Master thesis

CAMERA SHAKE REMOVAL AND IMPLEMENTATION ON ANDROID

NOUSIAS STAVROS

Supervisor
Evangelos Zigouris
Associate Professor

UNIVERSITY OF PATRAS, PHYSICS DEPARTMENT
PATRAS, MARCH 2016

This page is intentionally left blank

Ειδική επιστημονική εργασία

ΑΦΑΙΡΕΣΗ ΤΗΣ ΘΟΛΩΣΗΣ ΣΕ ΦΩΤΟΓΡΑΦΙΚΗ ΛΗΨΗ ΠΡΟΕΡΧΟΜΕΝΗΣ ΑΠΟ ΚΟΥΝΗΜΑ ΤΗΣ ΚΑΜΕΡΑΣ ΚΑΙ ΥΛΟΠΟΙΗΣΗ ΣΤΟ ANDROID

ΝΟΥΣΙΑΣ ΣΤΑΥΡΟΣ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 08/03/2016
στα πλαίσια του ΔΠΜΣ Ηλεκτρονική και Επεξεργασία της Πληροφορίας

____________________	____________________	____________________
Ζυγούρης Ευάγγελος	Οικονόμου Γεώργιος	Φωτόπουλος Σπύρος
Αναπληρωτής	Καθηγητής	Καθηγητής
Καθηγητής	Τμήμα Φυσικής	Τμήμα Φυσικής
Τμήμα Φυσικής	Παν/μιο Πατρών	Παν/μιο Πατρών
Παν/μιο Πατρών		

ΠΑΤΡΑ , ΜΑΡΤΙΟΣ 2016

This page is intentionally left blank

Master thesis

CAMERA SHAKE REMOVAL AND IMPLEMENTATION ON ANDROID

NOUSIAS STAVROS

Approved by the examining committee at 08/03/2016 for the partial completion of the degree of the Master of Science in electronics and information processing

____________________	____________________	____________________
Zigouris Evangelos	Ekonomou Georgios	Fotopoulos Spyros
Associate Professor	Professor	Professor
Physics Department	Physics Department	Physics Department
University of Patras	University of Patras	University of Patras

PATRAS, MARCH 2016

This page is intentionally left blank

Περίληψη

Ο σκοπός αυτής της διπλωματικής είναι η μελέτη της αφαίρεσης θόλωσης προερχόμενης από κούνημα από μονή φωτογραφική λήψη. Η διάταξη που χρησιμοποιείται αφορά μια φωτογραφική μηχανή που λαμβάνει μια λήψη υπό συνθήκες που ευνοούν την δημιουργία θολωμένης εικόνας. Οι εν λόγω συνθήκες μπορεί να αφορούν χαμηλό φωτισμό ή κίνηση της κάμερας κατά τη διάρκεια της λήψης. Το αποτέλεσμα είναι τα pixels της τελικής εικόνα να υπερτίθενται δημιουργώντας ασάφεια στις λεπτομέρειες της εικόνας λόγω της αφαίρεσης των υψηλών συχνοτήτων . Η αντιμετώπιση του παραπάνω ζητήματος είναι ένα πολύ σημαντικό ζήτημα γιατί υπάρχουν πολλές περιπτώσεις που μια θολωμένη εικόνα περιέχει πολύ σημαντική πληροφορία σχετικά με εφαρμογές που αφορούν ιατρικούς σκοπούς, ερευνητικούς σκοπούς ή νομικούς λόγους ενώ δεν είναι εφικτή η επανάληψη της λήψης. Χαρακτηριστικά παραδείγματα μπορεί να συμπεριλαμβάνουν αστρονομικές εικόνες, ακτινογραφίες ή φωτογραφίες της πινακίδας ενός αυτοκινήτου. Επίσης , το ίδιο ισχύει και για φωτογραφίες της καθημερινότητας όπως φωτογραφήσεις από ένα γάμο ή κάθε είδους οικογενειακές φωτογραφίες.
Σαν είσοδο θεωρούμε μια και μόνο μια φωτογραφική λήψη χωρίς περαιτέρω λεπτομέρειες σχετικά με τη σκηνή , τους ανθρώπους ή τα αντικείμενα που φωτογραφίζονται. Ούτε σχετικά με τον εξοπλισμό ή τον τρόπο που ο χρήστης χρησιμοποίησε τον εξοπλισμό κατά τη λήψη. Επιπλέον, θεωρείται πως ο αλγόριθμος που θα προκύψει θα υλοποιηθεί σε επεξεργαστές ενσωματωμένων συστημάτων τύπου ARM και συγκεκριμένα για το λειτουργικό σύστημα ANDROID. Αυτό σημαίνει ότι επιζητούμε μια γρήγορη και όσο πιο υπολογιστικά ελαφριά υλοποίηση γίνεται. Για να αντιμετωπίσουμε τα παραπάνω ζητήματα παρουσιάζουμε τη σχετική βιβλιογραφία που αναφέρεται σε αυτό το ζήτημα και επιλέγουμε αναλόγως τη μέθοδο που βρίσκεται εντός των προδιαγραφών που έχουμε θέσει. Συγκεκριμένα , η διπλωματική εργασία ακολουθεί την εξής δομή. Το πρώτο κεφάλαιο παρουσιάζει εισαγωγικά στοιχεία. Το δεύτερο κεφάλαιο παρουσιάζει την υπάρχουσα βιβλιογραφία. Ακολούθως, το τρίτο κεφάλαιο παρουσιάζει τη μέθοδο που χρησιμοποιούμε εμείς στην υλοποίηση μας. Στο τέταρτο κεφάλαιο παρουσιάζεται η υλοποίηση στο MATLAB, και στο πέμπτο η υλοποίηση σε C/C++. Έπειτα στο έκτο κεφάλαιο ενσωματώνεται η άνω υλοποίηση στο περιβάλλον του λειτουργικού συστήματος ANDROID. Τέλος στο κεφάλαιο επτά παρουσιάζονται τα αποτελέσματα της υλοποίησης και στο κεφάλαιο οκτώ αφιερώνεται χώρος για να παρατεθούν οι ελλείψεις της εργασίας αυτής και τα ζητήματα που χρήζουν περαιτέρω ανάλυσης.

Abstract

The purpose of this thesis is to investigate the issue of removing camera shake blur from a single image. The setup of the issue under investigation includes a camera taking a picture under conditions that favor a non-sharp result. These conditions may include dim light or camera motion during a shot. The result is a superposition of pixels of the sharp image leading to a blurry image with unclear details. This is because the blur has removed the high frequency components of the taken image. Dealing with such a matter is most important since there are several cases that blurry camera shots contain very important information for research, medical or image matters and the fact is that these shots cannot be retaken. Such cases can be astronomical images, car plate images or images from medical scans and microscopy. This is also a problem in everyday life photos. For example, pictures from friends' reunion, wedding pictures or family pictures.

As input we use a single image having no other information regarding the scene the people or the objects present in the shot nor do we possess any other information regarding the equipment user or the user. Additionally, it is assumed that the method will be implemented for embedded systems processors and specifically for the ANDROID platform, meaning that a fast and lightweight implementation is in order. To cope with the aforementioned matters, we present from the current literature the most common methods and after a presenting a benchmarking along with the criteria we select the most suitable method. Specifically, this master thesis has the following structure. The second chapter presents the current literature. Afterwards in chapter three the method is analyzed and in chapter four we present the implementation of the method in MATLAB. In chapter five we present the C/C++ implementation that is used for the android implementation within an android application that reads images from the image gallery upon which the algorithm is applied. Finally, in chapter seven the evaluation and benchmarking is presented.

This page is intentionally left blank

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1
INTRODUCTION

Recently the need for more efficient and better quality imaging has emerged for many applications and research fields such as photography, medical imaging, astronomy. The developments that have occurred in these areas have offered image acquisition with higher speed and higher resolution reaching high definition standards. On the other hand, factors related to extrinsic or intrinsic sources often result in degradation. Thus, the resulting image is blurry or carries artifacts. Several examples can be mentioned:

- A high speed vehicle captured by security cameras. In this case the plate number may not be as clear as needed.
- In photography it is difficult to capture a clear sharp image especially in conditions of dim light. In this case the non-stabilization of the camera is the main cause since it is held by hand resulting in blurry image
- Degradation can also occur due to out-of-focus setting. This happens because imaging equipment has only one focus resulting in blurry regions of the image.

The aforementioned deterioration of the image can be a real problem in applications that require high quality imaging. The restoration of the original image is known as image deblurring and is very important for recovering details in high quality images. More specifically image deblurring refers to the inverse problem where the sharp image is recovered from blurry and noisy captured images.

Several techniques have been developed in order to address the issue under investigation. In most cases blur is modelled as a convolution of a sharp image and a filter also known as blur kernel or point-spread function (PSF). Image deblurring can be classified in two distinct categories: non-blind and blind. The first refers to the case that the blur kernel is known while the second to the case that the blur kernel is unknown and has to be estimated. Regarding the first case one would assume that the knowing the blur kernel is enough so as to apply a simple deconvolution and recover the sharp image. However, this is not true since blur has destroyed high frequency features meaning that they have to be estimated. Additionally, blind deblurring includes the kernel estimation meaning that after the kernel has been estimated the problem becomes a non-blind deconvolution matter.

Blind deblurring methods can also be classified in four categories: a) Bayesian inference framework related methods, b) variational methods, c) sparse representation methods and d) homography based methods. All categories have a significant presence in the latest literature and they will be thoroughly analyzed in the second chapter.

1.1 THESIS OUTLINE

This thesis presents a solution for the camera shake case. After analyzing the current state of the art, we justify our choise of deblurring algorithm to be implemented. The implementation is at first presented in the MATLAB prototyping stage, then the C++ version is analyzed and finally it is integrated in the android environment. Specifically, the thesis is organized in the following manner: The first chapter is the introductory section stating the issue under investigation and the researcher's motivation to deal with it. The second chapter contains a state of the art review with details related to background and related work on methods dealing with the problem. The third section describes the method that was used by the author of this thesis to cope with the task at hand. The fourth chapter shows the MATLAB implementation and the fifth chapter the C/C++ implementation. The sixth chapter describes the android platform architecture and presents the android implementation. The seventh chapter is dedicated to benchmarking, evaluation and shows the results of this work. Finally, the eighth chapter provides a conclusion and makes a reference to future work.

Chapter 2
BACKGROUND AND RELATED WORK

In this section the various techniques that deal with image deblurring are presented as described in Wang and Tao 2014[1] . Several techniques have been proposed classified either as non-blind or blind deblurring. In each of the aforementioned classifications a different setting is used as input for the problem. For non-blind deblurring it is assumed that the blur kernel is known while for blind deblurring techniques the blur kernel is unknown meaning that it has to be estimates. In any of the above schemes the problem is ill-posed which means that there is an infinite set of blur kernels and sharp images that can lead to the same blurry result. Thus it is required to find the optimal kernel which recovers the latent image, meaning the result that would have occurred if a tripod had been used and the light conditions were optimal. There are several methods that have been developed aiming to cope with ill-posedness. As it has already been mentioned, these can be classified in four categories:

- Bayesian inference framework
- Variational methods
- Sparse representation framework
- Homography based methods

Regarding Bayesian Inference framework priors are used to deal with ill-posedness providing information for the probable values or sets of values the sharp(latent) image or the blur kernel or both can take. Variational methods derive a unique solution by using regularization techniques. The use of these methods is similar to the use of priors in the aforementioned Bayesian inference framework. Sparse representation takes advantage of the fact that natural images are sparse in certain domains. Finally, homography based methods are more relevant to non-uniform blur. This type of blur cannot be modelled by only one kernel. Thus this problem is approached by a union of multiple kernels also known as homographies while introducing even more unknowns. Additionally, in order to cope with ill posedness several hardware modifications where introduced in several studies but this methodology lies outside the scope of this thesis

2.1 BLUR MODELLING

The point spread function or the blur kernel the causes the value of a certain pixel to be a weighted sum of the values of multiple pixels of the sharp image. There are several blur types. Each type affects the original sharp image in a different way. Thus there is a need to describe the aforementioned procedure mathematically. Specifically, it can be modelled as:

$$y = Hx + n \tag{2-1}$$

Given the type of problem regarding either blind or non-blind deblurring setting, the problem involves finding x in the non-blind case or x and h in the blind case. Using a matrix-vector form we can formulate

$$\mathbf{y} = \mathbf{Hx} + \mathbf{n} \tag{2-2}$$

Where y, x, and n are ordered columns representing y, x and n correspondingly.

2.2 NOISE MODELS

The noise usually is generated during image acquisition, processing or transmission. Additionally, it is dependent on the image capturing system and usually term n is often modeled as

- Gaussian noise
- Poisson noise
- Impulse noise

Equation (2.1) is not suitable for describing these noise cases since it only describes the noise as an added value to the convoluted sharp image assuming that the image and the noise is uncorrelated. On the other hand, Poisson and impulse noise are usually correlated to the image or signal. **Error! Reference source not found.** shows a more detailed summary of the aforementioned noise models. At this thesis we assume the noise to be Gaussian.

Table 2-1 : Noise models.

Gaussian noise	$y = x * h + n$
Poisson noise	$y = n(x * h)$

For the sake of simplicity, it can be assumed that the point spread function is spatially invariant meaning that the blur is uniform and that the blurry image is the convolution of a sharp image and blur kernel[2]–[4] . However, this does not happen in real life since variations in motion during image capture always produce non-uniform blur. Even a small rotation around the visual axis during hand-held photography can cause the blur kernel to be different at the center of rotation and different at some other point. This means that spatially variant blur describes the real case but it is hard to cope with.

2.3 BLUR TYPES

2.3.1 OBJECT MOTION BLUR

In the case of a relatively moving object in the scene during image acquisition, object motion blur is caused. This type of blur usually is created when capturing a fast-moving object or when the exposure time is long enough. If the motion is very fast relative to the exposure period, the blur effect can be approximated by a linear motion blur. Specifically, it can be described by the a 1D local averaging of pixels near the moving object.

$$h(i,j\,|\,L,\theta)=\begin{cases}\frac{1}{L}, if\ \sqrt{i^2+j^2}\le\frac{L}{2}\ and\ \frac{i}{j}=-\tan\theta\\ 0, otherwise\end{cases} \tag{2-3}$$

In Equation (2-3) representing object motion blur, (i, j) is the coordinate originating from the center of h, L the moving distance and θ the moving direction. In real world, however, motions complex enough so as not to be approximated a simple model. The best way to deal with this is to use a non-parametric model assuming that the blur kernel needs to follow the motion path. Another issue that appears in this case is that only a certain region is affected by the blur kernel while all the other regions remain untouched meaning that the whole image cannot be processed in a uniform manner using a single point spread function. Existing methods in literature deal with this problem by segmenting the blurry region [5] or by approximating the motion as a series of homographies.

2.3.2 CAMERA SHAKE BLUR

Camera shake blur is generated by camera motion during the image acquisition process. This effect is more profoundly present in photography without a tripod and in low light conditions. This type of blur can be very described mathematically in a very complex manner, since the motion appears in a non-regular direction[6] .Ideally this effect can be described in the case that the user captures a long distance object and the result can be approximated by a spatially variant blur so as to be modelled with a linear motion blur.
Additionally, camera rotation is even more complicated. It includes in-plane rotation and out-of-plane rotation. In the first case the blur kernel varies significantly across the image, especially for the image parts far from the rotation axis. For out-of-plane rotation, the degree of spatial variance across the image is dependent on the focal length of the camera[6]

2.3.3 DEFOCUS BLUR

Due to low quality focusing or due to different depths of scene, the parts of the image being outside the focus field are defocused. This causes the so called defocus blur. Additionally, when only one lens exists on the camera scene parts being outside the depth of field appear blurred. Defocus blur can be modelled as:

$$h(i,j)=\begin{cases}\frac{1}{\pi R^2}, if\ \sqrt{i^2+j^2}\le R,\\ 0, otherwise,\end{cases} \tag{2-4}$$

In the equation above R is the radius of the circle. This equation models the out of focus blur properly if R is selected properly and the depth of scene does not vary much. Modern cameras deal with that problem as they have an autofocus function. However, several times it may not be possible to capture the entire image as a sharp image due to limited DOF.

2.3.4 ATMOSPHERIC TURBULENCE BLUR

Atmospheric turbulence blur is found in cases that long distance imaging systems are used. Such a case can be aerial photography. This happens due to atmospheric factors causing a superposition of pixels and in fact it is a mixture of multiple blur types. The atmospheric turbulence blur kernel can be described by a fixed Gaussian model[7]

$$h(i,j) = Z\exp\left(-\frac{i^2+j^2}{2\sigma^2}\right) \tag{2-5}$$

2.4 BAYESIAN INFERENCE

Bayesian inference is based on Bayes rule:

$$p(A|B) = \frac{p(B|A)\,p(A)}{p(B)} \tag{2-6}$$

where A stands for the hypothesis set and B corresponds to the evidence set. Specifically, it updates the probability of a hypothesis by using additional evidence. Bayes rule states that the true posterior probability p(A|B) is based on our prior knowledge of the problem being p(A), and is updated according to the compatibility of the evidence and the given hypothesis referring to the likelihood p(B|A). Regarding the issue under investigation two formulations can be made, one for the blind case and one for the non-blind case.

$$p(x|y,h) = \frac{p(y|x,h)\,p(x)}{p(y)} \tag{2-7}$$

$$p(x,h|y) = \frac{p(y|x,h)\,p(x)\,p(h)}{p(y)} \tag{2-8}$$

In the equations above x, y, and h are usually assumed to be uncorrelated. In order to use the equation (2.6) two methods are analyzed in this study i) maximum a posteriori, and ii) variational Bayesian methods.

2.4.1 MAXIMUM A POSTERIORI METHODS

Maximum a posteriori probability (MAP) is the most common estimator in a Bayesian inference. If we assume that A is a hypothesis set and B is the evidence meaning what we already know about the issue under investigation, MAP finds the best solution that maximizes the distribution of A given the evidence set B in (2.6). Regarding the blind case, the following formulation can be used:

$$(x^*,h^*) = \arg\max_{x,h}\, p(x,h|y) = \arg\max_{x,h}\, p(y|x,h)\,p(x)\,p(h) \tag{2-9}$$

- **MAXIMUM LIKELIHOOD**

In order to describe the maximum likelihood method, the classic non-blind algorithm of Richardson-Lucy (RL) deconvolution [2], [8] is used The certain algorithm has several applications in astronomical and medical imaging. However, in the standard RL no specific noise assumption is made. Assuming that the prior p(x) takes the form of a uniform distribution, the MAP estimator then becomes a maximum likelihood estimator (MLE):

$$x^* = \arg\max_x p(y \mid x, h) = \arg\min_x \left(-\log p(y \mid x, h)\right) \tag{2-10}$$

In the second part of the equation the minimized equation is known as negative log-likelihood. Under the noise assumption of Poisson distribution, the aforementioned formulation becomes

$$x^* = \arg\min_x \sum_i \left((x*h)_i - y_i \log(x*h)_i\right) \tag{2-11}$$

Taking the derivative with respect to x and setting it to zero, we can get

$$\left(1 - \frac{y}{x*h}\right) * h\_ = 0 \tag{2-12}$$

where:

- $\frac{y}{x*h}$ represents point-wise division
- h_ is the symmetrical reflection of h, meaning h_(i, j) = h(-i,-j).
- 1 is the spatially invariant function which is one everywhere
- 1*h_=1

Thus:

$$\frac{y}{x*h} * h\_ = 1 \tag{2-13}$$

Multiplying both sides by x and using the Banach fixed-point theorem:

$$x^{t+1} = x^t \odot \left[\left(\frac{y}{x^t * h}\right) * h\_\right] \tag{2-14}$$

Where $\odot$ is the point-wise multiplication. The above RL deconvolution procedure produces a sequence of estimations x^t and eventually converges to the optimal solution x*. If we take the blind case into consideration (for h) we get

$$x^{t+1} = x^t \odot \left[\left(\frac{y}{x^t * h^t} \right) * h_-^t \right] \tag{2-15}$$

$$h^{t+1} = \frac{h^t}{1 * x_-^t} \odot \left[\left(\frac{y}{x^t * h^t} \right) * x_-^t \right] \tag{2-16}$$

However, convergence will not reach a global solution. This happens because $\min_{(x,h)} -\log p(y \mid x, h)$ is not convex. Additionally, errors in blur kernel h will cause artifacts in the final image. To cope with fact, the penalized MLE has been introduced:

$$(x^*, h^*) = \arg\min_{x,h} \left[\sum_i \left((x*h)_i - y_i \log (x*h)_i \right) + \lambda_x \Psi_1(x) + \lambda_h \Psi_2(x) \right] \tag{2-17}$$

Where ψ1(.) and ψ2(.) are the penalty functions on x and h.

$$x^{t+1} = x^t \odot \left[\frac{\left(\frac{y}{x^t * h^t} \right) * h_-^t}{1 + \lambda_x \frac{\partial \Psi_1(x^t)}{\partial x^t}} \right] \tag{2-18}$$

$$h^{t+1} = h^t \odot \left[\frac{\left(\frac{y}{x^t * h^t} \right) * x_-^t}{1 * x_-^t + \lambda_h \frac{\partial \Psi_2(h^t)}{\partial h^t}} \right] \tag{2-19}$$

The above method is developed under the assumption that the blur kernel is uniform. Tai et al (2011)[9] developed the projective motion RL algorithm cope with this fact.

- **PRIORS FOR MAP**

The MAP is derived by the penalized MLE. This is because the penalty functions act as the priors in that $p(x) = e^{-\lambda_x \psi_1(x)}$ and $p(h) = e^{-\lambda_h \psi_2(h)}$. Specifically, the MAP based methods use priors and it is the quality of priors that creates different deblurring results.
It is known that sharp natural images tend to obey a heavy tailed distribution. This leads to the fact that the distribution of gradients has most of its mass on small values [3] meaning that natural images contain large regions of constant intensity. To use such a prior, several usages of the heavy-tailed distribution are introduced. One method is to use a Laplace distribution for the magnitude of gradients:

$$p^{Lap}(\nabla x) = \prod_i \frac{1}{2b} \exp\left(-\frac{\|\nabla x_i\|_1}{b} \right) \tag{2-20}$$

where ∇ is the gradient operator, $\|\cdot\|_1$ is the ℓ_1 norm, and b denotes the scale parameter. Additionally, for computation efficiency, a generalized Gaussian model is used by [10]in their design of an optimal aperture filter. The autocorrelation function of the blur kernel is shown in relation to the covariance matrix of the Gaussian distribution. Through the Fourier transform of the autocorrelation matrix and an additional phase retrieval stage, the blur kernel can be easily recovered. Unlike the MAP methods involving repeated reconstructions of the sharp image, this approach directly relies on basic statistics of the blurry image and is therefore efficient. To improve the fitness to the heavy-tailed distribution, Fergus et al (2006)[3] proposed a Gaussian mixture model (GMM) having finite mixture numbers. Chakrabarti et al (2010)[5] extended this case to Gaussian scale mixture (GSM) which is a mixture of infinite Gaussian models with a continuous range of variances. These are correspondingly:

$$p^{GMM}(\nabla x) = \prod_i \sum_{c=1}^{C} N(\nabla x_i \mid 0, \xi_c) \tag{2-21}$$

$$p^{GSM}(\nabla x) = \prod_i \int_{\xi} \mathrm{N}(\nabla x_i \mid 0, \xi)\, p(\xi)\, d\xi \tag{2-22}$$

where ξ is the standard derivation of the Gaussian distribution, c and C in GMM is the index and the total number of mixtures, and p(ξ) in GSM is a probability distribution on ξ. In terms of GSM, a critical issue is the infinite selection of ξ, which makes it computationally expensive.
Another relevant issue is whether the recovered Gaussian prior distribution will fit the heavy-tailed prior. In a recent study [11]it was shown that this does not happen because the priors are applied on all pixels thus, failing to capture the global statistics of gradients. To cope with this issue, the authors used a penalized Kullback-Leibler (KL) divergence between the gradient distribution of the estimated image and a reference distribution $p_r(\nabla x)$,

$$KL(p_e \| p_r) = \int_{\nabla x} p_e(\nabla x) \ln\left(\frac{p_e(\nabla x)}{p_r(\nabla x)}\right) d\nabla x \tag{2-23}$$

which introduces a constraint. The reference distribution is characterized as the generalized Gaussian model and immediately estimated from the blurry image by using the approach of Cho et al 2010[12].

- **EDGE PREDICTION BASED**

Based on the Bayesian framework an edge emphasizing operation is used in order to provide estimation on the latent/sharp image. This step may include shock filtering [13] fuzzy operator[14] ,morphological filtering [15], forward and backward diffusion process[16].This category also includes the edge prediction process proposed by Cho et al 2009. The drawback of this category of methods is that they misbehave in the case the image contains regions with edged that cannot be highlighted by the edge prediction step.

However, this drawback is compensated by embedding the aforementioned edge emphasizing operation to the MAP estimator. This way several blur types can be dealt with Cho and Lee 2009[17] improving performance. Additionally, Almeida and Almeida (2010) [18] used a set of edge detectors and assumed that the resulting prior would be sparse . Xu and Jia (2010)[19] showed that not all edges are useful for estimating the blur kernel and introduced a metric to measure how important is each strong edge.

- **MARGINALIZATION**

In this section a technique is analyzed for improving the MAP estimator. In the recent research of Levin et al.[20][21] the problems of the MAP were noted and a scheme was proposed to improve the estimator. For the eq. 2.9 it can be derived:

$$(x^*, h^*) = \underset{x,y}{\arg\min} \|y - x * h\|^2 + \lambda\left(\sum_i |\nabla_h x_i|^a + |\nabla_u x_i|^a\right) \tag{2-24}$$

The assumptions for the problem above are that the noise is Gaussian and the prior is the image derivatives which are sparse .Additionally, ∇_h and ∇_u are horizontal and vertical derivatives As Wang and Tao stated and according to Levin et al.'s conclusions, the solution of equation(2.24) under the sparse prior usually favors a blurry result rather than a sharp result.
As Levin et al also showed there is an asymmetry between the dimensionality of x and h providing a favorable property for handling the blind deconvolution. This means that while the dimensionality of x increases with the image size, the support of h remains fixed and is small relative to the image size. From this viewpoint, h can achieve an increased number of measurements when the image size becomes large. Thus, estimation theory tells us through sufficient measurements on h that the recovered blur kernel under MAP_h can be arbitrarily close to the true kernel. Mathematically, the MAP_h is

$$h^* = \underset{h}{\arg\max}\, p(h \mid y) = \underset{h}{\arg\max} \int p(x, h \mid y)\, dx \tag{2-25}$$

where h^* is the true kernel as stated in Claim 3 of Levin et al 2011[20].

Once the kernel is estimated, x can then be solved in a non-blind deblurring scheme. In their subsequent work [22] Levin et al. noted that MAP_h is generally complex and hard to directly compute because the marginalization in equation (2.25) involves all possible x explanations, which is computationally intractable. An approximation method was proposed to derive the MAP_h. To estimate the blur kernel, they assumed the $i.i.d.$Gaussian imaging noise and the GMM prior on image derivatives, as well as a uniform distribution on h. Equation (2.25) can then be written as

$$h^* = \arg\max_h p(y|h) = \arg\max_h \int p(x,y|h)\,dx \qquad (2\text{-}26)$$

The above problem is solved by the expectation-maximization (EM) framework that alternates between the estimation of $p(x|y,h)$ which is still a Gaussian (E-step), and the computation of h under the minimum mean square error (M-step). In the E-step, however, calculating the mean and covariance of $p(x|y,h)$ under a sparse prior is generally hard, so the authors proposed approximating the conditional distribution by using variational inference.
Wang et al (2013)[23] have discovered several intrinsic issues between edge emphasizing operations and image statistics through a large number of experiments on ImageNet composed of 1.2 million images in total. Their research points out that the limited number of large scale step edges within a natural image cannot ensure a robust estimation of the blur kernel. Additionally, due to the diversity of natural images, the sparse derivative priors are not consistent across them and it is almost impossible to find a robust measurement that favors sharp explanations for all of them. Different from their previous work [24]which uses $MAP_{x,h}$, they adopted the marginalization scheme in equation (2.25) and developed an adaptive sparse prior composed of two components to ensure robustness. The first component is the commonly-used sparse derivative prior, while the second encodes the edge emphasizing operation.

2.4.2 MINIMUM MEAN SQUARE ERROR

As it has already been stated the Bayesian framework aims to estimate x or h from the posterior $p(x,h|y)$. Thus a loss function $L((x^*,h^*),(x,h))$ can be formulated stating the difference between the optimal solution and every available combination of x and y. Specifically

$$\tilde{L}((x^*,h^*),(x,h)|y) = \int L((x^*,h^*),(x,h))\,p(x,h|y)\,dxdh \qquad \mathbf{(2\text{-}27)}$$

is the Bayesian expected loss [25]. In order to obtain the optimal combination of (x^*,h^*) requires to minimize the $\tilde{L}((x^*,h^*),(x,h))$.If an assumption of Dirac delta function is used for $L((x^*,h^*),(x,h))$ then the equation above is equivalent to the $MAP_{x,h}$ problem. Additionally, if the assumption is square of L then the minimum mean square error formulation is derived:

$$(x^*,h^*) = \arg\min_{\hat{x},\hat{h}} \int \|\hat{x}-x\|^2 \left\|\hat{h}-h\right\|^2 p(x,h|y)\,dxdh = \arg\min_{\hat{x},\hat{h}} \mathrm{E}\{x,h|y\} \qquad (2\text{-}28)$$

For Dirac delta loss function:

$$L((x^*,h^*),(x,h)) = 1-\delta((x^*,h^*)-(x,h)) \qquad (2\text{-}29)$$

And for the non-blind case:

$$x^* = \arg\min_{\hat{x}} \int \left\| \hat{x} - x \right\|^2 p\left(x \mid y, h\right) dx = \arg\min_{\hat{x}} \mathrm{E}\left\{x \mid y, h\right\} \tag{2-30}$$

Levin et al. [20] proved the equivalence of MMSE and MAP .
In spite of this, as Wang and Tao state the empirical demonstrations on image denoising have shown the advantages of MMSE over MAP approaches [26] .MAP solutions usually exhibit piecewise constant regions and result in incorrect statistics of the output image, as previously noted, whereas MMSE can achieve the desired statistics by exploiting the uncertainty of the model. Furthermore, the image restoration performance of the MMSE estimator is highly correlated with the generative quality of the model. This observation is particularly useful since MMSE benefits from a powerful learnt generative model even without any regularization weight. As is well known, the regularization parameter is related to the noise level of the degraded image. By taking the above superiority of MMSE, [27] integrated the noise estimation process into the MMSE framework by treating the noise standard deviation as a variable of the posterior, i.e., given h and y,

$$p\left(x, y \mid h\right) = \int p\left(x, \sigma \mid y, h\right) d\sigma \tag{2-31}$$

However one issue remains. Due to the lack of complete knowledge on the joint distribution $\left(p\left(x, h \mid y\right) \text{ and } p\left(x \mid y, h\right)\right)$ in real applications, it is difficult to take expectation in equations (2.29) and (2.30) over all possible explanations.
To handle this problem, Schmidt et al (2010, 2011)[26], [27] proposed to use the Gibbs sampling method to alternatively generate the sequence of the variable samples, e.g. in deblurring $\left\{\left(x^1, z^1, \sigma^1\right), \ldots, \left(x^T, z^T, \sigma^{\mathrm{T}}\right)\right\}$ where z is a latent variable.

2.4.3 VARIATIONAL BAYESIAN METHODS

Variational Bayesian methods provide a framework for estimating unknown variables without computing them in an explicit manner. This method estimates a distribution rather than a specific value. Three recent studies[3], [20], [28] refer to this method.
Specifically, mean-field variational Bayes is the most common method. It approximates the posterior $p\left(A \mid B\right)$ by finding a variational distribution $q\left(A\right)$ by using the Kullback–Leibler divergence to measure the difference between these two distributions.

$$L\left(q\right) := D_{KL}\left(q \parallel p\right) = \int q\left(A\right) \log \frac{q\left(A\right)}{p\left(A \mid B\right)} dA \tag{2-32}$$

Miskin and MacKay (2000)[29] developed an ensemble learning strategy in order to use the aforementioned scheme. Specifically, the latent image set A is an ensemble of the sharp image x, the blur kernel h and the noise variance σ^2 if the Gaussian noise is assumed. For simplification, a separable factorization of the q distribution is used :

$$q\left(x,h,\sigma^2\right)=q(x)q(h)q\left(\sigma^2\right) \tag{2-33}$$

Subsequently

$$L(q)=\mathrm{E}_{q(x)}\left\{\log\frac{q(x)}{p(x\mid y)}\right\}+\mathrm{E}_{q(h)}\left\{\log\frac{q(h)}{p(h\mid y)}\right\}+\mathrm{E}_{q(\sigma^2)}\left\{\log\frac{q(\sigma^2)}{p(\sigma^2\mid y)}\right\} \tag{2-34}$$

The variable B in (2.38) is the observed blurry image y. The purpose of this method is to minimize L(q) with respect to $q\left(x\mid h\mid\sigma^2\right)$. In order to solve the above factorization, we can take an alternating update procedure by minimizing one factor while marginalizing out the other factors ($q(h)$ and $q\left(\sigma^2\right)$). The updates are performed by computing the closed-form optimal parameter updates, and performing line-search along the direction of these updated values. According to Bishop (2006)[30], the optimal factors obtained from the corresponding sequential updates are given by the expectation of the joint distribution with respect to all unobserved variables except the one of interest, and thus

$$\begin{aligned}\log q^*(x)&=E_{q(h)q(\sigma^2)}\left\{\log p\left(x,h,\sigma^2,y\right)\right\}+const\\&=E_{q(h)}\left\{\log p\left(y\mid x,h\right)\right\}+\log p(x)+const\end{aligned} \tag{2-35}$$

$$\begin{aligned}\log q^*(h)&=E_{q(x)q(\sigma^2)}\left\{\log p\left(x,h,\sigma^2,y\right)\right\}+const\\&=E_{q(x)}\left\{\log p\left(y\mid x,h\right)\right\}+\log p(h)+const\end{aligned} \tag{2-36}$$

$$\log q^*\left(\sigma^2\right)=E_{q(x)q(h)}\left\{\log p\left(x,h,\sigma^2,y\right)\right\}+const=\log p(\sigma^2)+const \tag{2-37}$$

Following the calculation of the above equations, the final estimates of the sharp image x, the blur kernel h and the noise variance σ^2 are taken as the mean values of the distributions $q^*(x)$, $q^*(h)$ and $q^*\left(\sigma^2\right)$ correspondingly.

Using this framework, Fergus et al (2006)[3] and Whyte et al (2012)[6] operated the variational formulation on the gradient domain, i.e., x in the above equations is replaced by the image gradients, to facilitate the statistical assumption on model priors, such as sparse gradients. Levin et al (2011)[20] employed the variational inference to handle the marginalization problem in MAP_h. Following an iterative optimization procedure, the optimal distribution can be solved. Levin et al.'s method differs from Fergus et al (2006) and Whyte et al (2012) in that the target distribution to be approximated is

$p(x|y,h)$ rather than $p(x,h|y)$. However, Fergus et al.'s approach selects h^* from the estimated $q(x,h)$ distribution by marginalizing out all possible x's, and thus belongs to the MAP_h approach.

2.5 VARIATIONAL METHODS

Variational methods are derived from the calculus of variations and are typically used as approximation methods to convert an ill-posed problem into a well-posed problem which is characterized by using additional constraints to reduce the size of the solution space of the unknown variables[31] .To approximate the problem, a typical setting involves the maximum or the minimum of a functional composing a function and the associated constraints:

$$\min_{A} \Phi(A|B) + \lambda \Psi(A) \tag{2-38}$$

where A is the undetermined variables and B is the observations. In variational principle, Φ(A|B) is called the data-fidelity function, $\Psi(A)$ is the regularization function, and λ denotes the regularization parameter. Under this formulation, the non-blind image deblurring problem can be written as

$$\min_{x} \Phi(x|y,h) + \lambda_x \Psi_x(x) \tag{2-39}$$

while the blind case is

$$\min_{x,h} \Phi(x|y,h) + \lambda_x \Psi_x(x) + \lambda_h \Psi_h(h) \tag{2-40}$$

The term ϕ is determined according to the noise assumptions listed in **Error! Reference source not found.**. In this section we generally assume the noise to be Gaussian and the corresponding Φ is given by

$$\Phi = \|y - x * h\|_2^2 \tag{2-41}$$

2.5.1 REGULARIZATION METHODS

Similar to the character of the prior in the Bayesian inference framework, the regularizers in a variational framework express human knowledge on the interested blurry images. Such knowledge can constrain the solution space such that the deblurred images are favored by human sense.

- **SINGLE IMAGE DEBLURRING**

In order, to stabilize the deblurring result, the solution is expected to have a small norm, so as to apply the Tikhonov-Miller regularizer[32] on the sharp image:

$$\Psi_x(x) = \|x\|^2 \tag{2-42}$$

However, this is rarely used in deblurring tasks because the latent images will have very smoothed edges. Thus, using the first-order regularizers which maximally preserve the significant details is more frequently adopted. A usual example is the total variation (TV) method proposed by Rudin et al (1992)[33]

$$TV_i(x) = \left\| \sqrt{|\nabla_h x|^2 + |\nabla_u x|^2} \right\|_1 \tag{2-43}$$

where the subscript i means it is the isotropic version. Complementarily, the anisotropic TV is

$$TV_a(x) = \left\| |\nabla_h x| + |\nabla_u x| \right\|_1 \tag{2-44}$$

Both TV_i and TV_a enhance the visualization of edges in the latent images. They mainly differ from each other in their sensitivity to edge directions. From the formulations, we can see that TV_i enforces the same strength on the edges with different directions, whereas TV_a favors certain directions. Both methods have proven to be useful in numerous applications, such as image denoising, decomposition, super-resolution, inpainting, and non-blind deblurring. Nevertheless, when applied to blind deblurring problems, some failures occur.

It is important to emphasize that TV is intrinsically an ℓ_1 norm of the image gradients, and thus induces sparsity over image gradients. According to the delta-effect of MAP, simultaneously estimating x and h will result in a blurry image. Another perspective from the ℓ_1 properties can assist the understanding of TV failure. For a sharp image of natural scenes, the gradient magnitude is typically sparse, meaning that most values are either zero or very small, but may occasionally be large. If a blur kernel is operated on this image, the high-frequency bands will be attenuated, leading to the magnitudes being un-sparse. To recover the original sparsity, a natural choice is the ℓ_0 measure, an important property of which is the scale-invariance, i.e., $\min \ell_0(\nabla x) = \min \ell_0(a \cdot \nabla x)$ any positive values of a. Minimizing ℓ_0 will only lead to a sparse effect, without destroying the magnitudes of large values, thus preserving the energy of original gradients. However, ℓ_0 is difficult to optimize because of the lack of derivative information everywhere, and then ℓ_1 is utilized as an alternative to approximate ℓ_0. Unfortunately, the blurring process in itself reduces the ℓ_1 norm of the gradients. Minimizing ℓ_1 fails to preserve or recover the energy of the original gradients.

Additionally, the scale variant property makes ℓ_1 sensitive to the setting of the regularization parameter λ. Therefore, various methods of approximating the ℓ_0 norm while maintaining the scale-invariance property are proposed.Krishnan et al (2011)[34] recently extended the ℓ_1 norm to a normalized version:

$$\Psi(\nabla x) = \frac{\|\nabla x\|_1}{\|\nabla x\|_2} \tag{2-45}$$

To understand this regularizer, let us focus on the denominator, ℓ_2 norm. The blurring process reduces the ℓ_2 norm of the gradients as well. Fortunately, ℓ_2 is reduced more than the numerator ℓ_1 norm, leading to an increased ratio of the two terms. Therefore, minimizing this regularizer will deduce the blurry effect in the image without destroying the magnitude of the true gradient because ℓ_1/ℓ_2 is evidently scale invariant, just as we expected.

Another example of the approximation is the unnatural ℓ_0 regularizer which is proposed by Xu et al (2013)[35] . The unnaturalness stems from the observation that in most iterative deblurring methods, the intermediate image results only contain high-contrast and step-like structures while suppressing others. These images are different from natural scenes, and hence the term 'unnatural' is exploited. To incorporate the step-edge properties in an unnatural representation, the authors utilized the unnatural ℓ_0 scheme to preserve the salient changes (i.e., the gradients) in the image. The resultant regularizer is formulated as

$$\psi(\nabla_h x) = \sum_i \psi(\nabla_h x_i) \tag{2-46}$$

$$\psi(z) = \begin{cases} \frac{1}{\varepsilon^2}|z|^2, |z| \le \varepsilon, \\ 1, otherwise. \end{cases} \tag{2-47}$$

The definition on the vertical derivative ∇ is similar. Depending on the formulation, the gradient magnitudes smaller than ε are penalized by ψ(.) while the larger values result in a constant 1 in the objective function. Minimizing this regularizer will remove fine structures and keep useful salient details in the result.

Another property ensuring the unnatural ℓ_0 superior to ℓ_1 is its scale invariance property, as previously stated. By using this regularization technique in the estimation of blur kernels, the deblurring performance has been notably improved.

While the above regularizers are all based on first-order derivatives, second-order regularization techniques have also proven to be useful in image denoising tasks, and have recently been introduced to deblurring images Lefkimmiatis et al (2012)[36] extended the first-order TV functional to two second-order cases by defining the mixed norms including $\ell_1 - \ell_\infty$, $\ell_1 - \ell_2$. These regularizers maintain favorable properties of TV (such as convexity, homogeneity, rotation and translation invariance) well, and can effectively suppress the staircase effect.

To solve the resultant variational problem, an efficient algorithm is proposed based on the majorizationminimization approach. Rather than only enforcing the second-order regularization in deblurring tasks, Papafitsoros and Schonlieb (2014)[37] handled the combined problem involving both first and second order functionals. The benefit is that the first-order term recovers the step-edges as well as possible, while the second-order term eliminates the artifacts of the staircase produced by the first-order regularizer, without introducing any serious blur in the reconstructed image. Further, the existence and uniqueness of the solution to the combined problem is proved, and numerical solutions are provided based on the split Bregman iteration [38].

2.5.2 OPTIMIZATION METHODS

In a variational framework, as well as in other related schemes, a good optimization algorithm can achieve a fast convergence rate and produce an accurate solution. For the problem of deblurring, the general formulation is

$$\min \frac{1}{2}\|y - Hx\|^2 + \lambda\Psi(x) \tag{2-48}$$

where we utilize the matrix-vector expression $\mathbf{y} = \mathbf{Hx} + \mathbf{n}$. Even though this is for the non-blind deblurring problem, we can see from the discussion in previous sections that the blind case is generally decomposed into a two-step procedure, in which the blur kernel is first estimated and then the sharp image is calculated.

A standard algorithm for solving the problem of the afore mentioned minimization problem is called iterative shrinkage / thresholding algorithm (IST). For example, if we set Ψ as the ℓ_0 norm on x, the corresponding shrinkage / thresholding function is called the hard-threshold function[39]:

$$T_{\lambda\ell_0}(w) = w \odot 1_{w \geq \sqrt{2\lambda}} \tag{2-49}$$

where w is the observation to be approximated, $1_{w \geq \sqrt{2\lambda}}$ is the indicator function determined by the condition of the subscript. If ψ is the ℓ_1 norm, the soft-threshold function[39] is then utilized:

$$T_{\lambda\ell_1}(w) = sign(w) \odot \max(|w| - \lambda, 0) \tag{2-50}$$

For solving (2.48) the IST iteration is given by:

$$x^{t+1} = T_{\lambda\psi}\left(x^t - \frac{1}{\gamma}\mathrm{H}^*\left(\mathrm{H}x^t - y\right)\right) \tag{2-51}$$

where H* is the adjoint if the matrix H and $\frac{1}{\gamma}$ is the step size. The convergence rate of IST is determined by the parameter λ and the matrix H. Small values of λ and/or the ill-condition of H results in slow convergence. Another popular scheme for solving the problem (2.48) is the alternating direction method of multipliers (ADMM) [40][41], a variant of the augmented Lagrangian Method (ALM). Formally, (2.48) into the following constrained problem by introducing an auxiliary variable z:

$$\begin{aligned} &\min \; f_1(x) + f_2(z), \\ &s.t. \quad x = z \end{aligned} \tag{2-52}$$

where $f_1(x) = \frac{1}{2}\|\mathbf{y} - \mathbf{Hx}\|^2$ and $f_2(z) = \lambda\Psi(z)$. This procedure is called variable splitting. By incorporating the ALM (Adaptive Landweber method) techniques the equation above can be written as

$$\min_{x,z,\beta} L_\mu(x,z,\beta) = f_1(x) + f_2(z) + \frac{\mu}{2}\|x - z\|_2^2 - \beta^{\mathrm{T}}(x - z) \tag{2-53}$$

where $\mu \geq 0$ is the penalty parameter and β is a is a vector of Lagrange multipliers. ADMM alternatively optimizes x and z[40] , associated with an additional step to estimate the Lagrange parameters

$$x^{t+1} = \arg\min_x L_\mu\left(x, z^t, \beta^t\right) \tag{2-54}$$

$$z^{t+1} = \arg\min_z L_\mu\left(x^{t+1}, z, \beta^t\right) \tag{2-55}$$

$$\beta^{t+1} = \beta^t + \mu\left(x^{t+1} - z^{t+1}\right) \tag{2-56}$$

If a transformed signal (say the gradients) of x is regularized, the constraint in (2.55) is replaced by the corresponding equations (Gx = z where G is the matrix expression of the derivatives).

2.6 HOMOGRAPHY BASED METHODS

In this section and the next section, our discussion will mainly focus on the modeling and associated processing of the spatially variant blur effect. Homography-based modeling is generally proposed to simulate the blur effect induced by the camera's motion or camera shake. Recall that in the image formation process, a 3D scene point (u, v, m) is mapped to a 2D image plane point (i, j), which can be formulated in the homogeneous coordinates as

$$\left(i_t, j_t, 1\right)^T = Pt\left(u, v, m, 1\right)^T \tag{2-57}$$

where t denotes the time index. In the case of camera motion, Pt may vary with time as a function of camera translation and rotation, causing a fixed point in the scene to be projected onto different locations in the image plane at each time. As is well-known, when using a pinhole camera, all views seen by the camera are projectively equivalent except for the boundaries[6], [42] .This means that for a static scene with constant depth, the 2D images projected at different instances of time are related via a homography. Denoting the image point at time t = 0 as $\left(i_0, \mathrm{j}_0\right)$ the homography and projected point at time t are modeled as

$$H_t\left(d\right) = K\left(R_t + \frac{1}{d}T_t N^T\right)K^{-1} \tag{2-58}$$

$$\left(i_t, j_t, 1\right)^T = H_t\left(d\right)\left(i_0, j_0, 1\right)^T \tag{2-59}$$

for a particular depth d, where K is the camera's internal calibration matrix, Rt and Tt are the rotation matrix and translation vector at time t, and N is the unit vector orthogonal to the image plane. Based on this formulation, the image captured at any time is expressed by the initial image x_0, i.e.

$$x_t(i, j) = x_0\left(H_t\left(d\right)\left(i, j, 1\right)^T\right) \tag{2-60}$$

By expressing the coordinates as a column vector i and the Homography as Ht we get:

$$x_t(i) = x_0\left(H_t i\right) \tag{2-61}$$

Then the blurry image can be defined as the accumulated result over the exposure duration τ which is given by

$$y\left(i\right) = \int_{t=0}^{\tau} x_0\left(H_t i\right) dt \tag{2-62}$$

where the noise term is omitted. Writing $x_t(i) = x_0\left(H_t i\right)$ in a matrix-vector form we get $\mathbf{x_t} = \mathbf{H_t x_0}$ where $\mathbf{H_t}$ is a sparse resampling matrix that implements the image warping and resampling due to homography. Thus the blurry image can be defined as

$$y = \int_{t=0}^{\tau} \mathbf{H_t x_0} dt \tag{2-63}$$

However, due to the successive duration [0, τ] infinite instantiations of Ht will be created causing the ill-posedness of the problem. To handle this issue, two methods can be used:

- Time discretization
- Homography decomposition

2.6.1 TIME DISCRETIZATION

If we suppose that the duration is segmented to N equivalent periods, in each of which the homography **H** is approximately consistent, the equation (2-62) becomes

$$y(i) \approx \frac{1}{N}\sum_{t=1}^{N} x_0\left(H_t i\right) \tag{2-64}$$

The left hand side is equal to y(i) when $N \to \infty$. If N is set to a limited value, the unknown variables in $\{H_t\}$ will be well-constrained, reducing the ill-posedness of the problem. The equation above is expresses the projective motion blur model proposed by Tai et al 2011[9]. To estimate each homography, the motion is assumed to be uniform in the exposure time. Each H_t can therefore be computed directly according to the whole homography generated from t=0 to t=τ.

$$H_t = \sqrt[\mathrm{N}]{H_{[0,\tau]}} \tag{2-65}$$

A strong assumption in equation (2-72) is the consistent homography in each equivalent period, which may not t to the reality. A more general formulation is

$$y(i) \approx \sum_{t=1}^{N} w_t x_0\left(H_t i\right) \tag{2-66}$$

where $w_{\mathbf{t}}$ denotes the proportion of the period occupied by $\mathbf{H_t}$, and $\sum_t w_t = 1$. Under this model, Cho et al (2012b)[11] proposed a registration-based method to estimate the homographies $\{\mathrm{H_t}\}$ and the weights $\{w_{\mathrm{t}}\}$ by using the Lucas-Kanade algorithm[43]. This estimation method is extended into a multiple image deblurring scheme.

2.6.2 HOMOGRAPHY DECOMPOSITION

This strategy decomposes the homography into a set of basic operations, i.e. representing $\mathbf{H_t}$ as a weighted sum of predefined transformations or homographies. This leads to the formulation as

$$y = \sum_{l=1}^{L} w_l \mathbf{H_l x_0} \tag{2-67}$$

where $\{\mathrm{H}_l\}$ denotes the basis set with cardinality of L, w_ℓ is the weight assigned to the lth basis, and $\sum_l w_l = 1.$

A scheme for setting the basis set $\{H_l\}$ was proposed by Whyte et al (2010, 2012). Considering all-directional rotation of the camera, the homographies are correlated with the camera's orientation. The resultant formulation is

$$y(i) = \int_\Theta w(\theta) x_0 (H_\theta i) d\theta \tag{2-68}$$

and its discretized version is

$$y(i) = \sum_\theta w(\theta) x_0 (H_\theta i) d\theta \tag{2-69}$$

where H_θ is the homography specifying the orientation θ of the camera rotation, and w(θ) denotes the weighting function of θ. By setting $\{H_\theta\}$ according to the rotations along the three Cartesian axes, the spatially variant blur kernel can be well-approximated. This model is applied to a single image deblurring problem using the variational Bayesian method, as well as a blurry/noisy image pair deblurring problem under the regularized least-squares formulation. Note that an integral over ... implies a search over the full orientation space, incurring a high computational cost when the camera rotates significantly. Instead, Hu and Yang (2012a) proposed to constrain the camera poses by imposing an initial guess of the pose subspace and searching the optimal solution within this subspace. This initialization of the pose subspace is implemented using back-projection. Compared with the method in (Whyte et al 2010, 2012), Hu and Yang (2012a)'s method produces more favorable results.

2.7 PERFORMANCE AND EVALUATION

In order to evaluate any of the aforementioned methods there has to be a metric reflecting the difference between the latent image result and the ground truth sharp image. Of course this kind of testing is not possible in the case of blind deblurring an image blurred with an unknown blur kernel. Given a test set of blurry images with known sharp corresponding images and known blur kernels there can be a comparison between the resulting deblurred images of any method and the ground truth. For this procedure two metrics have been suggested[44],[45] : i) the peak-signal-to-noise-ratio(PSNR) and ii) the structural similarity(SSIM):

$$PSNR = 10 \cdot \log_{10} \cdot \left(\frac{N \times 255 \times 255}{\sum_i \left(x_i - \tilde{x}_i \right)^2} \right) \tag{2-70}$$

$$SSIM = \frac{\left(2\mu_x \mu_{\tilde{x}} + c_1\right)\left(2\sigma_{x\tilde{x}} + c_2\right)}{\left(\mu_x^2 + \mu_{\tilde{x}}^2 + c_1\right)\left(\sigma_x^2 + \sigma_{\tilde{x}}^2 + c_2\right)} \tag{2-71}$$

Where N is the number of pixels , μ_x and $\mu_{\tilde{x}}$ are the means of x and $\tilde{x}$, σ_x^2 and $\sigma_{\tilde{x}}^2$ are the variances of x and $\tilde{x}$, c_1 and c_2 are two variables to stabilize the division with weak denominator. Additionally, another error metric introduced by Levin et al. 2009[46] is the cumulative error ratio. The error ratio is defined in the following manner:

$$CER = \frac{\left\| I_{out} - I_{gt} \right\|^2}{\left\| I_{kgt} - I_{gt} \right\|^2} \tag{2-72}$$

Where I_{out} is the deblurred image, I_{gt} is the ground truth image and I_{kgt} is the blurred image as it has been deblurred by the true blur kernel. Based on literature review by Wang and Tao[1] a comparison between several deblurring methods appears in the following figures

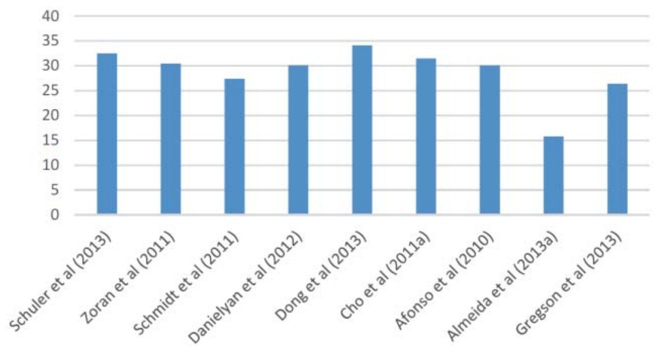


Figure 2-1: Mean PSNR (dB) results of non-blind uniform deblurring[1].

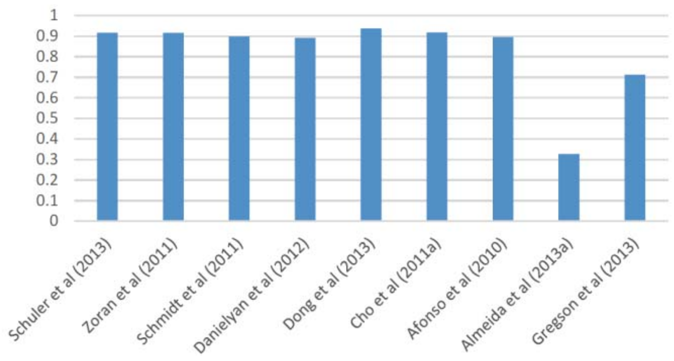


Figure 2-2: Mean SSIM results of non-blind uniform deblurring[1].

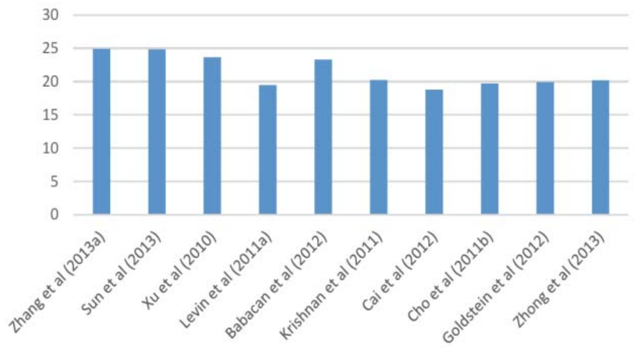


Figure 2-3 : Mean PSNR (dB) results of blind uniform deblurring[1].

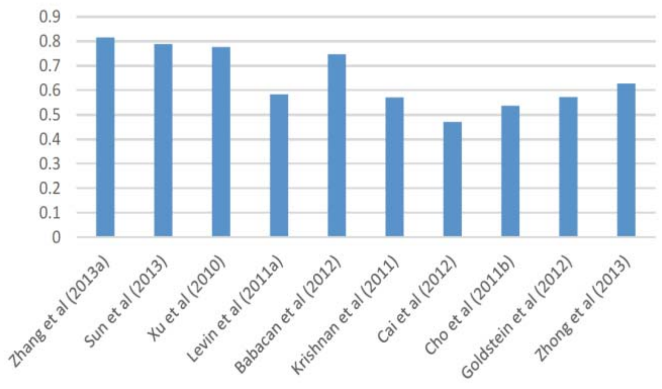


Figure 2-4 : Mean SSIM results of blind uniform deblurring

Another comparison of deblurring methods appears in Cho et al. 2011[47]. Figures below provide deblurring results and a comparison based on the cumulative error metric. Additionally, an important factor is the processing time. A figure showing processing time results is shown below regarding the method of Cho and Lee 2009.

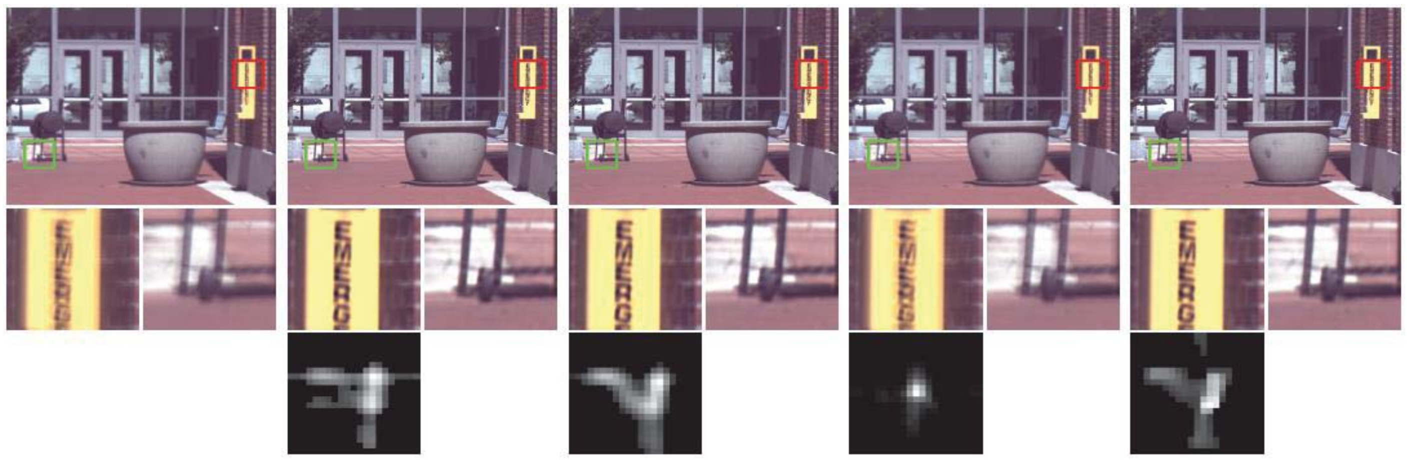


Figure 2-5 : Method comparison based on visual inspection between Fergus et al. [3], Shan et al.[48] , Cho and Lee[17] and Cho et al 2011[47] with regards to the blurry image(first in row). The current image was published in Cho et al. 2011[47].

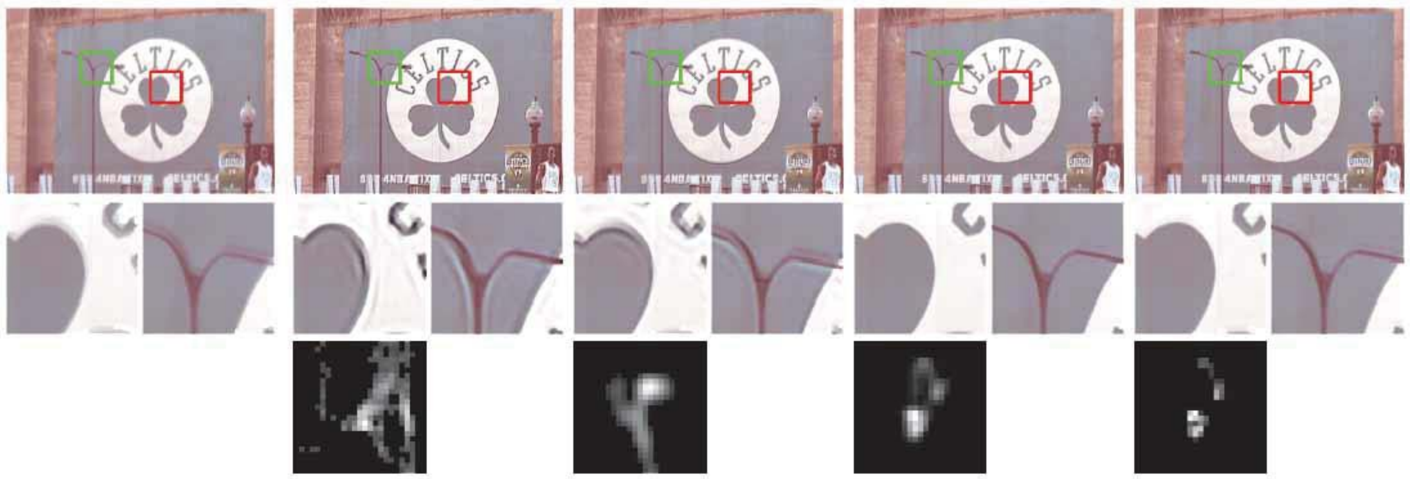


Figure 2-6 : Method comparison based on visual inspection between Fergus et al. [3], Shan et al.[48] , Cho and Lee[17] and Cho et al 2011[47] with regards to the blurry image(first in row). The current image was published in Cho et al. 2011[47].

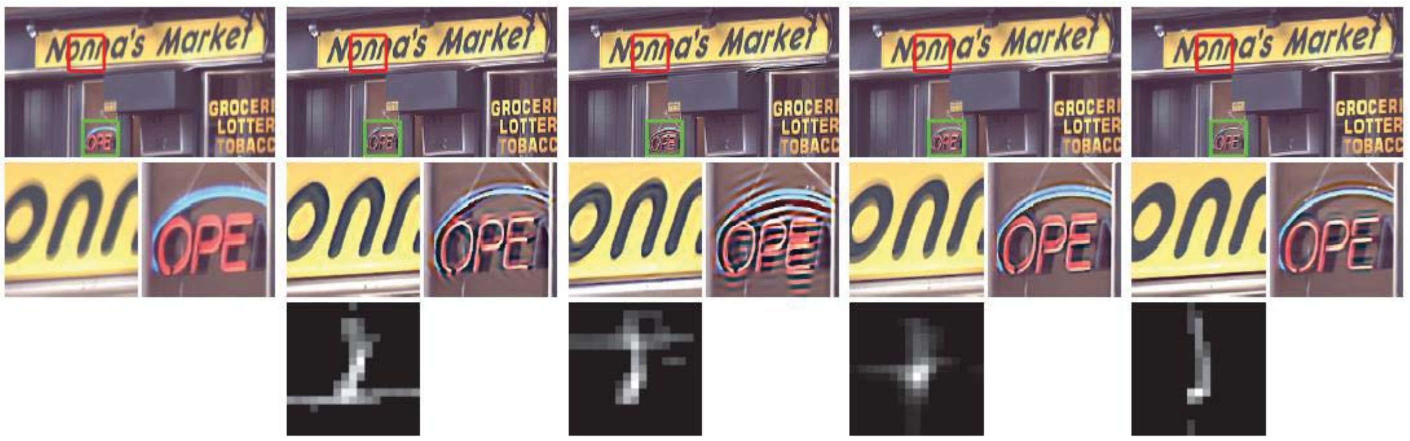


Figure 2-7: Method comparison based on visual inspection between Fergus et al. [3], Shan et al.[48] , Cho and Lee[17] and Cho et al 2011[47] with regards to the blurry image(first in row). The current image was published in Cho et al. 2011[47].

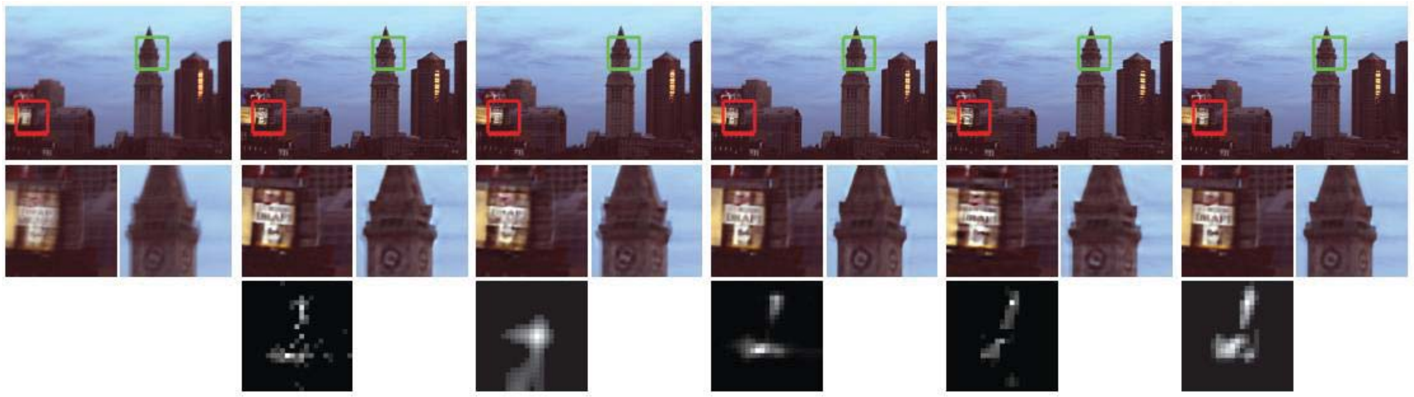


Figure 2-8 : Result comparison based on visual inspection between several methods[47]. The current image was published in Cho et al. 2011[47].

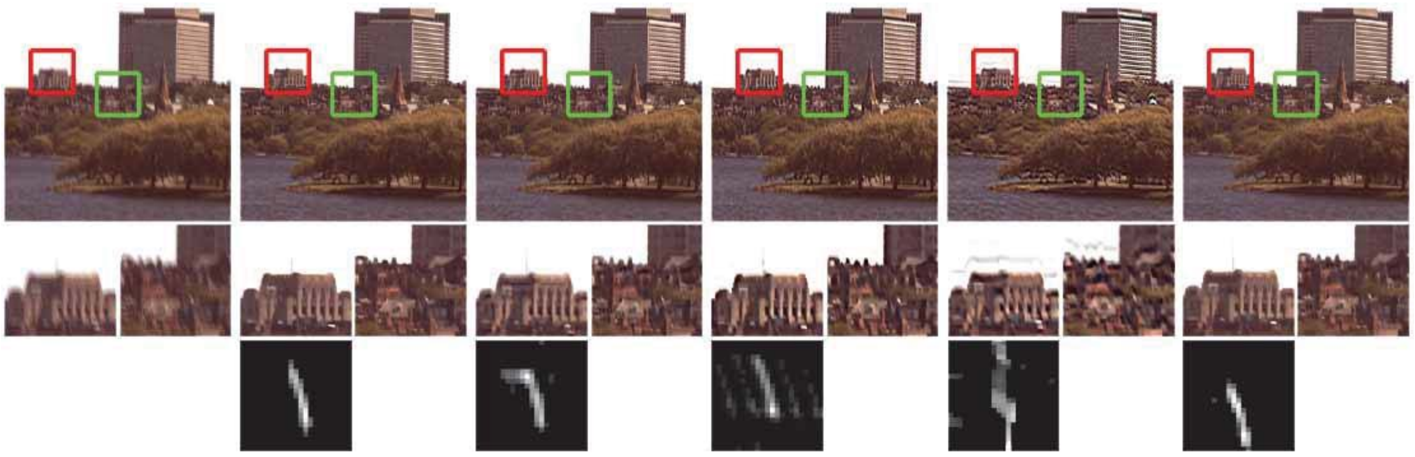


Figure 2-9 : Result comparison based on visual inspection between several methods[47]. The current image was published in Cho et al. 2011[47].

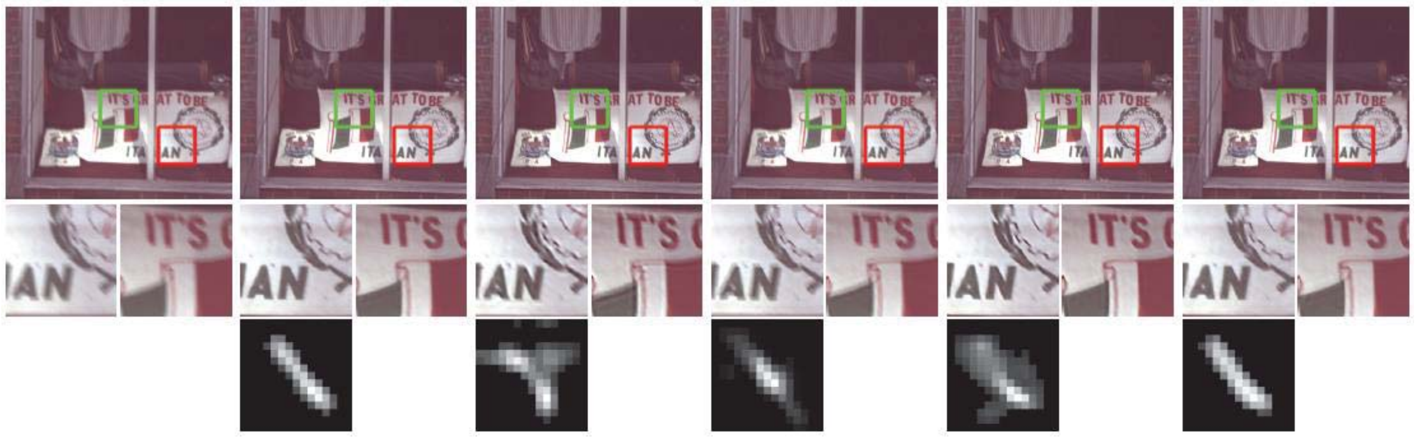


Figure 2-10 : Result comparison based on visual inspection between several methods[47]. The current image was published in Cho et al. 2011[47].

In the following figure we present the cumulative error metric for several deblurring methods. The error ratio plot (not cumulative) can be explained in the following manner: bin r = 3 counts the percentage of test examples achieving error ratio below 3. Consequentially, we can derive the description for the cumulative error metric plot. An error ratio of 1 is ideal, which means that the estimated kernel yields a result as good as the ground-truth kernel. Lower quality results correspond to higher ratios. The higher the curve, the better. For this test set eight blur kernels were used extracted from real-world blurry photos by Levin et al. Additionally a set of 6 one-megapixel color images that are closer to real life photography were used. Half of the images are rich in edges while the other half has fewer of them so that the methods can be tested. The eight kernels and the six images are combined. Additionally, Gaussian noise with + = 0.5% is added. The final result is 48 generated test images.

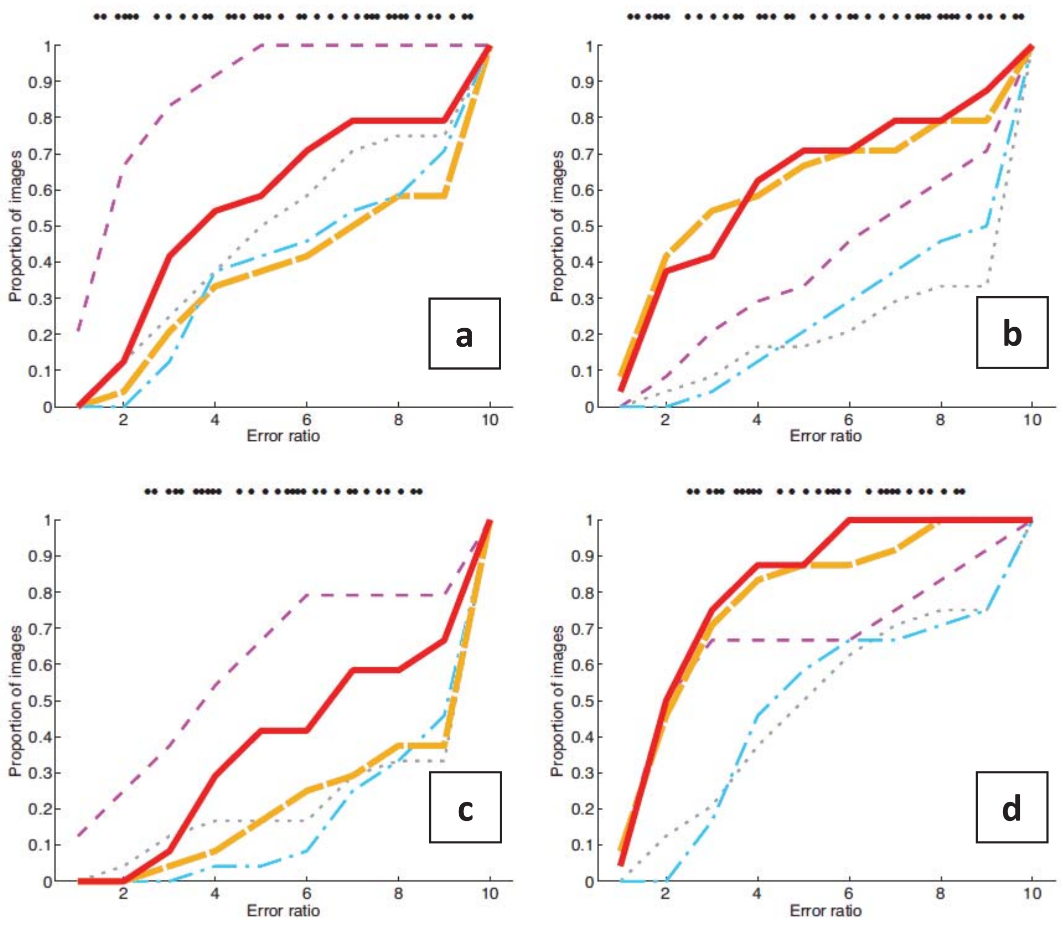


Figure 2-11 : Cumulative error ratios for the following algorithms: Fergus et al.[3], Shan et al.[48], Cho and Lee [17], Cho et al.2011 method based on directly inverting the Radon transform[47] and Cho et al. 2011 Radon MAP algorithm[47]. The current image was published in Cho et al. 2011[47]. Subfigure a refers to images with few edges, subfigure b refers to images with many edges, subfigure c refers to the case that the kernel is large and d to the case that the kernel is small.

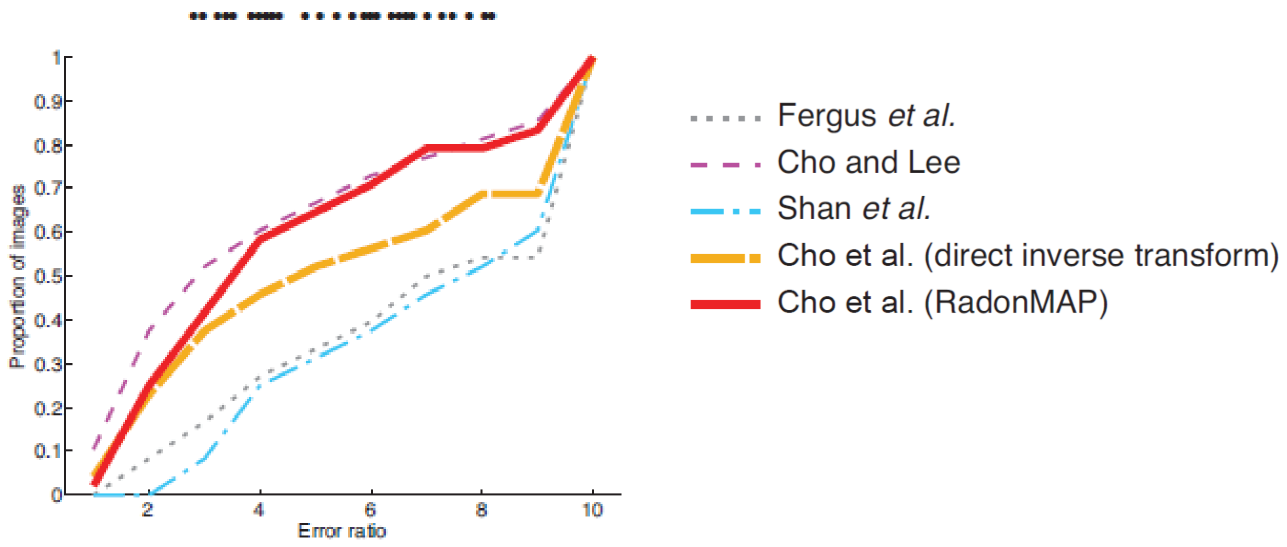


Figure 2-12 : Cumulative error ratios average plot for all images of the image set and for the following algorithms: Fergus et al.[3], Shan et al.[48], Cho and Lee [17], Cho et al.2011 method based on directly inverting the Radon transform[47] and Cho et al. 2011 Radon MAP algorithm[47]. The current image was published in Cho et al. 2011[47].

Image	Size		Processing time (sec.)		
	Image	Kernel	A	B	C
a	972 × 966	65 × 93	2.188	3.546	5.766
b	1024 × 768	49 × 47	1.062	1.047	2.125
c	483 × 791	35 × 39	0.343	0.235	0.578
d	858 × 558	61 × 43	0.406	0.297	0.703
e	846 × 802	35 × 49	0.516	0.406	0.922

Figure 2-13 : Processing times of the deblurring examples. A: kernel estimation. B: final deconvolution. C: total processing time[17]. The current table was published in Cho and Lee 2009[17].

Image	Size		Processing time (sec.)		
	Image	Kernel	A	B	C
Picasso	800 × 532	27 × 19	360	20	0.609
statue	903 × 910	25 × 25	762	33	0.984
night	836 × 804	27 × 21	762	28	0.937
red tree	454 × 588	27 × 27	309	11	0.438

Figure 2-14 : Processing time comparison. A: [Shan et al. 2008]. B: Cho and Lee 2009 method with C++ implementation. C: Cho and Lee 2009 method with GPU acceleration[17]. The current table was published in Cho and Lee 2009[17].

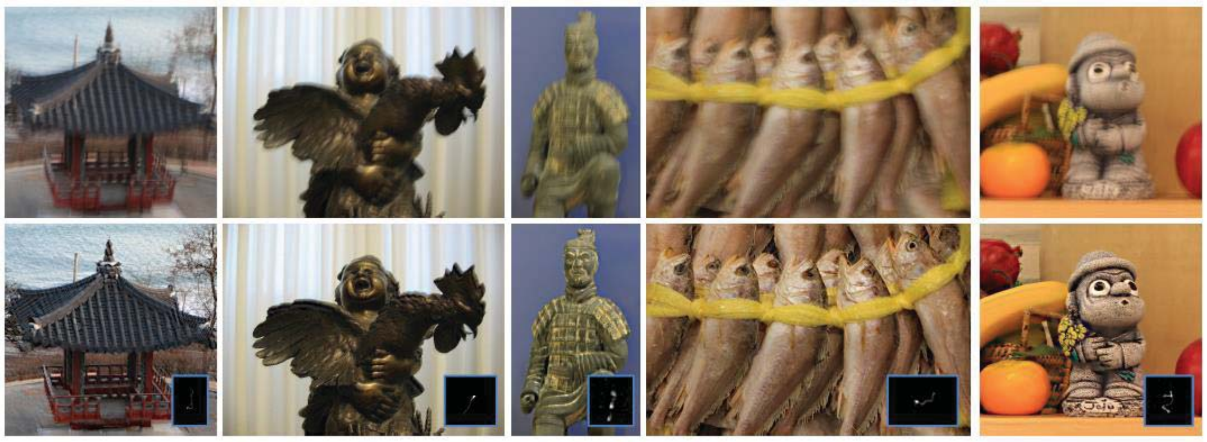


Figure 2-15 : Deblurring results of real photographs. Top row: input blurred images. Bottom row: deblurring results with estimated kernels[17].

In order to focus on a certain method for addressing camera shake removal we have to introduce a set of criteria. Specifically, the algorithm will be implemented on an android platform meaning that we have to take into consideration the embedded CPU processing ability, which is lower than an average desktop processor, and the lower RAM capacity of an ANDROID tablet or phone. Additionally, the implementation has to be as fast as possible in order to become a basis for a near real time application. Based on the aforementioned criteria and metrics we came to focus on the fast motion deblurring algorithm of Cho and Lee 2009 .This method is computationally efficient while yielding satisfactory enough results with relevance to other more computationally expensive methods.

Chapter 3
FAST MOTION DEBLURRING

As it has been analyzed in the previous chapter we chose to focus on the Cho and Lee 2009[17] deblurring method. This chapter analyzes the method presenting the mathematical background and the algorithm used for the implementation. Finally, we present shortly the results of the algorithm since they will be fully discussed in Chapter 4 and in Chapter 7.

3.1 STATING THE PROBLEM

In order to provide a method for dealing with image deblurring it is most important to provide a blur modelling practice. Specifically blur can be modelled in the following manner:

$$B = K * L + N \tag{3-1}$$

where

B	blurred image
K	blur kernel or PSF
L	sharp image
N	noise
*	convolution operation

The blind deconvolution approach involves alternating optimization of L and K iteratively

$$L' = \arg\min_L \left\{ \left\| B - K * L \right\| + \rho_L(L) \right\} \tag{3-2}$$

$$K' = \arg\min_K \left\{ \left\| B - K * L \right\| + \rho_K(K) \right\} \tag{3-3}$$

In the above set of equations ρL and ρK are regularization terms and $\left\| B - K * L \right\|$ is the data fitting term. For the latter term L2 norm is used. The terms ρL and ρK are regularization terms for latent image and kernel respectively.
In order to achieve image deblurring the blur kernel is refined iteratively. The final result comes from the non-blind deblurring of the original RGB image. The images produced during the kernel estimation process have no effect on the final result. In order to estimate the blur kernel latent image estimation in required. This can be performed by using two different procedures: a) Sharp edges restoration and b) noise suppression in smooth areas. Noise suppression is most important because otherwise the noise

might affect the data fitting term $\|B-K*L\|$. After estimating the latent image for a certain iteration the blur kernel is estimated by minimizing eq. (3.3). The blur kernel is the used for fast non-blind deconvolution of the original blurry image. The result, which is still blurry but less blurry than the original image, replaces the blurry input image and the process is repeated until the difference between the estimated blur kernel of the current iteration and the previous is very small. After estimating the blur kernel high quality deblurring is applied at the original image. In order to make the process faster the iterative process is performed in a multi-scale manner as it begins from a very small image which is resized to a larger one in each iteration until it reaches the size of the original image

3.2 PROCESS OVERVIEW

As mentioned before the blur kernel is being refined iteratively through a process including three steps: a) prediction, b) kernel estimation process and c) deconvolution. After the blur kernel has been recovered final deconvolution takes place and thus reveals the sharp image. Prediction and deconvolution processes are part of the sharp image estimation process

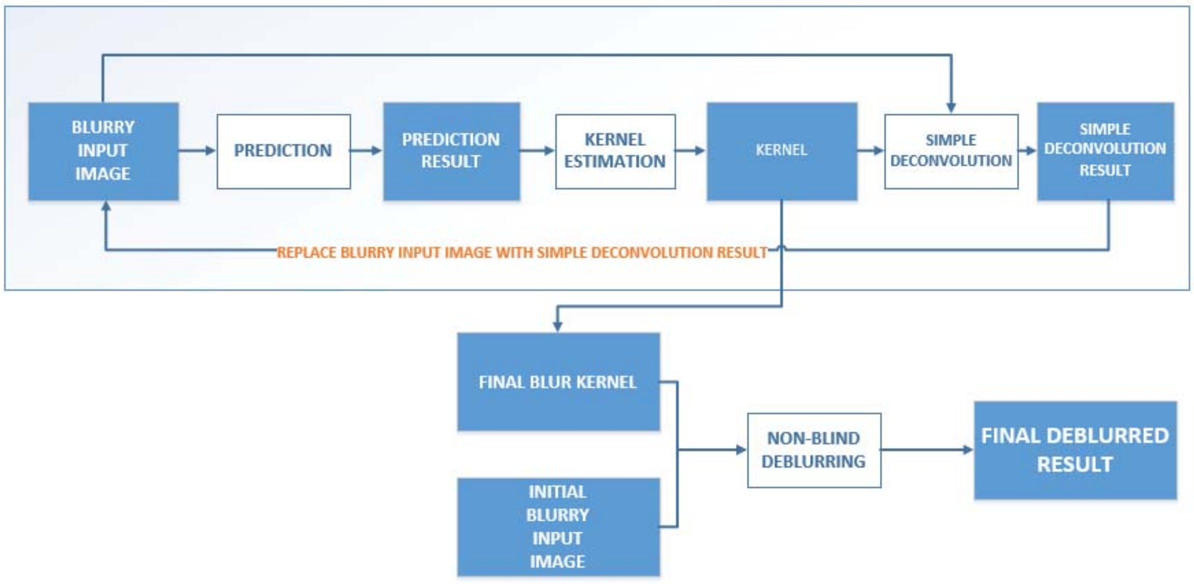


Figure 3-1 : Process overview

3.2.1 PREDICTION

In prediction process, image gradient maps P_x and P_y of the latent image L are estimated. Only strong edges remain while the other regions have zero gradient. The prediction step consists of bilateral filtering, shock filtering. At first apply bilateral filtering[49] is applied to the current estimate of L to suppress possible noise and small details. A shock filter is then used to restore strong edges of L. The resulting image from shock filtering contains strong edges and enhanced noise. A shock filter is an effective filter for enhancing image features. It can recover sharp edges from blurred step signals[13]. The evolution equation of a shock filter is formulated as

$$I_{t+1} = I_t - sign\left(\Delta I_t\right)\left\|\nabla I_t\right\| dt \qquad (3\text{-}4)$$

where I_t is the image at time t, ΔI_t is the Laplacian of image at time t and ∇I_t is the gradient .

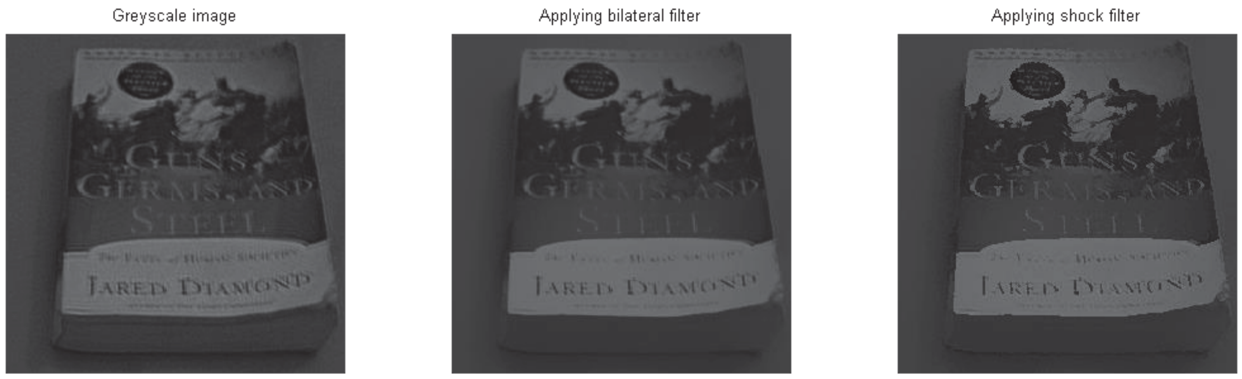


Figure 3-2 : Bilateral and shock filtering

Kernel estimation

In the kernel estimation step, only the salient edges have effect on the estimation process of the kernel (convolution of zero gradients is always zero regardless of the kernel.) In order to estimate a blur kernel using the predicted gradient maps {Px, Py} , the energy function is minimized

$$f_k\left(\mathbf{k}\right) = \left\|\mathrm{K} * P_* - B_*\right\|^2 + \beta\left\|\mathrm{K}\right\|^2 \qquad (3\text{-}5)$$

Each $K * P_* - B_*$ forms a map I. For I it can be assumed that

$$\left\|I\right\|^2 = \sum_{(x,y)} I\left(x,y\right)^2 \qquad (3\text{-}6)$$

where (x, y) are the indices of a pixel in I. β is a weight for Tikhonov regularization[32]. In this method only image derivatives are used without including the pixel values in the minimization function. Additionally the minimization function includes a Tikhonov regularization term.

The equation for kernel estimation can be written in the following manner[17]:

$$fk(\mathbf{k}) = \|\mathbf{Ak}-\mathbf{b}\|^2 + \beta\|\mathbf{k}\|^2 = (\mathbf{Ak}-\mathbf{b})^T(\mathbf{Ak}-\mathbf{b}) + \beta\mathbf{k}^T\mathbf{k} \tag{3-7}$$

Where:
A is a matrix representation of P
k is a matrix representation of the blur kernel
b is a matrix representation of B

To minimize f(k) a conjugate gradient method is employed. Thus, the gradient of f(k) in equation (3-8) is formulated in the following manner:

$$\frac{\partial f_k(\mathbf{k})}{\partial \mathbf{k}} = 2\mathbf{A}^T\mathbf{Ak} + 2\beta\mathbf{k} - 2\mathbf{A}^T\mathbf{b} \tag{3-8}$$

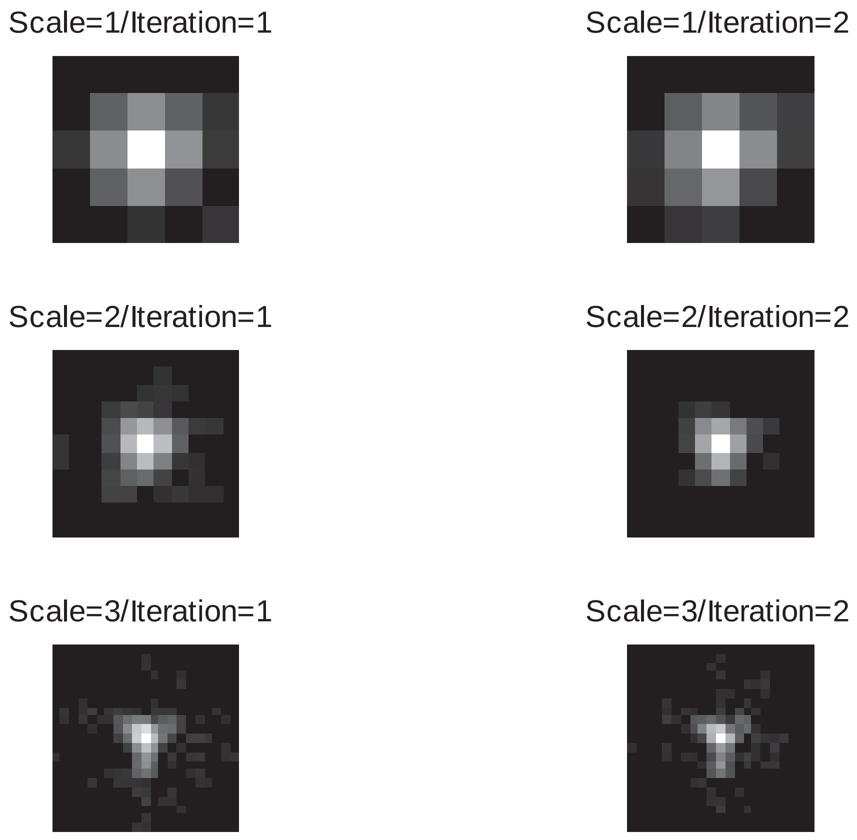


Figure 3-3 : Kernel iterative refinement

3.2.2 DECONVOLUTION

In the deconvolution step, we estimate the latent image L from a given kernel K and the input blurred image B. We use the energy function

$$f_L\left(L\right)=\sum_{\vartheta_*}\omega_*\left\|\mathrm{K}*\partial_* L-\partial_* B\right\|^2+\alpha\left\|\nabla L\right\|^2 \tag{3-9}$$

where $\vartheta_* \in \left\{\vartheta_0, \vartheta_x, \vartheta_y, \vartheta_{xx}, \vartheta_{yy}, \vartheta_{xy}\right\}$ denotes the partial derivative operator in different directions and orders $\omega_* \in \left\{\omega_0, \omega_1, \omega_2\right\}$ is a weight for each partial derivative, and a is a weight for the regularization term.

The first term in the energy is based on the blur model of Shan et al. 2008, which uses image derivatives for reducing ringing artifacts. The regularization term enchances favors $\left\|\nabla L\right\|^2$ with smooth gradients[10]. The recovery of the latent image L is performed by optimizing Eq. (3-10) . This is achieved by a very fast pixel-wise division in the frequency domain, which needs only two FFTs[17].
Optimizing Eq. (3-10) may not produce high-quality results, compared to sophisticated deconvolution methods[10], [48], [50], and the results can contain smoothed edges and ringing artifacts. However, due to the prediction step that sharpens edges and discards small details, this simple deconvolution creates satisfactory results. Below in figure 3-4 appears the iterative deblurring procedure. The RGB picture shows the final deconvolution result.

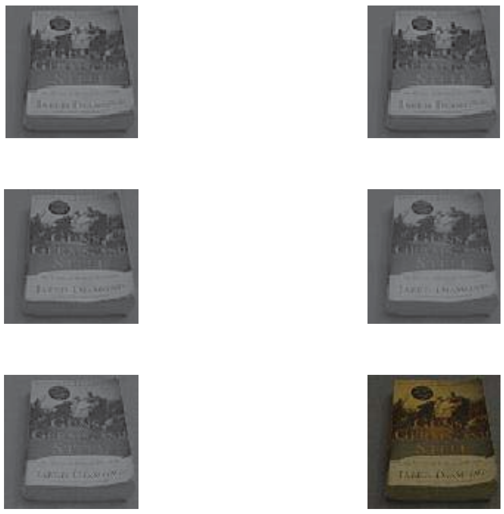


Figure 3-4 : Iterative deblurring procedure. The RGB picture shows the final deconvolution result.

3.2.3 FINAL DECONVOLUTION

Final deconvolution is the final step for extracting the sharp image. There is no difference between the method used for deconvolution during iterations and the method used for final deconvolution. The final deconvolution is applied on each channel of the initial blurry image using the refined kernel that has been estimated iteratively. The following figure shows the final deconvolution result.

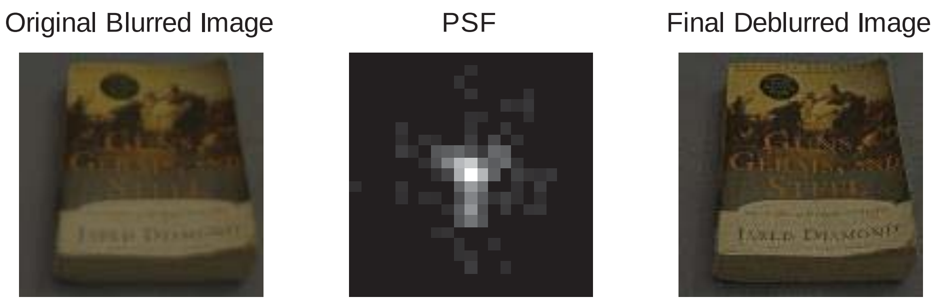


Figure 3-5: Final deconvolution process using the estimated blur kernel

After the final deconvolution the sharp image is recovered. The method analyzed in this chapter is going to be implemented in MATLAB and afterwards in C++ in order to be integrated into an ANDROID application. implementation of the aforementioned algorithm is analyzed in the following chapter.

Chapter 4
MATLAB PROTOTYPING

This chapter contains the MATLAB implementation for the fast motion deblurring algorithm. Specifically, the MATLAB implementation is divided in several parts. These are: i) initialization, ii) main loop iii) prediction iv) kernel estimation v) deconvolution and vi) final deconvolution. Additionally, a set of image filtering tools where used for the implementation of the aforementioned functions such as bilateral and shock filtering. All of the above are analyzed in the following subsections.

4.1 INITIALIZATION BLOCK

The initialization block defines parameters for the deblurring procedure. These are the maximum kernel size assumption, the number of iterations for the core deconvolution process, the alpha parameter describing how hard the deconvolution will be, the frequency cutoff parameter describing the magnitude limit for a part of the frequency limit to be used in the deblurring kernel, and the parameters of the bilateral filter. These parameters define the course of the process. For example, too much alpha would lead to a lot of noise and too small would lead to a blurry result. The figure bellow contains the MATLAB code for the initialization block.

```
%% Initialization
initsize=21;
iterScaleNum=20;
alpha=50;
belta=2;
freqCut=10;
%% readImage
img=pimg;
%% initialization
sigmaSpatial=0;
sigmaRange=0;
OriginalBlurred=img;
imgSize=size(img,1);
isFinal=false;
mode=0;
```

Figure 4-1 : MATLAB code for initialization block.

4.2 INITIAL KERNEL ANALYSIS BLOCK AND SCALES SIZES DEFINITION

This block produces the initial kernel size for each iteration. This way a kernel pyramid is created in order to refine the kernel in a multiscale manner. This is achieved by seperating the kernel side length to as many levels as possible. This is useful because the procedure is iterative starting from the lowest level meaning the smallest kernel and is gradually resized untill reaching the maximum size. The aforementioned process makes the deblurring process effinent.

```
%% analyzeKernel
    currentsize=initsize;
    i=1;
    flag=true;
    while flag
        kernelPyramid(i)=currentsize;
        currentsize=floor(currentsize/2);
        if currentsize<=5
            if rem(currentsize,2)==0
                currentsize=currentsize+1;
            end
            kernelPyramid(i+1)=currentsize;
            flag=false;
            break;
        end
        if rem(currentsize,2)==0
            currentsize=currentsize+1;
        end
        i=i+1;
    end
```

Figure 4-2 : MATLAB code for kernel analysis block.

This kernel produces the image pyramid. For each scale of the kernel a corresponding image size is preset . Implementing this addition we avoid inconsistencies in the way the resizing process takes place. The following block contains the MATLAB code genarating the scale kernel scales.

```
%% Define scales
Scales=length(kernelPyramid);
scalesArray = zeros(Scales,2);
[OriginalBlurredX,OriginalBlurredY]=size(r2g(OriginalBlurred));
for j=1:Scales
    i=Scales-j;
ScalesDenominator=2^(i);
ScalesFactor=1/ScalesDenominator;
scalesArray(j,1)=floor(ScalesFactor*OriginalBlurredX);
scalesArray(j,2)=floor(ScalesFactor*OriginalBlurredY);
end
%% Resize Original to minimum scale
OriginalBlurredScaledMinSingleChannel=r2g(imresize(OriginalBlurred,[scalesArray(1,1),scalesArray(1,2)]));
```

Figure 4-3 : MATLAB code for kernel pyramid generation block.

4.3 MAIN LOOP

The main loop section contains the iterative deblurring process. For each scale of kernel and image the kernel is being estimated and the deblurred image for the current scale is being produced. Initially the image is resized to the kernel size for the current scale. Then the PSF and the latent image are estimated. Finally, the latent image is resized to a greater level in order to perform the same procedure. The figure below presents the main loop MATLAB code

```
%% Main
for thisIter=1:Scales
    if thisIter==Scales
        isFinal=true;
    end
%Resize
     CurrentIterationFactor=Scales-thisIter+1;
    OriginalBlurredScaled=imresize(OriginalBlurred,[scalesArray(thisIter,1),scalesArray(thisIter,2)]);
    sizef=kernelPyramid(CurrentIterationFactor);
    InitialPSF=zeros(sizef,sizef);

[deblurredImage,PSFResult]=getImagePSF(OriginalBlurredScaled,OriginalBlurredScaledMinSingleChannel,InitialPSF,iterS
caleNum,isFinal,OriginalBlurred,sigmaSpatial,sigmaRange,alpha,belta,freqCut,mode,thisIter);
    if thisIter<Scales

OriginalBlurredScaledMinSingleChannel=imresize(deblurredImage,[scalesArray(thisIter+1,1),scalesArray(thisIter+1,2)]);
    end
end
pdeblurredImage=deblurredImage;
pPSFResult=PSFResult;
```

Figure 4-4 : MATLAB code for main loop block.

4.3.1 KERNEL ESTIMATION AND DECONVOLUTION

The PSF and a corresponding latent image is produced at each iteration of the deblurring procedure. This is achieved in the getImagePSF() function. The process receives as input the blurry image for the current scale, an empty Mat with size equivalent to the kernel size for the current scale, the number of iterations and the bilateral filter parameters. Finally the function produces the estimated kernel and the estimated latent image for the current scale.

```
function [deblurredImage,PSFResult]
=getImagePSF(OriginalBlurredScaled,OriginalBlurredMinimumScaleSingleChannel,PSF,iterScaleNum,isFinal,OriginalBlurr
ed,sigmaSpatial,sigmaRange,alpha,belta,freqCut,mode,currentScale)
Temp=OriginalBlurredMinimumScaleSingleChannel;
%Start loop
for iteration=1:iterScaleNum
```

Figure 4-5 : MATLAB code for kernel estimation and deconvolution block for a given input image(Part I).

```
for iteration=1:iterScaleNum
%initialize bilateral
[inputWidth,inputHeight] = size(Temp);
sigmaSpatial = min( inputWidth, inputHeight ) / 16;
if sigmaRange==0
edgeMin = min( Temp( : ) );
edgeMax = max( Temp( : ) );
edgeDelta = edgeMax - edgeMin;
sigmaRange = 0.1 * edgeDelta;
end
figure(4);
subplot(3,3,(currentScale-1)*3+1);
imshow(Temp);
title('Greyscale image');
%Temp = medfilt2(Temp);
Prediction=bilateralFilter(Temp,Temp,sigmaSpatial,sigmaRange,sigmaSpatial,sigmaRange);
subplot(3,3,(currentScale-1)*3+2);
imshow(Prediction);
title('Applying bilateral filter');
Prediction=shock(Prediction,30,0.9^(iteration-1),1,'org');
subplot(3,3,(currentScale-1)*3+3);
imshow(Prediction);
title('Applying shock filter');
OriginalBlurredScaledSingleChannel=r2g(OriginalBlurredScaled);
figure(10);
subplot(3,2,((currentScale-1)*2+iteration))
PSFResult=estK(Prediction,OriginalBlurredScaledSingleChannel,PSF,10,belta,freqCut,mode);
PSFMaxValue=max(max(PSFResult));
PSFResultImage=PSFResult/PSFMaxValue;
imshow(PSFResultImage),title(strcat('Scale=',num2str(currentScale),'/','Iteration=',num2str(iteration)));
```

Figure 4-6 : MATLAB code for kernel estimation and deconvolution block for a given input image(Part II).

```
figure(14);
if ((iteration~=iterScaleNum)||(~isFinal))
        deblurredImage=deconv(OriginalBlurredScaledSingleChannel,PSFResult,alpha);
        Temp=deblurredImage;
end
if ((iteration==iterScaleNum)&&(isFinal))
        deblurredImage=deconv(OriginalBlurred,PSFResult,alpha);
end
subplot(3,2,((currentScale-1)*2+iteration))
imshow(deblurredImage);
end
return;
```

Figure 4-7 : MATLAB code for kernel estimation and deconvolution block for a given input image(Part III).

4.3.2 PREDICTION

Prediction process includes bilateral filtering and shock filtering. The bilateral filter enhances the strong edges whiles smoothing smooth areas and the shock filter enhances strong edges. The prediction block is shown in the following block:

```
Prediction=bilateralFilter(Temp,Temp,sigmaSpatial,sigmaRange,sigmaSpatial,sigmaRange);
Prediction=shock(Prediction,30,0.9^(iteration-1),1,'org');
```

Figure 4-8 : MATLAB code for prediction process block

- **Shock filtering**

Shock filtering [13], [51]enhances strong edges. It uses iteratively dilation and erosion to create ruptures at the edge pixels. This process makes shock filters applied to edge enhancement and segmentation. The following blocks include the MATLAB implementation of chock filtering as it was implemented by Guy Gilboa[52] . This function evolves the image according to shock filtering process . It receives as input the image, timestep dt, size of grid steps h.

```
function I=shock(I0,iter,dt,h,meth,Par)
if ~exist('dt')
    dt=0.1;
end
if ~exist('h')
    h=1;
end
if ~exist('meth')
    meth='org';
end
if ((meth=='cmp') & (length(Par)<4))
    Par(4)=pi/1000;    % default theta value
end
[ny,nx]=size(I0);
I=I0;
h_2 = h^2;
% compute flow
```

Figure 4-9 : MATLAB code for shock filter (Part I)

```
for i=1:iter,    %% do iterations
    % estimate derivatives (Newmann BC)
    IShiftColsForward=I(:,[1 1:nx-1]);
    IShiftColsBackwards=I(:,[2:nx nx]);
    IShiftRowsForward=I([1 1:ny-1],:);
    IShiftRowsBackwards=I([2:ny ny],:);
    I_mx = I-IShiftColsForward;
    I_px = IShiftColsBackwards-I;
    I_my = I-IShiftRowsForward;
    I_py = IShiftRowsBackwards-I;
    I_x = (I_mx+I_px)/2;
    I_y = (I_my+I_py)/2;
    % minmod operator
    Dx = min(abs(I_mx),abs(I_px));
    ind=find(I_mx.*I_px < 0); Dx(ind)=zeros(size(ind));
    Dy = min(abs(I_my),abs(I_py));
    ind=find(I_my.*I_py < 0); Dy(ind)=zeros(size(ind));
     % estimate derivatives
        I_xx=IShiftColsBackwards+IShiftColsForward-2*I;
        I_yy=IShiftRowsBackwards+IShiftRowsForward-2*I;
         I_xy = (I_x([2:ny ny],:)-I_x([1 1:ny-1],:))/2;
    % compute flow
    a_grad_I = sqrt(Dx.^2+Dy.^2); % Abs Gradient of I
    a2_grad_I = (abs(I_x)+abs(I_y)); % second order abs grad
    dl=0.00000001;    % small delta
    I_nn = I_xx.*abs(I_x).^2 + 2*I_xy.*I_x.*I_y + I_yy.*abs(I_y).^2;
    I_nn = I_nn./(abs(I_x).^2+abs(I_y).^2+dl);
    I_ee = I_xx.*abs(I_y).^2 - 2*I_xy.*I_x.*I_y + I_yy.*abs(I_x).^2;
    I_ee = I_ee./(abs(I_x).^2+abs(I_y).^2+dl);
    ind = find(a2_grad_I==0);    % zero gradient n,e not defined
    I_nn(ind)=I_xx(ind);
    I_ee(ind)=I_yy(ind);
```

Figure 4-10 : MATLAB code for shock filter (Part II)

```
    if (meth=='org')
        I_t = -sign(I_nn).*a_grad_I/h;
    elseif (meth=='cmp')
      lam=Par(1); lam_tld=Par(2); a=Par(3); theta=Par(4);
      i=sqrt(-1); lam=lam*exp(i*theta);
      I_t = -atan(a/theta*imag(I)).*a_grad_I/h + lam*I_nn/h_2 + lam_tld*I_ee/h_2;
    elseif (meth=='alv')
     c=Par(1); sigma2=Par(2);
      g_I_nn=gauss(I_nn,51,sigma2);
      I_t = - sign(g_I_nn).*a_grad_I/h + c*I_ee/h_2;
    else
      error(['unknown method ' meth])
    end    % if
    I=I+dt*I_t;    %% evolve image by dt
end % for i
```

Figure 4-11 : MATLAB code for shock filter (Part III)

- **Bilateral filtering**

A bilateral filter is a non-linear, edge-preserving and noise-reducing smoothing filter for images. The intensity value at each pixel in an image is replaced by a weighted average of intensity values from nearby pixels. This weight can be based on a Gaussian distribution. Crucially, the weights depend not only on Euclidean distance of pixels, but also on the radiometric differences (e.g. range differences, such as color intensity, depth distance, etc.). This preserves sharp edges by systematically looping through each pixel and adjusting weights to the adjacent pixels accordingly.

4.3.3 KERNEL ESTIMATION

Kernel estimation is implemented by function estK().This function performs minimization of the cost function assuming that the prediction result is the sharp image L. The result of the minimization yields the optimal blur kernel K for the current iteration. The following blocks contain the MATLAB implementation for the kernel estimation procedure.

```
function ker=estK(Prediction,OriginalBlurredScaledSingleChannel,PSF,numberOfIterations,belta,freqCut,mode)
%Q:Prediction , P:Result , f:kernel
ex=PSF;error=1;iteration=1;ks=getPadSize(PSF);ksi=2*ks;
PredictionPadded=padarray(Prediction,ks);
BlurredPadded=padarray(OriginalBlurredScaledSingleChannel,ks, 'replicate', 'both');
while (error>0.0001)&&(iteration<numberOfIterations)
    ker=getK(PredictionPadded,BlurredPadded,PSF,belta,freqCut);%Q:Prediction , P:Result , f:kernel
    error=norm(ker-ex,2);ex=ker;
    if mode==0
    elseif mode==1
    BlurredPadded=OriginalBlurredScaledSingleChannel;
    BlurredPadded=padarray(OriginalBlurredScaledSingleChannel,ks, 'replicate', 'both');
    end
    iteration=iteration+1;
end
return;
```

Figure 4-12 : MATLAB code for kernel estimation (Part I).

```
function ker=getK(PredictionPadded,BlurredPadded,PSF,belta,freqCut) %Prediction , Result , kernel
A=fft2(PredictionPadded);      %Prediction
b=fft2(BlurredPadded);      %Result
ker=delta_kernel(size(PSF));
for i=1:1
        k=psf2otf(ker,size(A));
        d=conj(A).*b-(conj(A).*A+belta).*k;
        rentaNom=(conj(A.*d).*b-conj(A.*d).*A.*k-belta.*conj(d).*k);
        rentaDeNom=(conj(A.*d).*A.*d+belta*conj(d).*d);
        renta=rentaNom./rentaDeNom;
      k=k+renta.*d;

        ker=otf2psf(k,size(PSF));
        ker=abs(ker);
        maxValue=max(max(ker));
        pos_ind=find(ker<maxValue/freqCut);
        ker(pos_ind)=0;
        thesum=sum(sum(ker));
        ker=ker/thesum;
end
return;
```

Figure 4-13 : MATLAB code for kernel estimation (Part II).

4.4 DECONVOLUTION STEP

The deconv() function deconvolves the estimated kernel K with the original blurry image B in order to estimate the sharp image L for the current iteration. This function contains a preprocessing step for the blurred image by padding it before applying the deconvolution core function deconv_fn().

```
function deblurred_image=deconv(Blurred,PSF,w0alpha)
ks = (size(PSF) - 1)/2;
Blurred = padarray(Blurred, ks, 'replicate', 'both');
[theResult] = deconv_fn(Blurred, PSF, w0alpha);
d=size(theResult);
xmin=ks(1)+1;
ymin=ks(2)+1;
width=(d(1)-ks(1)-ks(1)-1);
height=(d(2)-ks(2)-ks(2)-1);
e=[xmin ymin height width];
deblurred_image = imcrop(theResult,e);
return;
```

Figure 4-14 : MATLAB code for deconvolution step (Part I)

The deconv_fn function is the core deconvolution function. It receives as input the blurred image and the point spread function and produces the latent image L.

```
function L=deconv_fn(Blurred,PSF,w0alpha)
k0=1;
kx = [1 -1]; ky = [1 -1]';kxx = [1 -2 1]; kyy = [1 -2 1]';kxy=[1 -1;-1 1];
Fk0=psf2otf(k0,size(Blurred));
Fkx=psf2otf(kx,size(Blurred));
Fky=psf2otf(ky,size(Blurred));
Fkxx=psf2otf(kxx,size(Blurred));
Fkyy=psf2otf(kyy,size(Blurred));
Fkxy=psf2otf(kxy,size(Blurred));
delta0=Fk0.*conj(Fk0);delta1=Fkx.*conj(Fkx);delta2=Fky.*conj(Fky);delta3=Fkxx.*conj(Fkxx);delta4=Fkxy.*conj(Fkxy);
delta5=Fkyy.*conj(Fkyy);delta12=delta1+delta2;delta345=delta3+delta4+delta5;
delta=w0alpha*(delta0+0.5*delta12+0.25*delta345);FFTy=fftn(Blurred);
FFTK=psf2otf(PSF,size(Blurred));
FFTLnom=(FFTy.*conj(FFTK));
FFTLnom=FFTLnom.*delta;
FFTLdenom=((FFTK.*conj(FFTK)).*delta+(Fkx.*conj(Fkx))+(Fky.*conj(Fky)));
FFTL=FFTLnom./FFTLdenom;
L=ifftn(FFTL);
return;
```

Figure 4-15 : MATLAB code for deconvolution step (Part II).

The following section concludes this chapter by presenting the results of this implementation. In order to conclude this section it is important to mention that the implementation had as prequisite the as fast as possible

4.5 RESULTS

This section presents the results of the implemented method using a test set of four images. The execution took place in a Desktop PC Intel Core i7 with 8GB of RAM. In this section we present the results from three test images.

- **Fruit basket test image**

In this subsection we present the results of the method we implemented when applied on a blurry test image and specifically the fruit basket test image. Figure 4-16 shows the deblurring process results and the kernel refinement process.

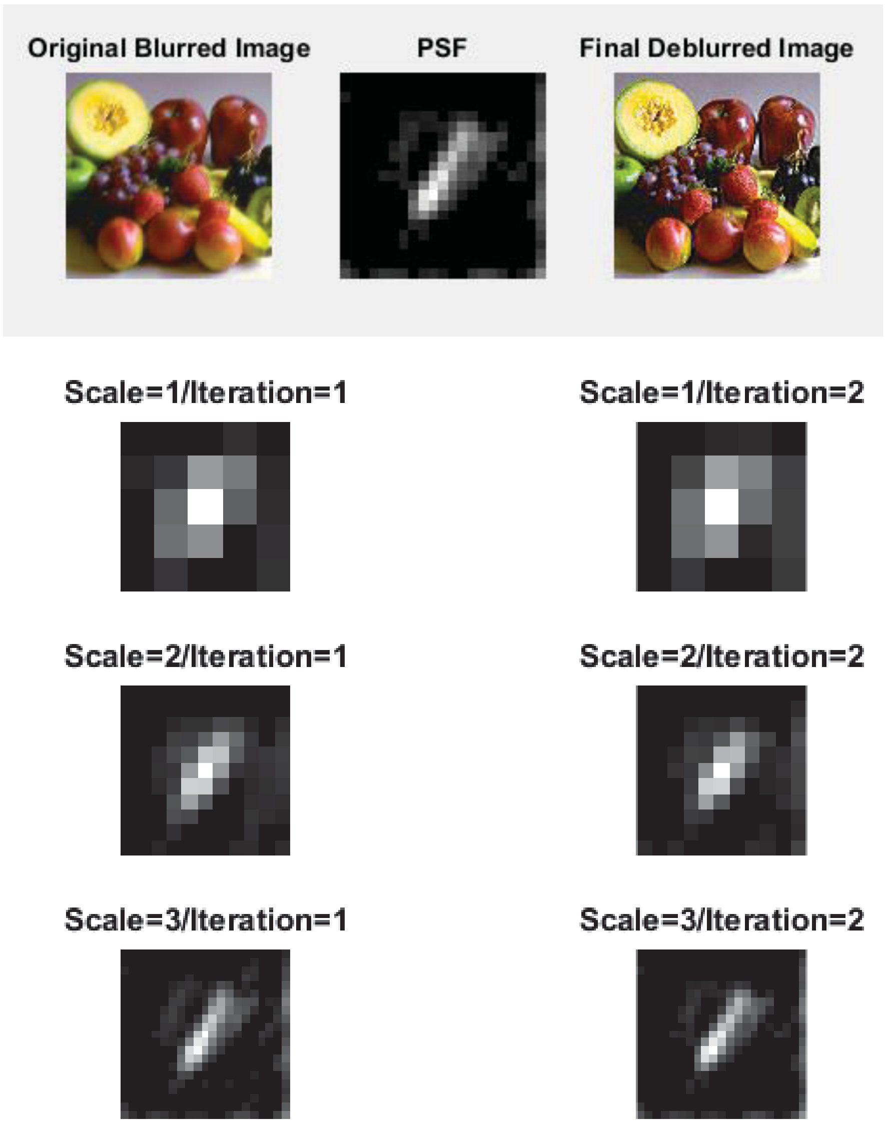


Figure 4-16 : Fruit basket blurry test image deblurring process. At the top of the figure we present the original blurry image, the point spread function and the product of our algorithm. Below we present the iterative kernel refinement process

- **City photo test image**

In this subsection we present the results of the method we implemented when applied on a blurry test image and specifically the city photo test image which is also a greyscale image. Figure 4-17 shows the deblurring process results and the kernel refinement process.

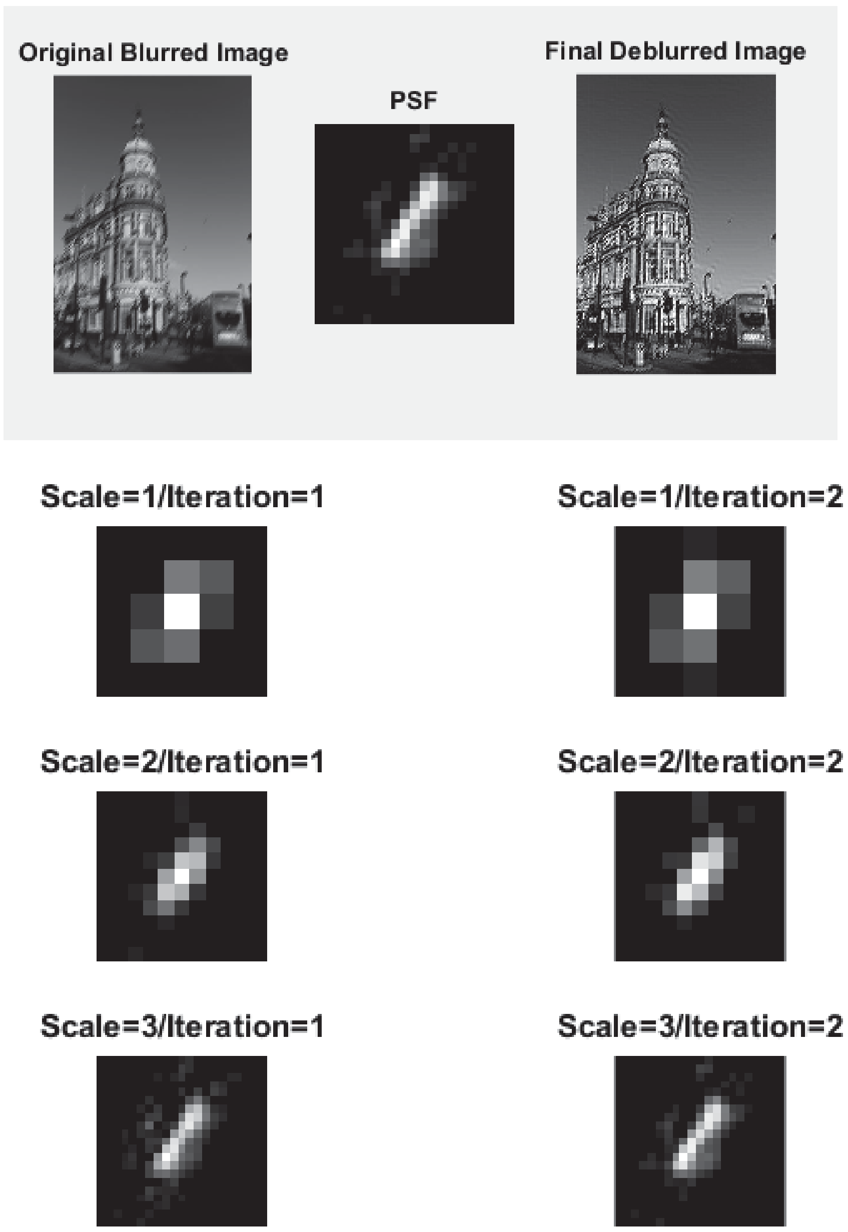


Figure 4-17 : City photo blurry test image deblurring process. At the top of the figure we present the original blurry image, the point spread function and the product of our algorithm. Below we present the iterative kernel refinement process

- **Book test image**

In this subsection we present the results for the book test image which is also a greyscale image. Figure 4-18 shows the deblurring process results and the kernel refinement process.

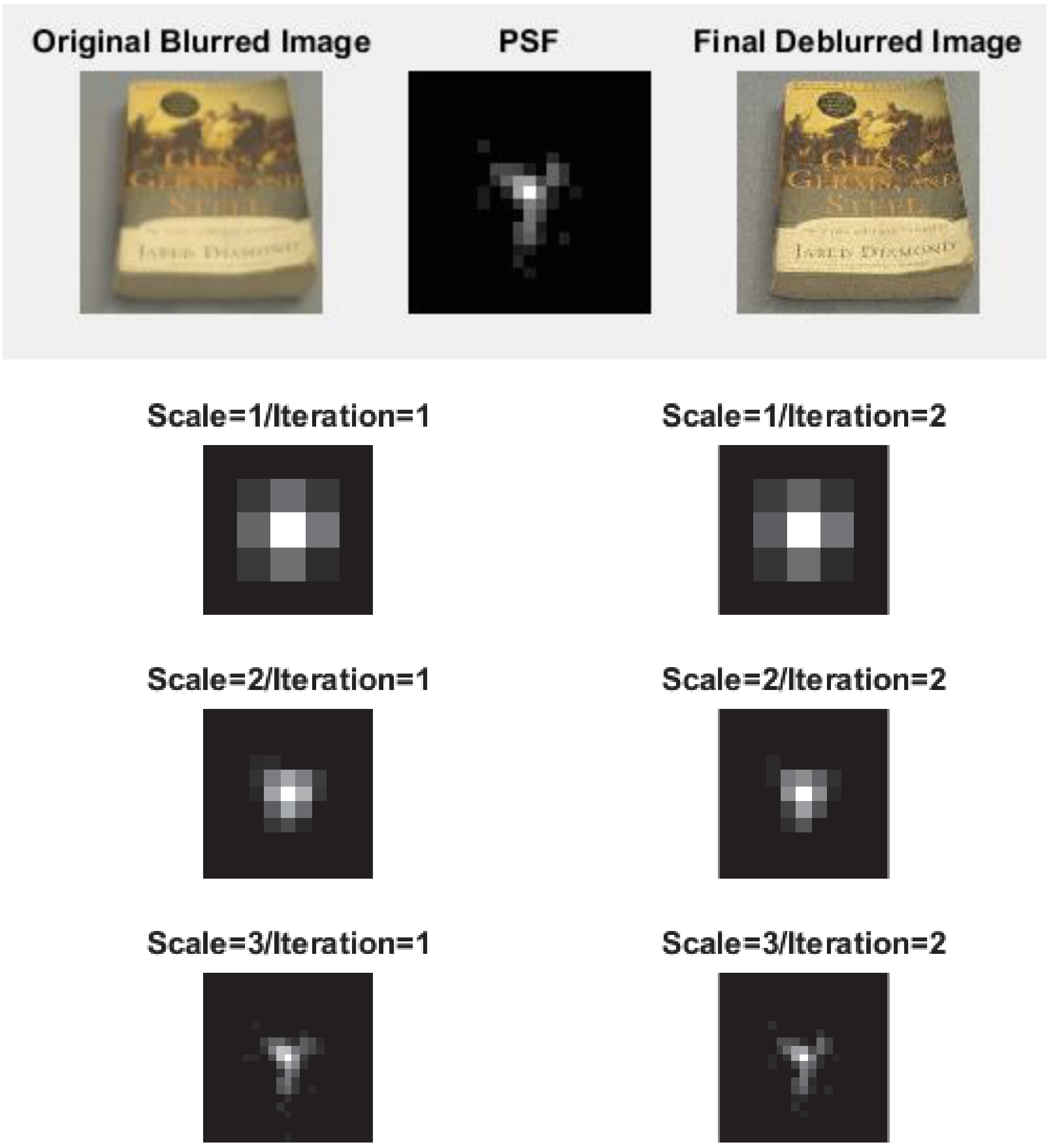


Figure 4-18 : Book blurry test image deblurring process. At the top of the figure we present the original blurry image, the point spread function and the product of our algorithm. Below we present the iterative kernel refinement process

Finally in the table below we present comparative results for different kernel and image sizes.

Table 4-1 : Rsults for MATLAB implementation

Image	Kernel Size	Image Size	Time
Fruit Basket	21x21	512x512	4.86
City	21x21	341x512	2.89
Book	21x21	252x252	1.65

Chapter 5
C/C++ IMPLEMENTATION

C++ implementation is based on the OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products. Being a BSD-licensed product, OpenCV makes it easy for businesses to utilize and modify the code.

The library has more than 2500 optimized algorithms, which includes a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms. These algorithms can be used to detect and recognize faces, identify objects, classify human actions in videos, track camera movements, track moving objects, extract 3D models of objects, produce 3D point clouds from stereo cameras, stitch images together to produce a high resolution image of an entire scene, find similar images from an image database, remove red eyes from images taken using flash, follow eye movements, recognize scenery and establish markers to overlay it with augmented reality, etc. OpenCV has more than 47 thousand people of user community and estimated number of downloads exceeding 7 million. The library is used extensively in companies, research groups and by governmental bodies.

Along with well-established companies like Google, Yahoo, Microsoft, Intel, IBM, Sony, Honda, Toyota that employ the library. OpenCV’ s deployed uses span the range from stitching street-view images together, detecting intrusions in surveillance video in Israel, monitoring mine equipment in China, helping robots navigate and pick up objects at Willow Garage, detection of swimming pool drowning accidents in Europe, running interactive art in Spain and New York, checking runways for debris in Turkey, inspecting labels on products in factories around the world on to rapid face detection in Japan.

It has C++, C, Python, Java and MATLAB interfaces and supports Windows, Linux, Android and Mac OS. OpenCV leans mostly towards real-time vision applications and takes advantage of MMX and SSE instructions when available.

5.1 THE DEBLUR CLASS

This section analyzes the main class used to implement the deblurring in an object oriented manner. Specifically, the deblur class contains all the parameters required to initialize the deblur procedure and all the functions to perform deblurring. In the following subsections, the function members and the data members of the deblur class will be presented.

```
class deBlur{
public:
        //GENERAL PURPOSE
        std::string filename;
        Mat color, img;
        Mat grayScaleImage;
        //SHOCK FILTER
        Mat shockFiltered;
        //BILATERAL FILTER
        Mat bilateralResult;
        double sigmaColor;
        double sigmaSpace;
        double sigmaRange;
        double shockFilterdt;
        //DECONVOLUTION RELATED
        double belta;
        double alpha;
        double freqCut;
        //KERNEL RELATED
        int initsize;
        //MULTISCALE APPROACH AND ITERATIVE APPROACH RELATED
        int iterScaleNum;
        int scales;
        int shockFilterIter;
        double scalesDenominator, scalesFactor;
        std::vector<int> kernelArray;
        bool isFinal;
        Mat OriginalBlurredScaledMinSingleChannel;
        Mat OriginalBlurredScaledSingleChannel;
        Mat OriginalBlurred;
        Mat OriginalBlurredScaled;
        Mat InitialPSF;
        int CurrentIterationFactor, sizef;
        double riterDenominator, riterFactor;
        //OUTPUT MATRICES
        Mat deblurredImage;
        Mat PSFResult;
        //INITIALIZATION FUNCTIONS
        void analyzeKernel(void);
        void resizeBlurredToMinimumScale(void);
        //IO
        void readImage(void);
        void readImage(string);
        void readImage(Mat theImage);
```

Figure 5-1 : C++ deBlur class code (Part I)

```
        //ITERATIVE DEBLURRING
        void iterativeDeblurring(void);
        void getImagePSF(Mat &OriginalBlurredScaled, Mat &OriginalBlurredMinimumScaleSingleChannel, Mat &PSF, int
&iterScaleNum, bool &isFinal, Mat &OriginalBlurred, double &sigmaRange);
        //FILTERING
        Mat shock_filter(Mat IO, int iter, double dt, double h);

        //FFT RELATED----------------------------------------
        Mat fft2(Mat input);
        Mat ifft2(Mat input);

        //PSF to OTF & reverse------------
        Mat psf2otf(Mat inputPSF, Size finalSize);
        Mat otf2psf(Mat inputOTF, Size outSize);

        //KERNEL RELATED-------------------------------------
        Mat estK(Mat Prediction, Mat OriginalBlurredScaledSingleChannel, Mat PSF, int numberOfIterations);
        Mat getKMulSpectrumSupport(Mat PredictionPadded, Mat BlurredPadded, Mat PSF, double belta);
        Mat delta_kernel(int s);

        //PADDING RELATED----------------------------------------------------
        Mat paddarray(Mat input, Size padding, std::string method, std::string direction);
        Size getPadSize(Mat f);

        //GRAYSCALE RELATED----------------------------------------------------
        Mat r2g(Mat input);

        //DECONVOLUTION RELATED
        Mat deconv(Mat &Blurred, Mat &PSF, double &w0alpha);
        Mat deconv_fnmulComplexMats(Mat Blurred, Mat PSF, double w0alpha);
        Mat deconv_fn(Mat &Blurred, Mat &PSF, double &w0alpha);
};
```

Figure 5-2 : C++ deBlur class code (Part II).

5.1.1 DATA MEMBERS

The deblur class includes the following data member categories. General purpose related, shock filter related, bilateral filter related, deconvolution related, kernel estimation related, multiscale approach related and output related data members. The general purpose data members are the image filename RGB and greyscale image matrices. The shock filtering and bilateral filtering data members include filter parameters and matrices used for storing the filter result. The deconvolution related data members include deconvolution related parameters. These are the alpha the belta. The kernel related data members include only the initial size integer. The multiscale iterative approach data members include the kernel pyramid and the image pyramid variables. Finally, the output related data members include the deblurred image and the final PSF matrices.

5.1.2 FUNCTION MEMBERS

Function members are divided in several categories. These are the initialization functions, the I/O functions, the iterative deblurring functions, the filter and Fourier transform functions, the optical transfer function related function members, the kernel estimation related function members, the deconvolution related function members and finally the grayscale and padding related auxiliary function members. In this section, each function member will be analyzed.

- **void analyzeKernel(void)**

This function analyzes the kernel based on the initial size into the number of levels used for the iterative deblurring including the kernel size for each level. Thus this function builds the kernel pyramid

- **void resizeBlurredToMinimumScale (void)**

This function resizes the blurry image to the minimum scale so as to commence the deblurring procedure.

- **void readImage(void)**

This function reads the input image normalizes the input image mat from 0 to 1. Additionally, this function is overloaded in case a string for the filename is used as input of a Mat object containing the blurred image.

- **void iterativeDeblurring(void)**

The iterative deblurring function is the mail loop function. For each iteration the latent image is predicted and the kernel estimated. After completing the iteration, the output is resized to the greater scale. It must be noted that the scales are predefined the in the initializations section

- **void getImagePSF(Mat &OriginalBlurredScaled, Mat** &OriginalBlurredMinimumScaleSingleChannel, Mat &PSF, int &iterScaleNum, bool &isFinal, Mat &OriginalBlurred, double &sigmaRange);

This function receives as input the blurry image scaled for the current iteration, an empty Mat for the blur kernel pre-resized for the current iteration scale, a flag to determine if the current iteration is final or not and parameters for the bilateral filter.

- **Mat shock_filter(Mat IO, int iter, double dt, double h);**

The shock filter function is an OpenCV implementation of the work of Guy GIlboa based on the work of Rudin and Osher [13] .

- **Mat fft2(Mat input);**

The fft2 function is the equivalent function for image fast fourier transform. It is implemented using the dft OpenCV function.

- **Mat ifft2(Mat input);**

This function uses the inverse dft function as is at the OpenCV library in order to implement the inverse fft function for images.

- **Mat psf2otf(Mat inputPSF, Size finalSize);**

The psf2otf produces the optical transfer function from a certain blur kernel.

- **Mat otf2psf(Mat inputOTF, Size outSize);**

The otf2psf function creates the psf taking as input an optical transfer function.

- **Mat estK(Mat Prediction, Mat OriginalBlurredScaledSingleChannel, Mat PSF, int numberOfIterations);**

The estK function estimates the blur kernel by receiving as input the predicted latent image and the blurry image. The operation is performed iteratively. For each iteration the get K function is executed which gives the kernel

- **Mat getKMulSpectrumSupport(Mat PredictionPadded, Mat BlurredPadded, Mat PSF, double belta);**

The get kernel function returns the blur kernel by receiving as input the predicted latent image and the blurry image

- **Mat delta_kernel(int s);**

The delta kernel function creates a kernel comprising of zeros with only one peak at the center of the kernel equal to the max value being 1 in the certain case.

- **Mat paddarray(Mat input, Size padding, std::string method, std::string direction);**

Paddarray function applies padding to a certain matrix. The function receives as input the input matrix, the padding size and the type of padding. The current function implements two types of padding: Zero padding and replicate padding equal to the pixel values of the image boundary pixels.

- **Size getPadSize(Mat f);**

This function returns the estimated required padding size for an image stored in matrix f

- **Mat r2g(Mat input);**

This function converts an image to grayscale if the image is at the RGB color space. If the image has only one channel the function returns the image as is.

- **Mat deconv(Mat &Blurred, Mat &PSF, double &w0alpha);**

The deconv() function deconvolves the blurred image with the selected kernel. If the image is intensity image, then the deconvolution is performed for only one channel. If the image is an RGB image the deconvolution occurs taking into consideration the multichannel image. In order to perform the deconvolution deconv_fn function is called.

- **Mat deconv_fn(Mat &Blurred, Mat &PSF, double &w0alpha);**

This function is the heart of the deconvolution procedure. It receives as input the blurry image and the PSF and perfoms deconvolution. The alpha parameter is a factor determining how intense the deconvolution will be. For a large alpha the final image may be destroyed. For a small alpha the image will be still blurry. During experiments we have determined that the optimal alpha ranged between 0.5 and 1.

5.2 GENERAL PURPOSE FUNCTIONS

The general purpose functions are more generic in the way they process the input data. These are presented in the following block. In this section a functional description of each functional provided

```
Mat equalizeIntensity(const Mat& inputImage);
Mat getChannel(Mat input, int theChannel);
Mat conjMat(Mat src);
void computeDFT(Mat& image, Mat& dest);
void computeIDFT(Mat& complex, Mat& dest);
void deconvolute(Mat& img, Mat& kernel);
Mat divideComplexMats(Mat InputA, Mat InputB);
void divSpectrums(InputArray _srcA, InputArray _srcB, OutputArray _dst, int flags, bool conjB);
void shift(const cv::Mat& src, cv::Mat& dst, cv::Point2f delta, int fill = cv::BORDER_CONSTANT, cv::Scalar value =
cv::Scalar(0, 0, 0, 0));
void equalizeHistogramIntensity(Mat &src, Mat&dst);
void bilateralFilter(InputArray src, OutputArray dst, double sigmaColor, double sigmaSpace);
void bilateralFilterImpl(Mat1d src, Mat1d dst, double sigmaColor, double sigmaSpace);
```

Figure 5-3 : C++ general purpose functions code

- **Mat equalizeIntensity(const Mat& inputImage);**

Equalize intensity function equalizes the histogram of an image with float or double format ranging from 0 to 1.

- **Mat getChannel(Mat input, int theChannel);**

This function returns a single channel matrix equal to the selected channel of a multichannel matrix.

- **Mat conjMat(Mat src);**

This function returns the conjugate of a complex opencv matrix.

- **void computeDFT(Mat& image, Mat& dest);**

This function computes the inverse DFT of an input matrix.

- **void computeIDFT(Mat& complex, Mat& dest);**

This function computes the DFT of an input matrix.

- **Mat divideComplexMats(Mat InputA, Mat InputB);**

This function performs a per-element division of single channel mats given that each matrix contains complex numbers. It uses OpenCV object Mat as inputs and output.

- **void divSpectrums(InputArray _srcA, InputArray _srcB, OutputArray _dst, int flags, bool conjB);**

This function performs a per-element division of single channel mats given that each matrix contains complex numbers. It uses OpenCV object inputArray as inputs and output.Additionally is allows the complex conjugate conversion of the second operand. This is done using the flag conjB.

- **void shift(const cv::Mat& src, cv::Mat& dst, cv::Point2f delta, int fill = cv::BORDER_CONSTANT, cv::Scalar value = cv::Scalar(0, 0, 0, 0));**

This function shifts cyclically the pixels of an image or matrix towards the selected direction.

- **void bilateralFilter(InputArray src, OutputArray dst, double sigmaColor, double sigmaSpace);**

This function applies a bilateral filter to the target image.

5.3 RESULTS

In this section we present the results for the C++ implementation. The execution took place in a Intel Core i7 3.2GHz Desktop PC with 8 GB of RAM. The test images were produced by sharp images after having blurred them with open source image processing software . We used the following sharp images

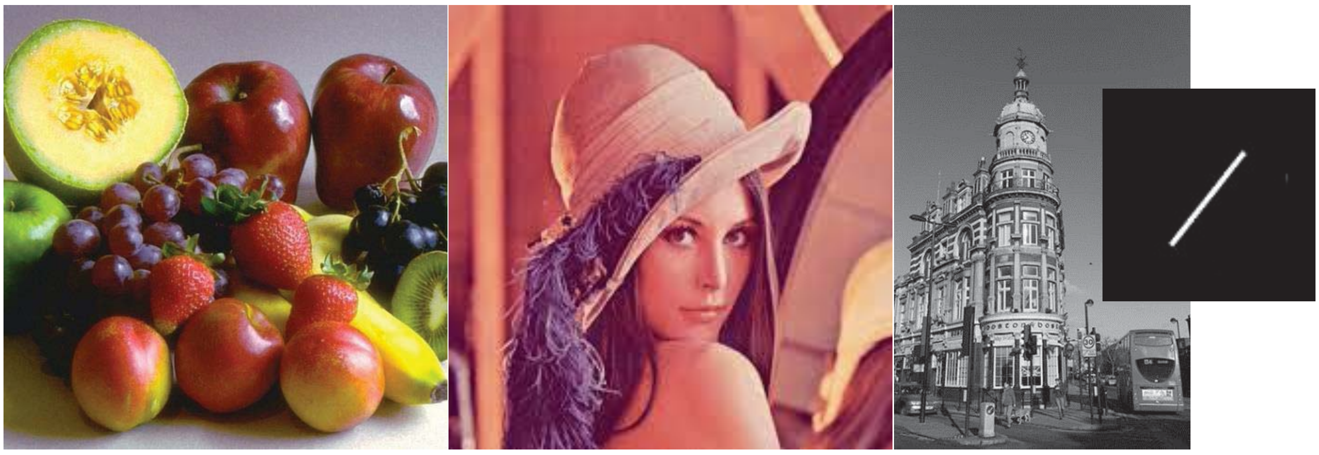


Figure 5-4 : Test set for C++ implementation

The artificial blur kernel had length 10 pixels and an angle of 58 degrees. The result was the following blurred images set

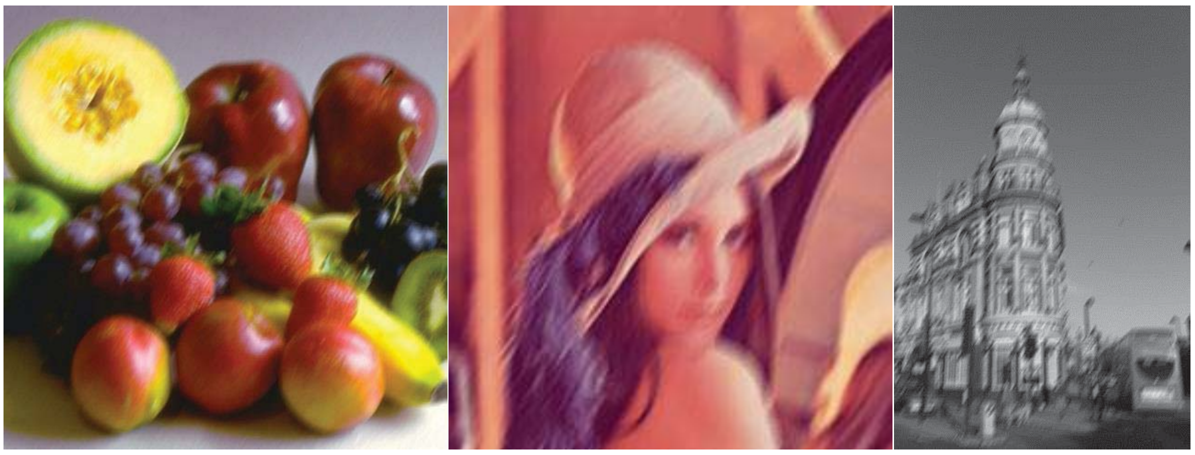


Figure 5-5 : Blurry images test set

Our implementation gave the results presented in Figure 5-6.In the right column we present the blurry test image and in the middle the recovered image result. Finally, in the right we present the estimated blur kernel.

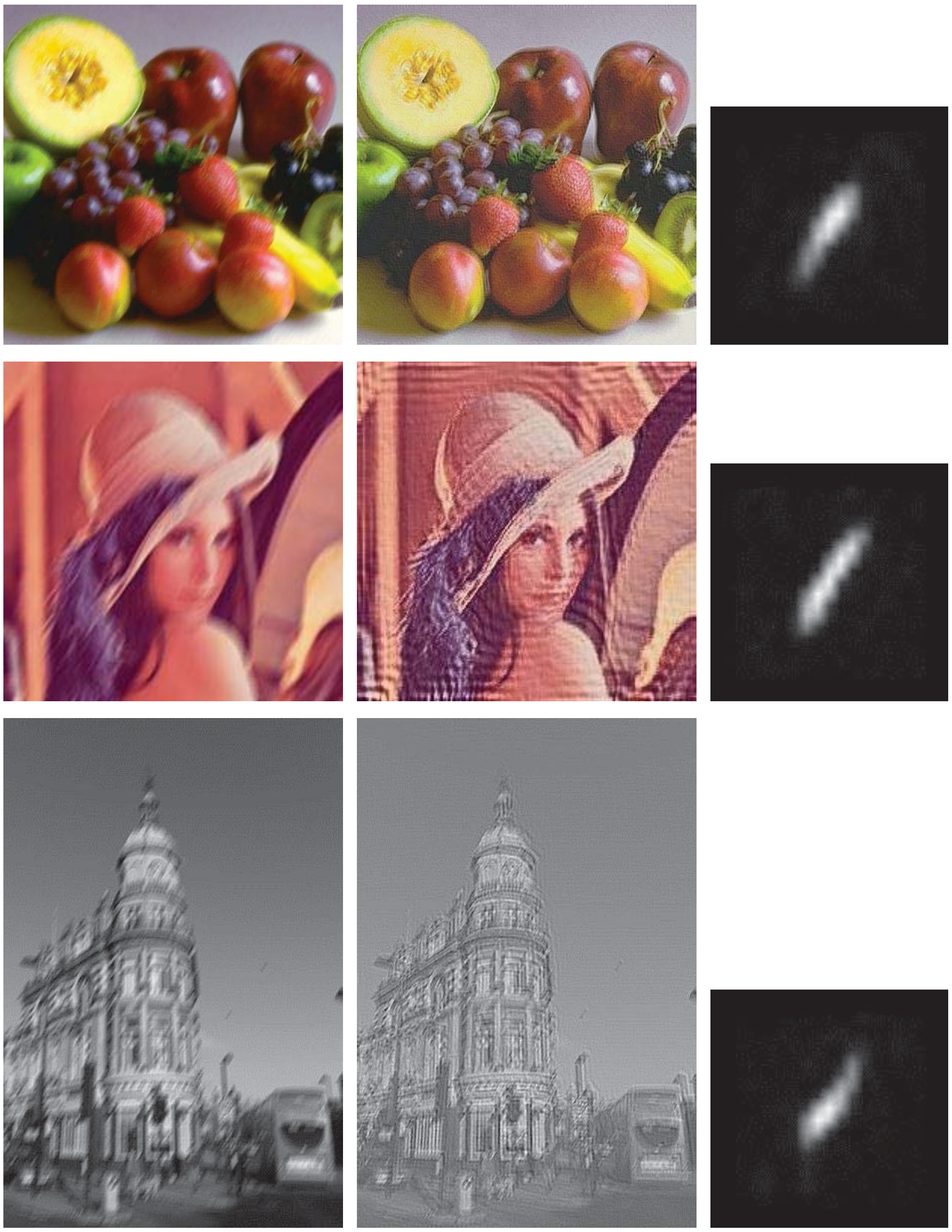


Figure 5-6 : Results of the deblurring process implementation in C++. In the left colunm we present the original blurry image, in the middle we present the latent sharp image and in the right we present the estimated blur kernel.

As it is made obvious the estimated kernel is very close to the artificial blur kernel used to blur the initially sharp images. The restored image is sharper but still noisy. The noisy result or the ringing artifacts are a common problem in the deblurring methods. The features of the original sharp image might be enhanced but this also happens for the noise. The following table includes time and execution results. The execution was single-threaded using a desktop PC Intel Core i7 3.2GHz with 8GB of RAM.

Table 5-1 : Results for C/C++ implementation

Image	Kernel Size	Image Size	Time
Fruit Basket	21x21	512x512	4.23
Lena	21x21	225x225	1.48
City	21x21	341x512	2.28

Chapter 6
DEVELOPMENT FOR THE ANDROID PLATFORM

Android is a mobile operating system (OS) currently developed by Google, based on the Linux kernel and designed primarily for touchscreen mobile devices such as smartphones and tablets. Android's user interface is mainly based on direct manipulation, using touch gestures that loosely correspond to real-world actions, such as swiping, tapping and pinching, to manipulate on-screen objects, along with a virtual keyboard for text input. In addition to touchscreen devices, Google has further developed Android TV for televisions, Android Auto for cars, and Android Wear for wrist watches, each with a specialized user interface. Variants of Android are also used on notebooks, game consoles, digital cameras, and other electronics. Initially developed by Android, Inc., which Google bought in 2005, Android was unveiled in 2007, along with the founding of the Open Handset Alliance – a consortium of hardware, software, and telecommunication companies devoted to advancing open standards for mobile devices. As of July 2013, the Google Play store has had over one million Android applications ("apps") published, and over 50 billion applications downloaded. An April–May 2013 survey of mobile application developers found that 71% of developers create applications for Android and a 2015 survey found that 40% of full-time professional developers see Android as their priority target platform, which is comparable to Apple's iOS on 37% with both platforms far above others. Android's source code is released by Google under open source licenses, although most Android devices ultimately ship with a combination of open source and proprietary software, including proprietary software required for accessing Google services. Android is popular with technology companies that require a ready-made, low-cost and customizable operating system for high-tech devices. Its open nature has encouraged a large community of developers and enthusiasts to use the open-source code as a foundation for community-driven projects, which add new features for advanced users or bring Android to devices originally shipped with other operating systems. At the same time, as Android has no centralized update system most Android devices fail to receive security updates: research in 2015 concluded that almost 90% of Android phones in use had known but unpatched security vulnerabilities due to lack of updates and support. The rest of this chapter is organized in the following manner. Subsection 6.1 presents features of the android platform based on a report of Dominique A. Heger on android internals[53] and subsection 6.2 presents the android implementation.

6.1 THE ANDROID PLATFORM

6.1.1 ANDROID ARCHITECTURE

Figure **6**-**2** outlines the current (layered) Android Architecture. The modified Linux kernel operates as the HAL, and provides device driver, memory management, process management, as well as networking functionalities, respectively. The library layer is interfaced through Java (which deviates from the traditional Linux design). It is in this layer that the Android specific libc (Bionic) is located. The surface manager handles the user interface (UI) windows. The Android runtime layer holds the Dalvik Virtual Machine (DVM) and the core libraries (such as Java or IO). Most of the functionalities available in Android are provided via the core libraries. The application framework houses the API interface. In this layer, the activity manager governs the application life cycle. The content providers enable applications to either access data from other applications or to share their own data. The resource manager provides access to non-code resources (such as graphics), while the notification manager enables applications to

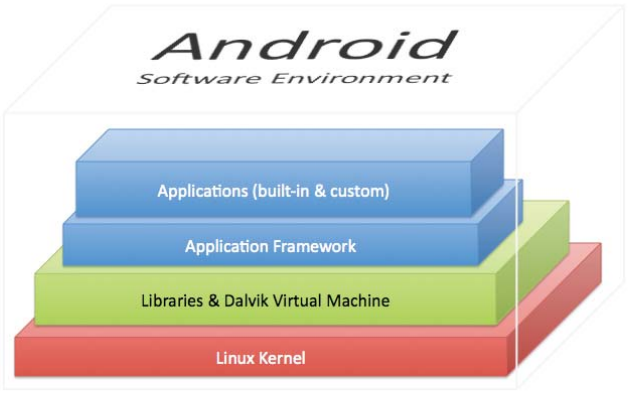

Android
Software Environment
Applications (built-in & custom)
Application Framework
Libraries & Dalvik Virtual Machine
Linux Kernel

Figure 6-1 : Android architecture

display custom alerts. On top of the application framework are the built-in, as well as the user applications, respectively. It has to be pointed out that a user application can replace a built-in application, and that each Android application runs in its own process space, within its own DVM instance. Most of these major Android components are further discussed (in more detail) in the next few sections of this report.

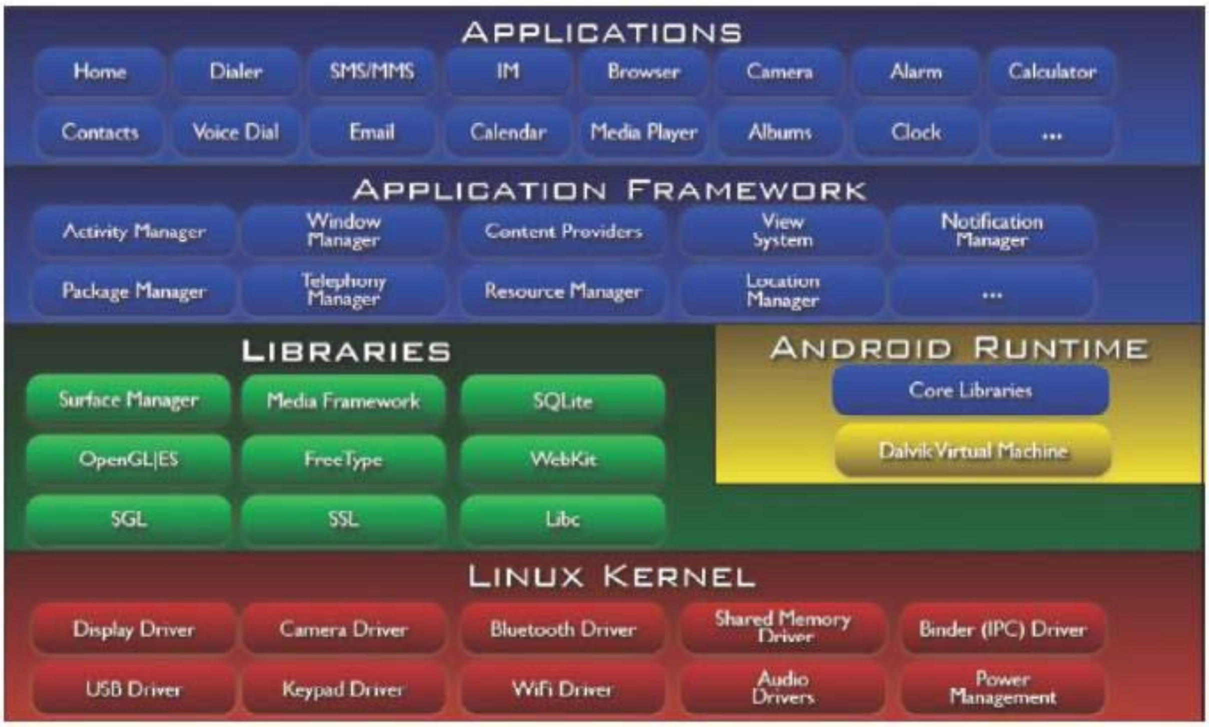


Figure 6-2 : Android architecture (courtesy of the OHA)

6.1.2 DALVIK VIRTUAL MACHINE

Android based systems utilize their own virtual machine (VM), which is known as the Dalvik Virtual Machine (DVM)[54]. The DVM uses special byte-code, hence native Java byte-code cannot directly be executed on Android systems. The Android community provides a tool (dx) that allows converting Java class files into Dalvik executables (dex). The DVM implementation is highly optimized in order to perform as efficiently and as effectively as possible on mobile devices that are normally equipped with a rather modest (these days normally a dual, or quad) CPU subsystem, limited memory resources, no OS swap space, and limited battery capacity. The DVM has been implemented in a way that allows a device to execute multiple VM's in a rather efficient manner. It also has to be pointed out that the DVM relies on the modified Linux kernel for any potential threading and low-level memory management functionalities. With Android 2.2, some major changes to the JVM infrastructure were implemented. Up to version 2.2, the JVM was an actual interpreter, similar to the original JVM solution deployed with Java 1.0. While the Android solution always reflected a very efficient interpreter, it was still an interpreter and hence, no native code was generated. With the release of Android 2.2, a just-in-time (JIT) compiler was incorporated into the solution stack, which translates the Dalvik byte-code into much more efficient machine code (similar to a C compiler). Currently, Android version 4 (Ice Cream Sandwich) and 4.1/4.2 (Jelly Bean) is deployed on some devices. It has to be pointed out though that currently, only a few devices are actually running version 4.1/4.2 or 4.0, while most devices are still operating on older Android versions. Down the road, additional JIT and garbage collection (GC) features will be deployed with Android, further busting aggregate systems performance.

6.1.3 TARGET PLATFORM – INSTRUCTION SET

To simplify the discussion, the statement made here is that most of the Linux 2.6 based devices are x86 based systems, whereas most mobile phones are ARM based products. While ARM represents a 32-bit reduced instruction set computer (RISC) instruction set architecture, x86 systems are primarily based on the complicated instruction set computer (CISC) architecture. In general, the statement can be made that ARM (RISC) is executing simpler (but more) instructions compared to an x86 (CISC) system. As already discussed, memory is at a premium in mobile devices due to size, cost, and power constraints. ARM addresses these issues by providing a 2nd 16-bit instruction set (labeled thumb) that can be interleaved with regular 32-bit ARM instructions. This additional instruction set can reduce the code size by up to 30% (at the expense of some performance limitations). Ergo, from an overall systems perspective, the incorporation of the thumb instruction set can be considered as an exercise in compromises. Compared to x86 processors, the ARM design reveals a strong focus on lower power consumption, which again makes it suitable for mobile devices [55]As with any other computer, the processor has a significant impact on aggregate systems performance. In the early days of Android, most devices featured the same ARM Qualcomm processor and hence, their performance behavior was pretty comparable. With the distribution of the Motorola Droid, the next generation of chipsets was introduced (supporting enhanced graphics processors). Today, there are basically 3 major chipsets being deployed in Android devices. To illustrate, HTC is utilizing the Qualcomm Snapdragon, Motorola uses the Texas Instruments OMAP, whereas Samsung designed their own Hummingbird chipset. It has to be pointed out though that all 3 processors discussed here are based on the ARM Cortex-A8 architecture (with vendor specific tweaks to offer unique features).

6.1.4 KERNEL AND STARTUP PROCESSES

It is paramount to reiterate that while Android is based on Linux 2.6, Android does not utilize a standard Linux kernel. Hence, an Android device should not be labeled a Linux operating system.
Some of the Android specific kernel enhancements include:

- alarm driver (provides timers to wakeup devices)
- shared memory driver (ashmem)
- binder (for inter-process communication),
- power management (which takes a more aggressive approach than the Linux PM solution)
- low memory killer
- kernel debugger and logger

During the Android boot process, the Android Linux kernel component first calls the init process (compared to standard Linux, nothing unusual there). The init process accesses the files init.rc and init.device.rc (init.device.rc is device specific). Out of the init.rc file, a process labeled zygote is started. The zygote process loads the core Java classes, and performs the initial processing steps. These Java classes can be reused by Android applications and hence, this step expedites the overall startup process. After the initial load process, zygote idles on a socket and waits for further requests. Every Android application runs in its own process environment. A special driver labeled the binder allows for (efficient) inter-process communications (IPC). Actual objects are stored in shared memory. By utilizing shared memory, IPC is being optimized, as less data has to be transferred. Compared to most Linux or UNIX environments, Android does not provide any swap space. Hence, the amount of virtual memory is governed by the amount of physical memory available on the device [56]

6.1.5 THE BIONIC LIBRARY

Compared to Linux, Androids incorporates its own c library (Bionic)[57]. The Bionic library is not compatible with the Linux glibc. Compared to glibc, the Bionic library has a smaller memory footprint. To illustrate, the Bionic library contains a special thread implementation that 1st, optimizes the memory consumption of each thread and 2nd, reduces the startup time of a new thread. Android provides run-time access to kernel primitives[58] . Hence, user-space components can dynamically alter the kernel behavior. Only processes/threads though that do have the appropriate permissions are allowed to modify these settings. Security is maintained by assigning a unique user ID (UID) and group ID (GID) pair to each application. As mobile devices are normally intended to be used by a single user only (compared to most Linux systems), the UNIX/Linux /etc/passwd and /etc/group settings have been removed. In addition, (to boost security), /etc/services was replaced by a list of services (maintained inside the executable itself). To summarize, the Android c library is especially suited to operate under the limited CPU and memory conditions common to the target Android platforms[58]. Further, special security provisions were designed and implemented to ensure the integrity of the system.

6.1.6 STORAGE MEDIA & FILE SYSTEM

When it comes to configuring and setting-up mobile devices, traditional hard drives are in general too big (size), too fragile, and consume too much power to be useful. In contrast, flash memory devices normally provide a (relative) fast read access behavior as well as better (kinetic) shock resistance compared to hard drives. Fundamentally, two different types of flash memory devices are common, labeled as NAND and NOR based solutions. While in general, NOR based solutions provide low density, they are characterized as (relative) slow write and fast read components. On the other hand, NAND based solutions offer low cost, high density, and are labeled as (relative) fast write and slow read IO solutions. Some embedded systems are utilizing NAND flash devices for data storage, and NOR based components for the code (the execution environment). From a file system perspective, as of Android version 2.3, the (well-known) Linux ext4 file system is being used[59] . Prior to the ext4 file system, Android normally used YAFFS (yet another flash file system). The YAFFS solution is known as the first NAND optimized Linux flash file system. Some Android product providers (such as Archos with ext3 in Android 2.2) replaced the standard Archos file system with another file system solution of their choice. As of the writing of this report, the maximum size of any Android application equals to a low 2-digit MB number, which compared to actual Linux based systems has to be considered as being very small. This implies that the memory and file system requirements (from a size perspective – not from a data integrity perspective) are vastly different for Android based devices compared to most Linux systems.

6.1.7 POWER MANAGEMENT

In the mobile device arena, power management is obviously paramount. That does not imply though that power management should be neglected on any other system. Hence, power management in any IT system, with any operating system, is considered a necessity due to the ever increasing power demand of today's computer systems. To illustrate, to reduce and manage power consumption, Linux based systems provide power-saving features such as clock gating, voltage scaling, activating sleep modes, or disabling memory cache. Each of these features reduces the system's power consumption (normally at the expense of an increased latency behavior)[59]. Most Linux based systems manage power consumption via the Advanced Configuration and Power Interface (ACPI). Android based systems provide their own power management infrastructure (labeled PowerManager) that was designed based on the premise that a processor should not consume any power if no applications or services actually require power. Android demands that applications and services request CPU resources via wake locks through the Android application framework and native Linux libraries. If there are no active wake locks, Android will shutdown the processor.

6.1.8 ANDROID APPLICATIONS

Android applications are bundled into an Android package (.apk) via the Android Asset Packaging Tool (AAPT). To streamline the development process, Google provides the Android Development Tools (ADT). The ADT streamlines the conversion from class to dex files, and creates the .apk during deployment. In a very simplified manner, Android applications are in general composed of:

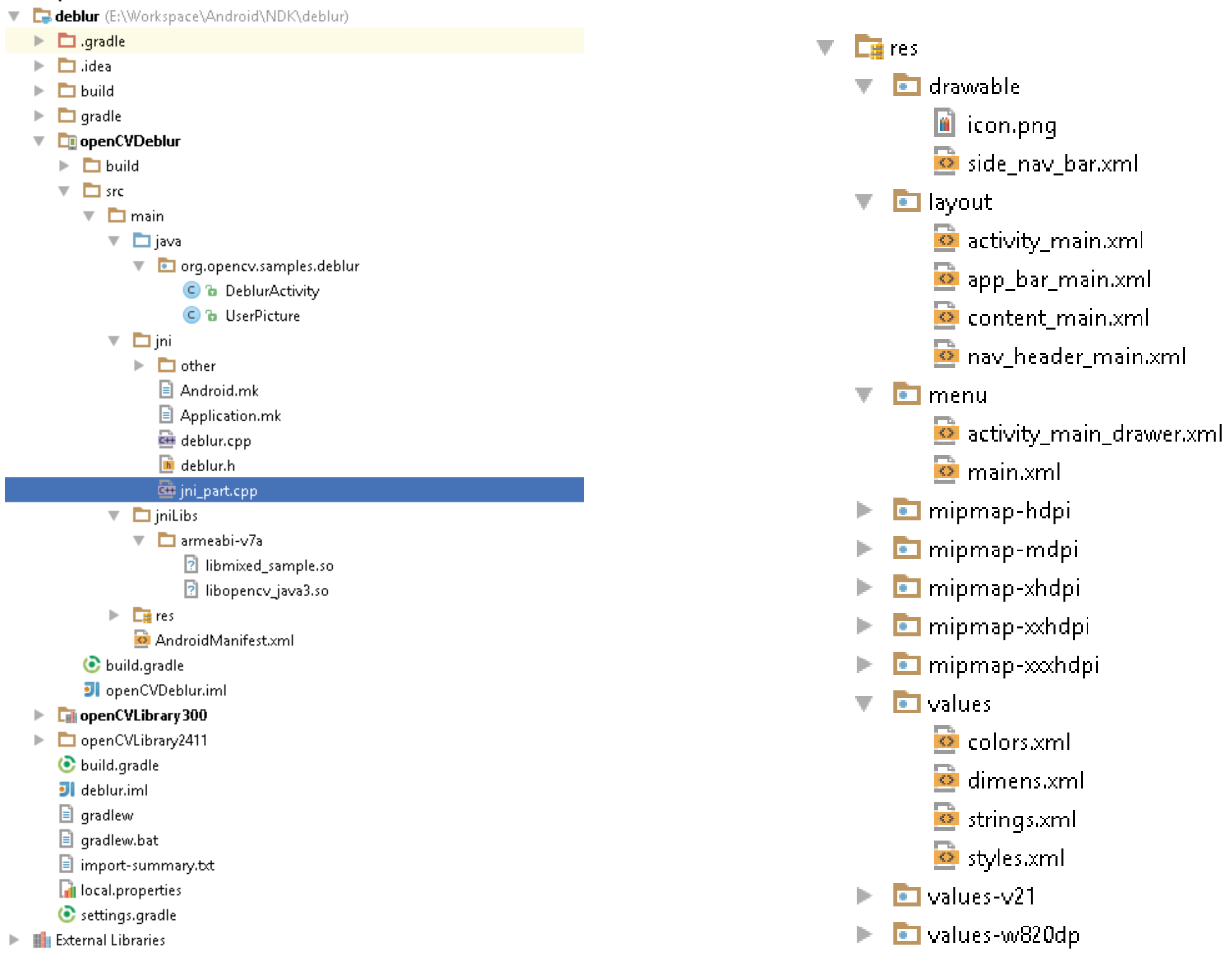
- Activities (needed to create a screen for a user application – classes with a UI).
- Intents (used to transfer control from one activity to another).
- Services (classes without a UI, so they can be executed in the background).
- Content Providers (allows the application to share information with other applications).

6.2 ANDROID IMPLEMENTATION

In this section we present the implementation of the algorithm presented in Chapter 4 for the android platform. Additionally, we provide information about the structure and specifications of the android platform.

6.2.1 PROJECT STRUCTURE

The android project is comprised of two modules. The openCVDeblur and the openCVLibrary300 modules. The openCVLibrary module is downloaded from the official OpenCV website as prebuilt and then was imported into the Android project. The openCVDeblur project contains the Java source code for the user interface and the native C++ code that performs the deblurring process. Specifically, as shown in Fig 6-3 the java folder contains the DeblurActivity.java and the UserPicture.java. For the C/C++ , the code is contained in the jni folder. The jni_part.cpp file contains the functions that provide the interface with the java code and the deblur.cpp and deblur.h files contain the source code as it was explained in Chapter 5.



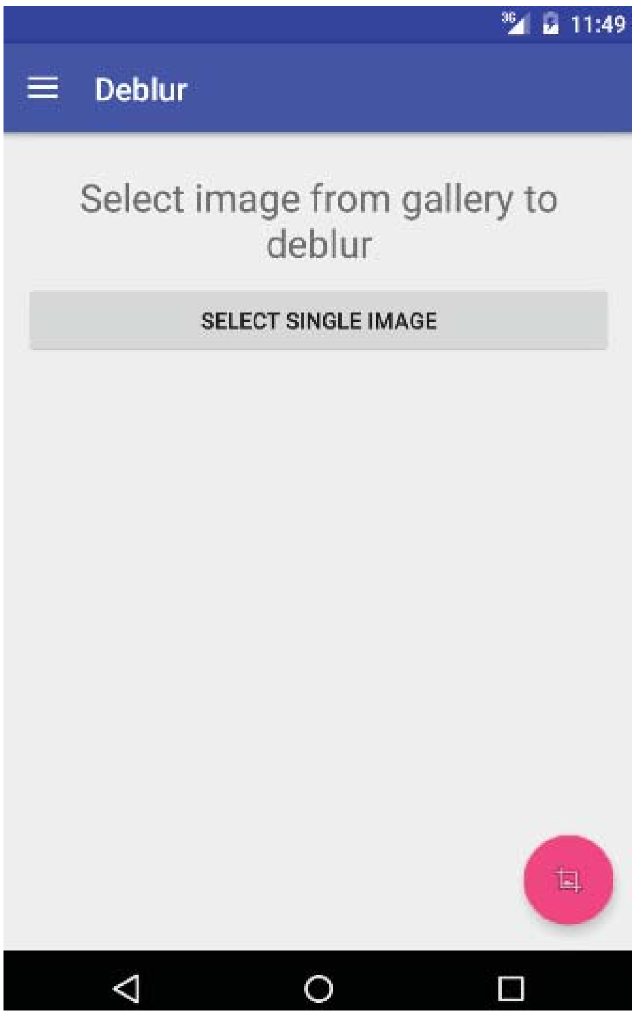
```
▼ deblur (E:\Workspace\Android\NDK\deblur)
    ▶ .gradle
    ▶ .idea
    ▶ build
    ▶ gradle
    ▼ openCVDeblur
        ▶ build
        ▼ src
            ▼ main
                ▼ java
                    ▼ org.opencv.samples.deblur
                            DeblurActivity
                            UserPicture
                ▼ jni
                    ▶ other
                        Android.mk
                        Application.mk
                        deblur.cpp
                        deblur.h
                        jni_part.cpp
                ▼ jniLibs
                    ▼ armeabi-v7a
                            libmixed_sample.so
                            libopencv_java3.so
                ▶ res
                    AndroidManifest.xml
            build.gradle
            openCVDeblur.iml
    ▶ openCVLibrary300
    ▶ openCVLibrary2411
        build.gradle
        deblur.iml
        gradlew
        gradlew.bat
        import-summary.txt
        local.properties
        settings.gradle
▶ External Libraries
```

```
▼ res
    ▼ drawable
            icon.png
            side_nav_bar.xml
    ▼ layout
            activity_main.xml
            app_bar_main.xml
            content_main.xml
            nav_header_main.xml
    ▼ menu
            activity_main_drawer.xml
            main.xml
    ▶ mipmap-hdpi
    ▶ mipmap-mdpi
    ▶ mipmap-xhdpi
    ▶ mipmap-xxhdpi
    ▶ mipmap-xxxhdpi
    ▼ values
            colors.xml
            dimens.xml
            strings.xml
            styles.xml
    ▶ values-v21
    ▶ values-w820dp
```

Figure 6-3 : Android project structure(on the left) and the resources folder(on the right)

6.2.2 USER INTERFACE

The user interface is built the xml files inside the layout folder and the menu folder and each part of the interface is linked to a functionality at the DeblurActivity.java file or any other source code java file. By pressing the select single image button the user can choose an image for the gallery for processing. Then the user can press the pink button in order to commence the deblurring process. A new dialog box appears suggesting the user to wait. After the procedure is complete the deblurred image appears.



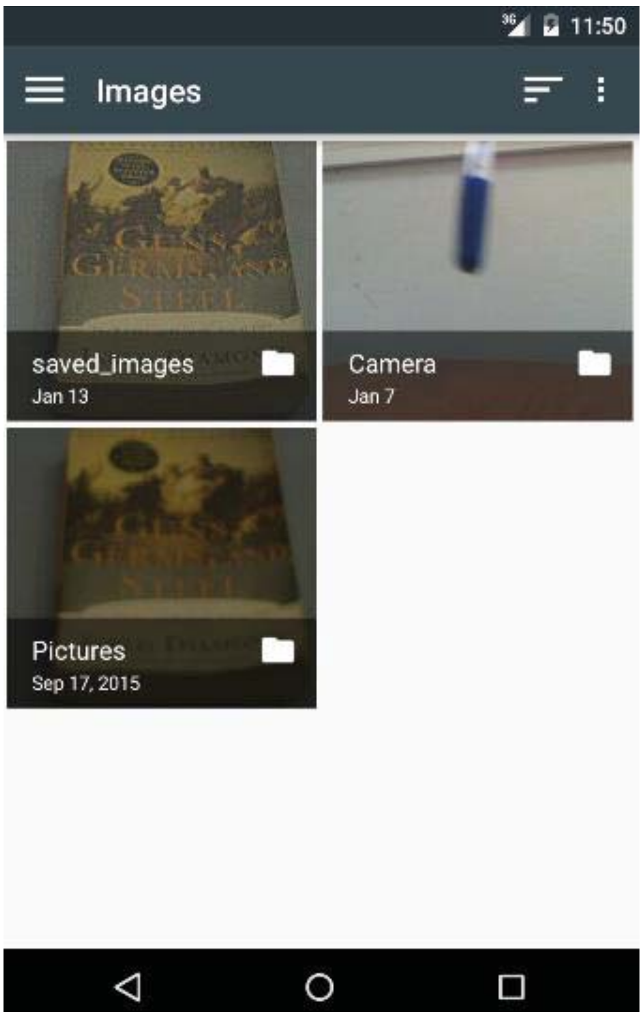
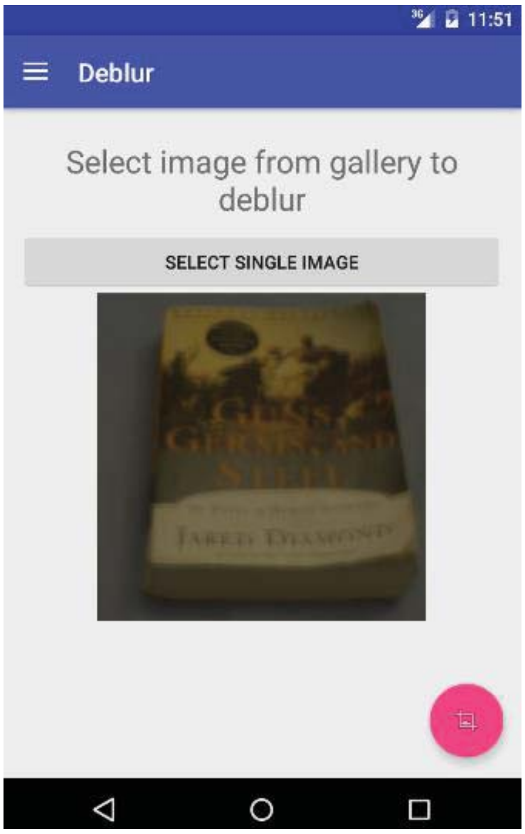


Figure 6-4:The user interface (on the left) and the gallery (on the right)



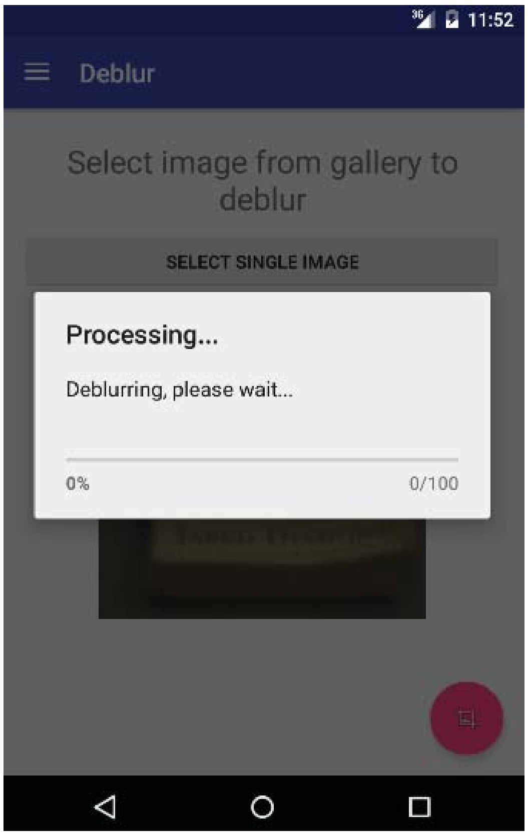

11:52
Deblur
Select image from gallery to deblur
SELECT SINGLE IMAGE
Processing...
Deblurring, please wait...
0%
0/100

Figure 6-5: The chosen image appears on the interface. The user can press the pink button in order to commence the deblurring procedure. The dialog box on the left informs the user that processing is under way suggesting patience

6.2.3 JAVA-NATIVE INTERFACE (JNI)

The JNI acts as a bridge between the java code and the code written in C++. The file jni_part.cpp contains the function that is being recognized by the main activity file of the project (DeblurActivity.java)

```
#include "deblur.h"
using namespace std;
using namespace cv;
Mat deblur(Mat image);
extern "C" {
JNIEXPORT void JNICALL Java_org_opencv_samples_deblur_DeblurActivity_Deblur(JNIEnv*, jobject, jlong addrIn, jlong
addrOut);
JNIEXPORT void JNICALL Java_org_opencv_samples_deblur_DeblurActivity_Deblur(JNIEnv*, jobject,jlong addrIn, jlong
addrOut)
{
        Mat& mInput    = *(Mat*)addrIn;
        Mat& mOutput = *(Mat*)addrOut;
        mOutput=deblur(mInput);
}
}
```

Figure 6-6 : C++ code for exporting native function to java.

6.2.4 NATIVE C++

The C/C++ implementation is present in jni folder in deblur.cpp and deblur.h files. The code is the same as described in Chapter 5. The jni_part.cpp calls the mOutput=deblur(mInput) function which initializes the parameters for deblurring and calls the blinddeblurmap function.

```
Mat    deblur(Mat image)
{
        Mat _input = image;
        Mat _out;

        int _initsize = (int)(std::min(_input.size().height, _input.size().width) / 10);
        _initsize = 21;
        int _iterScaleNum = 20;
        double _alpha = 2;
        double _freqCut = 16;
        int _belta = 2;
        double _sigmaSpace = 0;
        double _sigmaColor = 0;
        int _shockFilterIter = 5;
        double _shockFilterdt = 0.2;
        kernelmulkernelfactor = 0.5;
        maxkernelout = true;
        kernelmulkernel = true;
        equalize = false;

        deconvmethod = "RGB";
        deconvmethod = "YCbCr";

        _out = blinddeblurmap(_input, _initsize, _iterScaleNum, _alpha, _freqCut, _belta, _sigmaSpace, _sigmaColor,
_shockFilterIter, _shockFilterdt);

        _out = 255 * _out;
        _out.convertTo(_out, CV_8UC3);
        if (equalize){ _out = equalizeIntensity(_out); }
        return    _out;
}
```

Figure 6-7 : C++ code for d**eblur function with initialization**

```
Mat blinddeblurmap(Mat _inputImage, int _initsize = 21, int _iterScaleNum = 10, double _alpha = 5.0, double _freqCut =
16.0, int _belta = 2, double _sigmaSpace = 0, double _sigmaColor = 0, int _shockFilterIter = 3, double _shockFilterdt =
0.5)
{
	clock_t begin = clock(); deBlur * theProcess;

	Mat result;

	theProcess = new deBlur();
	theProcess->initsize = _initsize;
	theProcess->iterScaleNum = _iterScaleNum;
	theProcess->alpha = _alpha;
	theProcess->freqCut = _freqCut;
	theProcess->belta = _belta;
	theProcess->sigmaSpace = _sigmaSpace;
	theProcess->sigmaColor = _sigmaColor;
	theProcess->shockFilterIter = _shockFilterIter;
	theProcess->shockFilterdt = _shockFilterdt;
	theProcess->img = _inputImage;

	theProcess->readImage();
	theProcess->analyzeKernel();
	theProcess->iterativeDeblurring();
	clock_t end = clock();
	double elapsed_secs = double(end - begin) / CLOCKS_PER_SEC;
	std::cout << elapsed_secs << std::endl;
	result = theProcess->deblurredImage;
	delete theProcess;
	return result;
}
```

Figure 6-8 : C++ code for main blind deblur function

In this chapter we presented details related to the development of the android application. The application is written using ANDROID studio. The natively implemented deblurring functionality is called through the java-native-interface functionality. The next chapter focuses on evaluation and benchmarking and presents the deblurring results as they were produced by the ANDROID application.

Chapter 7
EVALUATION AND BENCHMARKING

This Chapter presents the results of this thesis. In order to provide a set of results we require a certain test set. Thus, we use eight images six of them blurred with known blur kernel and two of them are photos that characterized by an unknown point spread function.

7.1 TEST SET

The test set is presented in the figure below. Six sharp image were chosen and then they were blurred with the same blur kernel as shown in Figure 7-1 and in Figure 7-3. Additionally two more figures were chosen with unknown blur kernels as shown in Figure 7-2. The test set is used to evaluate the algorithm showing how successful the kernel estimation process in in each case.

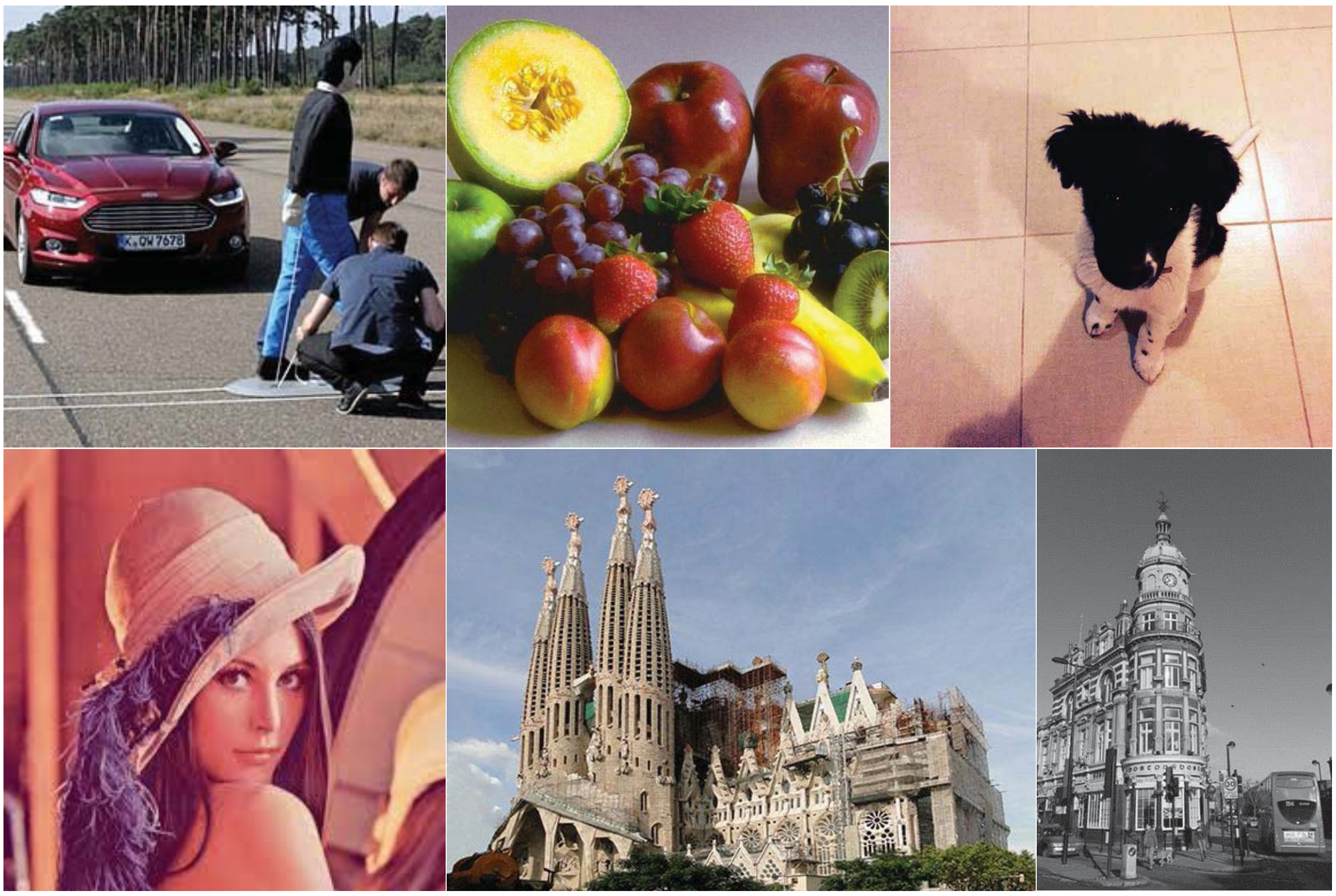


Figure 7-1 : Sharp images test set.

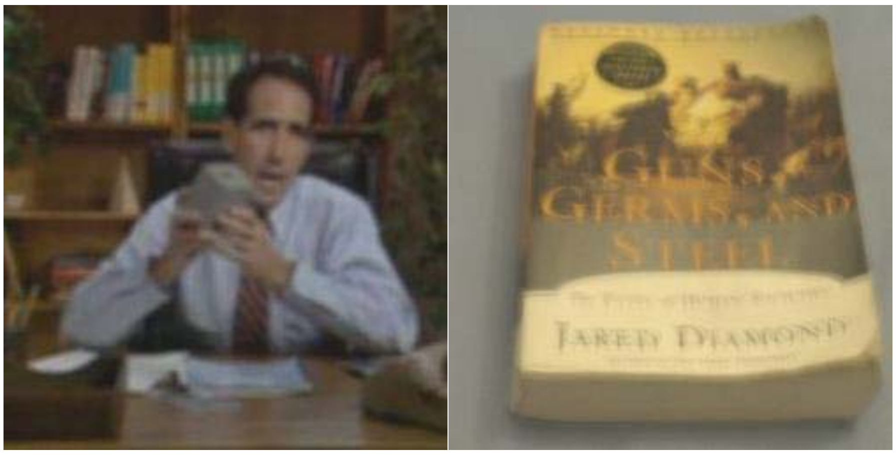


Figure 7-2 : Blurry images with unknown kernel test set. These are blurry photos and the point-spread function corresponding to the image blur is not known. They will be used to evaluate the algorithm.

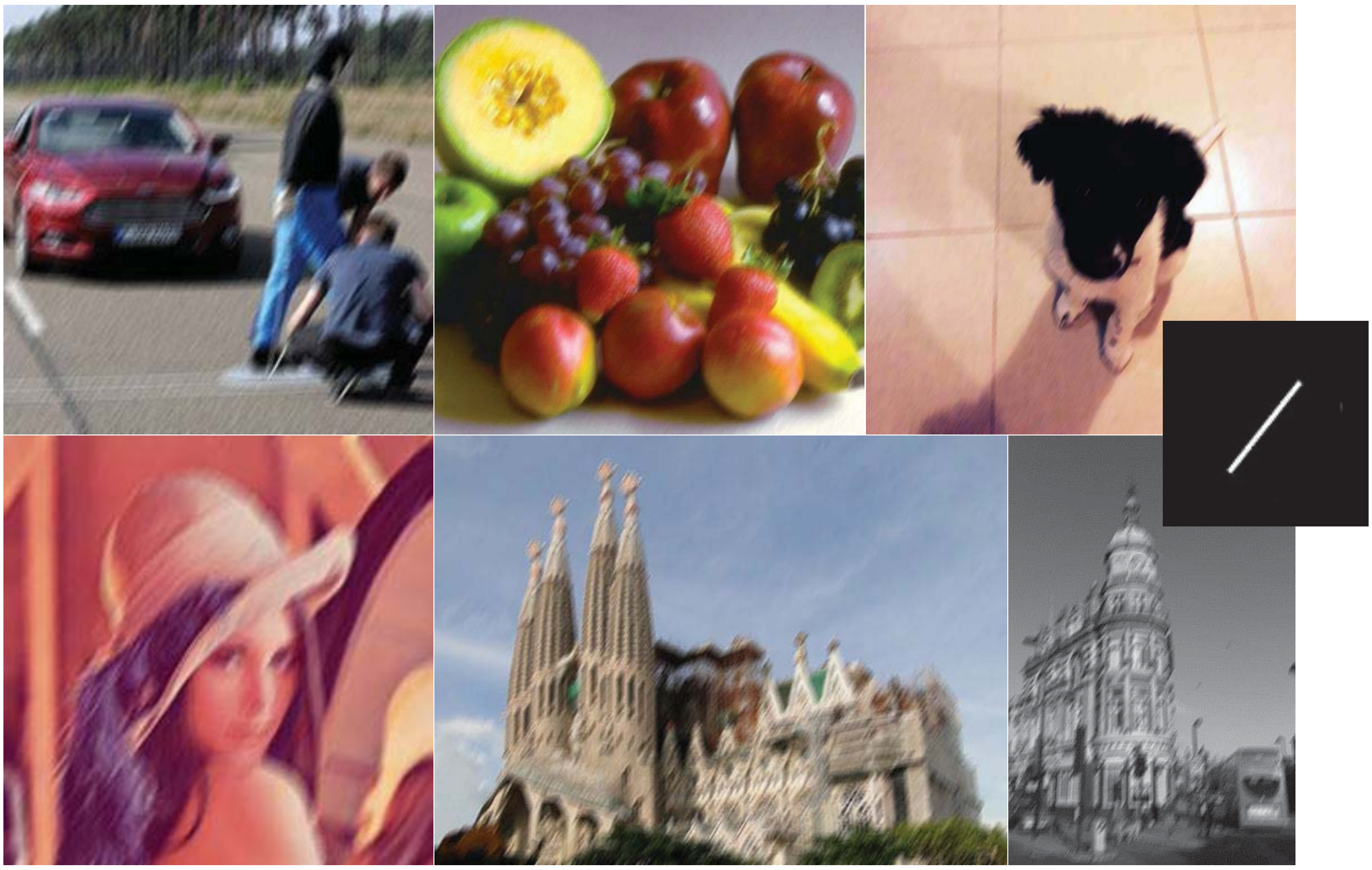


Figure 7-3 : Blurred images test set with known blur kernel having length equal to 10 pixels and angle equal to 58 degrees

7.2 EVALUATION

The results of the deblurring process appear in the following figures. Each figure contains the blurry image on the left, the recovered sharp image and the kernel on the right.

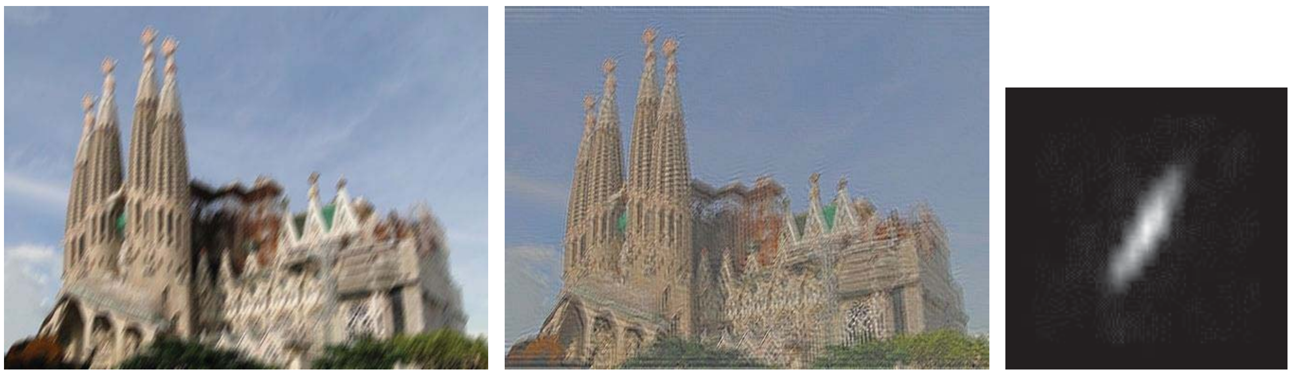


Figure 7-4 : Sagrada familia test image. The left image shows the original blurry image, blurred with the aforementioned artificial blur kernel with ten pixels length and 58 degrees angle. The middle image presents the recovered image and the right image presents the estimated blur kernel.

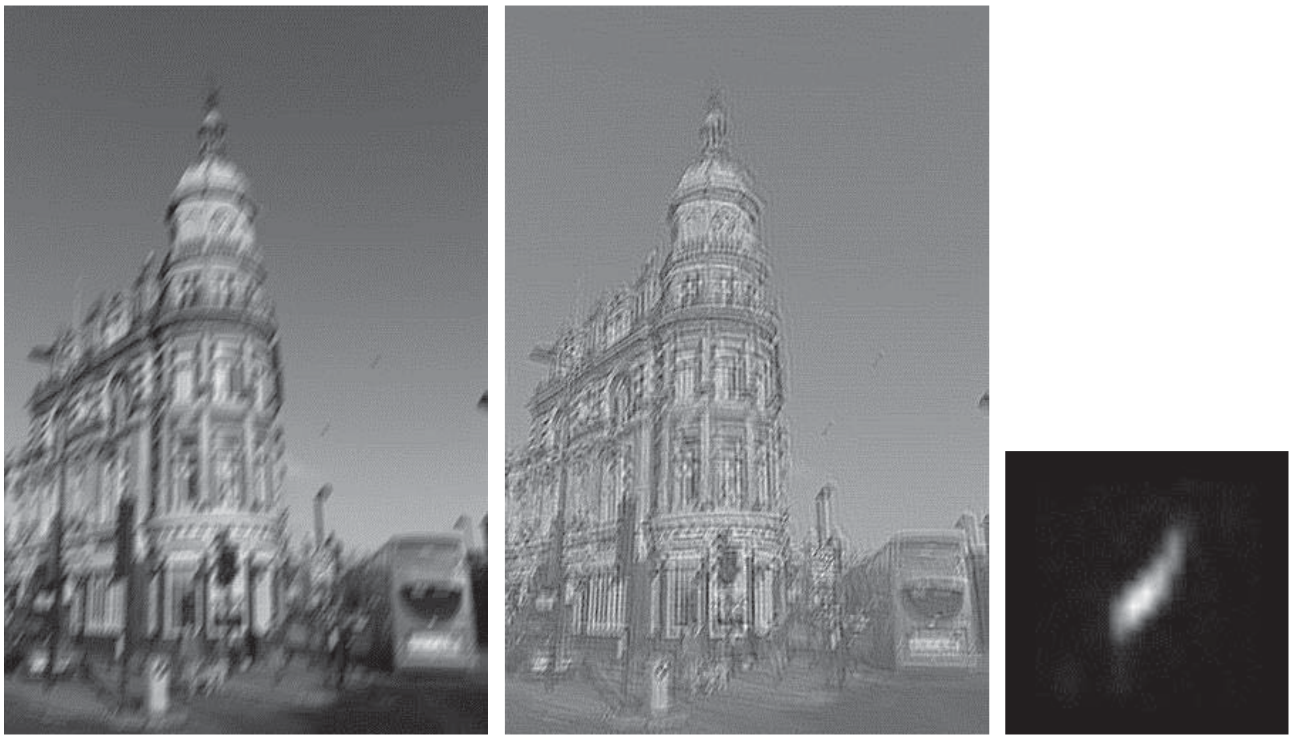


Figure 7-5 : City photo test image. The image left shows the original blurry image, blurred with the aforementioned artificial blur kernel with ten pixels length and 58 degrees angle. The middle image presents the recovered image and the right image presents the estimated blur kernel.

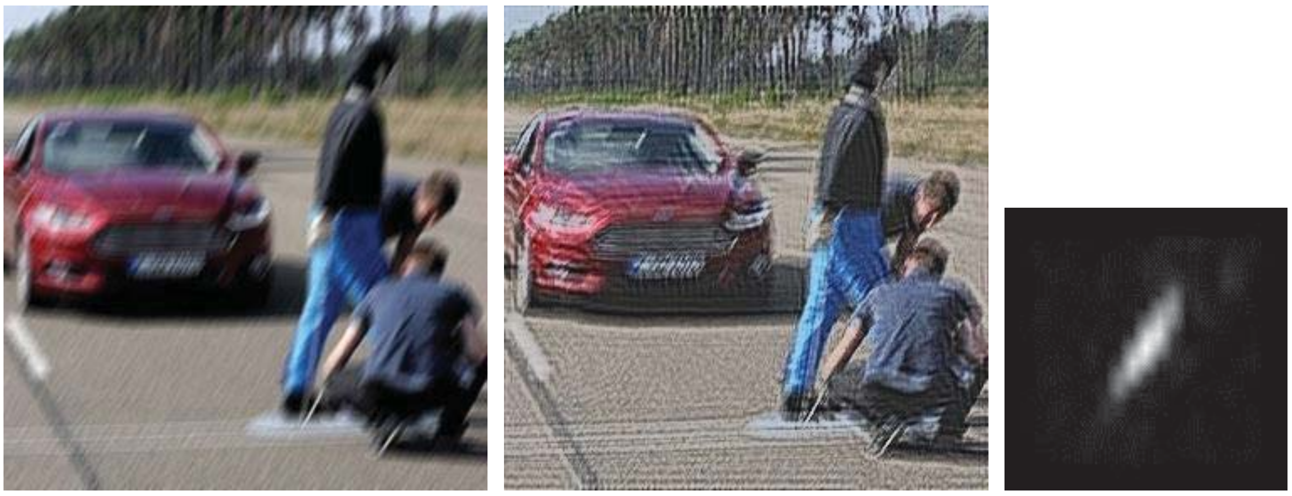


Figure 7-6 : Car scene test image. The left image shows the original blurry image, blurred with the aforementioned artificial blur kernel with ten pixels length and 58 degrees angle. The middle image presents the recovered image and the right image presents the estimated blur kernel.

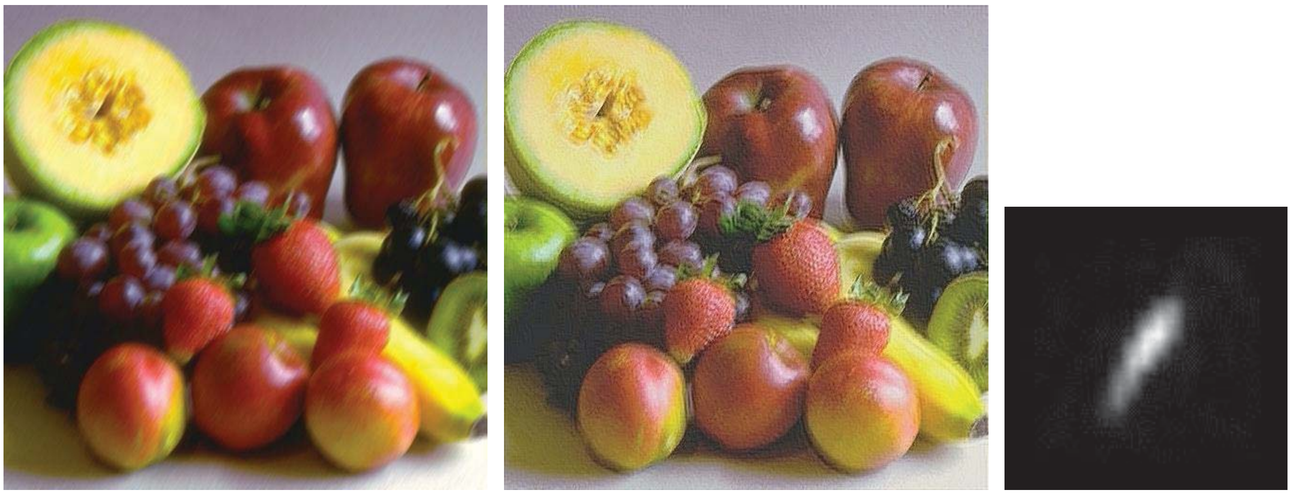


Figure 7-7 : Fruit basket test image. The left image shows the original blurry image. Blurred with the aforementioned artificial blur kernel with ten pixels length and 58 degrees angle. The middle image presents the recovered image and the right image presents the estimated blur kernel.

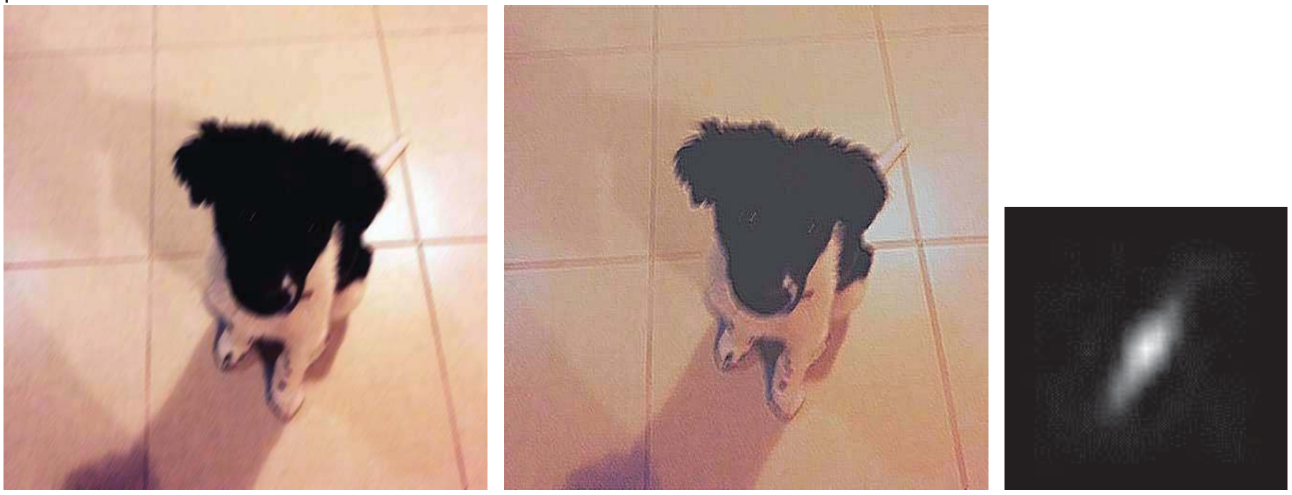


Figure 7-8 : Dog test image. The left image shows the original blurry image, blurred with the aforementioned artificial blur kernel with ten pixels length and 58 degrees angle. The middle image presents the recovered image and the right image presents the estimated blur kernel.

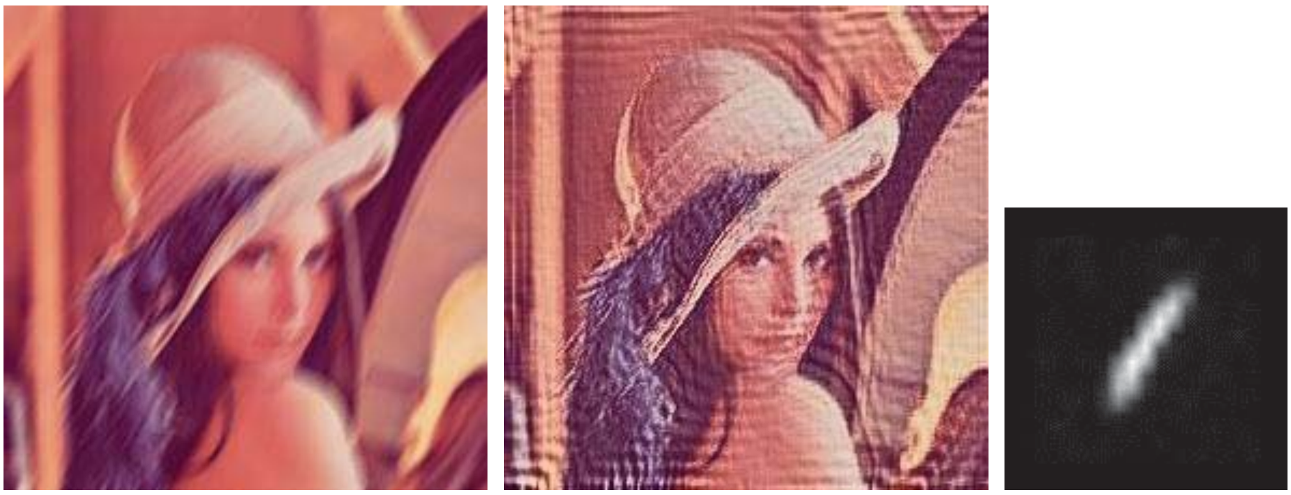


Figure 7-9: Lena test image. The left shows the original blurry image, blurred with the aforementioned artificial blur kernel with ten pixels length and 58 degrees angle. The middle image presents the recovered image and the right image presents the estimated blur kernel.

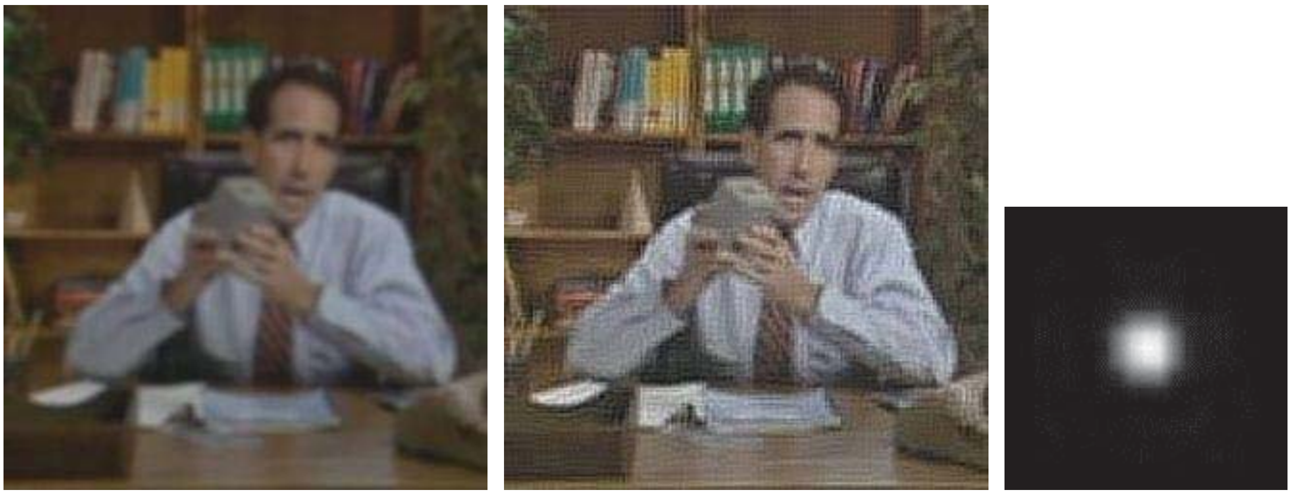


Figure 7-10: Salesman test image with unknown kernel. The left image shows the original blurry image. The middle image presents the recovered image and the right image presents the estimated blur kernel.

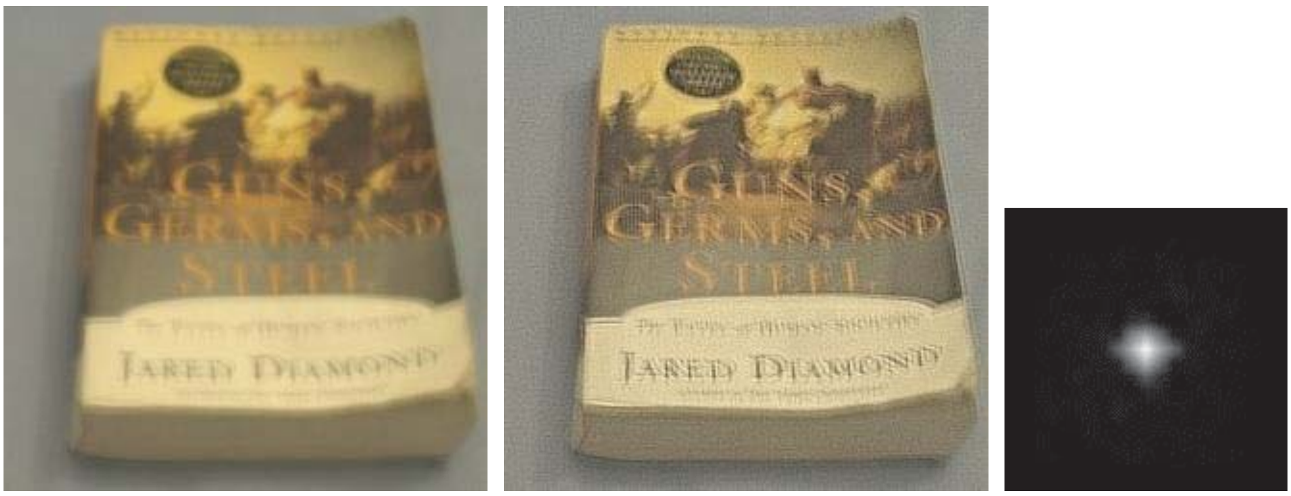


Figure 7-11 : Book test image with known kernel. The left image shows the original blurry image. The middle image presents the recovered image and the right image presents the estimated blur kernel.

7.3 ANDROID APPLICATION TEST

This section shows the results of the deblurring process as they appear in the android application. The following figures include the result for each of the test images. On the left column we present the original blurry image as it was selected by the user from the phone or tablet gallery. After having chosen the target image the user presses the pink button to start the deblurring process. The result presented in the right column.

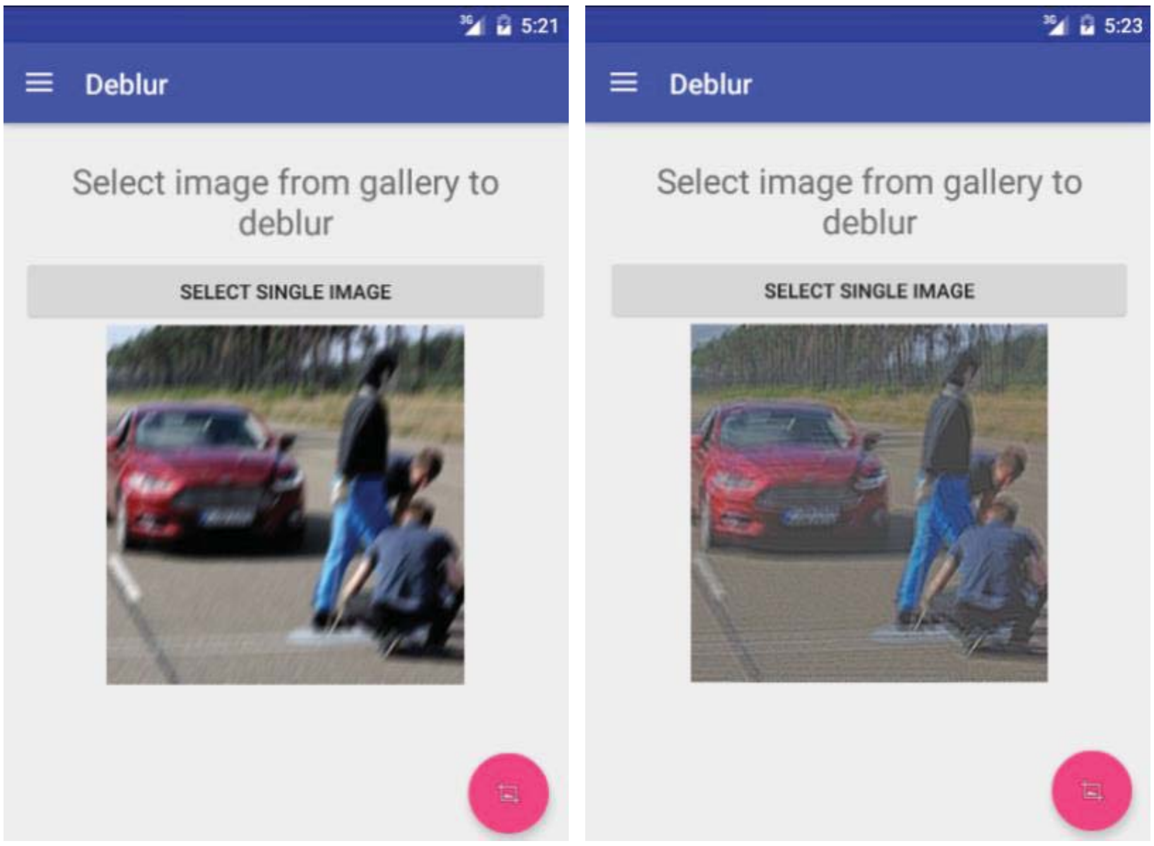

5:21
Deblur
Select image from gallery to deblur
SELECT SINGLE IMAGE
5:23
Deblur
Select image from gallery to deblur
SELECT SINGLE IMAGE

Figure 7-12 : Car test image example. The resulting image was generated by the android application after the user chose it from the android device library and pressed the pink execution button.

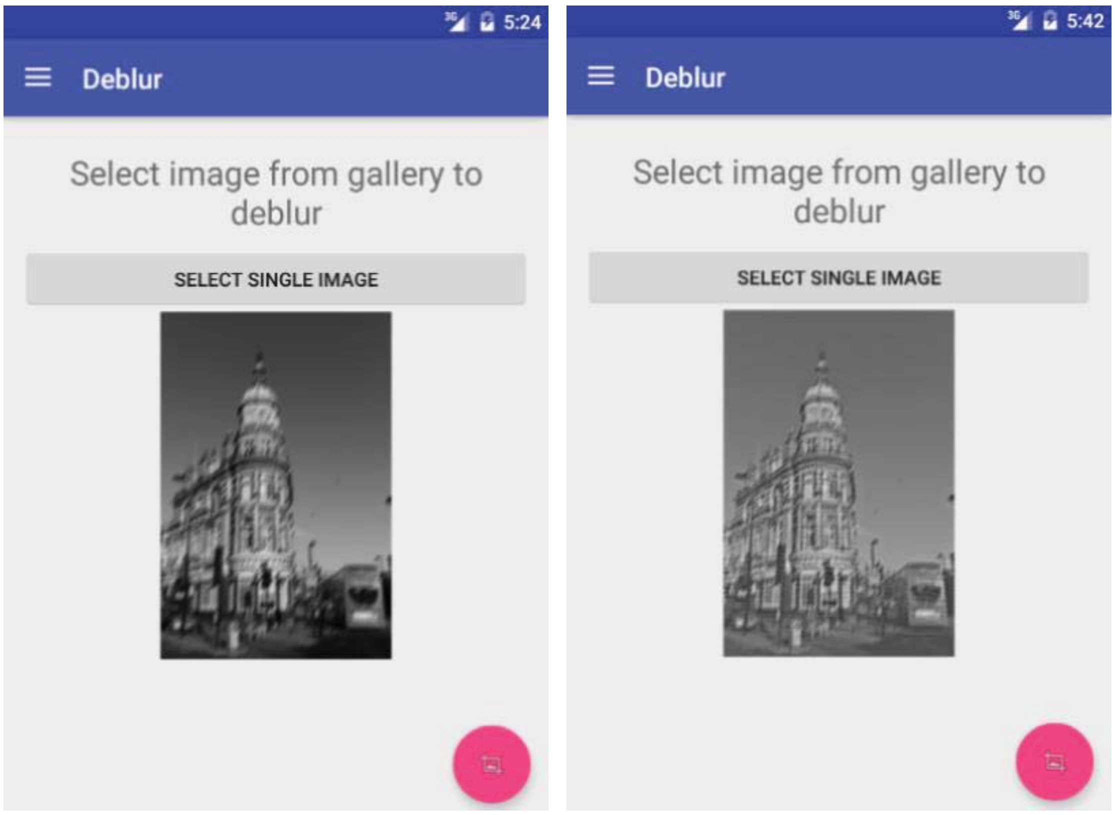

5:24
Deblur
Select image from gallery to deblur
SELECT SINGLE IMAGE
5:42
Deblur
Select image from gallery to deblur
SELECT SINGLE IMAGE

Figure 7-13 : City test image example. The resulting image was generated by the android application after the user chose it from the android device library and pressed the pink execution button.

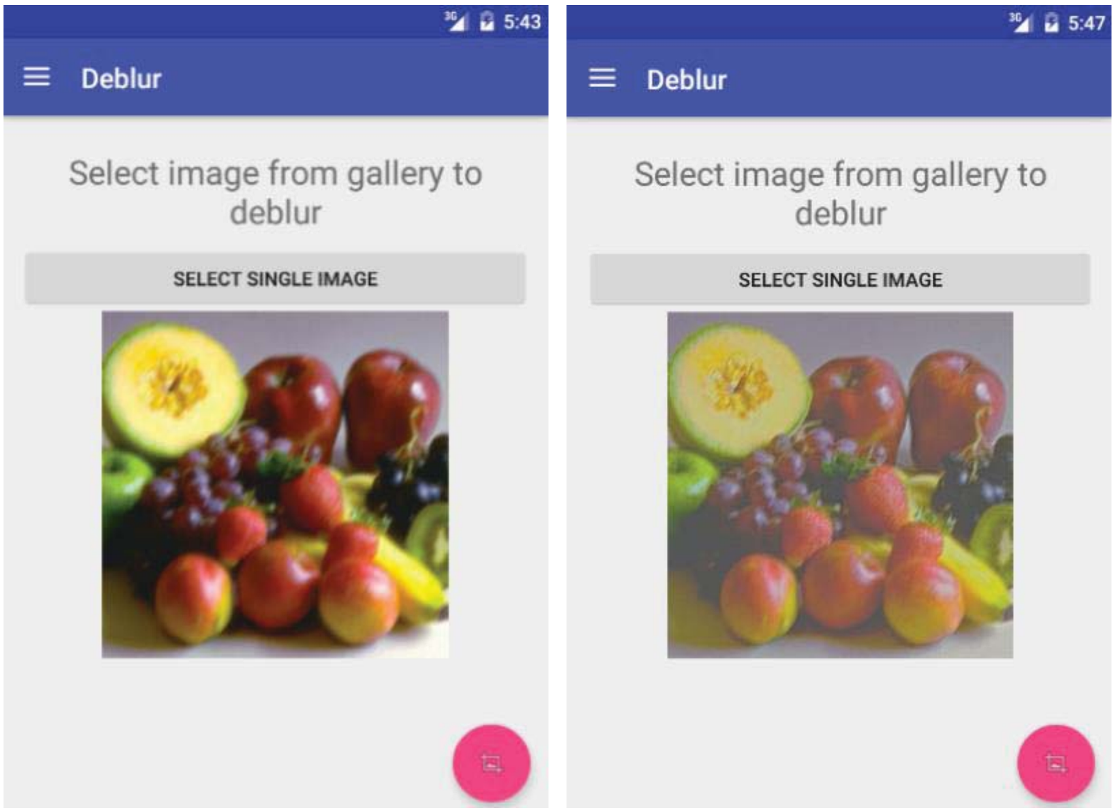

5:43
Deblur
Select image from gallery to deblur
SELECT SINGLE IMAGE
5:47
Deblur
Select image from gallery to deblur
SELECT SINGLE IMAGE

Figure 7-14 : Fruitbasket test image example. The resulting image was generated by the android application after the user chose it from the android device library and pressed the pink execution button.

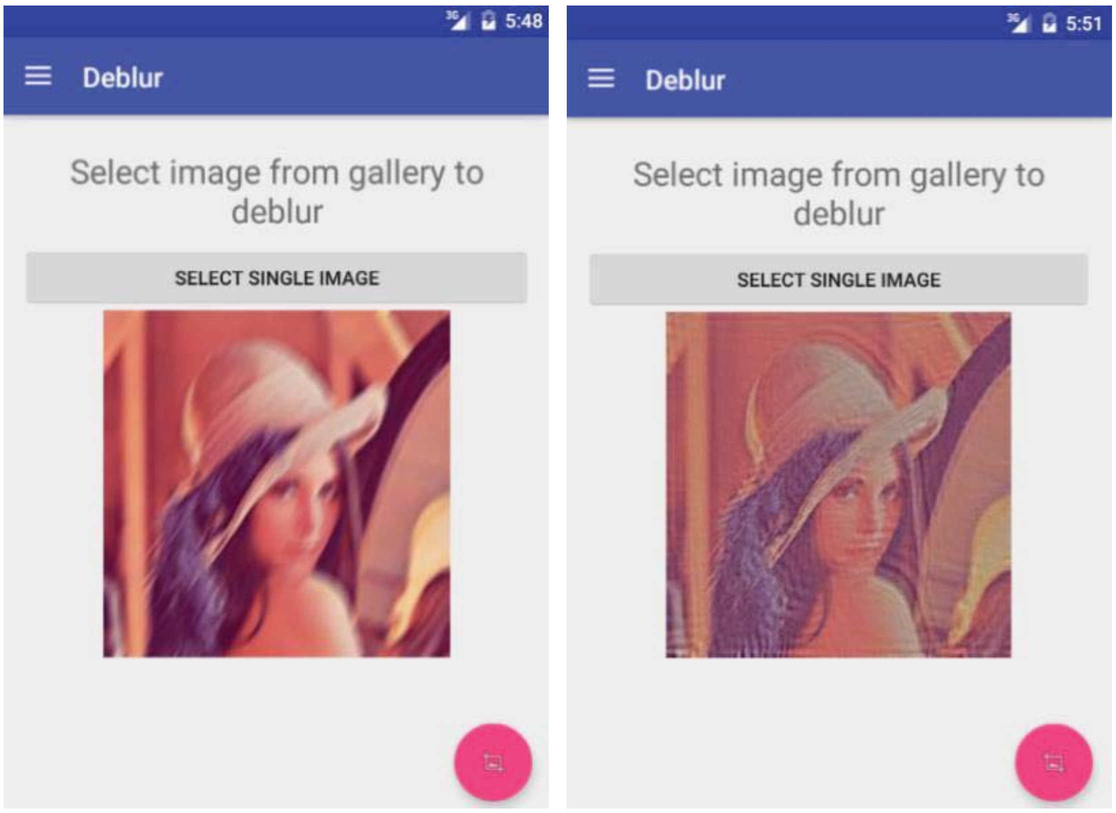

5:48
Deblur
Select image from gallery to deblur
SELECT SINGLE IMAGE
5:51
Deblur
Select image from gallery to deblur
SELECT SINGLE IMAGE

Figure 7-15 : Lena test image example. The resulting image was generated by the android application after the user chose it from the android device library and pressed the pink execution button.

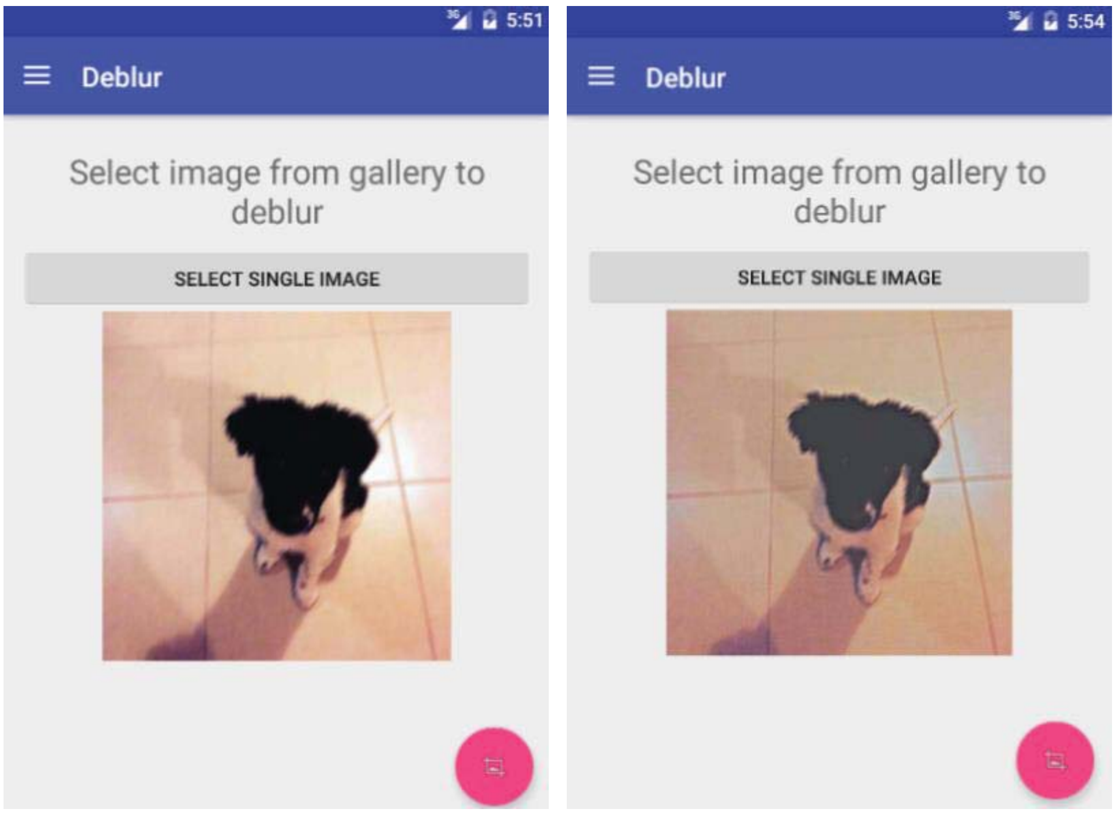
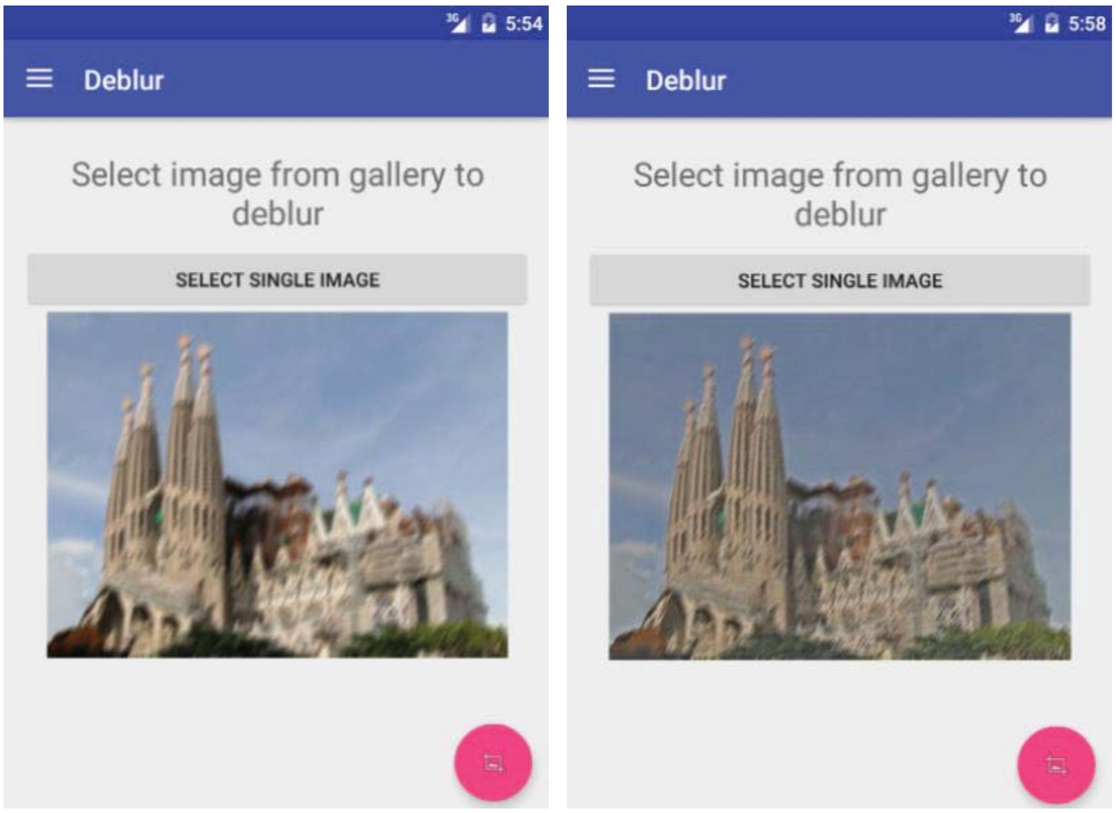

5:51
Deblur
Select image from gallery to deblur
SELECT SINGLE IMAGE
5:54
Deblur
Select image from gallery to deblur
SELECT SINGLE IMAGE

Figure 7-16 : Dog test image example. The resulting image was generated by the android application after the user chose it from the android device library and pressed the pink execution button.



5:54

Deblur

Select image from gallery to deblur

SELECT SINGLE IMAGE

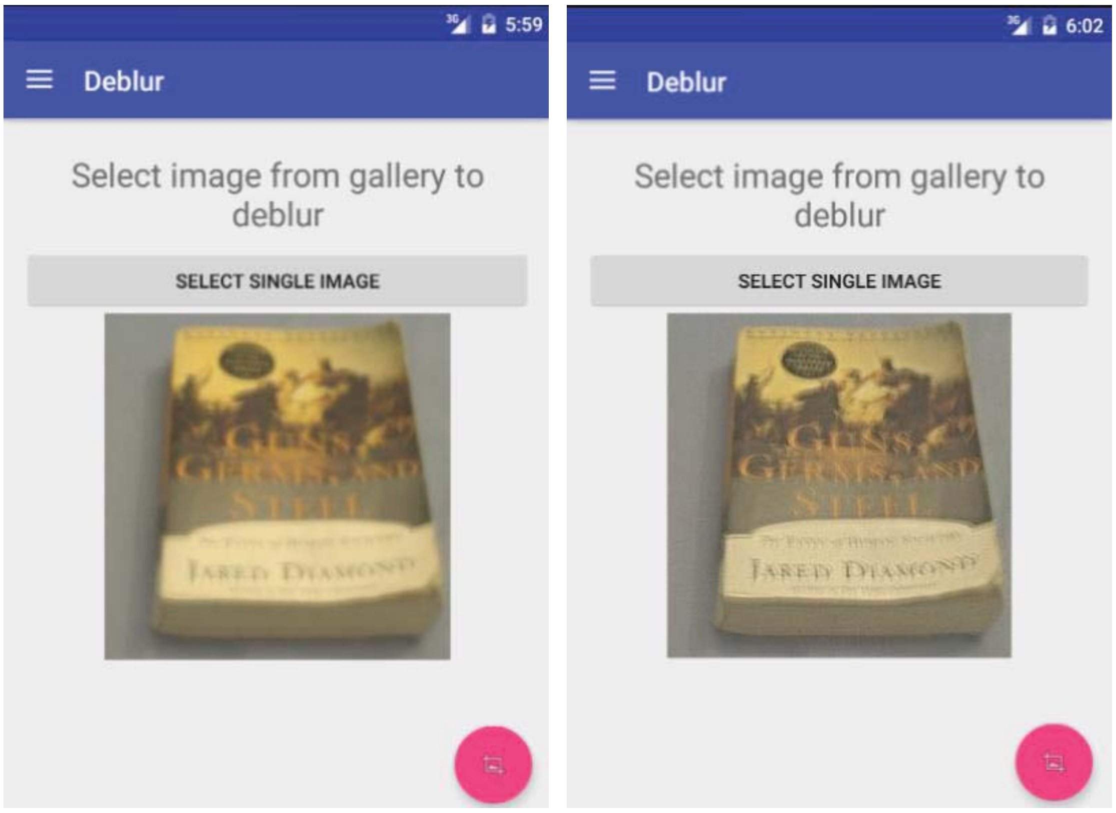
5:58

Deblur

Select image from gallery to deblur

SELECT SINGLE IMAGE

Figure 7-17 : Sagrada familia test image example. The resulting image was generated by the android application after the user chose it from the android device library and pressed the pink execution button.



5:59

Deblur

Select image from gallery to deblur

SELECT SINGLE IMAGE

6:02

Deblur

Select image from gallery to deblur

SELECT SINGLE IMAGE

Figure 7-18 : Book test image example. The resulting image was generated by the android application after the user chose it from the android device library and pressed the pink execution button.

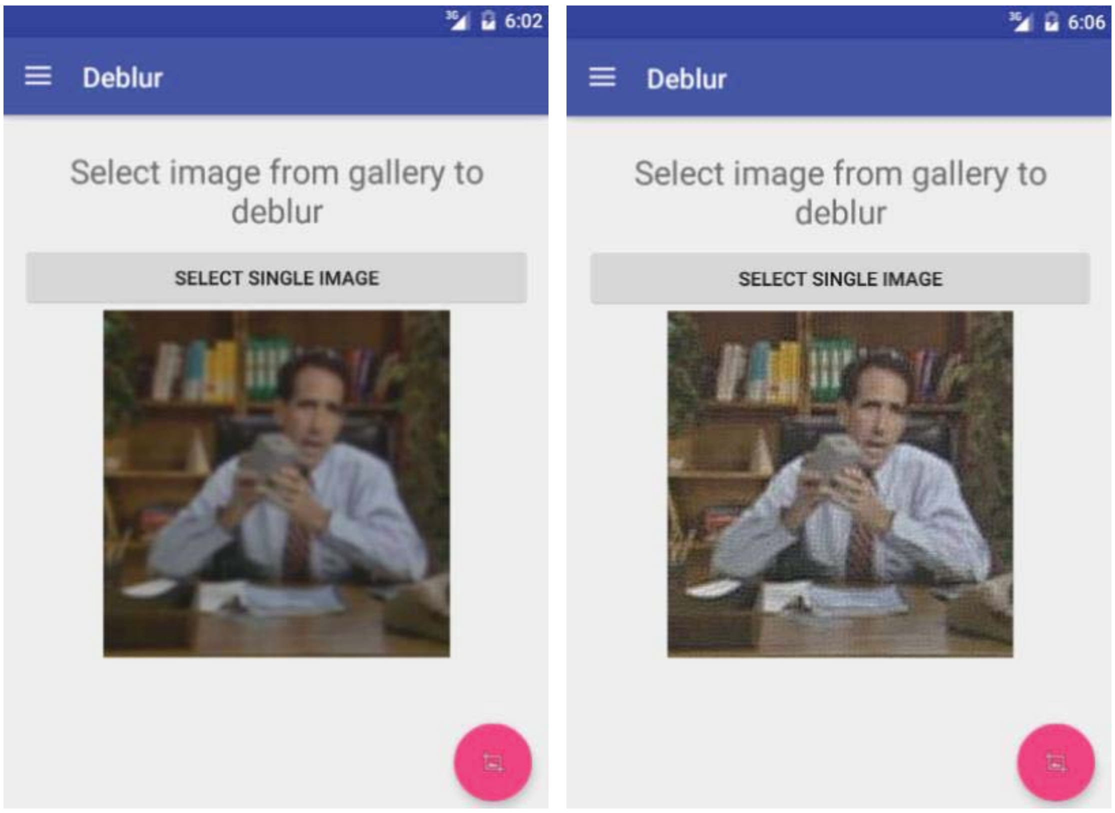

6:02
Deblur
Select image from gallery to deblur
SELECT SINGLE IMAGE
6:06
Deblur
Select image from gallery to deblur
SELECT SINGLE IMAGE

Figure 7-19 : Sales man test image example. The resulting image was generated by the android application after the user chose it from the android device library and pressed the pink execution button.

As it was made obvious the produced images of the same quality as the images shown in the C++ implementation Chapter. The latent images are indeed sharper and clearer that the initial input image. However, they still are not perfect. This is the result of the trade-off between performance and quality which has even more specific restrictions when referring to mobile devices. The results of this thesis are discussed in the following and final chapter of this thesis.

Chapter 8
DISCUSSION AND FUTURE WORK

As it has been obvious the issue of blind or non-blind image deblurring and image denoising is a very complex one. Since blur has removed the high frequency components from the image, even if the point spread function was known a simple deconvolution would not be enough, meaning that the latent image has to be estimated. The purpose of this thesis was to investigate methods for recovering the latent(sharp) image meaning the image that would have occurred if the camera was perfectly standstill. After analyzing the relevant literature and after taking into consideration hardware perquisites, we came to the conclusion that the most suitable method was a Bayesian inference framework based method that used a MAP scheme augmented by an edge detection feature.

More specifically, the edge detection and sharp edges augmentation procedure generates an image to be used as a prior for the MAP scheme. In order to retrieve the final deblurred image the MAP formulation was solved initially for the blur kernel, assuming that the latent image was known, and then form the latent image, assuming the point spread function in known. Thus, the process could not be completed directly but an iterative scheme was employed. In each iteration the estimated kernel was updated and the latent image was recalculated. After the estimated kernel was refined in a manner that the update between iterations was very small, the final deconvolution took place. In this part the estimated kernel and the initial blurry image are used to extract the final result.

In order to make the aforementioned scheme computationally efficient a multiscale approach was also used starting from the smallest scale, meaning the original blurry image resized to a fraction of the initial dimensions. The main benefit of this method was that it was fast and that it offered low computational cost while yielding satisfactory results. The aforementioned method was not only prototyped in MATLAB but also was implemented in C++ using the open source OpenCV computer vision library and then integrated for the ANDROID operating system for mobile devices such as phones and tablets.

On the other hand, there were some drawbacks. The recovered latent images were not always perfect. Artifacts were also present since noise was enhanced along with other features. If other more computationally expensive methods were used the result might have been better. Thus, the case would be to take into account the non-uniformity of the point spread function across the blurry image. This could be achieved by using a homography based method. Such methods were analyzed in the second chapter of this thesis based on Wang and Tao review paper[1]. Thus, it would be a challenge to implement a computationally costly method in an embedded system. Additionally, it would have been very interesting if one investigated the trade-off between speed and quality since each method yields different results for different processing time. Defining the threshold for a satisfactory or a unsatisfactory result may be many times subjective. On the greater scheme, there is always the main objective for a researcher to make fast methods faster or find new methods that yield results with better quality.

Regarding the application level, using a multicore approach instead of a single-threaded solution would be an important topic to continue the currently presented work. Another very interesting application would be to achieve a real time deblurring process meaning that the process time would need to be measured under a second and for a live video deblurring under below 1/24th of a second. Finally, it is important to conclude that integrating image processing techniques and methods with embedded systems for a real life application is a challenging but promising task.

This page is intentionally left blank

REFERENCES

[1] R. Wang and D. Tao, "Recent Progress in Image Deblurring," *ArXiv 1409.6838*, pp. 1–53, 2014.

[2] L. B. Lucy, "An iterative technique for the rectification of observed distributions," *Astron. J.*, vol. 79, p. 745, Jun. 1974.

[3] R. Fergus, B. Singh, A. Hertzmann, S. T. Roweis, and W. T. Freeman, "Removing camera shake from a single photograph," *ACM Trans. Graph.*, vol. 25, no. 3, p. 787, 2006.

[4] T. Kenig, Z. Kam, and A. Feuer, "Blind image deconvolution using machine learning for three-dimensional microscopy.," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 32, no. 12, pp. 2191–204, Dec. 2010.

[5] A. Chakrabarti, T. Zickler, and W. T. Freeman, "Analyzing spatially-varying blur," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2010, pp. 2512–2519.

[6] O. Whyte, J. Sivic, A. Zisserman, and J. Ponce, "Non-uniform deblurring for shaken images," *Int. J. Comput. Vis.*, vol. 98, no. 2, pp. 168–186, 2012.

[7] X. Zhu and P. Milanfar, "Image reconstruction from videos distorted by atmospheric turbulence," in *IEEE Transactions on Image Processing*, 2010, vol. 7543, p. 75430S–75430S–8.

[8] W. Richardson and W. Richardson, "Bayesian-based iterative method of image restoration," *J. Opt. Soc. Am.*, vol. 62, no. I, pp. 55–59, 1972.

[9] Y.-W. Tai, P. Tan, and M. S. Brown, "Richardson-Lucy Deblurring for Scenes under a Projective Motion Path.," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 33, no. 8, pp. 1603–18, Aug. 2011.

[10] A. Levin and W. T. Freeman, "Deconvolution using natural image priors," *Artif. Intell.*, no. 6, pp. 0–2, 2007.

[11] S. Cho, H. Cho, Y. W. Tai, and S. Lee, "Registration based non-uniform motion deblurring," *Comput. Graph. Forum*, vol. 31, no. 7 PART2, pp. 2183–2192, Sep. 2012.

[12] T. S. Cho, N. Joshi, C. L. Zitnick, S. B. Kang, R. Szeliski, and W. T. Freeman, "A content-aware image prior," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2010, pp. 169–176.

[13] S. Osher and L. I. Rudin, "Feature-Oriented Image Enhancement Using Shock Filters," *SIAM J. Numer. Anal.*, vol. 27, no. 4, pp. 919–940, Aug. 1990.

[14] F. Russo and G. Ramponi, "Fuzzy operator for sharpening of noisy images," *Electron. Lett.*, vol. 28, no. 18, pp. 1715–1717, Aug. 1992.

[15] J. G. M. Schavemaker, M. J. T. Reinders, J. J. Gerbrands, and E. Backer, "Image sharpening by morphological filtering," *Pattern Recognit.*, vol. 33, no. 6, pp. 997–1012, Jun. 2000.

[16] G. Gilboa, N. Sochen, and Y. Y. Zeevi, "Forward-and-backward diffusion processes for adaptive image enhancement and denoising," *IEEE Trans. Image Process.*, vol. 11, no. 7, pp. 689–703, Jan. 2002.

[17] S. Cho and S. Lee, "Fast motion deblurring," *ACM Trans. Graph.*, vol. 28, no. 5, pp. 145:1–145–8, 2009.

[18] M. S. C. Almeida and L. B. Almeida, "Blind and semi-blind deblurring of natural images," *IEEE Trans. Image Process.*, vol. 19, no. 1, pp. 36–52, Jan. 2010.

[19] L. Xu and J. Jia, "Two-phase kernel estimation for robust motion deblurring," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 6311 LNCS, no. PART 1, pp. 157–170, 2010.

[20] A. Levin, Y. Weiss, F. Durand, and W. T. Freeman, "Understanding blind deconvolution algorithms," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 33, no. 12, pp. 2354–2367, 2011.

[21] A. Levin, Y. Weiss, F. Durand, and W. T. Freeman, "Efficient marginal likelihood optimization in blind deconvolution," in *CVPR 2011*, 2011, pp. 2657–2664.

[22] A. Levin, "Efficient Marginal Likelihood Optimization in Blind Deconvolution ," in *Cvpr*, 2011.

[23] C. Wang, Y. Yue, F. Dong, Y. Tao, X. Ma, G. Clapworthy, H. Lin, and X. Ye, "Nonedge-specific adaptive scheme for highly robust blind motion deblurring of natural imagess.," *IEEE Trans. Image Process.*, vol. 22, no. 3, pp. 884–97, Mar. 2013.

[24] C. Wang, L. Sun, P. Cui, J. Zhang, and S. Yang, "Analyzing image deblurring through three paradigms," *IEEE Trans. Image Process.*, vol. 21, no. 1, pp. 115–129, Jan. 2012.

[25] D. H. Brainard and W. T. Freeman, "Bayesian color constancy.," *J. Opt. Soc. Am. A. Opt. Image Sci. Vis.*, vol. 14, no. 7, pp. 1393–411, Jul. 1997.

[26] U. Schmidt, Q. Gao, and S. Roth, "A generative perspective on MRFs in low-level vision," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2010, pp. 1751–1758.

[27] U. Schmidt, K. Schelten, and S. Roth, "Bayesian deblurring with integrated noise estimation," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2011, pp. 2625–2632.

[28] L. Zhang, A. Deshpande, and X. Chen, "Denoising vs. deblurring: HDR imaging techniques using moving cameras," *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, pp. 522–529, 2010.

[29] J. W. Miskin and D. J. C. Mackay, "Ensemble Learning for Blind Source Separation," *ICA Princ. Pract.*, 2000.

[30] C. M. Bishop, "Pattern Recognition and Machine Learning (Information Science and Statistics)," Aug. 2006.

[31] M. I. Jordan, Z. Ghahramani, T. S. Jaakkola, and L. K. Saul, "Introduction to variational methods for graphical models," *Mach. Learn.*, vol. 37, no. 2, pp. 183–233, 1999.

[32] A. N. Ti, "Solution of incorrectly formulated problems and the regularization method," *Sov. Math. Dokl*, vol. 4, no. 4, pp. 1035–1038, Nov. 1963.

[33] L. I. Rudin, S. Osher, and E. Fatemi, "Nonlinear total variation based noise removal algorithms," *Phys. D Nonlinear Phenom.*, vol. 60, no. 1–4, pp. 259–268, Nov. 1992.

[34] D. Krishnan, T. Tay, and R. Fergus, "Blind deconvolution using a normalized sparsity measure," *Comput. Vis. Pattern Recognit. (CVPR), 2011 IEEE Conf.*, pp. 233–240, 2011.

[35] L. Xu, S. Zheng, and J. Jia, "Unnatural L0 sparse representation for natural image deblurring," *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, pp. 1107–1114, 2013.

[36] S. Lefkimmiatis and M. Unser, "Poisson image reconstruction with hessian schatten-norm regularization," *IEEE Trans. Image Process.*, vol. 22, no. 11, pp. 4314–4327, 2013.

[37] K. Papafitsoros and C. B. Sch??nlieb, "A Combined First and Second Order Variational Approach for Image Reconstruction," *J. Math. Imaging Vis.*, vol. 48, no. 2, pp. 1–31, May 2013.

[38] T. Goldstein and S. Osher, "The Split Bregman Method for L1-Regularized Problems," *SIAM J. Imaging Sci.*, vol. 2, no. 2, pp. 323–343, Jan. 2009.

[39] D. L. Donoho and J. M. Johnstone, "Ideal spatial adaptation by wavelet shrinkage," *Biometrika*, vol. 81, no. 3, pp. 425–455, 1994.

[40] S. Boyd, N. Parikh, B. P. E Chu, and J. Eckstein, "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers," *Found. Trends® Mach. Learn.*, vol. 3, no. 1,

pp. 1–122, Jan. 2010.

[41] M. S. C. Almeida and M. A. T. Figueiredo, “Blind image deblurring with unknown boundaries using the alternating direction method of multipliers,” in *Image Processing (ICIP), 2013 20th IEEE International Conference on*, 2013, pp. 586–590.

[42] N. Joshi, S. B. Kang, C. L. Zitnick, and R. Szeliski, “Image deblurring using inertial measurement sensors,” *ACM Trans. Graph.*, vol. 29, no. 4, p. 1, Jul. 2010.

[43] R. Szeliski and H.-Y. Shum, “Creating full view panoramic image mosaics and environment maps,” in *Computer Graphics and Iteractive Techniques*, 1997, pp. 251–258.

[44] A. Almohammad and G. Ghinea, “Stego image quality and the reliability of PSNR,” in *2010 2nd International Conference on Image Processing Theory, Tools and Applications*, 2010, pp. 215–220.

[45] D. M. Rouse and S. S. Hemami, “Understanding and simplifying the structural similarity metric,” in *2008 15th IEEE International Conference on Image Processing*, 2008, pp. 1188–1191.

[46] A. Levin, Y. Weiss, F. Durand, and W. T. Freeman, “Understanding and evaluating blind deconvolution algorithms,” in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 1964–1971.

[47] T. S. Cho, S. Paris, B. K. P. Horn, and W. T. Freeman, “Blur kernel estimation using the radon transform,” *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, pp. 241–248, 2011.

[48] Q. Shan, J. Jia, and A. Agarwala, “High-quality motion deblurring from a single image,” *ACM Trans. Graph.*, vol. 27, no. 3, p. 1, 2008.

[49] C. Tomasi and R. Manduchi, “Bilateral Filtering for Gray and Color Images,” in *International Conference on Computer Vision*, 1998, pp. 839–846.

[50] J. Chen, L. Yuan, C. K. Tang, and L. Quan, “Robust dual motion deblurring,” in *26th IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, 2008, pp. 1–8.

[51] L. Alvarez and L. Mazorra, “Signal and Image Restoration Using Shock Filters and Anisotropic Diffusion,” *SIAM J. Numer. Anal.*, vol. 31, no. 2, pp. 590–605, Apr. 1994.

[52] G. Gilboa, N. Sochen, and Y. Y. Zeevi, “Regularized shock filters and complex diffusion,” *Comput. Vis. — ECCV 2002 7th Eur. Conf. Comput. Vis.*, no. 1, pp. 399–413, May 2002.

[53] D. Heger, “Mobile Devices-An Introduction to the Android Operating Environment Design, Architecture, and Performance Implications,” *DHTechnologies (DHT), retrieved March*. pp. 1–6, 2012.

[54] D. Bornstein, “Dalvik vm internals,” in *Google I/O Developer Conference*, 2008, vol. 23, pp. 17–30.

[55] F. Maker and Y. Chan, “A survey on android vs. linux,” *Univ. Calif.*, pp. 1–10, 2009.

[56] Dominique A. Heger, *Quantifying IT Stability, 2nd Edition : Grid, Cloud, Cluster, and SMP Systems*. 2011.

[57] P. Brady, “Android anatomy and physiology,” in *Google IO developer conference*, 2008, p. 119.

[58] Liang, “System Integration for the Android Operating System.” National Taipei University, 2010.

[59] “Linux Wikipedia,” 2013. [Online]. Available: http://en.wikipedia.org/wiki/Linux.

This page is intentionally left blank

APPENDIX

The appendix is organized in the following manner. Part A includes the MATLAB implementation, part B includes the C/C++ implementation and part C the android application.

A. MATLAB CODE

analyze.m

```
function kernelArray=analyze(initsize)
        elem=initsize;i=1;flag=true;
        while flag
                kernelArray(i)=elem;
                elem=floor(elem/2);
                if elem<=5
                        if rem(elem,2)==0
                                elem=elem+1;
                        end
                        kernelArray(i+1)=elem;
                        flag=false;
                        break;
                end
                if rem(elem,2)==0
                        elem=elem+1;
                end
                i=i+1;
        end
return
```

deblur.m

```
function deblur(blurImg)
initsize=21;iterScaleNum=10;origy=blurImg;truey=blurImg;
kernelArray=analyze(initsize);isFinal=false;
Scales=length(kernelArray);
availmage=r2g(truey(1:2^(Scales-1):end,1:2^(Scales-1):end,:));
for iter=1:Scales
        if iter==Scales
                isFinal=true;
        end
        riter=Scales-iter+1;
        y=truey(1:2^(riter-1):end,1:2^(riter-1):end,:);
        sizef=kernelArray(riter);
        f=zeros(sizef,sizef);
        [deblurred_image,psf]=coreDeconv(y,availmage,f,iterScaleNum,isFinal,origy,iter);
        if iter<Scales
                availmage=mycubic_2(deblurred_image);
        end
end
figure,imshow(img),title('Original Image')
figure,imagesc(psf),colormap(gray),title('kernel')
return;
```

deconv.m

```
function deblurred_image=deconv(Blurred,PSF,w0alpha)
ks = (size(PSF) - 1)/2;
Blurred = padarray(Blurred, ks, 'replicate', 'both');
for a=1:4
Blurred = edgetaper(Blurred, PSF);
end
[theResult] = deconv_fn(Blurred, PSF, w0alpha);
d=size(theResult);
xmin=ks(1)+1;
ymin=ks(2)+1;
width=(d(1)-ks(1)-ks(1)-1);
height=(d(2)-ks(2)-ks(2)-1);
e=[xmin ymin height width];
deblurred_image = imcrop(theResult,e);
return;
function L=deconv_fn(Blurred,PSF,w0alpha)
k0=1;
kx = [1 -1];
ky = [1 -1]';
kxx = [1 -2 1];
kyy = [1 -2 1]';
kxy=[1 -1;-1 1];
Fk0=psf2otf(k0,size(Blurred));
Fkx=psf2otf(kx,size(Blurred));
Fky=psf2otf(ky,size(Blurred));
Fkxx=psf2otf(kxx,size(Blurred));
Fkyy=psf2otf(kyy,size(Blurred));
Fkxy=psf2otf(kxy,size(Blurred));
delta0=Fk0.*conj(Fk0);
delta1=Fkx.*conj(Fkx);
delta2=Fky.*conj(Fky);
delta3=Fkxx.*conj(Fkxx);
delta4=Fkxy.*conj(Fkxy);
delta5=Fkyy.*conj(Fkyy);
delta12=delta1+delta2;
delta345=delta3+delta4+delta5;
delta=w0alpha*(delta0+0.5*delta12+0.25*delta345);
FFTy=fftn(Blurred);
FFTK=psf2otf(PSF,size(Blurred));
FFTLnom=(FFTy.*conj(FFTK));
FFTLnom=FFTLnom.*delta;
FFTLdenom=((FFTK.*conj(FFTK)).*delta+(Fkx.*conj(Fkx))+(Fky.*conj(Fky)));
FFTL=FFTLnom./FFTLdenom;
L=ifftn(FFTL);
return;
```

delta_kernel.m

```
function out = delta_kernel(s)

% Author: Rob Fergus
% Version: 1.0, distribution code.
% Project: Removing Camera Shake from a Single Image, SIGGRAPH 2006 paper
% Copyright 2006, Massachusetts Institute of Technology


    %%% if even size, make odd
    if (mod(s,2)==0)
        s = s + 1;
    end

    out = zeros(s);

    c = floor(s/2)+1;

    out(c,c)=1;


% Result
%--------------------
% 0    0    0    0    0
% 0    0    0    0    0
% 0    0    1    0    0
% 0    0    0    0    0
% 0    0    0    0    0

    return;
```

estK.m

```
function ker=estK(Prediction,OriginalBlurredScaledSingleChannel,PSF,numberOfIterations,belta,freqCut,mode)
%Q:Prediction , P:Result , f:kernel
ex=PSF;
error=1;
iteration=1;
ks=getPadSize(PSF);
ksi=2*ks;
PredictionPadded=padarray(Prediction,ks);
BlurredPadded=padarray(OriginalBlurredScaledSingleChannel,ks, 'replicate', 'both');
while (error>0.0001)&&(iteration<numberOfIterations)
    ker=getK(PredictionPadded,BlurredPadded,PSF,belta,freqCut);%Q:Prediction , P:Result , f:kernel
    error=norm(ker-ex,2);
    ex=ker;
    if mode==0
    elseif mode==1
    BlurredPadded=OriginalBlurredScaledSingleChannel;
    BlurredPadded=padarray(OriginalBlurredScaledSingleChannel,ks, 'replicate', 'both');
    end
    iteration=iteration+1;
end
return;
function ker=getK(PredictionPadded,BlurredPadded,PSF,belta,freqCut) %Prediction , Result , kernel
A=fft2(PredictionPadded);      %Prediction
b=fft2(BlurredPadded);      %Result
ker=delta_kernel(size(PSF));
for i=1:1
     k=psf2otf(ker,size(A));
     d=conj(A).*b-(conj(A).*A+belta).*k;
     rentaNom=(conj(A.*d).*b-conj(A.*d).*A.*k-belta.*conj(d).*k);
     rentaDeNom=(conj(A.*d).*A.*d+belta*conj(d).*d);
     renta=rentaNom./rentaDeNom;
    k=k+renta.*d;

     ker=otf2psf(k,size(PSF));
     ker=abs(ker);
     maxValue=max(max(ker));
     pos_ind=find(ker<maxValue/freqCut);
     ker(pos_ind)=0;
     thesum=sum(sum(ker));
     ker=ker/thesum;
end
return;

function ks=getPadSize(f)
ks=(size(f)-1)/2;
return;
```

getImagePSF.m

```
function
[deblurredImage,PSFResult]=getImagePSF(OriginalBlurredScaled,OriginalBlurredMinimumScaleSingleChannel,PSF,iterSc
aleNum,isFinal,OriginalBlurred,sigmaSpatial,sigmaRange,alpha,belta,freqCut,mode,currentScale)
Temp=OriginalBlurredMinimumScaleSingleChannel;
%Start loop
for iteration=1:iterScaleNum
%initialize bilateral
[inputWidth,inputHeight] = size(Temp);
sigmaSpatial = min( inputWidth, inputHeight ) / 16;
if sigmaRange==0
edgeMin = min( Temp( : ) );
edgeMax = max( Temp( : ) );
edgeDelta = edgeMax - edgeMin;
sigmaRange = 0.1 * edgeDelta;
end
figure(4);
subplot(3,3,(currentScale-1)*3+1);
imshow(Temp);
title('Greyscale image');
%Temp = medfilt2(Temp);
Prediction=bilateralFilter(Temp,Temp,sigmaSpatial,sigmaRange,sigmaSpatial,sigmaRange);
subplot(3,3,(currentScale-1)*3+2);
imshow(Prediction);
title('Applying bilateral filter');
Prediction=shock(Prediction,30,0.9^(iteration-1),1,'org');
subplot(3,3,(currentScale-1)*3+3);
imshow(Prediction);
title('Applying shock filter');
OriginalBlurredScaledSingleChannel=r2g(OriginalBlurredScaled);
figure(10);
subplot(3,2,((currentScale-1)*2+iteration))
PSFResult=estK(Prediction,OriginalBlurredScaledSingleChannel,PSF,10,belta,freqCut,mode);
PSFMaxValue=max(max(PSFResult));
PSFResultImage=PSFResult/PSFMaxValue;
imshow(PSFResultImage),title(strcat('Scale=',num2str(currentScale),'/','Iteration=',num2str(iteration)));
figure(14);
if ((iteration~=iterScaleNum)||(~isFinal))
    deblurredImage=deconv(OriginalBlurredScaledSingleChannel,PSFResult,alpha);
    Temp=deblurredImage;
end
if ((iteration==iterScaleNum)&&(isFinal))
    deblurredImage=deconv(OriginalBlurred,PSFResult,alpha);
end
subplot(3,2,((currentScale-1)*2+iteration))
imshow(deblurredImage);
end
return;
```

This page is intentionally left blank

B. C/C++ IMPLEMENTATION

- **Mat getChannel(Mat, int)**

```
Mat getChannel(Mat src, int theChannel){
	int index = 0;
	Mat spl[3];
	cv::split(src, spl);
	if (theChannel == 0 || theChannel == 1 || theChannel == 2){ index = theChannel; }
	return spl[index];
}
```

- **Mat deBlur::fft2(Mat input)**

```
Mat deBlur::fft2(Mat input){
	Mat fft = Mat::zeros(input.size(), CV_64FC2);

	if (input.channels() == 1){
		cv::dft(input, fft, DFT_COMPLEX_OUTPUT, input.rows);
	}
	else{
		exit(37);
	}
	return fft; //Matrix with 2 channels
}
```

- **Mat deBlur::ifft2(Mat input)**

```
Mat deBlur::ifft2(Mat input){
	Mat inputMat = input;
	Mat inverseTransform = Mat::zeros(input.size(), CV_64FC1);

	if (inputMat.channels() == 2){
		cv::dft(inputMat, inverseTransform, DFT_INVERSE | DFT_SCALE | DFT_REAL_OUTPUT, input.rows);
	}
	else{
		exit(37);
	}
	return inverseTransform;
}
```

- **Mat deBlur::r2g(Mat input)**

```
Mat deBlur::r2g(Mat input){
	Mat output;
	if (input.channels() >1){
		//cv::cvtColor(input, output, CV_BGR2GRAY);
		vector<Mat> channels(3);
		cv::split(input, channels);
		output = ((double)1 / (double)3)*(channels[0] + channels[1] + channels[2]);
	}
	else
	{
		output = input;
	}
	output.convertTo(output, CV_64FC1);
	return output;
}
```

- **cv::Size deBlur::getPadSize(Mat f)**

```
cv::Size deBlur::getPadSize(Mat f){
	return Size(((f.size().width - 1) / 2), ((f.size().height - 1) / 2));
}
```

- **cv::Mat deBlur::paddarray(Mat input, Size padding, std::string method, std::string direction)**

```
cv::Mat deBlur::paddarray(Mat input, Size padding, std::string method, std::string direction){
	Mat padded;
	int top, bottom, left, right;
	int borderType = BORDER_CONSTANT;
	Scalar value;
	top = padding.height;
	bottom = padding.height;
	left = padding.width;
	right = padding.width;
	value = Scalar(0, 0, 0);
	if (direction == "both")
	{
		top = padding.height;
		bottom = padding.height;
		left = padding.width;
		right = padding.width;
	}
	else if (direction == "post")
	{
		top = 0;
		bottom = padding.height;
		left = 0;
		right = padding.width;
	}
	else if (direction == "pre")
	{
		top = padding.height;
		bottom = 0;
		left = padding.width;
		right = 0;
	}
	if (method == "zero")
	{
		borderType = BORDER_CONSTANT;
	}
	else if (method == "replicate")
	{
		borderType = BORDER_REPLICATE;
	}
	copyMakeBorder(input, padded, top, bottom, left, right, borderType, value);
	return padded;
}
```

- **void mulComplexMats(Mat A, Mat B, Mat &result, int flag, bool what)**

```
void mulComplexMats(Mat A, Mat B, Mat &result, int flag, bool what){
	Mat    AC1, AC2, BC1, BC2, RC1, RC2;
	AC1 = Mat::zeros(A.size(), CV_64FC1);
	AC2 = Mat::zeros(A.size(), CV_64FC1);
	BC1 = Mat::zeros(B.size(), CV_64FC1);
	BC2 = Mat::zeros(B.size(), CV_64FC1);
	RC1 = Mat::zeros(B.size(), CV_64FC1);
	RC2 = Mat::zeros(B.size(), CV_64FC1);
	AC1 = getChannel(A, 0);
	AC2 = getChannel(A, 1);
	BC1 = getChannel(B, 0);
	BC2 = getChannel(B, 1);

	RC1 = AC1.mul(BC1) - AC2.mul(BC2);
	RC2 = AC1.mul(BC2) + AC2.mul(BC1);
	vector<Mat> complexConjugate;
	complexConjugate.push_back(RC1);
	complexConjugate.push_back(RC2);
	cv::merge(complexConjugate, result);

	return;
}
```

- **Mat deBlur::psf2otf(Mat inputPSF, Size finalSize)**

```
Mat deBlur::psf2otf(Mat inputPSF, Size finalSize){
	Mat otf;
	Mat psf;
	Size padSize = finalSize - inputPSF.size();
	psf = paddarray(inputPSF, padSize, "zero", "post");
	int pixelsShiftX = (int)(floor(inputPSF.size().width / 2));
	int pixelsShiftY = (int)(floor(inputPSF.size().height / 2));
	shift(psf, psf, Point(-pixelsShiftX, -pixelsShiftY), BORDER_WRAP);
	otf = fft2(psf);

	return otf;
}
```

- **Mat deBlur::otf2psf(Mat OTF, Size outSize)**

```
Mat deBlur::otf2psf(Mat OTF, Size outSize){
	//Init
	Mat psf;
	Mat psfResult;
	Mat newPSF;
	vector<Mat> complexOTF;
	if (OTF.channels() == 2){
		psf = ifft2(OTF);
		if (psf.channels() == 1){
			int pixelsShiftX = (int)(floor((outSize.width) / 2));
			int pixelsShiftY = (int)(floor((outSize.height) / 2));
			shift(psf, psf, Point(pixelsShiftX, pixelsShiftY), BORDER_WRAP);
			psfResult = psf(Rect(0, 0, outSize.width, outSize.height));
		}
	}
	else{
		exit(5);
	}
	return psfResult;
}
```

- **void deBlur::readImage()**

```
void deBlur::readImage(){
	Mat B;
	B = this->img;
	B.convertTo(B, CV_64F);
	B = (1.0 / 255.0)*B;
	this->OriginalBlurred = B;
	return;
}
```

- **void deBlur::readImage(std::string thefile)**

```
void deBlur::readImage(std::string thefile){
	Mat B;
	this->img = imread(thefile, CV_64F); //loads color if it is available
	B = this->img;
	B.convertTo(B, CV_64F);
	B = (1.0 / 255.0)*B;
	this->OriginalBlurred = B;
	return;
}
```

- **void deBlur::readImage(Mat thefile)**

```
void deBlur::readImage(Mat thefile){
        Mat B;
        B = thefile;
        B.convertTo(B, CV_64F);
        B = (1.0 / 255.0)*B;
        this->OriginalBlurred = B;
        return;
}
```

- **void deBlur::iterativeDeblurring(void)**

```
void deBlur::iterativeDeblurring(void){
        int currentIterationFactor = 0;
        double denominator;double resizeFactor;Mat InitialPSF;
        Mat temp;
        int InitialPSFSize;
        this->isFinal = false;
        this->scales = this->kernelArray.size();
        vector<Size> imagePyramid;
        for (int j= 0; j < this->scales; j++){
                int i = this->scales-1-j;
                Size thisScaleSize = Size((int)((1 / pow(2, i))*this->OriginalBlurred.size().width), (int)((1 / pow(2,
i))*this->OriginalBlurred.size().height));
                imagePyramid.push_back(thisScaleSize);
        }
        resize(this->OriginalBlurred, this->OriginalBlurredScaledMinSingleChannel, imagePyramid.at(0));
        OriginalBlurredScaledMinSingleChannel = r2g(OriginalBlurredScaledMinSingleChannel);
        for (int thisIter = 0; thisIter < this->scales; thisIter++){
                if (thisIter == this->scales - 1){
                        this->isFinal = true;
                }
                //Resize---------------------------------------------------------
                currentIterationFactor = scales - thisIter;// +1;
                resize(this->OriginalBlurred, this->OriginalBlurredScaled, imagePyramid.at(thisIter));
                //Get size-------------------------------------------------------------------
                InitialPSFSize = this->kernelArray.at(currentIterationFactor - 1); // ctor);
                InitialPSF = Mat::zeros(InitialPSFSize, InitialPSFSize, CV_64F);
                //[deblurredImage, PSFResult] ---------------------------------------------
                getImagePSF(this->OriginalBlurredScaled, this->OriginalBlurredScaledMinSingleChannel, InitialPSF,
iterScaleNum, isFinal, this->OriginalBlurred, this->sigmaRange);
                //Bicubic interp-----------------------------------------------------------------
                if (thisIter < this->scales-1){
                        cv::resize(this->deblurredImage, this->OriginalBlurredScaledMinSingleChannel,
imagePyramid.at(thisIter+1), 0, 0, CV_INTER_CUBIC);
                }
        }
        return;
}
```

- **void deBlur::resizeBlurredToMinimumScale(void)**

```
void deBlur::resizeBlurredToMinimumScale(void){
	this->scales = this->kernelArray.size();
	this->scalesDenominator = pow((scales - 1), 2);
	this->scalesFactor = 1 / scalesDenominator;
	cv::Size s = this->OriginalBlurred.size();
	cv::Size s2 = Size((int)((double)s.height * scalesFactor), (int)((double)s.width * scalesFactor));
	resize(this->OriginalBlurred, this->OriginalBlurredScaledMinSingleChannel, s2);	//resize image
	OriginalBlurredScaledMinSingleChannel = r2g(OriginalBlurredScaledMinSingleChannel);
	return;
}
```

- **Mat deBlur::deconv_fn(Mat &Blurred, Mat &PSF, double &w0alpha)**

```
Mat deBlur::deconv_fn(Mat &Blurred, Mat &PSF, double &w0alpha){
	Mat L, delta;Mat delta0, delta1, delta2, delta3, delta4, delta5, delta6, delta12, delta345;
	Mat Fk0, Fkx, Fky, Fkxx, Fkyy, Fkxy, FFTK, FFTy, FFTL, FFTLnom, FFTLdenom;
	Mat FFTLdenom1, FFTLdenom2, FFTLdenom3;
	Mat k0 = (Mat_<double>(1, 1) << 1);Mat kx = (Mat_<double>(1, 2) << 1, -1);
	Mat ky = (Mat_<double>(2, 1) << 1, -1);
	Mat kxx = (Mat_<double>(1, 3) << 1, -2, 1);
	Mat kyy = (Mat_<double>(3, 1) << 1, -2, 1);Mat kxy = (Mat_<double>(2, 2) << 1, -1, -1, 1);
	Fk0 = psf2otf(k0, Blurred.size());Fkx = psf2otf(kx, Blurred.size());Fky = psf2otf(ky, Blurred.size());Fkxx = psf2otf(kxx,
	Blurred.size());Fkyy = psf2otf(kyy, Blurred.size());
	Fkxy = psf2otf(kxy, Blurred.size());
	//delta = w0alpha*(conj(Fk0).mul(Fk0)) + 0.5*((conj(Fkx).mul(Fkx)) + (conj(Fky).mul(Fky))) +
0.25*((conj(Fkxx).mul(Fkxx)) + (conj(Fkxy).mul(Fkxy)) + (conj(Fkyy).mul(Fkyy)));
	Mat temp;
	mulSpectrums(Fk0, Fk0, delta0, 0, true);//mulSpectrums
	mulSpectrums(Fkx, Fkx, delta1, 0, true);
	mulSpectrums(Fky, Fky, delta2, 0, true);;
	mulSpectrums(Fkxx, Fkxx, delta3, 0, true);
	mulSpectrums(Fkxy, Fkxy, delta4, 0, true);mulSpectrums(Fkyy, Fkyy, delta5, 0, true);
	delta12 = delta1 + delta2;delta345 = delta3 + delta4 + delta5;
	delta = w0alpha*(delta0 + 0.5*delta12 + 0.25*delta345);
	Mat in = Blurred;FFTy = fft2(in);
	FFTK = psf2otf(PSF, in.size());
	Mat Product1, Product2;
	mulSpectrums(FFTy, FFTK, Product1, 0, true);
	mulSpectrums(Product1, delta, Product2, 0, false);
	Mat Product3, Product4, Product5;
	mulSpectrums(FFTK, FFTK, Product3, 0, true);
	mulSpectrums(Product3, delta, Product4, 0, false);
	FFTLnom = Product2;
	FFTLdenom = Product4 + delta1 + delta2;
	divSpectrums(FFTLnom, FFTLdenom, FFTL, 0, false);
	L = ifft2(FFTL);
	return L;
}
```

- **Mat deBlur::deconv_fnmulComplexMats(Mat Blurred, Mat PSF, double w0alpha)**

```
Mat deBlur::deconv_fnmulComplexMats(Mat Blurred, Mat PSF, double w0alpha){
	Mat L, delta;
	Mat delta0, delta1, delta2, delta3, delta4, delta5, delta6, delta12, delta345;
	Mat Fk0, Fkx, Fky, Fkxx, Fkyy, Fkxy, FFTK, FFTy, FFTL, FFTLnom, FFTLdenom;
	Mat FFTLdenom1, FFTLdenom2, FFTLdenom3;
	Mat k0 = (Mat_<double>(1, 1) << 1);
	Mat kx = (Mat_<double>(1, 2) << 1, -1);
	Mat ky = (Mat_<double>(2, 1) << 1, -1);
	Mat kxx = (Mat_<double>(1, 3) << 1, -2, 1);
	Mat kyy = (Mat_<double>(3, 1) << 1, -2, 1);
	Mat kxy = (Mat_<double>(2, 2) << 1, -1, -1, 1);
	Fk0 = psf2otf(k0, Blurred.size());
	Fkx = psf2otf(kx, Blurred.size());
	Fky = psf2otf(ky, Blurred.size());
	Fkxx = psf2otf(kxx, Blurred.size());
	Fkyy = psf2otf(kyy, Blurred.size());
	Fkxy = psf2otf(kxy, Blurred.size());
	Mat temp;
	temp = conjMat(Fk0);
	mulComplexMats(Fk0, temp, delta0, 0, false);//mulSpectrums
	temp = conjMat(Fkx);
	mulComplexMats(Fkx, temp, delta1, 0, false);
	temp = conjMat(Fky);
	mulComplexMats(Fky, temp, delta2, 0, false);
	temp = conjMat(Fkxx);
	mulComplexMats(Fkxx, temp, delta3, 0, false);
	temp = conjMat(Fkxy);
	mulComplexMats(Fkxy, temp, delta4, 0, false);
	temp = conjMat(Fkyy);
	mulComplexMats(Fkyy, temp, delta5, 0, false);
	delta12 = delta1 + delta2;
	delta345 = delta3 + delta4 + delta5;
	delta = w0alpha*(delta0 + 0.5*delta12 + 0.25*delta345);
	FFTy = fft2(Blurred);
	FFTK = psf2otf(PSF, Blurred.size());
	Mat Product1, Product2;
	temp = conjMat(FFTK);
	mulComplexMats(FFTy, temp, Product1, 0, false);
	mulComplexMats(Product1, delta, Product2, 0, false);
	Mat Product3, Product4, Product5;
	temp = conjMat(FFTK);
	mulComplexMats(FFTK, temp, Product3, 0, false);
	mulComplexMats(Product3, delta, Product4, 0, false);
	FFTLnom = Product2;
	FFTLdenom = Product4 + delta1 + delta2;
	FFTL = divideComplexMats(FFTLnom, FFTLdenom);
	//ifft
	L = ifft2(FFTL);
	return L;
}
```

- **Mat deBlur::deconv(Mat &Blurred, Mat &PSF, double &w0alpha)(Part I)**

```
Mat deBlur::deconv(Mat &Blurred, Mat &PSF, double &w0alpha){
        Mat theResult, deblurred_image;
        Mat result;
        Mat finalimage;
        Size ks;
        ks = getPadSize(PSF);
        Mat ycrcb;
        int channelsNumber = Blurred.channels();
        vector<Mat> allTheChannels;
        vector<Mat> channels;
        vector<Mat> deconved;
        string mode = deconvmethod;
        if (mode == "RGB"){
                cv::split(Blurred, allTheChannels);
                for (int j = 0; j < channelsNumber; j++){
                        Mat in = allTheChannels.at(j);    //Blurred;
                        in = paddarray(in, ks, "replicate", "both");        //#!CAUTION!#//
                        theResult = deconv_fn(in, PSF, w0alpha);
                        Size d = theResult.size();
                        // Setup a rectangle to define your region of interest
                        int xmin = ks.height;
                        int ymin = ks.width;
                        int height = (d.height - ks.height - ks.height);
                        int width = (d.width - ks.width - ks.width);
                        cv::Rect myROI(xmin, ymin, width, height);
                        deblurred_image = theResult(myROI);
                        normalize(deblurred_image, deblurred_image, 0, 1, NORM_MINMAX, CV_64FC1);
                        //equalizeHistogramIntensity(deblurred_image, deblurred_image);
                        deconved.push_back(deblurred_image);
                }
                cv::merge(deconved, finalimage);
        }
        else{
```

- **cv::Mat deBlur::delta_kernel(int s)**

```
cv::Mat deBlur::delta_kernel(int s){
        int c;
        Mat out;
        if (s % 2 == 0){
                s = s + 1;
        }
        out = Mat::zeros(s, s, CV_64FC1); //CHECK TYPE
        c = (int)floor(s / 2);
        out.at<double>(c, c) = 1;
        return out;
}
```

- **Mat deBlur::deconv(Mat &Blurred, Mat &PSF, double &w0alpha)(Part II)**

```
        if (mode == "YCbCr"){
            if (channelsNumber == 1){
                Mat in = Blurred;    //Blurred;
                in = paddarray(in, ks, "replicate", "both");        //#!CAUTION!#//
                theResult = deconv_fn(in, PSF, w0alpha);
                Size d = theResult.size();
                // Setup a rectangle to define your region of interest
                int xmin = ks.height;
                int ymin = ks.width;
                int height = (d.height - ks.height - ks.height);
                int width = (d.width - ks.width - ks.width);
                cv::Rect myROI(xmin, ymin, width, height);
                deblurred_image = theResult(myROI);
                finalimage = deblurred_image;
            }
            else if (channelsNumber >= 3){
                Blurred.convertTo(Blurred, CV_32F);
                cv::cvtColor(Blurred, ycrcb, CV_BGR2YCrCb);
                ycrcb.convertTo(ycrcb, CV_64F);
                Blurred.convertTo(Blurred, CV_64F);
                cv::split(ycrcb, channels);
                Mat in = channels[0];    //Blurred;
                in = paddarray(in, ks, "replicate", "both");
                theResult = deconv_fn(in, PSF, w0alpha);
                Size d = theResult.size();
                // Setup a rectangle to define your region of interest
                int xmin = ks.height;
                int ymin = ks.width;
                int height = (d.height - ks.height - ks.height);
                int width = (d.width - ks.width - ks.width);
                cv::Rect myROI(xmin, ymin, width, height);
                deblurred_image = theResult(myROI);
                channels[0] = deblurred_image;
                cv::normalize(channels[0], channels[0], 0, 1, NORM_MINMAX, CV_64FC1);

                cv::merge(channels, ycrcb);
                ycrcb.convertTo(ycrcb, CV_32F);
                cv::cvtColor(ycrcb, result, CV_YCrCb2BGR);
                result.convertTo(result, CV_64F);
                finalimage = result;
            }
        }
    }
    return finalimage;
}
```

- **void deBlur::getImagePSF(Mat &OriginalBlurredScaled, Mat &OriginalBlurredMinimumScaleSingleChannel, Mat &PSF, int &iterScaleNum, bool &isFinal, Mat &OriginalBlurred, double &sigmaRange)**

```
void deBlur::getImagePSF(Mat &OriginalBlurredScaled, Mat &OriginalBlurredMinimumScaleSingleChannel, Mat &PSF, int
&iterScaleNum, bool &isFinal, Mat &OriginalBlurred, double &sigmaRange){
    Mat    Temp;
    Mat Prediction;
    Size TempSize;
    Temp = OriginalBlurredMinimumScaleSingleChannel;
    //Bilateral--------------------------------------------------------------------------
    double minValue, maxValue, delta;
    cv::minMaxLoc(Temp, &minValue, &maxValue);
    delta = maxValue - minValue;
    if (this->sigmaColor == 0){
        this->sigmaColor = 0.2*delta;
    }
    if (this->sigmaSpace == 0){
        this->sigmaSpace = min(Temp.cols, Temp.rows);
        this->sigmaSpace = this->sigmaSpace /5.0;    //#!CAUTION#
    }
    for (int thisinnerIter = 0; thisinnerIter < iterScaleNum; thisinnerIter++){
        Temp.convertTo(Temp, CV_32F);
        bilateralFilter(Temp, Prediction, 5, sigmaColor, sigmaSpace);
        bilateralFilter(Temp, Prediction, 5, (sigmaColor/2.0), (sigmaSpace/2.0));
        bilateralFilter(Temp, Prediction, 5, (sigmaColor/4.0), (sigmaSpace/4.0));
        Prediction.convertTo(Prediction, CV_64F);
        cv::normalize(Prediction, Prediction, 0, 1, NORM_MINMAX, CV_64FC1);
        Prediction = shock_filter(Prediction, this->shockFilterIter, this->shockFilterdt, 1);
        cv::normalize(Prediction, Prediction, 0, 1, NORM_MINMAX, CV_64FC1);
        equalizeHistogramIntensity(Prediction, Prediction);
        this->OriginalBlurredScaledSingleChannel = r2g(OriginalBlurredScaled);
        this->PSFResult = estK(Prediction, this->OriginalBlurredScaledSingleChannel, PSF, 10);
        if ((thisinnerIter != iterScaleNum) || (!isFinal)){
            this->deblurredImage = deconv(this->OriginalBlurredScaledSingleChannel, this->PSFResult, this->alpha);
            cv::normalize(this->deblurredImage, this->deblurredImage, 0, 1, NORM_MINMAX, CV_64FC1);
            Mat Temp = this->deblurredImage;
        }
        //FINAL DECONVOLUTION
        if ((thisinnerIter == iterScaleNum - 1) && (this->isFinal)){
            this->deblurredImage = deconv(OriginalBlurred, this->PSFResult, this->alpha);

            cv::normalize(this->deblurredImage, this->deblurredImage, 0, 1, NORM_MINMAX, CV_64FC3);
        }
    }
    return;
}
```

- **Mat conjMat(Mat src)**

```
Mat conjMat(Mat src){
	Mat output = src;
	Mat theChannels[2];
	Mat * image;
	image = new Mat;
	cv::split(src, theChannels);
	if (src.channels() == 2)
	{
		theChannels[1] = -theChannels[1];
		cv::merge(theChannels, 2, output);
	}
	else
	{
		exit(19);
	}

	return output;
}
```

- **void deBlur::analyzeKernel(void)**

```
void deBlur::analyzeKernel(void){
	double elem = this->initsize;
	int i = 1;
	bool flag = true;
	while (flag){
		this->kernelArray.push_back((int)elem);
		elem = floor(elem / 2);
		if (elem <= 5){
			if ((int)elem % 2 == 0){
				elem++;
			}
			this->kernelArray.push_back((int)elem);
			flag = false;
		}
		if ((int)elem % 2 == 0){
			elem++;
		}
	}
	return;
}
```

- **Mat blinddeblurmap(Mat _inputImage, int _initsize, int _iterScaleNum, double _alpha , double _freqCut , int _belta, double _sigmaSpace, double _sigmaColor, int _shockFilterIter, double _shockFilterdt)**

```
Mat blinddeblurmap(Mat _inputImage, int _initsize, int _iterScaleNum, double _alpha , double _freqCut , int _belta,
double _sigmaSpace, double _sigmaColor, int _shockFilterIter, double _shockFilterdt )
{
	clock_t begin = clock();
	deBlur * theProcess;
	Mat result;
	theProcess = new deBlur();
	theProcess->initsize = _initsize;
	theProcess->iterScaleNum = _iterScaleNum;
	theProcess->alpha = _alpha;
	theProcess->freqCut = _freqCut;
	theProcess->belta = _belta;
	theProcess->sigmaSpace = _sigmaSpace;
	theProcess->sigmaColor = _sigmaColor;
	theProcess->shockFilterIter = _shockFilterIter;
	theProcess->shockFilterdt = _shockFilterdt;
	theProcess->img = _inputImage;
	theProcess->readImage();
	theProcess->analyzeKernel();
	theProcess->iterativeDeblurring();
	clock_t end = clock();
	double elapsed_secs = double(end - begin) / CLOCKS_PER_SEC;
	std::cout << elapsed_secs << std::endl;
	result = theProcess->deblurredImage;
	Mat PSFOUT;
	cv::normalize(theProcess->PSFResult, PSFOUT, 0, 1, NORM_MINMAX, CV_64F);
	resize(PSFOUT, PSFOUT, cv::Size(150, 150));
	v2 = rand() % 100 + 1;
	//imshow("Blur Kernel " + std::to_string(v2), PSFOUT);
	//imshow("Deblurred image " + std::to_string(v2), theProcess->deblurredImage);
	//imshow("Original Image " + std::to_string(v2), theProcess->img);

	imwrite("../results/"+std::to_string(v2)+"_kernel.jpg", 255*PSFOUT);

	delete theProcess;
	return result;
}
```

- **Mat equalizeIntensity(const Mat&)**

```
Mat equalizeIntensity(const Mat& inputImage)
{
	Mat ycrcb;
	Mat inter;
	if (inputImage.channels() >= 3)
	{
		cv::cvtColor(inputImage, ycrcb, CV_BGR2YCrCb);
		vector<Mat> channels;
		cv::split(ycrcb, channels);
		equalizeHist(channels[0], inter);
		channels[0] = 0.5*channels[0] + 0.5*inter;
		//equalizeHist(channels[1], channels[1]);
		//equalizeHist(channels[2], channels[2]);
		Mat result;
		cv::merge(channels, ycrcb);
		cv::cvtColor(ycrcb, result, CV_YCrCb2BGR);
		return result;
	}
	else {
		if (inputImage.channels() == 1){
			equalizeHist(inputImage, ycrcb);
			return ycrcb;
		}
	}
	return Mat();
}

void equalizeHistogramIntensity(Mat &src, Mat&dst)
{
	int type = dst.type();
	dst = 255.0 * src;
	dst.convertTo(dst, CV_8UC1);
	equalizeHist(dst,dst);
	dst.convertTo(dst, type);
	dst = dst/255.0;
	return;
}
```

- **Mat deblur(Mat)**

```
Mat    deblur(Mat image)
{
        //Set Params
        Mat _input = image;
        Mat _out;
        int _initsize = (int)(std::min(_input.size().height, _input.size().width) / 5);
        _initsize = 21;
        int _iterScaleNum = 10;
        double _alpha = 1.2;
        double _freqCut =10;
        int _belta = 2;
        double _sigmaSpace = 0;
        double _sigmaColor = 0;
        int _shockFilterIter = 2;
        double _shockFilterdt = 0.5;
        kernelmulkernelfactor = 0.5;
        maxkernelout = true;
        kernelmulkernel = true;
        equalize = false;
        kernelCrop = true;
        deconvmethod = "RGB";
        deconvmethod = "YCbCr";
        //Start
        _input.convertTo(_input, CV_32F);
        //medianBlur(_input, _input, 3);
        _input.convertTo(_input, CV_64F);
        _out = blinddeblurmap(_input, _initsize, _iterScaleNum, _alpha, _freqCut, _belta, _sigmaSpace, _sigmaColor,
_shockFilterIter, _shockFilterdt);
        _out = 255 * _out;
        _out.convertTo(_out, CV_8UC3);
        if (equalize){ _out = equalizeIntensity(_out); }
        imwrite("../results/" + std::to_string(v2) + "_deblurred.jpg",    _out);
        imwrite("../results/" + std::to_string(v2) + "_blurry.jpg", 255 * _input);
        return    _out;
}
```

- **deblur.h**

```
class deBlur{
public:
	//GENERAL PURPOSE
	std::string filename;Mat color, img;Mat grayScaleImage;
	//SHOCK FILTER
	Mat shockFiltered;
	//BILATERAL FILTER
	Mat bilateralResult;double sigmaColor,sigmaSpace,sigmaRange,double shockFilterdt,
	//DECONVOLUTION RELATED
	double belta;
	double alpha;
	double freqCut;
	//KERNEL RELATED
	int initsize;
	//MULTISCALE APPROACH AND ITERATIVE APPROACH RELATED
	int iterScaleNum,scales,shockFilterIter;
	double scalesDenominator, scalesFactor;
	std::vector<int> kernelArray;bool isFinal;
	Mat OriginalBlurredScaledMinSingleChannel;Mat OriginalBlurredScaledSingleChannel;Mat OriginalBlurred;
	Mat OriginalBlurredScaled;Mat InitialPSF;
	int CurrentIterationFactor, sizef;double riterDenominator, riterFactor;
	//OUTPUT MATRICES
	Mat deblurredImage;Mat PSFResult;
	//INITIALIZATION FUNCTIONS
	void analyzeKernel(void);void resizeBlurredToMinimumScale(void);
	//IO
	void readImage(void);void readImage(string);void readImage(Mat theImage);
	//ITERATIVE DEBLURRING
	void iterativeDeblurring(void);
	void getImagePSF(Mat &OriginalBlurredScaled, Mat &OriginalBlurredMinimumScaleSingleChannel, Mat &PSF, int
&iterScaleNum, bool &isFinal, Mat &OriginalBlurred, double &sigmaRange);
	//FILTERING
	Mat shock_filter(Mat IO, int iter, double dt, double h);
	//FFT RELATED-----------------------------------------
	Mat fft2(Mat input);Mat ifft2(Mat input);
	//PSF to OTF & reverse------------
	Mat psf2otf(Mat inputPSF, Size finalSize);Mat otf2psf(Mat inputOTF, Size outSize);
	//KERNEL RELATED------------------------------------
	Mat estK(Mat Prediction, Mat OriginalBlurredScaledSingleChannel, Mat PSF, int numberOfIterations);
	Mat getKMulSpectrumSupport(Mat PredictionPadded, Mat BlurredPadded, Mat PSF, double belta);
	Mat delta_kernel(int s);
	//PADDING RELATED-------------------------------------------------
	Mat paddarray(Mat input, Size padding, std::string method, std::string direction);Size getPadSize(Mat f);
	//GRAYSCALE RELATED---------------------------------------------------
	Mat r2g(Mat input);
	//DECONVOLUTION RELATED
	Mat deconv(Mat &Blurred, Mat &PSF, double &w0alpha);
	Mat deconv_fnmulComplexMats(Mat Blurred, Mat PSF, double w0alpha);
	Mat deconv_fn(Mat &Blurred, Mat &PSF, double &w0alpha);
};
```

C. ANDROID USER INTERFACE

a. Main/Java

- **DeblurActivity.java**

```
package org.opencv.samples.deblur;
import android.app.ProgressDialog;
import android.content.Intent;
import android.database.Cursor;
import android.graphics.Bitmap;
import android.graphics.drawable.BitmapDrawable;
import android.graphics.drawable.Drawable;
import android.net.Uri;
import android.os.AsyncTask;
import android.os.Bundle;
import android.os.Environment;
import android.os.Parcelable;
import android.provider.MediaStore;
import android.support.design.widget.FloatingActionButton;
import android.support.design.widget.NavigationView;
import android.support.v4.view.GravityCompat;
import android.support.v4.widget.DrawerLayout;
import android.support.v7.app.ActionBarDrawerToggle;
import android.support.v7.app.AppCompatActivity;
import android.support.v7.widget.Toolbar;
import android.util.Log;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;
import android.widget.ImageView;
import android.widget.Toast;
import org.opencv.android.BaseLoaderCallback;
import org.opencv.android.CameraBridgeViewBase;
import org.opencv.android.LoaderCallbackInterface;
import org.opencv.android.OpenCVLoader;
import org.opencv.android.Utils;
import org.opencv.core.CvType;
import org.opencv.core.Mat;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Random;
```

```
public class DeblurActivity extends AppCompatActivity implements NavigationView.OnNavigationItemSelectedListener {
    private static final String TAG = "OCVSample::Activity";
    public Bitmap mBmp;
    private Mat input, output;
    private Drawable mImgDrawable;
    private static final int VIEW_MODE_RGBA = 0;
    private static final int VIEW_MODE_GRAY = 1;
    private static final int VIEW_MODE_CANNY = 2;
    private static final int VIEW_MODE_FEATURES = 5;
    private int mViewMode;
    private Mat mRgba;
    private Mat mIntermediateMat;
    private Mat mGray;
    private Mat mInput;
    private Mat mOutput;
    private MenuItem mItemPreviewRGBA;
    private MenuItem mItemPreviewGray;
    private MenuItem mItemPreviewCanny;
    private MenuItem mItemPreviewFeatures;
    private CameraBridgeViewBase mOpenCvCameraView;
    private static final int SELECT_SINGLE_PICTURE = 101;
    private static final int SELECT_MULTIPLE_PICTURE = 201;
    public static final String IMAGE_TYPE = "image/*";
    private ImageView selectedImagePreview;
    private ProgressDialog progressDialog;
```

```
private BaseLoaderCallback mLoaderCallback = new BaseLoaderCallback(this) {
        @Override
        public void onManagerConnected(int status) {
            switch (status) {
                case LoaderCallbackInterface.SUCCESS: {
                    Log.i(TAG, "OpenCV loaded successfully");
                    // Load native library after(!) OpenCV initialization
                    System.loadLibrary("mixed_sample");
                    //mOpenCvCameraView.enableView();
                }
                break;
                default: {
                    super.onManagerConnected(status);
                }
                break;
            }}};
@Override
protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        start();
}
```

```
    public void start() {
                setContentView(R.layout.activity_main);
                //OpenCVLoader.initAsync(OpenCVLoader.OPENCV_VERSION_2_4_11, this, mLoaderCallback);
                Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
                setSupportActionBar(toolbar);
                FloatingActionButton fab = (FloatingActionButton) findViewById(R.id.fab);
                fab.setOnClickListener(new View.OnClickListener() {
                        @Override
                        public void onClick(View view) {
                                //Snackbar.make(view, "Replace with your own action", Snackbar.LENGTH_LONG).setAction("Action",
null).show();
                                new LoadViewTask().execute();
                        }
                });
                DrawerLayout drawer = (DrawerLayout) findViewById(R.id.drawer_layout);
                ActionBarDrawerToggle toggle = new ActionBarDrawerToggle(this, drawer, toolbar,
R.string.navigation_drawer_open, R.string.navigation_drawer_close);
                drawer.setDrawerListener(toggle);
                toggle.syncState();
                NavigationView navigationView = (NavigationView) findViewById(R.id.nav_view);
                navigationView.setNavigationItemSelectedListener(this);
                // no need to cast to button view here since we can add a listener to any view, this
                // is the single image selection
                findViewById(R.id.btn_pick_single_image).setOnClickListener(new View.OnClickListener() {
                        public void onClick(View arg0) {
                                // in onCreate or any event where your want the user to
                                // select a file
                                Intent intent = new Intent();
                                intent.setType(IMAGE_TYPE);
                                intent.setAction(Intent.ACTION_GET_CONTENT);
                                startActivityForResult(Intent.createChooser(intent,
                                                getString(R.string.select_picture)), SELECT_SINGLE_PICTURE);
                        }
                });
                selectedImagePreview = (ImageView) findViewById(R.id.image_preview);
        }
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }
```

```
    public void doit() {
        mInput = new Mat(mBmp.getHeight(), mBmp.getWidth(), CvType.CV_8UC3);
        mOutput = new Mat(mBmp.getHeight(), mBmp.getWidth(), CvType.CV_8UC3);
        Utils.bitmapToMat(mBmp, mInput);
        Deblur(mInput.getNativeObjAddr(), mOutput.getNativeObjAddr());
        Utils.matToBitmap(mOutput, mBmp);
        SaveImage(mBmp);
    }
```

```
    @Override
    public void onResume() {
        super.onResume();
        OpenCVLoader.initAsync(OpenCVLoader.OPENCV_VERSION_3_0_0, this, mLoaderCallback);
    }
```

```
@Override
    public void onBackPressed() {
        DrawerLayout drawer = (DrawerLayout) findViewById(R.id.drawer_layout);
        if (drawer.isDrawerOpen(GravityCompat.START)) {
            drawer.closeDrawer(GravityCompat.START);
        } else {
            super.onBackPressed();
        }
    }
```

```
@Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle action bar item clicks here. The action bar will
        // automatically handle clicks on the Home/Up button, so long
        // as you specify a parent activity in AndroidManifest.xml.
        int id = item.getItemId();
        return super.onOptionsItemSelected(item);
    }
    @SuppressWarnings("StatementWithEmptyBody")
    @Override
    public boolean onNavigationItemSelected(MenuItem item) {
        // Handle navigation view item clicks here.
        int id = item.getItemId();
        DrawerLayout drawer = (DrawerLayout) findViewById(R.id.drawer_layout);
        drawer.closeDrawer(GravityCompat.START);
        return true;
    }
```

```
    @SuppressWarnings("StatementWithEmptyBody")
    @Override
    public boolean onNavigationItemSelected(MenuItem item) {
        // Handle navigation view item clicks here.
        int id = item.getItemId();
        DrawerLayout drawer = (DrawerLayout) findViewById(R.id.drawer_layout);
        drawer.closeDrawer(GravityCompat.START);
        return true;
    }
```

```
/**
     * helper to retrieve the path of an image URI
     */
    public String getPath(Uri uri) {
        // just some safety built in
        if (uri == null) {
            // perform some logging or show user feedback
            Toast.makeText(getApplicationContext(), R.string.msg_failed_to_get_picture,
Toast.LENGTH_LONG).show();
            Log.d(DeblurActivity.class.getSimpleName(), "Failed to parse image path from image URI " + uri);
            return null;
        }
        // try to retrieve the image from the media store first
        // this will only work for images selected from gallery
        String[] projection = {MediaStore.Images.Media.DATA};
        Cursor cursor = managedQuery(uri, projection, null, null, null);
        if (cursor != null) {
            int column_index = cursor
                    .getColumnIndexOrThrow(MediaStore.Images.Media.DATA);
            cursor.moveToFirst();
            return cursor.getString(column_index);
        }
        // this is our fallback here, thanks to the answer from @mad indicating this is needed for
        // working code based on images selected using other file managers
        return uri.getPath();
    }
```

```
public void onActivityResult(int requestCode, int resultCode, Intent data) {
		Bitmap bm;
		if (resultCode == RESULT_OK) {
			if (requestCode == SELECT_SINGLE_PICTURE) {
				Uri selectedImageUri = data.getData();
				try {
					selectedImagePreview.setImageBitmap(new UserPicture(selectedImageUri,
getContentResolver()).getBitmap());
					mImgDrawable = selectedImagePreview.getDrawable();
					mBmp = ((BitmapDrawable) mImgDrawable).getBitmap();
					if (mBmp != null) {
					}
				} catch (IOException e) {
					Log.e(DeblurActivity.class.getSimpleName(), "Failed to load image", e);
				}
			} else if (requestCode == SELECT_MULTIPLE_PICTURE) {
				//And in the Result handling check for that parameter:
				if (Intent.ACTION_SEND_MULTIPLE.equals(data.getAction())
						&& data.hasExtra(Intent.EXTRA_STREAM)) {
					// retrieve a collection of selected images
					ArrayList<Parcelable> list = data.getParcelableArrayListExtra(Intent.EXTRA_STREAM);
					// iterate over these images
					if (list != null) {
						for (Parcelable parcel : list) {
							Uri uri = (Uri) parcel;
							// handle the images one by one here
						}
					}
					// for now just show the last picture
					if (!list.isEmpty()) {
						Uri imageUri = (Uri) list.get(list.size() - 1);
						try {
							selectedImagePreview.setImageBitmap(new UserPicture(imageUri,
getContentResolver()).getBitmap());
						} catch (IOException e) {
							Log.e(DeblurActivity.class.getSimpleName(), "Failed to load image", e);
						}
					}
				}
			}
		} else {
			// report failure
			Toast.makeText(getApplicationContext(), R.string.msg_failed_to_get_intent_data, Toast.LENGTH_LONG).show();
			Log.d(DeblurActivity.class.getSimpleName(), "Failed to get intent data, result code is " + resultCode);
		}
	}
```

```
private void SaveImage(Bitmap finalBitmap) {
        String root = Environment.getExternalStorageDirectory().toString();
        File myDir = new File(root + "/DCIM/Results");
        myDir.mkdirs();
        Random generator = new Random();
        int n = 10000;
        n = generator.nextInt(n);
        String fname = "Image-" + n + ".jpg";
        File file = new File(myDir, fname);
        if (file.exists()) file.delete();
        try {
            FileOutputStream out = new FileOutputStream(file);
            finalBitmap.compress(Bitmap.CompressFormat.JPEG, 90, out);
            out.flush();
            out.close();
        } catch (Exception e) {
            e.printStackTrace();
```

```
public static Bitmap scaleDown(Bitmap realImage, float maxImageSize,
                                    boolean filter) {
        float ratio = Math.min(
                (float) maxImageSize / realImage.getWidth(),
                (float) maxImageSize / realImage.getHeight());
        int width = Math.round((float) ratio * realImage.getWidth());
        int height = Math.round((float) ratio * realImage.getHeight());
        Bitmap newBitmap = Bitmap.createScaledBitmap(realImage, width,
                height, filter);
        return newBitmap;
    }
```

```
public native void FindFeatures(long matAddrGr, long matAddrRgba);
    public native void Deblur(long matAddrIn, long matAddrOut);
```

```
//To use the AsyncTask, it must be subclassed
        private class LoadViewTask extends AsyncTask<Void, Integer, Void> {
                //Before running code in the separate thread
                @Override
                protected void onPreExecute() {
                        //Create a new progress dialog
                        progressDialog = new ProgressDialog(DeblurActivity.this);
                        //Set the progress dialog to display a horizontal progress bar
                        progressDialog.setProgressStyle(ProgressDialog.STYLE_HORIZONTAL);
                        //Set the dialog title to 'Loading...'
                        progressDialog.setTitle("Processing...");
                        //Set the dialog message to 'Loading application View, please wait...'
                        progressDialog.setMessage("Deblurring, please wait...");
                        //This dialog can't be canceled by pressing the back key
                        progressDialog.setCancelable(false);
                        //This dialog isn't indeterminate
                        progressDialog.setIndeterminate(false);
                        //The maximum number of items is 100
                        progressDialog.setMax(100);
                        //Set the current progress to zero
                        progressDialog.setProgress(0);
                        //Display the progress dialog
                        progressDialog.show();
                }
```

```
//The code to be executed in a background thread.
@Override
protected Void doInBackground(Void... params) {
     /* This is just a code that delays the thread execution 4 times,
      * during 850 milliseconds and updates the current progress. This
      * is where the code that is going to be executed on a background
      * thread must be placed.
      */
    doit();
    try {
        //Get the current thread's token
        synchronized (this) {
            //Initialize an integer (that will act as a counter) to zero
            int counter = 0;
            //While the counter is smaller than four
            while (counter <= 4) {
                //Wait 850 milliseconds
                this.wait(850);
                //Increment the counter
                counter++;
                //Set the current progress.
                //This value is going to be passed to the onProgressUpdate() method.
                publishProgress(counter * 25);
            }
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return null;
}
```

```
//Update the progress
@Override
protected void onProgressUpdate(Integer... values) {
    //set the current progress of the progress dialog
    progressDialog.setProgress(values[0]);
}
```

```
//after executing the code in the thread
        @Override
        protected void onPostExecute(Void result) {

            progressDialog.dismiss();
            start();
            selectedImagePreview.setImageBitmap(mBmp);
            //setContentView(R.layout.activity_main);

        }
    }
}
```

b. Jni/jni_part.cpp

- **jni_part.cpp**

```
#include "deblur.h"
using namespace std;
using namespace cv;
Mat deblur(Mat image);
extern "C" {
JNIEXPORT void JNICALL Java_org_opencv_samples_deblur_DeblurActivity_Deblur(JNIEnv*, jobject, jlong addrIn, jlong
addrOut);
JNIEXPORT void JNICALL Java_org_opencv_samples_deblur_DeblurActivity_Deblur(JNIEnv*, jobject,jlong addrIn, jlong
addrOut)
{
    Mat& mInput    = *(Mat*)addrIn;
    Mat& mOutput = *(Mat*)addrOut;
    mOutput=deblur(mInput);
}
}
```

c. Resources/layout

- **Activity_mail.xml**

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools" android:layout_width="match_parent"
    android:layout_height="match_parent" android:fitsSystemWindows="true"
    tools:context=".MainActivity">

    <android.support.design.widget.AppBarLayout android:layout_height="wrap_content"
        android:layout_width="match_parent" android:theme="@style/AppTheme.AppBarOverlay">

        <android.support.v7.widget.Toolbar android:id="@+id/toolbar"
            android:layout_width="match_parent"
android:layout_height="?attr/actionBarSize"
            android:background="?attr/colorPrimary"
app:popupTheme="@style/AppTheme.PopupOverlay" />

    </android.support.design.widget.AppBarLayout>

    <include layout="@layout/content_main" />

    <android.support.design.widget.FloatingActionButton android:id="@+id/fab"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:layout_gravity="bottom|end" android:layout_margin="@dimen/fab_margin"
        android:src="@android:drawable/ic_menu_crop" />

</android.support.design.widget.CoordinatorLayout>
```

- **App_bar_main.xml**

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.widget.DrawerLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools" android:id="@+id/drawer_layout"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:fitsSystemWindows="true" tools:openDrawer="start">

    <include layout="@layout/app_bar_main" android:layout_width="match_parent"
        android:layout_height="match_parent" />

    <android.support.design.widget.NavigationView android:id="@+id/nav_view"
        android:layout_width="wrap_content" android:layout_height="match_parent"
        android:layout_gravity="start" android:fitsSystemWindows="true"
        app:headerLayout="@layout/nav_header_main" app:menu="@menu/activity_main_drawer"
/>

</android.support.v4.widget.DrawerLayout>
```

- **Content_main.xml**

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto" android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:paddingTop="16dp"
    android:paddingBottom="16dp"
    app:layout_behavior="@string/appbar_scrolling_view_behavior" tools:showIn="@layout/app_bar_main"
    tools:context=".MainActivity">
    <TextView
        android:id="@+id/txt_hello"
        android:text="@string/txt_info"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:textSize="24sp"
        android:padding="8dp"/>
    <Button
        android:id="@+id/btn_pick_single_image"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_below="@+id/txt_hello"
        android:text="@string/lbl_pick_single_image"
        />
    <ImageView
        android:id="@+id/image_preview"
        android:layout_width="match_parent"
        android:layout_height="250dp"
        android:layout_below="@+id/btn_pick_single_image"
        android:gravity="center"
        />
</RelativeLayout>
```

- **nav_bar.xml**

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="@dimen/nav_header_height"
    android:background="@drawable/side_nav_bar"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:theme="@style/ThemeOverlay.AppCompat.Dark" android:orientation="vertical"
    android:gravity="bottom">
    <ImageView android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:paddingTop="@dimen/nav_header_vertical_spacing"
        android:src="@android:drawable/sym_def_app_icon" android:id="@+id/imageView" />

    <TextView android:layout_width="match_parent" android:layout_height="wrap_content"
        android:paddingTop="@dimen/nav_header_vertical_spacing" android:text="Android Studio"
        android:textAppearance="@style/TextAppearance.AppCompat.Body1" />

    <TextView android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="android.studio@android.com" android:id="@+id/textView" />

</LinearLayout>
```