Università degli Studi di Cagliari
Dipartimento di Matematica e Informatica
Corso di Laurea Magistrale in Matematica

Fast convolution algorithms for image deblurring

Candidate:

Marco Ratto

Supervisors:

Prof. Per Christian Hansen

Prof. Giuseppe Rodriguez

A.A. 2022/2023

Contents

1 Introduction **6**
1.1 Inverse problems . . . 6
1.2 Tikhonov Regularization . . . 8
1.2.1 Parameter choice . . . 10
1.3 Bayesian inverse problems . . . 11
1.4 Uncertainty Quantification . . . 12
1.5 Hyperparameters . . . 13
1.6 Sampling methods . . . 14
1.6.1 Markov chain Monte Carlo (MCMC) algorithms . . . 14
1.7 Regularization in the Bayesian framework . . . 16

2 Image deblurring **18**
2.1 How images become arrays of numbers . . . 18
2.2 Blurred images . . . 21
2.2.1 The Point Spread Function (PSF) . . . 21
2.3 Noise . . . 24
2.4 Convolution . . . 25
2.5 Boundary conditions . . . 26
2.5.1 Zero boundary conditions . . . 26
2.5.2 Periodic boundary conditions . . . 27
2.5.3 Reflexive boundary conditions . . . 28
2.5.4 Two-dimensional problems . . . 28

3 CUQI project **29**
3.1 Examples of research in CUQI . . . 29
3.1.1 Defect Detection in X-Ray CT of Subsea Pipes . . . 30
3.2 CUQIpy . . . 32
3.2.1 Specifying and solving Bayesian Inverse Problems . . . 33
3.2.2 Basic Example . . . 33

4 Fast Convolution Algorithms for Image Deblurring **36**
4.1 Test problems . . . 36
4.2 New Implementation . . . 40
4.2.1 Zero Boundary Conditions . . . 40
4.2.2 Periodic Boundary Conditions . . . 41
4.3 Numerical experiments . . . 42
4.3.1 Zero boundary conditions . . . 44
4.3.2 Periodic boundary conditions . . . 44

5 Conclusions **48**

Acknowledgements

This thesis represents the culmination of the work that I have carried on during a summer internship at the Technical University of Denmark (DTU), in the department of Scientific Computing. I had the privilege of being part of the CUQI (Computational Uncertainty Quantification for Inverse problems) project, an experience that really enriched me both personally and professionally. For this opportunity I have to say thank you to Professor Per Christian Hansen, who made me feel really welcome from the very first day and whose advice was always of incredible use, to Dr Nicolai Riis, for being always ready to help me with every problem that I had, and to Professor Giuseppe Rodriguez who introduced me to the fascinating world of numerical analysis and was essential in enabling me to write this thesis.

1 Introduction

Before getting into the details of the theoretical and numerical work that has been developed in the thesis, we explain, in general terms, some of the ideas that stand behind this entire work and that will be fundamental to understand what comes next.
We start with an introduction of concepts such as Inverse Problems, Bayesian inverse problems and Uncertainty Quantification, focusing on what will be useful in the following chapters.
This chapter is based on the following references: [2], [13], [3], [16] and [6].

1.1 Inverse problems

In science and engineering, we often apply physical principles to develop a mathematical model of a physical system in order to capture some phenomena of interest.
Physics-based models are useful for acquiring insight and understanding about a system, explaining observations, guiding future experiments, discovering new scientific questions, and making predictions.
In an abstract way, models are built such that we have

$$F(\mathbf{x}) = \mathbf{y}, \tag{1}$$

where $\mathbf{x} \in \mathbb{R}^n$ is the input of the problem, F is a linear or non linear operator, and $\mathbf{y} \in \mathbb{R}^m$ is the output. When we solve the *direct problem*, we know the value of $\mathbf{x}$, for example, through a physical measurement, and we want to compute $\mathbf{y}$.
Usually this kind of problem is well-posed, which means that satisfies the following definition.

Definition 1 *A problem is* ***well-posed*** *if the following properties hold:*

- *the problem has a solution,*
- *the solution is unique,*
- *the solution's behavior depends continuously on input data.*

In real life problems, it may happen that we don't have access to the measures that we are interested in, in order to define a direct problem as described earlier, so we are forced to solve what is called the *inverse problem*, which means: in the model (1), we know the value of $\mathbf{y}$ and we want to reconstruct $\mathbf{x}$.
An example of this can be found in X-ray computed tomography (CT), where

we want to find an image of the cross-section of a part of the body, but clearly we don't have direct access to it, so we measure the attenuation of X-rays sent through the body. With the notation introduced before, $\mathbf{x}$ would be the image of the cross-section of the body, $\mathbf{y}$ would be our measurements, and F is the mathematical model that we have to "invert".
Inverse problems are often ill-posed, so at least one of the conditions presented above for being well-posed is violated. Also, it happens frequently that inverse problems are ill-conditioned, meaning that small errors in the inputs can result in much larger errors in the answers.
In linear problems (4), we define the condition of the problem as the condition number of the matrix A.

Definition 2 *Let $A \in \mathbb{R}^{m\times n}$ be a matrix. Let $A^\dagger$ be its Moore-Penrose inverse. Then the* ***condition number*** *of A is:*

$$k(A) = \|A\|\|A^\dagger\|. \tag{2}$$

This definition stands for every matrix norm, but if we use the 2-norm, it can be proven the following

Theorem 1 *Let $A \in \mathbb{R}^{m\times n}$ be a matrix with rank(A) $= r$ and singular values $\sigma_1, \dots, \sigma_r$. Then*

$$k_2(A) = \|A\|_2\|A^\dagger\|_2 = \frac{\sigma_1}{\sigma_r}. \tag{3}$$

As the singular values are in descending order, this theorem shows that a matrix tends to be very ill-conditioned when it has non-zero singular values that are very small with respect to the largest one.
When we have to deal with a ill-posed and/or ill-conditioned inverse problem we may adopt different techniques depending on the properties that are not satisfied.
For example, in a linear inverse problem of the type

$$A\mathbf{x} = \mathbf{y}, \tag{4}$$

where A is a $m \times n$ matrix, $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{y} \in \mathbb{R}^m$, it may occur that:

- $\mathbf{y}$ is not in the range of A, so the solution does not exists.
  In this case we can reformulate the problem in order to find a solution. Often we solve the alternative problem:

$$\min_{\mathbf{x}} \|A\mathbf{x} - \mathbf{y}\|_2^2 . \tag{5}$$

- The solution is not unique, so we want to change our problem to make it stricter. A popular choice is to pick, among all the solutions of the original problem, the one with minimum norm.
- The matrix A is ill-conditioned, so when our data $\mathbf{y}$ is perturbed by a (small) noise, the solution may not be close the one that we would have if we could have access to the real data, without noise. In this situation we use a regularization method, which penalizes some specific feature that we want to avoid. For example, we often want the solution to be smooth, so our regularization method will penalize high frequency oscillation, in order to have a balance between data fitting and smoothness.

To better explain the meaning of the condition number, we now see how the solution can change when we have to deal with perturbed data.
Think that we are studying two related problems:

$$A\mathbf{x} = \mathbf{y} \quad \text{and} \quad A\tilde{\mathbf{x}} = \tilde{\mathbf{y}}, \tag{6}$$

where $\tilde{\mathbf{y}}$ is a perturbed version of $\mathbf{y}$, so $\Delta\mathbf{y} = \tilde{\mathbf{y}} - \mathbf{y}$ is the perturbation of $\mathbf{y}$. We are interested in bounding the difference between the two solutions $\tilde{\mathbf{x}}$ and $\mathbf{x}$. More precisely, we can set an upper bound for the relative normwise difference, so our relation is independent on the scaling of A, $\mathbf{x}$ and $\mathbf{y}$.
If we compute the least squares solutions $\mathbf{x}_{LS} = \arg\min \|A\mathbf{x} - \mathbf{y}\|_2^2$ and $\tilde{\mathbf{x}}_{LS} = \arg\min \|A\mathbf{x} - \tilde{\mathbf{y}}\|_2^2$, we have that:

$$\frac{\|\tilde{\mathbf{x}}_{LS} - \mathbf{x}_{LS}\|_2}{\|\mathbf{x}_{LS}\|_2} \leq k_2(A)\frac{\|\Delta\mathbf{y}\|_2}{\|A\mathbf{x}_{LS}\|_2}, \tag{7}$$

which means that the perturbation is governed by A's condition number.

1.2 Tikhonov Regularization

The most important and used regularization method is *Tikhonov regularization.*
In real problems, as we said before, we don't access to exact data, but the latter is generally corrupted by some noise, so a linear problem like (4), becomes:

$$A\mathbf{x} = \mathbf{y}^\delta, \quad \textit{with } \mathbf{y}^\delta = \mathbf{y} + \mathbf{e}, \tag{8}$$

where $\mathbf{y}$ is the unknown error-free vector and $\mathbf{e}$ is the unknown error, which we assume to be bounded by δ, $\|\mathbf{e}\| \leq \delta$. We may solve this problem by computing the least-squares solution

$$\mathbf{x}_{LS} = \arg\min \|A\mathbf{x} - \mathbf{y}\|_2^2\,, \tag{9}$$

but, when the matrix A is ill-conditioned, this is not a meaningful approximation of the exact solution, because by the definition of ill-conditioned problem, a small error in the data produces a much bigger error in the solution.
To avoid this, in (9), we introduce a *regularization term* and the new problem becomes:

$$\mathbf{x}_\alpha = \arg\min \|A\mathbf{x} - \mathbf{y}\|_2^2 + \alpha\|\mathbf{x}\|_2^2 , \quad \alpha > 0. \tag{10}$$

We have that:

- the first term is the standard least squares term, which measures how well the solution fits the data;
- the second term is the regularization term, which penalizes the magnitude of $\mathbf{x}$;
- α is the regularization parameter, which controls the trade-off between fitting the data and keeping the solution vector $\mathbf{x}$ small.

The minimization problem (10) is commonly referred to as Tikhonov regularization in *standard form*.
To quantify the improvement given by introducing a regularization term, we can measure the condition number of the regularized problem.
It is true that if we want to solve the two related problems (6), and we do it by Tikhonov regularization, we have that:

$$\frac{\|\tilde{\mathbf{x}}_\alpha - \mathbf{x}_\alpha\|_2}{\|\mathbf{x}_\alpha\|_2} \leq k_\alpha \frac{\|\Delta\mathbf{y}\|_2}{\|A\mathbf{x}_\alpha\|_2}, \tag{11}$$

with

$$k_\alpha = \frac{\sigma_1}{\alpha}. \tag{12}$$

So, comparing (11) with (7), it is clear that the condition number of the problem has changed from $k_2(A) = \sigma_1/\sigma_r$ to $k_\alpha = \sigma_1/\alpha$, and this can be a big improvement when σ_r is very small, that is what happens in ill-conditioned matrices.
When we know exactly which is the feature that we want to penalize in the solution, it is well known that it is often possible to improve the quality of the approximation determined by Tikhonov regularization by replacing the problem (10) by:

$$\mathbf{x}_\alpha = \arg\min \|A\mathbf{x} - \mathbf{y}\|_2^2 + \alpha\|L\mathbf{x}\|_2^2 , \quad \alpha > 0, \tag{13}$$

where L is a suitable regularization operator. For example, when we want a smooth solution, we have to penalize high-frequency oscillation, so L can be the Total Variation operator, which works as a discretization of a derivative operator. This means that if $\mathbf{x}$ is smooth, $\|L\mathbf{x}\|_2$ is going to be small, and if $\mathbf{x}$ is rapidly oscillating, $\|L\mathbf{x}\|_2$ is going to be large, so this kind of solution is unlikely to be the optimal one. This minimization problem is referred to as Tikhonov regularization in *general form.*

1.2.1 Parameter choice

The greatest problem the we face when using Tikhonov regularization (both in standard and general form) is the choice of the regularization parameter α.
The intuition is that if we set α too small, the regularization term can grow without affecting too much the minimization problem, so the solution $\mathbf{x}_\alpha$ (13) is going to be very close to $\mathbf{x}_{LS}$ (9), which means that we are not applying a strong regularization; on the other hand, if α is too big we are going to find a solution that is very smooth, but may not fit the data at all.
There are many criterions to help us finding the best compromise, we describe briefly some of them.

1. **Generalized Cross Validation (GCV).**
   Using the cross-validation principle we take as validation set the i^{th} row of the system and as training set all the remaining rows. Then we compute the regularized solution of the problem on the training set and we take the α value that reconstruct better the i^{th} row, $\mathbf{a}_i^T\tilde{\mathbf{x}} = b_i$. After doing this for each row we can take α_{GCV} as the average of the previous α values.

2. **L-curve criterion.**
   The L-curve is a plot of the norm of the data fitting term versus the regularization term for various values of α, usually made on logarithmic scale. When we observe the shape of the curve, the L-curve often exhibits an elbow or corner. If this happens, we choose α at this point, where we can say, heuristically, that there is a good compromise between the two terms.

3. **Discrepancy Principle (DP).**
   We choose α such that:

$$\|A\mathbf{x}_\alpha - \mathbf{y}^\delta\|_2^2 = \tau^2\delta^2\,, \tag{14}$$

   for a chosen $\tau > 1$. This value of α can be proven to exist and to be unique. The idea behind this method is to find α such that the residual $\|A\mathbf{x}_\alpha - \mathbf{y}^\delta\|_2^2$ as the magnitude of the noise, in order to avoid overfitting.

The parameter $\tau > 1$ is necessary in order to prove the existence and the uniqueness of the solution.

1.3 Bayesian inverse problems

A different approach for solving the problem (1), in which F is an operator that represent the forward model, $\mathbf{y}$ is the vector of the noisy data and $\mathbf{x}$ is the solution that we want to find, is to consider both $\mathbf{x}$ and $\mathbf{y}$ as **random variables**. This is what we call **Bayesian setting**.
In this scenario the statistical model for our problem is then characterized by the *joint probability distribution* $p(\mathbf{x}, \mathbf{y})$.
A useful representation of the joint distribution $p(\mathbf{x}, \mathbf{y})$ is to list what is known or assumed about the parameters and data defining what in statistics is usually called *Bayesian generative model.* Said so, our problem is defined in terms of two distribution:

$$\begin{aligned} \mathbf{x} &\sim p(\mathbf{x}), \\ \mathbf{y} &\sim p(\mathbf{y}|\mathbf{x}). \end{aligned} \tag{15}$$

We use the terms *prior* and *data distribution* for the distributions associated with the solution parameters and the data, respectively, in (15).
In *Bayesian inverse problems*, the goal is not to compute a fixed value for the solution, but to infer the solution given a particular realization of the data. The *posterior distribution* $p(\mathbf{x}|\mathbf{y})$ characterizes the distribution of the solution $\mathbf{x}$ to the inverse problem, given the data $\mathbf{y}$. One of the keys of this approach to inverse problems is *Bayes' Theorem* for continuous probability densities.

Theorem 2 (Bayes' Theorem) *Given a probability distribution P, two events A and B, with $P(B) \neq 0$, we have that*

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}. \tag{16}$$

In our case, it allows us to express the posterior as

$$p(\mathbf{x}|\mathbf{y}) = \frac{p(\mathbf{y}|\mathbf{x}) \cdot p(\mathbf{x})}{p(\mathbf{y})}. \tag{17}$$

Given fixed observed data $\mathbf{y}^{\mathbf{obs}}$, $p(\mathbf{y}|\mathbf{x})$ considered as a function of $\mathbf{x}$ is known as the *likelihood function* or just *likelihood* and it is denoted by $L(\mathbf{x}|\mathbf{y} = \mathbf{y}^{\mathbf{obs}})$, which is not a probability density but a function. Furthermore, we can notice that $p(\mathbf{y})$ is a normalization constant that is usually omitted, so we write the posterior as proportional to the product of the likelihood and prior:

$$p(\mathbf{x}|\mathbf{y} = \mathbf{y}^{\mathbf{obs}}) \propto L(\mathbf{x}|\mathbf{y} = \mathbf{y}^{\mathbf{obs}})p(\mathbf{x}). \tag{18}$$

1.4 Uncertainty Quantification

This Bayesian setting has been introduced because when we compute the solution of an inverse problem, that usually is ill-posed and/or ill-conditioned, we don't know how much we can rely on our solution. Of course the solution can not be an exact value that we just consider to be right, because we don't have access to the exact data, but our solution is computed starting from data that are corrupted by some noise. As it is impossible not to have any uncertainty, we develop tools and techniques to measure it. We need to characterize and evaluate this uncertainty, such that we can make informed and safe decisions based on the computed results. This is the basic idea behind the field of *uncertainty quantification* (UQ).
This approach builds upon theory and methods from Bayesian inference that provide a powerful and flexible framework for measuring and quantifying uncertainty.
The Bayesian formulation gives two key advantages. First, it allows us to incorporate prior (i.e., before data is collected) information and/or beliefs about the $\mathbf{x}$ in our solution to the inverse problem. Second, it yields a probabilistic solution to inverse problems that quantifies uncertainty about the input/parameter, as the solution is not represented by a fixed value, but by a random variable.
In this Bayesian view, the probability assigned to each region of the input/-parameter space reflects our degree of belief that the it falls in that region. This belief is based on some combination of subjective belief and objective information. For general understanding, the spread of the probability density over the input/parameter space reflects our uncertainty about its value, while the more concentrated they are, the more certainty we have.
Our uncertainty about the solution is, clearly, different before and after we conduct an experiment, meaning the moment when we collect data, and compare this data with our model. Consequently, the solution has a prior density before the data are considered, as said before, based on our general knowledge of the problem that we are solving, then an updated, posterior density after we are enlightened by the data.

In the end, to tackle an inverse problem using uncertainty quantification, we need two main ingredients.

- The **prior distribution** of the solution, expressing our beliefs about it before the data are collected/observed. Constructing a prior is context-dependent and involves a degree of subjectivity, so it stands to us deciding what prior distribution fits well for each specific problem. Many things may be taken into account as we have that, for example, prior may be roughly categorized as diffuse, weakly informative, or informative, based on the amount of uncertainty it admits about the possible values where the solution is going to fit. An informative prior, e.g. a Gaussian distribution with a small variance, expresses a high degree of certainty about the solution. On the other hand, a diffuse prior, e.g. a uniform distribution, spreads its density widely over input/parameter space to express a very high degree of uncertainty. An informative prior influences the posterior more than a diffuse prior, which lets more freedom to the solution to follow what the data indicates. Generally, as we gather more data, the influence of the prior on the posterior tends to weaken as the data overrides the prior.

- The **likelihood function** of the solution, giving the probability density of the data $\mathbf{y}$ conditioned on each value of the $\mathbf{x}$. The likelihood function is constructed from the data and from our model of the data-generating process. The likelihood quantifies the support that the data lend to each value of the unknown solution.

1.5 Hyperparameters

It is often the case that the distributions depend on one or more unknown parameters.
In the Bayesian paradigm, we can assign probability densities to them and include them in the Bayesian Problem.
To give an example, in the linear inverse problem

$$A\mathbf{x} = \mathbf{y}, \tag{19}$$

the prior and the data distribution can both depend on different parameters, let's call them d and s. In this case, the general Bayesian problem for the joint probability distribution $p(\mathbf{x}, \mathbf{y}, s, d)$ would be

$$\begin{align} d &\sim p(d), \tag{20}\\ s &\sim p(s), \tag{21}\\ \mathbf{x} &\sim p(\mathbf{x}|d), \tag{22}\\ \mathbf{y} &\sim p(\mathbf{y}|\mathbf{x}, s). \tag{23} \end{align}$$

The posterior associated with this Bayesian Problem becomes

$$p(\mathbf{x}, d, s|\mathbf{y} = \mathbf{y}^{\mathbf{obs}}) \propto L(\mathbf{x}, s|\mathbf{y} = \mathbf{y}^{\mathbf{obs}})p(\mathbf{x}|d)p(d)p(s), \tag{24}$$

meaning that the task is now to infer about $\mathbf{x}$ as well as d and s. This is usually referred to as *hierarchical modeling* and we refer to such additional model parameters as *hyperparameters.*

1.6 Sampling methods

In this Bayesian framework, the solution to the inverse problem, which is the posterior distribution of the unknown input of the forward model, follows from the prior density and likelihood function of the input via Bayes' Theorem. The posterior density of the unknown solution gives the probability that the input falls in any given region of input space, conditioned on the data. The posterior updates the prior in light of the data, offers a compromise between the prior and the likelihood, and constitutes the raw solution to the inverse problem that quantifies uncertainty through its spread. In practice, except for a few cases where closed-form expressions are available, the complicated form of the posterior distribution makes it impossible to compute samples analytically from the distribution, so we have to make use of some sort of approximation to get samples that are independent and behave like they were actually taken from our posterior distribution. To do so, some sampling method has to be employed.

1.6.1 Markov chain Monte Carlo (MCMC) algorithms

The computation of samples is often based on *Markov chain Monte Carlo* (MCMC) algorithms. These methods have revolutionized the field of statistics and data analysis by providing powerful tools to sample from complicated probability distributions. Originally developed in the mid-20th century, MCMC techniques have found applications in various domains, from Bayesian statistics to machine learning and computational biology. At the heart of MCMC methods lies the concept of Markov chains.

Definition 3 *A Markov chain is a mathematical model that describes a sequence of events where the probability of each event depends only on the state attained in the previous event.*

The basic idea of MCMC algorithms is to build such Markov chains, which are easy to sample from, and whose stationary distribution is our target distribution, such that when following them, in the limit, we obtain samples from the target distribution.
There many MCMC algorithms that perform very well in different situations, we will now show briefly some of the most popular.

1. **Metropolis–Hastings (MH) sampling.**
   This classical method uses a two-stage procedure with proposal step and acceptance/rejection steps. The first step computes a proposal $\mathbf{x}'$ with the density $q(\mathbf{x}|\mathbf{x}')$ which is the conditional probability of $\mathbf{x}$ given

the proposed state $\mathbf{x}'$. The second step computes the acceptance ratio

$$\alpha(\mathbf{x}, \mathbf{x}') = \min\left(1, \frac{p(\mathbf{x}')q(\mathbf{x}'|\mathbf{x})}{p(\mathbf{x})q(\mathbf{x}|\mathbf{x}')}\right) \tag{25}$$

which expresses the conditional probability of accepting $\mathbf{x}'$.
At state k of the MH algorithm, we want to determine the next state $\mathbf{x}^{(k+1)}$, and this is chosen by first sampling a candidate point $\mathbf{x}'$ from the proposal density $q(\,\cdot\,|\mathbf{x}^{(k)})$. Then $\mathbf{x}'$ is accepted with probability (25) and $\mathbf{x}^{(k+1)} = \mathbf{x}'$, otherwise it is rejected and $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)}$.

2. **Preconditioned Crank–Nicolson (pCN) sampling.**
This method assumes that the prior has a Gaussian distribution. It uses the Crank–Nicolson finite-difference scheme to solve an underlying stochastic differential equation (SDE) that is invariant with respect to the posterior. When we choose the prior covariance matrix $\mathbf{S}$ as preconditioner, we obtain the following mechanism for producing the proposed state $\mathbf{x}'$ in the MH method:

$$\mathbf{x}' = \sqrt{1-s^2}\,\mathbf{x} + s\mathbf{C}, \quad \textit{where} \quad \mathbf{C} \sim \mathcal{N}(\mathbf{0}, \mathbf{S}),\; s \in (0, 1]. \tag{26}$$

Given a current state $\mathbf{x}^{(k)}$, it follows from (26) that the associated proposal distribution is Gaussian with mean vector $\sqrt{1-s^2}\,\mathbf{x}^{(k)}$ and covariance matrix $s^2\mathbf{S}$. We remark that while pCN is designed for a Gaussian prior, it can be extended to more general priors by applying a transformation to a standard Gaussian distribution.

3. **Gibbs sampling.**
This method is useful when the posterior is expressed as a joint distribution via conditional densities, and in connection with hierarchical models. Specifically, Gibbs sampling is useful when computing samples from the joint distribution is impractical, but drawing samples from the conditional distributions of given parameter components is feasible. To illustrate Gibbs sampling, consider a generic joint distribution $p(\mathbf{x}, \mathbf{y}, \mathbf{z})$. The next state in the chain for $\mathbf{x}, \mathbf{y}, \mathbf{z}$ is generated from the previous state as follows:

$$\begin{aligned} \mathbf{x}^{(k+1)} &\sim p(\mathbf{x}|\mathbf{y}^{(k)}, \mathbf{z}^{(k)}) \\ \mathbf{y}^{(k+1)} &\sim p(\mathbf{y}|\mathbf{x}^{(k+1)}, \mathbf{z}^{(k)}) \\ \mathbf{z}^{(k+1)} &\sim p(\mathbf{z}|\mathbf{x}^{(k+1)}, \mathbf{y}^{(k+1)}) \end{aligned} \tag{27}$$

The special case where a single random vector is sampled involves treatment of the elements component-by-component, and this scheme is known as the *component-wise Metropolis–Hastings* (CWMH) algorithm.

1.7 Regularization in the Bayesian framework

When we solve an inverse problem using the Bayesian framework it looks like we are just tackling the problem without any regularization. This could appear as a bad way of working, after all the things we have said about ill-posed and ill-conditioned problem, but we will show in this section that this is not the case.
Using Bayes' law (17), we have seen that we can compute the *maximum a posteriori* (MAP) estimator by maximizing the posterior density function, which is proportional to the product of the likelihood and the prior (18). Focusing on the specific problem:

$$\mathbf{y} = A\mathbf{x} + \mathbf{e}\,, \quad \mathbf{e} \sim \mathcal{N}(\mathbf{0}, \lambda^{-1}\mathbf{I}_m), \tag{28}$$

with $A \in \mathbb{R}^{m\times n}$, we model the unknown random vector $\mathbf{x}$ with a prior density function $p(\mathbf{x}|\delta)$, where δ is a positive scaling parameter.
For simplicity, we will show an example assuming normality, so we consider the prior and the data distribution to be normal:

$$\mathbf{x} \sim \mathcal{N}(\mathbf{0}, \delta^{-1}\mathbf{I}_n), \quad \mathbf{y} \sim \mathcal{N}(A\mathbf{x}, \lambda^{-1}\mathbf{I}_m). \tag{29}$$

This means that the prior distribution of $\mathbf{x}$ is

$$p(\mathbf{x}|\delta) = \left(\frac{\delta}{2\pi}\right)^{n/2} \exp\left(-\frac{\delta}{2}\|\mathbf{x}\|_2^2\right), \tag{30}$$

and the data distribution is

$$p(\mathbf{y}|\mathbf{x},\lambda) = \left(\frac{\lambda}{2\pi}\right)^{m/2} \exp\left(-\frac{\lambda}{2}\|A\mathbf{x}-\mathbf{y}\|_2^2\right). \tag{31}$$

As we said, with Bayes' law we compute the posterior:

$$p(\mathbf{x}|\mathbf{y},\lambda,\delta) \propto p(\mathbf{y}|\mathbf{x},\lambda)p(\mathbf{x}|\delta), \tag{32}$$

which becomes

$$p(\mathbf{x}|\mathbf{y},\lambda,\delta) \propto \exp\left(-\left(\frac{\lambda}{2}\|A\mathbf{x}-\mathbf{y}\|_2^2 + \frac{\delta}{2}\|\mathbf{x}\|_2^2\right)\right). \tag{33}$$

Now that we have the posterior, that defines a probability density for $\mathbf{x}$, conditioned of $\mathbf{y}$ and on the parameters λ and δ we can compute different estimators for the unknown vector $\mathbf{x}$, such as the mean and the standard deviation, in order to determine, for example, confidence intervals.

To find the maximum a posteriori estimator we have to maximize $p(\mathbf{x}|\mathbf{y}, \lambda, \delta)$, which is equivalent to minimize $-\ln(p(\mathbf{x}|\mathbf{y}, \lambda, \delta))$, i.e.,

$$\mathbf{x}_{MAP} = \arg\min \left\{ \frac{\lambda}{2} \|A\mathbf{x} - \mathbf{b}\|_2^2 + \frac{\delta}{2} \|\mathbf{x}\|_2^2 \right\}, \tag{34}$$

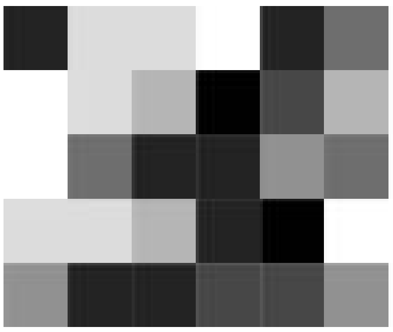
that is exactly the Tykhonov regularized solution, defined in (10), with the regularization parameter $\alpha = \delta/\lambda$.
This shows how the regularization in Bayesian inverse problems is included in the choice of the prior distribution. It is still true also for other situation, where we do not assume normality, so we do not find the Tykhonov regularization, but we find some other forms, specific for each assumption.

2 Image deblurring

The problem of image deblurring is the process of removing or reducing blurriness from an image to make it clearer and sharper. It is clear that whenever we take a picture, or we have to deal with any image, we want it to be as close to reality as possible, but every picture is more or less blurry so image deblurring is fundamental in making pictures nice and useful.
A digital image is composed of picture elements, called pixels. Each pixel is assigned an intensity, meant to characterize the color of a small rectangular portion of the scene that we want to capture. The number of pixels can be very different from one image to another, typically small images have around $256^2 = 65536$ pixels and high resolution images can have over 10 million pixels.
Some blurring always arise in the recording of a digital image and it is the consequence of the scene information "spilling over" to neighboring pixels.
The causes for this to happen can be many, and often more than one at the same time. For example the optical system in a camera lens may be out of focus, so that the incoming light is smeared out, or the camera may be moved when we take the shot, so the light intensity of a portion is spread toward the opposite direction of the movement of the camera.
In image deblurring, we seek to recover an image that is as close as possible to the original one by using a mathematical model of the blurring process. The key issue is that some of the information that we lose with the blurred image is indeed still present, but it can only be recovered if we know the details of the blurring process. So, in the end, there is no hope that we can recover the exact original image, because there are some errors in the recorded image that are unavoidable, such as fluctuations in the recording process and approximation errors when representing the image with a limited number of digits, and that cause some error in the restored image. The main challenge in image deblurring is to devise efficient and reliable algorithms for recovering as much information as possible from the given data. In this chapter we will see an introduction to how this is done, following as a reference the book [7].

2.1 How images become arrays of numbers

In order to process images using mathematical techniques is crucial to have a way of representing images as arrays of numbers.
The easiest case is represented by grayscale images, where the image can be easily represented by a matrix, where each entry represent the "amount of light" present in the corresponding pixel. For example, in Figure 1 we can see a simple image with 5×6 pixels, where, for each pixel, the value 0 represents



$$\Longleftrightarrow \begin{bmatrix} 6 & 1 & 1 & 0 & 6 & 4 \\ 0 & 1 & 2 & 7 & 5 & 2 \\ 0 & 4 & 6 & 6 & 3 & 4 \\ 1 & 1 & 2 & 6 & 7 & 0 \\ 3 & 6 & 6 & 5 & 5 & 3 \end{bmatrix}$$

Figure 1: 5×6 grayscale image (stack overflow)

white, the value 7 represents black and all the numbers in between represent the shades of gray. It is clear that having only 8 possibilities for the shades of gray does not allow to be very precise in representing more complex images, in fact, a more common choice is that pixels have value between 0 and 255, where 0 represents black and 255 represents white.
Color images can be represented using various formats, the most popular one is the RGB (Red, Green and Blue). The RGB color model is an additive color model in which the red, green and blue primary colors of light are added together in various ways to reproduce a broad array of colors. In the RGB format images are stored as three components, which represents their intensity on the red, green and blue scales, so an image is stored as a 3-dimensional array, where the 3 dimensions represents, in order, the number of rows of pixels, the number of columns of pixel and the number of components of colors (3, Red, Green and Blue). In practice, for each image we have to store 3 matrices. For example, in a scale where the three components are, as before, in $[0, 255]$, we have that $(255, 0, 0)$ represents red, $(0, 255, 0)$ represents green and $(0, 0, 255)$ represents blue. In order to have all the other colors we have to mix the three "ingredients" and the sum of the colors intensities will give the wanted color.
We see a simple example in Figure 2, which is the stored in the RGB color

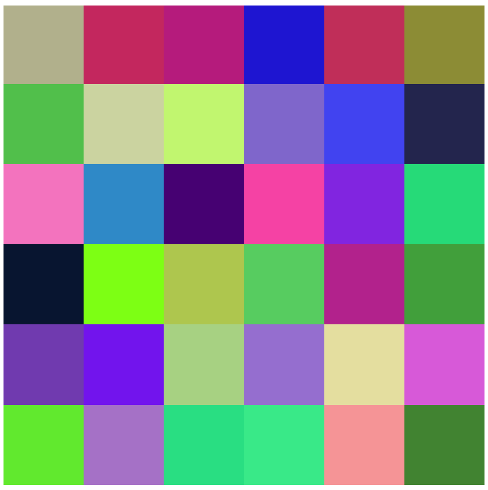


Figure 2: 6×6 RGB image

model as three matrices:

$$R = \begin{bmatrix} 177 & 195 & 181 & 30 & 192 & 140 \\ 81 & 203 & 193 & 127 & 65 & 35 \\ 243 & 47 & 70 & 245 & 129 & 38 \\ 8 & 125 & 174 & 87 & 178 & 65 \\ 112 & 114 & 167 & 149 & 228 & 215 \\ 97 & 165 & 41 & 57 & 245 & 65 \end{bmatrix}, \tag{35}$$

$$G = \begin{bmatrix} 176 & 39 & 27 & 21 & 46 & 140 \\ 191 & 211 & 246 & 102 & 67 & 37 \\ 115 & 137 & 1 & 66 & 37 & 218 \\ 21 & 255 & 198 & 204 & 34 & 159 \\ 58 & 20 & 209 & 110 & 222 & 89 \\ 233 & 113 & 222 & 233 & 148 & 131 \end{bmatrix}, \tag{36}$$

$$B = \begin{bmatrix} 140 & 94 & 124 & 209 & 89 & 53 \\ 75 & 160 & 111 & 203 & 240 & 77 \\ 190 & 199 & 114 & 164 & 224 & 120 \\ 48 & 20 & 78 & 96 & 140 & 59 \\ 175 & 237 & 130 & 207 & 159 & 216 \\ 46 & 198 & 130 & 136 & 150 & 49 \end{bmatrix}. \tag{37}$$

2.2 Blurred images

In this section we will focus on how we can build a mathematical model that explains the blurring in an image. In particular, we will see that most of the blurring caused by some mechanical or physical process can be described by a linear model. This will allow us to set up a system of linear equations whose solution, at least in principle, is the unblurred image.
For simplicity, we will focus on grayscale images, which we recall to be stored in computers as arrays, whose dimensions, $m \times n$ are the number of pixels, and each entry of the array represents the light intensity of the latter.
An image of size $m \times n$ can be rearranged into a column vector by stacking its columns one after the other, to form a vector of shape $N = mn$. This will be very useful, as we want to build a linear model and we need the $\mathbf{x}$ and the $\mathbf{y}$ to be vectors in order to apply what we have seen so far. Our notation will be the matrix one (i.e., $\mathbf{X}$) for images in their original shape, and the vector one (i.e., $\mathbf{x}$) for the rearranged vector form.
In our model we can assume the existence of an exact image, which is the image we would record if the blurring and noise were not present. We also assume that this ideal image, the one the would like to reconstruct, has the same size of the recorded one, that will be $m \times n$. We will refer to the exact image either with $\mathbf{X}$, for its original form, or $\mathbf{x}$ for its vector form, while the blurred image that we have recorded will be referred to with $\mathbf{Y}$ or $\mathbf{y}$.
In the linear model there exists a large matrix A of dimensions $N \times N$ (with $N = mn$), such that $\mathbf{x}$ and $\mathbf{y}$ are related by the equation

$$A\mathbf{x} = \mathbf{y}. \tag{38}$$

The matrix A represents the blurring.
Our next goal is to understand, in practice, how to construct this matrix.

2.2.1 The Point Spread Function (PSF)

Imagine the simple case where the exact image is all black, except for a single bright pixel. If we take a picture of this image, the blurring operation will cause the single bright pixel to spread over its neighboring pixels, as we can see in Figures 3 and 4. The single bright pixel is called *point source* and the function that produces this blurring is called *point spread function*.

Mathematically, the point source is the vector $\mathbf{e}_i$, which means all zeros except the ith pixel which is 1. From this we can reconstruct the matrix A by computing

$$A\mathbf{e}_i = i^{th} \text{ column of } A, \tag{39}$$

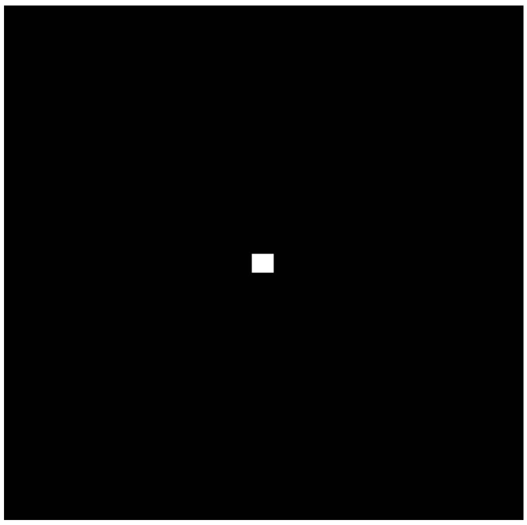


Figure 3: Point source

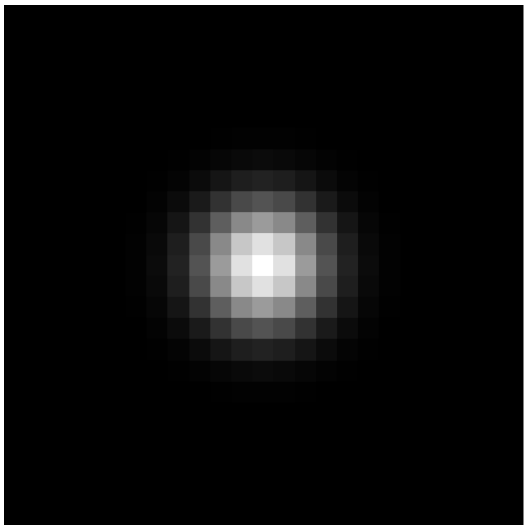


Figure 4: Blurred point source

and repeating the process for each $i = 1, \dots, N$. In the following sections we will show how this process is affected by the fact that we can only see a finite region of a scene that extends forever in all directions, so in order to fully determine A, we will need to set what are called *boundary conditions*.
One important aspect of blurring is that it is usually a local phenomenon, which means that the influence of the PSF is confined to a small area around the *center of the PSF* (the pixel location of the point source). Also, often we can assume that the PSF is the same regardless of the location of the point source. This happens, for example when the blurring is caused by the motion of the camera, or by the lens out of focus. In this case we say that the blurring is *spatially invariant*.
As a consequence of the local nature of the blurring and its linearity, to save storage we can often represent the PSF as an array P of much smaller dimension than the blurred image.
In some cases, knowledge of the physical process that causes the blur provides an explicit formulation of the PSF. When this happens we can give an exact expression of the element of the PSF array P.
For example, when the blur is caused by an out-of-focus lens, it generates a **circle of confusion**, an optical spot caused by a cone of light rays from a lens not coming to a perfect focus when imaging a point source (Figure 5). This happens because a lens can precisely focus objects at only one distance;

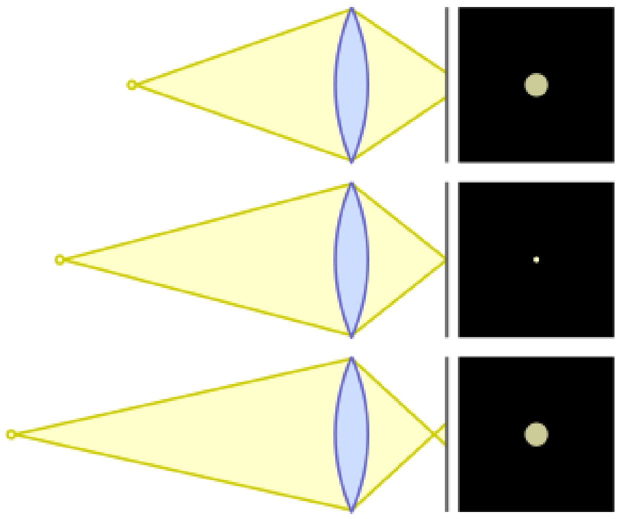


Figure 5: Diagram showing circles of confusion for point source too close, in focus, and too far (Wikipedia).

objects at other distances are defocused. Defocused object points are imaged as blur spots rather than points; the greater the distance an object is from the plane of focus, the greater the size of the blur spot. The PSF that describes a circle of confusion has elements p_{ij} given by:

$$p_{ij} = \begin{cases} 1/(\pi r^2) & \text{if} \quad (i-k)^2 + (j-l)^2 \leq r^2 \,, \\ 0 & \text{elsewhere.} \end{cases} \tag{40}$$

where (k,l) is the center of P, and r is the radius of the blur.

2.3 Noise

In addition to blurring, observed images are usually contaminated with *noise*. Noise can arise from different sources, but, in this section, we will focus on the ones the appear when recording grayscale images by means of a *charge-couple device* (CCD), which is an array of tiny detectors, arranged in a rectangular grid, able to record the amount of light that hits each detector.
In this case, noise comes essentially from three sources, and it can take the following three forms:

1. **Poisson noise.**
   When the noise comes from *background photons*, from both natural and artificial sources, it corrupts each pixel that we measure. This kind of noise is typically modeled by a Poisson distribution with a fixed parameter.

2. **Gaussian white noise.**
   The CCD electronics and the analog-to-digital conversion of measured voltage result in *readout noise*. This is usually assumed to consist of independent and identically distributed random values. A noise with this properties is called *white noise*. Often a good model for this kind of noise is the normal distribution with mean 0 and fixed standard deviation. Such random error are called Gaussian white noise.

3. **Quantization error.**
   The analog-to-digital conversion also results in *quantization error*, due to the fact that we have to represent signals by a finite (usually small) number of bits. Quantization error can be approximated by uniformly distributed white noise, whose standard deviation is inversely proportional to the number of bits used.

These different noises are additive, so, in our notation, we can write:

$$\mathbf{Y} = \mathbf{Y}_{\text{exact}} + \mathbf{E}, \tag{41}$$

where $\mathbf{Y}_{\text{exact}}$ is the blurred image without noise and $\mathbf{E}$ contains one or more of the described noises.

2.4 Convolution

The matrix-vector product $A\mathbf{x}$, where A contains the PSF and $\mathbf{x}$ is the vector form of the image $\mathbf{X}$, can be seen and computed as a *convolution* between 2-dimensional signals. To understand this, we give an introduction to what convolution is.

Definition 4 *Given $f, g : \mathbb{R}^2 \to \mathbb{R}^2$, the* ***convolution*** *between f and g is*

$$(f * g)(s,t) = \int_{\mathbb{R}^2} f(u,v)g(s-u, t-v)du\, dv\,. \tag{42}$$

In our applications we have to deal with arrays instead of continuous functions, so we discretize the concept of convolution.

Definition 5 *Given two 2-dimensional arrays* $\mathbf{F}$ *and* $\mathbf{G}$ *(potentially of infinite length), their* ***discrete convolution*** *is*

$$\mathbf{H}_{m,n} = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} \mathbf{F}_{i,j}\mathbf{G}_{m-i,n-j}, \quad m,n \in \mathbb{Z}. \tag{43}$$

To better understand how we can apply convolution to images, let's start with the 1-dimensional case.
Imagine that we have an exact signal $\mathbf{x} \in \mathbb{R}^5$, and a PSF $\mathbf{p} \in \mathbb{R}^5$ that blurs it in a way such that each element of the blurred signal depends on the corresponding element in the exact one, but also on the previous and on the following ones. So we have:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix}, \quad \mathbf{p} = \begin{bmatrix} 0 \\ p_{-1} \\ p_0 \\ p_1 \\ 0 \end{bmatrix}. \tag{44}$$

The blurred signal $\mathbf{b}$ is defined by their convolution:

$$b_j = \sum_u p_{j-u}x_u\,. \tag{45}$$

This is straightforward for the entries b_2, b_3 and b_4:

$$b_2 = p_{2-1}x_1 + p_{2-2}x_2 + p_{2-3}x_3 = p_1x_1 + p_0x_2 + p_{-1}x_3, \tag{46}$$
$$b_3 = p_{3-2}x_2 + p_{3-3}x_3 + p_{3-4}x_4 = p_1x_2 + p_0x_3 + p_{-1}x_4, \tag{47}$$
$$b_4 = p_{4-3}x_3 + p_{4-4}x_4 + p_{4-5}x_5 = p_1x_3 + p_0x_4 + p_{-1}x_5. \tag{48}$$

For b_1 and b_5 we have to add two terms, x_0 and x_6. Their expression is

$$b_1 = p_{1-0}x_0 + p_{1-1}x_1 + p_{1-2}x_2 = p_1x_0 + p_0x_1 + p_{-1}x_2, \tag{49}$$
$$b_5 = p_{5-4}x_4 + p_{5-5}x_5 + p_{5-6}x_6 = p_1x_4 + p_0x_5 + p_{-1}x_6. \tag{50}$$

If we write this again in the classic linear algebra notation, the problem is represented by the system of linear equations:

$$\begin{bmatrix} p_1 & p_0 & p_{-1} & 0 & 0 & 0 & 0 \\ 0 & p_1 & p_0 & p_{-1} & 0 & 0 & 0 \\ 0 & 0 & p_1 & p_0 & p_{-1} & 0 & 0 \\ 0 & 0 & 0 & p_1 & p_0 & p_{-1} & 0 \\ 0 & 0 & 0 & 0 & p_1 & p_0 & p_{-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{bmatrix}. \tag{51}$$

This is an **undetermined system**, because it has 7 unknowns with only 5 equations.
In the next section we will see how to deal with this kind of problem.

2.5 Boundary conditions

We have seen that when computing convolution the indices can go outside of the length of the arrays, generating more unknowns.
This problem can be avoided making assumptions on those values. These assumptions are called **boundary conditions** (BC) and can be made in different ways. We will see that different boundary conditions can have computational advantages or disadvantages, but also that depending on the problem that we are facing can produce more or less accurate results. We discuss the most popular boundary conditions choices.

2.5.1 Zero boundary conditions

The first boundary condition that we describe is probably the easiest and the most intuitive one. It consist in considering as zeros all the entries of the signals or the images that are unknowns as *zeros*.
In the example (51), we have $x_0 = x_6 = 0$. In this case the system becomes

$$\begin{bmatrix} p_0 & p_{-1} & 0 & 0 & 0 \\ p_1 & p_0 & p_{-1} & 0 & 0 \\ 0 & p_1 & p_0 & p_{-1} & 0 \\ 0 & 0 & p_1 & p_0 & p_{-1} \\ 0 & 0 & 0 & p_1 & p_0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{bmatrix}. \tag{52}$$

Now the linear system has 5 unknowns and 5 equations, and the coefficients matrix has the structure of a **Toeplitz matrix**, which means that its entries are constant on each diagonal. For this structure there exist very specialized algorithms [4].
Generally speaking, for image deblurring, this choice means assuming that everything that stands outside of the scene that we caught with the camera is black. This can be quite a good approximation in some cases, for example for astronomic pictures, that we usually have a black background, but can be very far from reality in lots of situations, where the edges of the picture are far from being black, and so it is true for what is just outside of the shot.

2.5.2 Periodic boundary conditions

We can also assume that the signal is periodic, which means that it repeats its behaviour multiple times.
In our example it would mean that $x_0 = x_5$ and $x_6 = x_1$, with the linear system becoming

$$\begin{bmatrix} p_0 & p_{-1} & 0 & 0 & p_1 \\ p_1 & p_0 & p_{-1} & 0 & 0 \\ 0 & p_1 & p_0 & p_{-1} & 0 \\ 0 & 0 & p_1 & p_0 & p_{-1} \\ p_{-1} & 0 & 0 & p_1 & p_0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{bmatrix}. \tag{53}$$

In this case the coefficients matrix is a **circulant matrix**, which means that all the rows are composed by the same elements of the previous one, shifted to the right by one position (wrapping around "cyclically" at the edges).
This kind of matrices have very interesting properties, such as the fact that their eigenvectors are Fourier base vectors:

$$\mathbf{v}^{(s)} = \left[e^{-2\pi i s/n} \; e^{-2\pi i 2s/n} \dots e^{-2\pi i (n-1)s/n} \right], \tag{54}$$

and their eigenvalues are the Fourier coefficients of the first column of the circulant matrix. This means that, given a circulant matrix C, we can compute its spectral factorization

$$C = F^* \Lambda F, \tag{55}$$

where Λ is a diagonal matrix, containing the eigenvalues of C, and F is the eigenvectors matrix, using the **Discrete Fourier Transform** (DFT).
This can be used to reduce the computational cost of the matrix-vector product $C\mathbf{w}$.
Computing

$$C\mathbf{w} = F^* \Lambda F \mathbf{w}, \tag{56}$$

we have that

- $F\mathbf{w}$ is computed using the *fast Fourier trasform* (FFT) algorithm, and it costs $O(n\log(n))$ flops.
- $\Lambda(F\mathbf{w})$ costs $O(n)$ flops, because Λ is diagonal.
- $F^*(\Lambda(F\mathbf{w}))$ is computed using the *inverse fast Fourier trasform* (IFFT) algorithm, and it costs $O(n\log(n))$ flops.

So we can perform the matrix-vector product with just $O(n\log(n))$ flops, instead of the $O(n^2)$ required for a generic matrix.

2.5.3 Reflexive boundary conditions

Another option is to assume that what is outside of the signal the we want to reconstruct is a mirrored version of the signal itself. In our example, this means that $x_0 = x_1$ and $x_6 = x_5$. So the linear system is now:

$$\begin{bmatrix} p_0+p_1 & p_{-1} & 0 & 0 & 0 \\ p_1 & p_0 & p_{-1} & 0 & 0 \\ 0 & p_1 & p_0 & p_{-1} & 0 \\ 0 & 0 & p_1 & p_0 & p_{-1} \\ 0 & 0 & 0 & p_1 & p_0+p_{-1} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{bmatrix}. \qquad (57)$$

With periodic boundary conditions the coefficients matrix has the structure of a **Toeplitz** (constant on the diagonals) plus a **Hankel matrix** (constant on the anti-diagonals). There exists fast algorithms for solving such linear systems.

2.5.4 Two-dimensional problems

What we said before for 1-D signal can be extended to image deblurring, which means that we have to deal with 2-dimensional arrays.
While computing convolution we still have to face the problem of choosing what to assume for the behaviour of the image outside of the edges. Following the same ideas the we described in the previous section, we have to set boundary conditions. Now, if we set **zero boundary conditions**, we will find a coefficients matrix that is Block Toeplitz with Toeplitz Blocks (**BTTB**), which means that the blocks forms a Toeplitz structure, and each block is a Toeplitz matrix itself. With **periodic boundary conditions** we have a Block Circulant with Circulant Blocks (**BCCB**) matrix and with **reflexive boundary conditions** the coefficients matrix is the sum of **BTTB**, **BTHB** (Block Toeplitz with Hankel Blocks), **BHTB** (Block Hankel with Toeplitz Blocks), and **BHHB** (Block Hankel with Hankel Blocks) matrices.

3 CUQI project

CUQI is a research project (6) dedicated to creating a mathematical, statistical, and computational framework for the implementation of uncertainty quantification (UQ) in solving inverse problems.

CUQI

Figure 6: CUQI project logo

As we said previously, inverse problems determine hidden information from measurements in, e.g., deconvolution, image deblurring, computed tomography, source reconstruction, and fault inspection, so the variety of fields that can make use of inverse problems is very broad.
The whole project has been carried on with the idea of creating a strong theoretical and computational base to quantify how the imprecision in the measurements, in the model, and any other possible error influence the solutions to inverse problems. To do this it is necessary to work on the mathematical and statistical foundation of uncertainty quantification, but also to create a user-friendly software package, designed to enable both experts and non-experts to apply UQ to their inverse problems. In fact, it has been developed CUQIpy, a software package for UQ modeling and computations in order to allow end-users of inverse problems, in science and engineering, to quantify the accuracy in their solutions, and in this way lower the risks and take more correct decisions.
Based on the work of all the people involved in the project, the aim is that UQ will become an integral component of solving inverse problems, both in science and engineering.

3.1 Examples of research in CUQI

Uncertainty quantification (UQ) serves as a valuable tool for evaluating the reliability of a reconstruction, which is the solution to an inverse problem. It provides insights into the degree of confidence we can place in the reconstruction and its finer details. UQ finds practical utility across a wide spectrum of disciplines, including science, engineering, medical imaging, and various other domains where the goal is to reveal hidden information through solving inverse problems. For instance, UQ can be applied in industrial inspection to detect anomalies or defects in objects, in medical imaging to identify malignant tissue, and in acoustics to locate the origins of unwanted sound sources. In the following section we will give a practical example of the use of the tools that we have been describing.

3.1.1 Defect Detection in X-Ray CT of Subsea Pipes

X-ray computed tomography (CT) is used to monitor the condition of subsea oil or gas pipes in operation, aiming to identify potential flaws that could lead to leaks. This example [5] is based on data obtained in the test facilities at FORCE Technology. The computational aspects of this process are executed using the CUQIpy software.
It is clear that we are facing an inverse problem, as the measurements that we would like to have is the condition of the subsea pipes, but, for obvious reasons, we cannot have it directly without breaking them. To still get information about the composition of the pipes, we use the fact that different materials cause different attenuation in the X-ray the we send through them. For this reason, we formulate a Bayesian inverse problem with built-in defect detection. The goal is not only to detect defects, but also to quantify their uncertainties.
We model this problem as

$$\mathbf{y} = A(\mathbf{x} + \mathbf{d}) + \mathbf{e}, \tag{58}$$

where $\mathbf{y}$ is the measured X-ray absorption, $\mathbf{e}$ is data noise, and A denotes the linear forward model representing the physics and geometry of the measurements.
We use a representation $\mathbf{x} + \mathbf{d}$ of the unknown image to be reconstructed, where:

- $\mathbf{x}$ contains the pipe structure,
- $\mathbf{d}$ contains potential defects.

In the Bayesian framework, we have to compute the joint posterior distribution representing the solution to the CT problem, that is given by the expression

$$p(\mathbf{x}, \mathbf{d}|\mathbf{y}) \propto p(\mathbf{y}|\mathbf{x}, \mathbf{d})p(\mathbf{x})p(\mathbf{d}), \tag{59}$$

where $p(\mathbf{y}|\mathbf{x}, \mathbf{d})$ is the likelihood that represents the data misfit, while $p(\mathbf{x})$ and $p(\mathbf{d})$ are the prior distributions representing the prior information that

we have about the unknown images.
We assume priors that encourage the structures we are looking for. For the pipe structure **x**, we use a Gaussian prior. For the defects, we expect them to be small and not so many, and therefore we set a prior that promotes both correlation and sparsity in the defect image **d**, a gamma Markov random field (as described in the paper [5] by Christensen, Riis, Pereyra and Jørgensen). Figure 7 reports the means of the **x**-samples (left), the **d**-samples (middle), and their sum with annotation on the detected defects(right). These outcomes affirm the effectiveness of our approach in effectively distinguishing defects from the broader pipe structure.
In Figure 8, we narrow our focus to the reconstruction of defects labeled in

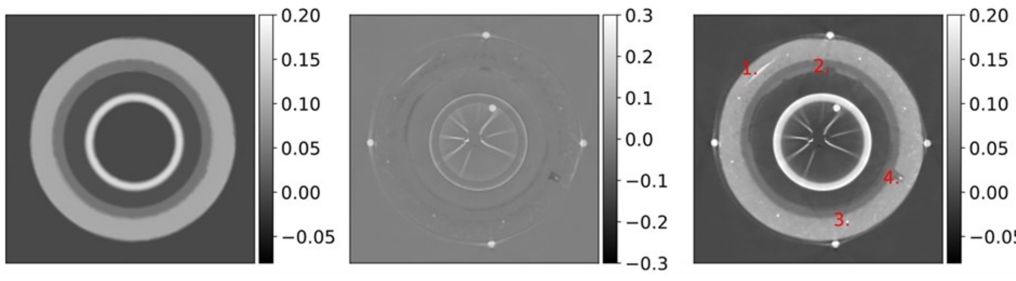


Figure 7: *left:* **x** samples mean, *middle:* **d** samples mean, *right:* their sum

the figure above, allowing for a more in-depth examination of these anomalies. The figure shows the mean of the posterior samples, accompanied by related UQ represented by the standard deviation of the samples.
Engineers can use these results to identify critical defects in the subsea pipes

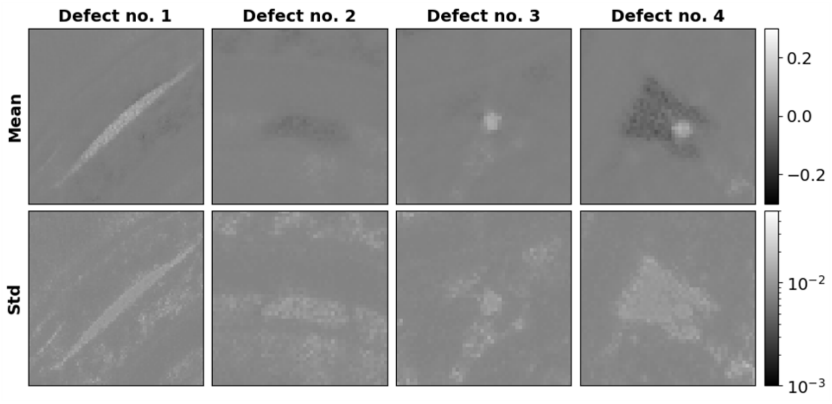


Figure 8: Mean and standard deviation of the samples in the defects

and eventually plan any maintenance intervention, knowing precisely where the damage are and how sure we are about the presence of them.

3.2 CUQIpy

The field of *computational UQ for inverse problems* is experiencing a rapid expansion, driven by the evolution of both new theories and methodologies. This growth is primarily focused on providing for efficient computational approaches for conducting UQ on large-scale inverse problems. While several software packages have emerged to address forward and inverse UQ, examples being UQLab [10], SIPPI [8] and MUQ [11], these packages frequently target specific applications, limiting their universality. Notably, there seems to be a shortage of UQ software packages capable of handling large-scale imaging problems, such as X-ray computed tomography (CT) and image deblurring.. Hence, **CUQIpy** (9) has been developed to address this gap. CUQIpy is an open-source Python [14] package designed for computational uncertainty quantification (UQ), with a specific focus on a variety of inverse problems related to imaging tasks. CUQIpy incorporates a range of efficient computational techniques. Users can define a Bayesian inverse problem (or choose from the built-in options) and subsequently conduct UQ calculations using the available methods. CUQIpy serves a dual purpose: it aims to assist individuals who may not be well-versed in Bayesian inference while also offering advanced users the freedom and comprehensive control over computational methodologies. Central to CUQIpy is a high-level modeling framework made for handling inverse problems within the Bayesian context. This framework incorporates a syntax that closely aligns with the underlying mathematical and statistical principles, enabling users to succinctly and intuitively define their inverse problem, provide prior information, and specify other essential statistical details.

Figure 9: CUQIpy logo

Additionally, CUQIpy includes an array-agnostic modeling framework, allowing users to replace conventional array libraries like **NumPy** [9] with alternative libraries like **PyTorch** [12]. This is done to benefit from GPU acceleration, which is fundamental to make CUQIpy able to support large-scale imaging-type inverse problems, such as computed tomography or image deblurring.

3.2.1 Specifying and solving Bayesian Inverse Problems

In the following sections we describe the most important software components of CUQIpy. Initially, we offer an outline of the package, followed by a systematic exploration of essential tools. This step-by-step approach aims to clarify the standard procedure for defining and resolving a Bayesian inverse problem.
The CUQIpy package is available from https://cuqi-dtu.github.io/CUQIpy/, along with information on how to get started, full documentation, and numerous demos and tutorials ([13], [1]).
Installation is straightforward using the Python package installer:

```
pip install cuqi
```

Once CUQIpy is installed, the user can import the complete package by

```python
import cuqi
```

or individual components directly. For example, these are components that we will use later:

```python
from cuqi.testproblem import Deconvolution1D, 
    Deconvolution2D
from cuqi.distribution import Gaussian, Gamma, 
    GMRF, JointDistribution
from cuqi.problem import BayesianProblem
from cuqi.sampler import UGLA
import numpy as np
```

CUQIpy provides a large variety of test problems, which contain pre-defined, configurable test problems. This test problems are linear deconvolution problems like 1D signal deblurring, 2D image deblurring, but there are also examples of PDEs problems, like Discrete Heat problem and Discrete 1D Poisson problem.

3.2.2 Basic Example

To illustrate the basic usage of CUQIpy, let's examine a 2D deconvolution problem, which can be formulated as the linear inverse problem $A\mathbf{x} = \mathbf{y}$. In this context, matrix A (assumed to be known) serves as the forward model,

where, in the case of 2D deconvolution, A represents the blurring caused by a point spread function. The vector $\mathbf{y}$ is a random variable, representing the blurred and noisy image, and we have a specific observed instance denoted as $\mathbf{y}^{\mathbf{obs}}$.
Given this forward model and observed data, our objective is to infer the sharp image represented by the random variable $\mathbf{x}$.
We load a specific example of a forward model and observed data from the collection of test problems available in CUQIpy:

```
A, y_obs, info = Deconvolution2D.get_components(
    dim=256, phantom="cookie")
```

In CUQIpy, a modeling language is at the user disposal to articulate the knowledge or assumptions regarding parameters in the context of a Bayesian Problem. In our current scenario, we presume that the data $\mathbf{y}$ is subject to additive Gaussian white noise with a mean of zero and an unknown precision represented by s, which follows a Gamma distribution. As for the image $\mathbf{x}$, we adopt a prior based on an edge-preserving Laplace Markov Random Field (LMRF), characterized by an unknown scale parameter d^{-1}, where d is also assumed to follow a Gamma distribution. Consequently, the Bayesian Problem is configured as follows:

$$d \sim \mathrm{Gamma}(1, 10^{-4}), \tag{60}$$
$$s \sim \mathrm{Gamma}(1, 10^{-4}), \tag{61}$$
$$\mathbf{x} \sim \mathrm{LMRF}(d^{-1}), \tag{62}$$
$$\mathbf{y} \sim \mathrm{Gaussian}(A\mathbf{x}, s^{-1}I). \tag{63}$$

The parameters d and s are called hyperparameters. In CUQIpy the syntax closely matches the mathematical specification of the Bayesian Problem:

```
d = Gamma(1, 1e-4)
s = Gamma(1, 1e-4)
x = LMRF(1/d, geometry=A.domain_geometry)
y = Gaussian(A @ x, 1/s)
```

Here the geometry keyword is used to specify that the LMRF prior should be defined on the domain of A, which is represented by a so-called *Image2D* geometry. After specifying the Bayesian Problem, we can ask CUQIpy to perform a UQ analysis of the problem:

```
BP = BayesianProblem(d, s, x, y) # Combine to
    Bayesian Problem
BP.set_data(y=y_obs) # Specify observed data
BP.UQ() # Run UQ analysis
```

The UQ() method analyzes the problem, selects a suitable sampler, samples the posterior distribution, and returns a summary and selected visualizations of the results, as shown in Figure 10. The posterior width, which serves as

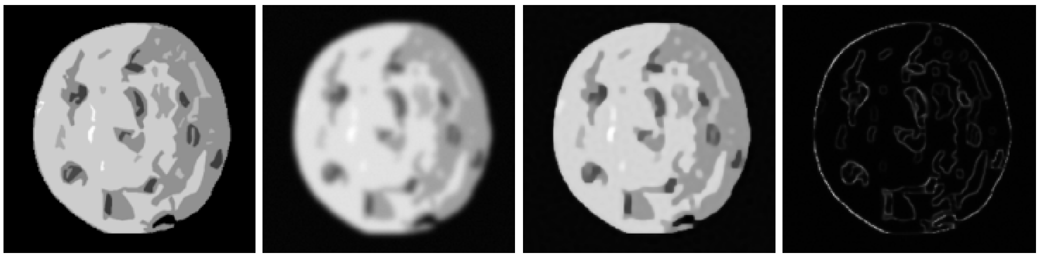


Figure 10: Far left: True sharp image. Middle left: Observed blurred and noisy image. Middle right: Deblurred image (posterior mean). Far right: Quantification of uncertainty (posterior width).

an indicator of the uncertainty in the deblurring process, highlights that the regions containing edges have the greatest degree of uncertainty. Furthermore, in addition to estimating the image $\mathbf{x}$, CUQIpy also enables the estimation of the hyperparameters d and s, along with quantifying their respective uncertainties. The true value of s is defined to be $7.716 \cdot 10^4$ in the Deconvolution2D test problem, and the posterior mean and 99% credibility interval are found to be $7.639 \cdot 10^4$ and $[7.535 \cdot 10^4, 7.744 \cdot 10^4]$. A true value for d is not known but its posterior mean and credibility interval are found to be $3.871 \cdot 10^1$ and $[3.824 \cdot 10^1, 3.918 \cdot 10^1]$.

We have shown an example of the high-level usage of CUQIpy, where the user do not need to select a specific sampler or tune any parameters since CUQIpy handles these tasks automatically, based on the problem structure. CUQIpy allows users to fully select and configure problem specification, samplers, visualization, etc., in order to improve the performance for expert users.

4 Fast Convolution Algorithms for Image Deblurring

We have described so far the idea behind Bayesian Inverse Problems, the basics of the problem of Image Deblurring and what the CUQI project, with its Python package CUQIpy are. Said that, we have to face with problems and difficulties that shows up in this kind of environment.
The work done for this thesis was directed to reducing computational time for image deblurring, this was achieved under some circumstances, that will be shown later in this section.

The computational time is a big issue in Bayesian inverse problems, especially, but not only, for large-scale problems. This happens because the solution that we compute is not a deterministic value, but it is a random variable. So after computing its posterior distribution we have to generate samples in order to determine useful estimator such as its mean and standard deviation. When computing these values, if we want them to be reliable and with the least error possible, we have to generate a lot of samples, a common choice is 1000, so the processes have to be repeated many times. This explain why even a small gain in a quite fast process like convolution can lead to a large save of time in the whole computation.

4.1 Test problems

As we said, the CUQIpy package had already the tools to solve image deblurring problems. Our goal was to make some little changes in the implementation in order to make the computation faster. Now we see precisely how to solve this kind of problem in CUQIpy and where the improvement was made. First of all, we import the necessary functions and packages, both from CUQIpy and from external libraries ([15], [9]):

```
    import numpy as np
    import matplotlib.pyplot as plt
    from scipy.fft import fft2, ifft2, fftshift,
       ifftshift
    from scipy.signal import fftconvolve, convolve

    import cuqi
    from cuqi.model import LinearModel
    from cuqi.geometry import Image2D
    from cuqi.array import CUQIarray
```

```python
from cuqi.distribution import Gaussian, LMRF,
   Gamma
from cuqi.problem import BayesianProblem
```

Then we set up the basic parameters for the problem. We also set up the "geometry" in which the problem is handled. The *Image 2D* geometry represents a pixeled image and describe the kind of object that will be passed to the forward model.

```python
# Parameters
dim = 256   # Image dimension
BC = "constant" # Boundary condition
P = cuqi.testproblem.Deconvolution2D(PSF_size =
   41).Miscellaneous["PSF"] # Extract PSF from
   testproblem. Can use other PSF types
noise_std=0.0036 # Noise standard deviation

# Set-up for 2D image
img_geo = Image2D((dim, dim))
```

We need an image to run our example on. We will start with an image taken from CUQI.DATA (Figure 11), but any custom image can be used. The image is wrapped into a CUQIarray for easy plotting.

```python
test_image = CUQIarray(cuqi.data.satellite(dim),
    is_par=False, geometry=img_geo)
test_image.plot();
```

We then define the convolution forward model the way that it was done in the old CUQIpy code.

```python
# Old convolution wrapped into CUQIpy model "A"
def _proj_forward_2D(X, P, BC):
    PSF_size = max(P.shape)
    X_padded = np.pad(X, PSF_size//2, mode=BC)
    Ax = convolve(X_padded, P, mode='valid')
    if not PSF_size & 0x1: # If PSF_size is even
        Ax = Ax[1:, 1:] # Remove first row and
            column to fit convolve math
    return Ax

def _proj_backward_2D(B, P, BC):
    P = np.flipud(np.fliplr(P)) # Flip PSF
    return _proj_forward_2D(B, P, BC)
```

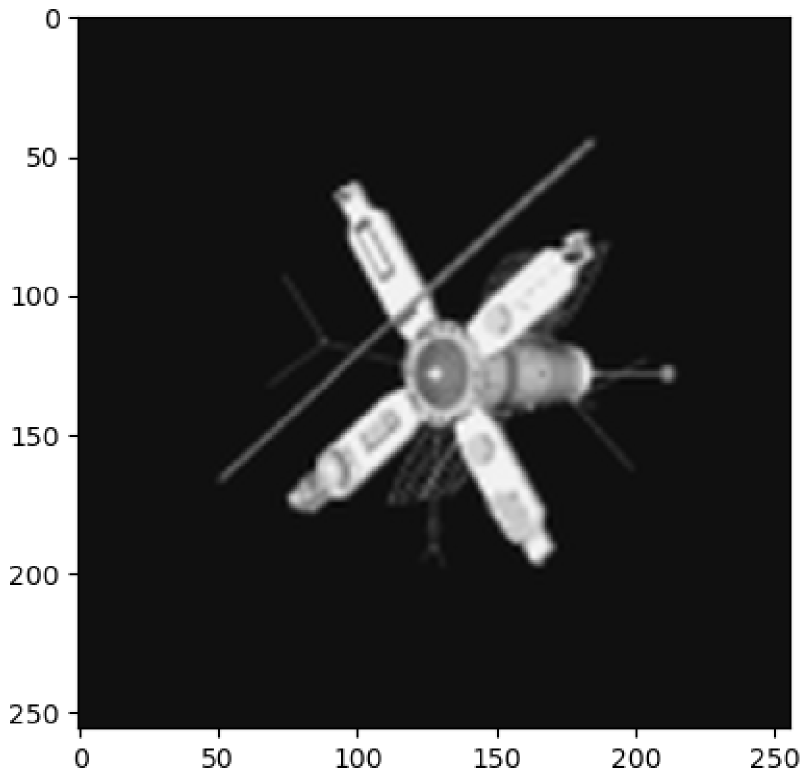


Figure 11: Sharp image of a satellite, 256x256 pixels.

```
    
    A = cuqi.model.LinearModel(
        forward=lambda x: _proj_forward_2D(x, P, BC)
           ,
        adjoint=lambda y: _proj_backward_2D(y, P, BC
           ),
        domain_geometry=img_geo,
        range_geometry=img_geo,
    )
```

This implementation, in the _PROJ_FORWARD_2D function, computes convolution using the Scipy "**convolve**" function. The good thing about this function is that allows us to perform convolution for every boundary condition, padding the image with the values that we need for each case.
After defining the forward model, we run the UQ method which does Bayesian sampling using CUQIpy's convolution.

```
    # Parameter assumptions
    x = LMRF(0, 0.1, geometry=img_geo)
    y = Gaussian(A@x, noise_std**2, geometry=img_geo
       ) # old forward model

    # Generate synthetic blurred image with noise by
    # conditioning y on x=test_image and sampling
    y_data = y(x=test_image).sample()
    y_data.plot()
    plt.title("Blurred␣image␣with␣noise")
    plt.show()

    # Now set up Bayesian Problem and run UQ
    BP = BayesianProblem(x, y)
    BP.set_data(y=y_data)
    BP.UQ(1000, exact=test_image)
```

So, the first thing that we plot is the blurred satellite, with Gaussian noise (Figure 12). This is our observed data that we have denoted by $\mathbf{y}^{\mathbf{obs}}$.

Then, with some assumption on the prior distribution, we are able to

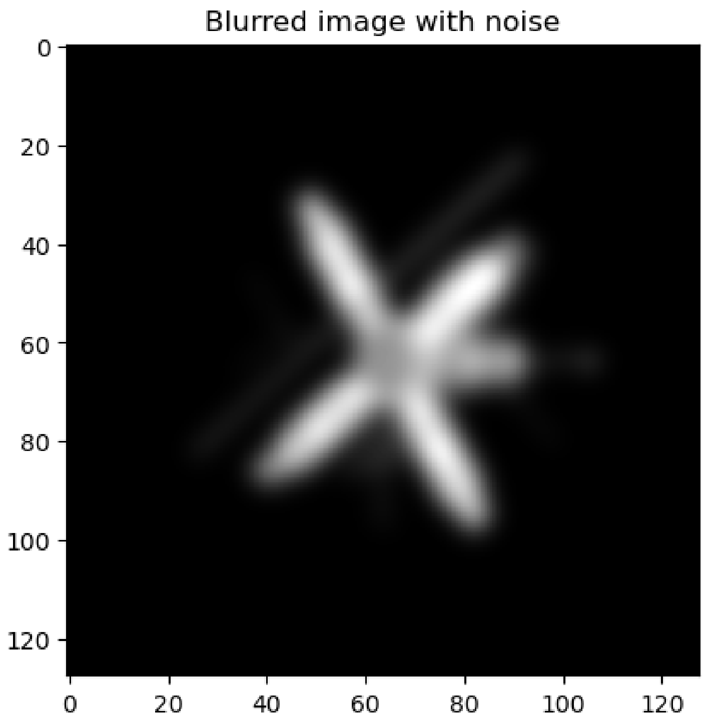


Figure 12: Blurred image of a satellite.

compute the posterior distribution of the reconstructed image $\mathbf{x}$. From this distribution we take 1000 sample and we compute their mean (Figure 13) and width of credibility interval (Figure 14). In the image representing the credibility interval, white stands for high uncertainty and black for low uncertainty. This shows that on the edges is where we are less confident about our reconstruction.

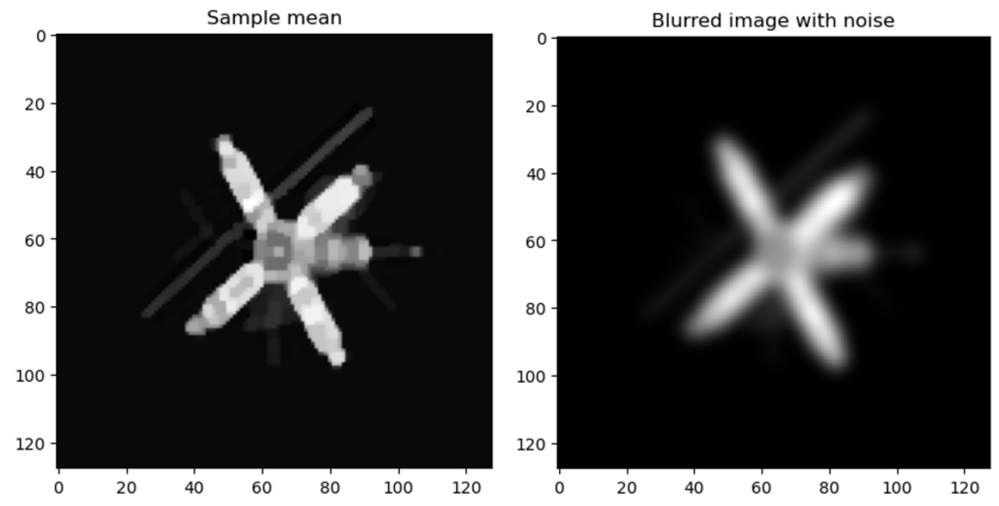


Figure 13: Left, sample mean of the reconstructed image. Right, blurred image.

4.2 New Implementation

As we can see in the function _PROJ_FORWARD_2D, in the old implementation, we padded the image with half of the PSF size in each direction, to incorporate the boundary conditions into our problem.
This means that we have to compute convolution on a matrix that is larger than the starting image, and then we reduce get the dimension back to the original size by setting the parameter *mode* = '*valid*'. The difference can be very large when the PSF is big.
What we can do to make the computation faster is to try to use the properties of the structured matrices that we have, along with the Scipy functions, to avoid padding the image, at least for some boundary condition.

4.2.1 Zero Boundary Conditions

For **zero** BC, we just used the fact that the Scipy function CONVOLVE is already implemented to assume zero boundary condition. So in the new

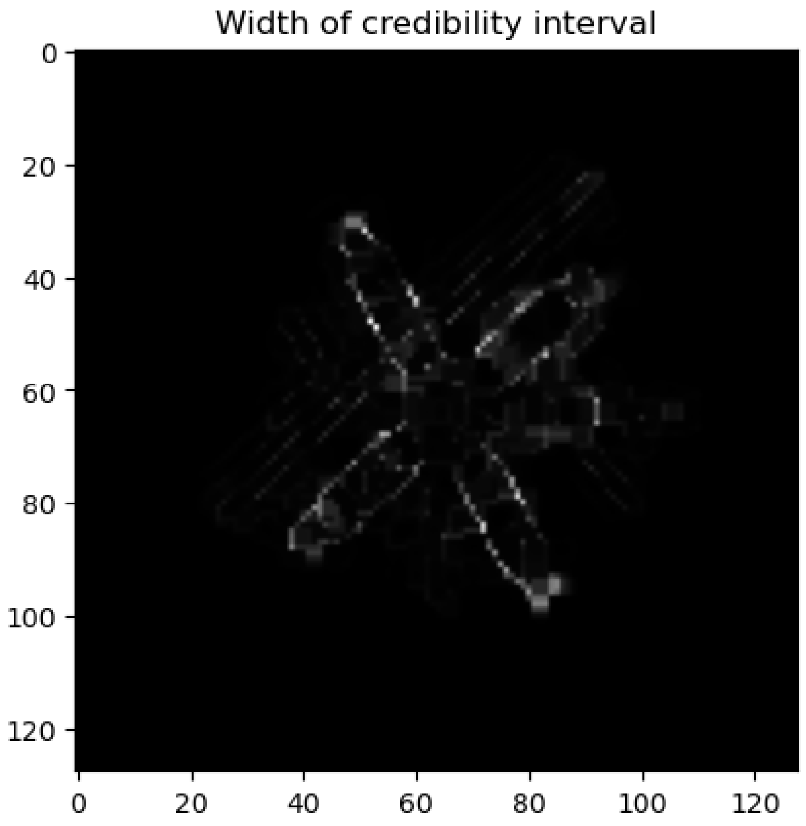


Figure 14: Credibility interval.

implementation we just use the Scipy function CONVOLVE, but this time with no padding for the image, and with the parameter *mode* = '*same*' that maintains the dimension of the convolved image equals to the input one.

```
    Ax = convolve(X,P,'same')
```

4.2.2 Periodic Boundary Conditions

For **periodic** BC the changes are more significant. We use a very important property of convolution, that stands with periodic boundary condition.
We have that, given

$$\mathbf{B} = \mathbf{X} * \mathbf{P}, \tag{64}$$

where $*$ represents 2-dimensional convolution. Denoting $\hat{\mathbf{B}}$ the Discrete Fourier Transform (DFT) of $\mathbf{B}$, we have:

$$\hat{\mathbf{B}} = \hat{\mathbf{X}} \odot \hat{\mathbf{P}}, \tag{65}$$

where $\odot$ represents the element-wise product.
So from the relation (65) we can compute the convolution between $\mathbf{X}$ and $\mathbf{P}$ just by computing the Discrete Fourier Transform of the two arrays with the fast fourier transform algorithm (fft), then we perform the element-wise product, and finally we compute the inverse DFT, again with the fft algorithm. This relation is very important because it allows to perform convolution in a much faster way. For example, for squared image with $N = n^2$ pixels, as it is defined, the convolution has a computational cost of $O(N^2)$ flops, while with the alternative method that we have just described we can compute it with $O(N \log(N))$ floating point operations.
One important aspect is that to perform the element-wise product as described before the two matrices need to have the same shape. In practice the PSF is always smaller than the image, so we have to pad it with zeros in order to perform convolution this way. So the new implementation for periodic boundary conditions is

```
1     if (X.shape[0]%2) == 0 and (P.shape[0]%2) == 0 : # Both even
2         P = np.pad(P,(X.shape[0]-P.shape[0])//2,mode='constant')
3     elif (X.shape[0]%2) == 1 and (P.shape[0]%2) == 1 : # Both odd
4         P = np.pad(P,((X.shape[0]-P.shape[0])//2 + 1, (X.shape[0]-P.shape[0])//2 - 1),mode='constant')
5     else: # One even and one odd
6         P = np.pad(P,((X.shape[0]-P.shape[0])//2 + 1, (X.shape[0]-P.shape[0])//2),mode='constant')
7     Ax = np.real(ifft2(fft2(X)*fft2(fftshift(P))))
```

As we said, after padding the PSF, we compute convolution (line 7) with the method described before. The NP.REAL function is necessary because when computing the DFT we convert the real matrices to complex ones. The complex part should be canceled with the Inverse DFT, but due to rounding errors, it doesn't completely go away, so we use this technique to have a purely real result.

4.3 Numerical experiments

Now we can compare the old and the new implementation, for both zero and periodic boundary conditions, testing them with different images and

different PSFs.
As we did for the old code, now we have to define the new forward model.

```python
# New convolution wrapped into CUQIpy model "A_new"
def _new_proj_forward_2D(X, P, BC):
    if BC == 'wrap': # Periodic BC
        if (X.shape[0]%2) == 0 and (P.shape[0]%2) == 0 : # Both even
            P = np.pad(P,(X.shape[0]-P.shape[0])//2,mode='constant')
        elif (X.shape[0]%2) == 1 and (P.shape[0]%2) == 1 : # Both odd
            P = np.pad(P,((X.shape[0]-P.shape[0])//2 + 1, (X.shape[0]-P.shape[0])//2 - 1),mode='constant')
        else: # One even and one odd
            P = np.pad(P,((X.shape[0]-P.shape[0])//2 + 1, (X.shape[0]-P.shape[0])//2),mode='constant')
        Ax = np.real(ifft2(fft2(X)*fft2(fftshift(P))))

    elif BC == 'constant': # Zero BC
        Ax = convolve(X,P,'same')

    else: Ax = _proj_forward_2D(X, P, BC) # Other BC
    return Ax

def _new_proj_backward_2D(B, P, BC):
    P = np.flipud(np.fliplr(P)) # Flip PSF
    return _new_proj_forward_2D(B, P, BC)

A_new = cuqi.model.LinearModel(
    forward=lambda x: _new_proj_forward_2D(x, P, BC),
    adjoint=lambda y: _new_proj_backward_2D(y, P, BC),
    domain_geometry=img_geo,
    range_geometry=img_geo)
```

The new forward model uses the techniques described in the previous sections for zero and periodic boundary conditions, and calls the old function to perform convolution for the other possibilities. In the NumPy sintax, periodic BC are called “wrap” and zero BC are called “constant”.
In all the following experiments we will compute 1200 samples, and the first 200 are discarded due to numerical reasons related to the sampling methods. This choice is a compromise between having reliable results for the mean and standard deviation and having reasonable computational time.

4.3.1 Zero boundary conditions

As we said before, zero boundary conditions work well with astronomic images, where the assumptions of having a black background is not so far from reality. For this, we have tested the code for the satellite image in Figure 11. We tested it reshaping the image to be 128×128 and 256×256. We considered a Gaussian blurring (Figure 12), using different sizes of the PSF. We compared the old CUQIpy implementation and the new one. For the smaller image, 128×128 the computational times (in seconds) are shown in Table 1. We see that we had a good improvement in computational times,

Size of PSF	**CUQIpy**	**New impl.**	**% gain**
20	188	148	21.2 %
30	198	157	20.7 %
40	262	178	32.1 %
50	288	192	33.3 %

Table 1: Computational times for the 128×128 satellite image

especially when the PSF was larger. This is what we expected to happen, because the old implementation pads the image with half of the size of the PSF rows and columns in both directions and compute the convolution on the padded image, while in the new implementation we compute convolution directly on the original image. So As the PSF gets larger, we gain more advantage.
The same things happens for the 256×256 image. We see the results in Table 2.

4.3.2 Periodic boundary conditions

For periodic boundary conditions we changed the test image (Figure 15). Also this image was taken from CUQI.DATA. Periodic BC gave good results in the reconstruction. We ran the same experiments as for zero boundary

Size of PSF	**CUQIpy**	**New impl.**	**% gain**
20	746	654	12.3 %
30	801	714	10.9 %
40	880	779	11.4 %
50	973	794	18.3 %

Table 2: Computational times for the 256 $\times$ 256 satellite image

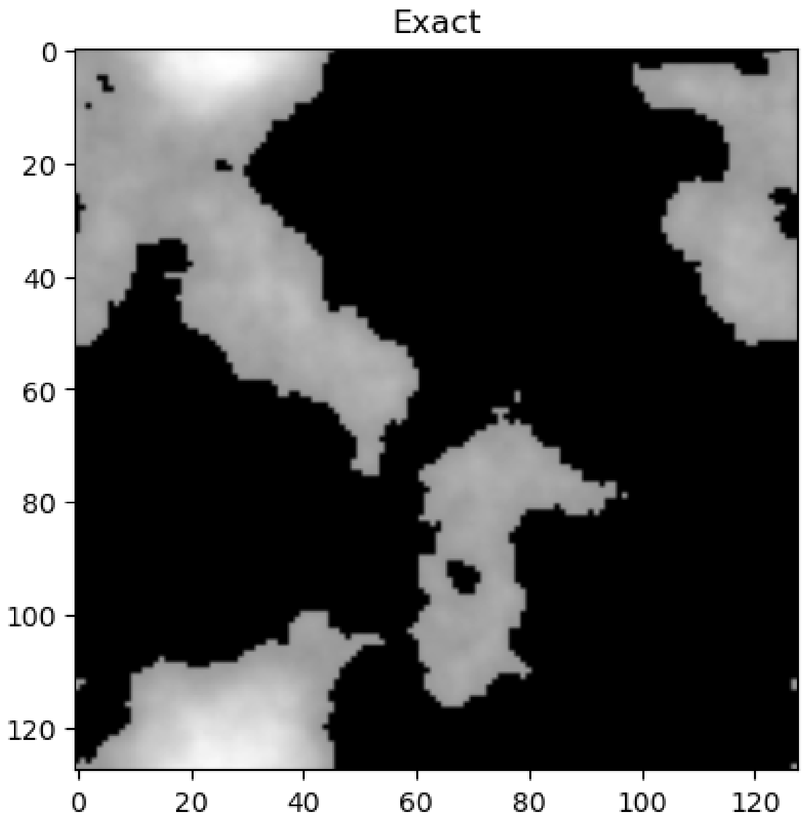


Figure 15: Test image for periodic BC.

condition, so we started with a 128 $\times$ 128 of this image. In Figure 16 we can see how well we were able to reconstruct the image. In Table 3 we can see how we reduced the computational time for the 128 $\times$ 128 image. for periodic BC the new implementation does not depend on the of the PSF, so again the larger the PSF gets, the more advantage we gain. Moving to a 256 $\times$ 256 version of the same image the computation becomes slower, but the difference between the two methods does not change very much, as we can in Table 4.

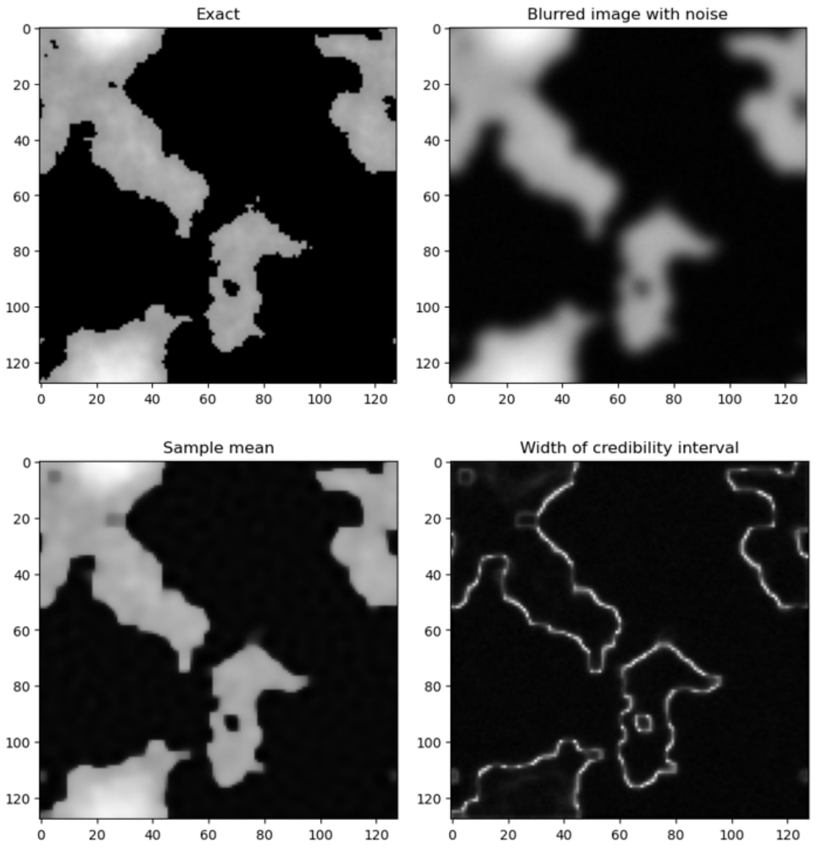


Figure 16: *Top left:* exact image. *Bottom left:* sample mean of the solution. *Top right:* blurred image. *Bottom right:* width of the credibility interval.

Size of PSF	**CUQIpy**	**New impl.**	**% gain**
20	187	146	21.9 %
30	209	149	28.7 %
40	353	146	58.6 %
50	379	147	61.2 %

Table 3: Computational times for the 128×128 image

Size of PSF	**CUQIpy**	**New impl.**	**% gain**
20	771	756	1.9 %
30	836	750	10.2 %
40	951	737	22.5 %
50	1012	767	24.2 %

Table 4: Computational times for the 256×256 image

5 Conclusions

In this work we gave a general explanation of the goals of the CUQI project, along with theoretical insights of inverse problems and the Bayesian approach that is used in the project. In particular we focused on the problem of image deblurring.
In the CUQIpy software there already existed tools to tackle this kind of problem, but due to the high number of sample required to solve effectively a Bayesian inverse problem, the computational time can be very high. For this reason we wanted to write a new implementation that allowed us to reduce the computational time maintaining the good performance that the old one already had.
We were able to do it for some specific cases, which are image deblurring problems with zero and periodic boundary conditions. A possible extension of this work could be to try to reduce the computational times for other boundary conditions, but the two types of boundary conditions the we have succeed on are among the most used ones, so even if we did not manage to improve the performances for all the possibilities, we can still be happy with the result.

References

[1] A. M. A. Alghamdi, N. A. B. Riis, B. M. Afkham, F. Uribe, S. L. Christensen, P. C. Hansen, and J. S. Jørgensen. *CUQIpy – Part II: computational uncertainty quantification for PDE-based inverse problems in Python.* 2023.

[2] J. M. Bardsley. *Computational Uncertainty Quantification for Inverse Problems.* SIAM, 2018.

[3] D. Calvetti and E. Somersalo. *Bayesian Scientific Computing.* Springer, 2023.

[4] R. H. Chan, M. Donatelli, S. Serra-Capizzano, and C. Tablino-Possio. *Application of multigrid techniques to image restoration problems.* Advanced Signal Processing: Algorithms, Architectures, and Implementations XII, Proceeding of SPIE 4791 (2002) pp. 210–221, 2021.

[5] S. L. Christensen, N. A. B. Riis, F. Uribe, and J. S. Jørgensen. *Structural Gaussian priors for Bayesian CT reconstruction of subsea pipes.* Applied mathematics in science and engineering, Vol. 31, 2023.

[6] P. C. Hansen. *Discrete Inverse Problems, Insight and Algorithms.* SIAM, 2010.

[7] P. C. Hansen, J. G. Nagy, and D. P. O'Leary. *Deblurring Images, Matrices, Spectra, and Filtering.* SIAM, 2006.

[8] T. M. Hansen, K. S. Cordua, M. C. Looms, and K. Mosegaard. *SIPPI: A Matlab toolbox for sampling the solution to inverse problems with complex prior information: Part 1–Methodology.* Computers and Geosciences, 2013.

[9] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, Sept. 2020.

[10] S. Marelli and B. Sudret. *UQLab: a framework for uncertainty quantification in MATLAB.* Second International Conference on Vulnerability and Risk Analysis and Management (ICVRAM 2014), 2014.

[11] M. Parno, A. Davis, and L. Seelinger. *MUQ: the MIT uncertainty quantification library.* The Journal of Open Source Software, 2021.

[12] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[13] N. A. B. Riis, A. M. A. Alghamdi, F. Uribe, S. L. Christensen, B. M. Afkham, P. C. Hansen, and J. S. Jørgensen. *CUQIpy – Part I: computational uncertainty quantification for inverse problems in Python.* 2023.

[14] G. Van Rossum and F. L. Drake. *Python 3 Reference Manual.* CreateSpace, Scotts Valley, CA, 2009.

[15] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[16] F. G. Waqar, S. Patel, and C. M. Simon. *A tutorial on the Bayesian statistical approach to inverse problems.* 2023.