IMAGE DEBLURRING - COMPUTATION OF CONFIDENCE INTERVALS

VIKTORIA TAROUDAKI
tarvic@math.umd.edu
AMSC Program, University of Maryland, College Park

Advisor: Prof.Dianne P. O'Leary
oleary@cs.umd.edu
Professor, Computer Science Department
and Institute for Advanced Computer Studies
University of Maryland

Spring Semester 2011

Abstract

Cameras often record blurred images of the original object. Restoration of the image is not a trivial procedure, and it can be very expensive for large images. In this project we are trying to efficiently compute confidence intervals for the digital values that represent the image and visualize them so that the viewer can distinguish truth from uncertainty.

1 Introduction

People always wanted to keep snapshots of their everyday life for reference at a later time or for research and educational purposes. Cavemen drew on the walls of the caves using colors made from nature. Later artists painted their houses, graves and other buildings, objects or paintings with various scenes. More recently, cameras were invented, first engraving, then analog cameras and at last digital cameras. In none of these cases is the object represented exactly in the image. But as technology progresses, the accuracy of the representation increases. Digital cameras give us very good representations of the true image but due to the procedure that the image passes through, blurring occurs. This blurring can be caused by the machine errors in transforming the image into data in the camera and from the background and the way of taking the picture. Having clear images is not a luxury. Sometimes it is a matter of life and death, like in surgeries where the doctor needs to know exactly where to operate, or in weather forecasts.

An image is divided into pixels that have values denoting the color of that pixel. A grayscale image, which we will use for simplicity, has one value for each pixel, an integer in the interval $[0, 255]$. 0 is the black color, and 255 is the white color.

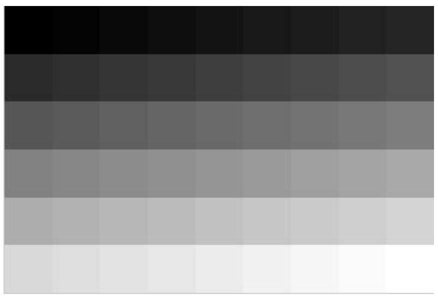


Figure 1: Part of the colorband of a gray-scale image.

Blurring occurs when a pixel value is affected by its neighbors. In this project, we will assume that this is caused by a linear transformation arising from the camera.

We will use the following notation:

Symbol	**Size**	**Explanation**
K	$m \times n$	Matrix defined through the Point Spread function (PSF) in the case of a linear problem
X		Original Clear Image
x	$n \times 1$	Vector containing the values corresponding to the pixels of the image X
B		The blurred image we measure
b	$m \times 1$	Vector which contains the values of the pixels of the blurred image B
e	$m \times 1$	Noise Vector

With the above notation, the model of the blurred image is described by the equation $b = Kx + e$, and we know that $0 \leq x_j \leq 255$, $j = 1, \dots, n$.

In general, the goal is, given the vector b and the matrix K and also given a distribution for the noise such that the mean value is 0 and the variance is a nonsingular matrix S^2, to compute confidence intervals (i.e. intervals in which the true pixel values of the object fall with a certain statistical confidence) for the quantities $\varphi_k^* = w_k^T x$ for $k = 1, 2, \dots, p$, where w_k are given vectors. If w_k is a column from the identity matrix, then we obtain a confidence interval for a single pixel.

Two different types of confidence intervals are of interest.

Definition 1 *The one-at a time confidence intervals bound each φ_k^* individually probability α ($100\alpha\%$confidence).*

$$Pr\{l_k \leq \varphi_k^* \leq u_k\} = \alpha, k = 1, 2, \dots, p$$

Definition 2 *The simultaneous confidence intervals bound all the φ_k^* simultaneously with a probability greater than α .*

$$Pr\{l_k \leq \varphi_k^* \leq u_k, k = 1, 2, \dots, p\} \geq \alpha$$

We define the S-norm of a vector y, $||y||_S$ as $||y||_S^2 = y^T S^{-2} y$

The following theorems are slight generalizations of two of O'Leary and Rust [9].

Theorem 3 *Suppose that the noise is normally distributed. Then, given α in $(0, 1)$, there is a $100\alpha\%$ probability that the true value of $w_k^T x$ is contained in the interval $[l_k, u_k]$ where*

$$l_k = \min_x \{w_k^T x : ||Kx - b||_S^2 = \kappa^2, 0 \leq x_j \leq 255, j = 1, \cdots, n\}$$

and

$$u_k = \max_x \{w_k^T x : \|Kx - b\|_S^2 = \kappa^2, 0 \le x_j \le 255, j = 1, \cdots, n\},$$

where $\|Kx-b\|_S^2 = [Kx-b]^T S^{-2}[Kx-b]$ *and* κ *is such that* $\alpha = \int_{-\kappa}^{\kappa} n(x;0,1)dx$ *where* $n(x;0,1)$ *is the normal distribution of* x *with mean value* 0 *and variance* 1.

The above theorem covers the one-at-a-time confidence intervals where each confidence interval is computed with probability exactly α.

Theorem 4 *Under the same assumptions, the probability that* φ *is contained in the interval* $[l_k, u_k]$ *is greater than or equal to* α *where*

$$l_k = \min_x \{w_k^T x : \|Kx - b\|_S \le \mu, 0 \le x_j \le 255, j = 1, \cdots, n\}$$

and

$$u_k = \max_x \{w_k^T x : \|Kx - b\|_S \le \mu, 0 \le x_j \le 255, j = 1, \cdots, n\},$$

$rank(K) = q$, $\int_0^{\gamma^2} \chi_q^2(\rho)d\rho = \alpha$, $r_0 = min_{0 \le x_j \le 255}\|Kx-b\|_S^2$,$\mu^2 = r_0 + \gamma^2$ *and* χ_q^2 *is the probability density function for the chi-squared distribution with* q *degrees of freedom.*

Other tools like those using Chebyshev's Inequality are useful for problems where the noise is not normally distributed, but this is not the main purpose of this project and so it is not examined here.

2 Approach

2.1 Point-Spread Function and Blurring Matrix K

In general, the matrix K can be experimentally measured using point spread functions for each pixel of the original image. An easy way to do this is by constructing an artificial image which contains only one white pixel (of value 255) as the target pixel, say the (i,j) pixel of the image X (or the $(j-1)\cdot m + i$ element of the vector x) and black anywhere else (value 0).

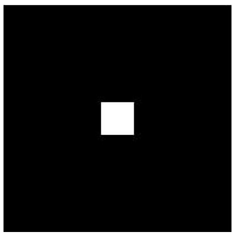


Figure 2: Example of artificial image.

We consider this as a clear image and we blur it the same way as we would blur the original image (or the vector corresponding to the original). Then, we measure the resulting blurred image B, the point spread function. The corresponding vector b is the $(j-1)\cdot m+i$ column of the matrix K.

If we know that the blur is spatially invariant, then measuring only one column of the blurring matrix K is enough to determine the whole matrix, as the rest of the columns of K are simply going to be some displacement of that one column.

If the blur is spatially variant, we need to move the source point to all the pixels of the image and measure the blur to compute all the columns of the blurring matrix, but we will not consider this case in this project. For more information, someone could consult [7].

For the purposes of this project, the blurring matrix K was constructed using spatially invariant blur and Gaussian Point Spread Functions. Usually these Point Spread Functions are of much smaller size than the original image. Let $\hat{p}$ be the size of the PSF. Then, for $k,l=1\dots\hat{p}$, define

$$PSF(k,l)=exp\Big(-\frac{1}{2}\frac{(k-c_1)^2}{s_1^2}-\frac{1}{2}\frac{(l-c_2)^2}{s_2^2}\Big)$$

where c_1 and c_2 are the coordinates of the center of the Point Spread Function which for a Point Spread Function which corresponds to the pixel (i,j) are equal to i and j respectively. In our experiments we set $\hat{p}=3$ and $\hat{p}=5$ and $s_1=s_2=3$. We also set $m=n$, making K square.

An example of this procedure is given for an image of size 5×5 and a Point Spread Function of size 3×3.

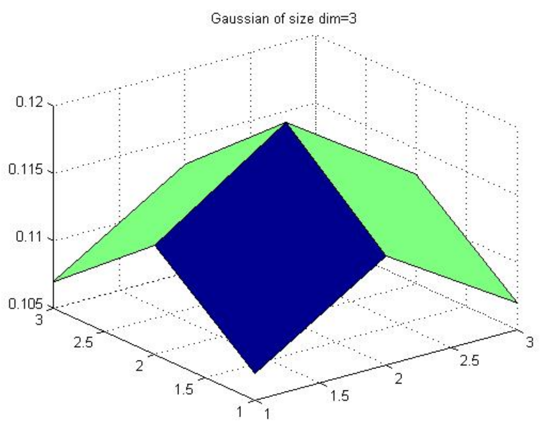


Figure 3: Point Spread Functions of size 3×3.

Let the PSF array be a matrix of the form: $\begin{matrix} \times & \times & \times \\ \times & \color{red}{\times} & \times \\ \times & \times & \times \end{matrix}$

where the red denotes the center of the matrix and the $\times$ denotes non zero elements. Then the Point Spread Function for the first pixel of the image will affect only the pixels which are immediate neighbors. Assuming 0 boundary conditions, that means that the blurred image of the artificially made image having a white color (255) at the first pixel and black (0) everywhere else will look like $\begin{matrix} \color{red}{\times} & \times & 0 & 0 & 0 \\ \times & \times & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{matrix}$
and reshaping this, column by column, we get the first column of the blurring matrix K.

As Matlab is column oriented, we count the pixels column by column, so the second pixel is the one which is in the second row but first column.

So, for the second pixel, the blurred image will look like: $\begin{matrix} \times & \times & 0 & 0 & 0 \\ \color{red}{\times} & \times & 0 & 0 & 0 \\ \times & \times & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{matrix}$
and reshaping this, column by column, we get the second column of the blurring matrix K.

If we do this for all the pixels of the image we will end up having the tri-block-diagonal blurring matrix K shown in figure 4.

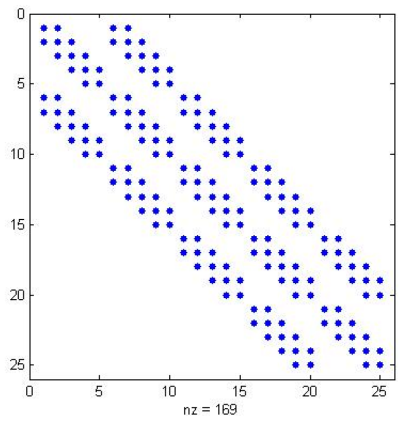


Figure 4: Blurring Matrix K for an image of size 5×5 with Point Spread Functions of size 3×3.

2.2 Blurring an Image and Constructing Noise

After the blurring matrix K has been computed, we blur the image by Kx. But in order to simulate the real case of blurred images, we need to add random noise. For this, we construct a random vector, e, with elements with mean 0 and standard deviation a specified number sdv. The variance matrix in this case is the identity matrix multiplied by the number sdv^2, so it is symmetric and invertible. The noisy blurred image thus corresponds to the sum $b = Kx + e$. To simulate truth even better, the noise differs in every run.

2.3 Computing κ^2 and μ^2

As the two types of confidence intervals need different parameters to be computed, we will discuss the two cases separately.

Simultaneous Confidence intervals

By Theorem 4, we need to compute μ^2 in order to define the ends of the confidence intervals. The rank of the blurring matrix K ($q = rank(K)$) should equal m. This can be easily verified using the Singular Value Decomposition (SVD) of K. We can find γ^2 from $\int_0^{\gamma^2} \chi_q^2(\rho)d\rho = \alpha$, with α being the desired probability that defines the confidence intervals. In the examples that follow, we use $\alpha = 0.95$ which is a common confidence level. In addition, we can compute $r_0 = \min_{0 \le x_j \le 255} \|Kx - b\|_S^2$ and finally get $\mu^2 = r_0 + \gamma^2$.

The γ^2 is computed in Matlab using the command *chi2inv(1-a,q)*. For the minimization of the norm, we use the *lsqlin* command of Matlab which performs linear least squares estimations with the constraints or the *quadprog* command which uses quadratic programming with the same constraints. To do this, the first thing we need to do is to transform the S-norm to the 2-norm that Matlab can handle. Thus,

$$\|Kx-b\|_S^2 = (Kx-b)^T S^{-2}(Kx-b) = (S^{-1}(Kx-b)^T)(S^{-1}(Kx-b)) = \|S^{-1}Kx-S^{-1}b\|_2^2$$

These matrices and vectors are manipulated by *lsqlin*. For *quadprog*, we need to modify the matrices and vectors to get the appropriate H and f that it takes as input.

One-at-a-time Confidence intervals

By Theorem 3 the one-at-a-time confidence intervals can be computed if we know the parameter κ^2. Using basic knowledge of integration and probability-statistics, the equation $\alpha = \int_{-\kappa}^{\kappa} n(x;0,1)dx$, is transformed into $\int_{-\infty}^{-\kappa} n(x;0,1)dx = \frac{1-\alpha}{2}$ or $\int_{-\infty}^{\kappa} n(x;0,1)dx = \frac{1+\alpha}{2}$. This is useful because now we can use the command of Matlab *norminv((1-a)/2)* to find $-\kappa$ or *norminv((1+a)/2)* to find κ. Taking the square of either of the two gives us the parameter κ^2.

The minimization problem is now handled in exactly the same way as in the case of the simultaneous confidence intervals.

2.4 Lower and Upper Bounds of Confidence Intervals

For the simultaneous confidence intervals, the ends of the confidence intervals, or else the bounds are given by the following equations:

$$l_k = \min_x \{w_k^T x : \|Kx - b\|_S \le \mu, 0 \le x_j \le 255, j = 1, \cdots, n\}$$

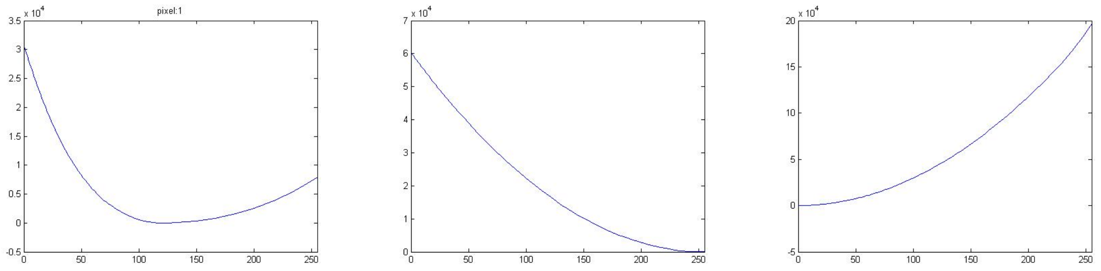
$$u_k = \max_x \{w_k^T x : \|Kx - b\|_S \le \mu, 0 \le x_j \le 255, j = 1, \cdots, n\}.$$

O'Leary and Rust ([9]) have proven the following theorem:

Theorem 5 *The values l_k and u_k are defined by the two extreme roots of $L(\varphi) - \mu^2 = 0$ where $L(\varphi) = \min_x\{\|Kx - b\|_S^2 : 0 \le x_j \le 255, j = 1, \cdots, n, w_k^T x = \varphi\}$*

The proof makes use of the convexity of the function. Also, the definition of the parameter μ assures us that the function $L(\varphi) - \mu^2$ can be negative. These two facts together and the continuity of the function gives us that there are two roots.

Three examples of such functions are those shown in the following figure.



(a) Function with two roots around 120 (b) Function with single root (c) function with two roots near 0

Figure 5: Examples of the function $L(\varphi) - \mu^2$.

Similarly, for the One-at-a-time confidence intervals the bounds are given by the equations:

$$l_k = \min_x \{w_k^T x : \|Kx - b\|_S^2 = \kappa^2, 0 \leq x_j \leq 255, j = 1, \cdots, n\}$$

$$u_k = \max_x \{w_k^T x : \|Kx - b\|_S^2 = \kappa^2, 0 \leq x_j \leq 255, j = 1, \cdots, n\},$$

Again, because of linear optimization these bounds can be computed as the two roots of $L(\varphi) - \kappa^2 = 0$ where $L(\varphi) = min_x\{\|Kx - b\|_S^2 : 0 \leq x_j \leq 255, j = 1, \cdots, n, w_k^T x = \varphi\}$ The function $L(\varphi)$ can be evaluated in Matlab by the *lsqlin* or the *quadprog* functions and transforming the data each time as described in the previous section. The main idea here was to compute a function $L(\varphi)$ that finds the minimum of the norm for all x with the constraints of the image, i.e., $0 \leq x_j \leq 255$ for all j. The output of this, the minimum norm, is used in computing $L(\varphi) - \mu^2$ or $L(\varphi) - \kappa^2$and we find the zeros. This would give us values for φ where $\varphi = w^T x$. If, as in this project, w are the columns of the identity matrix, then for each one of these vectors w, we get the value of one pixel as φ. The lower and upper bounds of the confidence intervals will then give us the lower and the upper limits of the value of each one of the pixels with the probability that defines the confidence intervals and which we used to compute the μ or the κ.

Extreme cases:

In some extreme cases, not both roots of the function $L(\varphi) - \mu^2$ or $L(\varphi) - \kappa^2$, where $L(\varphi) = min_x\{\|Kx - b\|_S^2 : 0 \leq x \leq 255, w_k^T x = \varphi\}$ lie in the interval $[0, 255]$. In a case like this, we need to choose appropriately 0 as the lower bound or 255 as the upper bound or both. If none of the roots falls into $[0, 255]$, then we choose 0 as the lower bound and 255 as the upper bound. This is the worst case that could appear as it doesn't really give any new information about the pixel. If there is a single root that falls into the interval $[0, 255]$, we define the bounds by examining the behavior of $L(\varphi) - \mu^2$ or $L(\varphi) - \kappa^2$ around the single root that falls into the interval $[0, 255]$. If the function is decreasing, 255 plays the role of the upper bound whereas if it is increasing, 0 plays the role of the lower bound of the interval. The single root is the remaining bound.

2.5 Sub-images and Sub-matrices

The blurring matrix K is defined by the point spread function. It is square, block diagonal and symmetric and has a size which is the square of the size of the original image. This matrix can be very large depending on the size of the image. For example an image of size 128×128 will have a blurring matrix of size 16384×16384. This matrix is used in the iterations and so the number of operations in the algorithm is extremely high.

To reduce the expense, we partition the blurring matrix into sub-matrices and partition the whole problem into d smaller problems which can be solved individually. These sub-problems are much easier to solve in the sense that they require fewer operations and as a result, less time. These sub-problems are independent from each other and so they can be solved simultaneously using parallel computing (for more details, see the implementation section).

Mathematically, let's consider again the problem: $b = Kx$ where K is the $n \times n$ PSF matrix, x is the vector corresponding to the original image and b the vector corresponding to the blurred image. If we want to make a sub-problem of size $r \times c$ (r rows and c columns of the sub-image defining the sub-problem), we proceed as follows. Using a matrix E with n rows and rc columns, with columns that are unit vectors corresponding to the pixels in the sub-image, and a matrix $\bar{E}$ that corresponds to the other $n - rc$ unit vectors of a matrix such that $I = [E\bar{E}]\Big(\begin{bmatrix} E^T \\ \bar{E}^T \end{bmatrix}\Big)$ we have that:

$$E^T b = E^T K x = (E^T K [E\bar{E}])\Big(\begin{bmatrix} E^T \\ \bar{E}^T \end{bmatrix} x\Big) = E^T[\hat{K}_s \hat{K}_t]\begin{bmatrix} x_s \\ x_t \end{bmatrix} = [K_s K_t]\begin{bmatrix} x_s \\ x_t \end{bmatrix} = K_s x_s + K_t x_t$$

where x_s is the vector corresponding to the sub-image. Bringing the second term of the right hand side to the left we have that $b_{st} = b_s - K_t x_t = K_s x_s$. The x_t that we need to use is a first approximation of the values of the pixels of the image, $\hat{x}$, that we can compute by minimizing the S-norm: $\|b - Kx\|_S^2$ under the constraints that $0 \leq x_j \leq 255$, $j = 1, \ldots, n$. Mathematically, $K_t x_t$ includes all the pixels of the image that are not contained in the sub-image. Practically though, because the matrix K is sparse, having zeros in a lot of places, only the neighboring pixels have an impact in the result. From now on, these neighboring pixels will be called the boundary of the sub-image. The role of the boundary will also be discussed later under the validation section.

The sub-problem we need to solve is $b_{st} = K_s x_s$.

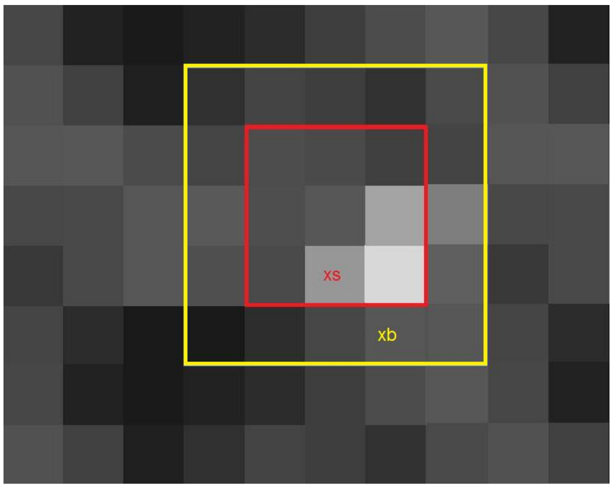


Figure 6: Sample sub-image and boundary

For symmetry with the full image model, notation for the sub-image problem is also given:

Symbol	**Size**	**Explanation**
K_s	$rc \times rc$	Part of the blurring matrix K that corresponds to the sub-image
X_s	$r \times c$	Original Clear Sub-Image
x_s	$rc \times 1$	Vector containing the values corresponding to the pixels of the sub-image X_s
B_s	$r \times c$	The part of the blurred image we measure that corresponds to the sub-image
b_s	$rc \times 1$	Vector which contains the values of the pixels of the blurred image B_s

E is a matrix of unit vectors that we choose, but in this project we will use the matrix which is part of the identity matrix corresponding to the sub-problem, i.e., if we have $x_s = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$, then we will take E to be the first two columns of the identity matrix with size $n \times n$ so that the matrix K_s is going to be 2×2.

Examples of sub-images are presented in the following figures

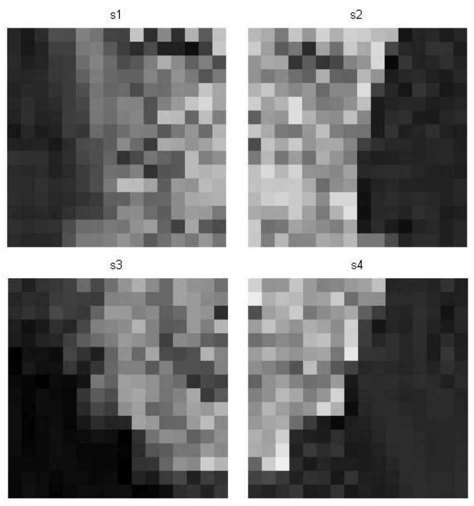


Figure 7: 4 16×16 sub-images.

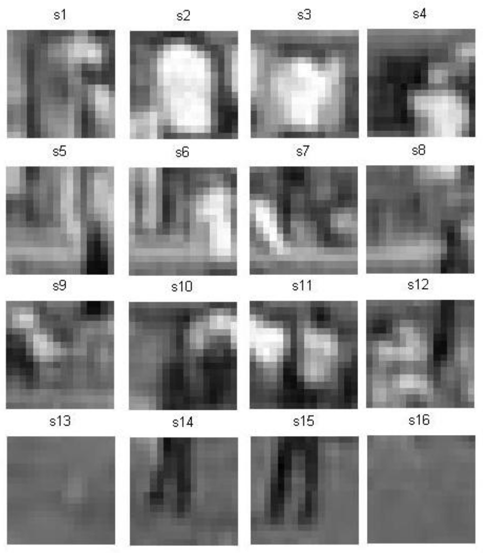


Figure 8: 16 16×16 sub-images.

The methods presented in the introduction can now be applied to these smaller problems. We compute the confidence intervals for each one of these so that we have confidence intervals for the whole image.

To display the computed confidence intervals, we can choose a uniformly distributed random value in each of the confidence intervals. We make many different samples. Then we can display these deblurred images in frames that change quickly and produce an effect which is called twinkle (Nagy and O'Leary [8]). If the confidence intervals are short, then the twinkle effect will show a somehow robust image. If the intervals are long, then we get information.

3 Implementation

The implementation of the algorithms was done completely in Matlab. The reason for this was that there are some already developed Matlab programs that solve some of the problems that appear in the algorithms and they were not the main goal of the project. These included the solution of minimization problems, the computation of the parameters μ and κ from the probabilities, "rootfinders" etc. In addition to that and more importantly, Matlab has already developed the Image Processing toolbox which includes codes which are mostly input, output and display tools.

Some commands of this type are the imshow(A) which displays the image A, the im2double(A) which converts the image A from uint8 to double etc. Some of these commands can be replaced by others which are provided by the general Matlab but then, different operations should be done to obtain the same results. For example, the command double(A) also transforms a uint8 image to a double image but then the values are from 0 to 255 whereas the values from the im2double(A) are from 0 to 1. The problem then is what we use to turn these values back into an image. When we are dealing with a grayscale image, then imshow(A) returns white for every value that is greater or equal to 1. That means that we will either use the floating point arithmetic from 0 to 1 to visualize the image or we will need to change accordingly the integer values. Similar outcomes come from the displaying tools of Matlab image(A) and imagesc(A). Appropriate values need to be used and they do not always coincide.

Furthermore, the problem involves matrices and vector operations which are easy to be handled using the available linear algebra tools of Matlab. These matrices and vectors can be of very small sizes or of big sizes depending on the original image and its size. Because of that, particular attention was used so that Matlab can control relatively big problems. For this, we had to take into account how operations are done in Matlab and how matrices and vectors are allocated in memory. Matlab cannot handle every size of initial image though. The limit is set by the Matlab memory. So one goal was to minimize as much as possible the number of variables that are saved without having to recompute them or other vectors and matrices whenever we need them. This would help us in reducing the running time and any machine and floating point arithmetic errors that may come up.

We divide the problem into smaller problems which can be solved more easily. If the blur is spatially invariant, these sub-problems involve the same matrix. These sub-problems can be solved using parallel computing. Parallel computing is useful to handle bigger images in about the same time that Matlab needs for a smaller image.

The implementation for the sub-images can be viewed by different perspectives. One way would be to use one sub-image for each pixel. That sub-image would be centered on the particular pixel. Even if the number of sub-images is the same as the number of pixels and not less than them, the number of minimization problems is the same in all the cases and so what matters is the size of the sub-image.

The idea that was followed in this project for sub-images was to check if the image was sufficiently small and, if so, deal with it as a whole. If it was classified as a big image, then it was divided into sub-images that do not overlap. That means than no two sub-images have common pixels. This procedure has some restrictions though. First the original image should be small or of size $2^{pow_1} \times 2^{pow_2}$. Whether an image is small or not is defined by a parameter that the user inserts. More precisely it means that the image is smaller or equal to the size of the sub-image we want to use. The sub-image should also be square of size $2^{pow} \times 2^{pow}$. Each sub-image could be treated individually using as boundary values the values computed by the initial minimization problem, $\hat{x}$ as stated in the previous section on Sub-images. At the end of the code, all the results are combined together to give results of the same type and size of the original image.

One may think that according to the above, that for the parallelization, we send each sub-image into a different worker. That could be a good idea if we had a lot of sub-images relative to the number of workers. But imagine an image that has one large sub-image. Then this sub-image would be worked by one of the workers and the others wouldn't help at all. Thus, the time would be similar to using the non-parallel code. This problem was overcome by parallelizing the minimizations inside each sub-image. Thus the maximum number of workers would be used and the running time would be minimized. Doing that, we can even parallelize the code when the image is small and we do not separate it into smaller ones.

The implementation and the runs of the codes were performed on a computer with only two cores, so our expected speed up is just a factor of two.

4 Databases

A big image database that can be used in Image Processing tests is the USC-SIPI Image Database which belongs to the Signal and Image Processing Institute of the University of Southern California and can be accessed here: `http://sipi.usc.edu/database/`. The database includes grayscale and color images saved in TIFF-format. They are of different sizes, 256×256, 512×512 and 1024×1024. The grayscale pictures have 8 bits/ pixel whereas the color images have 16 bits/pixel.

Also, a variety of image databases that can also be used when implementing codes that deal with images can be found through the Image Processing Place by the link: `http://www.imageprocessingplace.com/root_files_V3/image_databases.htm`

The images that have been used for the purpose of the project are grayscale images of various sizes. For initial validation of the parts consisting the main code, it was better to use small images that need less memory allocation and less running time. Bigger images produce huge blurring matrices that need care when used in Matlab. For this reason, the images that were used initially were of sizes 3×3, 7×7 and 16×16. Later, images of other sizes were also included in the database. The maximum size of the images that can be used by the code are determined by the memory that Matlab can handle in each computer. In our case, images up to 64×64 could be used.

An example of a test image is the 648×2736 firework image (figure 9) which was cropped in various sizes and used at the initial steps of validating the code. Figure 10 shows a 16×16 image.

Figure 9: The original 3648×2736 image

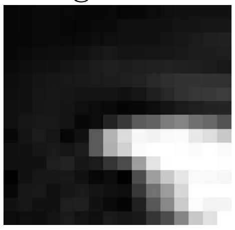


Figure 10: The cropped 16×16 image

Other images were developed by resizing a particular image, i.e., changing the analysis. The following is an example of such a created image series.

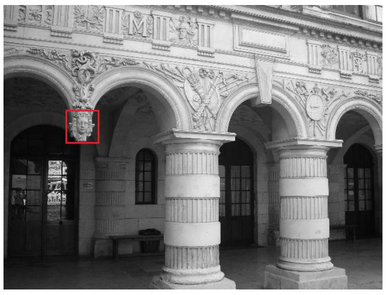


Figure 11: The original image

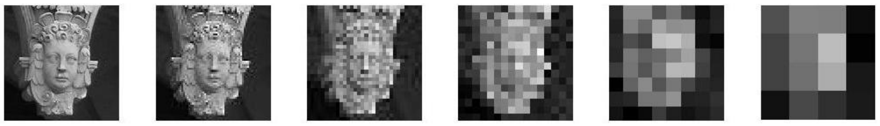


Figure 12: The "La Rochelle" image series. Sizes: 128×128, 64×64, 32×32, 16×16, 8×8 and 4×4 pixels.

5 Validation

In order to validate the code, we need to run the program using data which should give us a known or expected result as an output. By data, we mean blurred images for which we know the clear image. In detail, we take some images which are considered as the clear, original images. The confidence intervals that we compute should contain the values of the pixels of the original image with the probability that we used to compute the intervals. We blur the clear images to obtain the input for our code. Noise is also added to the blurred images to simulate reality and in order to have the symmetric and positive definite matrix S that the theory requires. We then compute the confidence intervals using our code and we count how many samples fall within the intervals.

The images that can be used for validation and testing can be taken from the databases mentioned in the database section if they are of the appropriate size that Matlab can handle or are cropped images from other bigger images that can also be obtained from these databases or from elsewhere. Examples of such images can be seen in the testing section.

Validation was done for the appropriate relation of image size and sub-image size but using for example a 64×64 image with 16×16 or 32×32 size of sub-image was forbidden by the running time. (Details on the running time can be found in the Testing section.)

The first thing we had to do was to check if all the parts that composed the code were running as expected. The main code was calling most of these subroutines individually and so these parts were run independently with input artificially constructed data for which we also knew the results we should get as output.

More specifically, to validate the construction of the blurring matrix, the spy command of Matlab was used to visualize the matrix and verify that the structure of the blurring matrix was the anticipated one. In addition, the clear image, the true blurred image and the blurred image infected by noise were shown to see the differences between them and the effect of the blurring matrix. During the construction, the spy command was also used for the Point Spread Function and the Embedded Point Spread Function matrix. This last matrix was not finally used as there was a cheaper way to construct the blurring matrix. We visualized it though to have one more way to verify the construction of the blurring matrix.

To compute the μ^2 (necessary for the simultaneous confidence intervals) we needed to compute the rank of the blurring matrix, to minimize a norm and find the γ^2 defined by the χ^2 cumulative distribution function. γ^2 was computed by Matlab using the command *chi2inv* given the rank of the blurring matrix and a probability α. The result of this was easily verified using tables of the χ^2 distribution. (The same way of verifying the result can be applied to the computation of κ which is needed for the one-at-a-time confidence intervals. κ or $-\kappa$ can be approximated by Matlab using the *norminv* and the results can be compared to the entries of the tables of the normal distribution.) Using a known matrix, the verification of the rank of the matrix was easy as we only had to use a matrix of a desired rank and compare this known rank with the computed one which should be the same. Finally, the minimization of the norm (the minimizer) was verified by checking that the solution it gave was close to the noise-free vector x.

The last and the most important part of the code is finding the lower and upper bounds of the confidence intervals. This was the main purpose of the project and it involves all of the other parts of the code that were discussed before and were validated, compared with the expected results to check any discrepancies with the theory. In order to validate the computation of the confidence intervals and the code in general, we use a known initial image as the clear image and compute the confidence intervals as described before. The main thing now is how to count the intervals that satisfy the condition imposed by the definition of the confidence intervals and the probability. Again, we will separate the description in the two confidence interval types but in both we will use confidence intervals defined by probability α ($0 < \alpha < 1$).

Simultaneous confidence intervals We run the code N times and we count how many times all of the confidence intervals include the true values of the pixels. Let this number be M. If $M/N \geq a$, then the code of computing the simultaneous confidence intervals is validated.

One-at-a-time confidence intervals In this case, we run the code N times and for each pixel we count how many times the true value of this pixel lies in the confidence intervals. For the pixel i, let this number be M_i. Now, if for **every** pixel k we have that $M_k/N \geq a$ or equivalently, $M_k \geq 100aN\%$, then the code of computing the one-at-a-time confidence intervals is validated.

The above validation techniques for the two types of confidence intervals are shown schematically in the following figure.

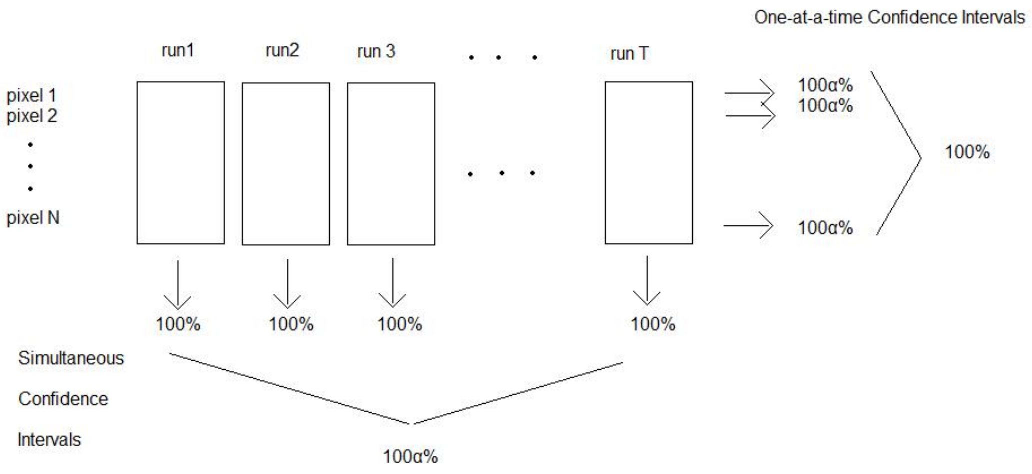


Figure 13: Validation scheme for the Confidence Intervals

In order to construct the blurring matrix we used zero boundary conditions. That means that we assume that the neighborhood around the clear image is all black. When we deal with sub-images, a boundary region around each sub-image has an effect on the sub-image, and this is why we use a first approximation of the values of the pixels by the solution of a minimization problem. The uncertainty that this introduces may ruin the method. For this reason, we should be careful and use sub-images relatively big with respect to the size of the boundary. This issue appeared during the validation process: with small sub-images, the results were not validated whereas for larger sub-images, the confidence intervals were correctly computed under the specific probability.

The formula to compute the number of pixels in the boundary (BP) given the size of the sub-image ($n \times n$) and the size of the PSF ($p \times p$) is the following

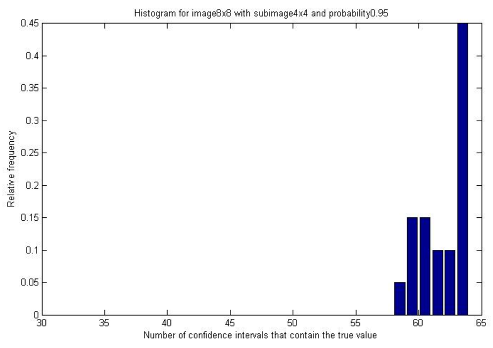
$$BP = 4\left(n + \frac{p-1}{2}\right)\frac{p-1}{2} = 2(p-1)\left(n + \frac{p-1}{2}\right)$$

The table, below, contains the number of the pixels on the boundary for several sub-image and PSF sizes.

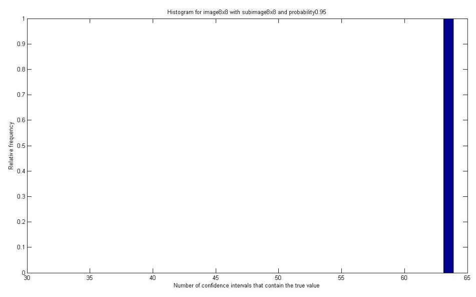
Sub-image Size \ PSF size		3	5	7	9	11	13
16	4×4	20	48	84	128	180	240
64	8×8	36	80	132	192	260	336
256	16×16	68	144	228	320	420	528
1024	32×32	132	272	420	576	740	912
4096	64×64	260	528	804	1088	1380	1680

The red numbers indicate that the boundary is larger than the sub-image. This will have a bad impact on the computation of the confidence intervals. The blue numbers indicate that for these sizes of sub-image and PSF function the code should be validated. PSFs of size 3×3 give us the greatest variety of sub-image sizes.

The following histograms show the relative frequency of the number of 95% simultaneous confidence intervals that include the true value for several runs. The first one is a histogram using 4×4 sub-images and it is clearly seen that even if the vast majority of the confidence intervals include the correct values, in 55% of the runs, there is at least one pixels for which the true value is out of the corresponding confidence interval. On the contrary, for the same level of confidence intervals, the 8×8 confidence intervals all include the true value. As this happens in more than 95% of the runs, the code is validated.



(a) Histogram with sub-images 4×4



(b) Histogram with sub-images 8×8

Figure 14: Validation Histograms.

6 Testing

The goal of this project was to implement a code for finding the confidence intervals for the values of the pixels of an image. But simply writing a correct, validated code is not enough. A good code should give the right results in a relatively short time without much cost in memory, in the frame of the problem. Several tests have been run and comparisons have been made.

Running time and storage, different Point Spread Functions and different types of error, i.e., different standard deviation matrices, were the first things that were tested. The reason for this was that the running time may depend on the size of the image, its format and the way that the values of the pixels are stored. The same things affect the memory. Also, for various types of blurring (Point Spread Function) the running time for the same image may be different due to the different blurring matrix and the minimization techniques.

All of the above were studied and compared using images constructed exclusively for this purpose.

First, we examine the effect that different sizes of Point Spread Functions have while blurring an image and the results in computing lower and upper bounds for the confidence intervals.

In Figure 15, we can see in the upper row the original 16×16 cropped fireworks image (left) with its clear blurred image (middle) and the noisy blurred image (right). Blurring occurred using the Gaussian Point Spread Function of size 5×5 and noise with standard deviation 0.1.

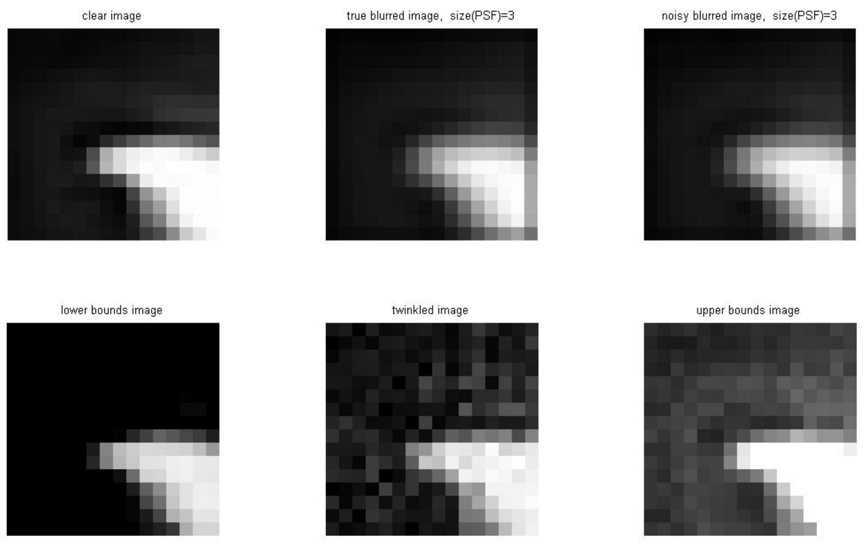

clear image
true blurred image, size(PSF)=3
noisy blurred image, size(PSF)=3
lower bounds image
twinkled image
upper bounds image

Figure 15: Upper row: Original image, Clear Blurred Image, Blurred Image with noise. Lower row: Lower bounds, random frame, upper bounds

It is not difficult to see that the original image has been faded but still it is clear where the dark or bright color is. The lower left image is the lower bounds that were computed and the lower right image is the image of the upper bounds of the confidence intervals. The image in the middle is an image that was constructed from random values for each pixel between the corresponding lower and upper bounds. This image is one of the many that produce the twinkle effect. We can see that it is not obvious to say that the recreated image is close to the original or at least closer than the blurred one. This is because 4×4 sub-images were used in the computations which as discussed in the validation section is not the best thing to do.

On the contrary, when we use the 7×7 image (upper left) in Figure 16 which was blurred using a Gaussian PSF function of size 5×5 (upper middle image) and noise was also added to it (upper right image), the new distorted image doesn't resemble the initial one. The original image had only the middle pixel white and everything else black but the blurred image is more close to a big dark gray square with a black frame.

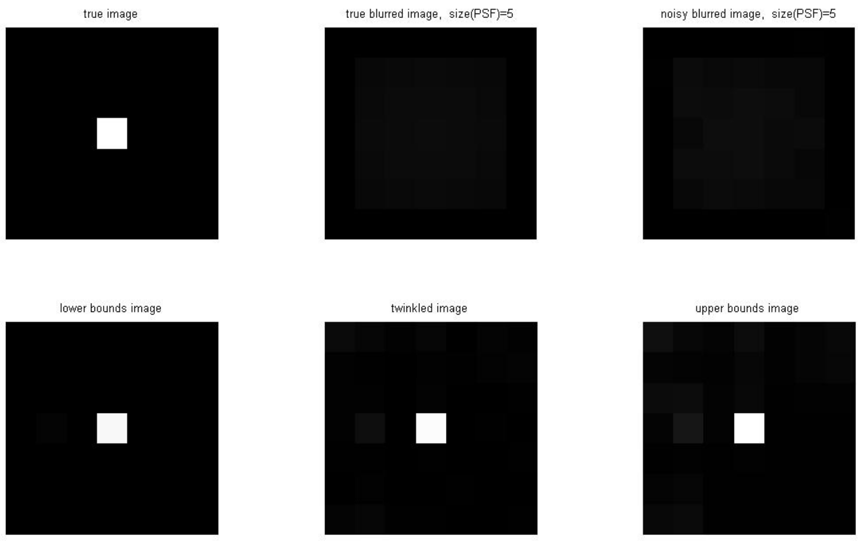


Figure 16: Upper row: Original image, Clear Blurred Image, Blurred Image with noise. Lower row: Lower bounds, random frame, upper bounds

Nevertheless, the restored image (lower middle) which lies between the lower and upper bounds is a very good approximation of the original one (upper left).

Let's now use small images that can be blurred using a 3×3 PSF and different standard deviation of the noise. A 3×3 PSF allows us to work with 7×7 images without problem. Let us first use as standard deviation matrix the identity matrix as in the domino examples. The results of the blurring and of the computation of the confidence intervals and of one frame of the twinkled image are as follows in figure 17.

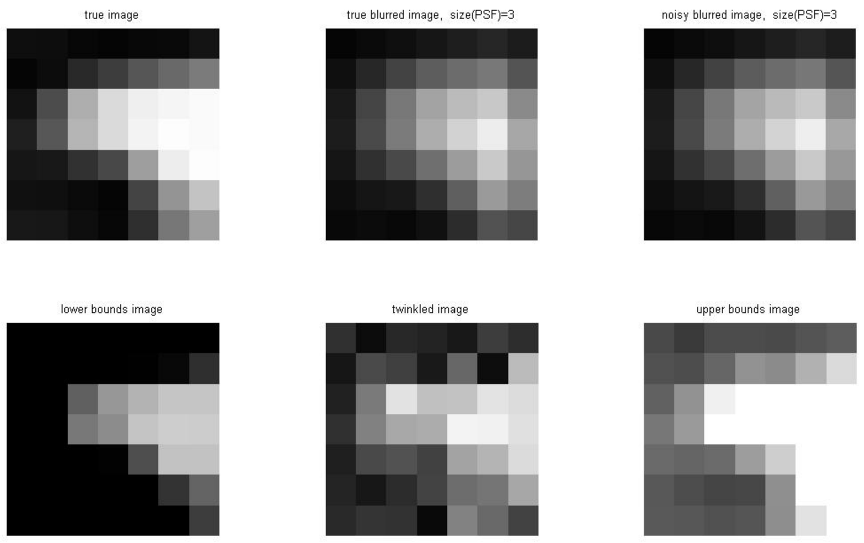

true image
true blurred image, size(PSF)=3
noisy blurred image, size(PSF)=3
lower bounds image
twinkled image
upper bounds image

Figure 17: SDV=I

To compare, we include in Figure 18 the same image blurred with less noise, noise with standard deviation $0.1I$ where I is the identity matrix.

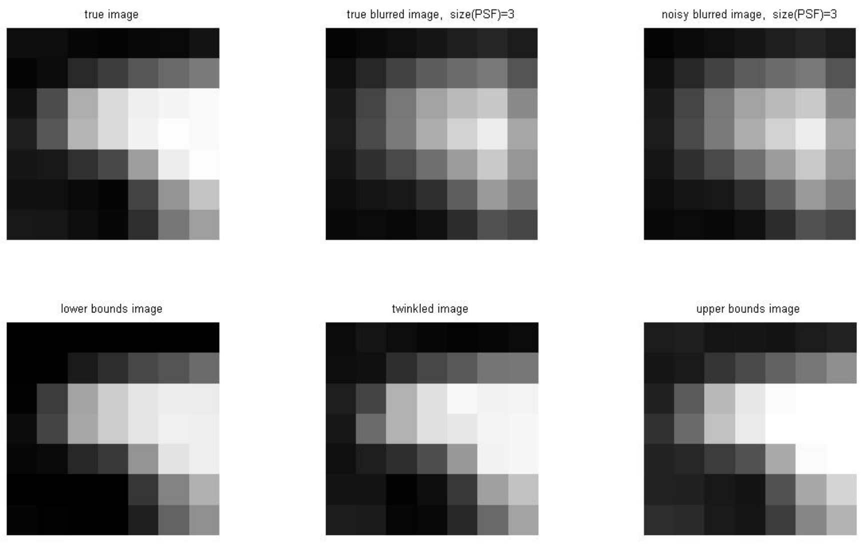

true image
true blurred image, size(PSF)=3
noisy blurred image, size(PSF)=3
lower bounds image
twinkled image
upper bounds image

Figure 18: SDV=0.1I

It can be easily seen that less noise creates less problems. The upper and lower bounds are much closer to reality and to each other and the twinkled image they produce is much closer to the original image. One can easily distinguish the truth from uncertainty.

From now on, the standard deviation matrix that will be used is $SDV = 0.1I$.

Questions arise on whether a particular pattern in the image will affect the blurring and the computation of the confidence intervals. One such image is an image of horizontal stripes.

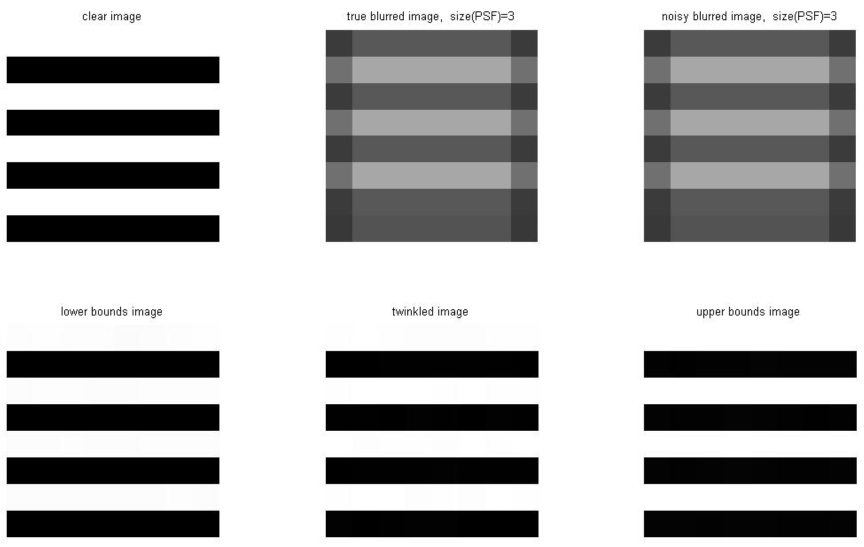

clear image
true blurred image, size(PSF)=3
noisy blurred image, size(PSF)=3
lower bounds image
twinkled image
upper bounds image

Figure 19: Horizontal stripes

What should be noticed here is that when the blurring occurs the white stripes become darker than those who were initially black. Nevertheless, the computation of the confidence intervals really gave very good approximations of the original image just by knowing the important parameters of the PSF and the general behavior of the noise.

After this example, it was the turn of a vertical line to be blurred.

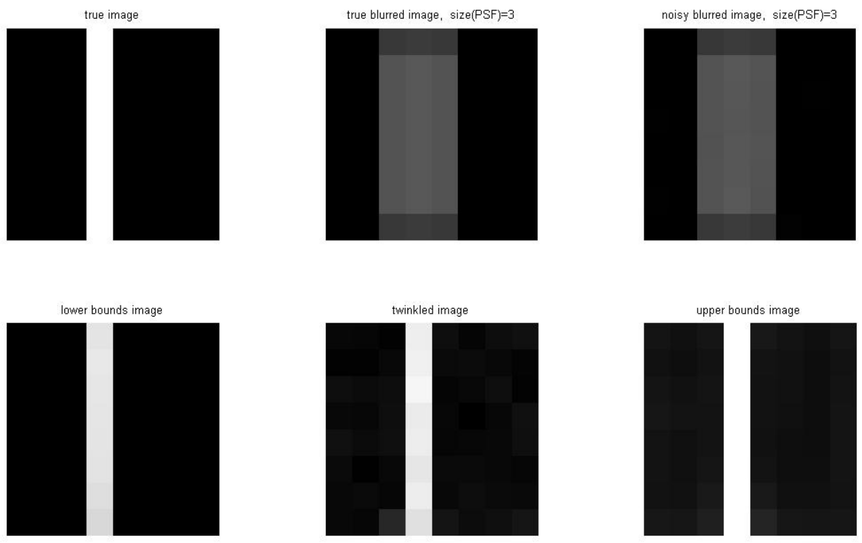

true image
true blurred image, size(PSF)=3
noisy blurred image, size(PSF)=3
lower bounds image
twinkled image
upper bounds image

Figure 20: Vertical line

The white line was expanded after the blurring and it became darker, gray. Once again, the two bounds approximate well the true image and the twinkled image looks robust.

We now include some results on bigger images so that we can use multiple methods, i.e., full image, sub-images of various sizes to compare them.

The first group of images contains the results of the computation of confidence intervals using the code that uses the image as a whole serially or in parallel.

Figure 21 shows that an image which has been blurred by a point spread function can be used to compute confidence intervals as a whole. The result of the computations is that there are two images, those of the lower and upper bounds that seem to reproduce the correct image but in a darker and lighter colors. This is good as the reconstructed image will be somewhere between.

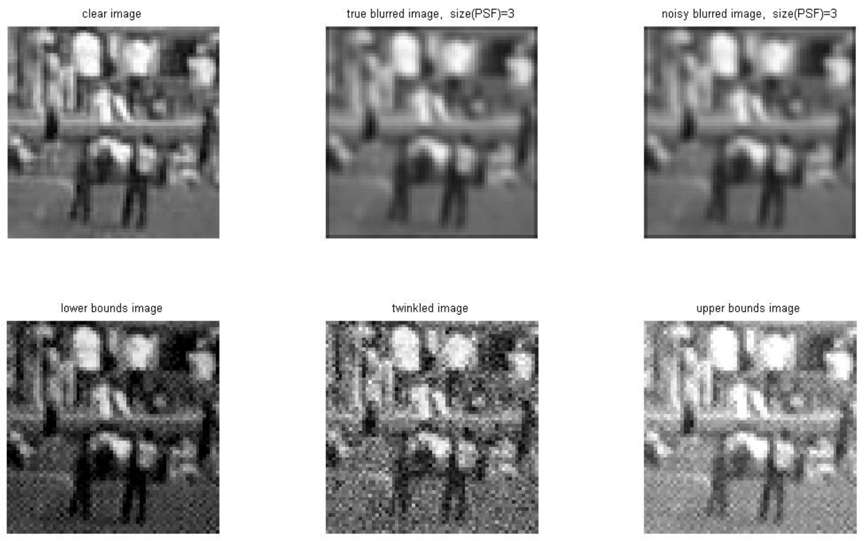

clear image
true blurred image, size(PSF)=3
noisy blurred image, size(PSF)=3
lower bounds image
twinkled image
upper bounds image

Figure 21: full 64×64 image

Figure 22 shows a 32×32 image which is used as a whole to produce a very good approximation of the original image. This will be opposed to the results that will be shown later when the image is separated into sub-images of various sizes.

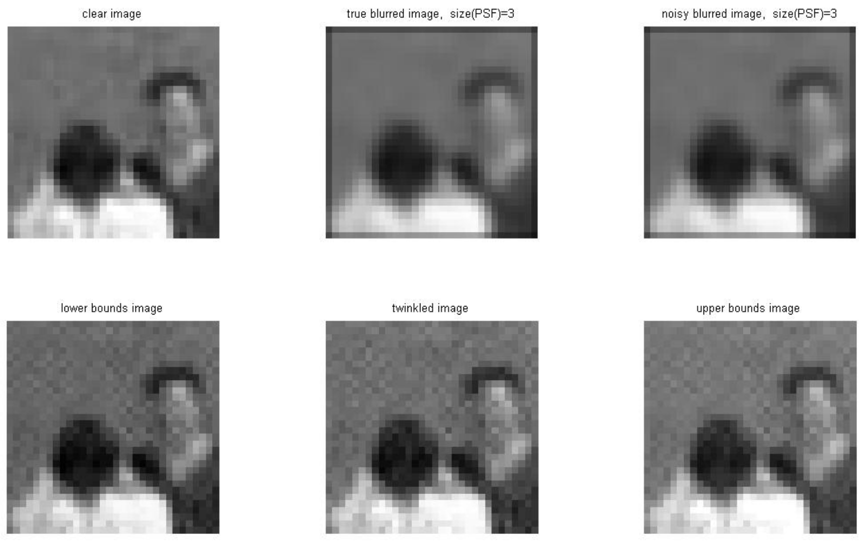

clear image
true blurred image, size(PSF)=3
noisy blurred image, size(PSF)=3
lower bounds image
twinkled image
upper bounds image

Figure 22: full 32×32 image

The following figures (23, 25 and 24) show the same image with different analysis. It is easily seen that the best reproduction is done in the biggest image (32×32).

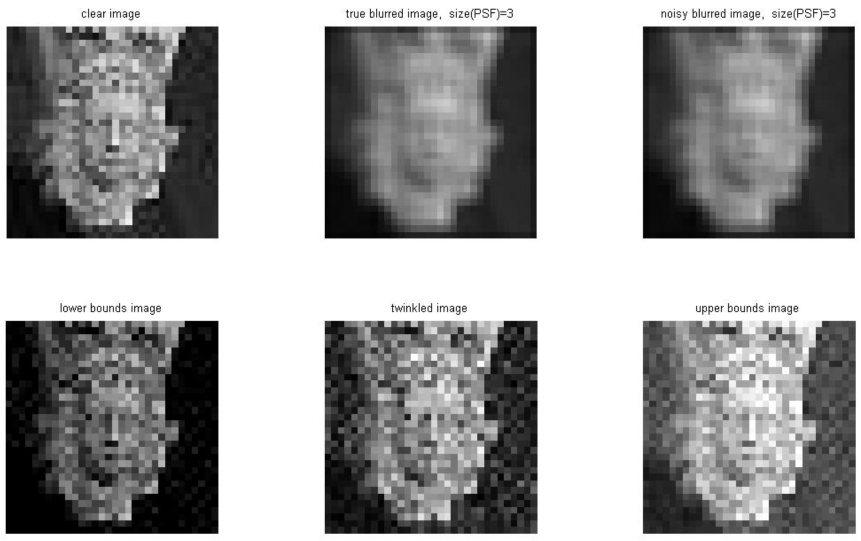

clear image
true blurred image, size(PSF)=3
noisy blurred image, size(PSF)=3
lower bounds image
twinkled image
upper bounds image

Figure 23: full 32×32 image

The small one is 4×4. Even if the image's and the boundary's sizes do not guarantee that the confidence intervals will be computed correctly, in this case they are, but they are relatively large. For example we can see from the lower and upper bounds that the image should have a dark right side but it is not easy to see if the bottom and the left side have the same colors or if the one is lighter than the other. This cannot be attributed to the fact that the blurred image looks symmetric though.

In the 8×8 image the results seem to be less robust. The lower bounds are very close to zero and the upper bounds are high enough. Probably if we reduce the confidence with which the intervals are computed, the bounds will be closer to each other but then we will not be that sure that we will enclose the real values in the intervals. This is something we have to pay for.

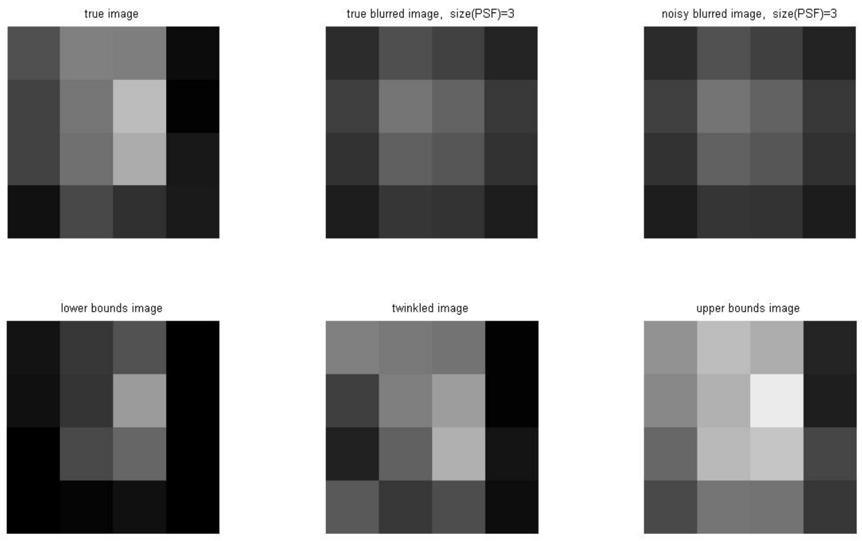

true image
true blurred image, size(PSF)=3
noisy blurred image, size(PSF)=3
lower bounds image
twinkled image
upper bounds image

Figure 24: full 4×4 image

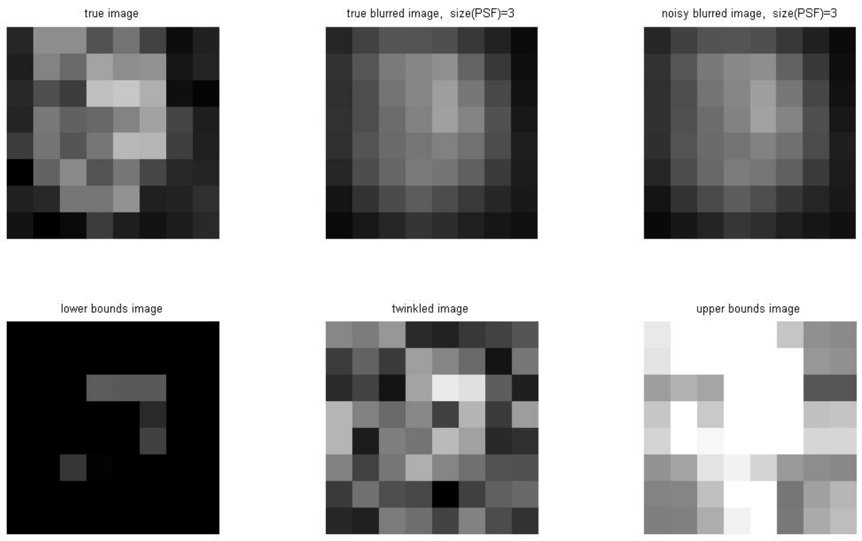

true image
true blurred image, size(PSF)=3
noisy blurred image, size(PSF)=3
lower bounds image
twinkled image
upper bounds image

Figure 25: full 8×8 image

Now, we include figures of results that were obtained using the method of sub-images again serially and in parallel.

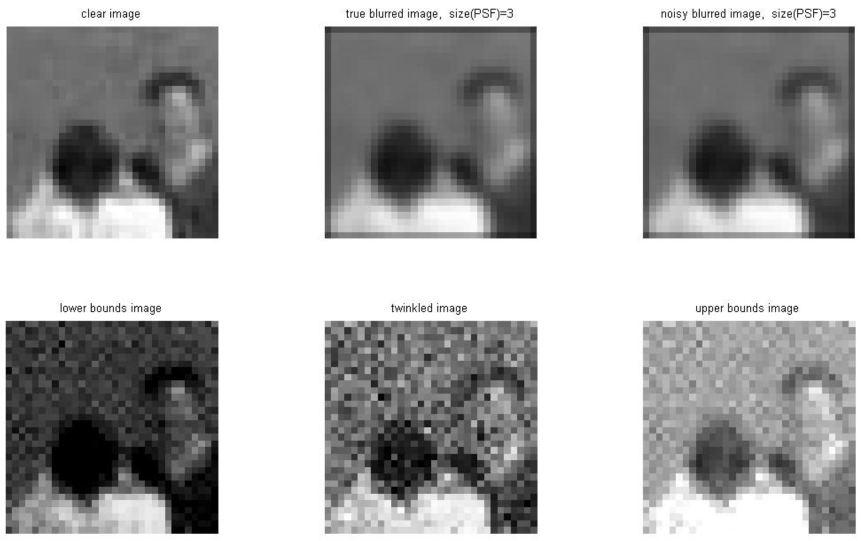

clear image
true blurred image, size(PSF)=3
noisy blurred image, size(PSF)=3
lower bounds image
twinkled image
upper bounds image

Figure 26: 32×32 image with sub-images 4×4

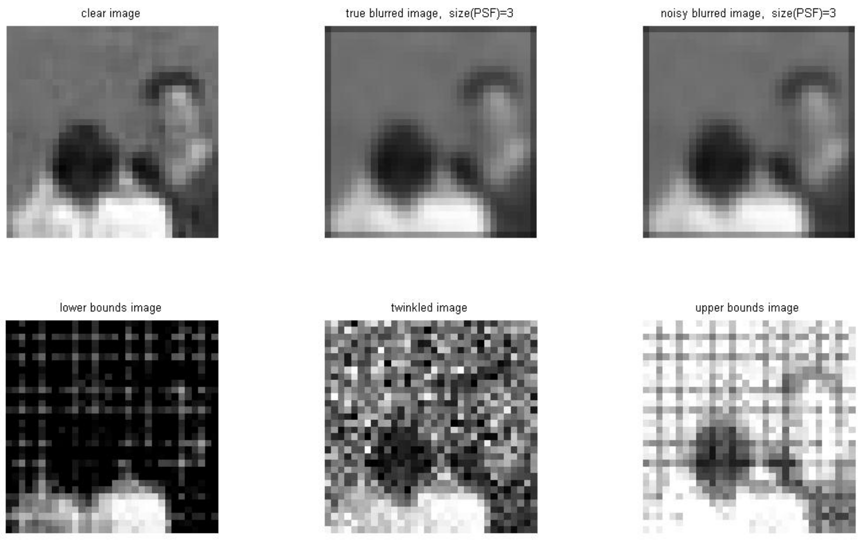

clear image
true blurred image, size(PSF)=3
noisy blurred image, size(PSF)=3
lower bounds image
twinkled image
upper bounds image

Figure 27: 32×32 image with sub-images 8×8

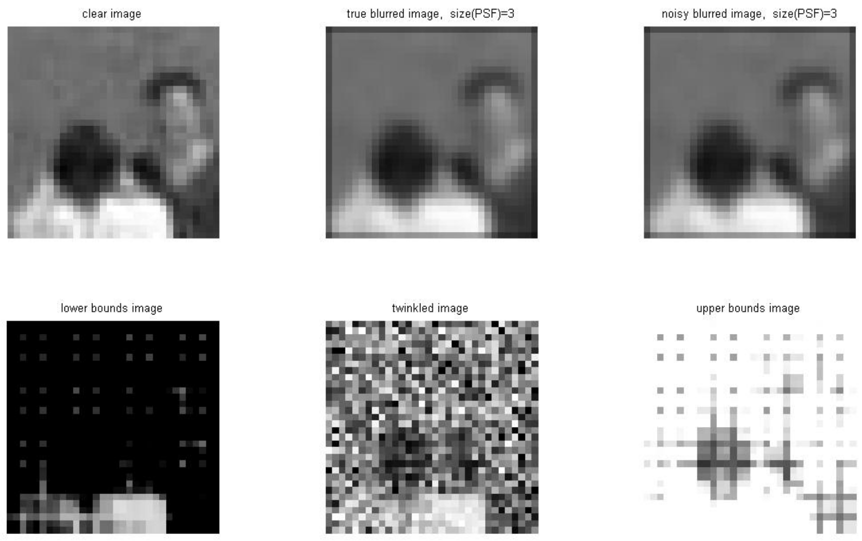

clear image
true blurred image, size(PSF)=3
noisy blurred image, size(PSF)=3
lower bounds image
twinkled image
upper bounds image

Figure 28: 32×32 image with sub-images 8×8

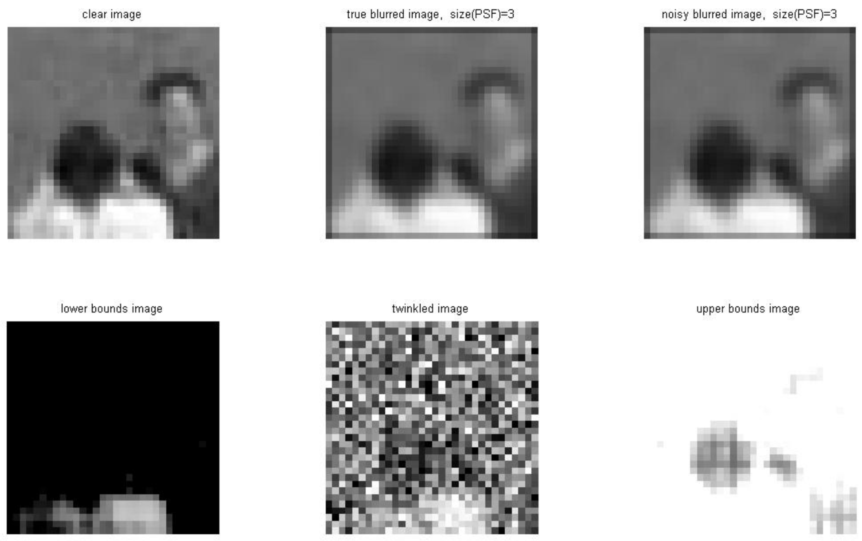

clear image
true blurred image, size(PSF)=3
noisy blurred image, size(PSF)=3
lower bounds image
twinkled image
upper bounds image

Figure 29: 32×32 image with sub-images 16×16

Figures 26, 27, 28 and 29 show the same image blurred with the same parameters and "deblurred" using sub-images of sizes 4×4, 8×8 in parallel, 8×8 serially and

16×16. Again, with the 4×4 images we cannot guarantee that the method will give us the confidence intervals that we want but here it did. Whether or not this happens depends on the noise.

It is easily seen that the results are ordered with decreasing robustness. This means that the confidence intervals are getting larger and larger. However, we can see that in all the images the person closer to the camera is easily distinguished from the background. The same does not happen for the second person. The second person is clearly seen in the first reconstructed image, less in the second and not in the other two.

This lead us believe that probably dealing with the whole image is better than using sub-images. But is that true? Further testing and comparison of the methods show that this is not that trivial.

The three figures that follow show the same image with different analysis but for which the code used had as a parameter the same size of sub-image. In this case, it looks like the most robust reconstructed image is the bigger one. And that is because it seems like there are more intervals that are short enough to depict a value for pixels close to the real one.

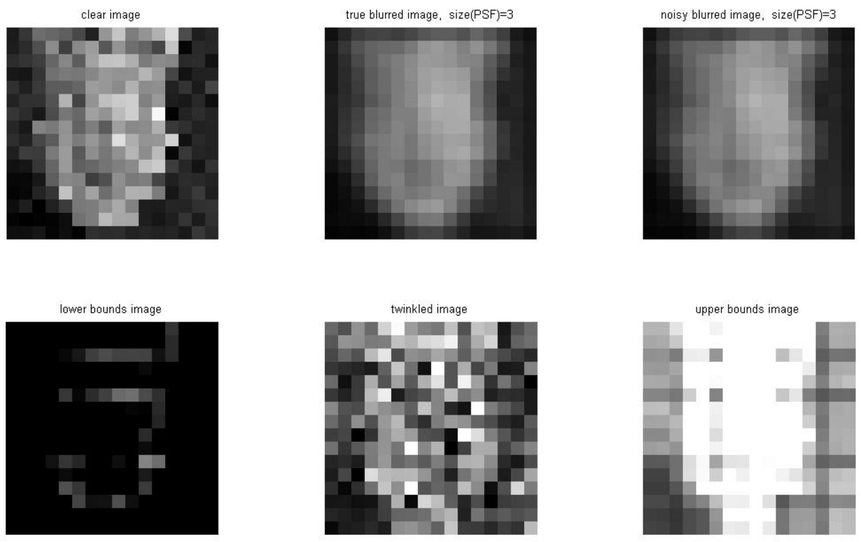

clear image
true blurred image, size(PSF)=3
noisy blurred image, size(PSF)=3
lower bounds image
twinkled image
upper bounds image

Figure 30: 16×16 image with sub-images 8×8

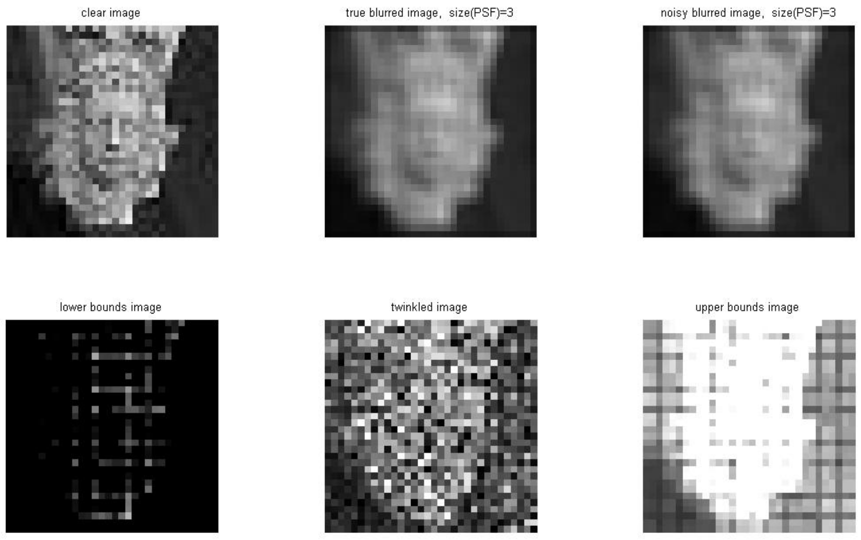

clear image
true blurred image, size(PSF)=3
noisy blurred image, size(PSF)=3
lower bounds image
twinkled image
upper bounds image

Figure 31: 32×32 image with sub-images 8×8

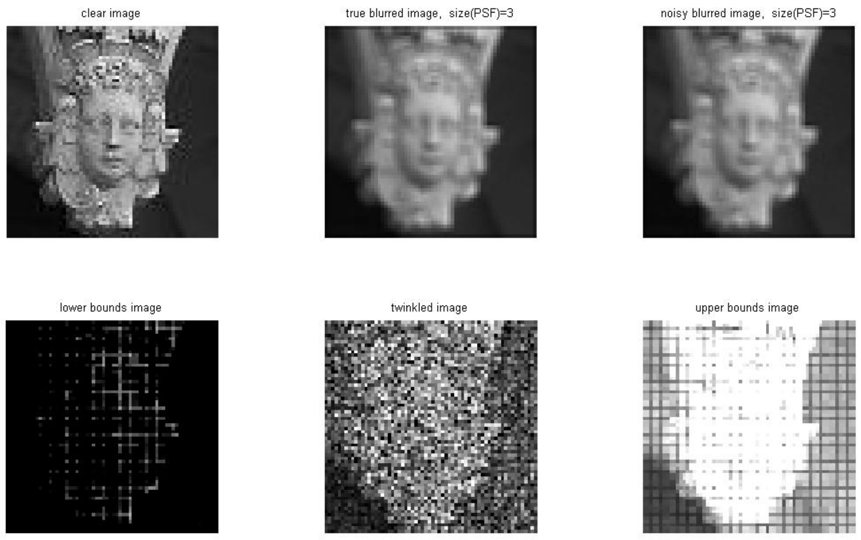

clear image
true blurred image, size(PSF)=3
noisy blurred image, size(PSF)=3
lower bounds image
twinkled image
upper bounds image

Figure 32: 64×64 image with sub-images 8×8

Finally, we discuss the running time of the codes. This is done in two ways. First, we increase the size of the image to see how the time changes and second, for a particular image size, we change the number of the sub-images used. The codes that are compared are the serial code that uses the image as a whole, the parallel code that uses the image as a whole, the serial code that separates the image into sub-images, and the parallel code that separates the image into sub-images.

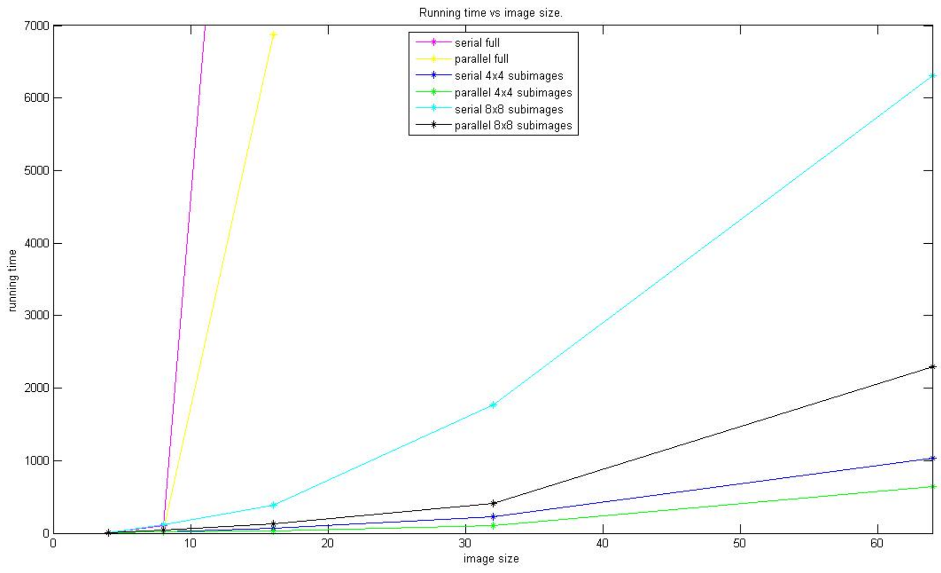


Figure 33: Running Time - Size of Image

The first plot (Figure 33) is time as a function of the size of the image (number of columns or rows). We include two sub-images of size 4×4 whenever that is validated, and 8×8. The time was the average of several runs that were made with the same parameters.

It is clear that with every method shown here, the time increases as the size of the image increases. The slowest methods seem to be those who deal with the image as a whole whereas the fastest are those with small sub-images. In each pair of methods, the parallel one is faster as expected.

This increase is not linear, and it depends on the number of operations in the code. The main source of operations, that changes with the different methods, is the minimization of the norm which is implemented using the linear least squares method using the command *lsqlin*. This is an iterative method that uses the blurring matrix corresponding to the whole image or to the sub-images. For a matrix of size $m_1 \times m_2$, the number of operations needed is of order $O(m_1 m_2^2)$. In this project, this matrix is square. In the case of a whole image, this matrix is $n \times n$ whereas in the case of the sub-images, it is $rc \times rc$, with $r = c$. Thus, the operation count is $O(n^3)$ and $O((rc)^3)$ respectively. This minimization is done once for every pixel in the image. So, in total, we have that for the whole image the operations are $O(n^4)$, and for the sub-images they are $O(n(rc)^3)$.

For the plots, we have used the length or the width of the image as the size of the image, say N. That means that for the whole image $n = N^2$ and for the sub-images we also used $r = c$, as mentioned before. With this, we can alternatively see the number of operations as $O(N^8)$ and $O(N^2 r^6)$. Thus, it is justified why the least time is needed for the smallest sub-images and why time for the whole image increases so rapidly.

If we only look at the time needed for the computation of the confidence intervals for an image 8×8 (Figure 34), we can see that the above is not true all the time. In a 8×8 image, using sub-images of size 8×8 is the same as working with the image as a whole. This code requires some more time though because it has to check whether or not the sub-images are applicable to the image and then use some time to put them in the correct place. For this reason, in this case the serial code for the 8×8 sub-images is the slowest and the immediately next is the serial code for the full image. The fastest again is the parallel code using sub-images of size 4×4.

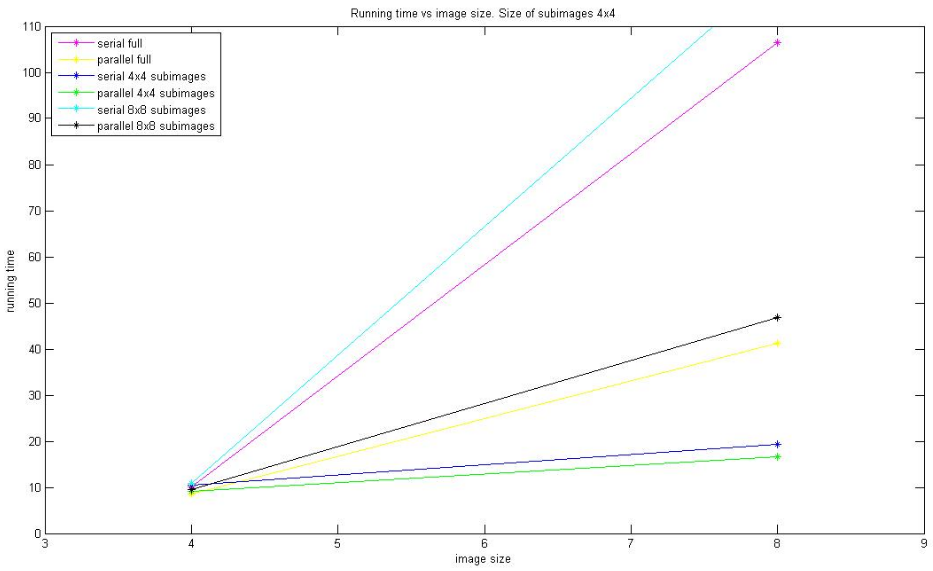


Figure 34: Running Time - Size of Image

To examine further how this increase began, we look at the smallest image we have tested, the 4×4, when all the methods work with the same manner but the one for the full image has less operations with logical expressions and rearrangements. Thus we expect that the method with the full image will work the fastest and that of the 8×8 sub-images the slowest. This is indeed what happens and the same order appears for both the serial codes and the parallel with all the parallel being faster than the serial.

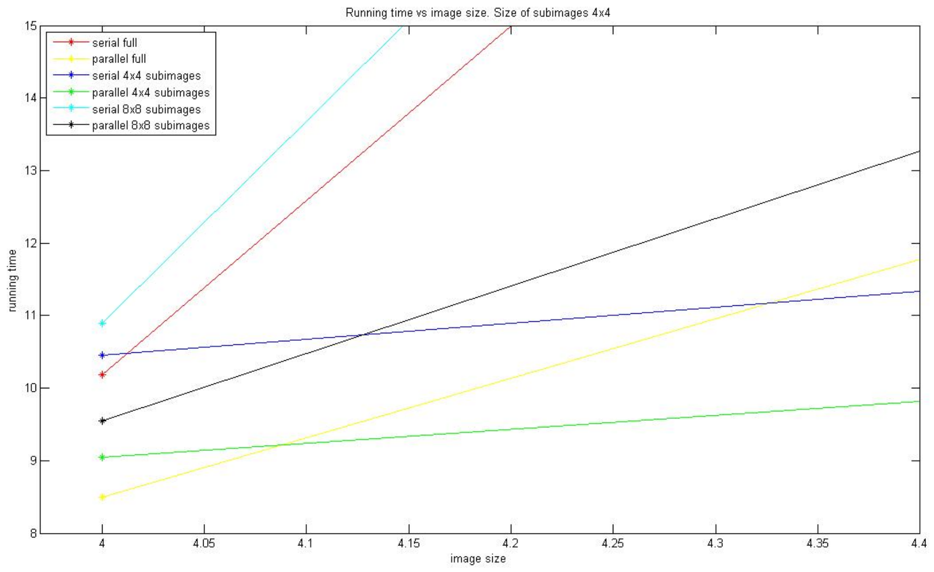


Figure 35: Running Time - Size of Image

The last figure is split into two sub-figures which are equivalent. The first shows the running time with respect to the size of the sub-images and the second with respect to the number of sub-images used. To create these plots, an image of size 32×32 was used and so the sub-images had size 4×4, 8×8 and 16×16. Using the whole image would even more increase the time but it was not tested for this.

One could expect that as the size of the sub-images is increasing and the number of sub-images is divided by the same number, the time for the program to run would be the same. This is not true and it can be justified by the time that the minimization needs to be done. The minimization uses the blurring matrix which has a much larger size than the image. It is actually the square of it. So the problems gets smaller but not that small so that the decrease in the number of images would be dominant. The increase does not follow some linear function for the same reason.

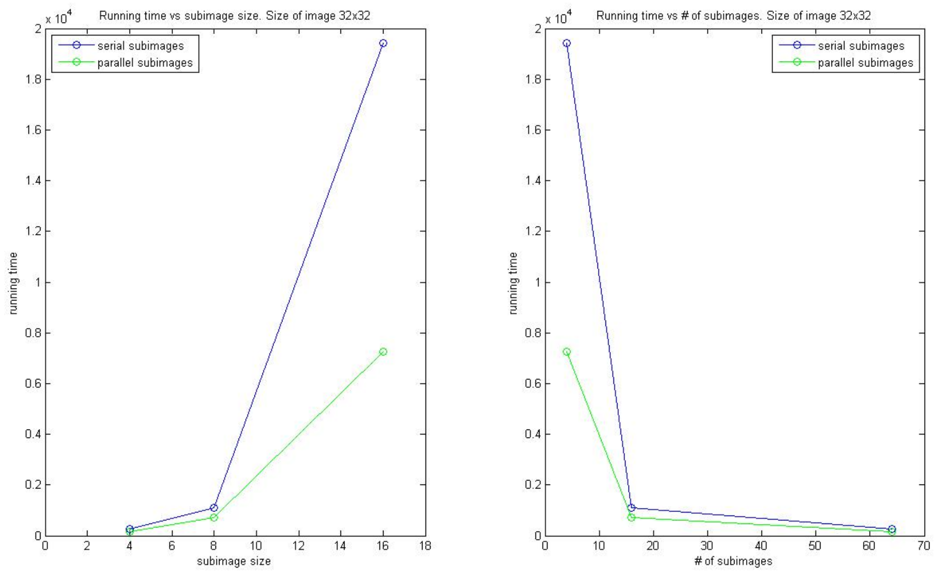


Figure 36: Running Time- sub-image size and number of sub-images

7 Summary

In this project 4 different codes strongly related to each other were developed. These codes compute simultaneous confidence intervals following the theory in [9] and [8]. Two of them are parallel versions of the corresponding serial codes computing the intervals from a whole image or an image separated into sub-images. These codes were validated and comparison with respect to time was performed.

8 Further Study

There are many questions that arose from this project. Some of them were answered but there are others that still need to be considered.

- Can the blurring matrix be computed every time we need it without much operation cost from only the middle column (or just from the PSF) so that we do not need to store the whole matrix?
- Can we only use the nonzero elements of the blurring matrix so that we can reduce even more the number of operations?
- What is the best way to compute confidence intervals? Shall we consider the

whole image or sub-images? And if we use sub-images, do we choose the most appropriate size?

- Can we improve the results by using previously computed confidence intervals?

Another useful approach would be using the FFT method to reduce the operations and memory involving the blurring matrix. This could also be examined later.

9 Project Schedule

The timeline that was set up at the beginning of the year at the project proposal and guided the progress of the project was the following.

September	Study the literature, get familiar with the image toolbox and the commands of Matlab for images, write and present project proposal.
October	Study the literature- understand all the aspects of the problem- write code.
November	Write the code and validate it. Prepare and complete Midyear Presentation
Early December	Write midyear report.
Late January	Test various images, begin to exercise on parallel computing.
February	Work on the parallel part of the code.
March	Test various images, validate and correct code if necessary.
April	Validate and correct code if necessary, write final report.
May	Present final report.

10 Milestones

Small variations to the above schedule have been done throughout the year as some tasks were accomplished sooner than planned, others took more time and some of them were also done simultaneously. Finally, though, everything finished on a timely manner according to the program.

The expected milestones can be found in the Project Proposal. Here I am apposing the real milestones as they were shaped throughout the year.

End of September	Enough studying of the literature was done so that I could write some first code using the Matlab Image Processing toolbox. The project proposal was prepared.
End of October	The presentation of the project proposal was done, a primitive image database was set and the first parts of the code were written.
End of November	The basic part of the code was still under examination as the results were not as expected. The midyear report was written and presented.
End of January	Serial code that deals with the image as a whole was finished.
End of February	Decision on working with the Parallel Computing Toolbox of Matlab was made. The code that uses sub-images was in its first steps.
End of March	Mid Semester Presentation was done and parallelization of the code was also done.
End of April	Codes are finalized.
Middle of May	Report and Presentation have been prepared and all of the deliverables are given to the instructors.

11 Deliverables

The main purpose of the course was to develop a code that solves an already solved problem, to document it, validate and support it. For this reason several presentations and reports including this one have taken place. The following is a list of the deliverables for the AMSC 664 project course.

Project Proposal and Proposal Presentation
Midyear Report and Midyear Presentation
Final Report and Final Presentation
Code
Database images
Validation module

References

[1] Tony F. Chan and Jianhong (Jackie) Shen, *"Image Processing and Analysis"*,SIAM, Philadelphia, 2005

[2] Martin Hanke, James Nagy and Robert Plemmons, *"Preconditioned Iterative Regularization For Ill-Posed Problems"*, IMA Preprint Series n 1024, 1992

[3] Per Christian Hansen, James G. Nagy and Dianne P. O'Leary, *"Deblurring Images Matrices, Spectra, and Filtering"*, SIAM, Philadelphia, 2006

[4] Richard A. Johnson, Gouri K. Bhattacharyya, *"Statistics: Principles and Methods"*, John Wiley & Sons, Inc., 2006

[5] Charles L. Lawson and Richerd J. Hanson, *"Solving Least Squares problems"*, SIAM, Philadelphia, 1995

[6] Jodi L. Mead, Rosemary A Renaut, *"Least squares problems with inequality constraints as quadratic constraints"*,Linear Algebra and its Applications, 432, 2010, p. 1936–1949

[7] James G. Nagy and Dianne P. O'Leary, *"Restoring Images Degraded By Spatially-Variant Blur"*, SIAM J. Sci. Comput., Vol 19, No 4, 1998, p. 1063-1082

[8] James G. Nagy and Dianne P. O'Leary, *"Image Restoration through Subimages and Confidence Images"*, Electronic Transactions on Numerical Analysis, 13, 2002, p. 22-37

[9] Dianne P. O'Leary and Bert W. Rust, *"Confidence Intervals for inequality constrained least squares problems, with applications to ill-posed problems"*, SIAM Journal on Scientific and Statistical Computing, 7, 1986, p. 473-489

[10] Bert W. Rust and Dianne P. O'Leary, *"Confidence intervals for discrete approximations to ill-posed problems"*, The Journal of Computational and Graphical Statistics, 3, 1994, p. 67-96

[11] L. Tenorio, A.Fleck and K. Moses, *"Confidence intervals for linear discrete inverse problems with non negativity constraint"*, Inverse Problems, 23, 2007, p. 669-681