Image Deblurring using Radial Basis functions for Interpolating Weights of Spatially Variant Blur

By

Mai Phuong Pham Huynh

James G. Nagy, Ph.D.
Advisor

Department of Mathematics

2022

Image Deblurring using Radial Basis functions for Interpolating Weights of Spatially Variant Blur

By

Mai Phuong Pham Huynh

James G. Nagy, Ph.D.
Advisor

An abstract of
a thesis submitted to the Faculty of Emory College of Arts and Sciences of
Emory University in partial fulfillment
of the requirements of the degree of
Bachelor of Science with Honors

Department of Mathematics

2022

Abstract

Image Deblurring using Radial Basis functions for Interpolating Weights of Spatially Variant Blur
By Mai Phuong Pham Huynh

Even though technology in applications from microscopy to astronomy has been improving significantly and producing much better pictures, the image deblurring problem is still relevant as a post-processing technique to improve image quality. A substantial amount of work has been done for spatially invariant blurs, but relatively little for the spatially variant case. One approach is to assume the blur is locally approximately spatially invariant, and use interpolation of the local blurring operators to obtain a global approximation of the spatially variant blur. By using different types of functions, piecewise constant, piecewise linear, and radial basis functions (RBF) for the interpolation, we found that piecewise linear functions and RBF outperform piecewise constant functions. We then conduct further experiments concerning the number of regions the image should be partitioned into and the type of interpolated functions used to represent the variation of the spatially variant blur in the $x-$ and $y-$directions.

Image Deblurring using Radial Basis functions for Interpolating Weights of Spatially Variant Blur

By

Mai Phuong Pham Huynh

James G. Nagy, Ph.D.
Advisor

A thesis submitted to the Faculty of Emory College of Arts and Sciences
of Emory University in partial fulfillment
of the requirements of the degree of
Bachelor of Science with Honors

Department of Mathematics

2022

Acknowledgments

I would like to express my appreciation to my thesis advisor, Dr. James G. Nagy, for his support since the beginning of my honors research work. I would also like to acknowledge Dr. Jeremy Jacobson and Dr. Lars Ruthotto for being on my honors thesis committee. Lastly, I would also like to express my gratitude to my family and dearest friends, who have been my firmest mental support during this journey.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the recent advancements in photography technology, images have become more realistic representations of captured objects. For example, the current iPhone 13 Pro has a telephoto feature [1], which means that its lens has a focal length of 60mm or longer. This feature could only be found in professional camera gear 5 year ago. However, despite the increasing accuracy of capturing technologies, it is almost impossible to produce a completely truthful image of objects due to external factors such as light sources, out-of-focus lens, etc. These problems can cause "spill over" effects from a pixel to its neighboring ones. Image deblurring algorithms using mathematical models seek to recover the original image that has been affected by such external factors. Nevertheless, this is not always a simple task as the blurring information is usually hidden. In this chapter, we will go through the basic mathematical set-up of the image deblurring problem.

1.1 Numerating images

In order to apply a mathematical model to an image deblurring problem, it is important to find a way to represent an image using arrays of numbers.

A small image has $256^2 = 65536$ pixels while a high-resolution one can have from

5 to 10 million pixels, each of which represents the color of a small rectangular or square fragment of the image (i.e, a pixel). In this thesis, since we mostly work with greyscale images, we only need a single matrix to represent our image. For example, consider the following matrix:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 9 & 0 & 9 & 0 & 7 & 0 \\ 0 & 2 & 0 & 0 & 0 & 9 & 0 & 9 & 0 & 7 & 0 \\ 0 & 2 & 0 & 0 & 0 & 9 & 9 & 9 & 0 & 7 & 0 \\ 0 & 2 & 0 & 0 & 0 & 9 & 0 & 9 & 0 & 0 & 0 \\ 0 & 2 & 2 & 2 & 0 & 9 & 0 & 9 & 0 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Using the greyscale color map, the above matrix will produce the following figure:

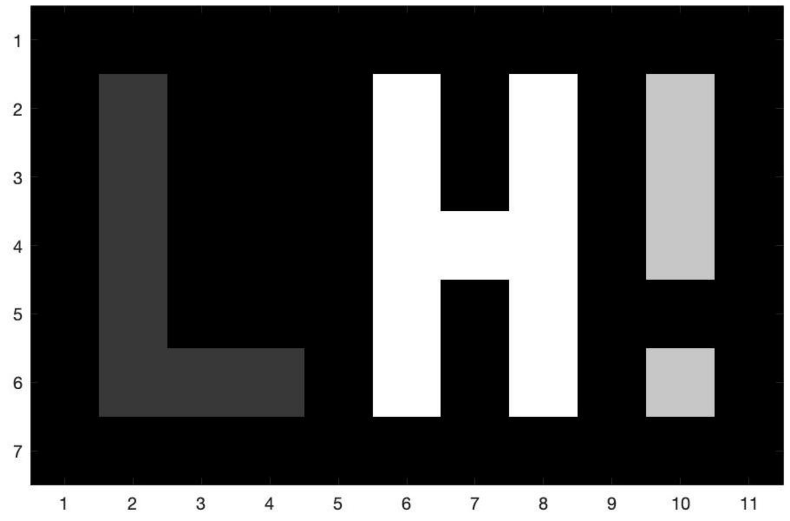


Figure 1.1: A greyscale image created from arrays of numbers.

The largest (9) and smallest (0) numbers in the array are assigned to white and black, respectively. Other values are assigned correspondingly to different shades of grey.

The mathematical representation of a colored image is quite similar; however, we need 3 matrices for an RGB image, each of which contains information on the shades of red, green, and blue colors. It is convenient to transform a colored image to a greyscale image using the `MATLAB` function `rgb2gray`. Further details on the mathematical construction of colored images and `MATLAB` implementation can be found in [9].

1.2 $\mathbf{Ax} = \mathbf{b}$ problem

In this thesis, we want to deblur a greyscale image of size $m \times n$. As discussed previously, this image has mn pixels, each of which represents the light intensity information of the captured scenery or object. The image can be written in the matrix form $\mathbf{B}$ of size $m \times n$ or with the vector representation of size mn: $\mathbf{b} = \text{vec}(\mathbf{B})$. For later reference, we call this the "blurred image." Our main goal is to find the "true image" $\mathbf{X}$ of the same size as $\mathbf{B}$. Similar to $\mathbf{B}$, $\mathbf{X}$ can also be written as a vector $\mathbf{x} = \text{vec}(\mathbf{X})$.

In a linear model, there exists a large matrix $\mathbf{A}$ of size $N \times N$, where $N = mn$, containing the blurring information of the problem. In a linear model, the problem can be set up as

$$\mathbf{Ax} = \mathbf{b}$$

Intuitively, matrix $\mathbf{A}$ represents the process of transforming a "true image" $\mathbf{x}$ to a blurred image $\mathbf{b}$. However, in reality, the blurring information is usually unknown to us, the "photo editor," causing difficulties in constructing the matrix $\mathbf{A}$. Another problem is that we usually do not know the image vector $\mathbf{b}$ exactly, but instead we can only know

$$\mathbf{Ax} + \mathbf{e} = \mathbf{b}$$

where $\mathbf{e}$ is unknown noise, or other errors in the measured data.

The detail on how to construct the matrix $\mathbf{A}$ is discussed further in Chapter 3.

Chapter 2

The Construction of the Blurring Function

As aforementioned, the accuracy of our algorithm heavily depends on how accurate our blurring function is; the construction of which is difficult, as usually the blurring information is not provided to us. In the context of a linear model, we can consider the standard linear image formation model [3]:

$$b(g,h) = \int_{\Re^2} a(g,h,s,t)x(s,t)dsdt \tag{2.1}$$

where x is the true image and b is the observed image (i.e the blurred image). The kernel function a is the blurring operation and is called the *point spread function* (PSF). Using discretization techniques, such as quadrature rules to approximate the integration, equation (2.1) can be written in the form $\mathbf{b} = \mathbf{A}\mathbf{x}$, or equivalently $\mathbf{A}\mathbf{x} = \mathbf{b}$. This will be discussed in more detail in the next section. In this chapter, we focus on the construction of an image deblurring problem, as well as how to solve it efficiently.

2.1 The blurring matrix A

In the case of simple spatially invariant blurs, the kernel has the form $a(g-s, h-t)$, and the matrix $\mathbf{A}$ has a Toeplitz structure. For general spatially variant blurs, matrix $\mathbf{A}$ does not have such a nice structure. However, as was studied in previous literature if the blur is assumed locally spatially variant, the matrix $\mathbf{A}$ can written as [5][6][8]

$$\mathbf{A} = \sum_{i=1}^{p} \mathbf{A}_i \mathbf{D}_i \tag{2.2}$$

where $\mathbf{A}_i$ are associated with a local spatially invariant blur (and hence, have a Toeplitz structure), p is the number of image regions, and $\mathbf{D}_i$s are interpolation matrices, whose construction will be discussed in Section 2.6 and Chapter 3.

The main focus of this section is to construct such a matrix $\mathbf{A}$ by using the composite Midpoint rule on equation (2.1), using a specific example for the kernel a.

Using the composite Midpoint rule with $N \times N$ equally spaced points, the integration in equation (2.1) can be approximated by

$$b(x,y) \approx \frac{1}{N^2} \sum_{i=1}^{N} \sum_{j=1}^{N} a(g, h, \bar{s}_i, \bar{t}_j) x(\bar{s}_i, \bar{t}_j) \tag{2.3}$$

where

$$\bar{s}_i = \frac{2i-1}{2N} \qquad \bar{t}_j = \frac{2j-1}{2N}\,.$$

Note that here, we use the integration in the domain $[0,1] \times [0,1]$ as we hypothesize that the image construction process captures all light.

Now, if we choose the PSF function, a, to be Gaussian function, we can partition the function a into two functions a_1 and a_2 as follows:

$$\begin{aligned} a(g,h,s,t) &= e^{-\rho_g(g-s)^2-\rho_h(h-t)^2} \\ &= e^{-\rho_g(g-s)^2}e^{-\rho_h(h-t)^2} \\ &= a_1(g,s)a_2(h,t)\,. \end{aligned}$$

Then, equation (2.3) becomes

$$\begin{aligned} b(g,h) &= \frac{1}{N^2}\sum_{i=1}^{N}\sum_{j=1}^{N} a_1(g,\bar{s}_i)a_2(h,\bar{t}_j)x(\bar{s}_i,\bar{t}_j) \\ &= \frac{1}{N^2}\begin{bmatrix} a_2(h,\bar{t}_1) & \cdots & a_2(h,\bar{t}_N)\end{bmatrix}\begin{bmatrix} x(\bar{s}_i,\bar{t}_1) & \cdots & x(\bar{s}_N,\bar{t}_1) \\ \vdots & \ddots & \vdots \\ x(\bar{s}_1,\bar{t}_N) & \cdots & x(\bar{s}_N,\bar{t}_N)\end{bmatrix}\begin{bmatrix} a_1(g,\bar{s}_1) & \cdots & a_1(g,\bar{s}_N)\end{bmatrix}. \end{aligned}$$

Similarly, if we continue applying the Midpoint rule to g and h, such that

$$\bar{g}_i = \frac{2i-1}{2N} \qquad \bar{h}_j = \frac{2j-1}{2N}$$

we get

$$\mathbf{B} = \mathbf{A_2XA_1}\,.$$

Or, equivalently, using properties of the Kronecker product and denoting $\mathbf{A}_r = \mathbf{A}_1^T$ and $\mathbf{A}_c = \mathbf{A}_2$, where $\mathbf{A}_r$ and $\mathbf{A}_c$ are the one-dimensional convolutions on the rows and columns of the matrix $\mathbf{A}$ respectively, we obtain

$$\mathbf{b} = (\mathbf{A}_r \otimes \mathbf{A}_c)\mathbf{x}\,. \tag{2.4}$$

with

$$\mathbf{X} = \begin{bmatrix} x(\bar{s}_i, \bar{t}_1) & \cdots & x(\bar{s}_N, \bar{t}_1) \\ \vdots & \ddots & \vdots \\ x(\bar{s}_1, \bar{t}_N) & \cdots & x(\bar{s}_N, \bar{t}_N) \end{bmatrix}$$

$$\mathbf{A_1} = \frac{1}{N} \begin{bmatrix} a_1(\bar{g}_1, \bar{s}_1) & \cdots & a_1(\bar{g}_N, \bar{s}_1) \\ \vdots & \ddots & \vdots \\ a_1(\bar{g}_1, \bar{s}_N) & \cdots & a_1(\bar{g}_N, \bar{s}_N) \end{bmatrix}$$

$$\mathbf{A_2} = \frac{1}{N} \begin{bmatrix} a_2(\bar{h}_1, \bar{t}_1) & \cdots & a_2(\bar{h}_N, \bar{t}_1) \\ \vdots & \ddots & \vdots \\ a_2(\bar{h}_1, \bar{t}_N) & \cdots & a_2(\bar{h}_N, \bar{t}_N) \end{bmatrix}$$

and $\otimes$ is the Kronecker product, defined as,

$$\mathbf{A}_r \otimes \mathbf{A}_c = \begin{bmatrix} a_{r_{11}}\mathbf{A}_c & a_{r_{12}}\mathbf{A}_c & \cdots & a_{r_{1n}}\mathbf{A}_c \\ a_{r_{21}}\mathbf{A}_c & a_{r_{22}}\mathbf{A}_c & \cdots & a_{r_{2n}}\mathbf{A}_c \\ \vdots & \vdots & & \vdots \\ a_{r_{m1}}\mathbf{A}_c & a_{r_{m2}}\mathbf{A}_c & \cdots & a_{r_{mn}}\mathbf{A}_c \end{bmatrix}$$

is the $pm \times qn$ block matrix for matrix $\mathbf{A}_r$ of size $m \times n$ and matrix $\mathbf{A}_c$ of size $p \times q$.

2.2 Boundary conditions

When dealing with an image deblurring problem, the existence of the “spill over” effect among pixels can significantly reduce the accuracy of our result, specially at the boundaries of the image. To overcome that effect, recall that in equation (2.2), we claim that a pixel in the blurred image, b_{ij}, can be represented by the weighted sum of the pixel x_{ij} and its neighbors. However, if b_{ij} is near the edge of the image, this task may be impossible since we do not have the information of its neighbors, which are outside of the captured image. Therefore, it is important to make assumptions regarding the boundary conditions of the image.

There are several common boundary conditions used in image processing problems. The first and also the simplest one is the zero boundary condition, where we assume the true image is surrounded by black color. Mathematically, it can be represented as

$$\mathbf{X}_{exact} = \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{X} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix}$$

where $\mathbf{0}$ submatrices represent a border of zero elements.

However, realistically, it is rarely the case that the outside region of the image is only black. Instead, this region is a continuation of the colors and features of the image's boundaries. In most image processing problems, the periodic boundary condition is assumed, where we assume the outside region is just a replica of the image $\mathbf{X}$. Mathematically, it can be represented as

$$\mathbf{X}_{exact} = \begin{bmatrix} \mathbf{X} & \mathbf{X} & \mathbf{X} \\ \mathbf{X} & \mathbf{X} & \mathbf{X} \\ \mathbf{X} & \mathbf{X} & \mathbf{X} \end{bmatrix}.$$

Another approach is to use the reflexive boundary condition, that is we assume the outside region is the mirror of the image itself. The construction of $\mathbf{X}_{exact}$ now becomes more complex. Denote $\mathbf{X}_{lr}$ where $\mathbf{x}_i^T = \mathbf{x}_{lr_{n+1-i}}^T$, where n is the column (or row) number of matrix $\mathbf{X}$ and the subscript j denotes the j-th row of any matrix. In other words, $\mathbf{X}_{lr}$ is the matrix $\mathbf{X}$ being flipped left to right. Similarly, we denote matrix $\mathbf{X}_{ud}$ as the matrix $\mathbf{X}$ being flipped upside down, i.e $\mathbf{x}_i = \mathbf{x}_{n+1-i}$. And lastly, $\mathbf{X}_{lu}$ is the matrix $\mathbf{X}_{ud}$ being flipped left to right.

For example, if

$$\mathbf{X} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

then

$$\mathbf{X}_{lr} = \begin{bmatrix} c & b & a \\ f & e & d \\ i & h & g \end{bmatrix} \qquad \mathbf{X}_{ud} = \begin{bmatrix} g & h & i \\ d & e & f \\ a & b & c \end{bmatrix} \qquad \mathbf{X}_{lu} = \begin{bmatrix} i & h & g \\ f & e & d \\ c & b & a \end{bmatrix}$$

With these notations, matrix $\mathbf{X}_{exact}$ can be written as

$$\mathbf{X}_{exact} = \begin{bmatrix} \mathbf{X}_{lu} & \mathbf{X}_{ud} & \mathbf{X}_{lu} \\ \mathbf{X}_{lr} & \mathbf{X} & \mathbf{X}_{lr} \\ \mathbf{X}_{lu} & \mathbf{X}_{ud} & \mathbf{X}_{lu} \end{bmatrix} .$$

This thesis only works with zero boundary conditions. Further details on how to construct the blurring matrices for the case of periodic and reflexive boundary conditions can be found in [9].

2.3 Solving the $\mathbf{Ax} = \mathbf{b}$ problem

As aforementioned in Section 1.2, the image deblurring problem is an $\mathbf{Ax} = \mathbf{b}$ problem, where, based on the assumption used throughout this thesis, $\mathbf{A} = \mathbf{A}_r \otimes \mathbf{A}_c$ is the blurring operator constructed in Section 2.1, while $\mathbf{x}$ and $\mathbf{b}$ are the vector representations of the true and blurred images respectively. A naive approach to solve this linear system is simply solving the linear algebraic system using properties of the

Kronecker products, for example

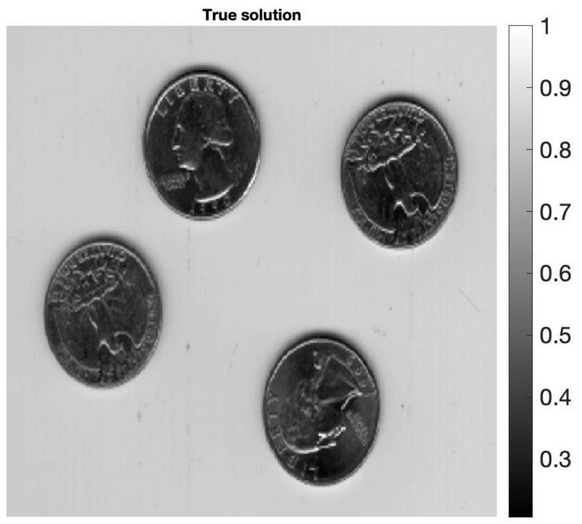
$$\begin{aligned}\mathbf{x} = \mathbf{A}^{-1}\mathbf{b} &= (\mathbf{A}_r \otimes \mathbf{A}_c)^{-1}\mathbf{b} \\ &= (\mathbf{A}_r^{-1} \otimes \mathbf{A}_c^{-1})\mathbf{b} \\ &= \mathrm{vec}(\mathbf{A}_c^{-1}\mathbf{B}\mathbf{A}_r^{-T})\,. \end{aligned} \tag{2.5}$$

Or, equivalently:

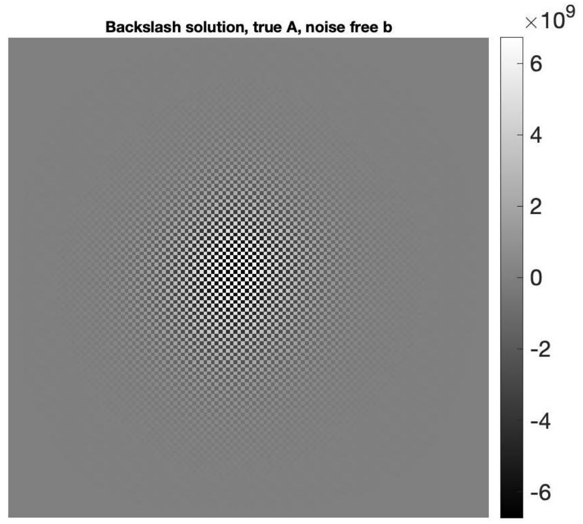
$$\mathbf{X} = \mathbf{A}_c^{-1}\mathbf{B}\mathbf{A}_r^{-T} \tag{2.6}$$

where $\mathbf{X}, \mathbf{Y} \in \mathbb{R}^{m\times n}$ are respectively the matrix representations of the exact and blurred images, such that $\mathrm{vec}(\mathbf{X}) = \mathbf{x}$ and $\mathrm{vec}(\mathbf{B}) = \mathbf{b}$. More details about this can be found in Section 1.2.

Using equation (2.6), we get the following result, which is far from the true image, even though we have the exact information of matrices **A** and **B**.



(a)



(b)

Figure 2.1: (a) True image and (b) Naive solution with noise free **A** and **B**.

The reason is that both $\mathbf{A}_r$ and $\mathbf{A}_c$ are large and ill-conditioned matrices and that the image **X** is dominated by the influence from rounding errors. Additionally, as aforementioned, in more practical situations, the information for the observed blurry image **B** we have is not quite accurate as it contains several types of noise and errors, which are excluded in the model [9]. These errors eventually add up and may give us an even worse solution as shown in this figure below:

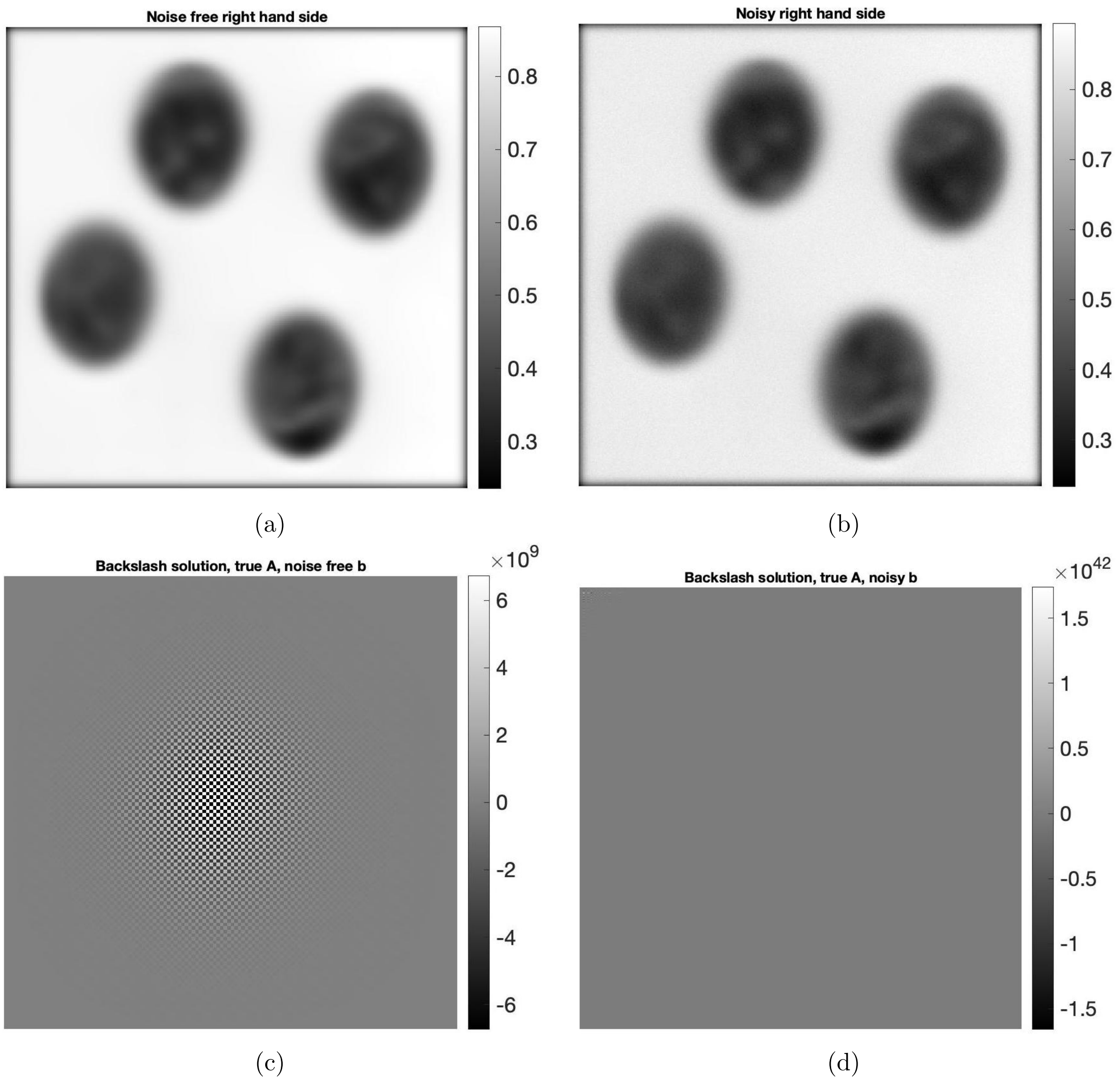


Figure 2.2: (a) Noise free **B**; (b) Noisy **B**; (c) Naive solution with noise free **A** and **B**; and (d) Naive solution with noise free **A** and noisy **B**.

Hence, the result of naively calculating **X** using equation (2.6) always consists of some forms of errors, such that

$$\mathbf{X}_{\text{naive}} = \mathbf{A}_c^{-1}(\mathbf{B} + \mathbf{E})\mathbf{A}_r^{-T} = \mathbf{A}_c^{-1}\mathbf{B}\mathbf{A}_r^{-T} + \mathbf{A}_c^{-1}\mathbf{E}\mathbf{A}_r^{-T}$$

where $\mathbf{E}$, which has the same dimension as $\mathbf{X}$ and $\mathbf{B}$, is the noise image representing both the noise and the other errors in the observed image $\mathbf{B}$.

Hence,

$$\mathbf{X}_{\text{naive}} = \mathbf{X} + \mathbf{A}_c^{-1}\mathbf{E}\mathbf{A}_r^{-T} \tag{2.7}$$

The term $\mathbf{A}_c^{-1}\mathbf{E}\mathbf{A}_r^{-T}$ denotes the inverted noise, which can dominate the solution if its elements are larger than that of $\mathbf{X}$. Unfortunately, this is usually the case, requiring us to come up with more complicated methods so as to retrieve the original image.

Performing the same analysis with the general model in equation (2.5), we also obtain

$$\mathbf{x}_{\text{naive}} = \mathbf{x} + \mathbf{A}^{-1}\mathbf{e} = \mathbf{x} + \mathbf{A}^{-1}\mathbf{e} \tag{2.8}$$

where $\mathbf{e} = \text{vec}(\mathbf{E})$.

From these two analyses, it is clear that the deblurred image, i.e $\mathbf{x}$, always contains two parts: the exact solution and the inverted noise. Our main task is to minimize the inverted noise term in order to get the most deblurred image for our model. To do so, we first need to gain some insights regarding this term through the use of the Singular Value Decomposition (SVD).

Recall that our blurring operator matrix $\mathbf{A}$ is a square matrix of size $N = mn$, whose SVD is

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$$

where $\mathbf{U}, \mathbf{V} \in \mathbb{R}^{N \times N}$ are orthogonal matrices ($\mathbf{U}^T\mathbf{U} = \mathbf{V}^T\mathbf{V} = \mathbf{I}$) and $\mathbf{\Sigma} = \text{diag}(\sigma_i)$

is a diagonal matrix of size N whose diagonal entries σ_i satisfy

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_N \geq 0$$

These σ_i values are called singular values.

Another representation of the SVD factorization of $\mathbf{A}$ is

$$\mathbf{A} = \sum_{i=1}^{N} \sigma_i \mathbf{u}_i \mathbf{v}_i^T$$

where $\mathbf{u}_i$ and $\mathbf{v}_i$ are the i-th column of the matrix $\mathbf{U}$ and $\mathbf{V}$, respectively. The columns $\mathbf{u}_i$ of $\mathbf{U}$ and $\mathbf{v}_i$ of $\mathbf{V}$ are called the left and right singular vectors, respectively. As $\mathbf{U}^T\mathbf{U} = \mathbf{V}^T\mathbf{V} = \mathbf{I}$, it is implied that $\mathbf{u}_i^T\mathbf{u}_j = \mathbf{v}_i^T\mathbf{v}_j$, which are equal to 0 when $i \neq j$ and equal to 1 when $i = j$.

Assuming that all these singular values are strictly positive, the matrix $\mathbf{A}$ is invertible. We also note that $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices so $\mathbf{U}^T = \mathbf{U}^{-1}$ and $\mathbf{V}^T = \mathbf{V}^{-1}$. The inverse matrix of $\mathbf{A}$ can then be written as

$$\mathbf{A}^{-1} = \left(\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T\right)^{-1} = \mathbf{V}\mathbf{\Sigma}^{-1}\mathbf{U}^T = \sum_{i=1}^{N} \frac{1}{\sigma_i}\mathbf{v}_i\mathbf{u}_i^T$$

where the diagonal entries of $\mathbf{\Sigma}^{-1}$ are $1/\sigma_i$.

The inverted noise now becomes

$$\mathbf{A}^{-1}\mathbf{e} = \sum_{i=1}^{N} \frac{1}{\sigma_i}\mathbf{v}_i\mathbf{u}_i^T\mathbf{e} = \sum_{i=1}^{N} \frac{\mathbf{u}_i^T\mathbf{e}}{\sigma_i}\mathbf{v}_i\,.$$

From this SVD representation, we can see that the main problem with the naive solution is that the noise is being magnified by the inversion term of very small singular values as these values decay to a value very close to zero [9].

Therefore, our main goal is to minimize the error to obtain the best solution for our image deblurring problem. We can begin rewriting the problem in least squares form

$$\min_{\mathbf{x}} ||\mathbf{A}\mathbf{x} - \mathbf{b}||_2^2 \tag{2.9}$$

or its regularized version

$$\min_{\mathbf{x}} ||\mathbf{A}\mathbf{x} - \mathbf{b}||_2^2 + \lambda ||\mathbf{x}||_2^2 \,. \tag{2.10}$$

Additionally, it is easy to solve the problem in equation (2.4) using the properties of the Kronecker products. However, when we use interpolation to approximate a general spatially variant blur, matrix $\mathbf{A}$ will be a sum of multiple Kronecker products (see Section 2.5). In this case, it is crucial to use iterative methods, where the main cost is matrix-vetor multiplications, which can be done efficiently.

In this project, we utilize the `IRcgls` function in the `IRtools` MATLAB package developed by Gazzola, Nagy, and Hansen [4], which uses the iterative version of the Conjugate Gradient algorithm to solve Least Squares problems and return a regularized solution. The software package `IRTools` provides implementations of a range of iterative solvers for large-scale ill-conditioned linear systems where regularization is needed to stabilize computations. These solvers include iterative regularization methods where the regularization is due to the semi-convergence of the iterations, Tikhonov-type formulations where the regularization is explicitly formulated in the form of a regularization term, and methods that can impose bound constraints on computed solutions. The package also contains "hybrid" methods that use iterative Krylov subspace methods combined with SVD-based direct regularization methods on small subproblems in each iteration. The advantage of the hybrid approach is that regularization parameters can be estimated at each iteration. The software package also contains a set of test problems that represent realistic large-scale problems found in image reconstruction and several other applications. The solvers in the toolbox use the naming convention `IRxxxx`, where `IR` denotes "Iterative Regularization", and `xxxx` refers to a specific method. In this thesis we make use of the `IRcgls` solver, which applies the conjugate gradient method for least squares problems (implicitly) to the problem $\mathbf{A}^T\mathbf{A}\mathbf{x} = \mathbf{A}^T\mathbf{b}$. Regularization can be enforced by terminating the iteration at an "optimal" stopping iteration. The discussion of iterative methods and regularization by early termination is beyond the scope of this thesis; for further details, see [4].

2.4 Sparse matrix-vector multiplication

Even though `MATLAB` is very fast and reliable for decently large matrices, matrices in imaging problems usually far exceed `MATLAB`'s ability, resulting in poor results and long running times. However, fortunately, our matrix structures are quite special, allowing us to use specific algorithms to help speed up the running time of the problem.

When using iterative methods to solve the Least Squares problem (2.10), if matrix $\mathbf{A}$ can be written as a sum of multiple Kronecker products then this structure can be exploited to efficiently compute matrix-vector products with matrix $\mathbf{A}$. This task is implemented in the `MATLAB` Package Restore Tools by Nagy, Palmer, and Perrone [8]. Specifically, the package defines a `kronMatrix` object and overloads the `mtimes` function.

Note that our matrix $\mathbf{A}$ is very large and sparse. Hence, applying iterative methods to solve large linear systems without taking into consideration the sparseness of matrix $\mathbf{A}$ can be very inefficient as those methods may require hundreds or even thousands of matrix-vector products to converge. Specifically, the number of floating point operations (FLOPs) for a sparse matrix-vector multiplication (SpVM) is always twice the number of nonzeros in the matrix as for each nonzero entry in the matrix, the matrix-vector multiplication requires one multiplication and one addition operation [2]. This FLOP calculation is independent of the matrix dimension; therefore, if matrix $\mathbf{A}$ is very sparse, we can come up with a more efficient way to compute SpVM. In this project, we utilize the class `kronMatrix` in the `IRtools` package (originally developed for the restore tool project to overcome this problem), along with `MATLAB`'s built-in sparse matrix capabilities.

A Kronecker product can be generalized as

$$\mathbf{A} = \mathbf{A}_r \otimes \mathbf{A}_c = \begin{bmatrix} a_{r_{11}}\mathbf{A}_c & a_{r_{12}}\mathbf{A}_c & \cdots & a_{r_{1n}}\mathbf{A}_c \\ \vdots & \vdots & & \vdots \\ a_{r_{n1}}\mathbf{A}_c & a_{r_{n2}}\mathbf{A}_c & \cdots & a_{r_{nn}}\mathbf{A}_c \end{bmatrix}$$

where $a_{r_{ij}}$ is the entry on the i-th row and j-th column of matrix $\mathbf{A}_r$. Storing this Kronecker product explicitly requires space of an $n^2 \times n^2$ matrix. Additionally, calculating the matrix-vector multiplication of an explicitly formed Kronecker product with a vector is also very costly. Specifically, we can write

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} \mathbf{a}_1^T\mathbf{x} & \mathbf{a}_2^T\mathbf{x} & \cdots & \mathbf{a}_n^T\mathbf{x} \end{bmatrix}^T$$

where $\mathbf{a}_i^T$ is the i-th row of matrix $\mathbf{A}$.

Calculating this matrix-vector multiplication requires $3n^4$ FLOPs in total (n^4 to explicitly form the Kronecker product and $2n^4$ to perform the matrix-vector multiplication $\mathbf{A}\mathbf{x}$). However, if we take advantage of the properties of Kronecker products, we can write

$$\mathbf{b} = \text{vec}(\mathbf{A}_c\mathbf{X}\mathbf{A}_r^T)\,.$$

This representation is both FLOP-efficient and storage-efficient as it only requires at most $4n^3$ FLOPs and the storage size of an $n \times n$ matrix. In our situation, $\mathbf{A}_r$ and $\mathbf{A}_c$ are very sparse, so we only need to store their nonzero entries and the SpVMs will require fewer than $4n^3$ FLOPs, depending on the level of sparsity. The `kronMatrix` object utilizes these properties to more efficiently compute and store matrix-vector multiplications. More details on this package can be found in [8].

2.5 Spatially invariant and variant Gaussian blur

We can sometimes further speed up these matrix-vector multiplications using the fast Fourier transform [7]. Without loss of generosity, assume that $\mathbf{A} = \mathbf{A}_r \otimes \mathbf{A}_c$, i.e. only one term in the sum of Kronecker products. If the blur is spatially invariant and assuming zero boundary condition, then matrices $\mathbf{A}_r$ and $\mathbf{A}_c$ are Toeplitz (or diagonal-constant) matrices. For example, a Toeplitz matrix of size 5×5 has the following form:

$$\begin{bmatrix} a & b & c & d & e \\ f & a & b & c & d \\ g & h & a & b & c \\ i & g & h & a & b \\ j & i & g & h & a \end{bmatrix}.$$

These matrices can be constructed from a point spread function. A single point spread function (i.e. an image of a point source) provides everything we need to know to construct $\mathbf{A}_r$ and $\mathbf{A}_c$. More details on this construction can be found in [9].

However, if the blur is spatially variant, then to construct matrices $\mathbf{A}_r$ and $\mathbf{A}_c$, we need to have a point spread function centered at each pixel location in the image, which is computationally infeasible for large images. The approach of Nagy and O'Leary is to obtain a sample of point spread functions distributed throughout the image domain, assume the blur is locally spatially invariant, then use interpolation to construct the point spread functions. This can be formally written as

$$\mathbf{A} = \mathbf{A}_1\mathbf{D}_1 + \mathbf{A}_2\mathbf{D}_2 + \cdots + \mathbf{A}_p\mathbf{D}_p$$

where each $\mathbf{A}_i = \mathbf{A}_{r_i} \otimes \mathbf{A}_{c_i}$ represents a spatially invariant blur corresponding to region i in the image, while p is the number of image regions the image is partitioned into, and $\mathbf{D}_i$ are interpolation matrices that satisfy

$$\mathbf{D}_1 + \mathbf{D}_2 + \cdots + \mathbf{D}_p = \mathbf{I}\,.$$

In their paper, Nagy and O'Leary discuss piecewise constant interpolation and mention piecewise linear, but they do not perform extensive testing. The implementation in Restore Tools uses only piecewise constant interpolation. However, this can cause boundary artifacts in neighboring regions. In this thesis, we look more deeply into piecewise linear and radial basis function interpolation methods. We conclude that both of these methods generally outperform constant interpolation.

2.6 The interpolation matrix $\mathbf{D}_i$

This section focuses on how to construct the weight interpolation matrices $\mathbf{D}_i$ of size $n \times n$, where n is the number of columns of the image, through the simple constant approach and the piecewise linear approach. Assuming that the PSF captures all light, we want each row sum of matrix $\mathbf{D}_i$ to be 1. As aforementioned, the main focus of this thesis is on using the radial basis function for the matrices $\mathbf{D}_i$, which is discussed later in Chapter 3. The resulting image using these interpolation weights is presented in Chapter 4.

2.6.1 Piecewise constant interpolation weight

The constant approach sets an equal weight of 1 for the center of each region and 0 for everywhere else. In the case that the image of size 512×1 is partitioned into 4 image regions, the representation of the piecewise constant interpolation weight is as follows:

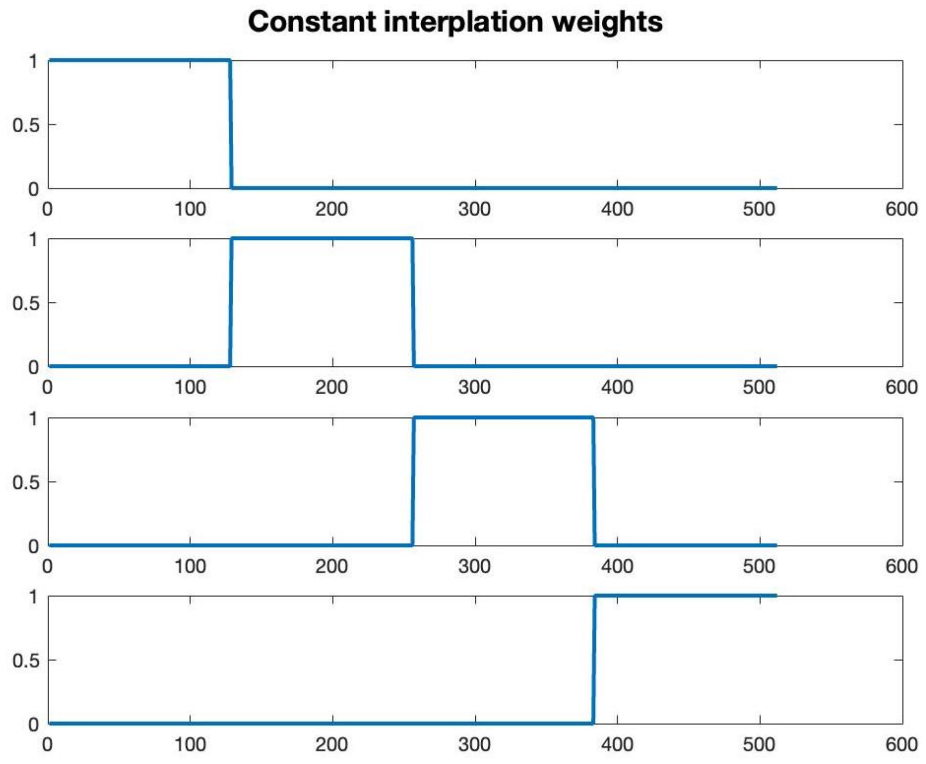


Figure 2.3: Piecewise constant interpolation weights.

The values shown in Figure 2.3 are of diagonal entries of matrix $\mathbf{D}_i$.

In this case, matrix $\mathbf{D}_i$ has size 512×512, $i = 1, 2, 3, 4$,... and is guaranteed to have $\mathbf{D}_1 + \mathbf{D}_2 + \mathbf{D}_3 + \mathbf{D}_4 = \mathbf{I}$, satisfying our above assumption.

2.6.2 Piecewise linear interpolation weight

Similar to the computation of interpolation weights using the piecewise constant approach, the piecewise linear approach also assigns the weight value 1 to the center of each region. However, instead of having the whole considered image region to be 1, only the center of the considered region is 1. Then, we interpolate a straight line, either with negative or positive slope, depending on whether it is a boundary or an interior region, passing through the center of the considered image region.

Specifically, if it is the first region, we set its value to constant 1 until its center, i.e if we define $\mathbf{d}_i = \text{diag}(\mathbf{D}_i)$, then $[\mathbf{d}_i]_{1:I_1} = 1$ where I_i is the i-th index of the

i-th region's center. Then we interpolate a straight downward line from the value 1 assigned for the center of the first region down to the value 0 assigned for the center of the second region. The slope of this line is calculated as

$$s = \frac{1}{I_1 - I_2}.$$

Then the weight corresponding to this region, or in other words, the weight corresponding to each point from the center of the first region to the center of the second region is

$$[\mathbf{d}_1]_k = s(k - I_2)$$

where $k \in \left\{ I_1 + 1; I_1 + 2; \cdots ; I_2 \right\}$ and $[\mathbf{d}_i]_k$ denotes the k-th entry of vector $\mathbf{d}_i$.

For the interior regions, assuming that it is the i-th region $(1 < i < p \leq n)$, we want a line upward passing through the value 0 assigned for the center of the $(i-1)$-th region and the value 1 assigned for the center of the i-th region, whose slope can be calculated as

$$s = \frac{1}{I_i - I_{i-1}}$$

and its associated weights are

$$[\mathbf{d}_i]_k = s(k - I_{i-1})$$

where $k \in \left\{ I_{i-1}; I_{j-1} + 1; \cdots ; I_j \right\}$.

Then, we want a downward line, similar to what we did to the first region, from the value 1 assigned for the center of the i-th region to the value 0 assigned for the center of the $(i+1)$-th region. The slope of this line and its corresponding weights can be calculated as

$$s = \frac{1}{I_i - I_{i+1}}, \qquad [\mathbf{d}_i]_k = s(k - I_{i+1})$$

with $k \in \left\{ I_i + 1; I_i + 2; \cdots ; I_{i+1} \right\}$.

Finally, for the last region, we want an upward line from the value 0 assigned for the center of the $(p-1)$-th region up to the value 1 assigned for the center of the p-th region. The slope and its corresponding weights are

$$s = \frac{1}{I_p - I_{p-1}}, \qquad [\mathbf{d}_p]_k = s(k - I_{p-1})$$

where $k \in \left\{ I_{p-1}; I_{p-1} + 1; \cdots ; I_p \right\}$.

Then the remaining values from the center of the last region until the end point of the last region will be constant at 1, i.e $[\mathbf{d}_p]_k = 1$ where $k \in \left\{ I_p + 1; I_p + 2; \cdots ; n \right\}$.

Continuing with the example of a 1-D 512×1 image being partitioned into 4 image regions in Section 2.6.1, we have the following interpolation weights shown in Figure 2.4. Again, these values are of diagonal entries of the interpolation matrix $\mathbf{D}_i$.

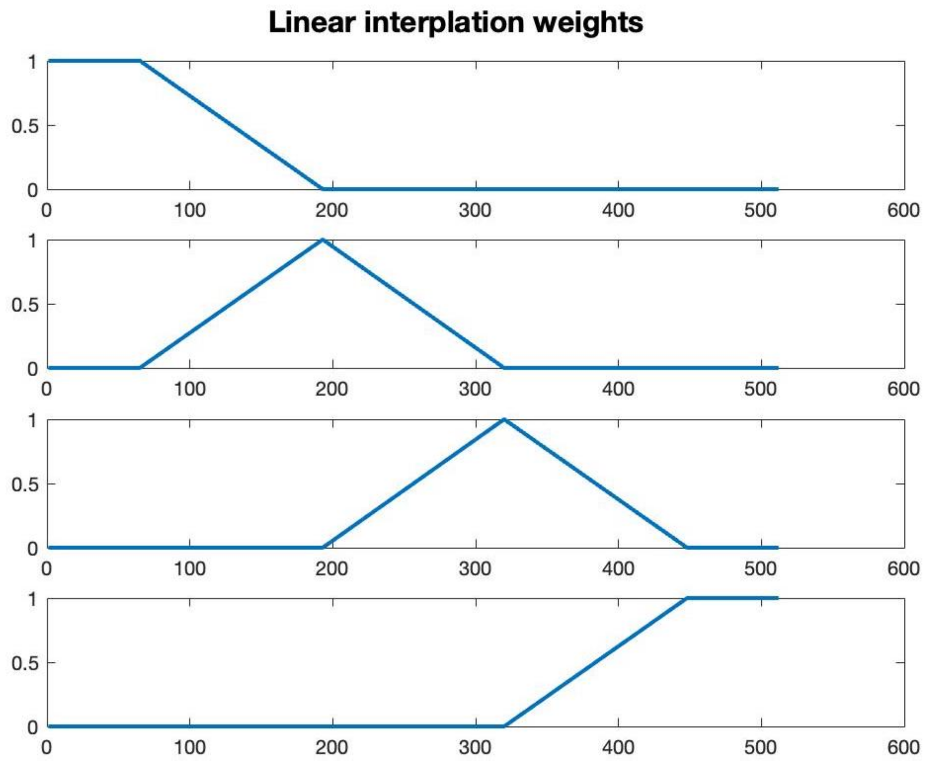


Figure 2.4: Piecewise linear interpolation weights.

Chapter 3

Radial Basis Function (RBF)

To calculate the interpolation matrices $\mathbf{D}_i$ in each image region, previous literature uses piecewise constant and linear approaches. In this chapter, we consider Radial Basis Function interpolation. Using the same idea as in the piecewise constant and linear approaches, our goal is to ensure that the RBF interpolation matrices sum to the identity matrix $\mathbf{I}$.

3.1 A brief summary of RBF

Radial Basis function (RBF) is a real-valued function, which takes in a vector and returns a scalar. The value of the RBF is solely based on the distance, usually Euclidean distance, between the input and a fixed point. There are many ways to set up an RBF depending on the choice of the basis function, such as multiquadric and polyharmonic spline. In this thesis, to ensure smoothness between regions, we work with the standard form of RBF using the Gaussian function as follows:

$$R(\mathbf{x}) = \sum_{i=1}^{n} w_i \exp\left(-\gamma ||\mathbf{x} - \mathbf{x}_i||^2\right)$$

where:

w_i: the weight of the RBF

γ: a parameter

$\mathbf{x}$: the input data

$\mathbf{x}_i$: the known data already in the model

To apply this function in interpolating the blurring function, we need to find suitable weights w_i of the RBF and the parameter γ of the basis function, which are discussed more specifically later on.

3.2 Choosing the parameter γ

To choose a suitable parameter γ for the radial function, we utilize the idea of the Full Width at Half Maximum coefficient (FWHM). Such coefficient corresponding to the Gaussian function is

$$\text{FWHM} = 2\sqrt{\ln(2)}\,.$$

Additionally, the spread of the Gaussian function, determined by the parameter γ, is defined independently in each image region as each region is independent of each other. Therefore, the FWHM is slightly different among regions. Hence, the γ parameter is calculated as

$$\gamma = \frac{I_{\text{end}} - I_{\text{start}}}{\text{FWHM}}$$

where

I_{end}: the index of the last point in the region

I_{start}: the index of the first point in the region

FWHM $= 2\sqrt{\ln(2)}$

3.3 Choosing the weight $\mathbf{w}$

To help with notation, as in previous sections, let $\mathbf{d}_i = \text{diag}(\mathbf{D}_i), i = 1, 2, \cdots, p$ where p is the number of regions being interpolated. Here we also define $\widehat{\mathbf{D}} = \begin{bmatrix} \mathbf{d}_1 & \mathbf{d}_2 & \cdots \mathbf{d}_p \end{bmatrix} \in \mathbb{R}^{n \times p}$.

The light intensity of the PSF is a local phenomenon as it is confined in a certain radius around the point source and 0 everywhere else. Additionally, another assumption is that the imaging process captures all light; hence, we want the sum of all the pixel values in the PSF to be 1. In this case, we want the sum of the RBF values corresponding to each row of $\widehat{\mathbf{D}}$ to be 1.

After calculating the RBF value for each pixel, we have a matrix $\hat{\mathbf{D}} \in \mathbb{R}^{n \times p}$ where n is the number of pixels in the image and p is the number of regions the image is partitioned into. As aforementioned, the goal is for all the row sums of this matrix $\widehat{\mathbf{D}}$ to be 1. In the case of constant and linear interpolation, this occurs by default, but in the case of RBF, the sums may be slightly different from 1. We therefore perform the following normalization.

1. Construct a diagonal matrix with the reciprocals of the current matrix $\widehat{\mathbf{D}}$'s row sums

$$\mathbf{S} = \text{diag}(1/(\text{row sum of matrix } \widehat{\mathbf{D}}))$$

2. Normalize and update $\widehat{\mathbf{D}}$ so its row sums are 1.

$$\widehat{\mathbf{D}} \leftarrow \mathbf{S}\widehat{\mathbf{D}}$$

Continuing with the example used in Sections 2.6.1 and 2.6.2, we have the following RBF interpolation weights on a 512×1 image that is partitioned into 4 image regions:

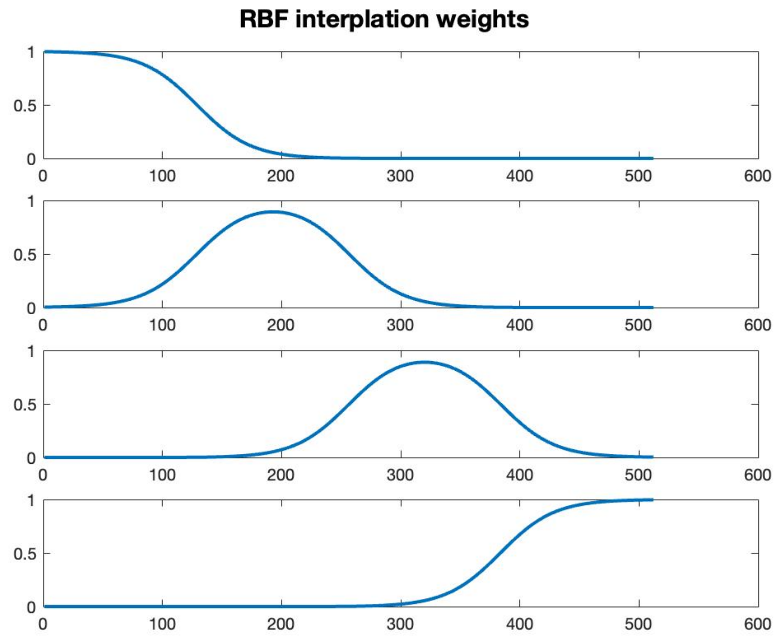


Figure 3.1: Radial basis functions interpolation weights.

3.4 Number of interpolated neighboring regions

When we partition the image into many regions, it is possible that we would get worse results as the region sizes become very small, and the PSF overlaps more than one region. In the previous sections, 2.6.1, 2.6.2, and 3.3, we only interpolate the nearest neighbor regions. One way to overcome this is to interpolate across more regions, for example the three nearest neighbors. Considering the following image, which is the example of piecewise linear interpolation over three neighboring regions

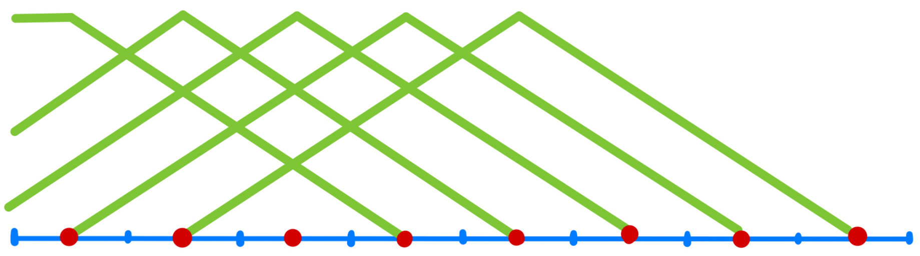


Figure 3.2: Piecewise linear interpolation over three neighboring regions.

Here, the blue dashes and the red dots denote the region boundaries and centers, respectively, and the green line is the diagonal entries of the interpolation matrix $\mathbf{D}_j$, where j denotes the j-th region.

However, we also need to be careful with regions near the image boundaries. That is, suppose we want to traverse through $j = 1, 2, 3, \cdots, p$, to construct weights. Let $\mathbf{c}$ denote an array that contains the index of every image regions' center; that is

$$\mathbf{c} = \begin{bmatrix} c_1 & c_2 & \cdots & c_p \end{bmatrix}$$

where c_j is the index of the j-th region's center and p is the number of image regions. We also define Δr to be the number of regions over which interpolating will occur (e.g. $\Delta r = 3$).

In the beginning, if

$$j - \Delta r < 0$$

the nonzero weights begin at index 1 and continue until $c_{j+\Delta r}$, after which the weights are zero. Otherwise, we begin at $c_{j-\Delta r}$ and end at $c_{j+\Delta r}$. To make implementations simpler, we could append 0 on the left of the array $\mathbf{c}$, then the left part will always start at

$$c_{\max(1,\, j-\Delta r)}.$$

Applying the same method for the other boundary, we can now write the general coordinates of nonzero weights for any j-th region for its left and right part, respectively, as

$$c_{\max(1,\,j-\Delta r)} : c_j \qquad \text{and} \qquad c_j : c_{\min(j+\Delta r,\,p)} \tag{3.1}$$

With such rules stated in equation (3.1), we can explicitly state the coordinates of our interpolation linear lines for the example in Figure 3.2 as follows

Region	Left part	Right part
1	$1 : c_1$	$c_1 : c_{1+\Delta r}$
2	$1 : c_2$	$c_2 : c_{2+\Delta r}$
3	$1 : c_3$	$c_3 : c_{3+\Delta r}$
4	$c_1 : c_4$	$c_4 : c_{4+\Delta r}$

Table 3.1: Coordinates of the interpolation linear weights.

The above explanation is for piecewise linear weights. However, in this thesis, we only implement such method for the RBF weights but a similar approach can be done for linear interpolation when we partition the image into many regions. Note that, for the weights to interpolate over Δr neighbor regions, our γ also needs to be readjusted as

$$\gamma = \Delta r \frac{I_{\text{end}} - I_{\text{start}}}{\text{FWHM}}$$

The following image is when we partition a 512×1 image into 4 regions with $\Delta r = 3$. We also notice here that the curve of the RBF weights is much smoother with this approach compared to the result shown in Figure 3.1.

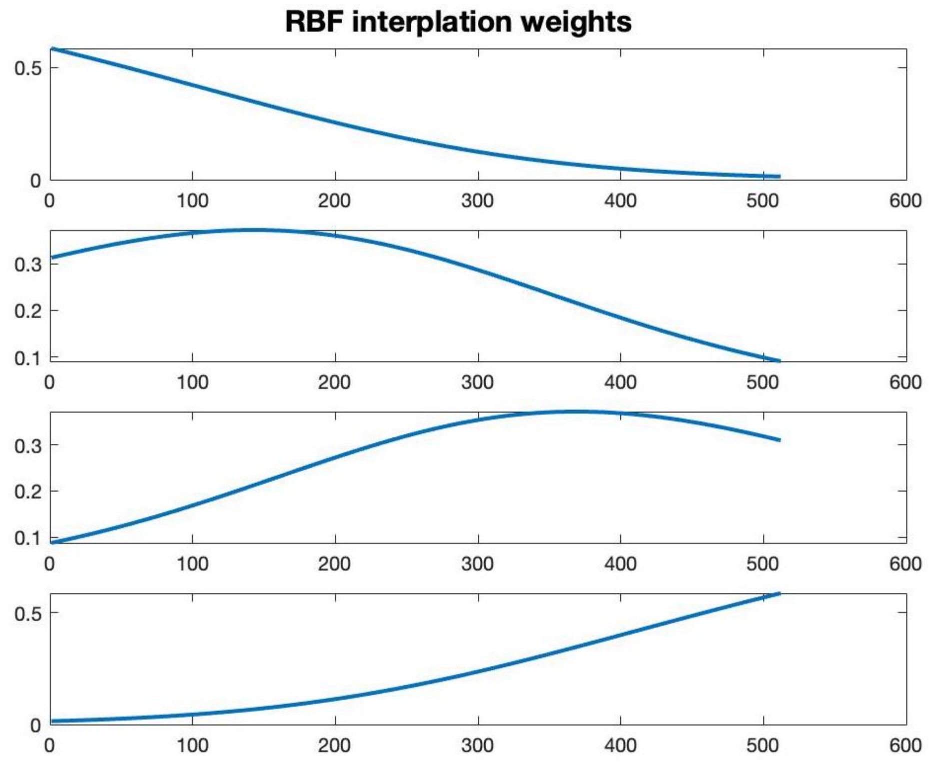


Figure 3.3: RBF interpolation over three neighboring regions

Until now we have illustrated the interpolation using 1-D images. The idea natually extends to 2-D using Kronecker products. That is, suppose we have 2-D image partitioned in 4×4 subregions, e.g.

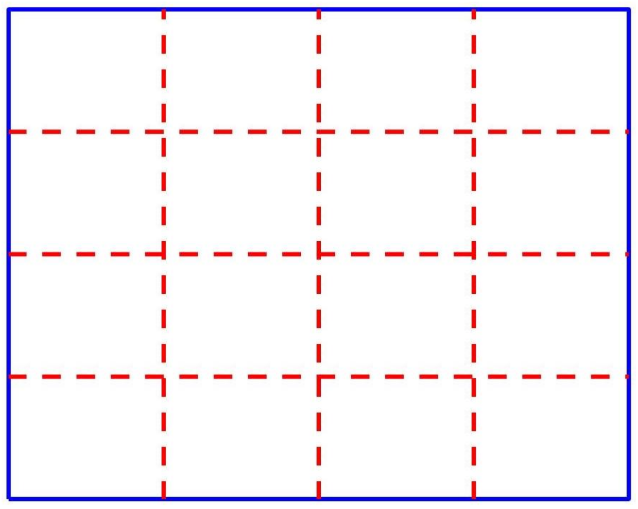


Figure 3.4: 2-D image partitioned into 4×4 subregions.

Then, we create a diagonal interpolation matrix for each region having the form

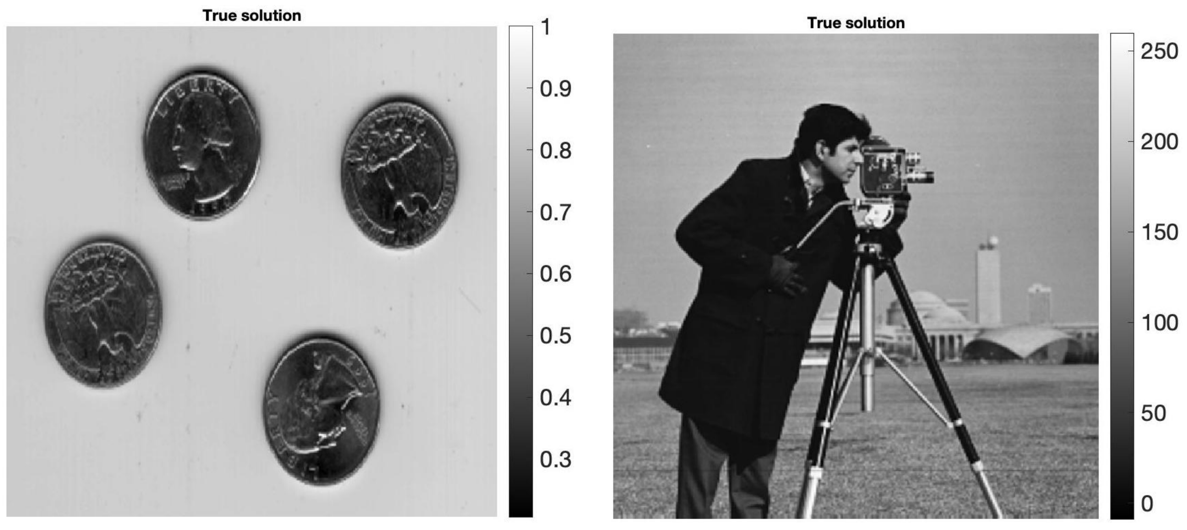
$$\mathbf{D}_{ij} = \mathbf{D}_i \otimes \mathbf{D}_j$$

where matrices $\mathbf{D}_i$ and $\mathbf{D}_j$ are the same as already discussed for 1-D problems, and $i = 1, 2, 3, 4,\ j = 1, 2, 3, 4.$

Chapter 4

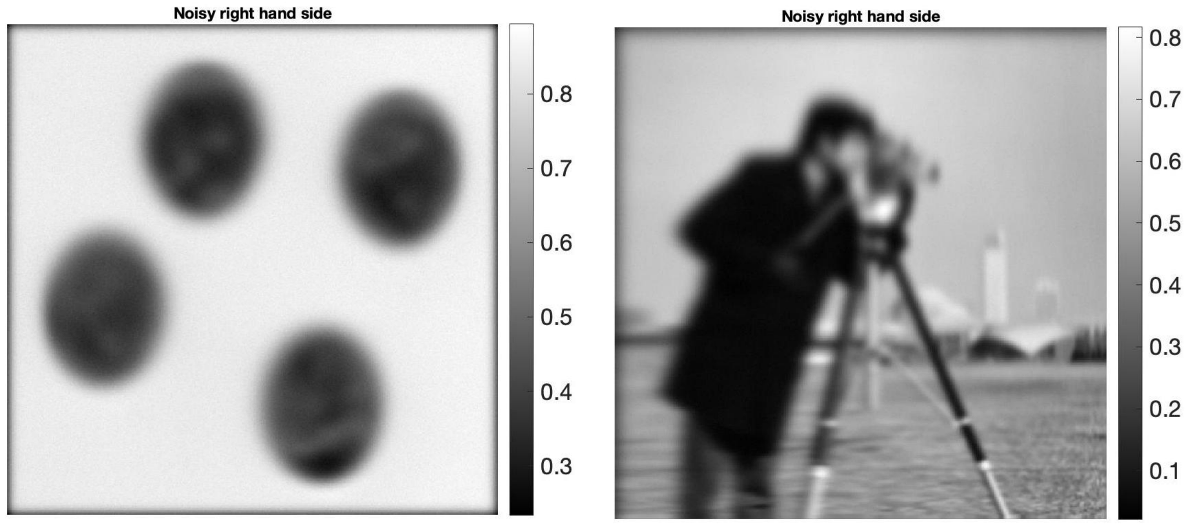
Numerical Experiments

In this section, we present several experiments for the image deblurring problem. For each experiment, if not specified, the number of regions is set at $n/16$ by default where n is the number of columns of the original image (all images are of size $n \times n$). Additionally, throughout this section, the two images that are used are `eight.tiff` and `cameraman.tiff` as shown in Figure 4.1, which are in the `MATLAB` image processing toolbox. An example of the recorded blurry images are also shown in Figure 4.2. The reason of choosing these two images is that the first one, `eight.tiff`, has a large white space with objects scattered around the image while the second one, `cameraman.tiff`, has more features in the center region. By using these two images, we want to illustrate that our three methods (constant, linear, and RBF) for interpolating matrix $\mathbf{A}$ work on both scattered and dense images. The size of both images used in this section is 512×512.



(a) (b)

Figure 4.1: Original image: (a) `eight.tiff`; (b) `cameraman.tiff`.



(a) (b)

Figure 4.2: Recorded blurry image from: (a) `eight.tiff`; (b) `cameraman.tiff`.

4.1 Spatially variant separable Gaussian blur

We use the spatially variant Gaussian blur function introduced in Section 2.1, using a variety of values for α_1 and α_2.

4.1.1 Construction of variant functions

In our first example, to construct the blurring operator, i.e matrix $\mathbf{A}$, we set the default values for α_1 and α_2 such that we have more blurring in the middle. Specifically, let α_1 and α_2 be quadratic functions describing how the Gaussian function varies along the $x-$ and $y-$directions respectively. Numerically, assume that the image is discretized on the $[0,1] \times [0,1]$ grid, then the center point of the image is $(x, y) = (0.5, 0.5)$. We then set

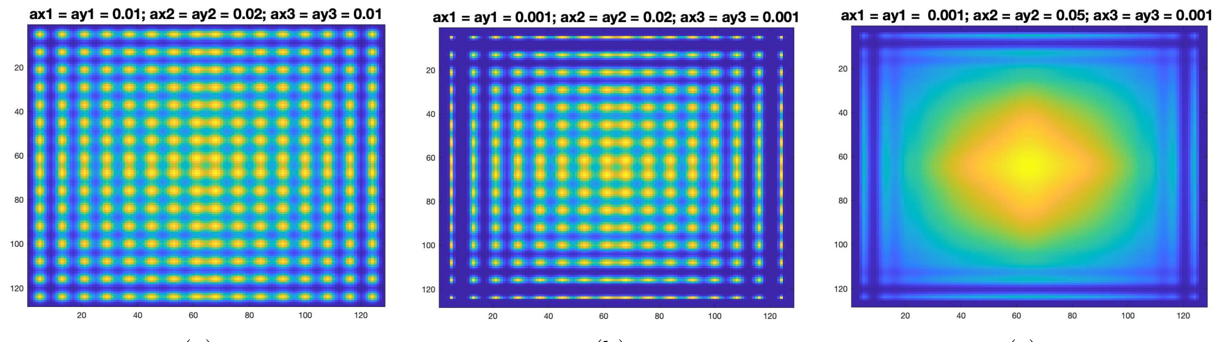
$$\alpha_1(0) = 0.001 \qquad \alpha_1(0.5) = 0.02 \qquad \alpha_1(1) = 0.001 \tag{4.1}$$

and then construct a quadratic polynomial for α_1 through these points.

Note that the greater the α_1 value, the more severe the blur is, making the image more blurry in the middle region and less blurry near the edge.

For simplicity, we set the two functions α_1 and α_2 to be identical. This is the default setting for all numerical experiments, unless otherwise specified.

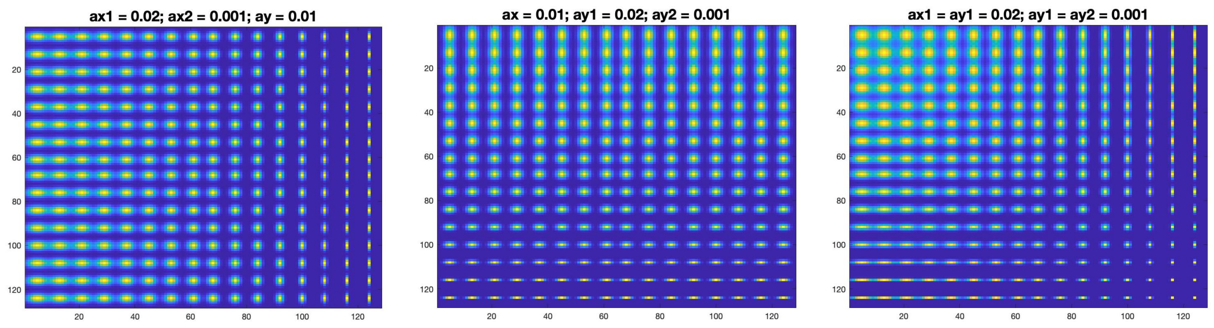
In this section, we perform several experiments with α_1 and α_2 values to show how the blur behaves with respective to these values. To illustrate, we use a true image of a grid of point sources and display the corresponding blurred images.



(a) (b) (c)

Figure 4.3: Spatially variant Gaussian blur simulation using quadratic polynomial.

Figure 4.3(a), (b), and (c) use quadratic functions α_1 and α_2 to represent the variation of the Gaussian function along the $x-$ and $y-$directions respectively. The first one, Figure 4.3(a), is the default setting of our algorithm, where most blurring concentrates in the middle region of the image, then tapering off slowly and quadratically to the boundaries. We use a Lagrange form of the quadratic interpolating polynomial through points mentioned in equation (4.1). Figure 4.3(b) and (c) are similar to (a) but in (b), the quadratic polynomial tapers off more quickly to the boundaries as we decrease α's values at the boundaries to 0.001. In Figure 4.3(c), we both decrease the boundary values and increase the value of the α functions at the center point to 0.05 to increase the blur level in the middle region of the image. It can be easily observed that the blur is most severe in Figure 4.3(c).



(a) (b) (c)

Figure 4.4: Spatially variant Gaussian blur simulation using linear function.

With a slightly different approach, Figure 4.4(a), (b), and (c) apply linear functions α_1 and α_2 to represent the variation of the Gaussian function. In Figure 4.4(a), we only change the blur variation along the $x-$direction by keeping the values of α_2 constant while setting $\alpha_1(0) = 0.02$ and $\alpha_1(1) = 0.001$. Similarly, in Figure 4.4(b), we change the blur variation only along the $y-$direction while that in the $x-$direction remains constant. Figure 4.4(c) is the combination of the two examples above, with most severe blurring concentrated in the upper left corner of the image and then tapering off linearly along the diagonal towards the lower right corner.

Lastly, it is interesting to combine both quadratic and linear functions to represent the Gaussian blur variation like in Figure 4.5, where we apply different types of functions to α_1 and α_2. Specifically, for Figure 4.5(a), we use a quadratic function for α_1 ($x-$direction) and a linear function for α_2 ($y-$direction). The blurring now mainly concentrates in the upper part of the image, then tapers off linearly along the $y-$direction. Figure 4.5(b) is the transpose of the case in Figure 4.5(a). The blurring here concentrates mostly on the left side of the image, then tapers off linearly along the $x-$direction.

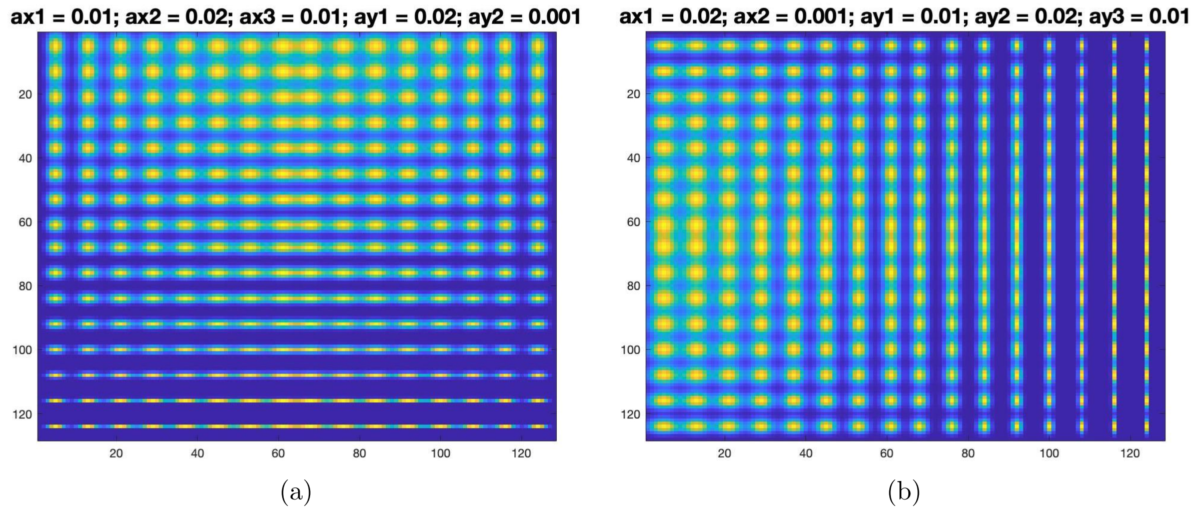


Figure 4.5: Spatially variant Gaussian blur simulation using quadratic and linear functions on the (a) $x-$ and $y-$directions; (b) $y-$ and $x-$directions, respectively.

4.1.2 Experimenting with images

In this section, we present the results from experiments with different α values and functions applied to the images `eight.tiff` and `cameraman.tiff`. Specifically, we examine the result of using the same α values as in the case of Figure 4.3(a) (only using quadratic interpolation for α_1 and α_2), and Figure 4.4(c) (only using linear interpolation for α_1 and α_2).

Firstly, when using quadratic functions for both α_1 and α_2 similar to the case of Figure 4.3(a), we obtain the following results:

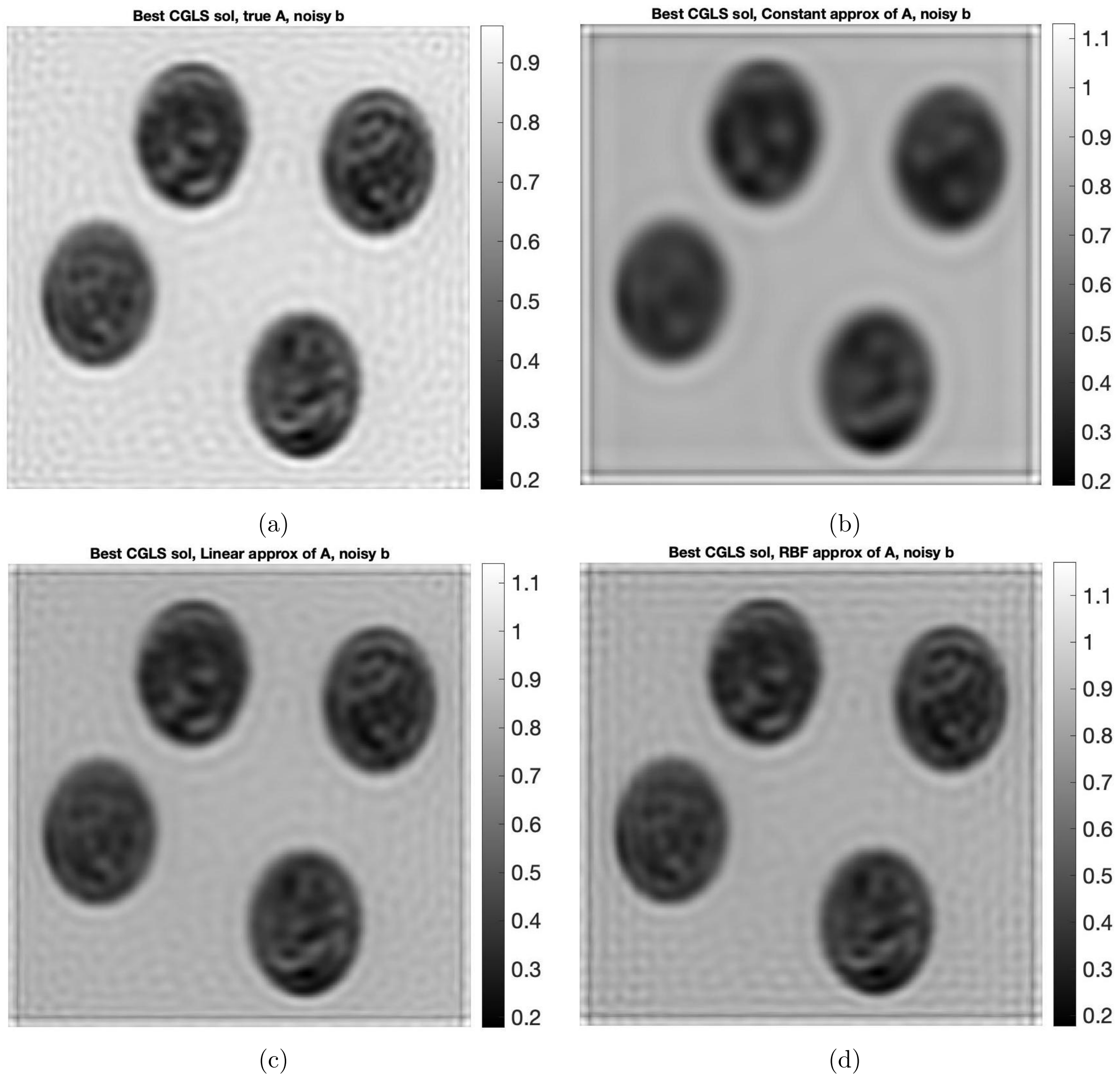

Best CGLS sol, true A, noisy b
Best CGLS sol, Constant approx of A, noisy b
(a)
(b)
Best CGLS sol, Linear approx of A, noisy b
Best CGLS sol, RBF approx of A, noisy b
(c)
(d)

Figure 4.6: The solution image for `eight.tiff` with quadratic interpolation of α_1, α_2 and (a) True $\mathbf{A}$; (b) Interpolated $\mathbf{A}$ using piecewise constant weights; (c) Interpolated $\mathbf{A}$ using piecewise linear weights; and (d) Interpolated $\mathbf{A}$ using RBF weights.

The solution in Figure 4.6(a), which uses the true matrix $\mathbf{A}$, is for reference to compare with other solutions using the interpolated matrix $\mathbf{A}$. It is clear from Figure 4.6 that the solution using piecewise linear interpolation weights produces the best image, followed closely by the one using RBF weights. Meanwhile, the image solution using piecewise constant weights is very blurry and cannot show us the coins' faces. The performance of these approaches to assign interpolation weights can be compared more easily through the plots of the relative residual norms and the error norm of the solutions in Figure 4.7. The relative residual measures how well the solution at each iteration fits the data, that is

$$\frac{||\mathbf{b} - \mathbf{A}\mathbf{x}_k||_2}{||\mathbf{b}||_2}$$

where $\mathbf{x}_k$ denotes the solution at the k-th iteration. The error norm is also calculated at each iteration as the difference between the solution of the current iteration and the true solution. These calculations are only possible because we have the information of the true image `eight.tiff` and `cameraman.tiff`.

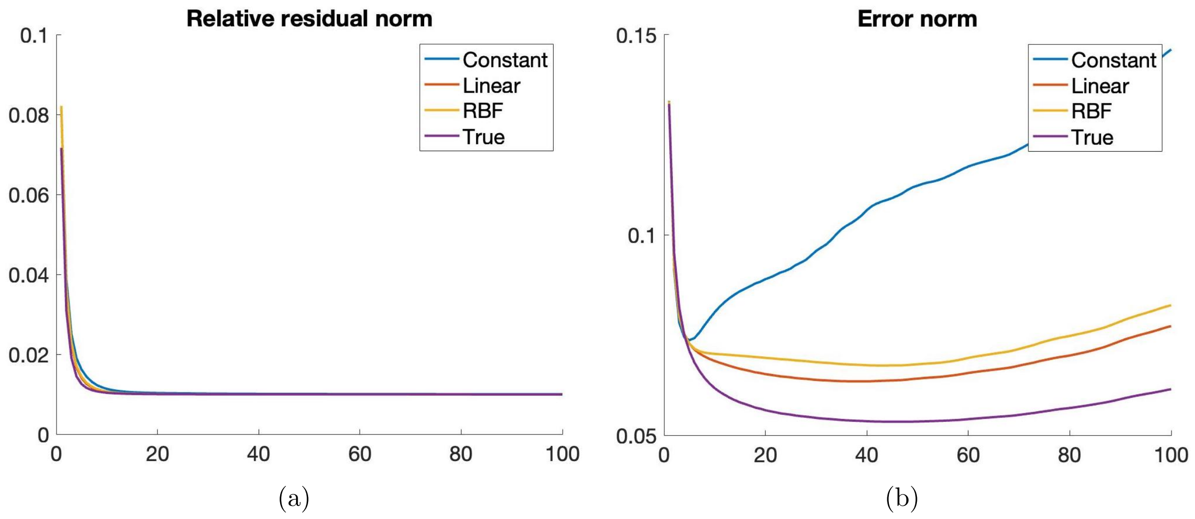


Figure 4.7: Errors for deblurring `eight.tiff` when using quadratic interpolations of α_1 and α_2: (a) relative error and (b) error norm.

Even though the relative residual norm plot (Figure 4.7(a)) cannot show the difference in accuracy between different approaches, the error norm plot in Figure 4.7(b) shows that piecewise linear weights and RBF weights perform well, both strongly outperforming piecewise constant weights.

A similar phenomenon happens when we apply linear functions to both α_1 and α_2 as in the case of Figure 4.4(c).

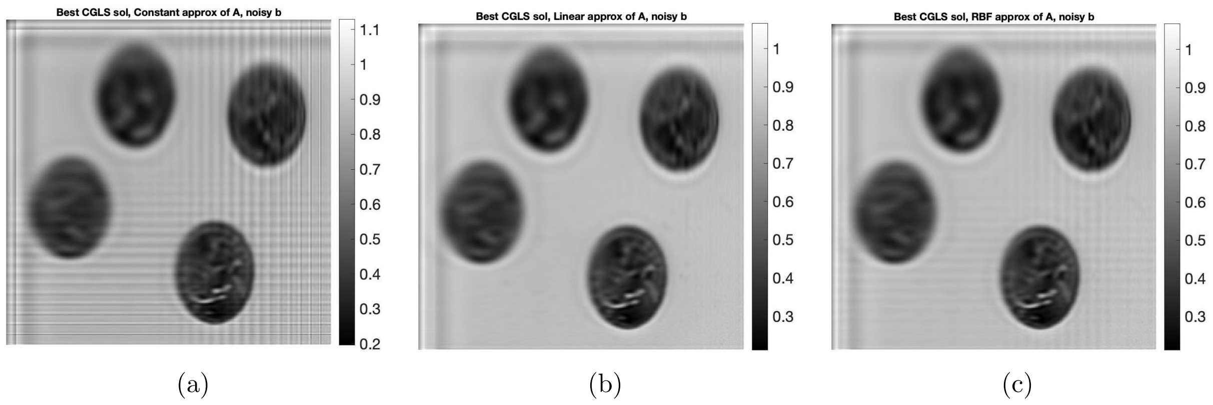

Best CGLS sol, Constant approx of A, noisy b
Best CGLS sol, Linear approx of A, noisy b
Best CGLS sol, RBF approx of A, noisy b
(a)
(b)
(c)

Figure 4.8: The solution image for `eight.tiff` with linear interpolation of α_1, α_2 and interpolated matrix $\mathbf{A}$ with (a) piecewise constant weights; (b) piecewise linear weights; and (c) RBF weights.

However, we note that in the case of the `eight.tiff` image, our interpolation weights fail to restore the white background surrounding the coins in the original image. Specifically, instead of a brighter background like in Figure 4.6(a) (where we have the true information of matrix $\mathbf{A}$), the background we obtain is a brighter shade of grey. This might be improved if we use better boundary conditions.

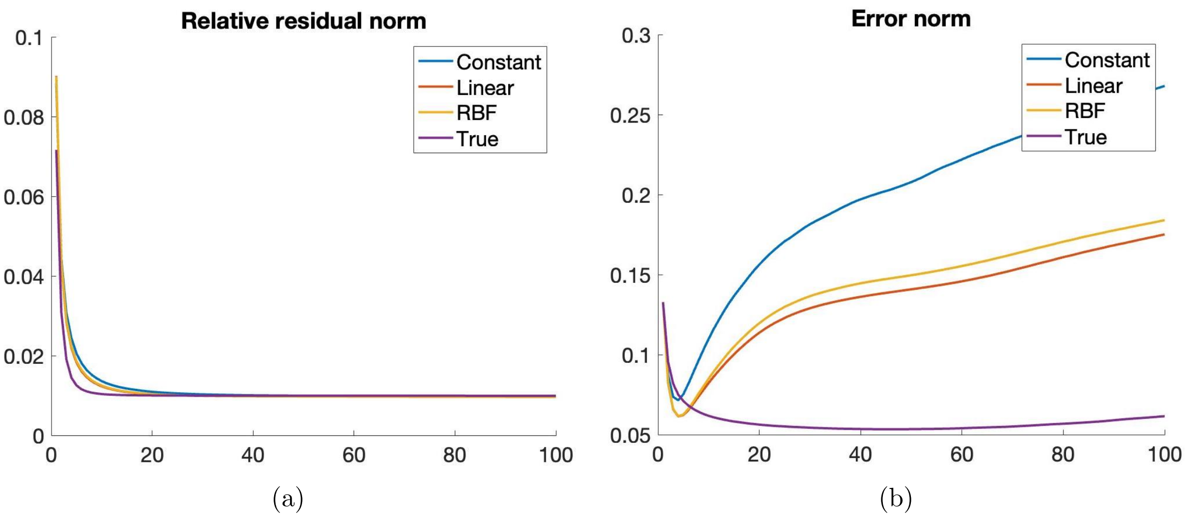


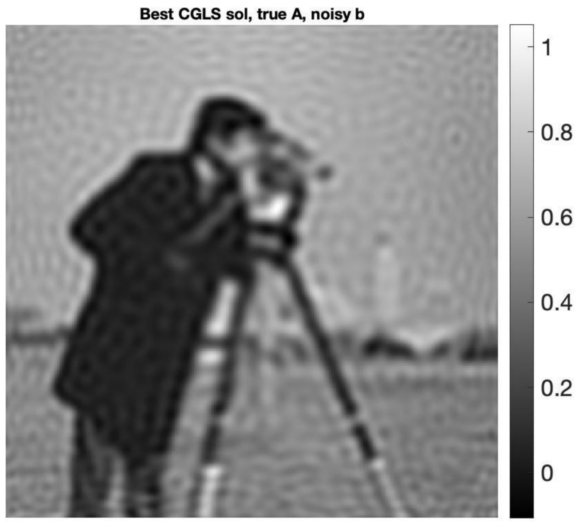
Figure 4.9: Errors for deblurring `eight.tiff` when using linear interpolations for α_1 and α_2: (a) relative error and (b) error norm.

Comparing the two results from using quadratic and linear functions for α_1 and

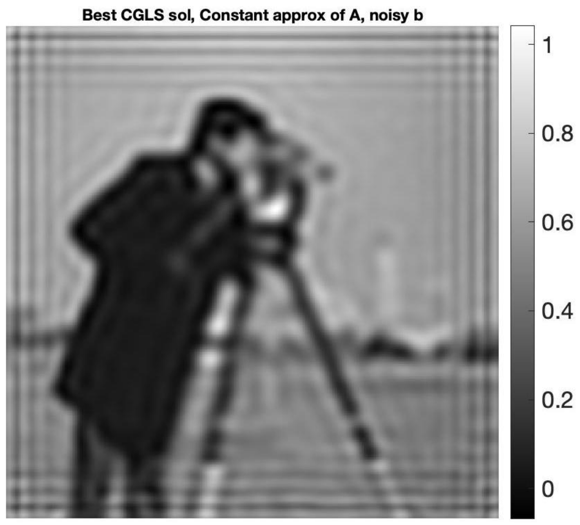
α_2, we conclude that for the case of images with scattered layouts like `eight.tiff`, using quadratic functions for α_1 and α_2 result in better image solutions.

Repeating these experiments for the `cameraman.tiff` image yield three similar main results. Firstly, piecewise linear and RBF interpolation weights consistently outperform piecewise constant weights, regardless of the type of functions used for α_1 and α_2; meanwhile, piecewise linear weights do a slightly better job than RBF weights in deblurring the image. Secondly, piecewise linear and RBF interpolation weights behave similarly in terms of error norms. Lastly, using quadratic functions for α_1 and α_2 give us better image solutions.

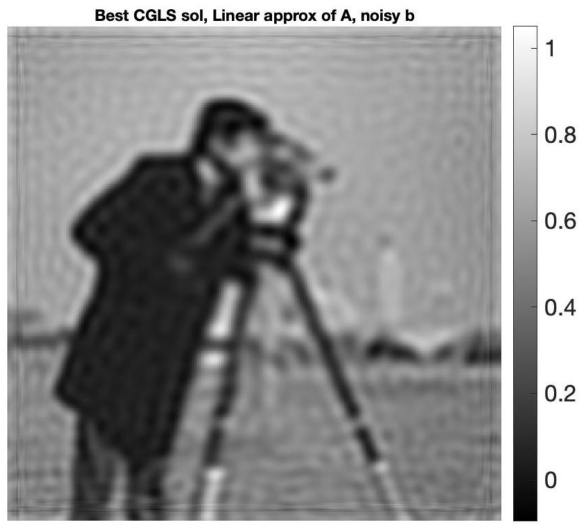
Indeed, we first apply quadratic functions to α_1 and α_2 and obtain the following results in Figure 4.10(b)-(d):



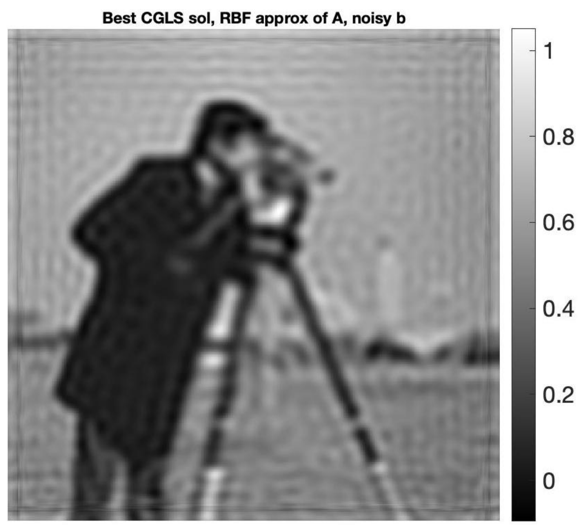
(a)



(b)

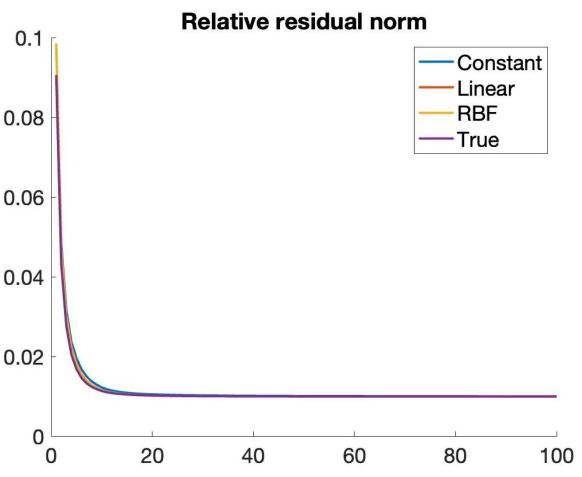


(c)

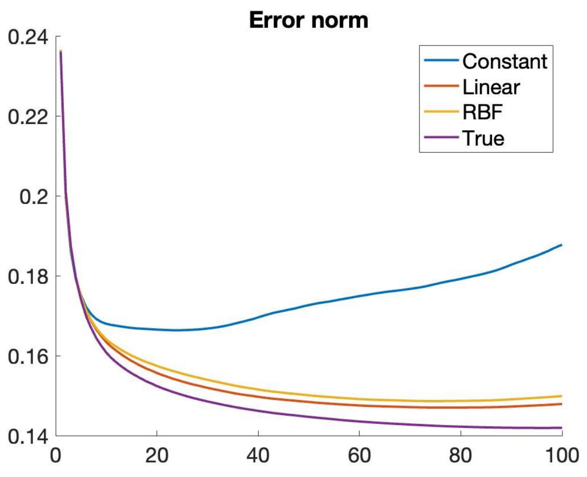


(d)

Figure 4.10: The solution image for `cameraman.tiff` with quadratic interpolations of α_1, α_2 and (a) True $\mathbf{A}$; (b) Interpolated $\mathbf{A}$ using piecewise constant; (c) Interpolated $\mathbf{A}$ using piecewise linear; and (d) Interpolated $\mathbf{A}$ using RBF weights.



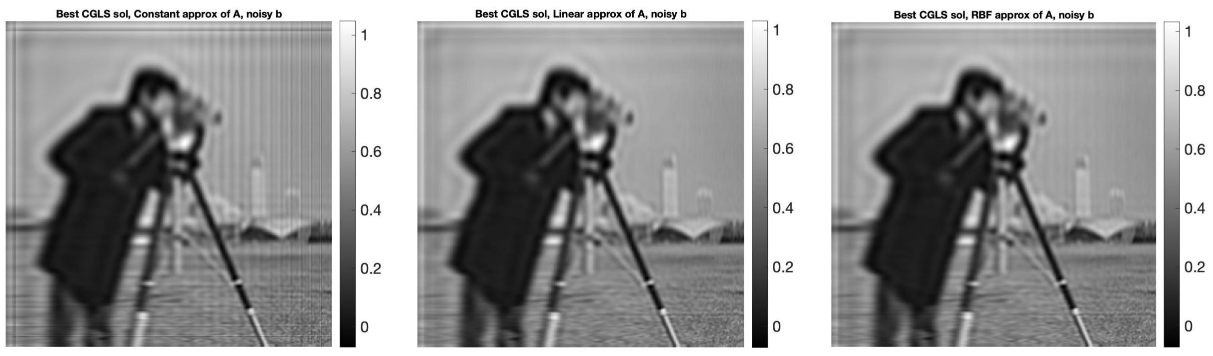
(a)



(b)

Figure 4.11: Errors for deblurring `cameraman.tiff` when using quadratic interpolations for α_1 and α_2: (a) relative error and (b) error norm.

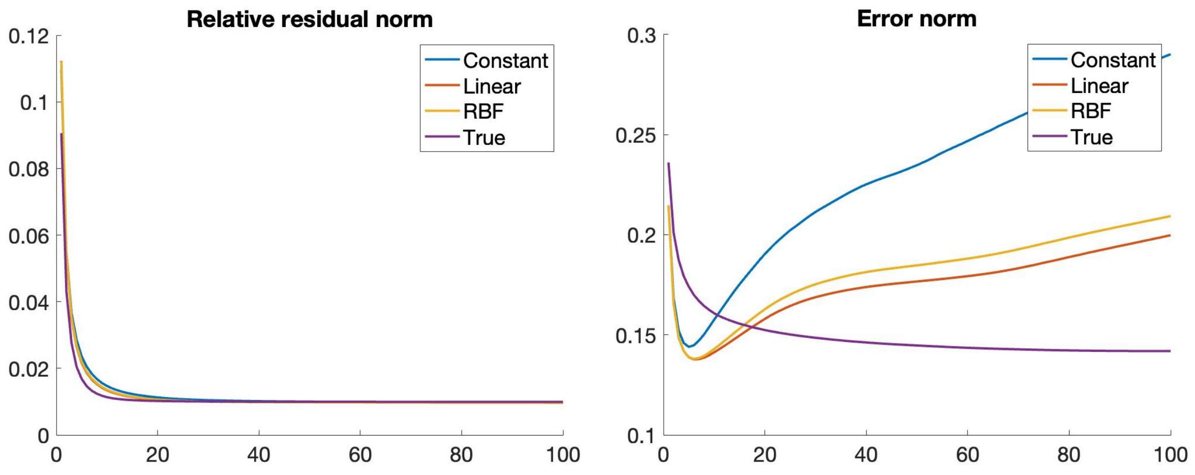
We then repeat the experiments by applying linear functions to α_1 and α_2 and obtain results consistent with those shown above.


Best CGLS sol, Constant approx of A, noisy b
Best CGLS sol, Linear approx of A, noisy b
Best CGLS sol, RBF approx of A, noisy b
1
0.8
0.6
0.4
0.2
0

(a) (b) (c)

Figure 4.12: The solution image for `cameraman.tiff` with linear interpolations for α_1, α_2 and interpolated $\mathbf{A}$ with (a) piecewise constant weights; (b) piecewise linear weights; and (c) RBF weights weights.

The corresponding error plots are presented in Figure 4.13. We notice here that for a denser image like `cameraman.tiff`, using piecewise linear and RBF interpolation weights can give us almost the same solutions visually as the one using the true matrix $\mathbf{A}$. However, we also notice from Figure 4.7, 4.9, 4.11, and 4.13 that the error norm for denser images is higher than that of more scattered images like `eight.tiff`.



(a) (b)

Figure 4.13: Errors for deblurring `cameraman.tiff` with linear interpolations for α_1 and α_2: (a) relative error and (b) error norm.

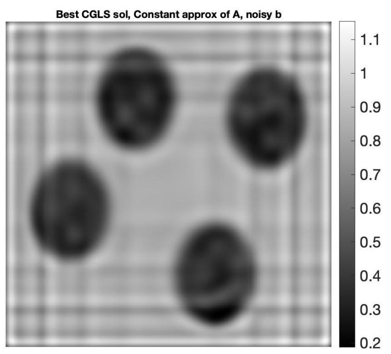
4.2 Number of partitioned image regions and its neighbor regions

An image can be partitioned into different regions for faster computation. However, it is important to pick the right number of regions as having too few regions would lead to bad approximations of matrix $\mathbf{A}$, resulting in a poor result. On the other hand, if we partition the image into many regions and only interpolate few of its nearest neighbor regions, the results may also be bad because the region size becomes too small, as discussed in Section 3.4. In this thesis, we set the default number of regions that the image is partitioned into as $n/16$ and the number of interpolated neighbor regions, Δr, as 1. For example, if the image is of size 512×512, the number of regions it is partitioned into is 32, and we interpolate over its two nearest neighbor regions, one on the left and one on the right part.

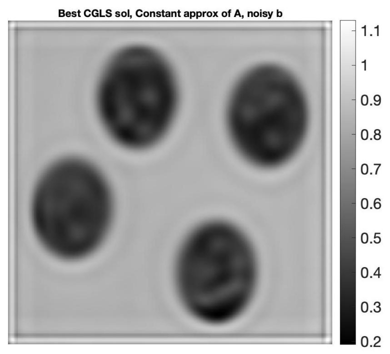
We will apply different number of regions (8, 32, and 64) to both `eight.tiff` and `cameraman.tiff` of size 512×512. Additionally, we also experiment with different numbers of interpolated neighboring regions, Δr, on the case of the image `cameraman.tiff` being partitioned into 64 image regions. We expect that being partitioned into 32 regions is enough to get a good approximation of the true image and partitioned into 64 regions with $\Delta r = 1$ would lead to bad result. However, with more neighboring regions being interpolated, the restored image would get better. It is also expected if the number of regions is equal to the size of the image with larger Δr, the solution will be the best one; however, this defeats the purpose of partitioning and computationally costly.

4.2.1 Number of partitioned image regions with $\Delta r = 1$

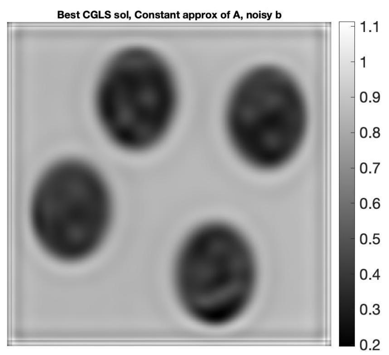
First, we look at the results with different cases of number of image regions (8, 32, and 64) applying to the `eight.tiff`.



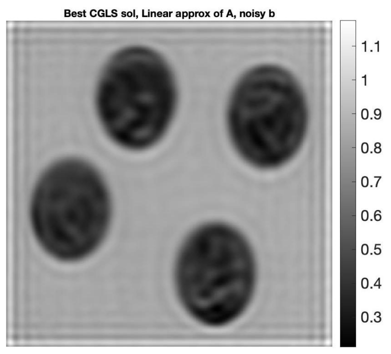
(a.1) 8 image regions



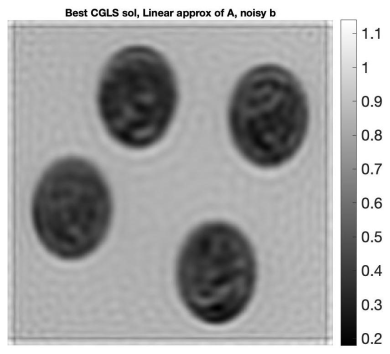
(a.2) 32 image regions



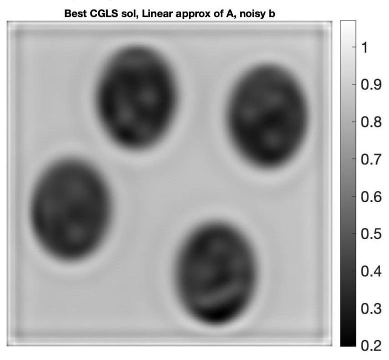
(a.3) 64 image regions



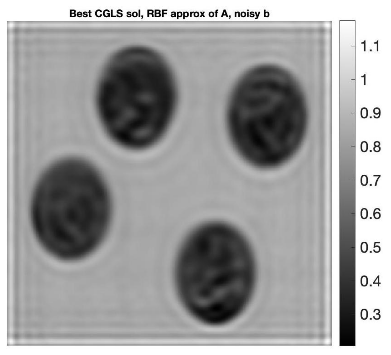
(b.1) 8 image regions



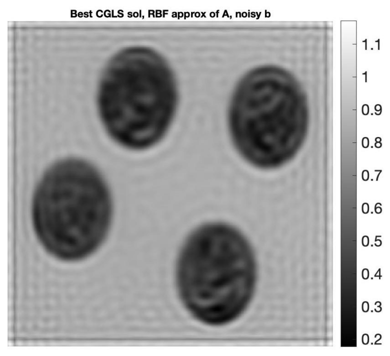
(b.2) 32 image regions



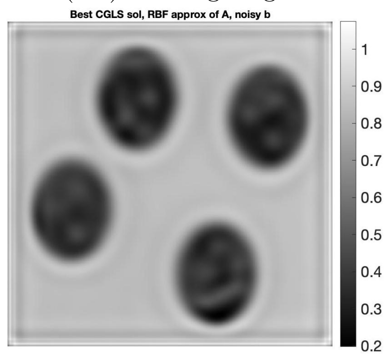
(b.3) 64 image regions



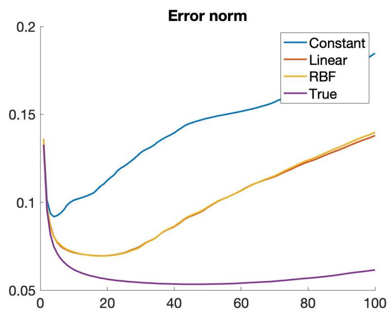
(c.1) 8 image regions



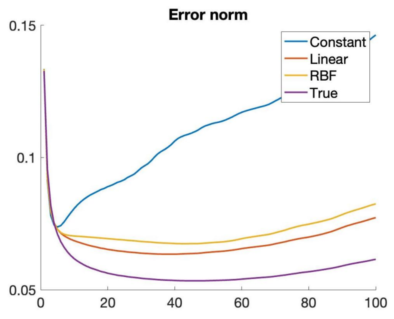
(c.2) 32 image regions



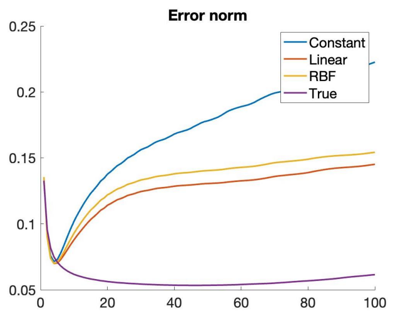
(c.3) 64 image regions



(d.1) 8 image regions



(d.2) 32 image regions



(d.3) 64 image regions

Figure 4.14: Solutions with different number of partitioned image regions: (a) Using piecewise constant weights; (b) Using piecewise linear weights; (c) Using RBF weights; and (d) Error norm.

Consistent with previous experiments in Section 4.1.2, we observed that using the piecewise linear weights produces the best result, which is followed closely by the result from using RBF weights. Meanwhile, solutions resulting from using the piecewise constant weights are significantly worse than the other two interpolated weights. Matching with our expectation that partitioning the image into more regions is not equivalent to getting a better image, Figure 4.14 shows that the result from having 64 regions is slightly worse than that of having 32 regions when $\Delta r = 1$.

Interpolation weights	8 image regions	32 image regions	64 image regions
Constant	0.0918	0.0737	0.0714
Linear	0.0695	0.0635	0.0699
RBF	0.0695	0.0674	0.0700

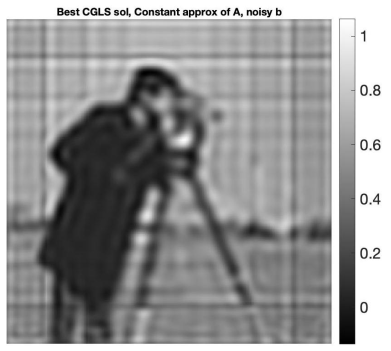
Table 4.1: Comparison of the lowest error norm for different number of image regions partitioned on `eight.tiff` with $\Delta r = 1$.

Except for the case of piecewise constant weights, Table 4.1 suggests that for the other two weights, even though increasing from 8 to 32 image regions lessens the error in our solution, increasing from 32 to 64 regions may worsen the image quality. Additionally, looking at the running time for `IRcgls` to solve our least squares problem in Table 4.2, it takes significantly longer time when the image is partitioned into more number of regions since the size of our interpolated matrices will be much larger. Specifically, increasing the number of image regions from 8 to 32 and from 32 to 64 will increase the running time by $5-6$ and $3-5$, respectively. Hence, from these error and running time information, partitioning a 512×512 image into 32 regions can give a good result within a decent time. Additionally, Table 4.4 also suggests that using piecewise linear and RBF weights to solve the problem consistently takes about $1.5 - 2$ times longer than using piecewise constant weights, regardless of the number of image regions.

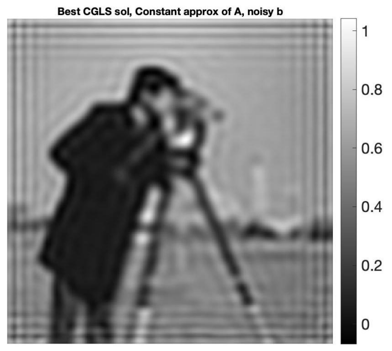
Interpolation weights	8 image regions	32 image regions	64 image regions
Constant	86.9151	502.2849	1264.1
Linear	150.7443	800.8729	2427.7
RBF	149.9406	800.3547	2149.5

Table 4.2: Comparison of running time (in seconds) for different number of image regions partitioned on `eight.tiff`.

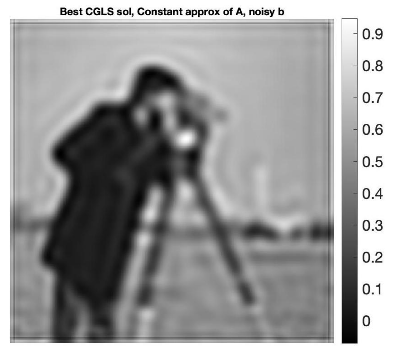
We, again, repeat the above experiment on a denser image, `cameraman.tiff`, and obtain the following results.



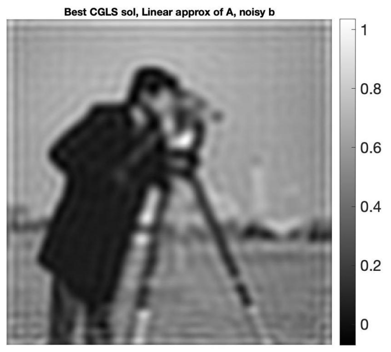
(a.1) 8 image regions



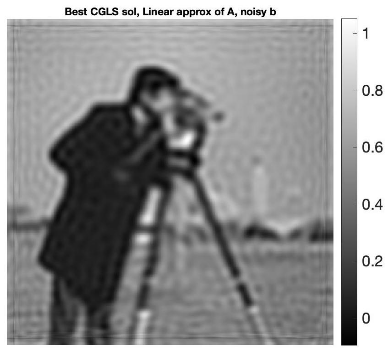
(a.2) 32 image regions



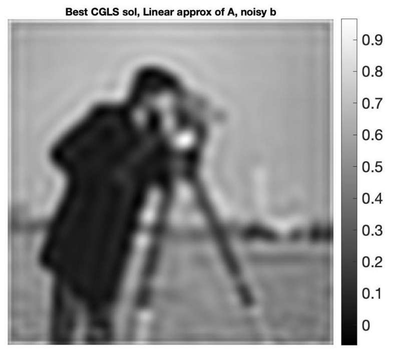
(a.3) 64 image regions



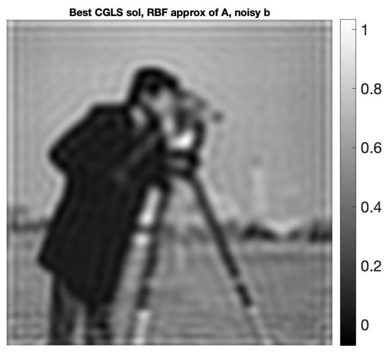
(b.1) 8 image regions



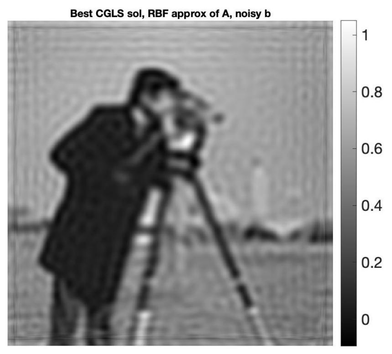
(b.2) 32 image regions



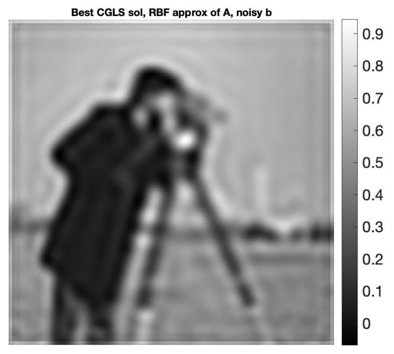
(b.3) 64 image regions



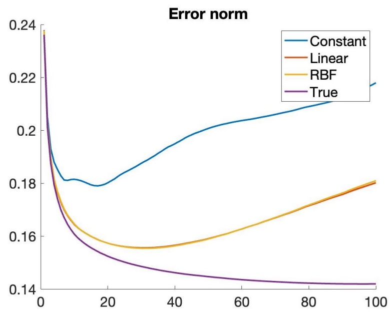
(c.1) 8 image regions



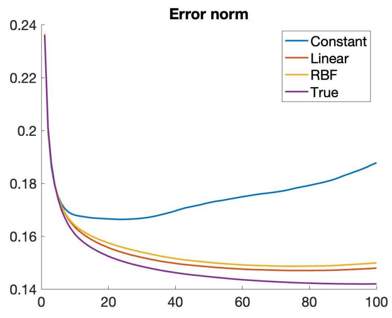
(c.2) 32 image regions



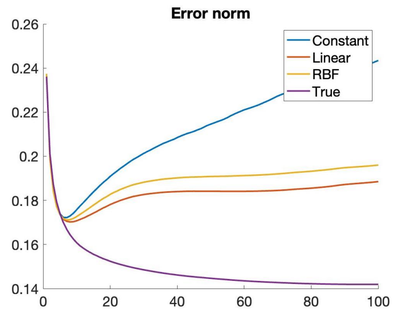
(c.3) 64 image regions



(d.1) 8 image regions



(d.2) 32 image regions



(d.3) 64 image regions

Figure 4.15: Solutions with different number of partitioned image regions: (a) Using piecewise constant weights; (b) Using piecewise linear weights; (c) Using RBF weights; and (d) Error norm.

Similar to the result from the previous experiment on `eight.tiff`, piecewise linear weights consistently produces the best result, following closely by RBF weights, and outperforms result from piecewise constant weights. Additionally, the same phenomenon happens such that error norm reduces when we increase the number of regions from 8 to 32 but that number increase when we further increase the number of regions to 64 as shown in Table 4.3.

Interpolation weights	8 image regions	32 image regions	64 image regions
Constant	0.1790	0.1664	0.1722
Linear	0.1556	0.1470	0.1702
RBF	0.1554	0.1486	0.1712

Table 4.3: Comparison of the lowest error norm for different number of image regions partitioned on `cameraman.tiff`.

The running time result is also consistent with that result of our previous experiment on the `eight.tiff` image as presented in Table 4.4.

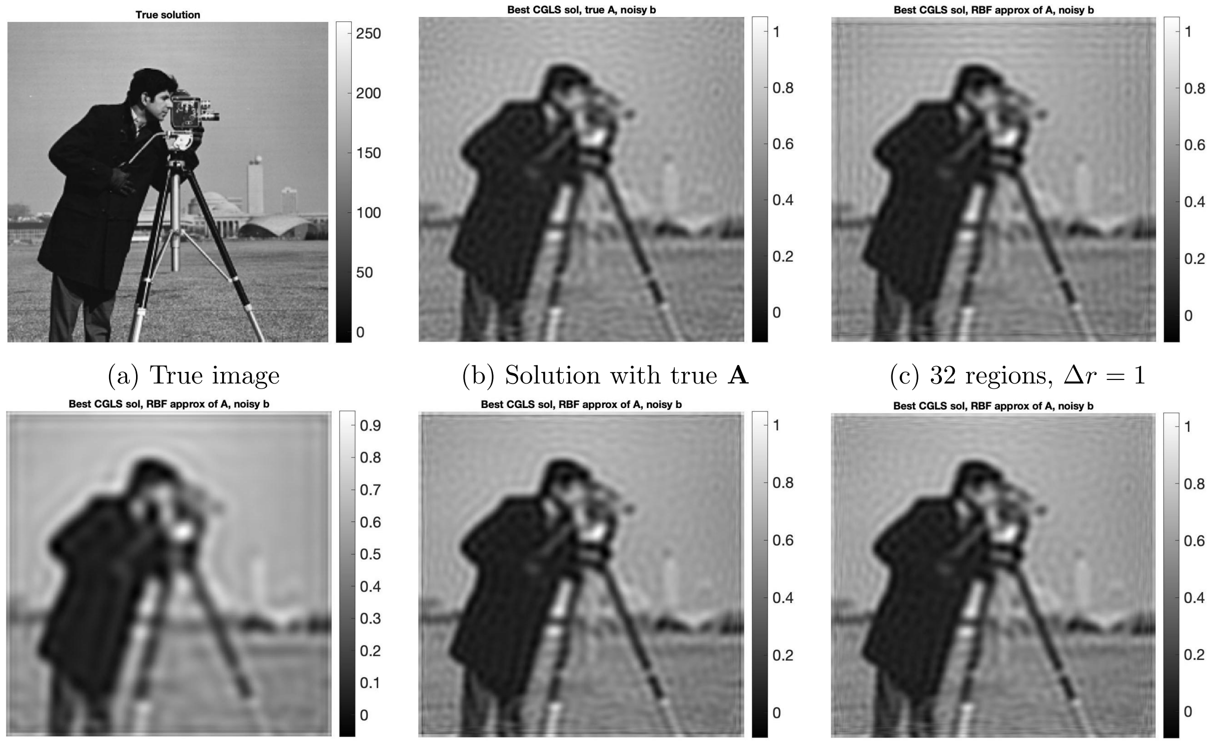
Interpolation weights	8 image regions	32 image regions	64 image regions
Constant	86.2316	500.4533	1259.8
Linear	151.2962	788.3943	2121.5
RBF	151.5769	795.4533	2118.8

Table 4.4: Comparison of running time (in seconds) for different number of image regions partitioned on `cameraman.tiff`.

4.2.2 Number of interpolated neighbor regions

As aforementioned in Section 3.4, when the image is partitioned into too many regions, the result may get worse as the region sizes are small and the PSF overlaps over multiple image regions. In this section, we experiment different number of interpolated neighbor regions ($\Delta r \in \{1, 3, 5\}$) on the 512×512 `cameraman.tiff` image being partitioned into 64 regions. Additionally, as previously stated in Section 3.4, we only conduct experiments with different number of Δr on the interpolated RBF weights.

We expect that as we increase the number of interpolated neighbor regions, Δr, the quality of the restored image also increases and gets closer to the image that we get from using the true matrix $\mathbf{A}$.


True solution
Best CGLS sol, true A, noisy b
Best CGLS sol, RBF approx of A, noisy b
Best CGLS sol, RBF approx of A, noisy b
Best CGLS sol, RBF approx of A, noisy b
Best CGLS sol, RBF approx of A, noisy b

(a) True image (b) Solution with true **A** (c) 32 regions, $\Delta r = 1$

(d) 64 regions, $\Delta r = 1$ (e) 64 regions, $\Delta r = 3$ (f) 64 regions, $\Delta r = 5$

Figure 4.16: Visual comparison between (a) True image; (b) Restored image using true matrix **A**; (c) Restored image with 32 regions, $\Delta r = 1$; and (c) - (e) Restored images with different Δr.

Matching with our expectations, Figure 4.16 shows that the image quality is improved as the Δr is increased. However, the difference is much clearer when we look at Figure 4.17 and Table 4.5, which show that the error norm of the restored image gets significantly smaller with larger Δr. More significantly, as we increase Δr, the error norm can get very close to the error norm of applying `IRcgls` on the true matrix $\mathbf{A}$ and partitioning the image into 32 regions. As shown in Figure 4.17, the error norm curve of partitioning the image into 32 regions with $\Delta r = 1$ and of partitioning the image into 64 regions with $\Delta r = 5$ is almost identical.

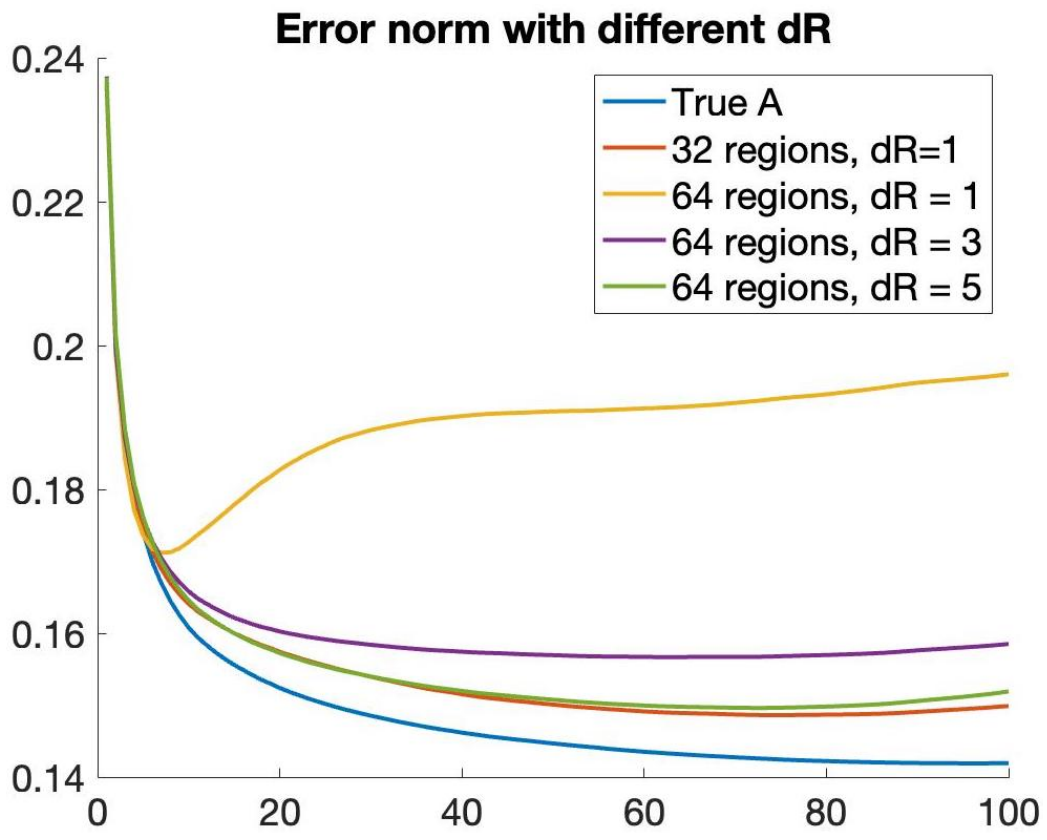


Figure 4.17: Error norm with different methods on deblurring `cameraman.tiff`.

Number of regions	Δr	Error norm	Running time
	`IRcgls` on true $\mathbf{A}$	0.1419	5.6066
32	1	0.1486	795.4533
64	1	0.1712	2118.8
64	3	0.1567	14,594
64	5	0.1496	21,981

Table 4.5: Comparison of the lowest error norm and running time (in second) for different values of Δr.

However, we also notice from Table 4.5 that the error norm for when partitioning the image into 64 regions with $\Delta r = 5$ is still higher than when partitioning the image into 32 regions with $\Delta r = 1$. Meanwhile, the running time for 64 image regions cases are all much larger than that for 32 image regions case. Therefore, we need to be careful when considering to increase the number of image regions and Δr as it can make the result worse andz be very computationally costly.

Chapter 5

Concluding Remarks

In this thesis, we use different types of functions to interpolate the weights of the blurring function: piecewise constant, piecewise linear, and radial basis functions. Even though previous literature [7] has done a successful job in using piecewise constant and linear functions to interpolate the weight matrices, they have not performed intensive tests on these two approaches. Through several numerical experiments presented in Chapter 4, we obtain the following findings:

1. Firstly, using piecewise linear functions to interpolate the weights of the blurring function consistently produces the best result, followed closely by the result from using RBF interpolation weights. Additionally, both piecewise linear and RBF weights outperform piecewise constant weights. This result is independent of the number of regions the image is partitioned into or the type of functions applied to α_1 and α_2, which dictates the variation of the Gaussian blur in the $x-$ and $y-$directions respectively. Hence, from these analyses, we suggest that when dealing with image deblurring problems, one should use piecewise linear functions to interpolate the weights of the blurring function.

2. Secondly, the choice of interpolation functions for the variation of the spatial blurs along the $x-$ and $y-$directions also affects our final result. In this thesis,

we found that when using quadratic functions to represent the variation of the blur, we are able to obtain better image restorations.

3. Thirdly, when deblurring an image, partitioning images into more regions may not lead to better final results as shown in Section 4.2. To better find a reasonable number of regions the image should be partitioned into, we should also balance the trade-off between accuracy and running time. As shown in Section 4.2, partitioning a 512×512 image into 64 image regions can be very computationally expensive and takes $2 - 3$ times longer than when we partition the image to 32 regions.

4. Fourthly, when we partition the image into too many regions, the region sizes get smaller, leading to the PSF being overlapped over multiple regions. This could lead to worse result as stated in point (3). To overcome this, we could choose to increase the number of interpolated neighbor regions, Δr; however, we need to be careful as even though the error norm is reduced, it still may not be as good as having smaller number of image regions and Δr as shown in our numerical experiments. Additionally, with larger Δr, the problem also becomes significantly more computationally costly.

5. Lastly, using piecewise linear and RBF weights consistently takes $1.5 - 2$ times longer than using piecewise constant weights, regardless of the number of partitioned image regions.

For the next step of the project, we plan to look into further implementation of different and more practical boundary conditions such as reflexive and periodic, which we do not touch on in this thesis. Another future direction is to find other interpolating functions that can produce better results than piecewise linear and RBF weights. Finally, in this thesis, we only perform experiments on greyscale images, hence, for future extensions, we also hope to expand our model to colored images.

Bibliography

[1] Apple. Apple unveils iPhone 13 pro and iPhone 13 pro max - more pro than ever before, 2022.

[2] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, 2009.

[3] S. Berisha and J. G. Nagy. Chapter 7 - Iterative methods for image restoration. In Joel Trussell, Anuj Srivastava, Amit K. Roy-Chowdhury, Ankur Srivastava, Patrick A. Naylor, Rama Chellappa, and Sergios Theodoridis, editors, *Academic Press Library in Signal Processing: Volume 4*, volume 4 of *Academic Press Library in Signal Processing*, pages 193–247. Elsevier, 2014.

[4] S. Gazzola, P. C. Hansen, and J. G. Nagy. IR Tools: a MATLAB package of iterative regularization methods and large-scale test problems. *Numer Algor*, 81:773–811, 2019.

[5] J. Kamm and J. G. Nagy. Kronecker product and SVD approximations in image restoration. *Linear Algebra and its Applications*, 284(1):177–192, 1998.

[6] J. Kamm and J. G. Nagy. Optimal Kronecker product approximation of block Toeplitz matrices. *SIAM Journal on Matrix Analysis and Applications*, 22(1):155–172, 2000.

[7] J. G. Nagy and D. P. O'Leary. Restoring images degraded by spatially variant blur. *SIAM Journal on Scientific Computing*, 19(4):1063–1082, 1998.

[8] J. G. Nagy, K. Palmer, and L. Perrone. Iterative methods for image deblurring: A matlab object-oriented approach. *Numerical Algorithms*, 36(1):73–93, 2004.

[9] P. C. Hansen J. G. Nagy and D. P. O'Leary. *Deblurring Images: Matrices, Spectra, and Filtering.* Fundamentals of Algorithms. Society for Industrial and Applied Mathematics, 2006.