PDE Based Image Compression

Thomas Bebbington
2019932

A thesis presented for the degree of
Master in Science

UNIVERSITY OF
BIRMINGHAM

Supervised by Michal Kočvara
Co-supervised by Daniel Loghin

School of Mathematics
University of Birmingham
2023

Acknowledgements

I would like to thank my supervisor, Michal Kočvara for his help and advice throughout this project and my parents, Stephen and Helen for their continued support.

Contents

1 Introduction

Image compression methods are widely used when digitally storing images. The main goal of image compression is to reduce the space needed to store an image, ideally losing as little quality as possible. This is good as it allows one to store more images in the same amount of computer storage and also allows for faster image transmission times over networks. We will be working with bitmap images. A bitmap images is simply a mapping of a grid of square pixels to a set of colours.

There are two types of image compression methods for imges, lossy and lossless. Lossless methods are able to compress images with no loss of information but can only achieve a storage saving of 50% in the best of conditions [19]. On the other hand, lossy methods allow for information loss which allows them to compress images so that they take up significantly less space. In the late 1980s the use of digital images became more common and methods for compression were being developed. The JPEG (Joint Photographic Experts Group) standard was introduced in the early 1990s [22], the JPEG standard uses a discrete cosine transform. The discrete cosine transform is applied to 8×8 pixel "blocks", representing each block as a sum of basis functions with various weights. Since introduction, the JPEG standard has become the most widely used method of image compression [15] and is the default image format for most digital cameras.

Here, we will present a method of compressing bitmap images that involves PDE based inpainting. Compression is done by selecting pixels from the original image and saving the colour of only these pixels. Decompression works by solving an elliptic partial differential equation which relates the colour of pixels which have not had their colour saved to the colour of other pixels in the image. The PDE represents a diffusion process and is solved using a discretisation. Firstly, we will present the notation that we will use, then we will take a brief look at past work on automatic digital inpainting before presenting the compression method. We will then go over some implementation details and also some experiments on various images, analysing the accuracy of the compression method and decompression times.

1.1 Notation

We write vectors in lower boldface, e.g., $\mathbf{u}$, and refer to the ith component of $\mathbf{u}$ by writing $\mathbf{u}_i$. The zero vector will be written as a boldface zero: $\mathbf{0}$. We

will only use column vectors. Matrices are written as upper boldface, e.g., $\mathbf{A}$.

We will be working with iterative methods on linear systems. If we are solving the linear system $\mathbf{Ax} = \mathbf{b}$ for $\mathbf{x}$, we will always write $\mathbf{u}$ for the exact solution and $\mathbf{v}^{(k)}$ for a solution generated by an iterative method after k iterations.

Partial derivatives are written using ∂. Suppose f is a function of x and y, then $\partial_x f$ is the partial derivative of f with respect to x and $\partial_{xx} f$ is the second partial derivative with respect to x. Also, the mixed partial derivative will be written as $\partial_{xy} f$.

Images will be regarded as functions, e.g., $u : \Omega \to R$, where $\Omega \subset \mathbb{R}^2$. The set R is the image codomain and as we will be using grayscale images only, we have $R = \mathbb{R}$. Note that we can also regard this function as a scalar field. To simplify notation, we will often use $u_{i,j} = u(i, j)$ for $i, j \in \mathbb{R}$ and $u_x = u(x)$ for $x \in \mathbb{R}^2$.

Sometimes, we will use the *gradient* of an image which we will define using the ∇ operator. For an image u, we define

$$\nabla u = \begin{pmatrix} \partial_x u \\ \partial_y u \end{pmatrix}. \tag{1.1.1}$$

2 Preliminaries

2.1 Compression Rates

As stated before, we will be working with bitmap images, which are mappings from a grid of square pixels to a set of colours. Computers represent (uncompressed) grayscale images as a two dimensional array of integer values, most commonly using a byte (8 bits) for each value, resulting in 256 different gray values. The number of bits used to store each colour value of an image is called the colour depth, if an image uses n bits for each pixel we say that it is an n-bit image. Colour images are typically represented using three colour channels. These channels give the values for a red, green and blue image, which are combined to give the full colour image. Most colour images are 24-bit, meaning they use 8 bits for each pixel of each colour channel. As an example, a grayscale image that is 500×500 pixels in size would take up $500 \cdot 500 = 250000$ bytes of memory, or equivalently, $250000 \cdot 8 = 2000000$

bits when saved in an uncompressed format. There is a nuance here, as additional data about the image such as colour depth and dimensions of the image which are stored in a section at the start of the image file called a header. However, when talking about compression rates we will not consider headers as they are much smaller in size than the pixel values.

Compression rates describe the ratio of the size of a compressed file to the size of an uncompressed file and are usually written as a ratio of the form p:1 for some $p > 1$, which would be read as "p to 1". Note that it would make sense for us to have a p less than 1, but this would mean that the compressed file would be larger than the uncompressed file, essentially making the compression pointless. To calculate the value of p one can simply divide the size of the uncompressed file by the size of the compressed file. In our context of image compression, we will instead usually express compression rates by the number of bits per pixel (bpp), obtained by dividing the number of bits used to store the compressed image by the total number of pixels in the image. Note that for an 8-bit uncompressed image we would say that it has a "compression rate" of 8 bpp.

2.2 Elliptic Partial Differential Equations

Later on, we will be working with elliptic PDEs (Partial Differential Equations). Here, we will follow [18], defining what elliptic PDEs are and go over some basic theory. We will only be working with PDEs defined on open connected sets Ω, where Ω is a subset of $\mathbb{R}^2$ and $\partial\Omega$ is the boundary of Ω. A standard Cartesian coordinate system x, y will be used.

Elliptic PDEs are second order partial differential equations, such equations have the form

$$a(\partial_{xx}u) + 2b(\partial_{xy}u) + c(\partial_{yy}u) + d(\partial_x u) + e(\partial_y) + fu = g,$$

where u is an unknown function on Ω, with $a, b, \ldots, f, g$ being given functions of x and y. These equations are said to be elliptic if it is the case that

$$b(x, y)^2 - a(x, y)c(x, y) > 0,$$

for all inputs (x, y).

In order to obtain a unique solution to PDEs, we must provide information about the function u on the boundary of its domain $\partial\Omega$. These condition are called boundary conditions. Usually, there are three kinds of boundary

conditions. The first kind are Dirichlet boundary conditions, named after the German mathematician Johann Lejeune Dirichlet, where the value of the function u is given on $\partial\Omega$. The second kind of boundary conditions are called Neumann conditions which give the value of the derivative of u in the direction of the normal to $\partial\Omega$. The third kind of boundary conditions, called conditions of the third kind or sometimes Robin conditions, give the value of a linear combination of Dirichlet conditions and Neumann conditions.

Given an elliptic PDE and boundary conditions, we have what's called an elliptic boundary value problem. In out image compression algorithm, the decompression step will amount to solving an elliptic boundary value problem, with Dirichlet boundary conditions given by the information that was saved in the compression step.

The question of which conditions guarantee the existence of a solution for an elliptic boundary value is a rather hard question to solve and has been considered since the middle of the 19th century[3].

The uniqueness of solutions is much easier to prove and solutions to elliptic PDEs have some nice properties. One of which, called the maximum principle, states that if a solution u exists, then its maximum occurs on the boundary $\partial\Omega$. This supports the use of the PDE decompression method as the recovered image should represent an interpolaiton between the saved points.

2.3 Matrix Properties

Decompression will require solving a system of linear equations, which we will represent using a matrix. Convergence of some iterative methods that we will discuss depends on properties of our matrix, which we will define here. Let $\mathbf{A}$ be a matrix with real entries, height and width n and

$$\mathbf{A} = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & & \vdots \\ \vdots & & \ddots & \vdots \\ a_{n,1} & \dots & \dots & a_{n,n} \end{pmatrix}.$$

We say that $\mathbf{A}$ is symmetric if $a_{i,j} = a_{j,i}$ for all $1 \leq i, j \leq n$, or equivalently, $\mathbf{A} = \mathbf{A}^T$ where $\mathbf{A}^T$ is the transpose of $\mathbf{A}$. Note that the eigenvalues of a symmetric matrix are always real. The matrix $\mathbf{A}$ is said to be diagonally

dominant when

$$|a_{i,i}| \geq \sum_{i \neq j} |a_{i,j}|, \quad \text{for } 1 \leq i \leq n.$$

That is for every row, the absolute value of the entry on the diagonal is greater than or equal to the sum of the absolute value of the other entries in the row.

The next property is positive definiteness. We say that a symmetric matrix $\mathbf{A}$ is positive definite if

$$\mathbf{x}^T \mathbf{A} \mathbf{x} > 0, \text{ for } \mathbf{x} \in \mathbb{R}^n \text{ and } \mathbf{x} \neq \mathbf{0}.$$

Equivalently, a symmetric matrix with strictly positive eigenvalues is positive definite [2]. If it is instead the case that $\mathbf{x}^T \mathbf{A} \mathbf{x} \geq 0$ for $\mathbf{x} \in \mathbb{R}^n \backslash \{\mathbf{0}\}$, then we say that $\mathbf{A}$ is positive semidefinite. A positive semidefinite matrix is positive definite if and only if it is invertible [2].

Finally, the spectral radius of a matrix is the maximum of the absolute values of the matrix's eigenvalues.

3 Image Inpainting

In this section we will look at inpainting as we will use it for the decompression step of our compression method.

Inpainting is a technique originally used by image restorers working in artwork conservation which has been used for a long time [16]. Inpainting is the process of filling in damaged parts of images using methods similar to the original artist, without changing the original, intact parts. Figure 1 shows an example of a piece of artwork before and after restoration using inpainting. This used to be done all by hand, but in the modern day of computers, automatic digital inpainting algorithms have been developed.

The first algorithm for automatic digital inpainting was presented by Bertalmio et al., in [1]. This algorithm works similarly to traditional inpainting techniques by continuing the structure of the area surrounding the region to be filled in, into the region. The goal of the algorithm uses the idea of isophote curves, which are curves of constant colour. Letting Ω be the region to be filled in and $\partial\Omega$ be the boundary of Ω, isophote curves arriving at $\partial\Omega$ should be continued through Ω smoothly, with the direction of arrival

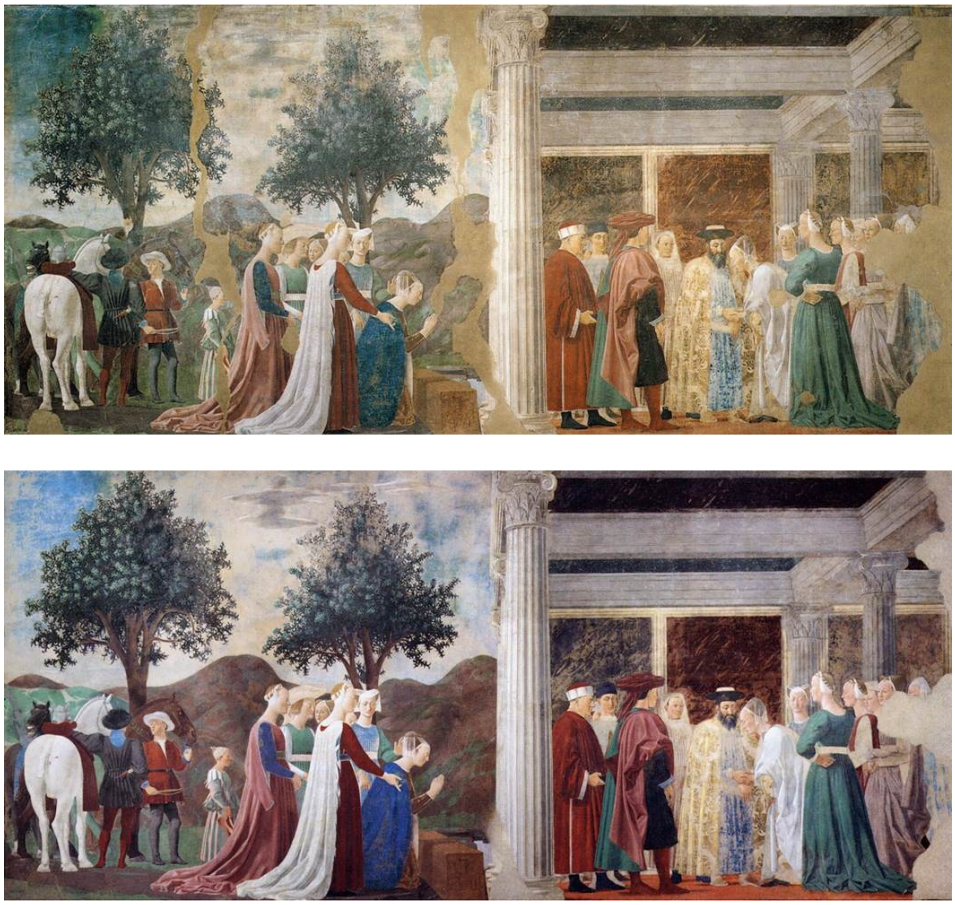

Figure 1: "Scene" by Piero della Francesca[1], before and after restoration using inpainting.

being preserved. The isophote lines should be prolonged such that different isophote curves do not cross each other.

This is done by defining a family of images, where the first image is the original image and subsequent images are generated by an inpainting step. The final output of the algorithm occurs when convergence in this family is reached. The inpainting step involves estimating the variation in image smoothness at a point in Ω using a discretisation of the 2D Laplacian operator. Then, this variation is projected in the direction of the isophote curve's direction. The projection is used to update the value of the points in

[1]Taken from the Web Gallery of Art, www.wga.hu.

Ω. Then, every few steps, a few iterations of a diffusion process is applied to the image, corresponding to curving isophote curves to avoid them crossing each other.

3.1 PDE Based Inpainting in Image Compression

In the conclusion of [1], it is stated that the inpainting method presented naively resembles a third order PDE, however there would be problems representing it as such. The paper then goes on to suggest the use of lower, second order PDEs to address the inpainting problem.

In [10], Galić et al. write about how PDEs have mainly only been used as pre or postprocessing tools in image compression algorithms. They present a compression algorithm in which PDE-based inpainting is used in the decompression step itself, which we will also use. This inpainting is used to interpolate between scattered data points, which are saved from the original image, to reconstruct the original image. The general model for interpolation is as follows.

Let $u : \Omega \to \mathbb{R}$ be the original image to be recovered, with $\Omega \subset \mathbb{R}^2$ and let $\partial\Omega \subset \Omega$ be the set of points for which the value of the function u is known. In standard notation, $\partial\Omega$ refers to just the boundary of the domain, however we will use $\partial\Omega$ for the known interior points throughout. The goal is to find a function $v : \Omega \to \mathbb{R}$ which is close to u on $\Omega \backslash \partial\Omega$ and identical to u on $\partial\Omega$.

The problem is placed in an evolution setting with some evolution parameter t, representing time and the desired interpolation function is given as the steady state, i.e., when $t \to \infty$. The evolution is initialised as

$$v(x,0) = \begin{cases} u(x) & \text{if } x \in \partial\Omega \\ 0 & \text{otherwise.} \end{cases} \tag{3.1.1}$$

The choice of 0 here is arbitrary. The evolution is then defined as

$$\partial_t v = Lv, \tag{3.1.2}$$

where L is some elliptic differential operator. As our interpolation function v is given at the steady state, when $\partial_t v = 0$, we then solve the equation

$$Lu = 0. \tag{3.1.3}$$

The "boundary" conditions are Dirichlet boundary conditions given by the known values on $\partial\Omega$. This then gives us our reconstruction of the original image.

As for the choice of the elliptic operator L, different possibilities are discussed in [10]. We will briefly cover some here. The first operator presented is the Laplacian operator $\Delta = \nabla^2$ which leads to the equation

$$\partial_t u = \Delta u = \partial_{xx} u + \partial_{yy} u. \tag{3.1.4}$$

This corresponds to linear isotropic diffusion, where diffusion occurs in the same direction as the concentration gradient.

Nonlinear operators are used to encourage smoothing along edges while discouraging smoothing against them. This helps preserve edges that are found in the original image. The first nonlinear operator that Galić et al. suggest is

$$\partial_t u = \operatorname{div}(g(|\nabla u|^2)\nabla u), \tag{3.1.5}$$

where g is some function that is decreasing in its input. A sensible choice is Charbonnier diffusivity [7]:

$$g(s^2) = \frac{1}{\sqrt{1 + \frac{s^2}{\lambda^2}}}, \tag{3.1.6}$$

where $\lambda > 0$ is some diffusivity constant. Note that although this equation is non-linear it is still represents isotropic diffusion.

However, [10] shows that the best results in terms of accuracy of reconstruction are obtained by using anisotropic diffusion, where diffusion occurs against the concentration gradient.

Due to nonlinear PDEs being much harder to solve, we will instead use the Laplacian operator. Thus, to decompress our image, we will solve the equation

$$\partial_{xx} u + \partial_{yy} u = 0, \tag{3.1.7}$$

with Dirichlet boundary condition given by the known points on $\partial\Omega$.

4 Compression

The compression part of the method we will present involves selecting pixels of our image and only saving the values of these pixels. In the decompression part, we will reconstruct the image from just these pixel values. This prompts the question: which pixels should we select to save the values from? There is also the issue of being able to store these pixels in a space efficient

format. In this section, we will look at various techniques that have been used. Throughout, we will assume that our image is square with dimensions $n \times n$ where $n = 2^k + 1$ for some integer $k \geq 1$, as if this is not the case, our image can be padded by adding extra pixels around the image with value 0.

4.1 B-Tree Triangular Coding

The approach in [10] is to use B-tree triangular coding [9] (BTTC). BTTC works by recursively decomposing the input image into right angled triangles which are arranged in a binary tree. For a given right angled triangle, the recursion is stopped when linear interpolation between the triangle vertices gives a good enough reconstruction of the original image.

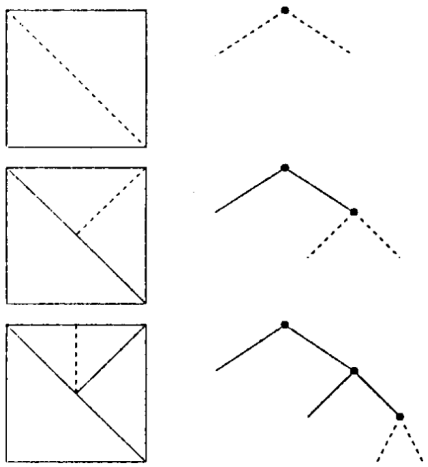

Figure 2: An example of BTTC decomposition taken from [9].

To see how the interpolation works, assume we have a gray image $u : \Omega \to \mathbb{R}$, where $\Omega \subset \mathbb{R}^2$. Let $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$ and $P_3 = (x_3, y_3)$ be vertices of a triangle with gray values c_1, c_2 and c_3 respectively. The interpolating function inside the triangle is defined as

$$v(x, y) = c_1 + \alpha(c_2 - c_1) + \beta(c_3 - c_1),$$

where α and β are defined by

$$\alpha = \frac{(x - x_1)(y_3 - y_1) - (y - y_1)(x_3 - x_1)}{(x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1)},$$

$$\beta = \frac{(x_2 - x_1)(y - y_1) - (y_2 - y_1)(x - x_1)}{(x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_3)}.$$

The interpolation function is tested by defining an error term $err(x, y)$:

$$err(x, y) = |u(x, y) - v(x, y)|.$$

If the error term for each point inside the triangle is greater than some parameter $\varepsilon > 0$, then the triangle is subdivided along its height relative to the hypotenuse. Alternatively, if we are looking for a certain compression rate, we can stop the subdivision when adding more points would be less than the desired compression.

When the subdivision has finished, the binary tree is stored as a binary string which is obtained from a breadth-first visit of the tree. In this string, the value 0 represents a leaf and 1 represents an intermediate vertex, corresponding to a triangle point. This string can be further simplified if the minimum level which contains a leaf is kept track of as we know that the preceding levels will only contain 1s, so they do not need to be stored. The colour values of the points of the triangle subdivision are also stored along with the tree. The lack of a need to store coordinates of these points is a huge advantage of BTTC when compared with other choices for selecting interpolation points, as this makes the storage of BTTC compressed images very efficient.

For colour images with three colour channels, we would simply perform BTTC on each colour channel and store each compressed channel individually.

4.2 Edge Detection

Chan and Shen's 2001 paper on inpainting [6] was mainly focused on using digital inpainting for applications such as restoration or object removal from images. However, in section 9, they discuss the use of inpainting for image compression. They suggest the use of edge detection for choosing points of the image to save.

Firstly, an edge detection algorithm is applied to the original image to find points that may be considered an edge, the result of this is an edge collection E of the image. A wide variety of edge detection algorithms are available, the authors suggest the use of the Canny edge detector [5]. Then, the neighborhood (the points within a small distance) of each edge point in

Figure 3: The result of applying the Canny edge detector to "peppers".

E is taken and this makes up T, which the authors is call an edge tube. This can be thought of a thickening of the edges in the image. Then finally, the edge tube is saved. Unfortunately, this saved format will likely be much less efficient than the BTTC approach as the position of each saved point will need to be stored as well as the colour value of that point.

4.3 Corner Detection

In [23], Zimmer suggests an approach using corner information to choose which points from the image $u : \Omega \to \mathbb{R}$ to save. This differs from the approach in [10] further, as rather than storing individual points, small regions called *corner regions* are stored instead. Corner regions are simply a neighborhood of a corner pixel, which is found by some corner detection technique. The neighborhood at points $(i, j) \in \Omega$ is defined by

$$B_r(i, j) = \{(i', j') \in \Omega : d((i, j), (i', j')) \leq r\},$$

where $d((i, j), (i', j')) = \sqrt{(i - i')^2 + (j - j')^2}$ is the Euclidean distance between the points (i, j) and (i', j').

The corner detection used by Zimmer involves using a local structure tensor J_ρ, a 2×2 matrix derived from the image gradient with

$$[J_\rho(\nabla u)]_{(i,j)} = \begin{pmatrix} K_\rho * \mathbf{B}_{xx}(i, j) & K_\rho * \mathbf{B}_{xy}(i, j) \\ K_\rho * \mathbf{B}_{xy}(i, j) & K_\rho * \mathbf{B}_{yy}(i, j) \end{pmatrix},$$

For a point $(i, j) \in \Omega$. Here, $\mathbf{B}_{xx}(i, j)$ is a matrix containing entries corresponding to $\partial_{xx}u(i', j')$ for $(i', j') \in B_r(i, j)$, the other $\mathbf{B}$s are defined similarly for the other derivatives. Also, K_ρ is a Gaussian kernel with standard deviation ρ and $*$ represents a convolution operation. Convolving K_ρ with $\mathbf{B}_{xx}(i, j)$ gives a bias to points closer to (i, j). The discrete image derivatives are calculated using a finite difference scheme, which we will later see in Section 5.1.

Let λ_1, λ_2 be the eigenvalues of $[J_\rho(\nabla u)]_{(i,j)}$ with corresponding eigenvectors $\mathbf{v}$ and $\mathbf{v}'$. Note that the eigenvalues are real as $[J_\rho(\nabla u)]_{(i,j)}$ is symmetric. The eigenvalues are used to determine if the point (i, j) is a corner. Without loss of generality, assume that $|\lambda_1| \geq |\lambda_2|$, there are the following cases:

- If $|\lambda_1| \approx |\lambda_2| \approx 0$, then the region surrounding (i, j) is a *flat region*,
- if $|\lambda_1| \gg |\lambda_2| \approx 0$, then we have a *straight edge* in the direction of $\mathbf{v}'$ at point (i, j),
- if $|\lambda_1| \geq |\lambda_2| \gg 0$, then we have a *corner* at point (i, j).

Figure 3 shows an example of the output from this corner detector.

For every corner point, the points in the neighborhood of the corner are to be saved, this gives us a sparse image where the value of the image is 0 for every point that is not in the neighborhood of a corner point. Zimmer notes that just storing the sparse image in a typical way would yield no compression, however the sparsity can be exploited. This was done by inventing a new file format called CRF (Corner Region Format), details of which can be found in seciton 4.2.3 of [23].

5 Decompression

Because BTTC is very efficient in terms of storage space needed, we would prefer to only use points from BTTC if possible. However, this may create parts of the image where we have no saved points, which could affect both reconstruction accuracy and decompression time. To solve this we can add some extra points from these parts, possibly choosing edge points or corner points. Storing these extra points will be less storage space efficient than in BTTC as we have to store the coordinates of these points, but it may be worth it to get better reconstruction accuracy.

Figure 4: The output of the corner detector used on "lena", with parameters $\rho = 1$, $r = 4$ and a corner threshold of 2, taken from [23].

Now we have our original image saved as only a set of scattered points. In this section, we will now look at how we would reconstruct the original image from the saved values using PDE based impainting. First we need to discretise our PDE as our image is defined at discrete points, rather than being continuous.

5.1 PDE Discretisation

To solve the PDE, we write it in terms of pixel values through discretisation. This involves taking the spacial derivatives in the equation and representing them as differences in pixel values. For first order derivatives, we have three options:

$$\partial_x u_{i,j} = \frac{u_{i,j} - u_{i-1,j}}{2},$$

$$\partial_x u_{i,j} = \frac{u_{i+1,j} - u_{i,j}}{2},$$

$$\partial_x u_{i,j} = \frac{u_{i+1,j} - u_{i-1,j}}{2}.$$

These are called the backward difference, forward difference and centered difference with respect to x.

For Laplace's equation we will be using the finite difference for the second derivative:

$$\partial_{xx} u_{i,j} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{2};$$

$$\partial_{yy} u_{i,j} = \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{2}.$$

Substituting this into Laplace's equation (3.1.7) then gives us

$$u_{i,j+1} + u_{i+1,j} - 4u_{i,j} + u_{i-1,j} + u_{i,j-1} = 0, \quad 1 \leq i, j \leq n \tag{5.1.1}$$

for the unknown pixel values, although we will instead multiply this equation by -1 to make things easier. Then we have

$$-u_{i,j+1} - u_{i+1,j} + 4u_{i,j} - u_{i-1,j} - u_{i,j-1} = 0, \quad 1 \leq i, j \leq n. \tag{5.1.2}$$

However, this presents a problem. Notice that when either i or j is equal to 1 or n, equation (5.1.2) refers to points that are outside of our image. To solve this problem, we will assume that the values of points just outside of our image domain will have the value of the point that is adjacent to them in the domain. For example, let $i = 1$, then we assume that $u_{1,0} = u_{1,1}$. This then gives us

$$-u_{1,j+1} - u_{2,j} + 3u_{1,j} - u_{1,j-1} = 0, \quad 2 \leq j \leq n - 1, \tag{5.1.3}$$

furthermore, if both i and j are equal to 1, i.e., the point at the top left of the image, we have

$$-u_{1,2} - u_{2,1} + 2u_{1,1} = 0. \tag{5.1.4}$$

Along with the known points, this gives us a system of linear equations which we will solve to find the values for our uncompressed image.

To put this into a matrix-vector form $\mathbf{Au} = \mathbf{f}$, we will need to order the points to put them into the one dimensional vector. The top left point of the image $u_{1,1}$ will be the first component of the vector, and we will continue along the row before returning to the start of the next row. For the matrix $\mathbf{A}$, the rows corresponding to unsaved points will contain the coefficients of (5.1.2) and the right hand side value will be zero. To help visualise this,

consider the case where we have a 3×3 image and suppose that we know that $u_{1,1} = 50$, $u_{2,2} = 100$ and $u_{2,3} = 20$. The corresponding linear system will be

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 3 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 2 & 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & -1 & 3 & -1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 2 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 3 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} u_{1,1} \\ u_{1,2} \\ u_{1,3} \\ u_{2,1} \\ u_{2,2} \\ u_{2,3} \\ u_{3,1} \\ u_{3,2} \\ u_{3,3} \end{pmatrix} = \begin{pmatrix} 50 \\ 0 \\ 0 \\ 0 \\ 100 \\ 20 \\ 0 \\ 0 \\ 0 \end{pmatrix}. \quad (5.1.5)$$

We then reduce this linear system by splitting the image vector $\mathbf{u}$ into two vectors, one that contains the unknown points and the other containing the known points. Let $\mathbf{u}'$ and $\hat{\mathbf{u}}$ be these vectors respectively, we have $\mathbf{u} = \mathbf{u}' + \hat{\mathbf{u}}$. Substituting this into $\mathbf{Au} = \mathbf{f}$ gives

$$\begin{aligned} \mathbf{A}(\mathbf{u}' + \hat{\mathbf{u}}) &= \mathbf{f} \\ \mathbf{Au}' + \mathbf{A}\hat{\mathbf{u}} &= \mathbf{f} \\ \mathbf{Au}' &= \mathbf{f} - \mathbf{A}\hat{\mathbf{u}}. \end{aligned} \quad (5.1.6)$$

Note that there are no unknowns on the right hand side so we can compute it. This eliminates the known points from the system. To see this more clearly, let's look at our example from before. Substituting into 5.1.6 gives

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 3 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 2 & 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & -1 & 3 & -1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 2 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 3 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} 0 \\ u_{1,2} \\ u_{1,3} \\ u_{2,1} \\ 0 \\ 0 \\ u_{3,1} \\ u_{3,2} \\ u_{3,3} \end{pmatrix} = \begin{pmatrix} 0 \\ 150 \\ 20 \\ 150 \\ 0 \\ 0 \\ 0 \\ 100 \\ 20 \end{pmatrix}.$$

Removing the rows where we have eliminated $u_{i,j}$ gives us the reduced system

$$\begin{pmatrix} 3 & -1 & 0 & 0 & 0 & 0 \\ -1 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & -1 & 0 & 0 \\ 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & -1 & 3 & -1 \\ 0 & 0 & 0 & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} u_{1,2} \\ u_{1,3} \\ u_{2,1} \\ u_{3,1} \\ u_{3,2} \\ u_{3,3} \end{pmatrix} = \begin{pmatrix} 150 \\ 20 \\ 150 \\ 0 \\ 100 \\ 20 \end{pmatrix},$$

which is only a system of the unknown values to be found. From now on, we will only work with the reduced system and refer to it as

$$\mathbf{Au} = \mathbf{f}. \tag{5.1.7}$$

Note that the matrix $\mathbf{A}$ is both symmetric and diagonally dominant. As the diagonal entries are all positive, it is clear that $\mathbf{A}$ is positive semidefinite. Furthermore, [17] shows that $\mathbf{A}$ is invertible, hence $\mathbf{A}$ is positive definite. The matrix $\mathbf{A}$ is also sparse, with the overwhelming majority of entries being 0. Note that the larger the image we work with, the more sparse $\mathbf{A}$ will become, i.e., the proportion of 0s in the matrix with be greater.

5.2 Basic Iterative Methods

To solve our linear system we will use an iterative method. We favour iterative methods over direct methods here as in theory, they should be faster since our linear system is so large. It has been shown that multigrid methods are a good choice to use for our problem[17]. This will require a relaxation step which takes a current approximate solution and refines it to produce a solution closer to the actual solution. We will look at two relaxation methods, starting with the Jacobi method.

5.2.1 Jacobi Relaxation Method

The Jacobi method, shown in [4], updates the solution on a component by component basis. Rearranging (5.1.2) gives us

$$u_{i,j} = \frac{1}{4}(u_{i,j+1} + u_{i+1,j} + u_{i-1,j} + u_{i,j-1}). \tag{5.2.1}$$

We can represent this in matrix form. First we decompose our matrix A into three parts:

$$\mathbf{A} = \mathbf{D} - \mathbf{U} - \mathbf{L}, \tag{5.2.2}$$

where $\mathbf{D}$ is the diagonal part of $\mathbf{A}$ and $-\mathbf{U}$, $-\mathbf{L}$ are the strictly upper diagonal and lower diagonal parts respectively. Substituting this into (5.1.7) gives

$$(\mathbf{D} - \mathbf{U} - \mathbf{L})\mathbf{u} = \mathbf{f};$$

$$\mathbf{D}\mathbf{u} = (\mathbf{U} + \mathbf{L})\mathbf{u} + \mathbf{f};$$

$$\mathbf{u} = \mathbf{D}^{-1}(\mathbf{U} + \mathbf{L})\mathbf{u} + \mathbf{D}^{-1}\mathbf{f}. \tag{5.2.3}$$

To make sense of (5.2.3), consider just one unsaved point of $\mathbf{u}$. The matrix $\mathbf{D}^{-1}$ corresponds to dividing by 4, $(\mathbf{U} + \mathbf{L})\mathbf{u}$ corresponds to surrounding unknown point values and $\mathbf{f}$ corresponds to the surrounding saved point values. Let $\mathbf{v}^{(k)}$ be a current solution. We then get an update rule for $\mathbf{v}^{(k)}$:

$$\mathbf{v}^{(k+1)} = \mathbf{D}^{-1}(\mathbf{U} + \mathbf{L})\mathbf{v}^{(k)} + \mathbf{D}^{-1}\mathbf{f}. \tag{5.2.4}$$

The Jacobi method converges when the spectral radius of the matrix $\mathbf{D}^{-1}(\mathbf{U}+\mathbf{L})$ is less than 1.

5.2.2 Gauss-Seidel Relaxation Method

The Gauss-Seidel method is similar to the Jacobi method, except it uses the components of the new solution as soon as they are computed. Using the same matrix decomposition (5.2.2), the update rule for the Gauss-Seidel method is

$$\mathbf{v}^{(k+1)} = (\mathbf{D} - \mathbf{L})^{-1}\mathbf{U}\mathbf{v}^{(k)} + (\mathbf{D} - \mathbf{L})^{-1}\mathbf{f}. \tag{5.2.5}$$

As component values are used as soon as they are computed, the order of computation matters, unlike with the Jacobi method. If we iterate through the components in the opposite order also, we get the symmetric Gauss-Seidel method. This can also easily be expressed in matrix-vector form with the update rule

$$\mathbf{v}^{(k+1)} = ((\mathbf{D} - \mathbf{L})^{T})^{-1}\mathbf{U}^{T}\mathbf{v}^{(k)} + (\mathbf{D} - \mathbf{L})^{-1}\mathbf{f}. \tag{5.2.6}$$

It is known that the Gauss-Seidel method converges for symmetric positive definite matrices as shown in [12], Theorem 10.1.2. Furthermore, [13] shows that when the Jacobi method converges for some matrix $\mathbf{A}$, then the Gauss-Seidel method will also converge and generally converge faster. Thus, we will use the Gauss-Seidel method over the Jacobi method in our application.

5.3 Multigrid Methods

To perform our decompression, we could simply use some initial guess for $\mathbf{v}$, then repeatedly apply the one of the two above basic iterative methods until we have an acceptably accurate solution. However, in practice the convergence rate of the Gauss-Seidel and Jacobi methods steeply diminishes after the first few iterations. This is due to the methods ineffectiveness at removing high frequency signal errors in the initial guess. A full analysis of effectiveness of Jacobi and Gauss-Seidel methods' ability to remove different frequency signal errors can be found in [4].

To help overcome the slow convergence rates of the two basic iterative methods, we will look at multigrid methods, also shown in [4]. These are iterative methods that use solutions to a coarser grid problem to update the solution on the full grid. For now we will look at the two grid method before looking at methods using more than two grids.

Let's say we have $\mathbf{v}$, a current solution of an iterative method on a system of linear equations. We can write this $\mathbf{v}$ as the actual solution plus some error term $\mathbf{e}$, we have

$$\mathbf{v} = \mathbf{u} + \mathbf{e}. \tag{5.3.1}$$

Multiplying by A then gives

$$\mathbf{Av} = \mathbf{A}(\mathbf{u} + \mathbf{e}) = \mathbf{Au} + \mathbf{Ae} = \mathbf{f} + \mathbf{Ae}. \tag{5.3.2}$$

Then, rearranging gives

$$\mathbf{Ae} = \mathbf{Av} - \mathbf{f}. \tag{5.3.3}$$

We will call $\mathbf{Av} - \mathbf{f}$ the residuum and denote it $\mathbf{r}$. If we can approximate the error, we can use this to improve our solution.

5.3.1 The Two Grid Scheme

In the two grid scheme, we use a coarse grid to approximate the error. Let Ω^f be our original grid, containing all points corresponding to pixels of the image and call this our fine grid. We will define the coarse grid as all the even points of Ω^f and call it Ω^c. We will write the fine grid solution as $\mathbf{u}^f$ and the coarse grid solution as $\mathbf{u}^c$.

To take our problem on the fine grid and transfer it to the coarse grid we will use a restriction operator. This will operate on the fine grid points and calculate what the coarse grid point should be. The simplest restriction

operator is called injection, this is when we just take the corresponding fine grid point value and use it for the coarse point. We will instead use a restriction operator that takes a weighted sum of the fine points that surround the coarse point. This operator is defined as

$$\begin{aligned} u^c_{i,j} = {} & u^f_{2i-1,2j-1} + u^f_{2i+1,2j-1} + u^f_{2i-1,2j+1} + u^f_{2i+1,2j+1} \\ & + 2(u^f_{2i,2j-1} + u^f_{2i,2j+1} + u^f_{2i-1,2j} + u^f_{2i+1,2j}) + 4u^f_{2i,2j}, \\ & \text{for } 1 \leq i,j \leq \lfloor n/2 \rfloor. \end{aligned} \tag{5.3.4}$$

Note that when n is even, (5.3.4) refers to points outside of our image domain. To fix this problem, we will just assume that these values are 0. We will use this restriction operator as a matrix and call it $\mathbf{R}$. To transfer a solution from the coarse grid back to the we will use a prolongation operator $\mathbf{P}$, taking $\mathbf{P} = \mathbf{R}^T$.

For the two grid scheme, we will also need reduce our fine matrix $\mathbf{A}^f$ to the coarse setting. This involves reducing the number of columns and rows of $\mathbf{A}^f$ to the length of our coarse vectors in the same way that they were obtained with the restriction operator. Thus, we calculate $\mathbf{A}^c$ by

$$\mathbf{A}^c = \mathbf{R}\mathbf{A}^f\mathbf{R}^T = \mathbf{R}\mathbf{A}^f\mathbf{P}. \tag{5.3.5}$$

Now we are in the position to give the two grid scheme.

The Two Grid Scheme:

1. initialise $\mathbf{v}^f$ with some initial guess.
2. Perform c_1 iterations of symmetric Gauss-Seidel on $\mathbf{A}^f\mathbf{v}^f = \mathbf{f}^f$.
3. Compute the fine residuum $\mathbf{r}^f$ and restrict it to the coarse residuum using $\mathbf{r}^c = \mathbf{R}\mathbf{r}^f$.
4. Solve $\mathbf{A}^c\mathbf{e}^c = \mathbf{r}^c$ exactly using some direct method.
5. Prolong $\mathbf{e}^c$ to the estimated fine error with $\mathbf{e}^f = \mathbf{P}\mathbf{e}^c$.
6. Update the fine solution with $\mathbf{v}^f \leftarrow \mathbf{v}^f + \mathbf{e}^f$.
7. Perform c_2 iterations of symmetric Gauss-Seidel on $\mathbf{A}^f\mathbf{v}^f = \mathbf{f}^f$.
8. If desired convergence is not reached, go back to step 3.

The integers c_1 and c_2 are parameters controlling the number of relaxation steps we take before and after visiting the coarse grid. They are usually 1, 2 or 3 and are often chosen based on experimental results.

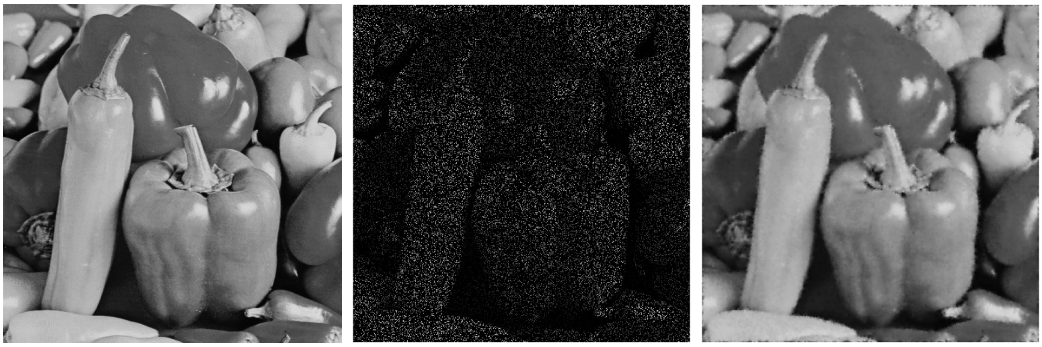

Figure 5: Left: The original image"peppers". Centre: 15% of pixels saved at random. Right: The reconstructed image generated from the saved pixels by the two grid scheme.

5.3.2 The V-Cycle Scheme

Note that step 4 of the two grid scheme we are solving the linear system $\mathbf{A}^c\mathbf{e}^c = \mathbf{r}^c$. This is exactly the same problem we are solving in fewer variables, so we can also apply the two grid scheme to this step. This idea of repeatedly applying the two grid scheme leads us to the V-cycle multigrid scheme, where eventually an exact solution of the error term will be calculated at the coarsest grid. We will define the V-cycle scheme recursively. Note that in our recursive definition, the operators $\mathbf{R}$ and $\mathbf{P}$ refer to the restriction and prolongation operators for the fine grid that the scheme is currently running on.

The V-Cycle Scheme on $\mathbf{A}^f\mathbf{v}^f = \mathbf{f}^f$:

1. If we are at the coarsest grid, solve $\mathbf{A}^f\mathbf{v}^f = \mathbf{f}^f$ using some direct method and go to step 6.

2. Initialise $\mathbf{v}^f$ with an initial guess of 0 and perform c_1 iterations of symmetric Gauss-Seidel on $\mathbf{A}^f\mathbf{v}^f = \mathbf{f}^f$.

3. Compute the fine residuum $\mathbf{r}^f$ and restrict it to the coarse residuum using $\mathbf{r}^c = \mathbf{R}\mathbf{r}^f$, then perform the V-cycle scheme on $\mathbf{A}^c\mathbf{e}^c = \mathbf{r}^c$.
4. Update the fine solution with $\mathbf{v}^f \leftarrow \mathbf{v}^f + \mathbf{P}\mathbf{e}^c$.
5. Perform c_2 iterations of symmetric Gauss-Seidel on $\mathbf{A}^f\mathbf{v}^f = \mathbf{f}^f$.
6. Return the value of $\mathbf{v}^f$.

5.3.3 The Full Multigrid Scheme

In step 2 of the V-cycle scheme, we initialise $\mathbf{v}^f$ with an initial guess. In theory, we would achieve faster convergence if this guess better. In the full multigrid method, we calculate an initial guess by solving the problem on a coarser grid, using the V-cycle scheme.

The Full Multigrid Scheme on $\mathbf{A}^f\mathbf{v}^f = \mathbf{f}^f$:

1. If we are at the coarsest grid, set $\mathbf{v}^f$ to 0 and go to step 4.
2. Restrict $\mathbf{f}^f$ to the coarse grid with $\mathbf{f}^c = \mathbf{R}\mathbf{f}^f$ and perform the full multigrid scheme on $\mathbf{A}^c\mathbf{v}^c = \mathbf{f}^f$.
3. Set $\mathbf{v}^f$ to $\mathbf{P}\mathbf{v}^c$.
4. Perform c_0 iterations of the V-cycle scheme with initial guess $\mathbf{v}^f$.

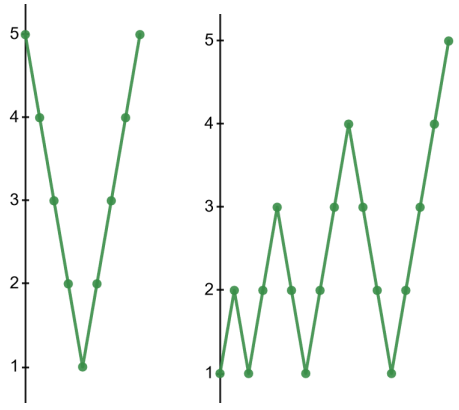

Figure 6: A diagram of the the grids visited by V-cycle and full multigrid schemes, level 5 represents the fine grid and level 1, the coarsest grid. Left: V-Cycle, Right: Full Multigrid with $c_0 = 1$.

Note that the full multigrid scheme is designed to be only run once. If we require better convergence, we use the scheme with a higher value of c_0.

5.4 Conjugate Gradient Method

One option for solving our linear system is the conjugate gradient method, which we will briefly explain. The conjugate gradient method [20] is a popular iterative method for solving large systems of linear equations $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is symmetric and positive definite. The method consists of minimising the function

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{Ax} - \mathbf{b}^T\mathbf{x},$$

which is called a quadratic form. It can be easily shown that if A is symmetric and positive definite, then $f(\mathbf{x})$ is minimised by the solution of $\mathbf{Ax} = \mathbf{b}$. Computing the gradient of f, defined similarly to (1.1.1) except with much more variables, we get

$$\nabla f(\mathbf{x}) = \frac{1}{2}\mathbf{A}^T\mathbf{x} + \frac{1}{2}\mathbf{Ax} - \mathbf{b}.$$

As $\mathbf{A}$ is symmetric, i.e., $\mathbf{A} = \mathbf{A}^T$, then ∇f is reduced to

$$\nabla f(\mathbf{x}) = \mathbf{Ax} - \mathbf{b}.$$

5.4.1 Method of Steepest Descent

Before we get into the conjugate gradient method for minimising f, we will first look at the method of steepest descent, as it is similar but much simpler. This is an iterative method where an current approximation $\mathbf{x}^{(k)}$ is moved in the direction that decreases the value of $f(\mathbf{x}^{(k)})$ the most quickly, i.e., the direction of $-\nabla f(\mathbf{x}^{(k)}) = \mathbf{b} - \mathbf{Ax}^{(k)}$. We let $\mathbf{r}^{(k)} = \mathbf{b} - \mathbf{Ax}^{(k)}$, and call $\mathbf{r}^{(k)}$ the residuum. Then our update rule is

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)}\mathbf{r}^{(k)}, \tag{5.4.1}$$

where $\alpha^{(k)}$ is a scalar which represents how far we move the point $\mathbf{x}^{(k)}$ in the direction of $\mathbf{r}^{(k)}$.

The value of $\alpha^{(k)}$ is chosen to minimise f along the line through $\mathbf{x}^{(k)}$ in the direction of $\mathbf{r}^{(k)}$. We won't go into details here, but we would find that

α should be chosen such that $\mathbf{r}^{(k)}$ and $\nabla f(\mathbf{x}^{(k+1)})$ are orthogonal, with

$$\alpha^{(k)} = \frac{(\mathbf{r}^{(k)})^T \mathbf{r}^{(k)}}{(\mathbf{r}^{(k)})^T \mathbf{A}\mathbf{r}^{(k)}}.$$

Note that as we have stated the iteration, it requires two matrix-vector multiplications, which dominate the computational cost of the steepest descent method. We can reduce this to just one matrix-vector multiplication. By multiplying (5.4.1) by $\mathbf{A}$ and adding $\mathbf{b}$, we get

$$\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha^{(k)} \mathbf{A}\mathbf{r}^{(k)}.$$

Instead of calculating $\mathbf{A}\mathbf{x}^{(k)}$, we now need to calculate $\mathbf{A}\mathbf{r}^{(k)}$, but this is already part of calculating $\alpha^{(k)}$ so we use the result for both update rules.

5.4.2 Method of Conjugate Directions

When performing an iteration of steepest descent, we often move the value of $\mathbf{x}^{(k)}$ in the same direction that it was moved in a previous iteration. Let l be the width (and also height) of $\mathbf{A}$. The conjugate direction method stops this by using a set of orthogonal vectors $\{\mathbf{d}^0, \mathbf{d}^1, \ldots, \mathbf{d}^{l-1}\}$ called search directions and performing an iteration moving in the direction of each one only once. Unfortunately, knowing what value of $\alpha^{(k)}$ to use is equivalent to knowing the solution of $\mathbf{A}\mathbf{x} = \mathbf{b}$. Instead, we use $\mathbf{A}$-orthogonal search vectors. We say that two vectors $\mathbf{d}$ and $\mathbf{d}'$ are $\mathbf{A}$-orthogonal if

$$\mathbf{d}^T \mathbf{A}\mathbf{d}' = 0.$$

Now, assume our search directions $\{\mathbf{d}^0, \mathbf{d}^1, \ldots, \mathbf{d}^{l-1}\}$ are $\mathbf{A}$-orthogonal. Similarly as with steepest descent, we find that $\mathbf{d}^k$ should be orthogonal to $\nabla f(\mathbf{x}^{(k+1)})$. Again, omitting the details, we get

$$\alpha^{(k)} = \frac{(\mathbf{d}^k)^T \mathbf{r}^{(k)}}{(\mathbf{r}^{(k)})^T \mathbf{A}\mathbf{d}^k}.$$

Now, it just remains to find a set of vectors $\{\mathbf{d}^0, \mathbf{d}^1, \ldots, \mathbf{d}^{l-1}\}$ that are $\mathbf{A}$-orthogonal. This can be done by the conjugate Gram-Schmidt process. This process takes a set of l linearly independent vectors $\{\mathbf{v}^1, \mathbf{v}^2, \ldots, \mathbf{v}^{l-1}\}$, for example the standard basis. For each vector $\mathbf{v}^k$, we produce $\mathbf{d}^k$ by subtracting

out any components that are not **A**-orthogonal to the previous vectors $\mathbf{d}^m$, $k < m$. Thus, we set $\mathbf{d}^0$ as $\mathbf{v}^0$ and use

$$\mathbf{d}^k = \mathbf{u}^k + \sum_{m=0}^{k-1} \beta_{km}\mathbf{d}^m, \quad \text{for } k > 0, \tag{5.4.2}$$

where β_{km} are defined for $k > m$. The derivation of the value of β_{km} can be found in [20], we get

$$\beta_{km} = \frac{(\mathbf{v}^k)^T\mathbf{A}\mathbf{d}^m}{(\mathbf{d}^m)^T\mathbf{A}\mathbf{d}^m}.$$

5.4.3 Method of Conjugate Gradients

For the conjugate gradient method, we simply use the the residuums for the conjugate Gram-Schmidt process. As each residuum is orthogonal to the previous residuums, it is **A**-orthogonal to the previous search directions. This is good, as we will only have to calculate β for the current step, also we do not need to remember previous search directions. Then, the conjugate gradient method becomes:

$$\mathbf{d}^0 = \mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)};$$

$$\alpha^{(k)} = \frac{(\mathbf{r}^{(k)})^T\mathbf{r}^{(k)}}{(\mathbf{d}^{(k)})^T\mathbf{A}\mathbf{d}^{(k)}};$$

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)}\mathbf{d}^k;$$

$$\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha^{(k)}\mathbf{A}\mathbf{d}^k;$$

$$\beta^{(k+1)} = \frac{(\mathbf{r}^{(k+1)})^T\mathbf{r}^{(k+1)}}{(\mathbf{r}^{(k)})^T\mathbf{r}^{(k)}};$$

$$\mathbf{d}^{k+1} = \mathbf{r}^{(k+1)} + \beta^{(k)}\mathbf{d}^k.$$

6 Implementation Details

The implementation of the compression method was done in MATLAB version R2022b [21], which was chosen because it handles matrix arithmetic well and also has useful in-built functions. Here, we will look at how these features can be used to aid implementation.

Firstly, MATLAB has efficient implementations of various matrix manipulation functions such as triu and tril, which return matrices containing only the upper triangular and lower triangular part of the input respectively. This is useful for matrix decomposition (5.2.2), which we use for the relaxation methods along with MATLAB's efficient matrix arithmetic.

A very important feature of MATLAB is sparse matrices. Throughout the decompression step, we use matrices that are very large. For example, if our image has dimensions 400×400, then the original discretisation matrix $\mathbf{A}$ would have size $400^2 \times 400^2 = 160000 \times 160000$. Storing this matrix as full in a 2 dimensional array would take up a huge amount of memory, much more than what the average computer has. Storing $\mathbf{A}$ as a sparse matrix exploits the fact that the overwhelming majority of entries in $\mathbf{A}$ are 0, and thus we can choose just to store the non-zero entries. Sparse matrices in MATLAB store matrices column by column, where each column is stored in a sparse vector format. This sparse vector format consists of two parts, the first is an array of the non-zero components of the vector and the second is an array containing the indices of the values in the first array. A full description of the design and implementation of sparse matrices can be found in [11]. In our implementation of the compression method sparse matrices are used for the storage of all restriction, prolongation and discretisation matrices.

MATLAB contains a suite of test matrices. One of these matrices is the matrix for Laplace's (and also Poisson's) equation, however it does not use the same equations for points at the edges (5.1.3) and corners (5.1.4) that we are using. Therefore, we will generate the discretisation matrix ourselves.

The implementation of BTTC used[2] was written by Edgar Simo-Serra of Waseda University, Tokyo.

7 Experiments

In this section, we will conduct some experiments on our PDE based method. Experiments were run on an intel i5-12500H CPU, clocked at 4.5 GHz.

[2]https://github.com/bobbens/libbttc

7.1 Effect of Compression Rate on Reconstruction Accuracy

To be able to compare image reconstruction accuracy, we will use the mean squared error (MSE). Let u be the original, uncompressed image with dimension $n \times n$ and v be our reconstructed image, then we let

$$\text{MSE of } v := \frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} (u_{i,j} - v_{i,j})^2.$$

The lower the MSE of a reconstruction, the more accurate it is.

We will use the images "peppers" and "jetplane" and compress them using BTTC at compression rates of 1.6 bpp, 0.8 bpp, 0.4 bpp and 0.2 bpp, corresponding to ratios of 1:5, 1:10, 1:20 and 1:40. For reconstruction, we will use a direct method. This will be slow, but here we are interested in reconstruction accuracy and will test convergence rates of iterative methods later. Figure 8 shows the image reconstructions and Table 1 shows the mean squared errors.

As we expect, the higher the compression rate, the higher the MSE. Looking at the reconstruction of "peppers", we can see that as we increase the compression rate, we lose detail in the image. The edges of each pepper gets blurred and the surfaces of the peppers lose texture and becomes smooth. In the reconstructions of "jetplane", we can see that the text on the plane becomes less and less clear when we increase the compression rate and almost all of the detail in the clouds behind the plane is lost. When using our compression method, the choice of compression rate will be chosen based on the situation, as there will be a trade-off between reconstruction accuracy and storage space saving.

Compression rate	Peppers MSE	Jetplane MSE
1.6 bpp	0.00052	0.0014
0.8 bpp	0.0026	0.0045
0.4 bpp	0.0056	0.0075
0.2 bpp	0.0110	0.0115

Table 1: The MSE of reconstructions of "peppers" and "jetplane" at varying compression rates.

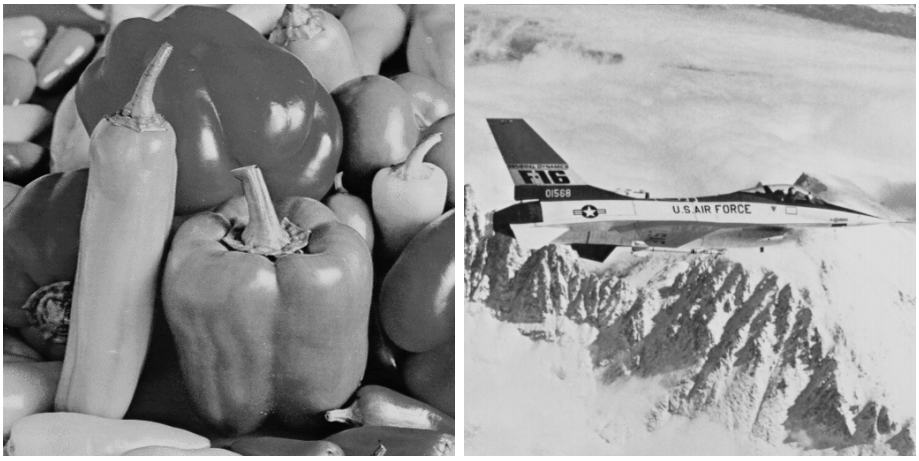

Figure 7: The uncompressed images "peppers", and "jetplane" that we will perform tests on.

7.2 Convergence Rates of Decompression methods

Now, we will test the convergence rates of different decompression methods. We will use the "peppers" image at a compression rate of 0.2 bpp. The MSE will be calculated between the image reconstructed by the decompression method and the exact solution which will be calculated using a direct method. Time values were retrived using MATLAB's in-built timing function and calculation of MSE was not included in the timing.

First, let us compare the convergence rates of just using the Gauss-Siedel basic iterative method against the V-cycle scheme, a more advanced method. Figure 9 shows the results. We can see that the Gauss-Seidel method performs very poorly compared to the V-Cycle scheme. We conclude that the Gauss-Seidel should not be used on its own due to it's poor convergence rate.

Now, let's compare the conjugate gradient and full multigrid methods. Figure 10 shows the results of these tests. Conjugate gradient initially seems to be converging slowlyin comparison, but it overtakes the full multigrid scheme at around 0.3s after the convergence rate of the full multigrid scheme falls off.

We might conclude from this that we should use the conjugate gradient method, however we should consider at what point we should stop the iterative method, i.e., when is the reconstruction "good enough". Let's look at the reconstructions created by the conjugate gradient method and full

Figure 8: The reconstructions of “peppers” and “jetplane” at compression rates of 1.6 bpp, 0.8 bpp, 0.4 bpp and 0.2 bbp from top to bottom.

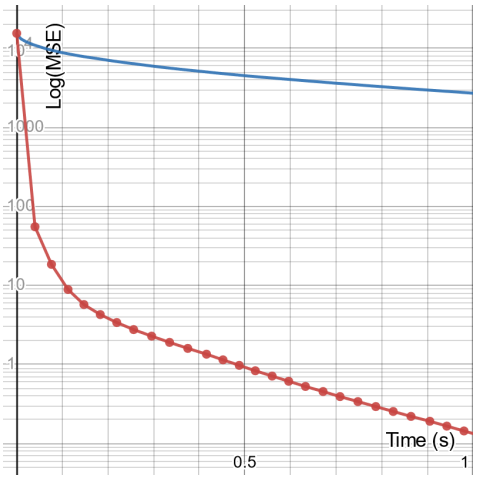

Figure 9: The convergence of the Gauss-Seidel method and V-Cycle scheme in logarithmic MSE scale when decompressing "peppers" from 0.2 bpp. Blue: Gauss-Seidel, Red: V-Cycle.

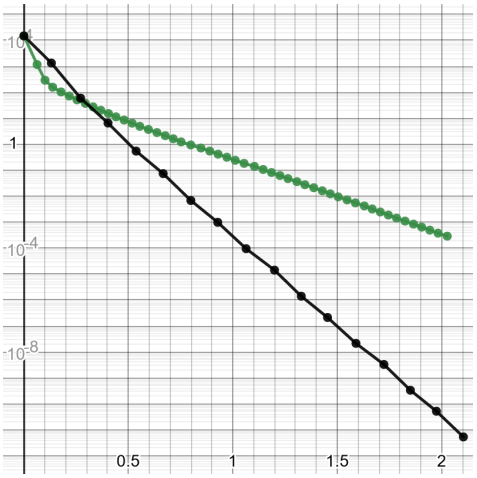

Figure 10: The convergence of the conjugate gradient method and full multigrid scheme in logarithmic MSE scale when decompressing "peppers" from 0.2 bpp. Black: Conjugate Gradient, Green: Full Multigrid.

multigrid scheme when we only let them run for 0.3s. Figure 11 shows these reconstructions.

Looking at the reconstructions after 0.3s, we can see that it is hard to tell a difference between the reconstructions and the only apparent differences are some parts of the image being slightly darker or lighter. This suggests that in cases where fast decompression is paramount, the full multigrid scheme may be preferable over the conjugate gradient method for this particular image.

Now, let's see if this still holds for a larger image. We will work with the image "mountains", which has a height and width of 2049, and compress it using BTTC at a compression rate of 0.4 bpp. Figure 12 shows the image we will use and Figure 13 shows the convergence of both the conjugate gradient and full multigrid method.

Looking at Figure 13, we can see that in this case of a larger image, the full multigrid scheme is ahead of the conjugate gradient method at all points throughout the 10 second test. This shows that for large images, multigrid methods can outperform conjugate gradient and they may be the better choice.

7.3 Effect of Compression Rate on Decompression Time

Let us now look at the effect of compression rate on decompression time. We will consider the case where we are most interested in the reconstruction accuracy and so we will look for convergence. This test was done using the image "peppers", thus, we will use the conjugate gradient method as we have seen that the convergence rate of multigrid methods fall off compared to CG for this particular image. We will run CG until the difference between each point of the exact solution and the solution generated by CG is less that 0.5 as the values would be rounded to the nearest integer. We will again use rates of 1.6 bpp, 0.8 bpp, 0.4 bpp and 0.2 bpp compressed using BTTC. Table 2 shows the results.

We can see that the compression rate heavily influences the convergence rate of the conjugate gradient method. The more compressed our image is, the longer it takes to attain convergence. This tells us that not only is there a trade-off between storage space taken and image quality, but also decompression time.

Figure 11: The exact solution and results of the conjugate gradient method and full multigrid after 0.5s. Left: Exact Solution, Centre: Conjugate Gradient, Right: Full Multigrid.

Figure 12: The image "mountains", with dimension 2049×2049.

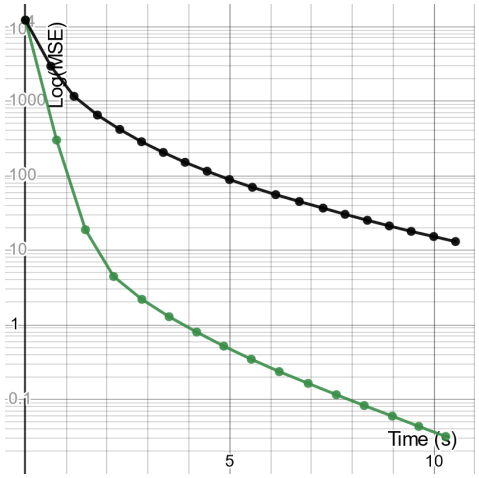

Figure 13: The convergence of the conjugate gradient method and full multigrid scheme when decompressing "mountains" from 0.4 bpp. Black: Conjugate Gradient, Green: Full Multigrid.

Compression rate	Decompression Time (s)	Number of Iterations
1.6 bpp	0.2545	87
0.8 bpp	0.4132	168
0.4 bpp	0.5892	200
0.2 bpp	0.6923	244

Table 2: The time taken for conjugate gradient to achieve less than 0.5 difference per pixel when decompressing "peppers" and the number of iterations taken.

8 Conclusion

We have presented a method of image compression using PDE based inpainting and explained how it works, looking at different options for both pixel selection for compression and different methods for decompression. We have also performed some experiments to see how the different options for decompression perform on images compressed using BTTC.

In the decompression step, we only used Laplace's equation, a linear PDE. However, in [10], Galić et al. show that non-linear PDEs can give better results for reconstruction accuracy. The decompression method that was used was a semi-implicit time discretisation, our experiments suggest that both multigrid methods and the conjugate gradient method could be useful. Of course, this would instead require the use of non-linear multigrid methods [14] and a non-linear version of the conjugate gradient method [8].

Future work could also include testing the use of edge and corner points and if the use of these points is worth the extra storage space needed to save such points due to the need to store points' location as well as the points' colour values.

References

[1] Bertalmío, M., Sapiro, G., Caselles, V. and Ballester, C. [2000], Image inpainting, *in* 'Proceedings of the 27th annual conference on Computer graphics and interactive techniques', pp. 417–424.

[2] Bhatia, R. [2007], *Positive Definite Matrices*, Princeton University Press.

[3] Brezis, H. and Browder, F. [1998], 'Partial differential equations in the 20th century', *Advances in Mathematics* **135**(1), 76–144.

[4] Briggs, W., Henson, V. and McCormick, S. [2000], *A Multigrid Tutorial, 2nd Edition*, SIAM.

[5] Canny, J. [1986], 'A computational approach to edge detection', *Pattern Analysis and Machine Intelligence, IEEE Transactions on* **PAMI-8**, 679 – 698.

[6] Chan, T. F. and Shen, J. [2001], 'Mathematical models for local nontexture inpaintings', *SIAM Journal on Applied Mathematics* **62**(3), 1019–1043.

[7] Charbonnier, P., Blanc-Feraud, L., Aubert, G. and Barlaud, M. [1997], 'Deterministic edge-preserving regularization in computed imaging', *IEEE Transactions on Image Processing* **6**(2), 298–311.

[8] Dai, Y.-H. and Yuan, Y.-x. [1999], 'A nonlinear conjugate gradient method with a strong global convergence property', *SIAM Journal on Optimization* **10**.

[9] Distasi, R., Nappi, M. and Vitulano, S. [1997], 'Image compression by b-tree triangular coding', *IEEE Transactions on Communications* **45**(9), 1095–1100.

[10] Galić, I., Weickert, J., Welk, M., Bruhn, A., Belyaev, A. and Seidel, H.-P. [2005], Towards PDE-based image compression, *in* N. Paragios, O. Faugeras, T. Chan and C. Schnörr, eds, 'Variational, Geometric, and Level Set Methods in Computer Vision', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 37–48.

[11] Gilbert, J. R., Moler, C. and Schreiber, R. [1992], 'Sparse matrices in matlab: Design and implementation', *SIAM Journal on Matrix Analysis and Applications* **13**(1), 333–356.

[12] Golub, G. and Van Loan, C. [2013], *Matrix Computations*, Johns Hopkins Studies in the Mathematical Sciences, Johns Hopkins University Press.

[13] HarpinderKaur, H. [2012], 'Convergence of jacobi and gauss-seidel method and error reduction factor', *IOSR Journal of Mathematics* **2**, 20–23.

[14] Henson, V. [2003], 'Multigrid methods for nonlinear problems: An overview'.

[15] Hudson, G., Léger, A., Niss, B., Sebestyén, I. and Vaaben, J. [2018], 'JPEG-1 standard 25 years: past, present, and future reasons for a success', *Journal of Electronic Imaging* **27**(4), 040901.

[16] Idelson, A. I. and Severini, L. [2018], *Inpainting*, John Wiley & Sons, Ltd, pp. 1–4.

[17] Mainberger, M., Bruhn, A., Weickert, J. and Forchhammer, S. [2011], 'Edge-based compression of cartoon-like images with homogeneous diffusion', *Pattern Recognition* **44**(9), 1859–1873. Computer Analysis of Images and Patterns.

[18] Pinchover, Y. and Rubinstein, J. [2005], *An Introduction to Partial Differential Equations*, Cambridge University Press.

[19] Roelofs, G. [1999], *PNG: The Definitive Guide*, Creating and programming portable networks graphics, O'Reilly.

[20] Shewchuk, J. R. [1994], 'An introduction to the conjugate gradient method without the agonizing pain'.

[21] The MathWorks Inc. [2022], 'Matlab version: 9.13.0 (r2022b)'.
URL: *https://www.mathworks.com*

[22] Wolfram, S. [2002], *A New Kind of Science*, 1 edn, Wolfram Media.

[23] Zimmer, H. [2007], 'PDE-based image compression using corner information'.

A Appendix

This appendix contains MATLAB code for matrix generation, multigrid methods and the conjugate gradient method.

Jacobi Method

```
function u = relaxj(u,L,U,D,b)
    u = D\(b-(L+U)*u);
end
```

Gauss-Siedel Method

```
function u = relax(u,L,U,b)
    u = L\(b - U*u);
    u = transpose(L)\(b - transpose(U)*u);
end
```

Restriction Matrix Generation

```
function R = generateRestrict(nf)
    even = (mod(nf,2) == 0);
    nc = floor(nf/2);

    if(even)
        nf = nf+1;
    end


    is = zeros(1,nc^2);
    js = zeros(1,nf^2);
    vs = zeros(1,nf*nc);
    index = 1;

    for i = 1:nc
        for j = 1:nc
            is(index) = (i-1)*nc + j;
            js(index) = (2*(i-1)*nf + 2*(j-1) + 1);
            vs(index) = 1;
            index = index + 1;

            is(index) = (i-1)*nc + j;
            js(index) = (2*(i-1)*nf + 2*(j-1) + 2);
            vs(index) = 2;
            index = index + 1;

            is(index) = (i-1)*nc + j;
            js(index) = (2*(i-1)*nf + 2*(j-1) + 3);
            vs(index) = 1;
            index = index + 1;

            is(index) = (i-1)*nc + j;
            js(index) = (2*(i-1)*nf + 2*(j-1) + nf+1);
            vs(index) = 2;
            index = index + 1;

            is(index) = (i-1)*nc + j;
            js(index) = (2*(i-1)*nf + 2*(j-1) + nf+2);
            vs(index) = 4;
            index = index + 1;

            is(index) = (i-1)*nc + j;
            js(index) = (2*(i-1)*nf + 2*(j-1) + nf+3);
            vs(index) = 2;
            index = index + 1;

            is(index) = (i-1)*nc + j;
            js(index) = (2*(i-1)*nf + 2*(j-1) + 2*nf + 1);
            vs(index) = 1;
```

```
            index = index + 1;

            is(index) = (i-1)*nc + j;
            js(index) = (2*(i-1)*nf + 2*(j-1) + 2*nf + 2);
            vs(index) = 2;
            index = index + 1;

            is(index) = (i-1)*nc + j;
            js(index) = (2*(i-1)*nf + 2*(j-1) + 2*nf + 3);
            vs(index) = 1;
            index = index + 1;
        end
    end

    R = sparse(is,js,vs,nc^2,nf^2);

    if(even)
        cols = 1:((nf-1)*nf);
        rem = nf * (1:(nf));
        cols = setdiff(cols,rem);

        R = R(:,cols);
    end
end
```

Laplace Matrix Generation

```
function A = generatematrix(h,w)
    is = zeros(1,h*w);
    js = zeros(1,h*w);
    vs = zeros(1,h*w);
    index = 1;

    for i = 2:h-1
        for j = 2:w-1
            for deltax = -1:2:1
                is(index) = sub2ind([h;w],i,j+deltax);
                js(index) = sub2ind([h;w],i,j);
                vs(index) = 1;
                index = index + 1;
            end
            for deltay = -1:2:1
                is(index) = sub2ind([h;w],i+deltay,j);
                js(index) = sub2ind([h;w],i,j);
                vs(index) = 1;
                index = index + 1;
            end
            is(index) = sub2ind([h;w],i,j);
            js(index) = sub2ind([h;w],i,j);
            vs(index) = -4;
            index = index + 1;
        end
    end

    for i = 2:h-1
        is(index) = sub2ind([h;w],i,1);
        js(index) = sub2ind([h;w],i,1);
        vs(index) = -3;
        index = index + 1;
        for deltay = -1:2:1
            is(index) = sub2ind([h;w],i+deltay,1);
            js(index) = sub2ind([h;w],i,1);
            vs(index) = 1;
            index = index + 1;
        end
        is(index) = sub2ind([h;w],i,1+1);
        js(index) = sub2ind([h;w],i,1);
        vs(index) = 1;
        index = index + 1;

        is(index) = sub2ind([h;w],i,w);
        js(index) = sub2ind([h;w],i,w);
        vs(index) = -3;
        index = index + 1;
        for deltay = -1:2:1
            is(index) = sub2ind([h;w],i+deltay,w);
```

```
        js(index) = sub2ind([h;w],i,w);
        vs(index) = 1;
        index = index + 1;
    end
    is(index) = sub2ind([h;w],i,w-1);
    js(index) = sub2ind([h;w],i,w);
    vs(index) = 1;
    index = index + 1;
end

for j = 2:w-1
    is(index) = sub2ind([h;w],1,j);
    js(index) = sub2ind([h;w],1,j);
    vs(index) = -3;
    index = index + 1;
    for deltax = -1:2:1
        is(index) = sub2ind([h;w],1,j+deltax);
        js(index) = sub2ind([h;w],1,j);
        vs(index) = 1;
        index = index + 1;
    end
    is(index) = sub2ind([h;w],1+1,j);
    js(index) = sub2ind([h;w],1,j);
    vs(index) = 1;
    index = index + 1;

    is(index) = sub2ind([h;w],h,j);
    js(index) = sub2ind([h;w],h,j);
    vs(index) = -3;
    index = index + 1;
    for deltax = -1:2:1
        is(index) = sub2ind([h;w],h,j+deltax);
        js(index) = sub2ind([h;w],h,j);
        vs(index) = 1;
        index = index + 1;
    end
    is(index) = sub2ind([h;w],h-1,j);
    js(index) = sub2ind([h;w],h,j);
    vs(index) = 1;
    index = index + 1;
end

is(index) = sub2ind([h;w],1,1);
js(index) = sub2ind([h;w],1,1);
vs(index) = -2;
index = index + 1;
is(index) = sub2ind([h;w],1,2);
js(index) = sub2ind([h;w],1,1);
vs(index) = 1;
index = index + 1;
is(index) = sub2ind([h;w],2,1);
js(index) = sub2ind([h;w],1,1);
vs(index) = 1;
index = index + 1;

is(index) = sub2ind([h;w],h,1);
js(index) = sub2ind([h;w],h,1);
vs(index) = -2;
index = index + 1;
is(index) = sub2ind([h;w],h-1,1);
js(index) = sub2ind([h;w],h,1);
vs(index) = 1;
index = index + 1;
is(index) = sub2ind([h;w],h,2);
js(index) = sub2ind([h;w],h,1);
vs(index) = 1;
index = index + 1;

is(index) = sub2ind([h;w],1,w);
js(index) = sub2ind([h;w],1,w);
vs(index) = -2;
index = index + 1;
is(index) = sub2ind([h;w],1,w-1);
js(index) = sub2ind([h;w],1,w);
vs(index) = 1;
index = index + 1;
is(index) = sub2ind([h;w],2,w);
```

```
    js(index) = sub2ind([h;w],1,w);
    vs(index) = 1;
    index = index + 1;

    is(index) = sub2ind([h;w],h,w);
    js(index) = sub2ind([h;w],h,w);
    vs(index) = -2;
    index = index + 1;
    is(index) = sub2ind([h;w],h,w-1);
    js(index) = sub2ind([h;w],w,w);
    vs(index) = 1;
    index = index + 1;
    is(index) = sub2ind([h;w],h-1,w);
    js(index) = sub2ind([h;w],h,w);
    vs(index) = 1;
    index = index + 1;


    A = sparse(js,is,vs,h*w,h*w);
end
```

Two Grid Scheme

```
function u = twogrid(uf,A,AL,AU,ACL,ACU,R,P,b,nc)
    for relaxationcount = 1:2
        uf = relax(uf,AL,AU,b);
    end

    rf = b - (A*uf);

    rc = R*rf;

    ec = zeros(nc^2,1);
    for relaxationcount = 1:3
        ec = relax(ec,ACL,ACU,rc);
    end

    ef = P*ec;

    uf = uf + ef;
    for relaxationcount = 1:2
        uf = relax(uf,AL,AU,b);
    end
    u = uf;
end
```

V-Cycle Scheme

```
function u = Vmgrid(As,bf,uf,nf,rc,R,P,level,coarsest)
if(not(level==coarsest))
    AfL = tril(As{level},0);
    AfU = triu(As{level},1);
    for i = 1:rc
        uf = relax(uf,AfL,AfU,bf);
    end
    nc = floor(nf/2);

    uc = zeros(nc^2,1);
    bc = R{level}*(bf - As{level}*uf);
    uc = Vmgrid(As,bc,uc,nc,rc,R,P,level+1,coarsest);

    uf = uf + P{level}*uc;

    for i = 1:rc
        uf = relax(uf,AfL,AfU,bf);
    end
else
    uf = As{level}\bf;
end
u = uf;
end
```

Full Multigrid Scheme

```
function u = fmgrid(As,Bs,nf,rc,c0,R,P,level,coarsest)
    if(not(level==coarsest))
        nc = floor(nf/2);
        uc = fmgrid(As,Bs,nc,rc,c0,R,P,level+1,coarsest);
        uf = P{level}*uc;
    else
        uf = zeros(nf^2,1);
    end
    for c=1:c0
        uf = Vmgrid(As,Bs{level},uf,nf,rc,R,P,level,coarsest);
    end
    u=uf;
end
```

Conjugate Gradient Method

```
function u = cg(A,b,it,x)
    d = b - A*x;
    r = d;
    for i = 1:it
        Ad = A*d;
        a = (r'*r)/(d'*Ad);
        x = x + a*d;
        rn = r - a*Ad;
        bt = (rn'*rn)/(r'*r);
        d = rn + bt*d;
        r = rn;
    end

    u = x;
end
```