POINT SPREAD FUNCTION ESTIMATION MODELING AND NON-BLIND DECONVOLUTION OF PHOTOS WITH SPATIALLY-VARIANT MOTION BLUR THROUGH INERTIAL SENSOR DATA

THESIS

Submitted to fulfill one of the requirements

for a *Sarjana Komputer* degree

from the Informatics Study Program

Compiled by:

Ghazali Ahlam Jazali

Student Identification Number: 215314191

FACULTY OF SCIENCE AND TECHNOLOGY

SANATA DHARMA UNIVERSITY

YOGYAKARTA

2025

PEMODELAN ESTIMASI *POINT SPREAD FUNCTION* DAN *NON-BLIND DECONVOLUTION* CITRA DENGAN BURAM GERAKAN YANG BERVARIASI SECARA SPASIAL BERDASARKAN DATA SENSOR GERAK

SKRIPSI

Diajukan untuk memenuhi salah satu syarat

memperoleh gelar Sarjana Komputer

Program Studi Informatika

Disusun oleh:

Ghazali Ahlam Jazali

NIM: 215314191

FAKULTAS SAINS DAN TEKNOLOGI

UNIVERSITAS SANATA DHARMA

YOGYAKARTA

2025

THESIS

POINT SPREAD FUNCTION ESTIMATION MODELING AND NON-BLIND DECONVOLUTION OF PHOTOS WITH SPATIALLY-VARIANT MOTION BLUR THROUGH INERTIAL SENSOR DATA

Written and prepared by:

Ghazali Ahlam Jazali

Student Identification Number: 215314191

Thesis Advisor,

Drs. Hari Suparwito, S.J., M.App.I.T.

24 January 2025

POINT SPREAD FUNCTION ESTIMATION MODELING AND NON-BLIND DECONVOLUTION OF PHOTOS WITH SPATIALLY-VARIANT MOTION BLUR THROUGH INERTIAL SENSOR DATA

THESIS

Written and prepared by:

Ghazali Ahlam Jazali

Student Identification Number: 215314191

EXAMINING BOARD STRUCTURE

POSITION		FULL NAME	SIGNATURE
Chairman (and member)	:	Dr. Ir. Ridowati Gunawan, S.Kom., M.T.	.....
Secretary (and member)	:	Dr. Sri Hartati Wijono S.Si., M.Kom.	.....
Member	:	Drs. Hari Suparwito, S.J., M.App.I.T.	.....

Yogyakarta, 24 January 2025

Faculty of Science and Technology

Sanata Dharma University

Dean,

Ir. Drs. Hans Sriwindono, M.Kom., Ph.D.

STATEMENT OF AUTHENTICITY

I declare that this thesis does not contain the work or parts of the work of others, except those that have been mentioned in quotations and bibliography by following the provisions as befits scientific work.

If in the future there are indications of plagiarism in this manuscript, I am willing to bear all sanctions in accordance with applicable laws and regulations.

Yogyakarta, 19 December 2024

Writer,

Ghazali Ahlam Jazali

STATEMENT OF APPROVAL FOR PUBLICATION OF SCIENTIFIC WORK FOR ACADEMIC PURPOSES

The undersigned, a student of Sanata Dharma University:

Name	:	Ghazali Ahlam Jazali
Student Identification Number	:	215314191

For the development of science, I give to the Sanata Dharma University Library my scientific work entitled:

"Point Spread Function Estimation Modeling and Non-Blind Deconvolution of Photos with Spatially-Variant Motion Blur through Inertial Sensor Data"

along with the necessary equipment (if any). I hereby grant the Sanata Dharma University Library the right to store, transfer in the form of other media, process in the form of databases, distribute in a limited manner, and publish it on the internet or other media for academic purposes without the need to ask permission from me or give royalties to me as long as I keep my name as the author.

Thus I make this statement truthfully.

Written in Yogyakarta

on the date: 24 January 2025

Writer,

Ghazali Ahlam Jazali

CONTENTS

LIST OF FIGURES

LISTINGS

LIST OF ABBREVIATIONS

PSF	:	Point Spread Function; synonymous with “kernel”.
ConvNets	:	Convolutional Neural Networks
ISO	:	International Organization for Standardization; in photography, the term refers to the (tunable) sensitivity of a camera’s image sensor to light.

PREFACE

For a long time, I have always found the problem of non-blind image deblurring—which is the topic of this research—as something interesting. However, it was not until I studied as an undergraduate exchange student at the University of Pennsylvania that the research presented in this thesis first came into being. It started as a topic that I came up with for the final project of *CIS 5810: Computer Vision & Computational Photography*. Because of that, I would like to extend my gratitude to Professor Jianbo Shi for encouraging me to dive into this subdiscipline of computer vision and for providing me with guidance. Although challenging, I enjoyed every minute of the studying that culminated in this thesis.

I also would like to express my gratitude toward my thesis advisor, Drs. Hari Suparwito, S.J., M.App.I.T. This thesis would not be what it is today were it not for the advice and instructions he provided. Moreover, the help provided by Dr. Sri Hartati Wijono S.Si., M.Kom as well as Dr. Ir. Ridowati Gunawan, S.Kom., M.T. during the proposal-stage of this research was highly invaluable to its development.

Finally, I would like to acknowledge the unwavering support of my family and friends, whose encouragement and belief in my abilities have kept me motivated throughout this journey.

I sincerely hope that the work presented in this thesis will add to the growing body of knowledge in the field of computer vision and push for more research in non-blind image deblurring.

Yogyakarta, 19 December 2024

Writer,

Ghazali Ahlam Jazali

Abstract

Despite the growth in image sensor technology, key limitations remain. Modern image sensors in smartphones, due to their smaller size—relative to their DSLR counterparts—often have to balance between ISO value and shutter speed. When the value of the former is raised, the sensor will allow more light in but the resulting photos will gain noise due to the increased sensitivity. When the latter is lowered, more light can be gained without trading it off with noise; although, motion blur will be easier to unintentionally introduce to the image. This research offers a way to take photographs without having to painstakingly optimize for the least amount of noise and motion blur. By initially allowing motion blur to be present, the camera sensor can then be set towards lowering its ISO value to suppress noise. The motion blur in the resulting image can then be removed through non-blind deconvolution based on a known (estimated) kernel. The kernel (or PSF) that is used for the deconvolution process is obtained through a modeling technique that utilizes data from inertial sensors. To conduct the modeling, this research presents a new algorithm to generate spatially-variant PSFs given the appropriate inertial sensors data. The algorithm fits into an end-to-end image deblurring pipeline. Additionally, unlike most computer vision literature dealing with motion blur removal, this research provides a comprehensive but concise guide on the implementation of the said PSF modeling technique.

Keywords. Non-blind deconvolution, motion blur, computer vision, computational photography, ISO, shutter speed

CHAPTER I

INTRODUCTION

1.1 Background

Over the course of decades, the quality of phone cameras have improved dramatically. However, because they are limited by their relatively smaller sensor size—compared to DSLR cameras—the quality of the images produced by them degrades once the shooting condition is less-than-ideal [1]. In low-light environments, phone cameras will attempt to balance between the shutter speed (lowered) and the sensor's ISO value[1] (raised) in order to compensate for lack of sufficient lighting. Letting the shutter open for a longer period of time will allow for more light enter. The main trade-off to this, however, is the increased likelihood of motion blur being introduced to the final image; since it is highly probable that the photographer moved during the time window where the sensor is exposed to light[2]. Multiple research have attempted to address this problem, mainly by defining a system where motion blurs can initially be tolerated—with the premise that it would be removed later during post-processing.

A classic example of such research, [2], described the use of a secondary camera to work in tandem with a primary camera. The secondary camera captures video starting from the exact moment the primary camera starts capturing image—up to the point when it stops doing so. The frames of the video effectively captures the magnitude and direction of the primary camera's sensor displacement during image capturing. According to its testing results, this approach yields a high degree of accuracy given some conditions. However, it operates under the assumption that the motion blur in an image as a whole is spatially-invariant, i.e., can be described by a single point spread function. Realistically, however, the cause of motion blurs is a combination of translation and rotation of the sensor in three-dimensional space. This means that a degree of spatial variance will ultimately be present.

[1]The ISO value indicates how sensitive the camera sensor is to light. A sensor set to a higher ISO value will generate more luminance and color noise on the image it takes.

[2]In addition to the photographer's movement during shutter exposure, motion blur may also be caused by the movement of the subject.

Another way to gain knowledge on the spatial displacement of the sensor during the time when it was exposed to light can be done through the recording device's inertial sensor. The one type of standard inertial sensor that is embedded in most modern smartphones is the gyroscope. Gyroscopes have the capability to detect angular velocity. Through the use of integration, the gyroscope can effectively give readings on the camera sensor's rotational movement on three-dimensional space—that is, all x, y, and z axes. One approach that took advantage of inertial sensor measurements is [3]: by using a convolutional neural network.

Through the implementation of an encoder-decoder architecture, the authors of the paper demonstrated the viability of their model in eliminating a high degree of motion blur. However, due to the nature of convolutional neural networks, artistic types of blurs that may be desirable to retain (e.g., shallow depth-of-field), may be falsely considered as motion blur and subsequently deblurred. In addition to that, the motion blur model that was used had to be simplified into a linear approximation so that it can be concatenated to the back of the image tensor as two matrix representation of the model ("blur fields" in the x and y directions). Moreover, the method by [3] is unable to detect translational movement due to the lack of accelerometer data—although, it has been noted throughout deconvolution literature that the effect of translational displacement on motion blur is limited for certain situations (and adds unnecessary complexity to the motion blur model).

This research aims to create a non-blind image deconvolution pipeline that considers both the motion blur spatial variance from translation and rotation. The pipeline consists of two main sections: point spread function (PSF) estimation and non-blind deconvolution. Additionally, as an integral part of the pipeline, this research presents a new algorithm that is capable of producing realistic PSFs through inertial sensor data.

1.2 Problem Formulation

This research attempts to tackle the problem of motion deblurring by proposing a photo processing software pipeline. The pipeline takes an image degraded by natural motion blur along with the motion data of the device during the time the photograph was taken. Inside

the pipeline, a realistic model of the motion blur is inferenced using the provided inertial sensor data to produce a PSF matrix; which in the image degradation model detailed in Eq. 2, substitutes the kernel k. Motion blur is then removed from the image through non-blind deconvolution, which takes in a blurry image and an estimation of its corresponding PSF. The output of the pipeline is the latent image.

As it is computationally-efficient, the pipeline will be able to run on mobile devices. The pipeline also allow users to capture even higher quality images than what their camera normally allows. This is because users will no longer be constrained to shorter shutter speeds; a longer shutter speed will allow more light to be captured by the sensor, hence details within the image—which would otherwise be lost to noise due to high ISO—will show up more pronounced. Although non-blind deconvolution tend to leave artifacts—the degree of which depends on the PSF estimation quality—degradation from them tend to not obscure fine details as much as image noise (and motion blur) do. This research trades-off noise degradation in favor of deconvolution artifacts because of its tendency.

1.3 Research Objectives

The goal of this research is summarized by the following:

1. To create an end-to-end image motion deblurring pipeline, as described in the problem formulation;

2. to describe a novel algorithm that converts inertial sensor data to PSFs;

3. to provide a comprehensive reference for obtaining realistic PSF models through the use of inertial sensor data to the computer vision literature;

4. to provide quantitative measurements and qualitative comments of the quality of images deblurred through non-blind deconvolution and the quality's relationships to other parameters, such as the attributes of the PSF.

1.4 Research Benefits

An extensive number of research have been written on image deblurring—ranging from the direct use of non-blind deconvolution algorithms to approaches such as ConvNets. However, to date, there has not been a comprehensive and concise reference, gathered under a single paper, on the use of inertial sensor data to model PSFs along with the implementation of the modeling. This research contributes to the deconvolution literature by providing a framework to convert inertial sensor data into realistic discrete PSFs.

In addition to providing benefits for public use (i.e., for smartphone photography), under the assumption that its use grows popular enough, the pipeline could potentially set standards for photograph metadata. If the pipeline is made easily accessible for any developers to implement into their products, camera manufacturers will be more inclined to include inertial sensor data accompanied by accurate timestamps (down to nanoseconds) as part of their device's default photograph metadata. The adoption of such standard would mean that most photographs taken will be accompanied by inertial sensor data. This means that if any of those photographs turned out to be affected by motion blur, their clear version can be retrieved regardless of the device used to capture them. This can be beneficial for the field of computer vision. For instance, in training a ConvNets for image recognition, blurry photographs in the datasets that would have otherwise been discarded during the preprocessing stage can be kept and deblurred.

1.5 Scope and Limitations

Any motion blur that was not caused by the capturing camera sensor motion might negatively affect the result of the image deblurring. For instance, if a car was passing through when an input photograph was taken, the non-blind deconvolution algorithm applied on the image might refine the initial estimation kernel based on motion blur that was produced by the passing car instead of the motion blur caused by the camera's movement. Because of that, it is likely that the photo processing pipeline will perform worse on photos where objects moves independently, especially if the objects' movements are not parallel to the movement

of the camera sensor.

In addition, another one of the pipeline's point of failure is the accuracy of readings from the hardware involved (to a degree, this excludes the reading accuracy of the inertial sensor data). For instance, the camera that captures the image needs to be calibrated with respect to its focal length, optical center, and skew coefficient [4]. With a faulty calibration, the set of point spread functions inferred from the inertial sensors may not match the actual motion blur present in the image.

Due to the reasons iterated above, the pipeline operates under the following assumptions:

1. Objects within the captured images are stationary; the motion blur within the images are purely caused by the sensor's movement.

2. The calibration variables (e.g., the camera matrix, etc.) for the necessary hardware components that the software pipeline operates on are accurately known.

1.6 Writing Systematics

This research proposal has been written in a way that assumes the reader's basic knowledge of computer vision techniques and familiarity to basic terminologies. However, more obscure topics such as deconvolution will be preceded by short introductions.

Chapter 1 includes some background knowledge to give a motivation on the need to solve motion deblurring—an ill-posed inverse problem in computer vision. It also gives the problem formulation to define the questions that the research is aiming to answer. The objectives and benefits of the research is later outlined to give a view on the final products of the research and their expected impacts on its target disciplines respectively. This chapter defines the scope and limitations of the research to set some boundaries on what it will and will not explore.

Chapter 2 contains a review on approaches commonly used to address the problem of image deblurring, as well as methods that are less common but are noteworthy. This chapter also includes a theoretical framework that are critical to understanding the motion deblurring technique presented by this reasearch.

Chapter 3 outlines the tools and resources that this research intends to use to construct the product. It also gives a general overview on how the research is to be carried out, as well as details on its implementation and testing scenario. Chapter 4 Discusses the the methodology implementation as well as its result, and Chapter 5 concludes the research.

CHAPTER II

LITERATURE REVIEW

2.1 Previous Research

Motion deblurring is an ill-posed problem. Over the past couple of years, great advances have been made in an attempt to resolve it, by attempting to find ways to constrain the solution space presented by this problem.

2.1.1 Using Multiple Cameras

A method of adding a secondary, high–temporal resolution camera to capture video that can be decoded into motion data was proposed by [2]. To ensure that the video from this camera can accurately represent the spatial displacement of the sensor in the primary camera, three of prototypes were proposed—each of which has their own disadvantages. Prototype (a) places the secondary camera next to the primary one—where each camera are independent of each other, in a sense that they have their own lenses and sensors. There are a number of problems with prototype (a), namely the fact that the images produced by the two has to be identical in all aspects other than resolution, color channels, and noise. This condition is particularly difficult to satisfy, given the different focal length and size of both the lenses and the sensors. Prototype (b) improves upon these issues by using only one lens but still with two separate lenses. Using a beam splitter, the images produced by the sensor can have an identical perspective because they came from the same lens. Prototype (c) takes this further by combining the two devices into a system of one lens and a binned CMOS sensor.

The motion data provided by the secondary camera can then be converted into a point spread function (PSF) that describes the motion blur applied to the latent image. In order to do so, each frame of the video is treated as discrete samples of the sensor's displacement over the period of time when it was exposed to light. That set of discrete samples can be made into a continuous PSF through the use of spline interpolation. Once the PSF is found, a non-blind deconvolution operation can be applied in order to extract the latent image. In

this particular case, the iterative Richardson-Lucy algorithm is used due to its convenient property of never producing negative value outputs, which, according to the authors, "make better physical sense than linear methods."

This accuracy of this method is in the subpixels level, given that the secondary camera has a high enough sensor resolution and a low enough luminance noise standard deviation. However, due to the assumption that a single PSF can describe the motion blur of the whole image, this method does not address the possibility of a spatially-variant motion blur within the image—where each object within the picture may exhibit a different PSF.

2.1.2 Using Inertial Sensor Data and ConvNets

On the other hand, [3] utilized a encoder-decoder–structured convolutional neural network trained using a set of $100{,}000$ (1) degraded images, (2) their motion data obtained from the capturing device's gyroscope, and (3) ground truth images that describes how the latent images should look like. In order for the motion data to be fed to the ConvNets, they first need to be converted into a form that can be concatenated with the image tensor. The authors do this by converting the angular displacement on all x, y, and z axes from the gyroscope data into what they refer to as "blur fields" ($\mathcal{B}$). The blur field consists of two 2-dimensional matrices that are of the same $H \times W$ size as the motion-blurred image (excluding color channels). The each element in the $\mathcal{B}_x$ and $\mathcal{B}_y$ blur fields are the value of the blur vector in the x and y axes; hence $\mathcal{B}$ as a whole represent the PSFs at every pixel of the image. These PSFs are assumed to be linear in nature in order to represent each of them as a vector of two elements $[x, y]$.

As indicated by the description of blur fields generation above, [3]'s method does not directly deal with PSFs, and the non-blind deconvolution operations are done within the decoder part of the ConvNets. This approach simplifies the pipeline of image deblurring, albeit with a number of trade-offs, including the granularity of control in crafting the pipeline itself.

The nature of the blur fields generation described in the paper only takes into account the spatial variance of the motion blur with respect to the motion data provided by the

gyroscope—it does not directly consider the distance of objects within the images relative to the camera sensor. Which means that, during training, while object distances *may* be picked up by the convolutional layers as features, and their relationships with the provided blur fields *may* be determined by other parts of the hidden layers—how well the resulting model performs depends almost solely on the quality of the training dataset and how well it represents real-world conditions. With a two-step process such as with [2], certain pre-processing operations can be applied on—for instance—converting discrete motion samples into a continuous PSF if it was found that it did not perform well enough when deconvolved against degraded images. In addition, there is no way guarantee that the model deconvolves motion blurs only. As stated by [5], the network's training may not be sufficient to enable the resulting model to distinguish between degradative and intentional (e.g., shallow depth of field) blurs every time.

2.1.3 Using Inertial Sensor Data for Non-Blind Deconvolution

Instead of letting a ConvNets to conduct the deconvolution—a class of solution called blind deconvolution—[6] proposed a two-step method to motion deblurring. This is done by first estimating the motion blur kernel associated with the input image. If a blur kernel is known, then the problem—the restoration of the latent image—becomes solvable through non-blind deconvolution. The authors employed a non-blind deconvolution method described by [7], which is an improved version of the Richardson-Lucy non-blind deconvolution algorithm that prevents ringing artifacts that are especially prevalent in deconvolved low-light photos featuring clipped[3] highlights.

The paper by [6] also pointed out several area where the innacuracies might appear in kernel estimation, including the rolling shutter effect, the effects of gravity on sensor readings, as well as the possibility that the rotation center of the motion is not the sensor itself. For these described problems, [6] gave a detailed analysis and proposed an online calibration method that utilizes light streaks and autocorrelation map–based methods in refining the

[3] In photography, clipped highlights refer to areas within a photo that shows up as pure white because of luminance exceeding the camera sensor's dynamic range.

initial motion sensor–provided kernel estimations.

2.2 Theoretical Framework

2.2.1 Convolution

In computer vision, convolution refers to 2-dimensional discrete convolution. Convolution is an operation that can be used to mathematically express many forms of image degradation. An image that has been artificially blurred, for instance, can be modeled the following way:

$$B = I \otimes k \tag{1}$$

where I, $\otimes$, k, and B respectively denotes the latent image, the convolution operator, the blur kernel[4] to which I is convolved with, and the blurred image. However, in real-world circumstances, there is often the presence of additive noise [8]. Therefore, it is more common in computer vision literature to model motion blur as

$$B = I \otimes k + \varepsilon \tag{2}$$

where ε, according to [9], denotes a "zero-mean, identically- and independently-distributed noise term at every pixel."

Eq. 3 breaks down how the convolution operator $\otimes$ in Eq. 2 works:

$$B_{ij} = \left[\sum_{m=0}^{M} \sum_{n=0}^{N} I(i-m, j-n) k(m, n) \right] + \varepsilon_{ij} \tag{3}$$

For Eq. 1 and Eq. 2, there are many possible blur kernels that can substituted into k. The effects of box blur is well known: it effectively reduces the inter-element differences within a certion region of a matrix—a "fuzzy" operator. A box blur kernel is one in which all its elements are equal to one another. However, for the resulting image to retain the "brightness" of its original counterpart, the elements within the kernel must add up to one. Meaning that

[4]Note that in the field of computer vision—especially in the context of deconvolution—the term "kernel" is often used interchangeably with "point spread function" (PSF) and "impulse response function."

a 3×3 box blur kernel will have $\frac{1}{9}$ spread over uniformly throughout the matrix:

$$k = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{4}$$

Another kernel that is often used apply artistic blur to images is the Gaussian kernel[5].

$$k = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \tag{5}$$

The use of convolution is not limited to blurs such as the ones described above. To emulate motion blur, k from Eq. 2 can be substituted using kernels such as

$$k = \frac{1}{3} \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \tag{6}$$

The resulting output B will exhibit a diagonal motion blur—as if the photographer moved the camera from the bottom left to the top right when capturing the image.

Intuitively, kernels in convolution can be thought of as a description of how every single point within an image is going to be "spread out," hence the alternative term "point spread function" (PSF). In other words, according to [10], it is a " system response to a point source placed at the origin of the image." In order for the convolved image to retain the original image's luminosity, the cumulative value of the elements inside the kernels must add up to one.

The images in Figure 1 illustrates how kernels like Eq. 5 and Eq. 6 affects an image through convolution:

[5]Looking through the values within the matrix k in Eq. 5, one should notice that the Gaussian kernel (3×3 in size) is based on the Gaussian distribution. Hence, the kernel is a discrete approximation of the actual continuous distribution.

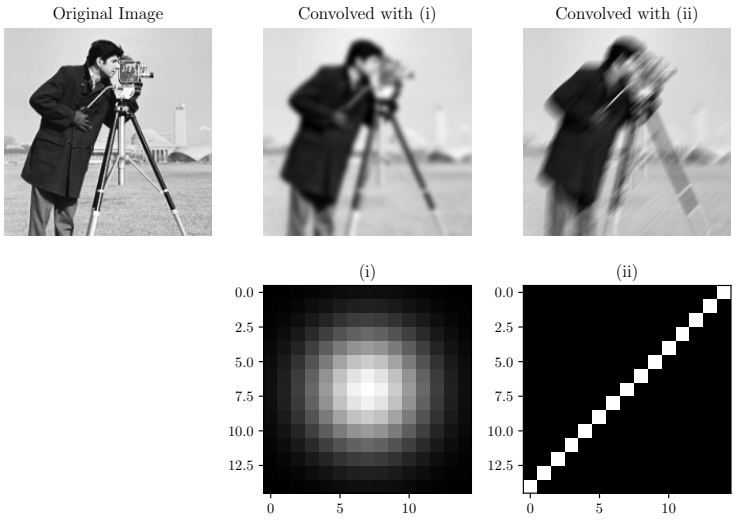


Figure 1. Effects of convolution with a 15×15 Gaussian and linear (top right to bottom left) motion blur kernels (PSF).

2.2.2 Deconvolution

The inverse to the convolution operation is known as deconvolution, as reported by [11]. To a certain degree of accuracy, deconvolution makes it possible to recover an original image I from a degraded one B (as modeled in Eq. 2). This reversion process, given a known PSF—or at least a known estimate of it—is referred to as non-blind deconvolution. Conversely, when no knowledge of the PSF is provided, the process is known as blind deconvolution [12].

The idea behind classical non-blind deconvolution lies in the convolution theorem, which is discussed by [13]:

$$f(\mathbf{x}) \otimes g(\mathbf{x}) \Leftrightarrow F(\boldsymbol{\omega})G(\boldsymbol{\omega}) \tag{7}$$

The theorem states that the convolution of two functions in the time domain is equivalent to the product of their respective Fourier transforms[6] in the frequency domain. The terms in Eq. 1 may be expressed as functions

$$h(\mathbf{x}) = (f \otimes g)(\mathbf{x}) \tag{8}$$

[6]The Fourier transforms of functions are denoted by capital letters.

where $f(\mathbf{x})$, $g(\mathbf{x})$, and $h(\mathbf{x})$ denotes the latent image, the PSF, and degraded image respectively; and $\mathbf{x} = (x, y)$. With Eq. 8, deconvolution—that is, finding the unknown term $f(\mathbf{x})$ given a known $g(\mathbf{x})$—can be done by first computing the discrete Fourier transform of each known terms, which can be symbolically expressed as $\mathcal{F}\{\bullet\}$:

$$H(\boldsymbol{\omega}) = \mathcal{F}\{h(\mathbf{x})\} \tag{9}$$

$$G(\boldsymbol{\omega}) = \mathcal{F}\{g(\mathbf{x})\} \tag{10}$$

Taking note of Eq. 7, the resulting Fourier transforms of the terms can then be arranged the following way:

$$H(\boldsymbol{\omega}) = F(\boldsymbol{\omega})G(\boldsymbol{\omega}) \tag{11}$$

Here, we also denote the Fourier transform of the unknown term $f(\mathbf{x})$ as $F(\boldsymbol{\omega})$ for convenience. To find $F(\boldsymbol{\omega})$, Eq. 11 can be rearranged as

$$F(\boldsymbol{\omega}) = \frac{H(\boldsymbol{\omega})}{G(\boldsymbol{\omega})} \tag{12}$$

Once $F(\boldsymbol{\omega})$ is calculated, its inverse Fourier transform (denoted as $\mathcal{F}^{-1}\{\bullet\}$) can be computed to find the original image $f(\mathbf{x})$:

$$f(\mathbf{x}) = \mathcal{F}^{-1}\{F(\boldsymbol{\omega})\} \tag{13}$$

However, as mentioned previously, it is more accurate to model natural motion blur with an additive noise term

$$h(\mathbf{x}) = (f \otimes g)(\mathbf{x}) + \psi(\mathbf{x}) \tag{14}$$

Because of the linearity property of Fourier transforms (while Eq. 7 states that convolution in the time domain is multiplication in the frequency domain), addition in the time domain is equivalent to addition in the frequency domain. When the noise term $\psi(\mathbf{x})$ is involved, Eq.

11 becomes

$$H(\boldsymbol{\omega}) = F(\boldsymbol{\omega})G(\boldsymbol{\omega}) + \Psi(\boldsymbol{\omega}) \tag{15}$$

$$F(\boldsymbol{\omega}) = \frac{H(\boldsymbol{\omega}) - \Psi(\boldsymbol{\omega})}{G(\boldsymbol{\omega})} \tag{16}$$

Eq. 16 above shows that to find $F(\boldsymbol{\omega})$—and subsequently $f(\mathbf{x})$—in addition to $g(\mathbf{x})$ and $h(\mathbf{x})$, the noise term $\psi(\mathbf{x})$ would also need to be known. In natural images, knowledge of additive noise is highly limited. Granted, a perfect knowledge of $g(x)$, the PSF, can also be difficult to obtain—although to a much lesser extent[7].

Moreover, despite its relative simplicity, deconvolution using division in the Fourier domain (Eq. 16), there is a well-known tendency of noise amplification—to the point where the original image becomes deteriorated to an unrecognizable extent rather than improved. According to [14], the division in Eq. 16 is the main cause of the noise amplification: the division of a small complex number by another small complex number will result in the enlargement of the term $F(\boldsymbol{\omega})$. In addition, due to its nature of being a low-pass filter, the Fourier transform $G(\boldsymbol{\omega})$ may be zero for high frequency. Because of its position in the denominator of Eq. 16, a zero value for $G(\boldsymbol{\omega})$ in an expression containing a division-by-zero.

Wiener deconvolution is an algorithm that seeks to improve Eq. 12:

$$F'(\boldsymbol{\omega}) = \frac{H(\boldsymbol{\omega})}{G(\boldsymbol{\omega})} \left[\frac{1}{1 + \frac{|\Psi(\boldsymbol{\omega})|^2 / |F(\boldsymbol{\omega})|^2}{|G(\boldsymbol{\omega})|^2}} \right] \tag{17}$$

with $\frac{|\Psi(\boldsymbol{\omega})|^2}{|F(\boldsymbol{\omega})|^2}$ being the signal-to-noise ratio (SNR), which is easier to estimate in the sense that being off, as suggested by Eq. 17, allows for the noise to be attenuated rather than amplified (as the case with Eq. 12).

The authors of [15] further improves upon a more advanced variation of the Wiener filter—the Wiener-Hunt deconvolution algorithm. Their paper detailed an unsupervised version of the algorithm (it jointly estimates the hyperparameters alongside the PSF and the image of interest). The posterior law is obtained through the Bayes rule. In the algorithm, the mean of the posterior law is used as the estimate, which is computed using Monte-Carlo

[7] especially with methods shown by [2], [7], and [9]

Markov chain algorithms.

2.2.3 Camera Movement and Spatial Variance

Images that are degraded by a single kernel are relatively easier to restore than ones that have a spatially-variant set of kernels—that is, every pixel of the latent image may not be convolved with the same kernel k; but rather a series of different kernels $k_1, k_2, k_3, \ldots, k_n$. Unfortunately, this is often the case with naturally-degraded images—as opposed to images that are deliberately degraded by convolving them with some arbitrary kernel.

Motion blurs are caused by translational and rotational movements of the sensor in three-dimensional space. With translational movements, the spatial variance depends on the depth of the objects within the image due to perspective distortion, especially in an image where its objects are of a sufficiently different distance. Objects that sits closer to the camera will create much more motion blur that the ones that are farther away. With rotational movements, spatial variance depends on the difference between pixel positions. The pixel $\mathbf{x} = (1, 1)$ may not be convolved with the same PSF as, say, $\mathbf{x} = (27, 103)$.

The diagrams in Figure 2 and 3 illustrates the different spatial variance[8] produced by translational and rotational movements on all three axes.

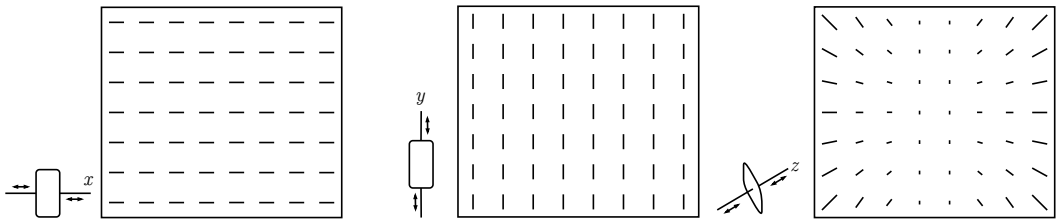


Figure 2. PSF field produced by translational movements on the x, y, and z axes.

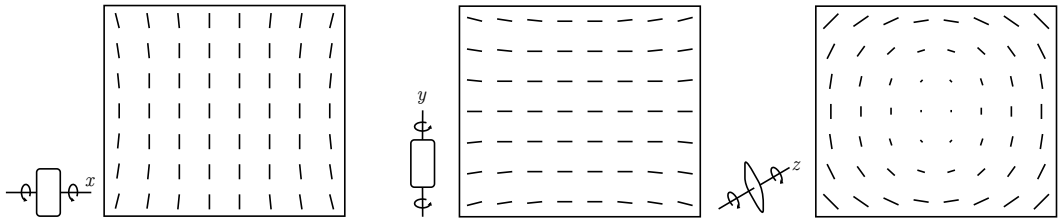


Figure 3. PSF field produced by rotational movements on the x, y, and z axes.

[8]With translational movements, spatial variance additionally depends on the depth of the scene.

According to [16], for a given blur amount δ, distance of the object from the camera d, and focal length f, the amount of translation X and rotation θ needed to cause to cause motion blur at the size of δ can be obtained through

$$X = \frac{\delta}{f} d \tag{18}$$

$$\theta = \tan^{-1}\left(\frac{\delta}{f}\right) \tag{19}$$

With Eq. 18 and Eq. 19, we can infer the fact that it takes more translational displacement than rotational displacement to create the same amount of motion blur, especially if the d is sufficiently large.

2.2.4 Inertial Sensors Data

In most modern cameras, especially with smartphones camera, there are at least two inertial sensors: the gyroscope and the accelerometer. The former captures the angular velocity of the device on its x, y, and z axes at any given time. The latter captures the acceleration of the device at those same three axes. Both sensors capture movements in a discrete fashion with a specific sampling rate.

Figure 4 and 5 illustrates the kind of movements that can be recorded by a gyroscope and an accelerometer respectively.

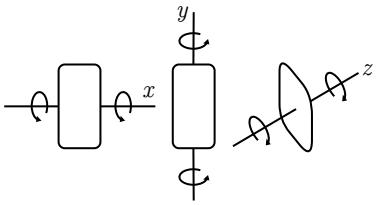


Figure 4. Rotational movement, the velocity of which is captured by the gyroscope.

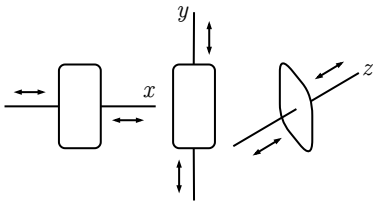


Figure 5. Translational movement, the acceleration of which is captured by the accelerometer.

Rotation Since the gyroscope captures the device's angular *velocity* rather than *displacement*, the data directly obtained from it must be integrated to find the device's displacement. Let $\theta(t)$ be the angular displacement and $\alpha(t)$ be the angular velocity. The function $\theta(t)$ can

be found by

$$\theta(t) = \int \alpha(t)\,\mathrm{d}t \tag{20}$$

Since the $\theta(t)$ given by the gyroscope is discrete rather than continuous, Eq. 20 can be rewritten into

$$\theta(t) \approx \sum_{\forall t} \alpha(t)\tau \tag{21}$$

where τ denotes the sampling interval. Eq. 21 is calculated for each x, y, and z axes of the device. Alternatively, spline interpolation can be performed on the discrete gyroscope data points to generate the polynomials representing the estimated continuous function. Due to the nature of continuity, this approach allows for sampling at arbitrary values of t.

Translation The accelerometer of a device captures its acceleration on each of its x, y, and z axes. Applying spline interpolation on the discrete accelerometer dataset allows for the resulting continuous curve to have a twice-differentiability property, enabling for the function's second-order antiderivative to be found:

$$v(t) = \int a(t)\,\mathrm{d}t \tag{22}$$

$$\delta_{\text{translation}}(t) = \int v(t)\,\mathrm{d}t \tag{23}$$

Here, $v(t)$ and $a(t)$ denotes the velocity and acceleration respectively. The translational displacement is obtained through integrating the accelerometer spline twice. When the second-order antiderivative is evaluated at t, the result, written as $\delta_{\text{translation}}(t)$, is the translation vector itself (of size 3×1), which shows the amount of translational displacement that has occured since t_0.

2.2.5 Homography

The PSF can of a motion-blurred image can be obtained through the capturing device's inertial sensor data. The motion of every pixel in the image sensor can be modeled using projective transformation, often referred to as homography in the computer vision litera-

ture. The following equation describes how the homography transformation matrix can be obtained [17]:

$$\mathbf{H} = \mathbf{K}\mathbf{R}\mathbf{K}^{-1} \tag{24}$$

According to [18], the homography transformation "describes the relationship between the real-world scene and the picture on the image plane of the camera". Given a known rotation matrix $\Delta_{\text{rotation}}(\theta)$, translation vector $\delta_{\text{translation}}(t)$, distance d, and normal matrix of the scene $n = [0, 0, 1]^\top$, $\mathbf{R}$ in Eq. 24 becomes

$$\mathbf{R}(\theta, t) = \Delta_{\text{rotation}}(\theta) - \frac{\delta_{\text{translation}}(t) n^\top}{d} \tag{25}$$

where $\Delta_{\text{rotation}}(\theta)$ is defined as

$$\Delta_{\text{rotation}}(\theta) = \begin{bmatrix} \cos\theta_x & -\sin\theta_x & 0 \\ \sin\theta_x & \cos\theta_x & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta_y & 0 & \sin\theta_y \\ 0 & 1 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_z & -\sin\theta_z \\ 0 & \sin\theta_z & \cos\theta_z \end{bmatrix} \tag{26}$$

The rotation matrix is a product of the three standard elemental rotation matrix in x, y, and z, as seen in [19]. The matrix $\mathbf{K}$, according to [16], is the internal calibration matrix

$$\mathbf{K} = \begin{bmatrix} f & s & P_x \\ 0 & f & P_y \\ 0 & 0 & 1 \end{bmatrix} \tag{27}$$

that is described by the manufacturer of the camera. Here, P_x and P_y denotes the principal points (normally defined as the center of the image sensor) and s denotes the camera's skew parameter.

When the distance d is far enough that translation no longer produces a significant enough motion blur, i.e., d is larger than the focal length f, $\mathbf{R}$ in Eq. 24 can be simplified into

$$\mathbf{H} = \mathbf{K}\Delta_{\text{rotation}}(\theta)\mathbf{K}^{-1} \tag{28}$$

2.2.6 Spline Interpolation

Spline interpolation is a method of constructing a smooth curve that passes through a given set of data points. The most commonly used type of spline interpolation is the cubic spline interpolation, where the curve is made up of piecewise cubic polynomials.

In a cubic spline interpolation, given a set of data points $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$, a cubic polynomial $S_i(x)$ is constructed for each interval $[x_i, x_{i+1}]$. In doing so, the following conditions must hold: (1) the resulting spline is continuous,

$$S_i(x_i) = y_i, \quad \forall i = \{0, 1, \ldots, n-1\}, \tag{29}$$

and (2) a continuous first and a second derivatives exists for all points,

$$\frac{\mathrm{d}}{\mathrm{d}x_i} S_i(x_i) = \frac{\mathrm{d}}{\mathrm{d}x_i} S_{i+1}(x_i), \quad \frac{\mathrm{d}^2}{\mathrm{d}x_i^2} S_i(x_i) = \frac{\mathrm{d}^2}{\mathrm{d}x_i^2} S_{i+1}(x_i), \quad \forall i = \{0, 1, \ldots, n-1\}, \tag{30}$$

Additionally, for a natural cubic spline, the second derivatives at the endpoints are set to zero:

$$\frac{\mathrm{d}}{\mathrm{d}x_0} S_0(x_0) = 0, \quad \frac{\mathrm{d}^2}{\mathrm{d}x_0^2} S_0(x_0) = 0. \tag{31}$$

Each cubic spline $S_i(x)$ that sits between two points x_i and x_{i+1} is defined by the cubic polynomial

$$S_i(x) = a_i(x - x_i)^3 + b_i(x - x_i)^2 + c_i(x - x_i) + d_i \tag{32}$$

where a_i, b_i, c_i, and d_i are coefficients. To find these coefficients, in which there are $4n$ of them, $4n$ equations needs to be set up and subsequently solved.

2.2.7 Structural Similarity Index (SSIM) for Image Similarity Measurement

First descibed by [20] in 2004, the Structural Similarity Index (SSIM) is a metric used to measure the similarity between two images which, unlike traditional methods like Mean

Squared Error (MSE) or Peak Signal-to-Noise Ratio (PSNR)—which focus on pixel-wise differences, SSIM takes into account structural information, luminance, and contrast. This image similarity measure provides a more perceptually relevant measure of image similarity—as it was designed to better reflect human visual perception, which is more sensitive to structural patterns.

SSIM is based on three main factors, luminance, contrast, and structure: it measures the degree of similarity between these three components for corresponding patches in two images.

Given two images X and Y, SSIM can be defined as

$$\text{SSIM}(X, Y) = \frac{(2\mu_X\mu_Y + c_1)\,(2\sigma_{XY} + c_2)}{(\mu_X^2 + \mu_Y^2 + c_1)(\sigma_X^2 + \sigma_Y^2 + c_2)}, \tag{33}$$

where μ_X and μ_Y denotes the mean pixel intensities of X and Y, σ_Y^2 and σ_Y^2 denotes the variances of X and Y, and σ_{XY} is the covariance between X and Y. To prevent division-by-zero or division by a very small quantity, two stabilizer variables $c_1 = (k_i L)^2$ and $c_2 = (k_i L)^2$ are introduced. Here L is the dynamic range of the image values—which would be 255 for 8-bit images and $65{,}535$ for 16-bit images (grayscale) and $k \ll 1$ is a constant of miniscule quantity.

CHAPTER III

METHODOLOGY

3.1 Tools and Resources

In obtaining the inertial sensor data, the hardware used is an Apple iPhone 13, which provides all the inertial sensors needed to construct PSFs, i.e., a gyroscope, an accelerometer, and a g-force sensor (used to determine the gravity vector from which the accelerometer data can be subtracted to get linear acceleration). The software used to obtain the data is MATLAB® Mobile, which is available on the device. MATLAB® Mobile allows for the logging of inertial sensor data at a maximum frequency of 100 Hz.

The high-resolution images used to experiment with is obtained from the DIV2K dataset, which provides a total of $1{,}200$ high-resolution images to work with. Additionally, an Apple iPhone 6 is used to capture samples of "real" PSFs to compare them with the PSF models obtained through the PSF-generator algorithm presented in this research. The use of the specific device is due to the lack of any optical and sensor-shift stabilization schemes in its back camera, which could interfere with the resulting "real" PSFs.

The programming language of choice in this research is Python due to its ease of use in defining and manipulating matrices—operations that are abundantly-done within this research. All of the processing in this research is done on a remote Oracle® Cloud compute unit with an Ampere A1 CPU, which is an ARM processor. The machine is configured to have four OCPU[9] cores and 24 gigabytes of memory.

3.2 General Overview

The product of this research is a photo processing pipeline that takes two inputs:

1. The image degraded by natural motion blur, which may have spatially-variant PSFs.

[9]The term OCPU refers to the Oracle CPU, which is a unit of measurement specific to Oracle Cloud Infrastructure compute instances. According to oracle, a single OCPU is worth "at least two vCPUs."

2. The data regarding the spatial displacement of the camera sensor during the time the sensor was exposed to light. This data is recorded by the device's inertial sensors (see 2.2). The inertial sensor data, which includes the translational and rotational displacement helps provide an accurate information to estimate the blurry image's PSF.

Within the pipeline, the inertial sensor data is used to model a set of PSFs for different area of the degraded image by passing it into a PSF-generator algorithm. The said algorithm consists of two phases: (1) converting the inertial sensor data into Cartesian-represented PSFs and (2) converting the Cartesian-represented PSFs into their matrix representation. The PSF is then passed as a parameter, alongside the degraded image, into the unsupervised Wiener-Hunt non-blind deconvolution algorithm so that the latent image can be recovered from the degraded image.

Figure 6 shows the overall outline of the proposed software pipeline.

3.3 Implementation

The point of deconvolution is described in Chapter 2.2.2: to recover the unknown latent image $f(\mathbf{x})$ given a degraded image $h(\mathbf{x})$ and estimated PSF $g(\mathbf{x})$ (see Eq. 8). The implementation details below is additionally given the appropriate inertial sensor data to (1) generate the PSFs, which is the term $g(\mathbf{x})$ in Eq. 8 critical to finding $f(\mathbf{x})$ if $h(\mathbf{x})$ is known. After obtaining the PSFs, the Wiener-Hunt deconvolution algorithm is applied with $h(\mathbf{x})$ and $g(\mathbf{x})$ as inputs in order to get $f(\mathbf{x})$.

3.3.1 Processing Raw Inertial Sensor Data

The raw gyroscope, accelerometer, and g-force sensor data that was obtained needs to be processed before it can be used to infer displacement. Linear acceleration is obtained by subtracting the total acceleration data at each axis by the g-force data at each axis multiplied

by 9.80665 (recall that $1\,\text{g} = 9.80665\,\text{m/s}^2$) at any given time t:

$$a_{\text{linear}}(t) = a_{\text{total}}(t) - 9.80665\,g(t) \tag{34}$$

where $a_{\text{linear}}(t) = [a^x_{\text{linear}}, a^y_{\text{linear}}, a^z_{\text{linear}}]^\top$, $a_{\text{total}}(t) = [a^x_{\text{total}}, a^y_{\text{total}}, a^z_{\text{total}}]^\top$, and $g(t) = [g_x, g_y, g_z]^\top$.

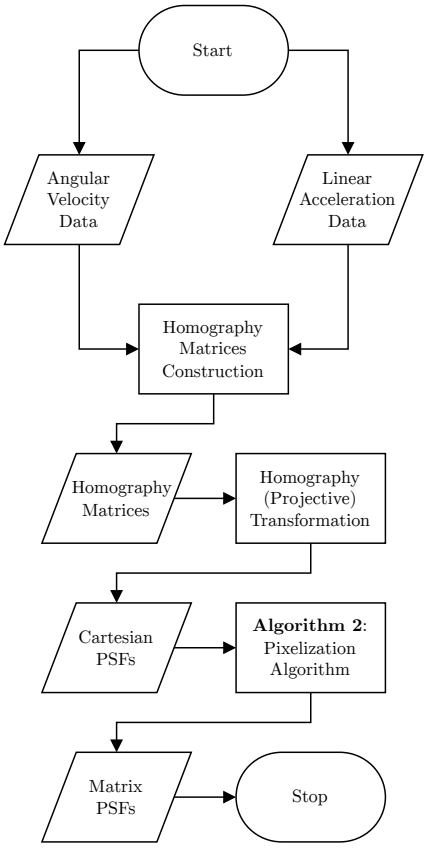
To make integrating more convenient, and to make evaluating the accelerometer data at any t value possible, the inertial sensor data points can be interpolated to produce polynomials that estimates the continuous version of the motion data. Spline interpolation, as detailed in Chapter 2.2.6 (see Eq. 32), can be used in order to ensure twice-differentiability, which is a property that is needed in order to obtain translational displacement.

For the gyroscope data, the first-order antiderivative is the rotational displacement of the capturing device at its x, y, and z axes. For the linear acceleration data, the second-order antiderivative is the translational displacement of the device.

3.3.2 Generating Point Spread Functions

Given a linear accelerometer data and a gyroscope data that captures the movement of the image sensor as it was exposed to light (the movement during integration creates motion blur), this research presents a new algorithm to generate PSFs for every possible points within the image frame. The basic outline of the algorithm (devoid of implmentation detail) is given below in Algorithm 1.

Algorithm 1 The conversion of inertial sensor data into Point Spread Functions



It is common in the image deconvolution literature to utilize inertial sensor data to construct homography matrices (partly because it is intuitive to do so). However, as the use of a two-phase PSFs construction—where the first phase creates Cartesian-represented PSFs, represented as vector-valued functions (see Eq. 35), which is then further processed into matrix-represented PSFs—as far as is known, has yet been discussed; their implementation details will be discussed in the following paragraphs.

Constructing the Homography Matrix The homography matrix is defined by Eq. 25. Before its construction, the translation vector and the rotation matrix needs to be obtained. The translation vector is simply the vector that contains the values of the displacement of the capturing device at x, y, and z. The rotation matrix, as instructed by Eq. 26, is a combination of three 3×3 elemental transformation matrix (for 3-dimensional space): the rotation matrix for the x axis, y axis, and z axis.

After obtaining the translation vector and rotation matrix, the value of the intrinsic matrix is also needed. The matrix accepts the focal lengths in unit of pixels. The unit conversion requires knowledge on the size of the pixels in millimeters. In the case of the iPhone 13, it is 0.0017 for the wide angle camera. The principal points, on the other hand, is normally put in as the center of the image sensor. However, in the case where the gyroscope is away from the center of the sensor, the principal points can be moved towards the location of that sensor—away from the image frame. This is an ideal calibration model if the capturing device that is used does not have its motion sensors aligned to the center of the camera sensor.

The final homography matrix can then be calculated using the formula presented in Eq. 24.

Generating the Cartesian Representation of the Point Spread Functions Now that the homography matrix has been obtained, determining the point-spread in every part of the image is convenient. It can be done simply by taking the homography matrix and the desired homogeneous coordinate that falls within the original image frame (before the motion began) and multiply them together. To convert an ordinary $[x, y]^\top$ coordinate into a homogeneous coordinate, augment the value 1 as the third value of that column vector, turning it into $[x, y, 1]^\top$. After the transformation, the resulting homogeneous coordinate must be normalized back by dividing every element with whatever the augmented element became after the projective transformation (i.e., if the augmented value yielded a value of 6.3 after the transformation, the each member of the vector needs to be divided by 6.3). After the division, the value of that augmented value would return to one. At this point, that augmented value can be removed again.

The idea behind generating a continuous PSF is by computing the projective transformation of the desired points within the image at multiple times until the end of the desired movement interval. Essentially, this is a sampling of the projective transformations in some desired interval. The smoothness of the Cartesian-represented PSF depends on the how many samples are taken within an interval.

Since the existance of rotational movements introduces spatial variance in PSFs, the

transformations can be done for evenly-sized sections of the image.

The Cartesian representation of the PSF is a vector-valued function (which is a function where the domain is a real number while the range is a vector) that receives t (represented as z) as input and outputs a 2×1 vector $[x, y]^\top$:

$$\vec{r}(z) = \langle f(z), g(z), z \rangle, \tag{35}$$

where $f(z)$ and $g(z)$ are the component functions showing the x- and y-based displacement of the point of interest during the homography transformation.

Generating the Matrix Representation of the Point Spread Functions This research introduces a simple algorithm that converts the Cartesian representation of a PSF into its matrix representation. Given a Cartesian plane with each of its 1×1 grid representing a single pixel, this algorithm decides the actual pixel values in the matrix representation of the PSF based on where coordinate falls on the PSF's Cartesian plane representation.

More formally, given a point (x', y') in a Cartesian plane, its point of origin (the pixel center point (x, y) that (x', y') sits closest to), and the pixel size 1×1; where that continuous point will lie can be computed if the Cartesian plane is discretized into an $H \times W$ "image," (where each pixel is of size 1×1), by Algorithm 2:

Algorithm 2 The conversion of a Cartesian plane coordinate into pixel values

1. Determine the displacement $(\Delta x, \Delta y)$ of the point (x', y') from the point of origin (x, y).
2. Based on the sign of $(\Delta x, \Delta y)$, either negative or positive, determine the direction of the neighboring pixels (i.e., north, northeast, east, southeast, south, southwest, west, and northwest) that are "encroached" by the psuedo-pixel (the pixel projected by the coordinate; that is, if the coordinate lies in the point of origin, the psuedo-pixel aligns perfectly with the actual pixel) of (x', y').
3. For each affected neighboring pixels, determine the area that the psuedo-pixel of (x', y') encroaches by the following equation:

$$A_{\text{encroached}} = \begin{cases} |\Delta x| \left(1 - |\Delta y|\right), & \text{encroached pixel is in the west or east of the origin pixel;} \\ |\Delta y| \left(1 - |\Delta x|\right), & \text{encroached pixel is in the north or south of the origin pixel} \\ |\Delta x| \, |\Delta x|, & \text{otherwise} \end{cases} \tag{36}$$

Let K be the matrix representation of the PSF:

$$K = \sum_{\forall z \in Z} \Phi\left(\vec{r}(z)\right), \tag{37}$$

where $\vec{r}(z)$ denotes the vector-valued function defined in Eq. 35 that is the Cartesian-represented PSF. The values of the matrix-represented PSF, owing to the natural requirement of post-convolution brightness retention, has to be normalized so that the matrix elements sum up to 1,

$$\tilde{K} = \frac{K}{\vec{1}^{\top} K \, \vec{1}}, \tag{38}$$

where $\vec{1}$ is a column vector of ones $[1, 1, \ldots, 1]^{\top}$. The expression $\vec{1}^{\top} K \, \vec{1}$ denotes the sum of the elements of K. Here, the function $\Phi : \mathbb{R}^3 \to \{0, 1\}^{N \times M}$ is defined as

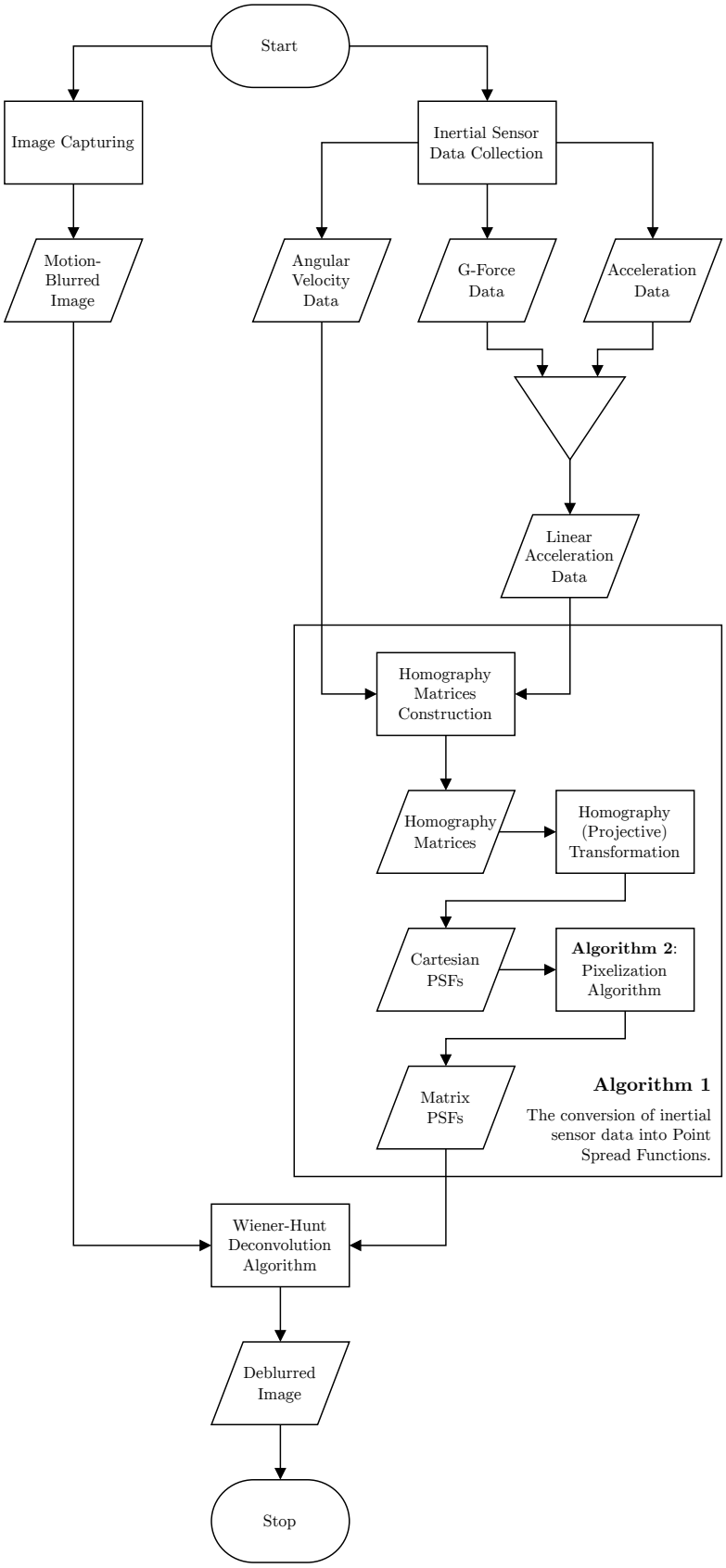


Figure 6. A diagram showing the basic outline of the proposed software pipeline.

$$\Phi\left(\begin{bmatrix} x \\ y \\ z \end{bmatrix}\right) \coloneqq \begin{bmatrix} \ddots & \vdots & \vdots & \vdots & \iddots \\ \cdots & \begin{cases} -\Delta x \Delta y, & \Delta x < 0 \text{ and } \Delta y > 0, \\ 0, & \text{otherwise;} \end{cases} & \begin{cases} \Delta y \left(1 - |\Delta x|\right), & \Delta y > 0, \\ 0, & \text{otherwise;} \end{cases} & \begin{cases} \Delta x \Delta y, & \Delta x > 0 \text{ and } \Delta y > 0, \\ 0, & \text{otherwise;} \end{cases} & \cdots \\ \cdots & \begin{cases} -\Delta x \left(1 - |\Delta y|\right), & x < 0, \\ 0, & \text{otherwise;} \end{cases} & 1 - \vec{1}^{\top} \Phi\left([x, y, z]^{\top}\right) \vec{1} & \begin{cases} \Delta x \left(1 - |\Delta y|\right), & x > 0, \\ 0, & \text{otherwise;} \end{cases} & \cdots \\ \cdots & \begin{cases} -\Delta x \left(-\Delta y\right), & \Delta x < 0 \text{ and } \Delta y < 0, \\ 0, & \text{otherwise;} \end{cases} & \begin{cases} -\Delta y \left(1 - |\Delta x|\right), & \Delta y < 0, \\ 0, & \text{otherwise;} \end{cases} & \begin{cases} \Delta x \left(-\Delta y\right), & \Delta x > 0 \text{ and } \Delta y < 0, \\ 0, & \text{otherwise;} \end{cases} & \cdots \\ \iddots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}_{N \times M}, \tag{39}$$

where $\Delta x = x - (\lfloor x \rfloor + 0.5)$ and $\Delta y = y - (\lfloor y \rfloor + 0.5)$. With a slight abuse of notation, the function $\Phi\left(\vec{v}\right)$ with $\vec{v} = [x, y, z]^{\top}$ is defined in terms of itself to mean that the expression $1 - \vec{1}^{\top} \Phi\left([x, y, z]^{\top}\right) \vec{1}$ denotes 1 subtracted by the sum of all the other elements within the matrix. Note that the $\lfloor \bullet \rfloor$ operator denotes an operator that takes the integer part of a floating point number.

Figure 7 below illustrates how Algorithm 2 works:

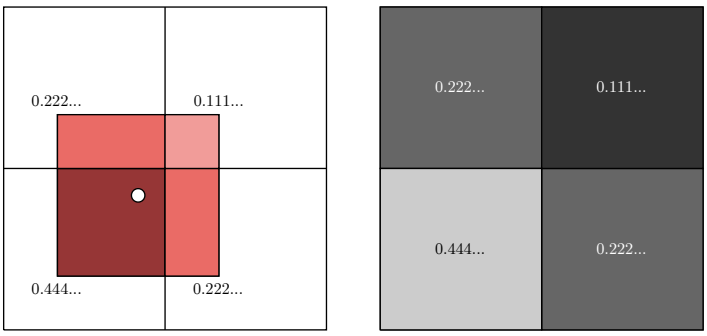


Figure 7. A visualization of how Algorithm 2 works.

It can be seen that the coordinate in the Cartesian-represented PSF, located northeast of its "origin" (left), projects its psuedo-pixel that encroaches into its neighboring grids (i.e., north, northeast, and south). In the resulting matrix representation of the PSF (right), the psuedo-pixel is distributed as discrete values in each entry of the matrix.

3.3.3 Non-Blind Deconvolution

The unsupervised Wiener-Hunt non-blind deconvolution algorithm takes in two inputs: the degraded image and an estimated of the PSF that caused the image degradation. The output of the algorithm is the restored image. The quality of the deblurring is dependent on the accuracy of the provided PSF, i.e., how structurally similar it is to the true PSF. The `scikit-image` library in Python provides an easy-to-use functional interface to this algorithm, `skimage.restoration.unsupervised_wiener()`.

Since the model of the motion blur is spatially-variant, each part of the degraded image should be deconvolved with PSFs that are specific for that part. In addition to that, as will be discussed later on, the Wiener-Hunt deconvolution algorithm, through testing, has the tendency to give better deconvolution results when the image is relatively larger in its $H \times W$ size relative to the PSF it is deconvolved against. Therefore, to maximize the deconvolution result, the pipeline deconvolves the image as many times as there are the number of PSFs and subsequently join the parts of the image that are deconvolved against the their correct offending PSFs.

3.4 Testing Scenario

To measure the difference in the Wiener-Hunt deblurring quality between the true and noisy PSFs modeled by this pipeline, the mean structural similarity (see Chapter 2.2.7) of (1) the image deblurred using the true PSF and (2) the image deblurred using the noisy PSF is computed. The mean and variance of each patch of the images are spatially-weighted using a normalized Gaussian kernel with $\sigma = 0.2$.

Additionally, to measure the accuracy of the PSF models themselves, this research provides "real" PSF samples obtained from deliberately motion-blurred images of small, bright white points on a pitch-black background. The "real" PSF samples are obtained using a camera that does not have any form of hardware image stabilization.

The qualitative features of the deblurred images are also explored. One point of interest regarding the quality of the deblurred images is how much deconvolution artifacts are present in the images after going through the deconvolution process and how complex are they are.

CHAPTER IV

RESULTS AND DISCUSSION

4.1 Methodology Implementation

4.1.1 Inertial Sensor Data Collection

The inertial sensor data for this research was collected through the embedded sensors in an Apple iPhone 13, namely its gyroscope, accelerometer, and g-force sensor. The reason for the inclusion of the g-force sensor data is so that gravitational acceleration can be subtracted from the accelerometer data to obtain linear acceleration (i.e., the actual acceleration of the device as it moves in the x, y, and z direction).

Ensuring Twice-Differentiability All inertial sensor data that has been obtained needs further processing to obtain the device's rotational and translational displacement. To do so, the discrete inertial sensor data needs to be interpolated (for continuity) and then integrated: once for the gyroscope data to convert angular velocity to angular displacement, and twice for the linear accelerometer data to obtain the amount of displacement.

According to [2], spline interpolation is a class of interpolation that ensures twice-differentiability. In Python, spline interpolation can be utilized through `scipy` library's `interpolate.UnivariateSpline` class:

Listing 1. Using the `scipy.interpolate.UnivariateSpline` class

```
gyro_x_cont = scipy.interpolate.UnivariateSpline(t, gyro_x, s=0.002307)
gyro_y_cont = scipy.interpolate.UnivariateSpline(t, gyro_y, s=0.002307)
gyro_z_cont = scipy.interpolate.UnivariateSpline(t, gyro_z, s=0.002307)

accel_x_cont = scipy.interpolate.UnivariateSpline(t, accel_x, s=0.0782)
accel_y_cont = scipy.interpolate.UnivariateSpline(t, accel_y, s=0.0782)
accel_z_cont = scipy.interpolate.UnivariateSpline(t, accel_z, s=0.0782)
```

The use of spline interpolation in obtaining a continuous representation of the discrete inertial sensor data also doubles as a noise filter. The `s` parameter in the `UnivariateSpline` class constructor refers to a smoothing factor used to determine the number of knots used in the

interpolation [21]. Assigning 0 as the value of `s` will guarantee that the resulting continuous curve passes through *all* data points. By successively increasing the value of `s` on (separately-taken) noisy stationary inertial sensor data until a flat curve is obtained, one can effectively interpolate through "genuine" data points only—ignoring noise. Through trial and error, it is found that the sensor data provided by the iPhone 13 can be effectively filtered-out by setting the smoothing factor to 0.002307 for the gyroscope spline and 0.0782 for the accelerometer spline. (The accelerometer data has more noise compared to the gyroscope data, hence the higher smoothing factor.)

Obtaining Displacement from Velocity and Acceleration The gyroscope provides data on the angular velocity of the device in each of the x, y, and z axis. Taking the first-order indefinite integral (antiderivative) of the of the angular velocity spline returns the spline representing the angular displacement. Integrating `UnivariateSpline` objects can be conveniently done by calling its `antiderivative()` method:

Listing 2. Obtaining the first-order antiderivative of the gyroscope data

```
gyro_x_antiderivative = gyro_x_cont.antiderivative()
gyro_y_antiderivative = gyro_y_cont.antiderivative()
gyro_z_antiderivative = gyro_z_cont.antiderivative()
```

On the other hand, the accelerometer data needs to be integrated twice in order to achieve translational displacement. The `antiderivative()` method can be called with an `n` parameter value of 2 in order to compute the second antiderivative of the `UnivariateSpline` object:

Listing 3. Obtaining the second-order antiderivative of the accelerometer data

```
accel_x_antiderivative = accel_x_cont.antiderivative(n=2)
accel_y_antiderivative = accel_y_cont.antiderivative(n=2)
accel_z_antiderivative = accel_z_cont.antiderivative(n=2)
```

Figure 8 and 9 illustrates the first three seconds of the gyroscope and accelerometer data splines, along with their first-order and second-order antiderivatives respectively.

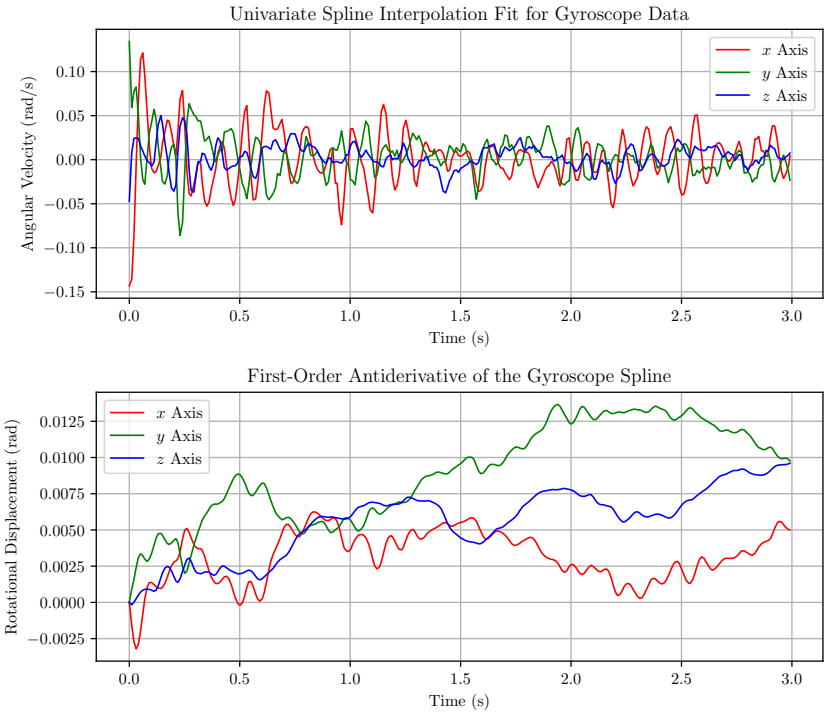


Figure 8. Gyroscope data spline and its first-order antiderivative.

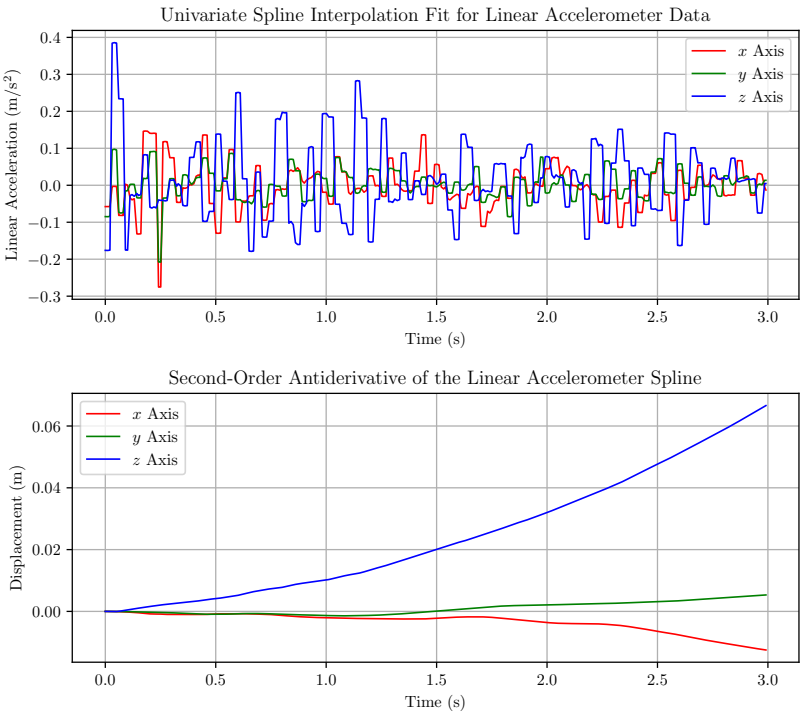


Figure 9. Linear accelerometer data spline and its second-order antiderivative.

4.1.2 Applying Homography to Estimate Point Movement

Translation Vector Constructing the translation vector is straightforward once the translational displacement data is already obtained. The translation vector for a movement that spans from t_0 (of the entire data points) to t can be obtained directly from evaluating $\delta_{\text{translation}}(t)$ at t, which is the function that was produced from integrating $a_{\text{linear}}(t)$ twice.

The following listing is a function that extracts a translation vector given the three twice-integrated linear acceleration splines (on x, y, and z)

Listing 4. Function to extract a translation vector given t (relative from t_0)

```
def translation_vector(t, x_func, y_func, z_func):
  return displacement(t, x_func, y_func, z_func)
```

where the `displacement()` function is defined as

Listing 5. Function to evaluate `UnivariateSpline` objects at t

```
def displacement(t, x_func, y_func, z_func):
  assert isinstance(x_func, scipy.interpolate.UnivariateSpline) \
    and isinstance(y_func, scipy.interpolate.UnivariateSpline) \
    and isinstance(z_func, scipy.interpolate.UnivariateSpline), \
    '`x_data`, `y_data`, and `z_data` must all be `scipy.interpolate.UnivariateSpline` objects.'
  result = numpy.array([x_func(t), y_func(t), z_func(t)])
  return result
```

Rotation Matrix The rotation matrix, $\Delta_{\text{rotation}}(t, \theta)$ is more complicated to construct. Eq. 26 detailed that the complete rotation matrix is a product of three matrices: the x, y, and z rotation matrices. The following function computes the final rotation matrix. It accepts `x`, `y`, and `z` as either the individual values of the integrated gyroscope data at time t if the `t` parameter is not provided or the integrated splines themselves at all three axes if `t` is given a value:

Listing 6. Function to construct the rotation matrix

```
def rotation_matrix(x, y, z, t=None):
  if isinstance(x, scipy.interpolate.UnivariateSpline) \
    and isinstance(y, scipy.interpolate.UnivariateSpline) \
    and isinstance(z, scipy.interpolate.UnivariateSpline):
```

```
    assert t is not None, \
      '`t` cannot be empty if `x`, `y`, and `z` are instances of `scipy.interpolate.UnivariateSpline`.'
    x_eval, y_eval, z_eval = displacement(t, x, y, z)
  else:
    x_eval, y_eval, z_eval = x, y, z
  matrix_x = numpy.array([[numpy.cos(x_eval), -numpy.sin(x_eval), 0],
                          [numpy.sin(x_eval), numpy.cos(x_eval),  0],
                          [0,              0,               1]])
  matrix_y = numpy.array([[numpy.cos(y_eval),  0, numpy.sin(y_eval)],
                          [0,               1, 0             ],
                          [-numpy.sin(y_eval), 0, numpy.cos(y_eval)]])
  matrix_z = numpy.array([[1, 0,              0              ],
                          [0, numpy.cos(z_eval), -numpy.sin(z_eval)],
                          [0, numpy.sin(z_eval), numpy.cos(z_eval) ]])
  matrix = matrix_x @ matrix_y @ matrix_z
  return matrix
```

Homography Matrix Eq. 25 detailed the construction of the homography matrix **H**. The function below computes the homography matrix based on a parameterized focal length (in millimeter), pixel size (in millimeter), distance between the image sensor and the object of interest (in millimeter), and the height and width of the image (in pixels):

Listing 7. Function to combine the computed rotation matrix and translation vector to generate a homography matrix

```
def homography(focal_length,
               pixel_size: tuple,
               distance,
               image_shape: tuple,
               gyro_x, gyro_y, gyro_z,
               accel_x, accel_y, accel_z,
               natural=numpy.array([0, 0, 1]),
               t=None):
  if distance <= focal_length:
    R = rotation_matrix(gyro_x, gyro_y, gyro_z, t=t) \
        - ((translation_vector(t, accel_x, accel_y, accel_z) \
        if t != None else numpy.array([accel_x, accel_y, accel_z]) @ natural) \
        / (distance / pixel_size[0]))
  else:
    R = rotation_matrix(gyro_x, gyro_y, gyro_z, t=t)
```

```
    f_x = focal_length / pixel_size[0]
    f_y = focal_length / pixel_size[1]
    c_x = image_shape[0] / 2
    c_y = image_shape[1] / 2
    K = numpy.array([[f_x, 0,   c_x],
                     [0,   f_y, c_y],
                     [0,   0,   1  ]])
    H = K @ R @ numpy.linalg.inv(K)
    return H
```

Adhering to Eq. 28, the function only considers using the translation vector information if the distance is less than or equal to the focal length of the camera.

4.1.3 Generating Cartesian-Represented Point Spread Functions

The following listing is a class that represents an interval cut from the integrated angular velocity spline and twice-integrated acceleration splines in all three axes:

Listing 8. An `Interval` helper class to represent an interval cut from the inertial sensor data

```
class Interval:
  def __init__(self, start, end, resolution=1000):
    assert start < end, '`start` must be less than `end`.'
    self.start = start
    self.end = end
    self.num = math.ceil((end - start) * resolution)
  def __str__(self):
    return f'Interval({self.start}, {self.end})'
  def __repr__(self):
    return self.__str__()
  def apply(self, *args, noise=None, upper_bound=None):
    for arg in args:
      assert isinstance(arg, scipy.interpolate.UnivariateSpline), \
        str(arg) + ' is not an instance of `scipy.interpolate.UnivariateSpline`.'
    if noise is not None:
      assert isinstance(noise, float) or isinstance(noise, int), \
        '`noise` must be an instance of `float` or `int`.'
      assert upper_bound is not None, \
        '`upper_bound` must also be defined if `noise` is defined.'
      t = list()
      for f in args:
```

```
        rand = random.uniform(-noise, noise)
        # Check if index shifted by the random value will be valid
        if (self.start + noise < 0) or (self.end + noise > upper_bound):
          rand = -rand
        t.append(numpy.linspace(self.start + rand, self.end + rand, num=self.num))
      t.append(numpy.linspace(self.start, self.end, num=self.num))
    else:
      t = numpy.linspace(self.start, self.end, num=self.num)
    evaluated = list()
    for i in range(len(args)):
      f_0 = args[i](self.start)
      f_eval = args[i](t) if noise is None else args[i](t[i])
      f_eval_normalized = [j - f_0 for j in f_eval]
      evaluated.append(f_eval_normalized)
    return t if noise is None else t[-1], *evaluated
```

The object on its own is a placeholder for the actual interval, which is cut from the splines and returned when the `apply()` method is called. The method itself also accepts a `noise` parameter that allows for inaccurate interval cutting, which later, will enable the simulation of noisy PSF capturing. The actual returned value of the method is the values of each splines between `start` and `end` with a sampling rate defined by the `resolution` parameter in the object's initialization.

The `Interval` class is used to conveniently define the interval from the splines to use when generating the continuous PSF with the `generate_psf()` function below:

Listing 9. Function to generate the continuous PSF

```
def generate_psf(interval: Interval,
                 focal_length: float,
                 pixel_size: tuple,
                 distance: float,
                 image_shape: tuple,
                 gyro_x_func, gyro_y_func, gyro_z_func,
                 accel_x_func, accel_y_func, accel_z_func,
                 patches: tuple,
                 natural=numpy.array([0, 0, 1]),
                 noise=None,
                 upper_bound=None):
  assert isinstance(interval, Interval), \
```

```
    'Input `interval` is not an instance of the `Interval` object.'
  t, gyro_x, gyro_y, gyro_z, accel_x, accel_y, accel_z
    = interval.apply(gyro_x_func, gyro_y_func, gyro_z_func,
                     accel_x_func, accel_y_func, accel_z_func,
                     noise=noise, upper_bound=upper_bound)
  patch_width = image_shape[0] / patches[0]
  patch_height = image_shape[1] / patches[1]
  centers = list()
  for i in range(patches[0]):
    for j in range(patches[1]):
      center_x = (j + 0.5) * patch_width
      center_y = (i + 0.5) * patch_height
      centers.append(numpy.array([center_x, center_y]))
  psf = list()
  for i in range(len(t)):
    H = homography(focal_length,
                   pixel_size,
                   distance,
                   image_shape,
                   gyro_x[i], gyro_y[i], gyro_z[i],
                   accel_x[i], accel_y[i], accel_z[i])
    # Compute transformations for each points
    psf.append(list())
    for j in centers:
      psf[i].append(transform(H, j))
  complete_psf = numpy.array([[numpy.append(y, t[i]) for y in x] \
    for i, x in enumerate(psf)]).transpose((1, 0, 2))
  return complete_psf
```

The return value of the `generate_psf()` function is a list of $n \times 3$ matrix where each column represents an axis (n is the number of samples of the splines value that was taken by the `Interval` object). The z axis represents time, while x and y represents a coordinate in the Cartesian plane at time z. This (x, y) coordinate denotes where a point in the original image frame (that is, the coordinate at z_0) have moved to at z_n. This PSF is noted as continuous because of the nature of coordinates in a Cartesian plane. (The z axis, however, is discrete as it is the result of the `Interval` class' sampling from the splines.)

The `generate_psf()` function also accepts a `patches` tuple of two that defines how the target image is divided when its corresponding PSF is computed. The product of the two

numbers in the tuple determines the number of grids the image is divided into, with one PSF computed for each grid. This allows for the computation of PSFs that are spatially-variant, i.e., different for parts of the image. The accuracy of the spatial variance can be determined freely through manipulating the number passed into the `patches` parameter.

The following is an example usage of the `generate_psf()` function, where the translational and rotational displacement data provided by the splines are taken from $t_a = 21$ and $t_b = 21.2$ (0.2 seconds in total):

Listing 10. An example usage of the `generate_psf()` function

```
psf = generate_psf(Interval(21, 21.2),
                   26,
                   (0.0017, 0.0017),
                   30,
                   (4032, 3024),
                   gyro_x_antiderivative, gyro_y_antiderivative, gyro_z_antiderivative,
                   accel_x_antiderivative, accel_y_antiderivative, accel_z_antiderivative,
                   (4, 3))
```

The other parameters, namely the focal length, the pixel size, and the image shape are taken from the iPhone 13's specification. The grid size is set to 4×3 meaning that there is a total of 12 PSFs generated.

Figure 10 and 11 illustrates the PSF that was computed for each of the twelve sections within the image; in 3-dimensional space, where the z axis indicates time (s); and in 2-dimensional space, where the z axis is hidden, showing only the movement in x and y.

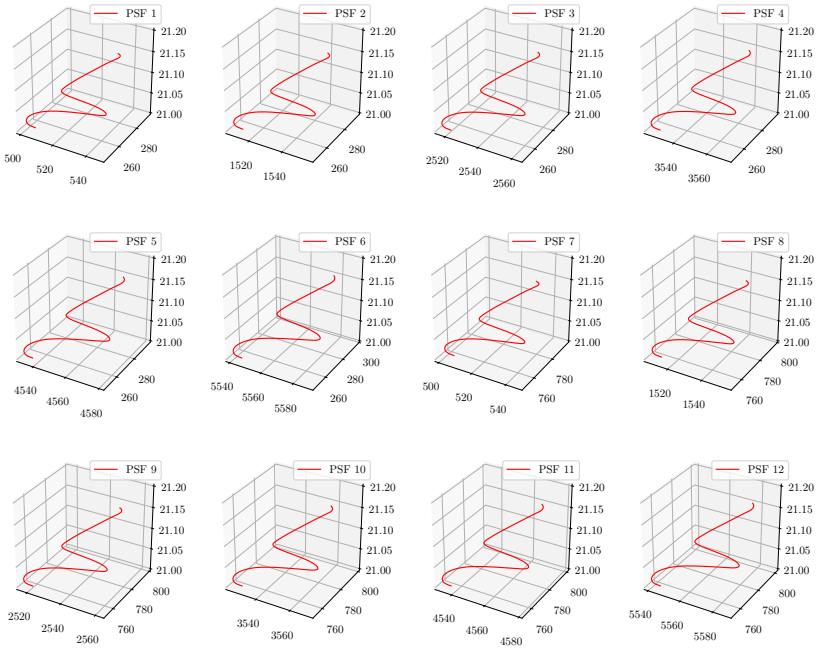


Figure 10. Cartesian-Represented PSF plotted in 3-dimensional space, with z axis as time.

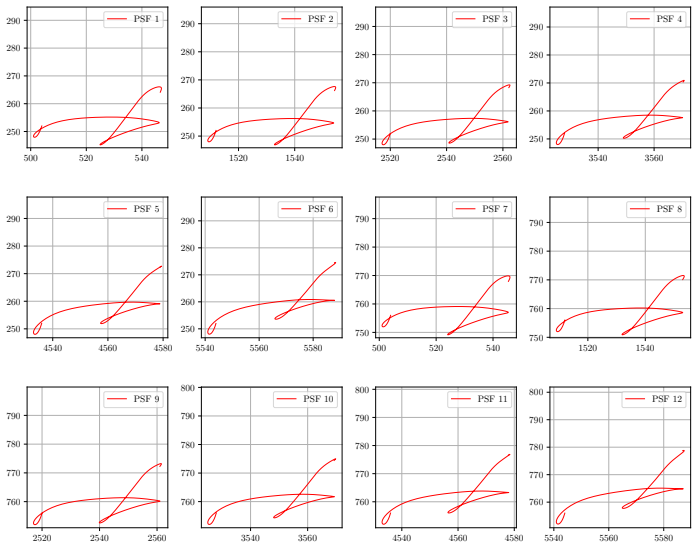


Figure 11. Cartesian-Represented PSF plotted in 2-dimensional space, omitting the z.

4.1.4 Converting Cartesian-Represented Point Spread Functions into their Matrix Representations

The listing below is the Python implementation of Algorithm 2 into the function `pixelate()`:

Listing 11. Python implementation of Algorithm 2

```
def pixelate(x, y, x_origin, y_origin):
  x_disp = x - x_origin
  y_disp = y - y_origin
  encroached = dict()
  if x_disp > 0 and y_disp > 0:
    encroached.update({'north': None, 'northeast': None, 'east': None})
  elif x_disp > 0 and y_disp < 0:
    encroached.update({'east': None, 'southeast': None, 'south': None})
  elif x_disp < 0 and y_disp < 0:
    encroached.update({'south': None, 'southwest': None, 'west': None})
  elif x_disp < 0 and y_disp > 0:
    encroached.update({'west': None, 'northwest': None, 'north': None})
  elif x_disp > 0 and y_disp == 0:
    encroached['east'] = None
  elif x_disp < 0 and y_disp == 0:
    encroached['west'] = None
  elif x_disp == 0 and y_disp > 0:
    encroached['north'] = None
  elif x_disp == 0 and y_disp < 0:
    encroached['south'] = None
  else:
    pass
  for pixel in encroached:
    if pixel == 'west' or pixel == 'east':
      encroached[pixel] = abs(x_disp) * (1 - abs(y_disp))
    elif pixel == 'north' or pixel == 'south':
      encroached[pixel] = abs(y_disp) * (1 - abs(x_disp))
    else:
      encroached[pixel] = abs(x_disp) * abs(y_disp)
  encroached['self'] = 1 - sum(encroached.values())
  return encroached
```

The function above is meant to be applied to a coordinate in the Cartesian plane, where each 1×1 grid represents a single pixel. It returns a dictionary detailing how much value should

each pixel in the PSF's discrete representation be given relative to the origin pixel.

The function below, `pixelate_psf()` applies the `pixelize()` function to a continuous PSF generated by `generate_psf()`:

Listing 12. Function to apply Algorithm 2 to continuous PSFs

```
def pixelate_psf(psf):
  # Figure out the minimum and maximum values of x and y
  x_min = min(psf[:, :1].flatten())
  x_max = max(psf[:, :1].flatten())
  y_min = min(psf[:, 1:2].flatten())
  y_max = max(psf[:, 1:2].flatten())
  # Normalize all x and y values
  x_all = psf[:, :1] - x_min
  y_all = psf[:, 1:2] - y_min
  x_bound_original = x_max - x_min
  y_bound_original = y_max - y_min
  # Round up `x_bound_original` and `y_bound_original` and shift all x and y
  # values accordingly
  x_bound = math.ceil(x_bound_original)
  y_bound = math.ceil(y_bound_original)
  x_all += 0.5 * (x_bound - x_bound_original)
  y_all += 0.5 * (y_bound - y_bound_original)
  # If `x_bound` and/or `y_bound` are even, make them odd and shift all x and
  # y values accordingly
  if x_bound % 2 == 0:
    x_bound += 1
    x_all += 0.5
  if y_bound % 2 == 0:
    y_bound += 1
    y_all += 0.5
  # Add a 1-padding on all sides (meaning 2-padding on x and 2-padding on y)
  # and shift all x and y values accordingly
  x_bound += 2
  x_all += 1
  y_bound += 2
  y_all += 1
  # Make the x and y axis equal and shift all x and y values
  # Since both `max_dimension` and `min_dimension` are odd, the difference will be even
  dimension_difference = x_bound - y_bound
  if dimension_difference > 0: # x is larger
```

```
  y_bound += abs(dimension_difference)
  y_all += 0.5 * abs(dimension_difference)
elif dimension_difference < 0: # y is larger
  x_bound += abs(dimension_difference)
  x_all += 0.5 * abs(dimension_difference)
# Construct empty matrix
matrix = numpy.zeros((y_bound, x_bound))
# Combine `x_all` and `y_all`
points = numpy.concatenate((x_all, y_all), axis=1)
for point in points:
  x, y = point
  x_origin, y_origin = find_origin(x, y)
  area = pixelate(x, y, x_origin, y_origin)
  origin_index_row = math.floor(y)
  origin_index_col = math.floor(x)
  # Because of the difference in `numpy.ndarray` indexing and Cartesian
  # indexing, also factoring in the fact that the matrix will be flipped
  # later, north here becomes the pixel below while west remains west
  for k in area:
    if k == 'north':
      matrix[origin_index_row + 1, origin_index_col] += area[k]
    elif k == 'northeast':
      matrix[origin_index_row + 1, origin_index_col + 1] += area[k]
    elif k == 'east':
      matrix[origin_index_row, origin_index_col + 1] += area[k]
    elif k == 'southeast':
      matrix[origin_index_row - 1, origin_index_col + 1] += area[k]
    elif k == 'south':
      matrix[origin_index_row - 1, origin_index_col] += area[k]
    elif k == 'southwest':
      matrix[origin_index_row - 1, origin_index_col - 1] += area[k]
    elif k == 'west':
      matrix[origin_index_row, origin_index_col - 1] += area[k]
    elif k == 'northwest':
      matrix[origin_index_row + 1, origin_index_col - 1] += area[k]
    elif k == 'self':
      matrix[origin_index_row, origin_index_col] += area[k]
flipped_matrix = numpy.flip(matrix, 0)
normalized_flipped_matrix = flipped_matrix / numpy.sum(flipped_matrix)
return normalized_flipped_matrix
```

Due to its design, the `pixelate_psf()` function, in creating the discrete PSF, will assign higher values to pixels with more coordinates that falls in its Cartesian plane representation. When the capturing device's image frame moves in 3-dimensional space, it does not move with a constant velocity. Slower movement will result in a denser sampling from the `Interval` object's `apply()` method, which translates to brighter regions in the discrete PSF.

Additionally, in order to preserve brightness of the image the PSF is going to be convolved against, the `pixelate_psf()` function normalizes the PSF matrix values in a way that causes them to sum into 1 (see Eq. 38).

Figure 12 shows the discrete version of the PSFs shown in Figure 10 and 11.

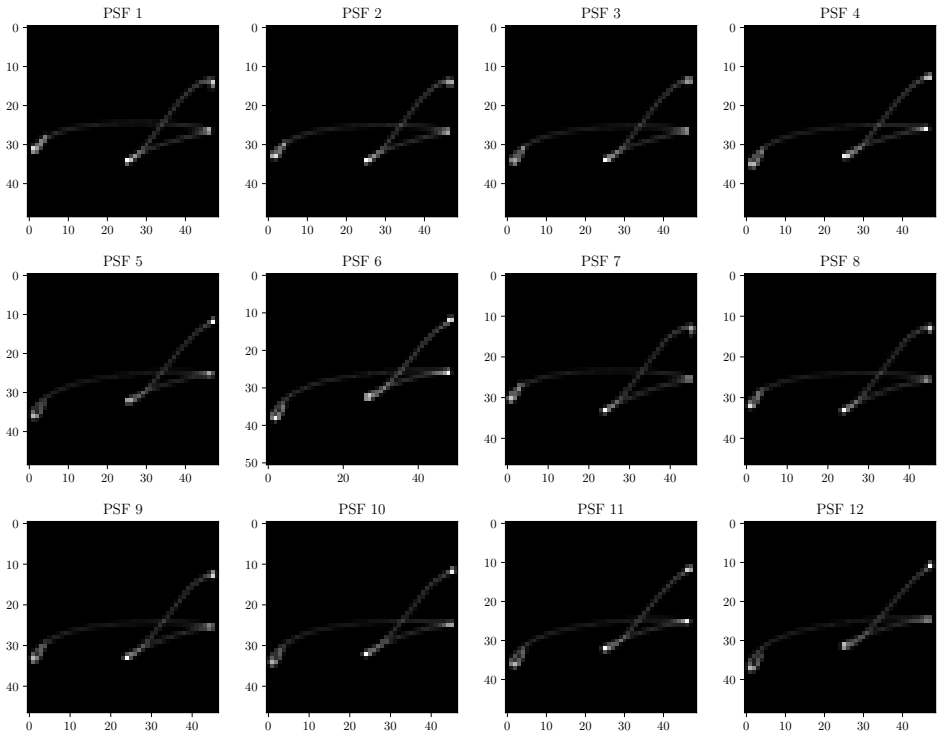


Figure 12. Discrete PSF plotted as an image, where black means zero values.

4.1.5 Realistic Modeling of Motion-Blurred Images

The discrete PSFs produced by `pixelate_psf()` is a model of how pixels within an image is "spread" throughout the area around it as the image frame of the capturing device moves

(during the time the image sensor is exposed to light). Performing convolution on an arbitrary image of a size for which the PSFs were designed for (through the `image_shape` tuple in the parameter of the `homograpy()` function) will yield in a motion-blurred image that reflects what an image captured by the moving device's camera (assuming the capturing device's principal point is at the center of the image sensor) with a shutter speed roughly equal to the size of the PSF sampling interval (`Interval.apply()`) would look like. The following code listing shows the use of the `cv2` library's `filter2D()` function to convolve an image against a kernel (PSF):

Listing 13. Convolving an RGB image against a PSF

```
image = cv2.imread('./original.png')
convolved_image = cv2.filter2D(image, -1, cv2.flip(psf_matrix, -1))
```

Because the `filter2D()` function performs correlation (otherwise known as filtering), i.e.—convolution where the kernel is not mirrored against its anchor point—the kernel would have to be flipped first by 180° counter-clockwise before passing it to the `filter2D` function.

However, as mentioned by Eq. 2, a realistic modeling of an image with a natural motion blur involves an (often) Gaussian additive noise that is independent at every pixel of the image. The following code snippet generates an $H \times W \times 3$ tensor of noise sampled from a standard normal Gaussian distribution and adds it to the image tensor:

Listing 14. Adding additive Gaussian noise to the motion-blurred image

```
gaussian_additive = numpy.random.normal(0, 1, (convolved_image.shape[0],
                                                convolved_image.shape[1],
                                                convolved_image.shape[2]))
gaussian_additive = gaussian_additive.reshape(convolved_image.shape[0],
                                              convolved_image.shape[1],
                                              convolved_image.shape[2])
gaussian_additive = numpy.rint(gaussian_additive)
noisy_convolved_image = convolved_image + gaussian_additive
noisy_convolved_image = numpy.clip(noisy_convolved_image, 0, 255).astype(numpy.uint8)
```

The resulting degraded image will faithfully replicate the motion blur model provided in Eq. 2. This can be seen from comparing the PSF models to "real" PSFs obtained through photographing small, bright white points (with deliberate motion blur) against a pitch black-

background.

4.1.6 Removing Motion Blur from Images

With consumer grade inertial sensors, noise is an unavoidable factor that one has to consider when modeling PSFs. With sampling short intervals from inertial sensors, issues such as drift are negligible. During the fitting of the spline curve on the raw data points, the application of smoothing factors also contributes to noise reduction. However, there is the possibility of noise caused by sensor reading delay. (It is understandable for the motion sensors of consumer-grade devices to have this property, since they are mainly used for purposes that does not require pin-point precision, such as gaming controls.) Hence, with such inertial sensors, to a certain degree, discrepancies exist between an image's and estimated PSFs. This, in turns, suggest that in a realistic situation, the PSF estimation fed to the non-blind deconvolution algorithm alongside the degraded image has a degree noise caused by sensor reading delay.

To model the noise caused by sensor reading delay, normally, a Gaussian probability distribution would be used. However, in the case where the parameters that is required to build the Gaussian probability density function (PDF) is not given, i.e., the mean (μ) and the standard deviation (σ), a crude approach using a uniformly-distributed PDF can be used instead. The uniform distribution's two parameters (θ_1 and θ_2) can be adjusted manually in a way that does not cause the resulting noisy PSF to deviate too far from the true PSF. (A Gaussian distribution with $\mu = 0$ and a manually-adjusted σ can also be used. However, $\mu = 0$ will cause a large portion of the resulting noisy PSF to not deviate from the true PSF at all.) The uniform distribution's θ_1 and θ_2 parameters can be passed on to the `Interval.apply()` method's `noise` parameter described earlier in order to generate a noisy PSF.

The code listing below shows how the unsupervised version of the Wiener-Hunt algorithm would be applied to a motion-blurred image with a (likely noisy) PSF estimation:

Listing 15. Using the unsupervised version of the Wiener-Hunt deconvolution algorithm

```
deconvolved_image = numpy.zeros(noisy_convolved_image.shape)
```

```
for i in range(3):
  deconvolved_image[:, :, i], _
    = skimage.restoration.unsupervised_wiener(noisy_convolved_image[:, :, i] / 255,
                                              psf_estimation)
  deconvolved_image = numpy.clip(deconvolved_image, 0, 1)
```

This applies the non-blind deconvolution algorithm to all three color channels of the degraded image. (The input image is also normalized so that its individual pixel values lie in the interval $[0, 1]$ as required by the `skimage.restoration` module.)

Figure 13, 14, 15, and 16 shows the difference between deconvolving using true and noisy PSFs.

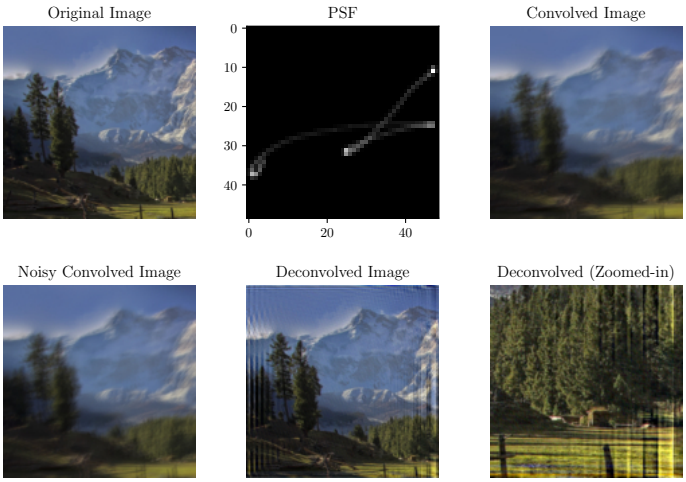


Figure 13. Convolving an image and deconvolving it with its true PSF (example 1).

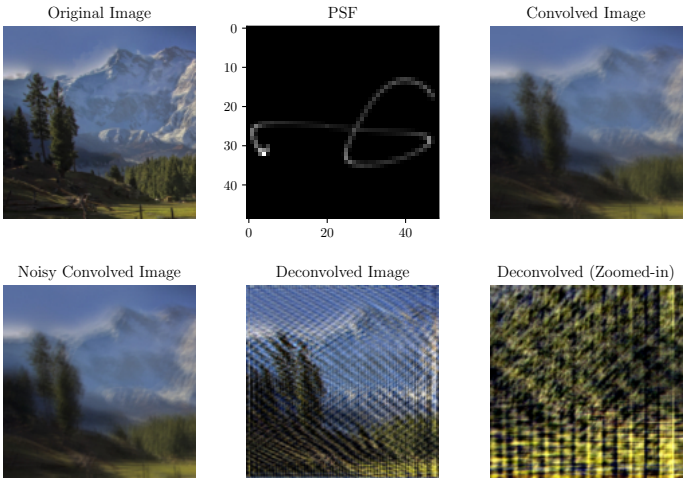


Figure 14. Convolving an image and deconvolving it with a noisy estimate PSF (example 1).

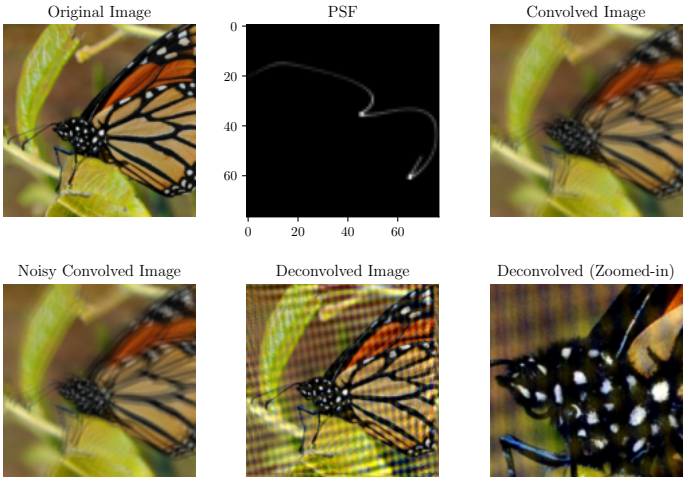


Figure 15. Convolving an image and deconvolving it with its true PSF (example 2).

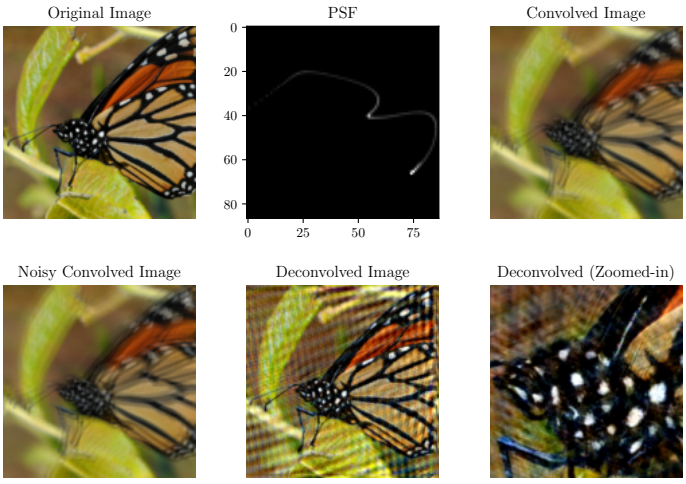
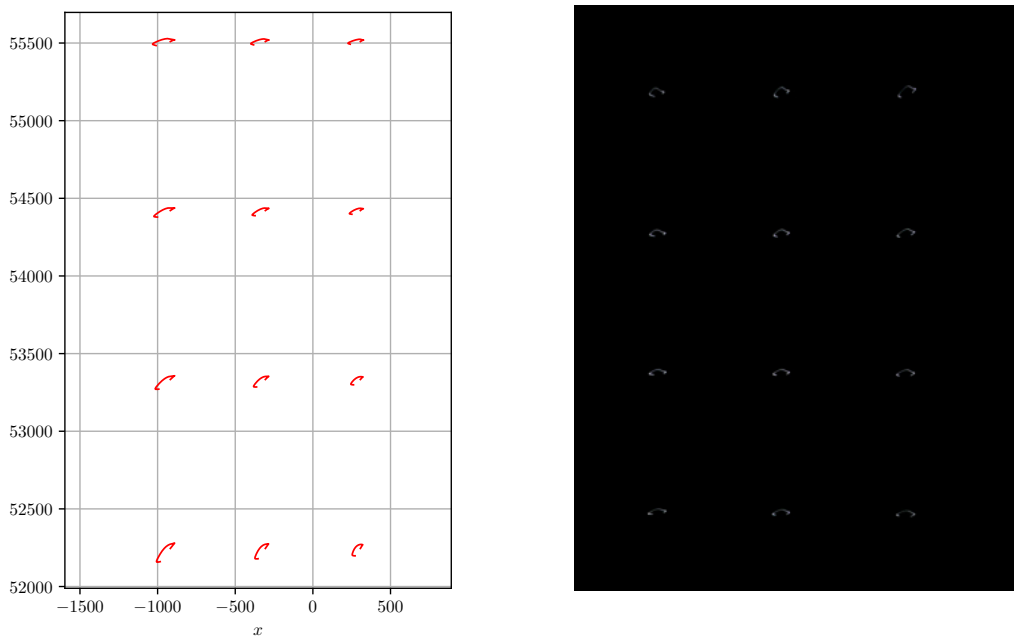


Figure 16. Convolving an image and deconvolving it with a noisy estimate PSF (example 2).

4.2 Results

4.2.1 PSF Model Accuracy

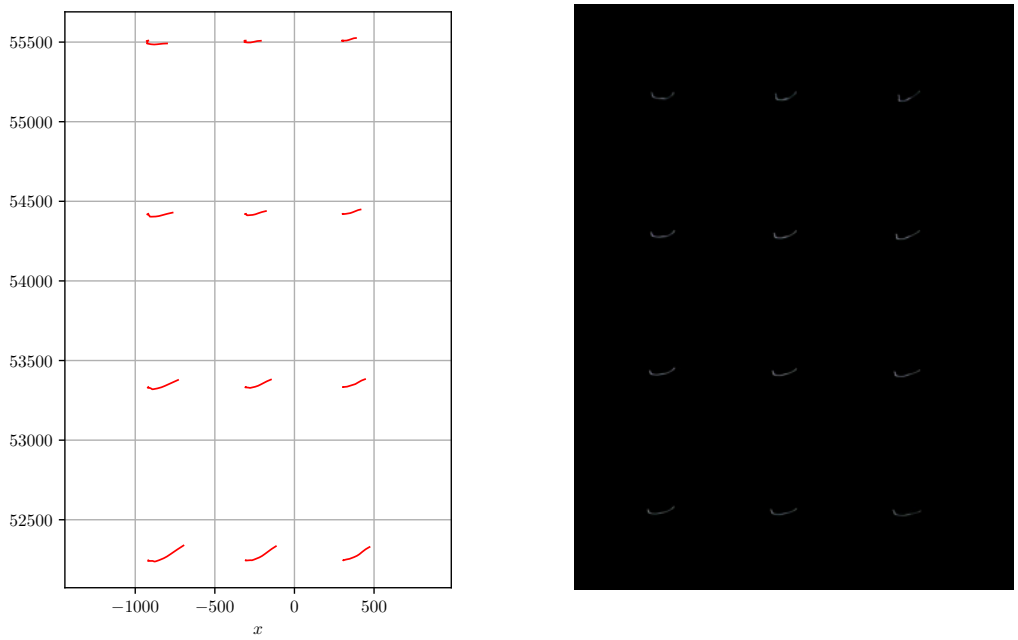
The figures 17 and 18 shows the similarity between the model PSFs generated by Algorithm 1 and the "real" PSFs captured through a camera, as mentioned in Chapter 3.4:



(a) PSF models.

(b) "Real" PSFs.

Figure 17. Sample 1 illustrating the similarity between model PSFs and "real" PSFs.



(a) PSF models.

(b) "Real" PSFs.

Figure 18. Sample 2 illustrating the similarity between model PSFs and "real" PSFs.

Through the use of the SSIM metric detailed in Chapter 2.2.7, it is found that the average model PSFs have an average similarity of 0.901 (a SSIM score of 1 would indicate a perfect match while -1 means that the compared images does not share any kind similarity).

4.2.2 Deblurring Results with Increasingly Noisy PSF

The following figures, 19, 20, 21, 22, and 23 shows deconvolution results using noisy PSF estimates that are increasingly dissimilar from the true PSF:

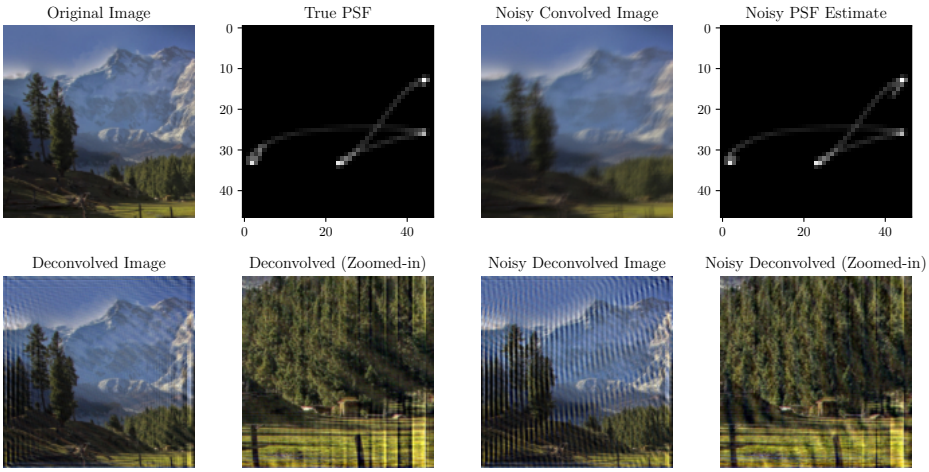


Figure 19. Difference in deconvolution results using a true PSF and a noisy PSF with a noise factor fixed at 0.01 seconds.

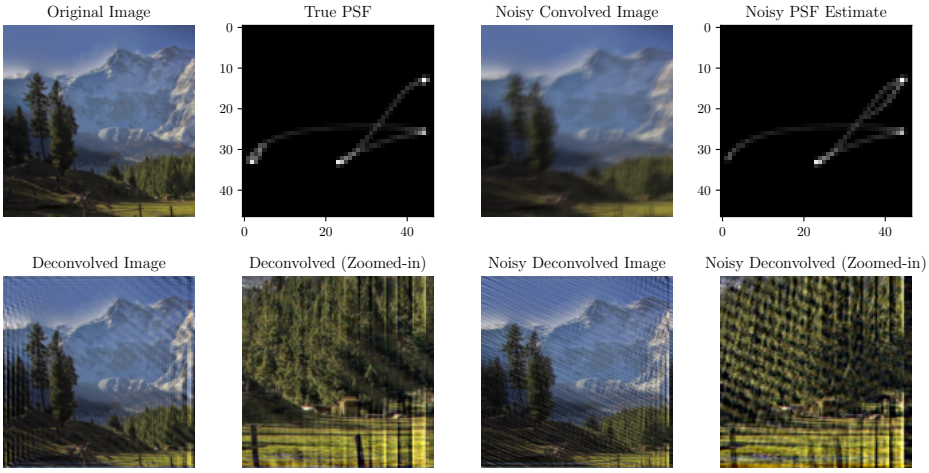


Figure 20. Difference in deconvolution results using a true PSF and a noisy PSF with a noise factor fixed at 0.025 seconds.

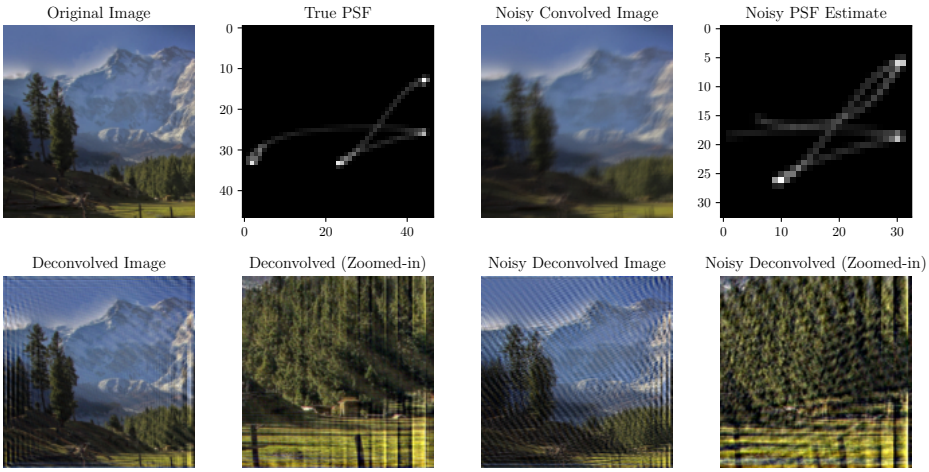


Image Structural Similarity: 0.830399988472709, PSF Structural Similarity: 0.8619838899216408, PSF Noise: 0.05

Figure 21. Difference in deconvolution results using a true PSF and a noisy PSF with a noise factor fixed at 0.05 seconds.

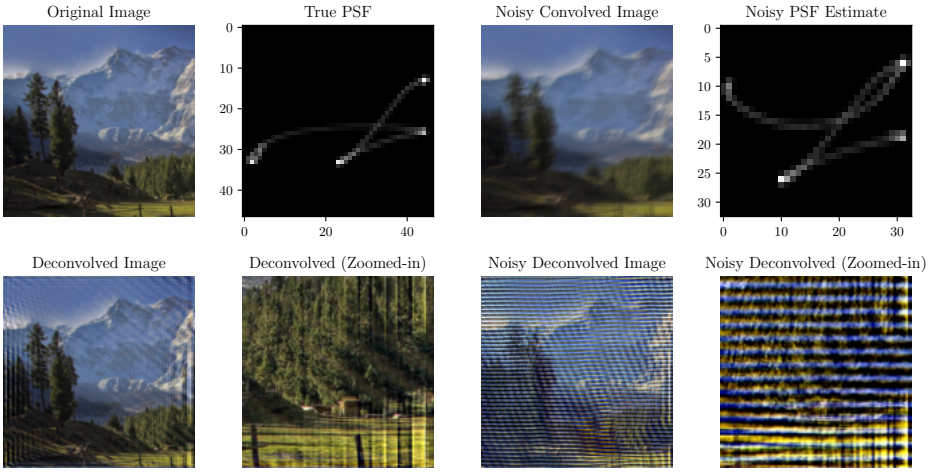


Image Structural Similarity: 0.5438606211479362, PSF Structural Similarity: 0.8495355369654515, PSF Noise: 0.075

Figure 22. Difference in deconvolution results using a true PSF and a noisy PSF with a noise factor fixed at 0.075 seconds.

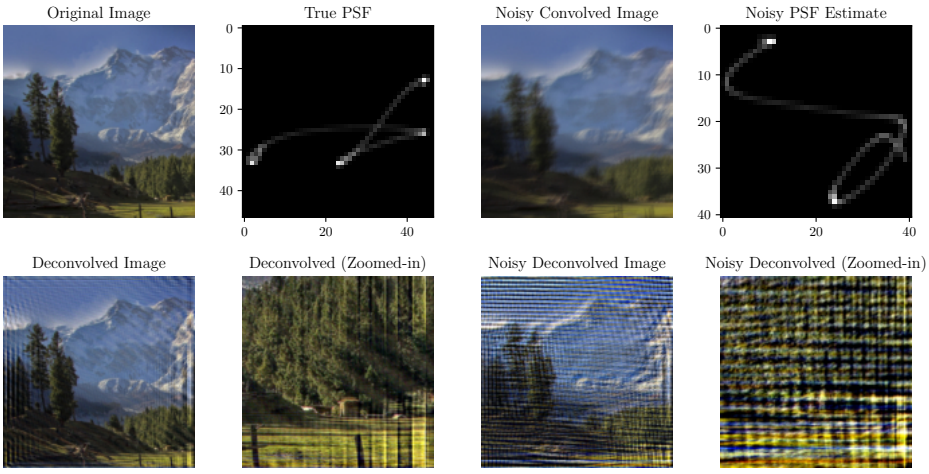


Figure 23. Difference in deconvolution results using a true PSF and a noisy PSF with a noise factor fixed at 0.5 seconds.

4.2.3 Deconvolution Results with Spatially-Variant PSFs

The figures 24, 25, and 26 shows the result of deconvolution when the image is degraded by a set of PSFs that are spatially-variant:

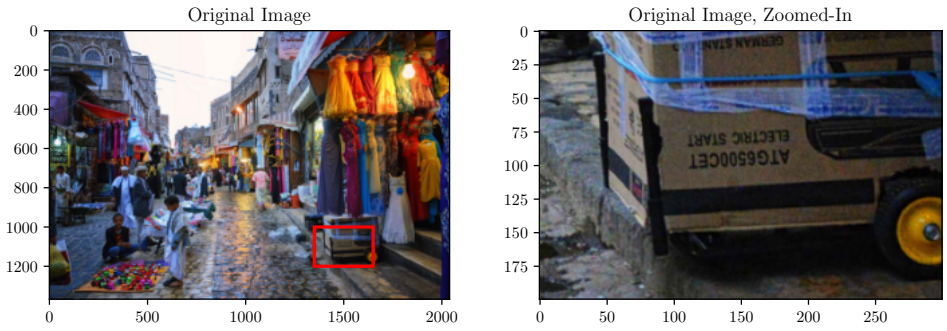


Figure 24. The original image before a spatially-variant motion blur degradation.

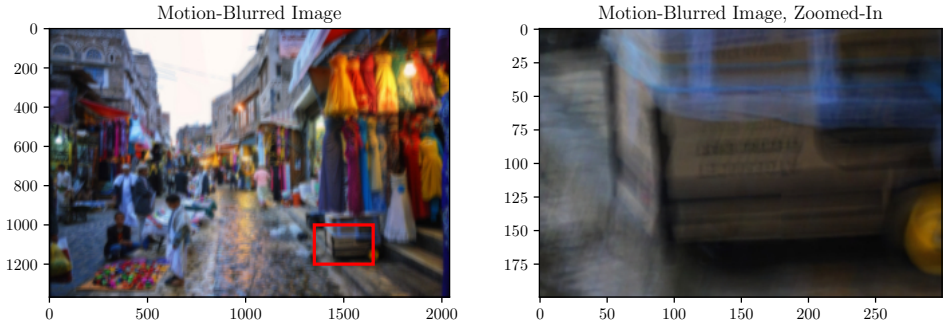


Figure 25. The spatially-variant motion-blurred image.

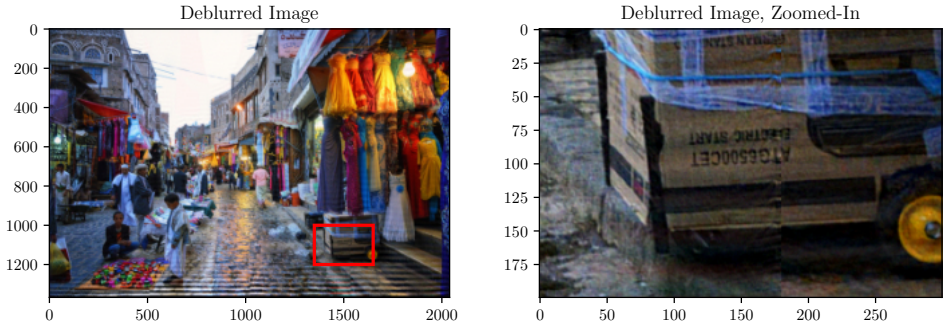


Figure 26. The result of the spatially-variant deconvolution.

The PSF models for the image above was computed with a 3×4 ($H \times W$) grid through the information provided from motion sensors. The image itself was deconvolved twelve ($3 \times 4 = 12$) times, after which, the accurately-deblurred parts of each resulting images are assembled to construct the final latent image.

It is also found that with an average PSF SSIM-based difference of 0.928 (between the PSF used to simulate the motion blur and the PSF used in the deconvolution process), the deblurred image and the original image produced an average SSIM score of 0.881.

4.3 Discussion

The part of this research detailing the design and implementation of the pipeline responsible for converting inertial sensor data into discrete PSFs provides a uniquely comprehensive discussion on the topic, compared to other research in the deconvolution literature such as [3], [6], [5], and [18].

In using the unsupervised Wiener-Hunt non-blind deconvolution algorithm, this research finds that as the spatial similarity between the true PSF and the noisy PSF estimate (that is used to deconvolve the image) shrinks, the spatial similarity between the images deconvolved with the true PSF and the noisy PSF estimate also drops. In Figure 19, partly because of the low noise introduced to the sampling interval, both the PSFs and the deconvolved images exhibit very high similarity to their noisy counterparts. As the spatial similarity of the PSFs get smaller, so do the spatial similarity of the deconvolved images. When a noise of 0.075 is introduced to the sampling interval, which causes the PSF spatial similarity to drop from 0.862 (when the noise was set to 0.05) to 0.849, the quality of the deblurring noticeably

degrades, dropping the deconvolved images' spatial similarity to 0.544.

However, as noted by Figure 23, this trend is not always the case.

Additionally, this research finds that the quality of the deblurring is correlated to the ratio of the degraded image size to the PSF size; as the PSF gets relatively smaller compared to the degraded image that it is deconvolved against, the quality of deblurring improves. (This is partly caused by the fact that larger PSFs mean more motion blur for the deconvolution algorithm to attempt to reverse.) In this case, the improvement of the deblurring quality mostly lies on the density of the image artifacts caused by the deconvolution. However, even with dense image artifacts, the deblurring manages to yield almost all the details that was highly latent in the motion-blurred image, especially when the PSF estimate that was passed into the deconvolution algorithm does not deviate highly from the true PSF. As the estimated PSF deviates from the true PSF, the deblurred image loses details that would otherwise be present without the deviation.

Because of this property, the deconvolution part of the pipeline convolves the *entire* image with every PSFs that were computed for it through the inertial sensor data. This is done to maximize the size ratio between the PSF and the degraded image, which in turns, minimizes the appearance of artifacts in the deblurring results.

Aside from the relative size difference between the degraded image and its corresponding PSF, the deconvolution quality also depends on the complexity of the PSF, i.e., an image motion-blurred by a linear PSF will deconvolve to a clearer image. The complexity of the kernel affects the "shapes" taken by the deconvolution artifacts, so a more complex PSF also tend to cause more complex patterns in the artifacts (which are harder to see image details through).

CHAPTER V

CONCLUSION AND RECOMMENDATIONS

5.1 Conclusion

This research highlights several critical factors influencing the quality of image deblurring when using the unsupervised Wiener-Hunt non-blind deconvolution algorithm—with PSFs estimated through the motion data–to–PSF pipeline detailed in the previous section. First, the relationship between the spatial similarity of the true and (noisy) estimated PSFs and the resulting deconvolved images is clear: as the spatial similarity between the PSF estimate and the true PSF decreases, the quality of the deconvolution also deteriorates. This is particularly evident when noise is introduced into the sampling interval, which causes both the PSFs and the deconvolved images to lose fidelity as their spatial similarity declines. The findings underscore the importance of accurate PSF estimation in non-blind deconvolution processes, with even small deviations from the true PSF having the potential to significantly affect the clarity and detail of the deconvolved images.

Additionally, this research shows that the relative size of the PSF to the degraded image plays a pivotal role in determining the deconvolution quality. As the PSF becomes smaller relative to the degraded image, the deconvolution process tends to yield clearer results, primarily due to a reduction in the density of image artifacts that partly obscures fine details. However, even in cases with high artifact density, the deblurring process can still recover substantial latent image details, especially when the PSF estimate is relatively accurate. This underscores the importance of both the size and the shape complexity of the PSF in achieving high-quality deblurring.

5.2 Recommendations

Based on the findings of this research, it is recommended that future work on image deblurring focus on enhancing PSF estimation techniques in noisy environments. Researchers should explore inertial sensor data denoising techniques that can estimate the probability

density function (PDF) of an inertial sensor's reading delay. This method can replace the uniform PDF approach that was used in this research to estimate the sensor reading delays.

Further research could also build on the end of the image deblurring pipeline provided in this research. One possible avenue to explore is the retrieval of latent images from behind the deconvolution artifacts produced by the Wiener-Hunt algorithm.

REFERENCES

[1] A. Abdelhamed, S. Lin, and M. S. Brown, "A high-quality denoising dataset for smartphone cameras," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 1692–1700.

[2] S. Nayar and M. Ben-Ezra, "Motion-based motion deblurring," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 6, pp. 689–698, 2004.

[3] J. Mustaniemi, J. Kannala, S. Särkkä, J. Matas, and J. Heikkila, "Gyroscope-aided motion deblurring with deep networks," in *2019 IEEE Winter Conference on Applications of Computer Vision (WACV)*, 2019, pp. 1914–1922.

[4] "What is Camera Calibration?" https://www.mathworks.com/help/vision/ug/camera-calibration.html, Accessed: 15 September 2024.

[5] N. Joshi, S. B. Kang, C. L. Zitnick, and R. Szeliski, "Image deblurring using inertial measurement sensors," *ACM Trans. Graph.*, vol. 29, no. 4, Jul 2010. [Online]. Available: https://doi.org/10.1145/1778765.1778767

[6] Z. Hu, L. Yuan, S. Lin, and M.-H. Yang, "Image deblurring using smartphone inertial sensors," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 1855–1864.

[7] O. Whyte, J. Sivic, and A. Zisserman, "Deblurring shaken and partially saturated images," in *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, 2011, pp. 745–752.

[8] A. Gopatoti, "Image denoising using spatial filters and image transforms: A review," *International Journal for Research in Applied Science and Engineering Technology*, vol. 6, pp. 3447–3452, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:57598681

[9] A. Goldstein and R. Fattal, "Blur-kernel estimation from spectral irregularities," in *Computer Vision – ECCV 2012*, A. Fitzgibbon, S. Lazebnik, P. Perona, Y. Sato, and C. Schmid, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 622–635.

[10] C.-M. Kao, P. L. Rivière, and X. Pan, "Chapter 6 - Basics of imaging theory and statistics," in *Emission Tomography*, M. N. Wernick and J. N. Aarsvold, Eds. San Diego: Academic Press, 2004, pp. 103–126. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780127444826500090

[11] S. W. Smith, "Chapter 17 - Custom filters," in *Digital Signal Processing*, S. W. Smith, Ed. Boston: Newnes, 2003, pp. 297–310. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780750674447500546

[12] J. M. Blackledge, "Chapter 12 - image restoration and reconstruction," in *Digital Image Processing*, ser. Woodhead Publishing Series in Electronic and Optical Materials, J. M. Blackledge, Ed. Woodhead Publishing, 2005, pp. 404–438. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9781898563495500124

[13] ——, "Chapter 2 - 2D Fourier theory," in *Digital Image Processing*, ser. Woodhead Publishing Series in Electronic and Optical Materials, J. M. Blackledge, Ed. Woodhead Publishing, 2005, pp. 30–49. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9781898563495500021

[14] M. F. Wahab and T. C. O'Haver, "Peak deconvolution with significant noise suppression and stability using a facile numerical approach in Fourier space," *Chemometrics and Intelligent Laboratory Systems*, vol. 235, p. 104759, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0169743923000096

[15] F. Orieux, J.-F. Giovannelli, and T. Rodet, "Bayesian estimation of regularization and point spread function parameters for Wiener–Hunt deconvolution," *Journal of the Optical Society of America A*, vol. 27, no. 7, p. 1593, Jun 2010. [Online]. Available: http://dx.doi.org/10.1364/JOSAA.27.001593

[16] O. Whyte, J. Sivic, A. Zisserman, and J. Ponce, "Non-uniform deblurring for shaken images," in *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2010, pp. 491–498.

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

[18] K. S. Singh, M. Diwakar, P. Singh, and D. Garg, "Inertial sensor aided motion blur kernel estimation for cooled IR detector," *Optics and Lasers in Engineering*, vol. 175, p. 108014, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0143816623005420

[19] S. Kim and M. Kim, "Rotation representations and their conversions," *IEEE Access*, vol. 11, pp. 6682–6699, 2023.

[20] Z. Wang, A. Bovik, H. Sheikh, and E. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004.

[21] "`scipy.interpolate.UnivariateSpline`," https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.UnivariateSpline.html, Accessed: 27 September 2024.