IMU-Aided Deblurring With Smartphone

Jason Xu
ECE
jiachenx@andrew.cmu.edu

Sanjay Salem
BCSA
svsalem@andrew.cmu.edu

Abstract

This paper presents an implementation of a modernized version of the deblurring algorithm using IMU data given in [4]. We use a combination of smartphone cameras and smartphone MARG (Magnetic, Angular Rate, and Gravity) sensor data to estimate a blur model from the camera rotation and translation, across the duration of an exposure. This blur model is then used to solve an optimization, to deconvolve the blurred image and remove the effects of camera shake.

1. Introduction

Figure 1. Example of a nighttime cityscape we want to accurately capture.

When attempting to take images of nighttime cityscapes (such as figure 1) with a smartphone camera, it is difficult to capture a well-lit image due to the overall dimness of the scene. The ambient and direct lighting created by the sun during the daytime is not present, so the only light sources available are the very ones trying to be captured. There are three main ways to counter this issue photographically and ultimately receive a decently lit image, but each method has its own drawbacks. The following methods and drawbacks refer to figure 2.

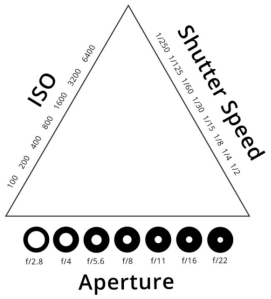

Figure 2. Exposure triangle diagram.

Increasing the ISO will brighten the image, but at the cost of additional noise in the capture, which is not ideal. Decreasing the aperture size of the camera is another option, but most modern smartphones and smaller cameras have a set aperture size which cannot be changed, so this method is nearly impossible. Finally, decreasing the shutter speed will result in a longer-exposed image, but may be blurry due to hand movement or camera shake.

Out of these three issues, the simplest issue to solve computationally is removing blur caused by camera shake during an exposure, since it is deterministic, unlike noise.

There are currently various methods of removing blur using deconvolution, but they are suboptimal. The naïve approach is to use blind deconvolution, which employs guessing a blur kernel, and solving a least squares optimization to perform the deconvolution. Due to the nature of guessing a blur kernel beforehand, many other prior assumptions need to be made, and the method will result in a wide range of possible solutions for the loss function. Simply picking one of these solutions or taking the average will not give a very precise or accurate result, and thus will output a subpar sharpened image.

Our improved method builds on the algorithm described in [4] and in figure 2 to perform a more precise deconvolution. At a high level, we use an MARG sensor, incorporating an IMU plus an additional magnetometer, to record camera movement data, and then use the derived camera rotation matrix and translation vector, coupled with camera

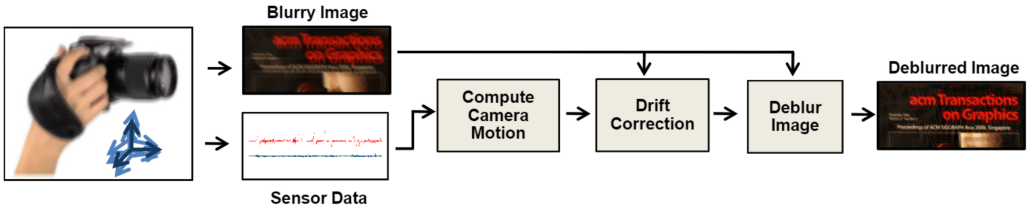

Figure 3. Our image deblurring algorithm, as a flowchart.

intrinsics, to compute a blur kernel. Finally, this blur kernel is used to deconvolve the blurry image.

1.1. Improvements

One of the improvements we make on the original algorithm was to use a smartphone for our setup and testing, since the components required to capture and compute data are already integrated, rather than having to building a more complex capturing rig. Because of this prior integration, there is less setup required in terms of calibrating the different components to work together and sending data between them.

Our second improvement ties in with the previous paragraph, in that due to the advancement of smartphone technology, we now have more integrated components to draw our data from. The most significant of these is the magnetometer, which allows us to increase our degrees of freedom from 6 (in the original paper's setup) to 9, providing a more precise computation for camera movement data. We can now use advanced algorithms incorporating all 3 types of data, typically used in robotic pose estimation, to calculate the camera motion, providing much better accuracy over the paper's method.

1.2. Data Collection

We use an Android app on the phone to log gyroscope and accelerometer data [3], modified to also log magnetometer data, which is synchronized to the start of the exposure window. With our smartphone, the lowest shutter speed available to set is 240ms, which is what we use to capture each exposure. Since exposures are essentially captured as video frames, we are able to parse each frame individually and use its data for computation.

2. Deblurring With Camera Data

To first obtain our camera rotation and translation data, we can run some operations on the IMU data. To obtain the translation vector for the camera at some timestep t, we perform a double integral over the linear acceleration recorded by the accelerometer. Rotation is a bit more computationally involved - we improve over the original method of integration by using a Madgwick filter [1], which performs an optimized gradient descent algorithm with quaternion derivatives using gyro, accelerometer, and magnetometer data to get the attitude of the camera much more accurately.

2.1. Forming the Blur Kernel and Warping Matrix

The typical formula for forming a blurred image with a noise factor and a blur kernel is given as:

$$B = I \otimes K + N, \tag{1}$$

where K is the blur kernel, and $N \sim \mathcal{N}(0, \sigma^2)$ is the noise. The original paper states that there are many properties of a camera that can lead to spatially-varying blur, with the examples being (1) depth dependent defocus blur, (2) defocus blur due to focal length variation over the image plane, (3) depth dependent blur due to camera translation, (4) camera roll motion, and (5) camera yaw and pitch motion during strong perspective effects. We handle only camera-induced motion blur, which includes the latter 3 properties. We additionally make the assumption that noise will not be as big of an issue, so our equation is simplified to

$$B = I \otimes K. \tag{2}$$

We then want to compute the blur matrix A_t for each timestep t in our exposed image, which is proportional to a planar homography H_t. This relation is modeled as follows:

$$A_t \propto H_t(d) = K(R_t + \frac{1}{d} T_t N^T) K^{-1}, \tag{3}$$

where R_t is the camera rotation matrix at timestep t, T_t is the camera translation vector at timestep t, and N is the unit normal of the camera direction. We made the assumption that our focal distance would be fixed, and decided to set $d = 4.5$ as a constant.

To compute A_t from our homography matrix, we first use the following:

$$(u_t, v_t, 1)^T = H_t(d)(u_0, v_0, 1)^T \tag{4}$$

$$(u_0, v_0, 1)^T = H_t(d)^{-1}(u_t, v_t, 1)^T, \tag{5}$$

where $(u_0, v_0, 1)$ is the initial pixel UV coordinate, $H_t(d)$ is our homography matrix, and $(u_t, v_t, 1)$ is our warped pixel

UV coordinate. To get the value of the original coordinate within our sparse matrix A_t, we perform bilinear interpolation on the 4 nearest pixel values and update the weights being accumulated.

2.2. Recovering the deblurred image

After computing A_t, we can use it to compute our column-vectorized blur model:

$$\vec{I_t} = A_t \vec{I} \tag{6}$$

$$\vec{B} = \sum_{t=0}^{s} A_t \vec{I} \tag{7}$$

Where $\vec{I_t}$ is the column-vectorized blurred image at timestep t formed by applying A_t to the original image $\vec{I}$, and $\vec{B}$ is the column-vectorized blur model formed from taking the sum of the blur matrices for each timestep and applying it to the original image. Once this blur model is obtained, we can perform deconvolution to get the deblurred image by solving the following regularized least-squares optimization:

$$I = \arg\min_{I} ||B - \sum_{t=0}^{s} A_t I||^2/\sigma^2 + \lambda|\nabla I|^{0.8} \tag{8}$$

3. Implementation

This project is implemented using our Android logger to record data and transfer to computer, and Python for the actual deblurring algorithm. Additional libraries include `numpy`, `scipy`, and `ahrs`.

3.1. Setup and Data Importing

We first write a Python class `IMUFrame` to create a frame object to use for integration. The data stored within in instance of this object includes the start and end of the exposure duration, the timestamp, and the acceleration, gyro, and magnetometer data at that timestamp. Gyroscope, accelerometer, and magnetometer data are stored as `numpy` arrays to allow for easy usage of the `numpy` library. Within the class initialization, we also compute the duration between each sensor reading, `dt`, (for later input to the Madgwick filter).

3.2. Data Standardization

We must now remove the effect of gravity from the accelerometers. To do this, we assume that the measured acceleration on the y-axis, pointed towards the bottom of the camera, normally distributed about the constant force of gravity:

$$g_y = \frac{1}{T+1} \sum_{t=0}^{T} a_y(t) \tag{9}$$

Therefore, we can offset the acceleration on the y-axis by subtracting offset gravity from each value:

$$a'_y = a_y - g_y \tag{10}$$

We now assume that the initial pose only had a rotation along the x-axis, which means we must also remove the effect of gravity on the orthogonal z-axis. To calculate this rotation, we find the angle between the acceleration of the y-axis, pointing towards the bottom of the camera, and the constant gravitational acceleration:

$$\theta = \cos^{-1}\left(\frac{a'_y}{g}\right) \tag{11}$$

and use trigonometry to derive the component of gravity affecting the z-axis and subtract it from the acceleration values:

$$a'_z = a_z - g\sin(\theta) \tag{12}$$

We also use the calculated rotation angle relative to gravity to set our initial orientation for integration. We later removed this as it was affecting our image coordinate results, which is discussed in more detail within subsection 4.2.

3.3. Camera Extrinsics

Once the data has been properly formatted, we are able to begin calculating the camera rotation matrix and translation vectors. As stated before, the translation was fairly trivial by passing in our acceleration values for each exposure into `scipy.integrate.cumtrapz` twice, to integrate from acceleration $\rightarrow$ velocity $\rightarrow$ position.

For the rotation matrix, we take the `dt` property of the frame and passed it into an `ahrs.filters.Madgwick` object to obtain an integrator. We reformat the starting orientation as a quaternion and pass it into the integrator, along with the gyro and accel data, to get the quaternion data for each gyro timestep. We then turn this quaternion into a `scipy.spatial.transform.Rotation` object to convert to rotation matrices.

3.4. Camera Intrinsics

We construct the camera intrinsics matrix K using information about the camera [2]:

$$K = \begin{pmatrix} f_x & s & x_0 \\ 0 & f_y & y_0 \\ 0 & 0 & 1 \end{pmatrix} \tag{13}$$

where f_x, f_y are focal lengths on the x and y axis (which we assume to be equal), x_0, y_0 are offsets to the center of the image plane (which we assume to be 0), and s is the axis skew, which we know to be 0 due to it being a conventional pinhole-like camera.

We use the `computeIntrinsic` function provided by Assignment 6 to obtain this camera matrix. In practice, a smartphone integrator who has full access to sensor and lens parameters or more precise calibrating equipment can set this matrix with much higher accuracy.

3.5. Computing the Blur Matrix

Once the code to compute camera extrinsic and intrinsic data is set up, we are able to use equation 3 to compute the homography matrix using the `numpy` library. After warping each pixel coordinate in the image at timestep t with H_t, we use bilinear interpolation on the nearest integer coordinates, to set specific values in our sparse blur matrix A_t. To ensure our resulting image coordinates are homogenous, we divide the entire vector of form (u_0, v_0, z) by the value of z to obtain $(u_0/z, v_0/z, 1)$. We initialize A_t as a `scipy.sparse.lil_matrix` in order to not use too much memory, and convert it to a `scipy.sparse.csr_matrix` after setting specific values with our bilinearly interpolated pixel weights.

3.6. Performing Deconvolution

Once we have the code to compute each A_t, we simply run it across each frame and accumulate the sparse matrices to get our overall blur matrix for our blur model, given by the summation term in equation 6. We then use `scipy.sparse.linalg.lsmr` to run a regularized least-squares minimization and deconvolve the blurry image. We use a damping factor of $\lambda = 0.01$ as an additional input.

4. Implementation Issues

Throughout our implementation sprint, we ran into several issues, some of which we were able to overcome, and some which halted the process entirely. The main two issues are described below.

4.1. Data Parsing & Conversion

The first issue we ran into was not being able to build our rotation matrix properly. We knew we would have to do some kind of integration similar to the double integral for translation, but the abundance of data was confusing. After digging around for a while, we found the Madgwick filter within a robotics library, which coincidentally used the types of data we had acquired form the camera: accelerometer, gyroscope, and magnetometer, while simultaneously giving us 3 more degrees of freedom than the original paper.

4.2. Homography Matrix Usage

The biggest issue we've run into with building our homography matrix is that it always outputs out-of-bounds coordinates when its inverse is applied to a warped image coordinate. We initially used the camera intrinsic code given in `cp_hw6.py` to computationally calibrate our camera matrix, but we also tried to switch to building our own intrinsic matrix with sensor information we could find online, which still results in a homography that gives us negative image coordinates (see figure 4). We got a bit stuck at this step, since we followed the paper's mathematics step by step, and did no additional work on any of the steps except for constructing the intrinsics matrix. Since we are unsure on how to scale down the coordinates we received to be valid image coordinates within [0, 1920) $\times$ [0, 1080), we are thus unable to fully test the implementation. Therefore, we are submitting a Github repository containing all the work we have so far to demonstrate our progress (linked at the end of this report).

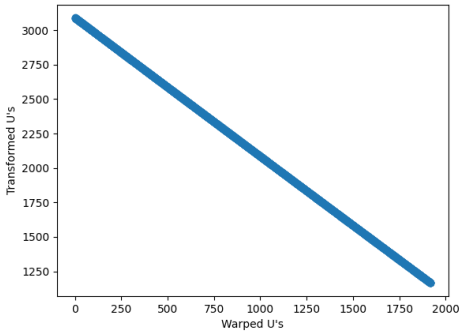

Figure 4. Plot of warped u coordinates and their mapped transformed coordinates.

5. Results and Discussion

We base our discussions for this section off of the results from [4], but assume that our improved implementation works as we had hoped.

5.1. Deblurring Results and Discussions

In the original paper, images taken outside of a controlled lab setting and deconvolved with the given algorithm showed a clear improvement over the input blurry image (see figure 5). The authors state that there is still some residual artifacting due to frequency loss when the image becomes blurred, but not as much artifacting as in the results from using Shan et al and Fergus et. al's blind deconvolution deblurring methods. All images shown in figure 5 were shot with 1/2 to 1/10 second exposures with a 40mm lens on a Canon 1Ds Mark III.

Theoretically, this means that the algorithm should be generalizable to a smartphone camera and IMU setup (and still produce fairly decent sharp image results), and that

there are just some confounding variables in our code implementation that we may have missed which may be preventing us from moving forward. We do believe that the mathematical equations related to doing the actual convolution are correct.

5.2. Further Improvements

Due to constraints on time, we did not implement the drift compensation algorithm mentioned in the paper, where a search box of defined size and an optimization algorithm are used to search for the most likely endpoint of the camera motion for a certain exposure. This seems to work well for them, and is definitely something we would like to try ourselves.

Since we did not have access to the equipment that the paper had, there are also some calibrations that we did not do - namely, calculating the drift of the sensors by rotating it at a known rate and observing outputs, and mapping the motions of the sensor to the true location on the camera plane. These are all crucial to the performance of our deblurring algorithm, that we hope to be able to do. However, smartphone integrators may have conducted some of these calibrations for us in the process of designing the phone's hardware and software.

The original paper proposed multiple aspects that could be further improved. The one easiest for us was to use more sensors to help with calculating camera motions, which we implemented. However, the other two main areas for improvement mentioned in the paper - using depth information and getting a better initial position - were not implemented by us, though we have some ideas for high-level approaches.

To obtain depth information, we could use a camera system that could capture it, such as a lightfield camera or a parallel Time of Flight sensor. The ToF sensor is particularly interesting, as it is already part of some smartphone camera systems such as the iPhone 13 Pro, so it is definitely realistic to incorporate that in this project.

The initial position is important, as it provides us with crucial info about the direction of the gravity vector in relation to each principal axis of motion. Having this would enable us to calculate translational velocity (and therefore position) in a much more accurate way. Unfortunately, there does not seem to be a hardware solution practical enough to use in a smartphone, and the math that we performed to attempt to calculate the direction of the gravity vector makes many assumptions and is unable to calculate one of the axes. A better way of measuring the gravity direction would be immensely helpful for our implementation of this project.

We realize that the smartphone's MARG sensor is not designed for highly precise applications such as pose estimation, inertial navigation, or our algorithm. This is could potentially degrade our results severely, since we measure motion in the millimeter to micrometer range, which could expose imperfections in the sensor to extreme levels. Higher end phones may give us better sensors, but we imagine that to get the absolute best result, a robotics-grade sensor should be installed in the phone.

Finally, the ideal setup would involve all of this processing occurring on the smartphone itself, as the user would probably want their shaky images to be deblurred almost immediately. Due to the nature of the logger app, it was difficult to add separate image processing code within the same app itself, and we had to work with the logged data elsewhere. We decided to go with this approach because of our experience with the relevant Python libraries, but shifting everything to be on the Android app would require finding Java or C++ equivalents to these libraries and sufficient documentation to translate the implementation over.

6. Github Repository

Link to the Github repository for this project is below: `https://github.com/jxu12345/15463_final`

References

[1] M. Garcia. Madgwick orientation filter. From Python AHRS documentation at https://ahrs.readthedocs.io/en/latest/filters/madgwick.html.

[2] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521540518, second edition, 2004.

[3] J. Huai, Y. Zhang, and A. Yilmaz. The mobile ar sensor logger for android and ios devices. In *2019 IEEE SENSORS*, Oct 2019.

[4] N. Joshi and et al. Image deblurring using inertial measurement sensors., 2010.

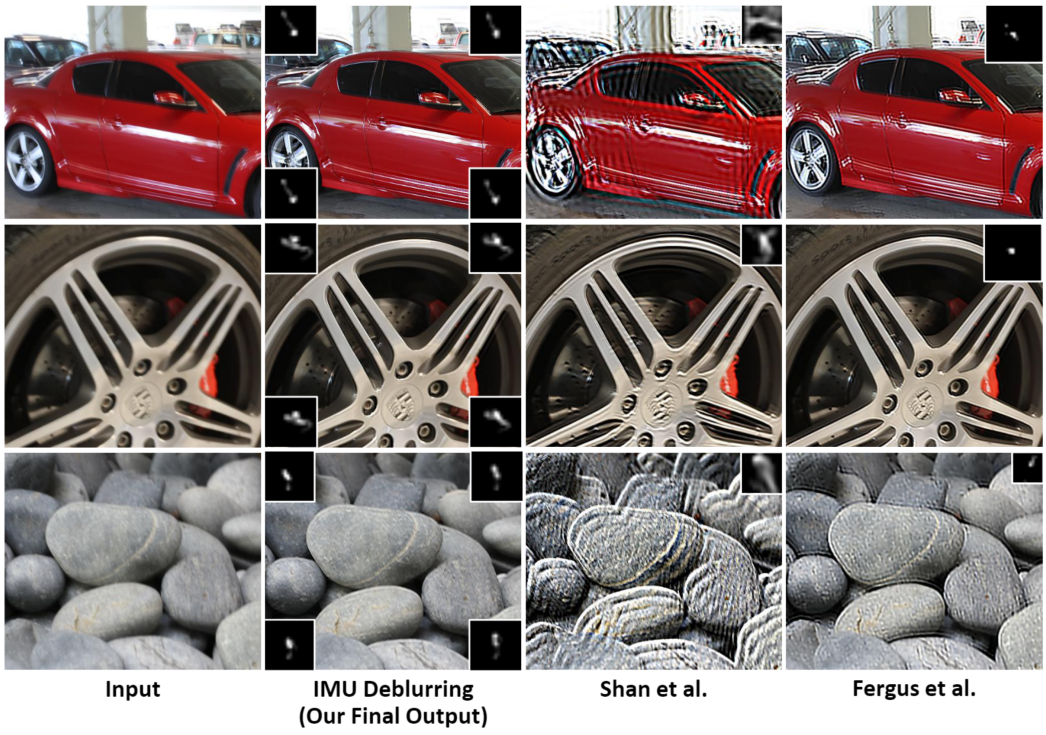

Figure 5. Image deblurring results from the original paper, with the computed blur kernels.