$$

The authors used the iteratively reweighted least squares algorithm to solve this equation, but as I will discuss, I tried a few different methods.

2.2. Extracting Transformations from Sensor Data

The authors describe in detail the process for extracting linear and angular positions from an accelerometer and a gyroscope mounted on a camera. I will sum up their equations and explanation in a few steps. This whole process can be found on page 3 of the paper.

1. Given gyroscope readings (rad/s), integrate once to get angular positions

2. Use these angular positions to create a list of rotation matrices, transforming sensor readings at each time t back to the frame at time 0

3. Rotate each of the accelerometer readings (m/s^2) back into the canonical frame (frame at t=0) using the corresponding rotation matrix from Step 2

4. Integrate these rotated values twice to get linear positions

Armed with these two sections, we have enough information to solve for the resulting sharp image using sensor data. The last piece of the paper's methodology describes an outer loop to optimize for drift in the raw sensor readings, but due to my issues in obtaining accurate linear positions from the raw sensors (which I will discuss later), I decided to omit that part from my implementation, instead opting for using the ground-truth camera motion provided by the dataset.

3. Implementation

To implement this paper, I used the ETH3D dataset [1], an open source dataset meant for SLAM benchmark testing. I chose this dataset because it has well-organized IMU data along with RGB images. They even have a few datasets of the camera shaking, but I came to realize that their definition of "shaking" was too much for this algorithm to handle, as I will discuss later on. I tested my algorithm using images and IMU data from the "camera shake 1" and "plant 5" datasets. In both datasets, there is about 1 image per 0.03 seconds, one IMU sample per 0.005 seconds (6-7 per image), and one ground-truth position sample every 0.01 seconds (3-4 per image).

3.1. Sensor Data Processing

To start, I created a data loading script that wrapped up each image with its associated sensor data based on the timestamps of the sensor readings and the images. Using this data, for each image I carried out the algorithm laid out in the previous section to recover angular and linear positions. I was able to do a good job of recovering angular positions fairly accurately; however after following the procedure for using these angular positions to recover linear positions, I must have some bug or misunderstanding in my implementation. I have a sneaking suspicion that I have some reference frame discrepancy in my code, but after debugging for a few days, I was unable to find the bug. Qualitatively, it looks to be a scale issue, as the shapes of the curves look similar enough, but the scales look off, although not by the same factor along each axis (5x discrepancy in x position but 10x discrepancy in z position). However, I know it could possibly be a reference frame transform issue as well, as I spent a lot of time debugging that type of error in order to get the angular positions to line up.

Because of this incompleteness in the sensor processing, I was unable to justify the use of the outer-loop optimization discussed in Section 3.4 of the paper, which is meant

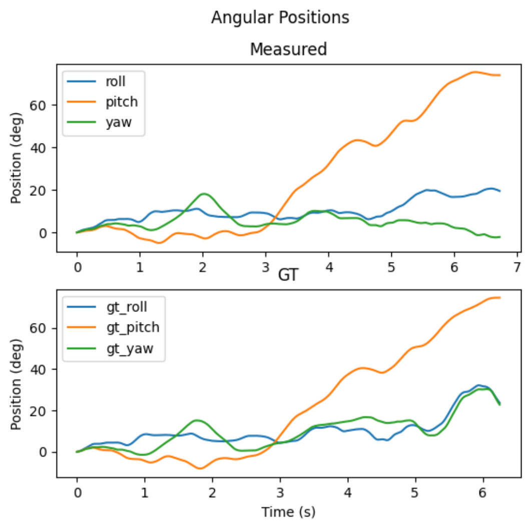

Figure 1. Angular positions derived from gyroscope readings compared to ground truth angular positions

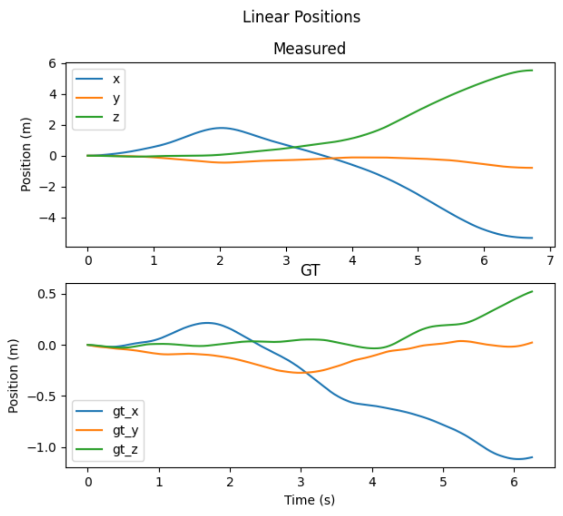

Figure 2. Linear positions derived from accelerometer readings compared to ground truth linear positions

to counteract sensor drift. Instead, from here on I use the ground truth position measurements provided by the dataset to infer camera motion, so it is safe to assume that these measurements do not require drift correction.

In Figures 1 and 2, I show the calculated relative rotations and translations from the canonical frame. For easier visibility for the sake of this report, I show the data for the entire length of the dataset. In practice, I take the sensor data associated with the given frame and rotate every sample into the frame of the first sample for the given frame, as specified in the paper.

3.2. Blur Function Formulation

Using the ground truth motion provided in the dataset, I follow a similar procedure as was outlined in Section 2.2. After loading all the ground truth data for each image's exposure time window, I loop through each data sample and performed the following steps.

1. Sample the relative angular position, and convert that into a homography using Equation 5. Here, I assume N to be [0,0,1] (*i.e.*, pointing out of the camera's principal point, perpendicular to the image plane), and I assume a constant depth of 1. The outer loop optimization in the paper implicitly solves for this depth value, but I found that picking a constant worked fairly well on the data I used.

2. Using this homography, warp every coordinate in the image to the new plane coordinates defined by the homography. Using these new coordinates, populate the sparse **A** matrix with the four weights used for bilinear interpolation of neighboring pixel values.

3. Loop through Steps 1 and 2 for each sensor reading, adding in the newest bilinear interpolation weights to the existing **A** matrix.

4. Divide every element in **A** by the number of total samples, so that the sum of each row is 1.

At this point, I slightly deviated from the method described in the paper given the limitations in optimization code I had accessible to me. My math skills are admittedly limited in this area, and while I understood the optimization at a high level, I did not have the time to implement it myself, so I had to depend on third party code to perform the optimization. In what I was able to find, most deblurring optimization code takes as input a single spatially-invariant blur kernel instead of the **A** matrix that the paper describes.

On Yannis's suggestion, I went looking for an implementation of the Alternating Direction Method of Multipliers algorithm, which solves the deconvolution problem while enforcing an L-1 norm minimization on the resulting sharp image. While not exactly the same as the paper, I figured this would be a good approximation of the solution. To implement this algorithm, I used the existing "Plug-and-Play-ADMM" MATLAB codebase provided by [2], and re-implemented it in Python. Their algorithm takes in a blurry image and spatially-invariant blur kernel, and solves for the resulting sharp image. To obtain a blur kernel, I arbitrarily pick some index in the image and, using a square kernel with size 35x35, I create the kernel using the corresponding values in the **A** matrix.

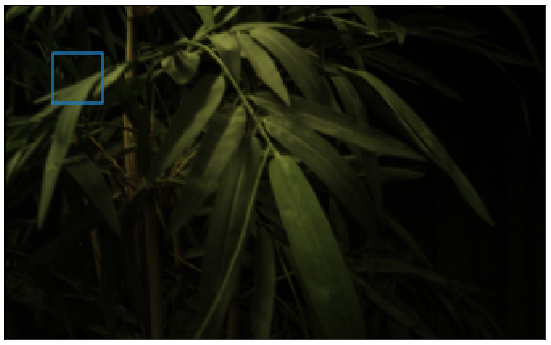
Figure 3. (Example 1) Original image. The blue box shows the 35x35 region of the image for which I take the blur kernel to feed into the ADMM optimizer

4. Results

4.1. Plant Dataset

Here I discuss some results of the deblurring procedure on the plant image set from the dataset. This dataset shows a camera being slowly walked around a plant. The small inter-frame motion is key here, as the formulation of blur laid out in the paper only holds for small motions.

4.1.1 Example 1

Figure 3 shows the original blurry image, along with the box indicating which region the blur kernel (Figure 5) is formulated around. In this case, I build the kernel around the pixel x=50, y=50. Between this frame and the previous frame, there is only a small translation, and not much rotation, so it follows that the blur kernels across the image (Figure 4) are mostly similar. The output (Figure 6) shows a very successful deblurring result, since the original kernel location, as well as other parts in the image, look distinctly sharper. The oblique stem that runs downward along the middle of the image is an especially good indicator of the deblurring result, as it appears much clearer and its edges are sharper. As a quantitative measure, the peak signal to noise ratio (PSNR) between the deblurred result and the original is 27.32 dB, which is a very strong result.

4.1.2 Example 2

For the second example, I chose a frame in the dataset with a little bit more motion between frames. Figure 7 shows the original frame, as well as the box highlighting the pixel (115,110) that I used to build the blur kernel (shown in Figure 9). This clearly is a larger motion, as the effects from more pixels farther from the original are incorporated into the pixel's final value. We can also see in Figure 8 that the

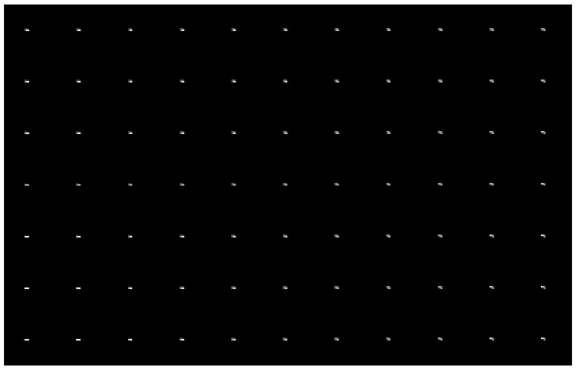
Figure 4. (Example 1) Spatially-varying blur kernels generated by scene motion in Figure 3

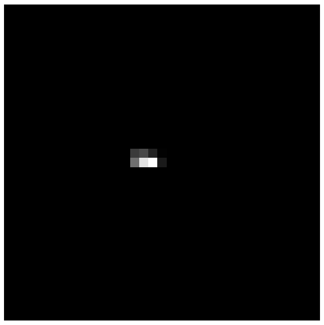
Figure 5. (Example 1) Kernel used for ADMM deblurring optimization, built around pixel coordinate (50,50)

blur varies much more over the scene. Given this blur kernel at (115,110), I was able to obtain the results shown in Figure 10. Here, the edges are definitely sharper (especially on the edges of leaves near the kernel location, and the little yellow hairs coming from the stem), but there are some strange color artifacts that do not look very good. Still, this did denoise the image reasonably, as there was still a PSNR of 20.86 dB.

To try and see if I could improve the results on this image, I built a blur kernel around the pixel (180,130), on the central yellow stem, shown in Figure 11. The kernel generated at this location is shown in Figure 12. In this new blur kernel, there is a much less even distribution of neighboring pixel values, as shown by the one noticeably brighter pixel in the kernel. As we can see in Figure 13, this kernel does a much better job of approximating the scene blur, as the stem is much clearer and there are fewer strong color artifacts. Additionally, the qualitative observations are backed up by the quantitative observation that the PSNR here has increased to 21.32 dB.

The results shown on this image show that the deblurring optimization is definitely spatially-varying, so my method is not a reliable way to deblur images, since I have to manually try different kernel locations to approximate which one seems to work the best over the whole image. This simply proves the authors' point: blur in a moving scene is absolutely dependent on the location in the image.

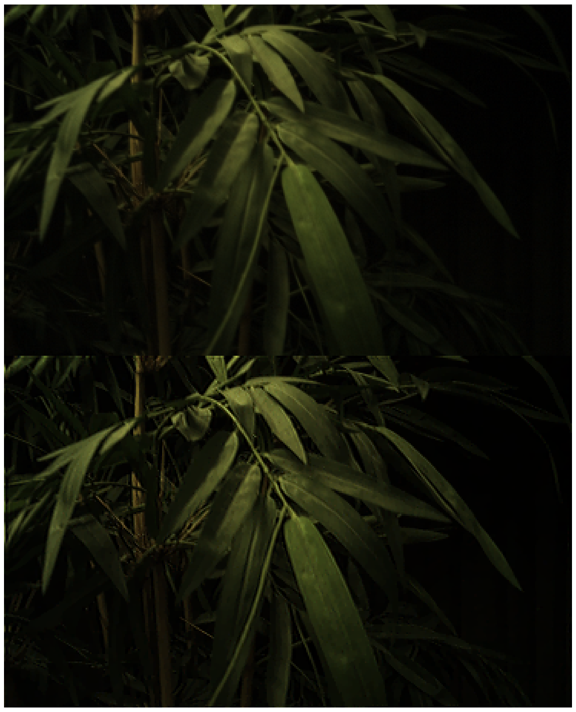
Figure 6. (Example 1) Initial blurry image (top) compared with final deblurred output (bottom). PSNR = 27.32 dB

4.2. Camera Shake Dataset

The results shown on the plant dataset, though presented first, were the last things that I tested. For the majority of my development process, I was using the camera shake dataset. However, the dataset curators' definition of camera shake and the computational photography literature's definition differ greatly, and I feel silly for not realizing this earlier. The camera shake dataset is more meant to stress-test localization systems than image deblurring, so the camera is shaken wildly during capture. Because of such large motions, it leads to the generated blur kernels being sparse and non-continuous, producing poor results. Here I'll show one example of my output on an image from the camera shake dataset.

4.2.1 Example 3

The original image is shown in Figure 14, with the area around pixel (50,40) shown in the box. The spatially varying kernels are shown in Figure 15, and the kernel built around (50,40) is shown in Figure 16. The final output is shown in Figure 17. Clearly we can see that these results are poor, but intuitively it makes a lot of sense. The camera

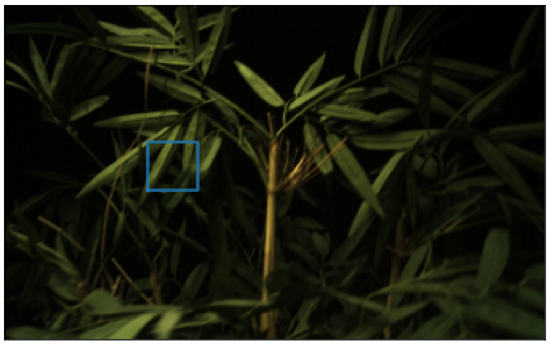

Figure 7. (Example 2.1) Original image. The blue box shows the region of the image for which I take the blur kernel to feed into the ADMM optimizer

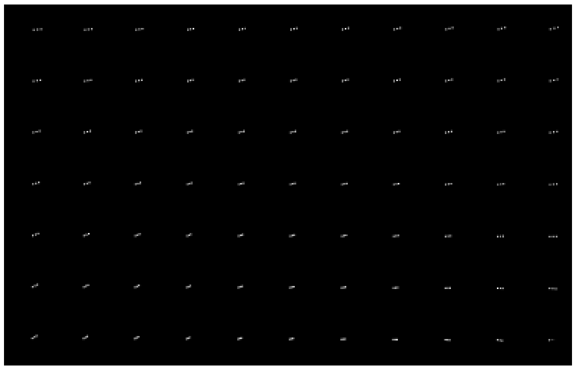

Figure 8. (Example 2.1) Spatially-varying blur kernels generated by scene motion in Figure 7

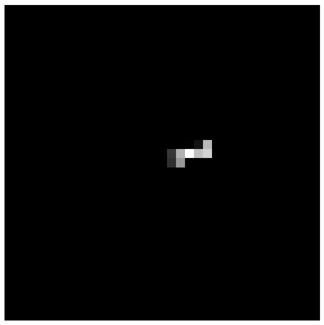

Figure 9. (Example 2.1) Kernel used for ADMM deblurring optimization, built around pixel coordinate (115,110)

is moving too fast for the sensors to keep up, so we are left with discontinuous blur kernels. This causes a lot of strong artifacts in the resulting image, including a large black rectangle inside the bottom left border, which is a product of the blank space between nonzero blur kernel entries. Interestingly there is still some positive denoising outcome, as the resulting image has a PSNR of 13.42 dB, but I wouldn't

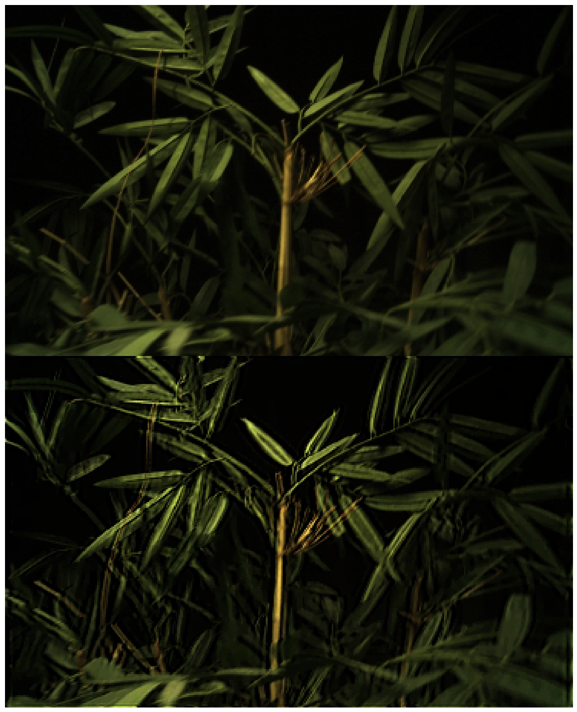

Figure 10. (Example 2.1) Initial blurry image (top) compared with final deblurred output (bottom). PSNR = 20.87 dB

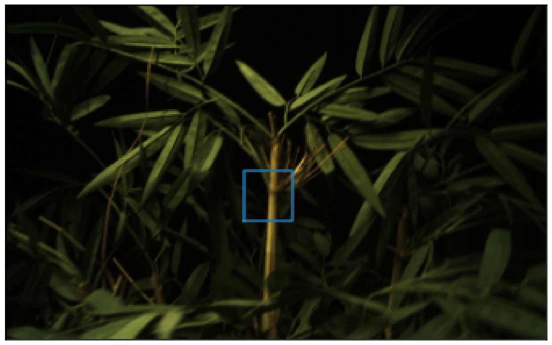

Figure 11. (Example 2.2) Original image. The blue box shows the region of the image for which I take the blur kernel to feed into the ADMM optimizer

really consider that a valid metric, since so many pixels in the resulting image are fully black where there was color in the original.

5. Discussion

5.1. Other Optimization Constraints Attempted

While experimenting with the results of the "Plug-and-Play ADMM" in the early stages of development, I was not satisfied with the results, so I tried to pursue methods that actually used the 0.8-norm optimization that the paper uses.

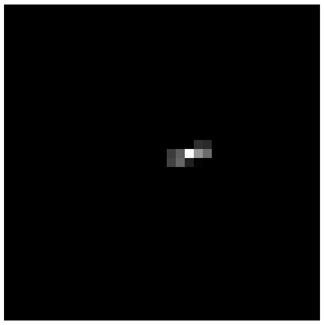

Figure 12. (Example 2.2) Kernel used for ADMM deblurring optimization, built around pixel coordinate (180,130)

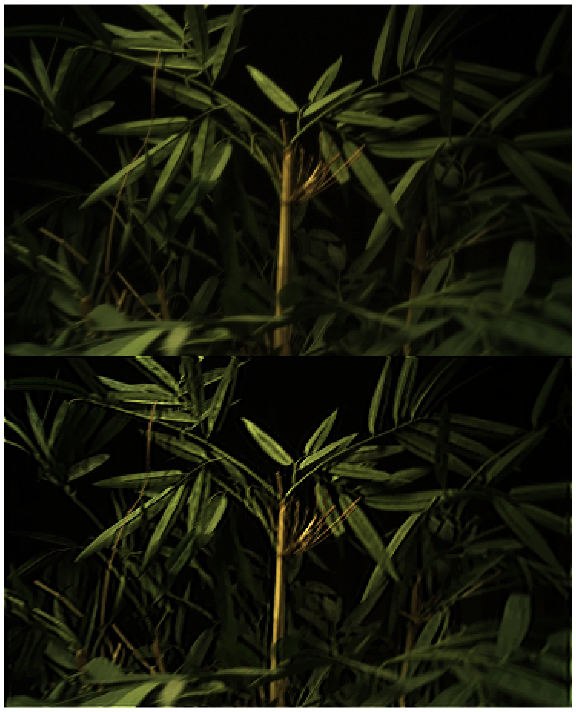

Figure 13. (Example 2.2) Initial blurry image (top) compared with final deblurred output (bottom). PSNR = 21.32 dB

I found a good explanation of the correct sparse-gradient deconvolution algorithm in [4] that seemed, on paper, to handle the spatially-invariant blur matrix instead of a single blur kernel, but upon looking at their implementation, it seemed like they also required a single kernel as input to the deblurring optimization. However, the kernel that they use in their demo was a coded aperture, which by its nature would be spatially-invariant, as that coding would be applied to every pixel in the blurry image. I did try their MATLAB implementation using one of my images and got really horrible results (white images with some black streaks, but nothing like an image), so I decided to stick with my L-1 ADMM implementation.

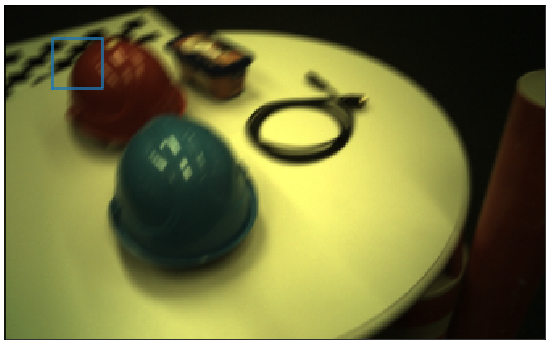

Figure 14. (Example 3) Original image. The blue box shows the region of the image for which I take the blur kernel to feed into the ADMM optimizer

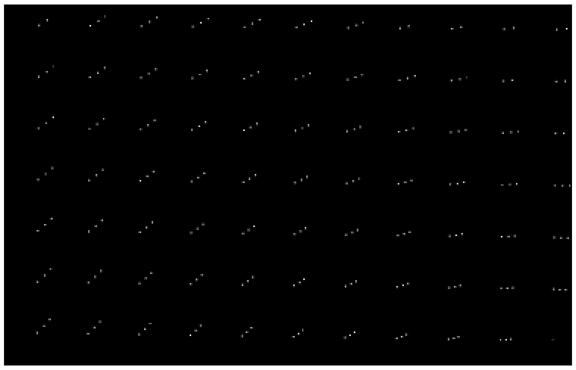

Figure 15. (Example 3) Spatially-varying blur kernels generated by scene motion in Figure 14

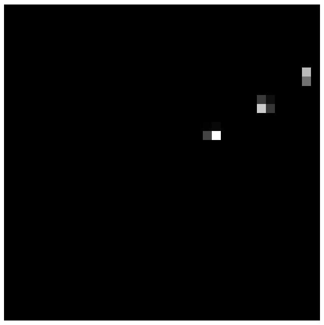

Figure 16. (Example 3) Kernel used for ADMM deblurring optimization, built around pixel coordinate (50,40)

I also tried translating the L2-norm implementation from the “Plug-and-Play ADMM” MATLAB codebase, but experienced some bugs.

5.1.1 Future Work

To better match the results in the original paper, I would have to figure out a good way to incorporate the entire

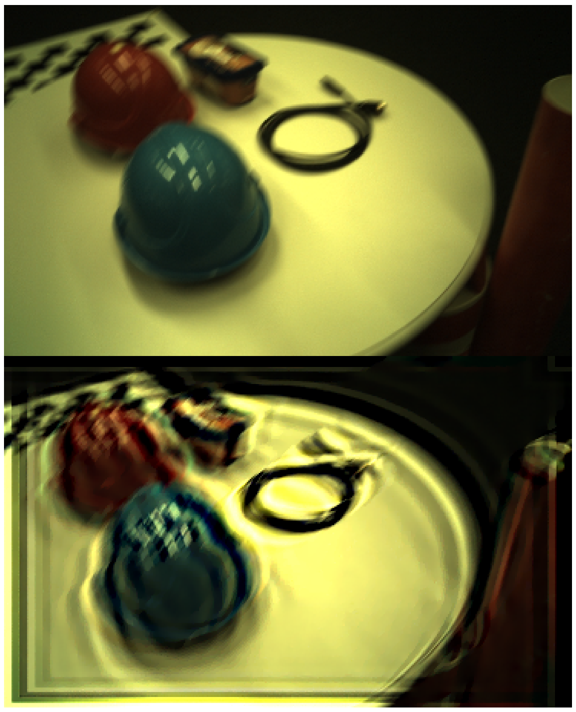

Figure 17. (Example 3) Initial blurry image (top) compared with final deblurred output (bottom). PSNR = 13.42 dB

spatially-varying blur matrix into the deblurring optimization, not just one arbitrarily located blur kernel. To do this, I would have to spend more time studying the math and implementation of this algorithm without using the Fourier domain, as I am unsure about the feasibility of taking the Fourier transform of a the large, sparse blur matrix.

Additionally, to fully match the results of the paper, I would have to fix whatever bug I am experiencing in the accelerometer processing, so that the data more closely matches the ground truth. This would then allow for the outer loop optimization to minimize sensor drift, thus fully replicating the method described in the paper.

6. Conclusion

In this report, I have presented my efforts and discoveries in exploring the paper “Image Deblurring Using Inertial Measurement Sensors.” I have implemented the majority of the techniques described in the paper, and have learned a great deal along the way.

I have uploaded my code for this project to

https://github.com/kob51/image_deblurring

if you’d like to take a look!

7. Acknowledgements

I wanted to thank you, Yannis, for all your help and support during this project. Our office hour sessions really helped solidify my understanding, and I’d certainly not have been able to show results without your help. Thanks again and have a great holiday!

References

[1] ETH3D SLAM Datasets. https://www.eth3d.net/slam_datasets. 3

[2] Stanley H. Chan, Xiran Wang, and Omar A. Elgendy. Plug-and-play ADMM for image restoration: Fixed point convergence and applications. *CoRR*, abs/1605.01710, 2016. 4

[3] Neel Joshi, Sing Bing Kang, C. Zitnick, and Richard Szeliski. Image deblurring using inertial measurement sensors. *ACM Trans. Graph.*, 29, 07 2010. 1

[4] Anat Levin, Rob Fergus, Fr Durand, and William Freeman. Deconvolution using natural image priors. *ACM Trans Graphic*, 26, 01 2007. 7