Academic Year 2021 – 2022
Research Unit: MAP5
Supervisor(s): Lionel Moisan

Internship Report

Anastasiia Storozhenko

September 7, 2022

Abstract

The general topic of the internship is image editing. In this report two approaches are discussed: multiresolution blending (based on pyramids) and Poisson editing (based on a variational problem). The latter is studied in more detail in application to different examples using different numerical methods. For experiments Python code was written. Main goals of this internship are to compare these two approaches to image editing and to study several numerical methods in application to Poisson editing.

This report consists of the following sections: *Introduction, Multiresolution blending, Poisson image editing, Numerical methods for Poisson editing, Comparison of numerical methods, Applications of Poisson editing, Conclusions, References,* and *Appendix*

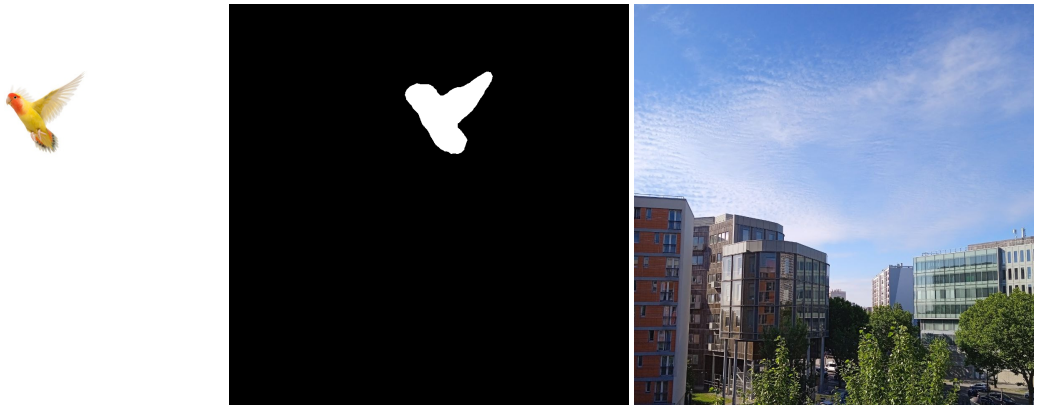
Keywords: seamless cloning, Poisson editing, pyramids, multiresolution blending.

Contents

1 Introduction

Let us consider the following problem called seamless cloning: we have two images and want to insert some region of one image (we will call it the "source") into another ("target") in accordance with some binary mask. Binary mask is an image every pixel of which is either 0 or 1. Every value simply shows us if the corresponding pixel from the source image is selected for the cloning. For convenience, we will assume that target, source and mask all have the same size of height N and width M. Additionally, we want the seam between the target and inserted region to be as hard to notice as possible.

Before we move further, let us discuss some important concepts. Throughout this report we will consider RGB images. Every channel (red, green and blue) is stored in computer as a matrix of size $N \times M$ each element of which is a number from 0 to 255. The combination of this three channels gives us the colour image. However, from mathematical point of view it's more convenient to look at images as maps from a set of coordinate pairs to (for example) the closed interval $[0, 1]$. So for an RGB image we get three maps that describe every channel.



(a) Source (b) Mask (c) Target

Figure 1.1: Example of a seamless cloning problem. The goal is to get an image showing the bird flying in the sky.

To better understand the problem, let us take a look at an example (Fig. 1.1). We have an image of a parrot and a photo of a city, and we want some combined image where it would look like the parrot is flying in the sky. Obviously, we want the result to look "natural". Simply cutting the region and pasting it into the target (Fig. 1.2a) is clearly not enough in most situations. Before the Poisson editing introduction, multiresolution blending was one of the widely used algorithms for solving this problem (Burt and Adelson [1983]). It relies on the use of Laplacian pyramids and produces much better results (Fig. 1.2b) compared to the naive approach. We will discuss in more details this method in the Section 2.

However, results obtained with multiresolution blending are still not satisfactory in a wide variety of cases. In 2003 a new method was proposed in Pérez et al. [2003] that considers gradients of both images. The main idea behind this is that adding a slow gradient of intensity over some region won't be noticeable to the human eye. This approach can be further expanded with a guidance field framework that allows to adjust the algorithm to different situations and tasks. All of this is covered in the Section 3.

(a) Naive (b) Multiresolution (c) Poisson

Figure 1.2: Results obtained by different approaches. We can see that multiresolution blending softens the white area around the bird. Poisson editing removes the white area completely, but makes the parrot visibly darker.

In terms of implementation Poisson editing boils down to solving a system of linear algebraic equations. Thus, we have to choose the best numerical method for this problem. In the Section 4 different numerical methods are described and in the Section 5 their performance is compared.

After studying numerical nuances we can finally move to real applications of Poisson editing. In the Section 6 the flexibility of the algorithm is demonstrated on multiple examples. Also, some cases are shown where this method performs poorly and some ideas for improvement are proposed.

2 Multiresolution blending

Multiresolution blending is based on the concept of an image pyramid — a concept that is widely used image processing. In this case we are particularly interested in two types of pyramids: Gaussian and Laplacian. Both of them are sequences of images at different scales obtained from the original image with certain procedures.

2.1 Scale-space representation

Before moving to the description of the pyramids it will be useful to discuss a term "scale-space" and connection between the Gaussian and Laplacian of an image. Let us define an image $I(x, y)$ as a function $I : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$. Then a convolution of this image with some filter kernel $K(x, y)$ is defined as

$$(I * K)(x, y) = \int_{\mathbb{R}^2} I(v, w)K(v - x, w - y)dxdy = \int_{\mathbb{R}^2} I(v - x, w - y)K(v, w)dxdy,$$

where both $I(x, y)$ and $K(x, y)$ are considered integrable. By applying a Gaussian filter in this case we will mean convolving the image with a Gaussian function:

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{(x^2+y^2)}{2\sigma^2}}, \tag{2.1}$$

where σ is a scale coefficient. As shown in Lowe [2004], it is possible to derive an interesting connection between a difference of Gaussians (DoG) at different scales and Laplacian of a Gaussian (LoG) using the heat equation in relation to the scale .

Proposition 2.1. *For some $k \geq 2$ the following equation is true:*

$$(k - 1)\sigma^2\Delta G = G(x, y, k\sigma) - G(x, y, \sigma) + O((k - 1)\sigma). \tag{2.2}$$

Proof. As was mentioned before, we will use the heat equation, given for some function $f(x, y, t)$ by

$$\frac{\partial f}{\partial t} = \alpha\Delta f,$$

where $t \geq 0$ and $\Delta \overset{\text{def}}{=} \nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$ is the Laplacian operator. It is easy to prove that $G(x, y, \sigma)$ is a solution to this equation by taking derivatives.

$$\frac{\partial G}{\partial \sigma} = \left(\frac{(x^2 + y^2)}{2\pi\sigma^5} - \frac{1}{\pi\sigma^3}\right) e^{-\frac{(x^2+y^2)}{2\sigma^2}} = \left(\frac{x^2 + y^2}{2\pi\sigma^3} - \frac{1}{\pi\sigma}\right) G(x, y, \sigma)$$

$$\frac{\partial G}{\partial x} = -\frac{x}{2\pi\sigma^4} e^{-\frac{(x^2+y^2)}{2\sigma^2}}$$

$$\frac{\partial^2 G}{\partial x^2} = \left(\frac{x^2}{2\pi\sigma^6} - \frac{1}{2\pi\sigma^4}\right) e^{-\frac{(x^2+y^2)}{2\sigma^2}}$$

$$\Delta G = \left(\frac{x^2+y^2}{2\pi\sigma^4} - \frac{1}{\pi\sigma^2}\right) G(x, y, \sigma)$$

From this results we get the equation

$$\frac{\partial G}{\partial \sigma} = \sigma \Delta G. \tag{2.3}$$

It is not exactly the heat equation, but it can be transformed into it with a substitution $r = \frac{1}{2}\sigma^2$. Now we approximate the partial derivative over scale with a finite difference for $\sigma' > \sigma$ and get

$$\frac{\partial G}{\partial \sigma} = \frac{G(x, y, \sigma') - G(x, y, \sigma)}{\sigma' - \sigma} + O(\sigma' - \sigma).$$

If $\frac{\sigma'}{\sigma} = k = const$ then it gives us the expression for the LoG,

$$(k-1)\sigma^2 \Delta G = G(x, y, k\sigma) - G(x, y, \sigma) + O((k-1)\sigma).$$

□

The Laplacian of Gaussian is widely used for edge detection. These edge detectors look for extrema of LoG applied to the image, so the constant $(k-1)$ in equation 2.2 doesn't have any influence in this application. That's why DoG is considered an approximation for the LoG operator. More information about scale-space theory can be found in paper Lindeberg [1994].

2.2 Image pyramids

In image processing applying a Gaussian filter to an image I means computing a convolution of this image with a discrete Gaussian kernel K. Gaussian kernel is a square array representing discrete version of the function 2.1, all elements of which sum up to 1. The result of convolution at a pixel with coordinates i, j is given by

$$(I * K)[i, j] = \sum_{m=1}^{M} \sum_{n=1}^{N} I[m, n] K[i - m, j - n].$$

To compute convolution for pixels close to the image border we need to artificially add pixel values around it. Some approaches for border-handling are zero padding, extending and mirroring as shown in Fig. 2.1. In this report mirroring was used for experimental results.

As was mentioned before, the pyramid of an image is a sequence of different versions of this image at different scales. For a Gaussian pyramid every level is constructed by applying a Gaussian filter to the previous level and then downsampling the result (usually by a factor of 2). Downsampling means a simple removal of every second pixel. So, we have the following procedure:

$$G_1 = I, \quad G_i = (K * G_{i-1})_{\downarrow 2}, \quad i \in \{1, \ldots, N\}.$$

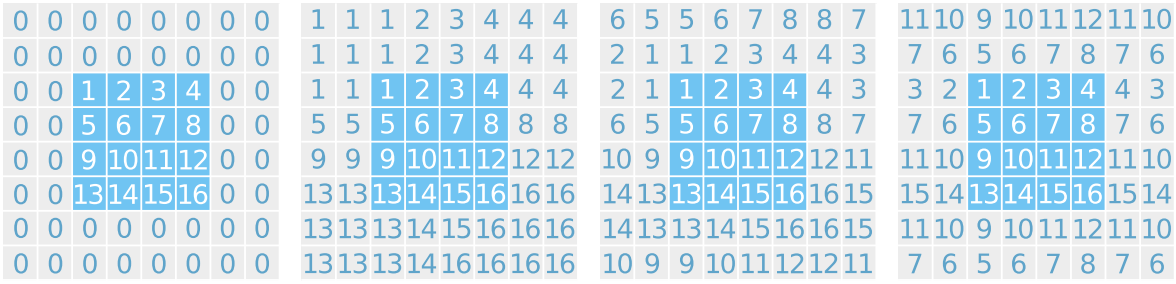


Figure 2.1: Different ways of border-handling: (a) — padding with zeroes; (b) — extending by cloning border pixels; (c) — mirroring including border pixels; (d) — mirroring excluding border pixels. Blue pixels corresponds to the initial image, the "thickness" of the added padding is defined by the size of the kernel. This example demonstrates a case of 5×5 kernel.

The Laplacian pyramid is then computed from the Gaussian pyramid by subtracting from every level the one above it as

$$L_N = G_N, \quad L_i = G_i - (G_{i+1})_{\uparrow 2}, \quad i \in \{1, \dots, N\}, \tag{2.4}$$

where $\uparrow 2$ denotes upsampling. There are different interpolation algorithms for upsampling; in experiments bicubic Keys interpolation was used. There is also an alternative way to define the Laplacian pyramid as

$$L_i = G_i - (K * G_i)\,. \tag{2.5}$$

In both cases we basically compute the difference of Gaussians which, as was shown before, is an approximation for the LoG, hence the name of the pyramid. Note that Equation 2.4 is the classic definition. The process of pyramids computation is shown in the Figure 2.2.

Laplacian pyramids have one important property: they are sufficient to perfectly reconstruct the image, as shown in

Proposition 2.2.

$$I = \sum_{i=1}^{N} (L_i)_{\uparrow 2^i}\,.$$

Proof. The proof is quite straightforward: we have to substitute terms in summation according to equations 2.4:

$$\sum_{i=1}^{N} (L_i)_{\uparrow 2^i} = (G_N)_{\uparrow 2^{N-1}} + \big(G_{N-1} - (G_N)_{\uparrow 2}\big)_{\uparrow 2^{N-2}} + \big(G_{N-2} - (G_{N-1})_{\uparrow 2}\big)_{\uparrow 2^{N-3}} + \dots +$$
$$+ \big(G_1 - (G_2)_{\uparrow 2}\big) = (G_N)_{\uparrow 2^{N-1}} + (G_{N-1})_{\uparrow 2^{N-2}} - (G_N)_{\uparrow 2^{N-1}} + (G_{N-2})_{\uparrow 2^{N-3}} -$$
$$- (G_{N-1})_{\uparrow 2^{N-2}} + \dots + G_1 - (G_2)_{\uparrow 2} = G_1 = I.$$

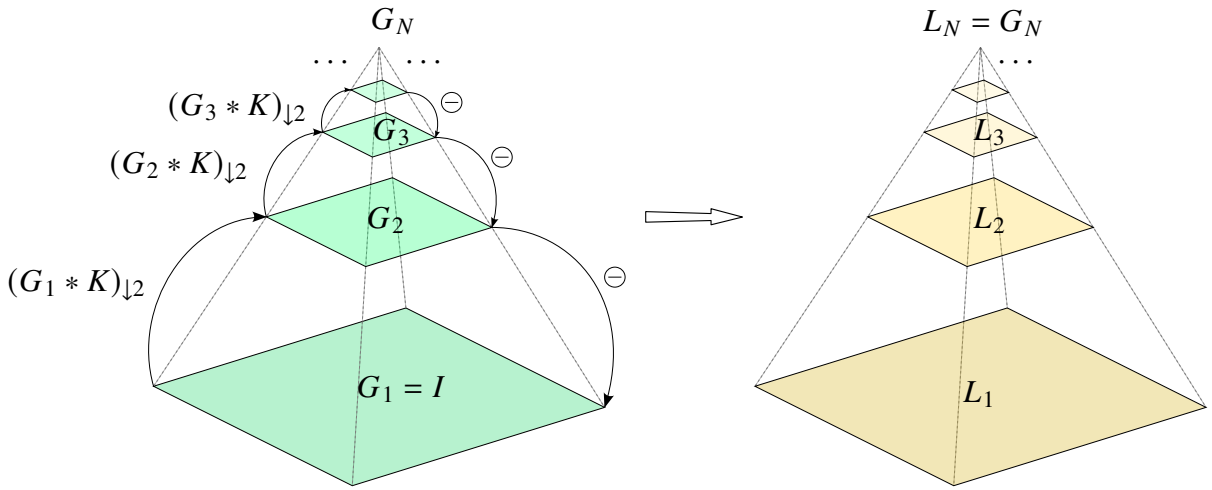


Figure 2.2: Visual explanation of how the Gaussian (on the left) and Laplacian (on the right) pyramids are computed. Arrows on the left of the Gaussian pyramid show the procedure for computing each level of the Gaussian pyramid itself. Arrows on the right show the procedure for the Laplacian pyramid. Symbol $\ominus$ stands for upsampling the level and subtracting it from the level below (pointed at by a corresponding arrow).

This statement also holds true for the alternative formulation:

$$\sum_{i=1}^{N} (L_i)_{\uparrow 2^i} = (G_N)_{\uparrow 2^{N-1}} + (G_{N-1} - K * G_{N-1})_{\uparrow 2^{N-2}} + (G_{N-2} - K * G_{N-2})_{\uparrow 2^{N-3}} +$$
$$+ \cdots + (G_1 - K * G_1) = (K * G_{N-1})_{\uparrow 2^{N-2}} + \left((K * G_{N-2})_{\downarrow 2} - K * G_{N-1}\right)_{\uparrow 2^{N-2}} +$$
$$+ \left((K * G_{N-3})_{\downarrow 2} - K * G_{N-2}\right)_{\uparrow 2^{N-3}} + \cdots + G_1 - K * G_1 = G_1 = I.$$

□

2.3 Application of multiresolution blending

The property of reconstruction is the main idea in the multiresolution blending. For a target image T, a source image S and a mask M the algorithm is described by following steps:

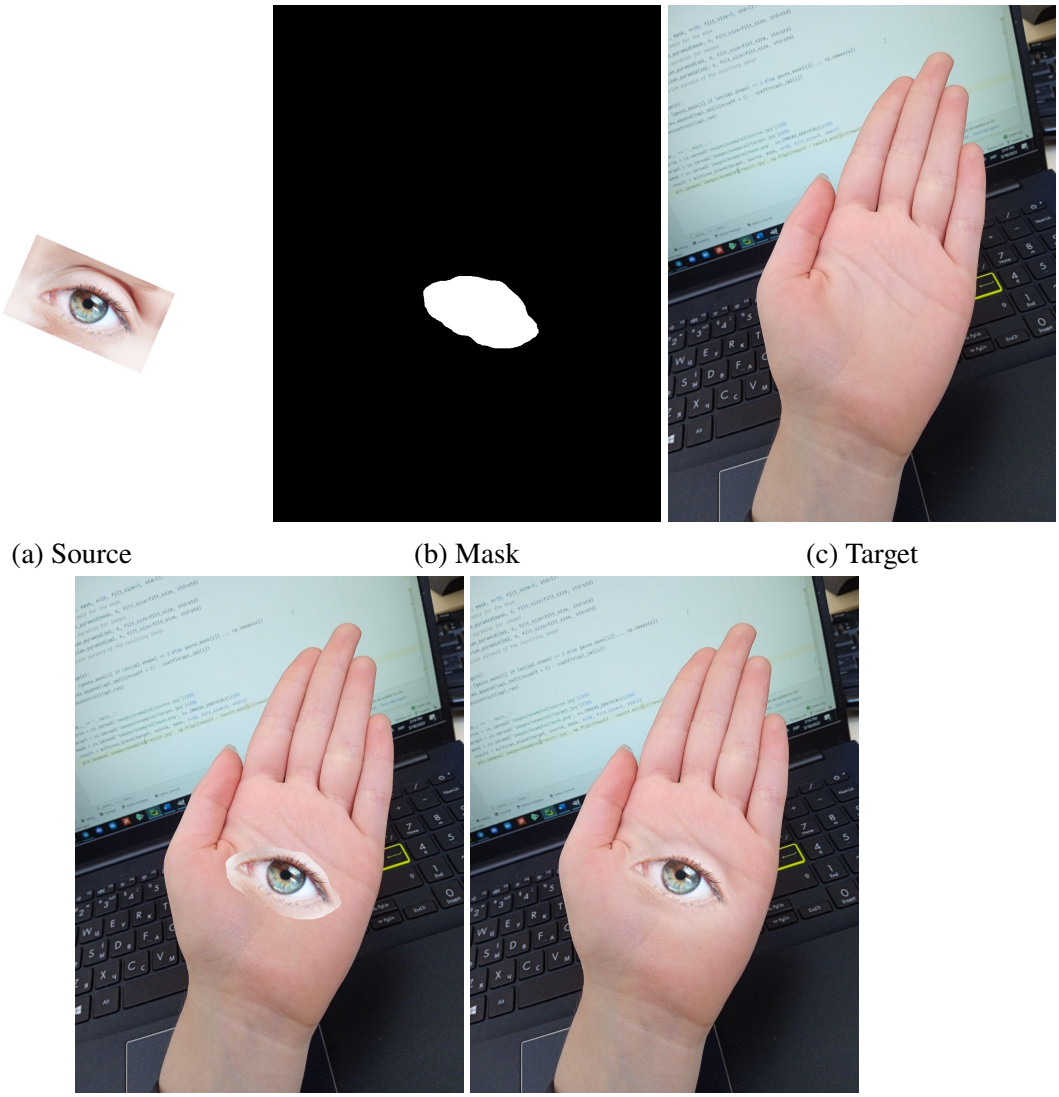
1. Compute Laplacian pyramids for T and S (L^T and L^S respectively).
2. Compute a Gaussian pyramid for M (G^M).
3. Compute a combined Laplacian pyramid for the resulting image R:

$$L_i^R = G_i L_i^S + (1 - G_i) L_i^T,$$

   where i is an index of the level.
4. Reconstruct the result from the pyramid L^R.

This simple and fast algorithm can be used in some easier cases of image editing. It provides good results in cases where textures and colors of source and target are already close enough (Fig. 2.3). It also can be used to quickly get results for bigger regions as shown in Fig. 2.3. However, in a lot of cases results are not satisfying: for example, when the source contains the object of interest with a background obviously different in colour from the target image. In Fig. 2.5 we can see an example of this. Also, this tool is limited to blending and thus doesn't perform well in other applications such as texture swapping. In Fig. 2.6 we can observe that while the algorithm does an acceptable job colour-wise, it preserves the orange colour in the result.

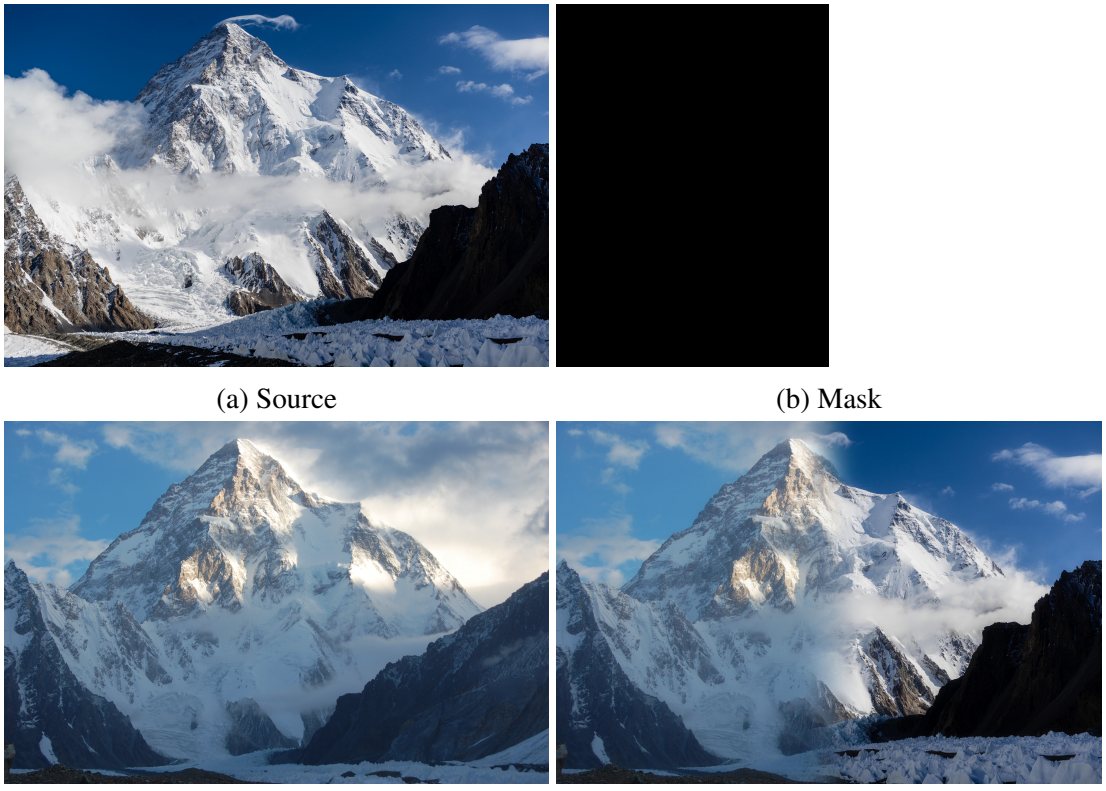
These examples show us that multiresolution blending is a decent method, but unfortunately it is limited to simple situations. But in real-life applications it would be desirable to make loose selections of objects on different type of backgrounds and still get reasonable results. Besides, for adequate results we need extra pixels from the source outside the masked region (this is caused by the blur applied to the mask). Also, image editing is not limited to the blending, so it would be great to have a single tool that is useful for different applications. Soon we will see that this is precisely what Poisson editing provides.



(a) Source (b) Mask (c) Target

(d) Naive (e) Multiresolution

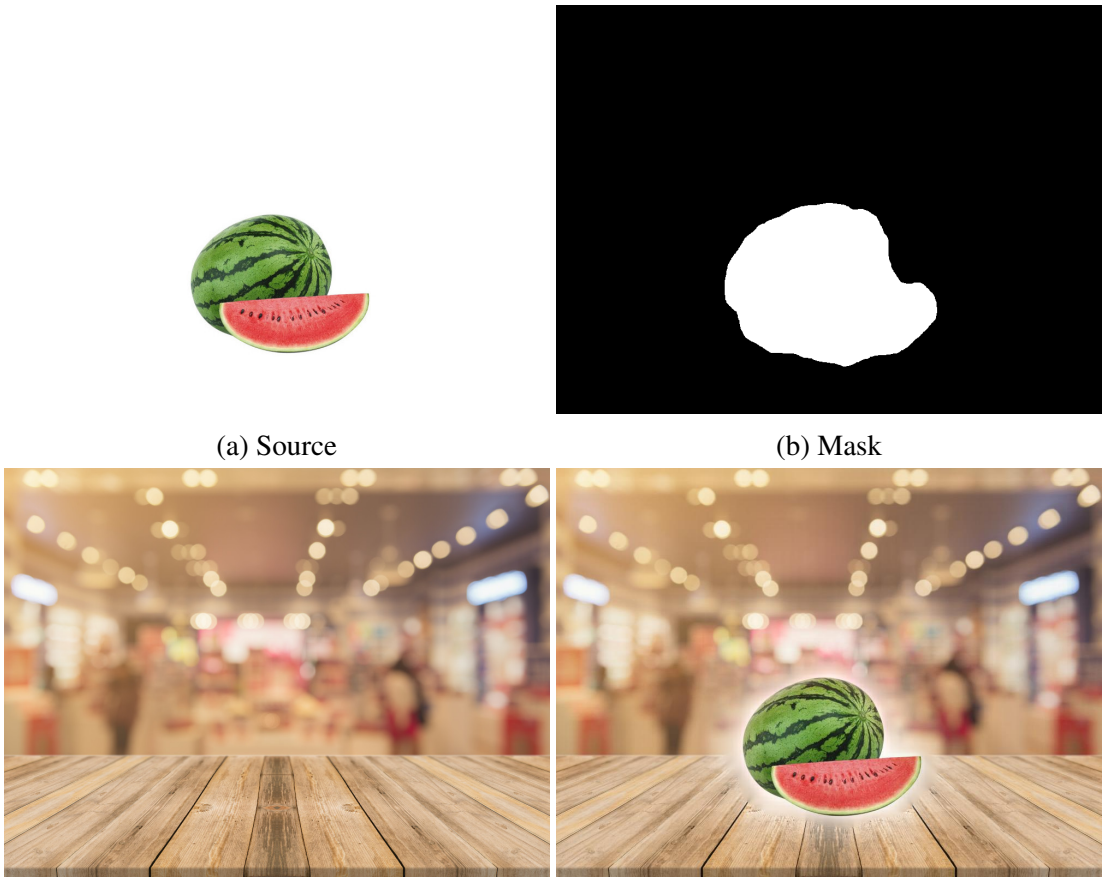
Figure 2.3: Successful case of multiresolution blending: seam in the final result is not visible. Note that this is one of simpler cases where textures and colours of the object and background are quite close. However, it is not trivial, and multiresolution blending shows obvious superiority to the naive approach.



(a) Source (b) Mask

(c) Target (d) Result

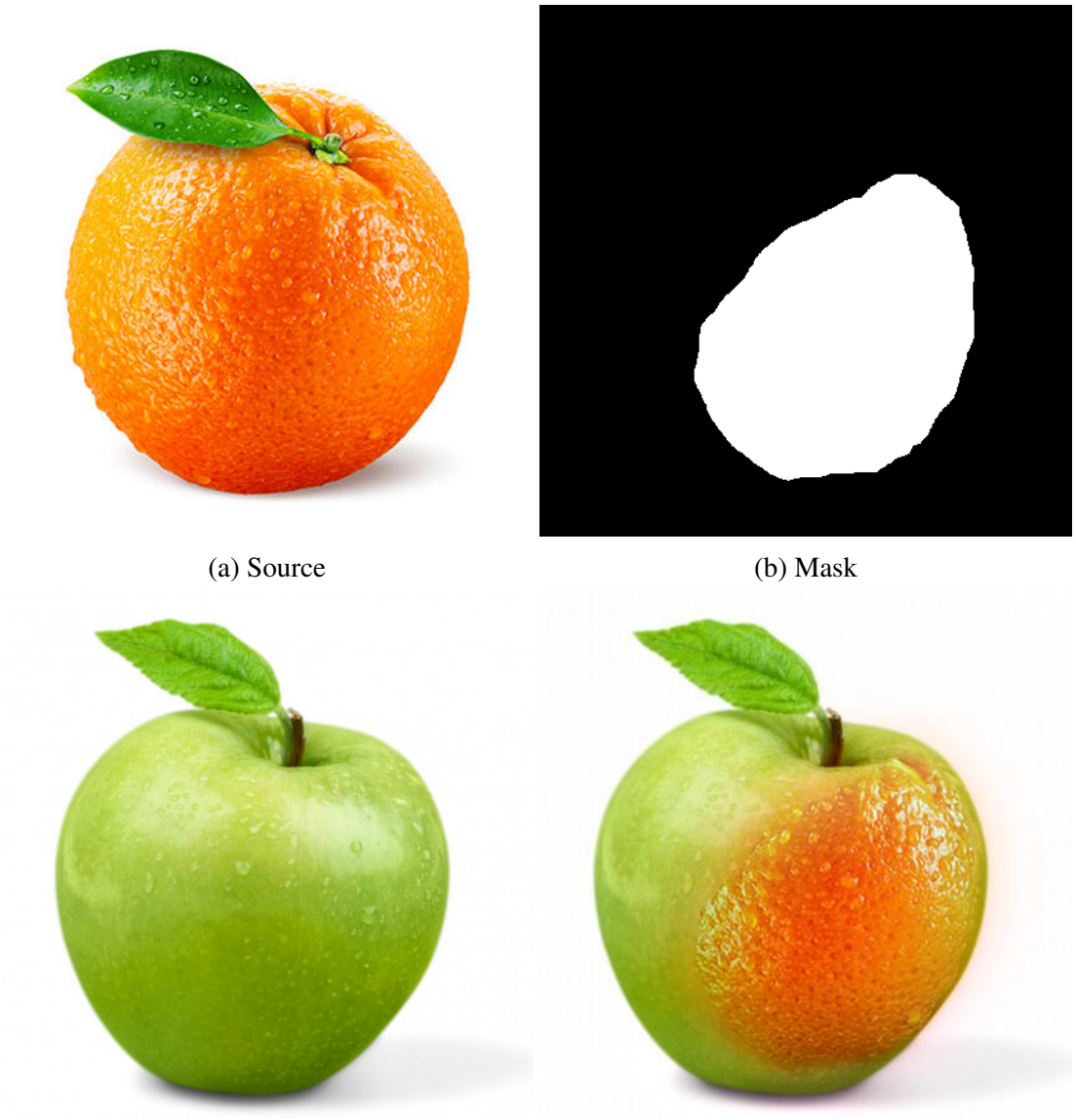
Figure 2.4: Multiresolution blending can also be used to some extent in the task of combining two pictures of the same object in slightly different conditions. In these example the mask was blurred before application of the method. That helped to get more details from both halves of the picture in the final result.



(a) Source (b) Mask

(c) Target (d) Result

Figure 2.5: Failure case: object has a distinguishable halo around it in the result. The reason for that is that the mask clips the object together with the white background which is different from the colour of the table and watermelon. Also, the algorithm does not consider the texture of the table and we see an unwanted blurring around the watermelon in the result.



(a) Source (b) Mask

(c) Target (d) Result

Figure 2.6: Semi-successful case: texture swapping. In this example we see that the algorithm smoothens the transition between two textures. However, it doesn't provide a way to complitely blend orange texture in apple colour-wise (e.g., making this patch green).

3 Poisson image editing

3.1 Continuous problem

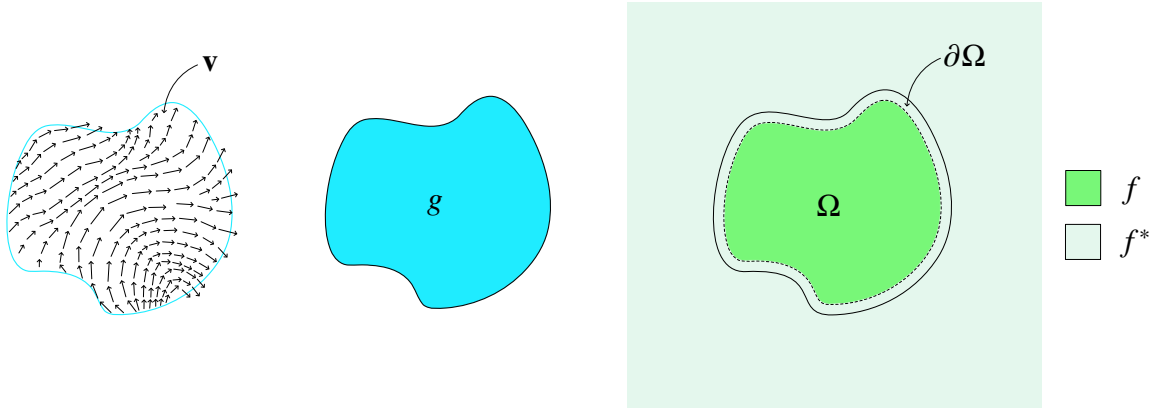


Figure 3.1: Notations used in the report. The image on the left depicts the guidance field, in the centre — the source image and on the right — unknown image f defined on Ω.

First we have to formalize the problem in the continious setting. Let S be the image definition domain and a closed subset of $\mathbb{R}^2$, and let Ω be a measurable open subset of S with boundary $\partial\Omega$. Let $f : \Omega \to [0, 1]$ be the unknown intensity function in the region of interest, $f^* : S\backslash\Omega \to [0, 1]$ — known function that represents the intensities in the target image outside of the region and $g : \Omega \to [0, 1]$ — intensity function of the source image (see Figure 3.1). We are going to assume that both f and g are differentiable.

For a simpler application like cloning a region of one image into another, we can formulate the optimization problem

$$\begin{cases} \min\limits_f \iint\limits_\Omega |\nabla f - \nabla g|^2 dxdy \\ f\big|_{\partial\Omega} = f^*\big|_{\partial\Omega}, \end{cases}$$

where $|\cdot|$ denotes the Euclidean norm. For other applications we can generalize this with the use of guidance fields. Let $\mathbf{v}$ be some vector field (non-conservative in general). Then we can reformulate our problem as

$$\begin{cases} \min\limits_f \iint\limits_\Omega |\nabla f - \mathbf{v}|^2 dxdy \\ f\big|_{\partial\Omega} = f^*\big|_{\partial\Omega}. \end{cases} \tag{3.1}$$

Proposition 3.1. *The solution to the minimization problem 3.1 also satisfies the following Poisson equation with Dirichlet boundary conditions:*

$$\Delta f = \mathrm{div}\mathbf{v} \text{ in } \Omega, \text{ with } f\big|_{\partial\Omega} = f^*\big|_{\partial\Omega}.$$

3.2 Discrete problem

Now it's time to move to numerical solving. There are two ways to approach this: discretize the corresponding Poisson equation or the problem 3.1 itself. It is a good practice to discretize the

integral formulation as it gives more control over energy function itself, and we will also follow this path. First, we define some discrete grid on $\mathbb{R}^2$. For convenience, we will keep notations S and Ω for the discrete counterparts of these sets defined on the grid. For each pixel p in S let N_p be the set of its 4-connected neighbours that are in S (4-connected neighbourhood consists of the pixels right above, below, left and right of p). Let $p \sim q$ denote an adjacency relation between pixels p and q based on the 4-connectivity. The boundary of the region is now defined as $\partial\Omega = \{p \in S \setminus \Omega : N_p \cap \Omega \neq \emptyset\}$. Let f_p be the value of f at p. With this we obtain the discrete form for the energy:

$$\min_{f|_\Omega} \sum_{(p,q):\ p\sim q \wedge \{p,q\}\cap\Omega\neq\emptyset} (f_p - f_q - v_{pq})^2,\ \forall p \in \partial\Omega : f_p = f_p^*, \tag{3.2}$$

where f_p^* are known values from the target image.

Proposition 3.2. *A solution to the optimization problem 3.2 also satisfies the following system of linear algebraic equations:*

$$\forall p \in \Omega : \quad |N_p| f_p - \sum_{q\in N_p\cap\Omega} f_q = \sum_{q\in N_p\cap\partial\Omega} f_q^* + \sum_{q\in N_p} v_{pq} \tag{3.3}$$

Proof. Let $E = \sum_{(p,q):\ p\sim q\wedge\{p,q\}\cap\Omega\neq\emptyset} (f_p - f_q - v_{pq})^2$. Then the minimum of E satisfies the system of equations

$$\forall x \in \Omega : \frac{\partial E}{\partial f_x} = 0.$$

Let's compute this derivative for each x in Ω.

$$\frac{\partial E}{\partial f_x} = \frac{\partial}{\partial f_x}\left[\sum_{q\in N_x} (f_x - f_q - v_{xq})^2\right] = 2\left[\sum_{q\in N_x} (f_x - f_q) - \sum_{q\in N_x} v_{xq}\right] =$$

$$= 2\left[|N_x| f_x - \sum_{N_x\cap\Omega} f_q - \sum_{N_x\cap\partial\Omega} f_q^* - \sum_{q\in N_x} v_{xq}\right] = 0$$

This gives us the desired system of equations,

$$\forall x \in \Omega : \quad |N_x| f_x - \sum_{q\in N_x\cap\Omega} f_q = \sum_{q\in N_x\cap\partial\Omega} f_q^* + \sum_{q\in N_x} v_{xq}.$$

□

So we transformed the minimization problem into a system of linear equations. For future uses let's rewrite it in matrix form

$$A\mathbf{f} = \mathbf{b}. \tag{3.4}$$

This is a sparse $|\Omega|^2 \times |\Omega|^2$ matrix with each row containing at most 5 non-zero elements. To construct it we have to choose the ordering of the pixels of the region. In this report a so-called

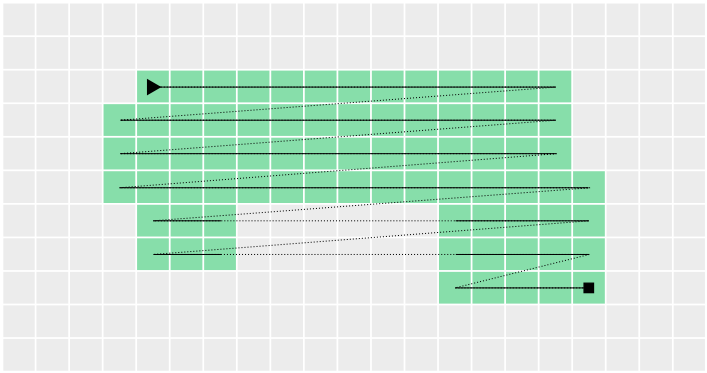


Figure 3.2: Visualization of the raster ordering procedure: triangle corresponds to the first pixel, square - to the last one. Green colour indicates the region Ω.

"raster scan order" is used: we "scan" every row from left to right, starting from the top row, and assign a number to each pixel as shown in figure 3.2. This matrix is also symmetric and positive definite, and to prove semi-positive definiteness we will use the Gershgorin circle theorem (the proof can be found at Bárány and Solymosi [2017]).

Theorem 3.1 (Gershgorin circle theorem)**.** *Let A be a square $n \times n$ matrix with elements $a_{ij} \in \mathbb{C}$. For $i \in \{1, \dots, n\}$ let R_i be the sum of the absolute values of the non-diagonal elements in i-th row:*

$$R_i = \sum_{j \neq i} |a_{ij}|.$$

Let $D(a_{ii}, R_i) \subseteq \mathbb{C}$ be a closed disc with a center in a_{ii} and radius R_i. Then every eigenvalue of A lies within at least one of the discs $D(a_{ii}, R_i)$.

Proposition 3.3. *The matrix A from the system 3.4 is symmetric and semi-positive definite.*

Proof. Symmetry is almost obvious. Non-diagonal elements of the matrix A can be either a 0 or -1. For some pixel p_i non-zero non-diagonal entries in i-th row correspond to neighbours of p_i. Due to the symmetry, these neighbours will have same entries in i-th column of corresponding rows.

Positive semi-definiteness follows from the Theorem 3.1. Diagonal entries correspond to the number of neighbours of the pixel; i-th row contains at most $|N_{p_i}|$ non-zero elements each of which is equal to -1. This gives us the diagonal dominance:

$$|a_{ii}| \geq \sum_{j \neq i} |a_{ij}|.$$

According to the theorem every eigenvalue of A lies within at least one disc and therefore all of them are non-negative. □

The choice of guidance field depends on the task. We already mentioned a conservative field $\mathbf{v} = \nabla g$ (in discrete form: $v_{pq} = g_p - g_q$). In some cases (for example, when we have strong

gradients in the target image that we don't want to be overlapped with the source) it is better to use mixed gradients:

$$\forall x \in \Omega : \mathbf{v}(x) = \begin{cases} \nabla f^*(x), & \text{if } |\nabla f^*(x)| > |\nabla g(x)| \\ \nabla g(x), & \text{otherwise} \end{cases}$$

or in discrete form:

$$\forall \langle p, q \rangle \in \Omega : v_{pq} = \begin{cases} f_p^* - f_q^* & \text{if } |f_p^* - f_q^*| > |g_p - g_q| \\ g_p - g_q, & \text{otherwise.} \end{cases} \tag{3.5}$$

There are many other types of guidance fields, but we will concentrate on these two in this work. In conclusion it is worth mentioning that for multi-channel images, Poisson editing is applied to every channel separately.

4 Numerical methods for Poisson editing

Now that we have discrete formulation of the problem we can apply different numerical methods and compare them. In this work the following numerical methods were studied:

1. Gradient descent.
2. Modified Richardson iteration.
3. Gauss-Seidel method.
4. Successive over-relaxation.
5. Conjugated gradient method.
6. Gradient descent with momentum.
7. Nesterov accelerated gradient.

By t we will denote the index of iterations, so f_t is an estimation of the solution at the t-th iteration. We will say that a numerical method *converges* for a given norm $\|\cdot\|$ if

$$||f_t - f^*|| \xrightarrow[t\to\infty]{} 0,$$

where f^* is the true solution. An important class of numerical methods are fixed-point methods, each iteration of which is given by the formula

$$x_{t+1} = g(x_t),$$

where g is some function.

Let A be a linear operator $A : X \to Y$, where X and Y are some real normed vector spaces with norms $\|\cdot\|_X$ and $\|\cdot\|_Y$ respectively. Then the norm of this operator is defined as

$$\|A\| = \sup_{\mathbf{x}:\|\mathbf{x}\|_X \le 1} \|A\mathbf{x}\|_Y \tag{4.1}$$

From the definition 4.1 we can get two important properties of the norm:

$$\|A\mathbf{x}\|_Y \le \|A\|\|\mathbf{x}\|_X, \tag{4.2}$$

$$\|AB\| \le \|A\|\|B\|, \tag{4.3}$$

where B is another linear operator, $B : Y \to W$. In the context of our problem we will look at operators that act inside one space, i.e, $A : X \to X$. These operators can be represented as a square matrix. Let $\|\cdot\|_X$ and $\|\cdot\|_Y$ be euclidean norms.

For the operator norm defined by Equation 4.1 and for any linear operator A it can be proved that

$$\|A\| = \max_{\lambda\in\Lambda} |\lambda|, \tag{4.4}$$

where Λ is the spectrum of A.

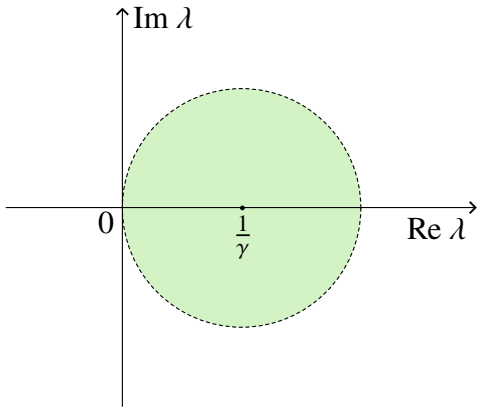


Figure 4.1: Region containing eigenvalues.

Proposition 4.1. *The iteration*

$$\mathbf{f}_{t+1} = \mathbf{f}_t - \gamma(B\mathbf{f}_t - \mathbf{d})$$

converges for any symmetric positive definite matrix B and γ such that

$$0 < \gamma < \frac{2}{\max \Lambda},$$

where Λ is the spectrum (set of eigenvalues) of B.

Proof. First, let us rewrite the iteration formula as

$$\mathbf{f}_{t+1} = \mathbf{f}_t - \gamma(B\mathbf{f}_t - \mathbf{d}) = (I - \gamma B)\mathbf{f}_t + \gamma\mathbf{d}.$$

Next we write down the norm of the absolute error on the step t

$$\begin{aligned}\|\mathbf{f}^* - \mathbf{f}_t\| &= \|\mathbf{f}^* - (I - \gamma B)\mathbf{f}_{t-1} + \gamma\mathbf{d}\| = \|(I - \gamma B)\mathbf{f}^* - (I - \gamma B)\mathbf{f}_{t-1}\| = \\ &= \|(I - \gamma B)\mathbf{f}^* - (I - \gamma B)\left[(I - \gamma B)\mathbf{f}_{t-2} + \mathbf{d}\right]\| = \|(I - \gamma B)^2\mathbf{f}^* - (I - \gamma B)^2\mathbf{f}_{t-2}\|\end{aligned}$$

After repeating substitution for t times we get

$$\|\mathbf{f}^* - \mathbf{f}_t\| = \|(I - \gamma B)^t(\mathbf{f}^* - \mathbf{f}_0)\|.$$

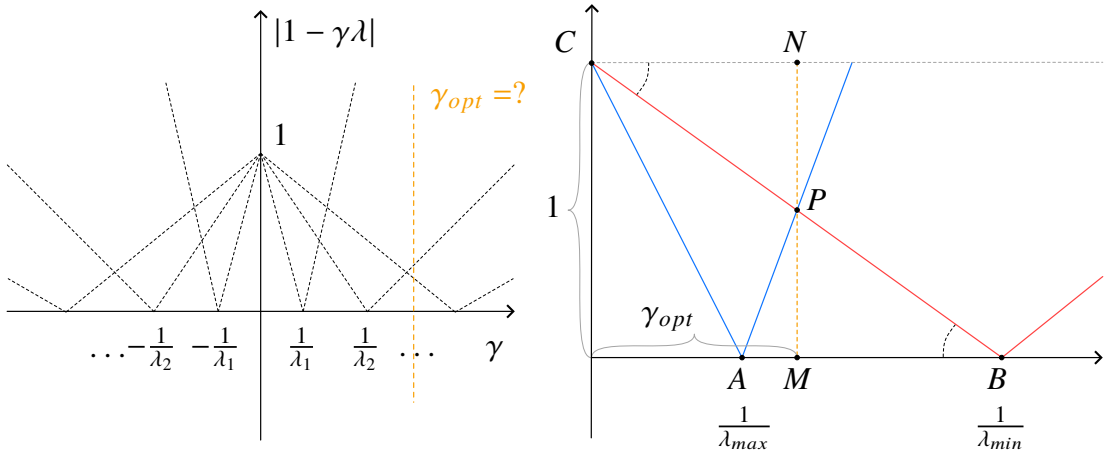
Using inequalities 4.3 and 4.2 we obtain

$$\|\mathbf{f}^* - \mathbf{f}_t\| \leq \|(I - \gamma B)^t\|\|\mathbf{f}^* - \mathbf{f}_0\| \leq \|I - \gamma B\|^t\|\mathbf{f}^* - \mathbf{f}_0\|$$

This expression converges to zero only if $\|I - \gamma B\| < 1$. For norm 4.4 we get

$$\|I - \gamma B\| = \max_{\lambda \in \Lambda} |1 - \gamma\lambda| < 1 \implies \forall \lambda_k \in \Lambda : |1 - \gamma\lambda_k| < 1,$$

where Λ is the spectrum of B. In general eigenvalues can be complex. Conditions above mean that for convergence all eigenvalues have to lie inside a circle on the complex plane of a radius $\frac{1}{\gamma}$ centered in $\frac{1}{\gamma}$ (see Figure 4.1). It is clear that all these eigenvalues must have positive real components. Now it is time to recall that in our case matrix B is symmetric, so its eigenvalues are real numbers. Now we can get bounds for γ through simple transformations



(a) (b)

Figure 4.2: Illustrations for the proof.

$$-1 < 1 - \gamma\lambda_k < 1,$$

$$-2 < -\gamma\lambda_k < 0,$$

$$\frac{2}{\lambda_k} > \gamma > 0.$$

Using the fact $\frac{2}{\lambda_k} > \frac{2}{\max \lambda_j}$ we get the final conditions for the convergence of the method for some symmetric positive definite matrix B,

$$0 < \gamma < \frac{2}{\max \lambda_j}.$$

We can also find optimal γ by solving the problem

$$\min_{\gamma} \max_{\lambda \in \Lambda} |1 - \gamma\lambda|. \tag{4.5}$$

It is convenient to use a geometrical approach. Let us consider curves given by the function $F_\lambda(\gamma) = |1-\gamma\lambda|$ for different eigenvalues (see Figure 4.2a). To present geometric interpretation of this optimisation problem let us enumerate eigenvalues in descending order ($\lambda_1 \geq \lambda_2 \geq \dots$). Now let $x_1, x_2, \dots$ be points of intersection of the line $\gamma = \gamma_{opt}$ with curves $F_\lambda(\gamma)$. To solve the optimization problem 4.5 we have to find such γ_{opt} that minimizes the maximum value of x_i for this γ. As was shown before, the optimal γ should lie between zero and $\frac{2}{\max \lambda_j}$. In Figure 4.2b the corresponding part of the plot is shown.
It is obvious from this image that for the optimal γ the line $\gamma = \gamma_{opt}$ will go through the intersection point of $F_{\lambda_{min}}(\gamma)$ and $F_{\lambda_{max}}(\gamma)$. Using the facts $\tan(\angle BCN) = \lambda_{min}$ and $\tan(\angle PBM) = \lambda_{max}$ we can compute $NP = \lambda_{min}\gamma_{opt}$ and $PM = \lambda_{max}(\gamma_{opt} - \frac{1}{\lambda_{max}}) = \lambda_{max}\gamma_{opt} - 1$. As $NP + PM = NM$ and $NM = 1$ we can combine two previous expressions to get the following equation:

$$\lambda_{min}\gamma_{opt} + \lambda_{max}\gamma_{opt} - 1 = 1,$$

which gives us the optimal value for γ:

$$\gamma_{opt} = \frac{2}{\lambda_{min} + \lambda_{max}}.$$

□

4.1 Gradient descent

We start with the classic gradient descent. As we have a system of linear equations, the first step is to reformulate it as a minimization problem:

$$\min_{\mathbf{f}\in\mathbb{R}^d} ||A\mathbf{f} - \mathbf{b}||^2 = \min_{\mathbf{f}\in\mathbb{R}^d} L(\mathbf{f}).$$

Gradient of the target function is $\nabla L = 2A^T(A\mathbf{f} - \mathbf{b})$. This gives us the following procedure:

1. Compute the error $\mathbf{r}_t = A\mathbf{f}_t - \mathbf{b}$.
2. Update $\mathbf{f}_{t+1} = \mathbf{f}_t - 2\gamma A^T\mathbf{r}_t$.
3. Repeat until $||\mathbf{r}_t|| < \varepsilon$.

Here γ is a gradient descent step. We will see later that gradient descent is not an effective numerical method for this problem. But before that let us prove the convergence of this method.

Proposition 4.2. *Gradient descent converges for a symmetric positive definite matrix A for any γ such that*

$$0 < \gamma < \frac{1}{(\max \Lambda)^2},$$

where Λ is the spectrum of the matrix A.

Proof. Let $2A^TA = B$ and $2A^T\mathbf{b} = \mathbf{d}$. Then we can simply use Proposition 4.1 which gives us

$$0 < \gamma < \frac{2}{\max \tilde{\Lambda}},$$

where $\tilde{\Lambda}$ is the spectrum of $2A^TA$. For a symmetric positive definite matrix A we get

$$0 < \gamma < \frac{1}{(\max \Lambda)^2},$$

where Λ is the spectrum of the matrix A.

□

4.2 Modified Richardson iteration

This method was proposed by Richardson and Glazebrook [1911]. It relies on the fixed-point iteration. Steps of this method are shown below, where γ is a constant parameter.

1. Compute the error $\mathbf{r}_t = A\mathbf{f}_t - \mathbf{b}$.
2. Update $\mathbf{f}_{t+1} = \mathbf{f}_t - \gamma\mathbf{r}_t$
3. Repeat until $||\mathbf{r}_t|| < \varepsilon$.

Convergence conditions are the same as described in Proposition 4.1.

Proposition 4.3. *Modified Richardson iteration converges for any symmetric positive definite matrix A and any γ that satisfies inequalities*

$$0 < \gamma < \frac{2}{\max \Lambda}.$$

This method is also connected to the gradient descent. Let B be a matrix such that $A = B^T B$ (considering the symmetry it can be rewritten as $A = B^2$). This matrix exists as A is positive definite. Also, positive definitiveness leads to the fact that A and B are invertible. That allows us to rewrite the system of equations as

$$B\mathbf{f} = B^{-1}\mathbf{b}.$$

And so this method can be interpreted as a gradient descent for the following problem:

$$\min_{\mathbf{f}\in\mathbb{R}^d} \frac{1}{2}||B\mathbf{f} - B^{-1}\mathbf{b}||^2 = \min_{\mathbf{f}\in\mathbb{R}^d} \tilde{L}(\mathbf{f}),\ \nabla\tilde{L} = A\mathbf{f} - \mathbf{b}.$$

4.3 Gauss-Seidel method

Gauss-Seidel method is a widely known iterative method used to solve systems of linear equations and especially sparse ones. It is based on the decomposition of the matrix A into its lower triangular and strictly upper triangular components ($A = L + U$). Then iterations are given by the formula:

$$\mathbf{f}^{(t+1)} = L^{-1}(\mathbf{b} - U\mathbf{f}^{(t)}).$$

However, forward substitution can be used to get the following (where a_{ij} and f_i are components of A and $\mathbf{f}$ correspondingly and n is the length of $\mathbf{f}$):

$$f_i^{(t+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{t+1} - \sum_{j=i+1}^{n} a_{ij}x_j^t\right),\ \ i = 1, 2, \ldots, n.$$

Gauss-Seidel method converges for any symmetric positive definite matrix A (see successive over-relaxation).

4.4 Successive over-relaxation

Successive over-relaxation (or SOR) is a generalization of the Gauss-Seidel algorithm. Every iteration is computed by the formula:

$$f_i^{(t+1)} = (1-\omega) f_i^{(t)} + \frac{\omega}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{t+1} - \sum_{j=i+1}^{n} a_{ij} x_j^t \right), \quad i = 1, 2, \ldots, n,$$

where $\omega > 1$ is a parameter. As was proven by A.Ostrowski (or as discussed in James and Riha [1975]), for a symmetric positive definite matrix A SOR converges for $0 < \omega < 2$.

4.5 Conjugate gradient method

Conjugate gradient method is used for systems with a symmetric positive definite matrix A. It is based on the fact that the exact solution of a system can be expressed as a linear combination of n mutually conjugated vectors $\mathbf{p}_k$ where n is the dimensionality of the system. Two vectors $\mathbf{x}$ and $\mathbf{y}$ are said to be conjugated with respect to matrix A if $\mathbf{x}^T A \mathbf{y} = 0$.

However, if these conjugated vectors are built in a specific way, it is possible to obtain a good approximation of the solution without computing all n vectors. A full algorithm is shown below.

1. Initialize $\mathbf{r}_0 = \mathbf{b} - A\mathbf{f}_0$, $\mathbf{p}_0 = \mathbf{r}_0$.
2. Until $||\mathbf{r}_t|| < \varepsilon$ (or until meeting other criteria):
   (a) Compute $\alpha_t = \frac{\mathbf{r}_t^T \mathbf{r}_t}{\mathbf{p}_t^T A \mathbf{p}_t}$.
   (b) Update $\mathbf{f}_{t+1} = \mathbf{f}_t + \alpha_t \mathbf{p}_t$.
   (c) Compute new $\mathbf{r}_{t+1} = \mathbf{r}_t - \alpha_t A \mathbf{p}_t$.
   (d) Compute $\beta_k = \frac{\mathbf{r}_{t+1}^T \mathbf{r}_{t+1}}{\mathbf{r}_t^T \mathbf{r}_t}$.
   (e) Compute $\mathbf{p}_{t+1} = \mathbf{r}_{t+1} + \beta_t \mathbf{p}_t$.

4.6 Gradient descent with momentum

The main idea behind this method is to use information about the previous step which adds "momentum" to the descent.

1. Compute the step $\mathbf{d}_{t+1} = \mu \mathbf{d}_t - \gamma \nabla L(\mathbf{f}_t)$.
2. Update $\mathbf{f}_{t+1} = \mathbf{f}_t + \mathbf{d}_{t+1}$.
3. Repeat until the stop criterion is met.

Substituting $\nabla L(\mathbf{f}_t)$ with the corresponding expression gives the following algorithm:

1. Compute the error $\mathbf{r}_t = A\mathbf{f}_t - \mathbf{b}$.
2. Compute the step $\mathbf{d}_{t+1} = \mu\mathbf{d}_t - 2\gamma A^T\mathbf{r}_t$.
3. Update $\mathbf{f}_{t+1} = \mathbf{f}_t + \mathbf{d}_{t+1}$.
4. Repeat until $||\mathbf{r}_t|| < \varepsilon$.

4.7 Nesterov accelerated gradient

There are different implementations of this algorithm. One of them (discussed in Botev et al. [2017]) shows its connection to the gradient descent with momentum:

1. Compute the step $\mathbf{d}_{t+1} = \mu_t\mathbf{d}_t - \gamma\nabla L(\mathbf{f}_t + \mu_t\mathbf{d}_t)$.
2. Update $\mathbf{f}_{t+1} = \mathbf{f}_t + \mathbf{d}_{t+1}$.
3. Repeat until the stop criterion is met.

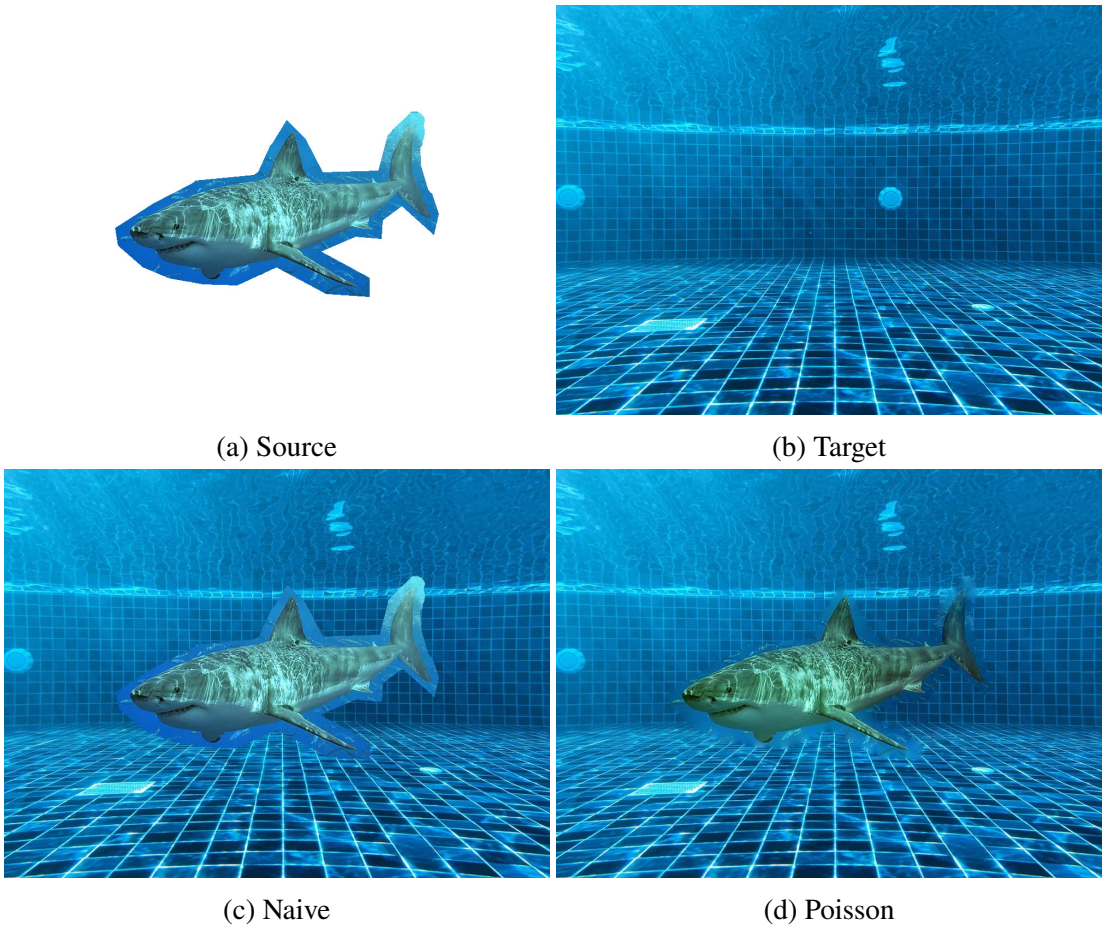
In application to our problem:

1. Compute the step $\mathbf{d}_{t+1} = \mu_t\mathbf{d}_t - 2\gamma A^T(A(\mathbf{f}_t + \mu_t\mathbf{d}_t) - \mathbf{b})$.
2. Update $\mathbf{f}_{t+1} = \mathbf{f}_t + \mathbf{d}_{t+1}$.
3. Repeat until the stop criterion is met.

Here μ_t is a schedule that is different for every iteration and computed as $\mu_t = 1 - 3/(5 + t)$ and γ is a fixed parameter.

5 Comparison of numerical methods

In this section we will see how different numerical methods perform compared to each other. First, let us discuss the notion of convergence. It is said that numerical method has a geometric convergence if $\|\mathbf{f}_{t+1} - \mathbf{f}^*\| \leq \lambda \|\mathbf{f}_t - \mathbf{f}^*\|$, where $\mathbf{f}^*$ is the true solution and $0 < \lambda < 1$. Let $e_t = \|\mathbf{f}_t - \mathbf{f}^*\|$. On practice, it is more useful to take a logarithm of both sides of this inequality: $\log e_t \leq \log e_{t-1} + \lambda$. So, if in logarithmic scale the plot for $e(t)$ is close to a line, then we have a geometric convergence. There are other types of convergence, for example, quadratic: $e_t \leq e_{n-1}^2$.



(a) Source (b) Target

(c) Naive (d) Poisson

Figure 5.1: Basic example of the Poisson editing used for the analysis of numerical methods.

Before moving forward let us discuss some programming details. As we already have an easy mathematical rule for multiplying matrix A by any vector (as shown in formula 3.3), we can treat this matrix more like an operator in our code. That means that instead of constructing this matrix explicitly we can just define a function that computes the product.

For all results presented in this report initialization with zeroes was used. To further speed up the convergence process it is possible to compute a more optimal initialization to start with a smaller energy value. It is also worth mentioning that in this report (and in the original paper) we deal with unconstrained optimization. In some cases it can lead to inappropriate values for intensity in the solution (in our case, greater than 1 or less than 0). To avoid that clipping was used.

Let us consider an example shown in the Figure 5.1. For this example we study every method with following parameters:

1. Gradient descent (GD): $\gamma = 0.02$.
2. Modified Richardson iteration (RI) $\gamma = 0.2$.
3. Gauss-Seidel method (GS): no parameters.
4. Successive over-relaxation (SOR): $\gamma = 1.9$.
5. Conjugate gradient method (CG): no parameters.
6. Gradient descent with momentum (GDM): $\gamma = 0.02$, $\beta = 0.99$.
7. Nesterov accelerated gradient (NAG): $\gamma = 0.006$.

This parameters were picked based on practical results so the speed of convergence can be improved by computing optimal parameters for each method. For the error e_t we will consider two cases: $e_t = \|\mathbf{f}_{t+1} - \mathbf{f}^*\|_2$ and $e_t = |E_t - E^*|$, where $E = \sum_{\langle p,q\rangle \cap \Omega \neq \emptyset} (f_p - f_q - v_{pq})^2$. E^* and f^* are estimated by computing a numerical solution with great precision using the fastest method (conjugate gradient).

Figures 5.3, 5.4, 5.5 show plots for all methods in one picture. From them we can already conclude that conjugate gradient method converges much faster than other methods. Figure 5.6 gives more detailed look at the convergence of each method for $e_t = \|\mathbf{f}_{t+1} - \mathbf{f}^*\|_2$ with logarithmic axis for the error.

Table 1: Comparison of methods for different region sizes, $\varepsilon = 0.01$.

Size	GD		RI		GS		SOR		CG		GDM		NAG	
	t, sec	Iter.	t, sec	Iter.	t, sec	Iter.	t, sec	Iter	t, sec	Iter	t, sec	Iter	t, sec	Iter.
481416	247.62	37724	0.92	148	1.21	59	1.80	106	0.84	84	7.25	892	10.5	874
281786	149.31	40648	0.69	163	0.71	68	0.72	106	0.53	98	4.74	962	6.34	872
158809	52.32	30028	0.41	144	0.47	60	0.53	108	0.39	92	2.38	914	2.84	778
42571	7.37	27777	0.24	134	0.28	58	0.32	116	0.24	114	0.59	944	0.67	737

Table 2: Comparison of methods for different region sizes and images, $\varepsilon = 0.001$.

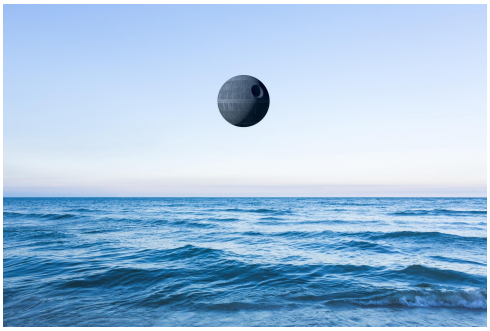
Size		RI		GS		SOR		CG		GDM		NAG	
		t, sec	Iter.	t, sec	Iter.	t, sec	Iter	t, sec	Iter	t, sec	Iter	t, sec	Iter.
Ex.1	481416	8.79	998	4.79	389	2.49	151	11.59	646	238.19	17203	105.12	5935
	281786	2.88	1011	1.98	390	0.92	153	3.00	548	83.28	17548	41.28	5959
	158809	1.28	964	1.04	372	0.57	160	1.08	416	37.03	15932	19.71	5850
	42571	0.36	955	0.4	371	0.29	155	0.31	273	6.22	16881	4.17	6902
Ex.2	99792	1.33	1477	1.01	600	0.40	141	0.42	275	65.43	47996	23.19	12135

Let us also take a look at how region sizes and convergence are connected. For the first experiment same pictures were used, but source image was presented in 4 different sizes (see Figure 5.2). Table 1 shows results for $\varepsilon = 0.01$. From experimental results it was concluded that after reaching error of $\varepsilon = 0.0001$ changes in the numerical solution become almost invisible to human eye. In most cases setting $\varepsilon = 0.001$ is enough for satisfactory results. So we need another experiment with a smaller value of ε. In the second experiment we won't be studying gradient descent as it converges too slow, but instead add another example for comparison (Ex.2 corresponds to the Figure 5.1). Results are presented in Table 2

Interestingly enough, smaller size of the region doesn't necessarily mean smaller number of iterations even when we use same object and background, but it does help to shorten the time it takes for each iteration to be computed. Another interesting conclusion can be made from the second table: different methods can show better performance in different circumstances (notice how GDM and GS are much slower in the second example, but SOR and CD produce the opposite results). Combining results from tables with convergence plots we can conclude that two best methods are SOR and CG. SOR seems to be faster in some cases, but CG provides higher precision in less time as is seen from plots.

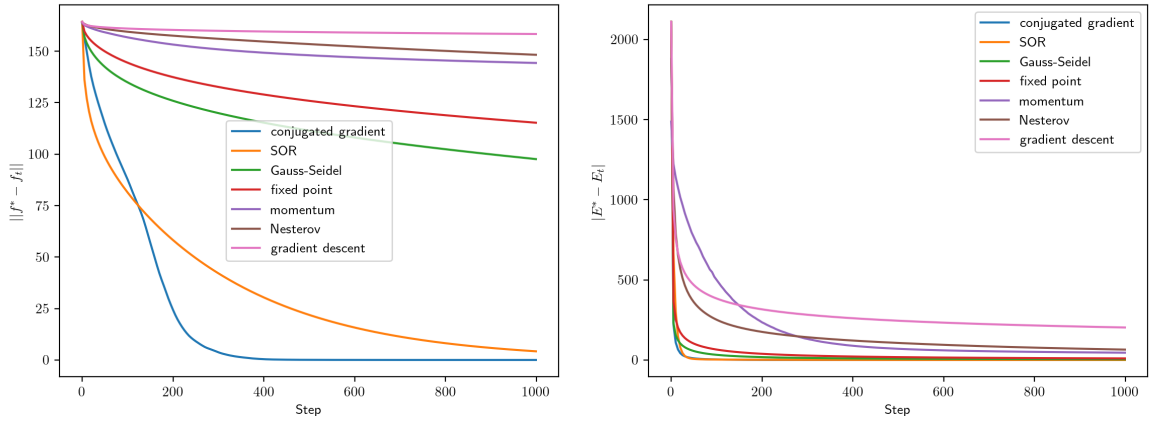
(a) Naive (b) Result

(c) Scaled 1 (d) Scaled 2



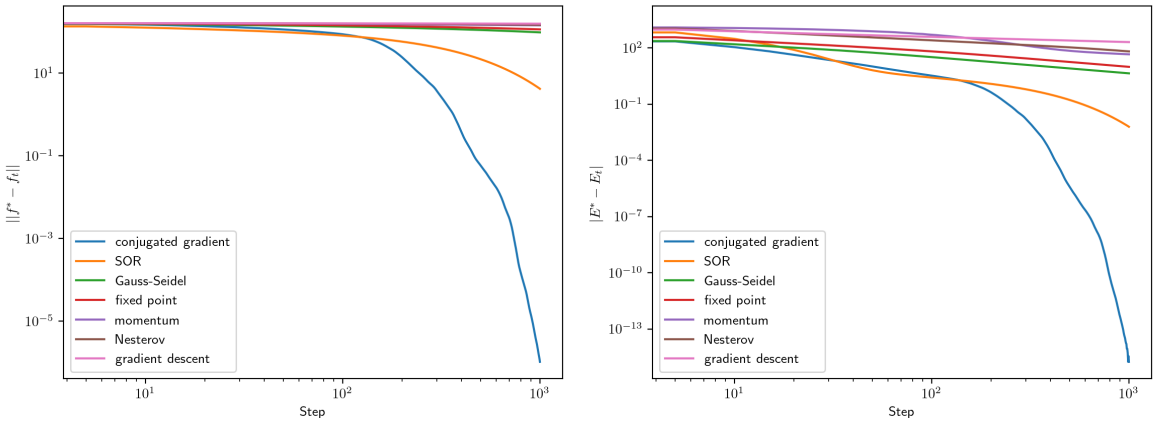
(e) Scaled 3

Figure 5.2: Example used to study the convergence speed for different sizes of the selected region.



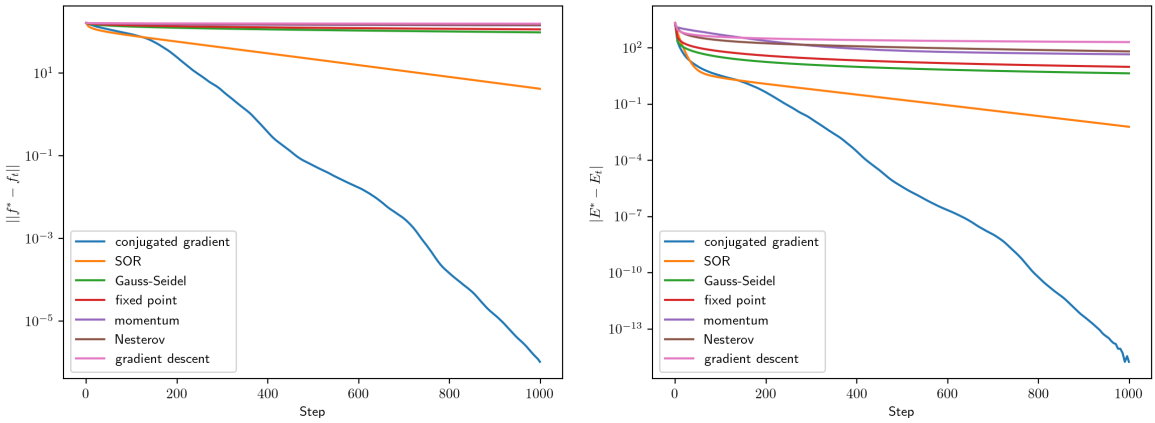
(a) Image (b) Energy

Figure 5.3: Convergence plots in linear scale: (a) convergence for the result f_t; (b) convergence for the energy.



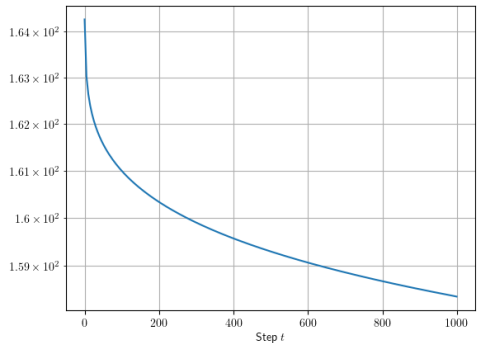
(a) Image (b) Energy

Figure 5.4: Convergence plots in log-log scale (logarithmic ordinate and linear abscissa): (a) convergence for the result f_t; (b) convergence for the energy.

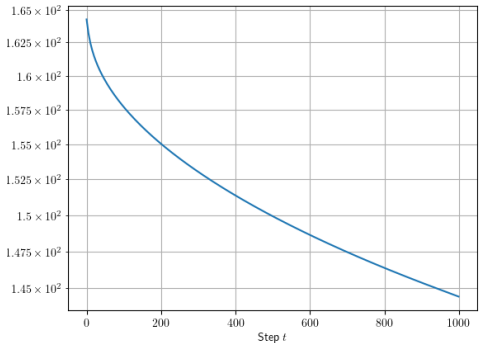


(a) Image (b) Energy

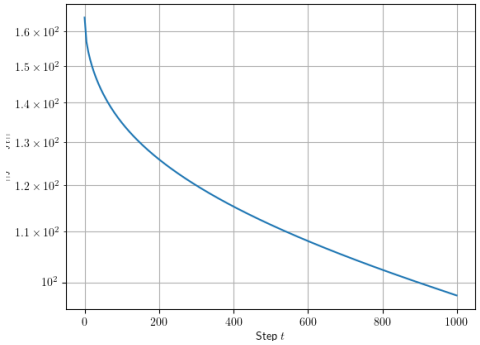
Figure 5.5: Convergence plots in linear-log scale (logarithmic ordinate and logarithmic abscissa): (a) convergence for the result f_t; (b) convergence for the energy.



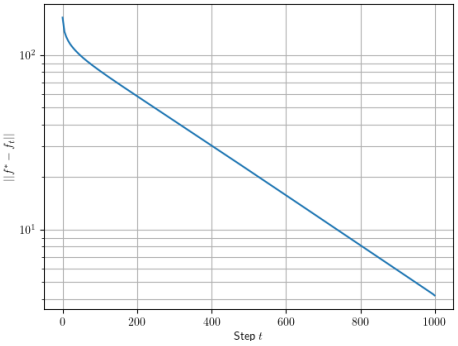
(a) Gradient descent



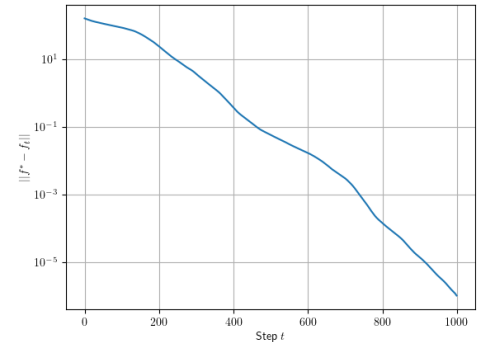
(b) Richardson



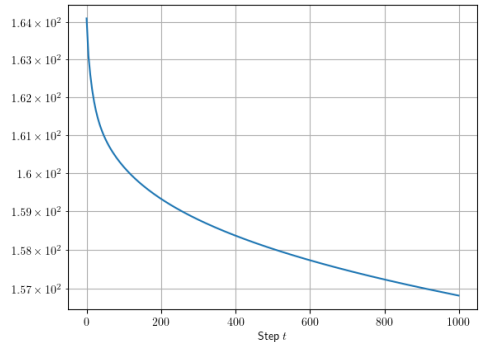
(c) Gauss-Seidel



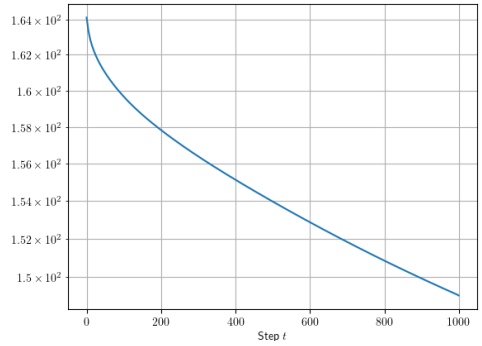
(d) SOR



(e) Conjugate gradient



(f) Momentum



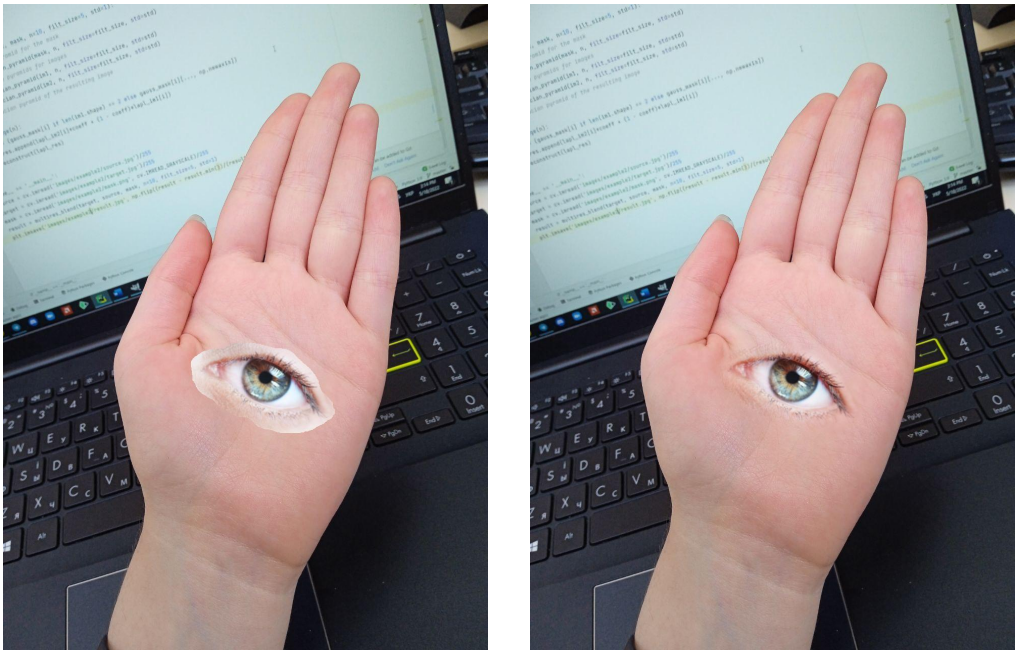
(g) Nesterov

Figure 5.6: Convergence plots in linear-log scale (logarithmic ordinate and linear abscissa) for every method. We can see that plots for successive over-relaxation and conjugate gradient methods are quite close to a line. This means that they have geometric convergence.

6 Applications of Poisson editing

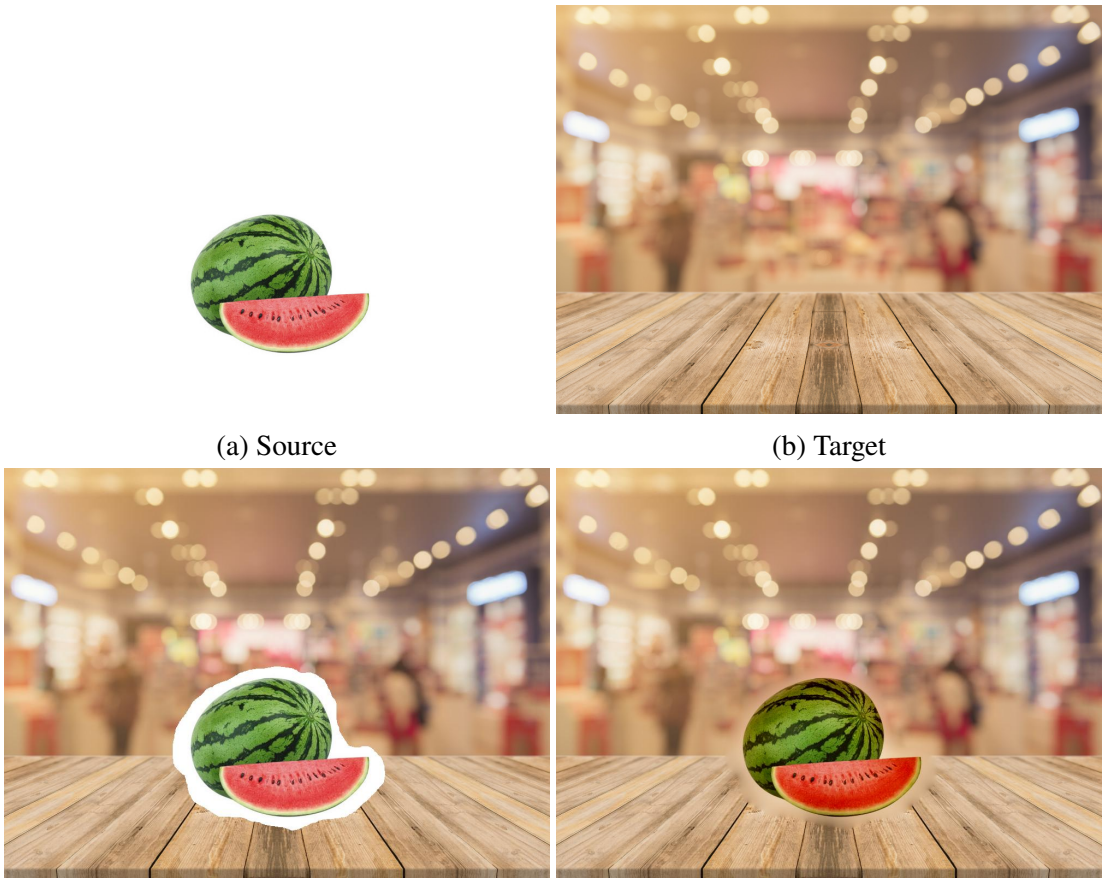
Now it is finally time to see actual results obtained with Poisson editing. We have already seen some examples in the Introduction and Section 5. Figure 5.1 demonstrated significant improvement compared to the naive method. However, one could argue that colours of the shark became less "natural" as it is now closer to green than to blue. Other figures in this section demonstrate different levels of performance and applications: satisfactory results for basic blending (Figure 6.1 and Figure 6.2), successful examples of gradient mixing (Figure 6.4 and Figure 6.5), an interesting example of texture mixing (Figure 6.6), texture replacing (Figure 6.3) and also some examples of poor performance (Figure 6.7 and Figure 6.8). The last two figures allow to conclude that Poisson editing can run into problems when illumination or colours in the source and target images differ too much.

Otherwise, it demonstrates a decent performance in a wide variety of applications and settings.



(a) Naive (b) Poisson

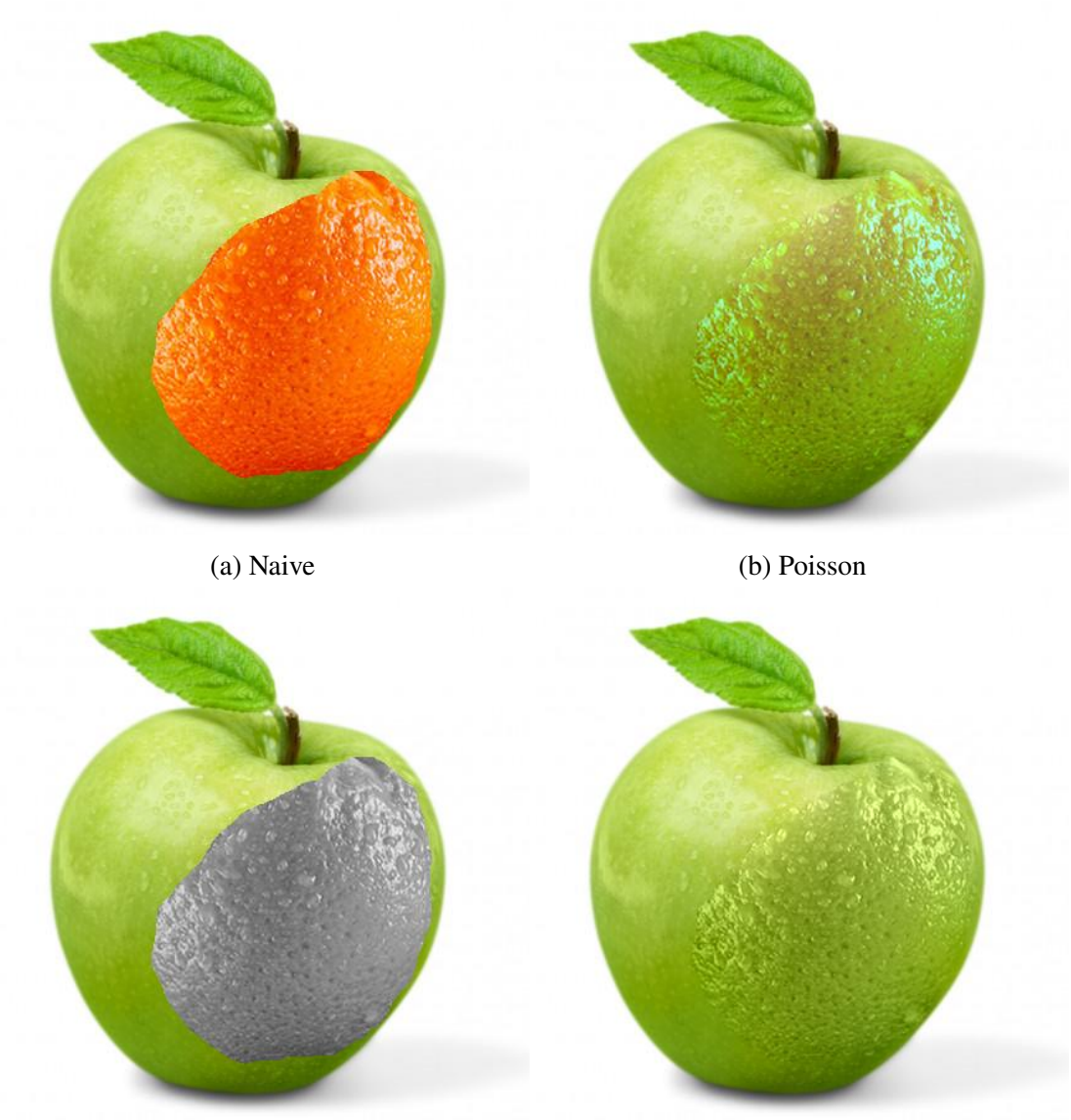
Figure 6.1: Example 1. This example was shown before in Section 2.3. We can see that here Poisson method performs really good. However, it is hard to say that its performance is significantly better compared to the multiresolution blending.



(a) Source (b) Target

(c) Naive (d) Poisson

Figure 6.2: Example 2. Colours in the result are well blended, however we can observe a blurry halo around the watermelon.



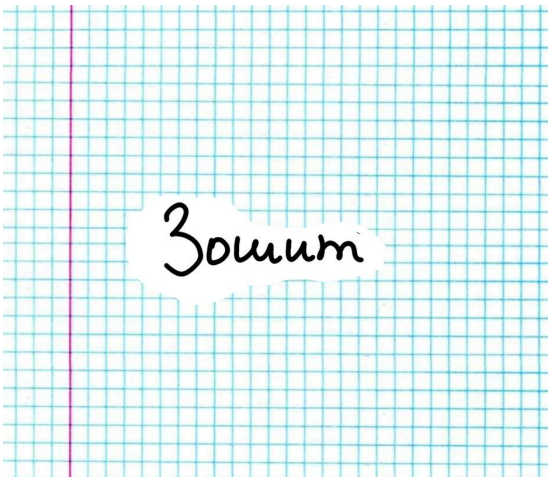
(a) Naive

(b) Poisson

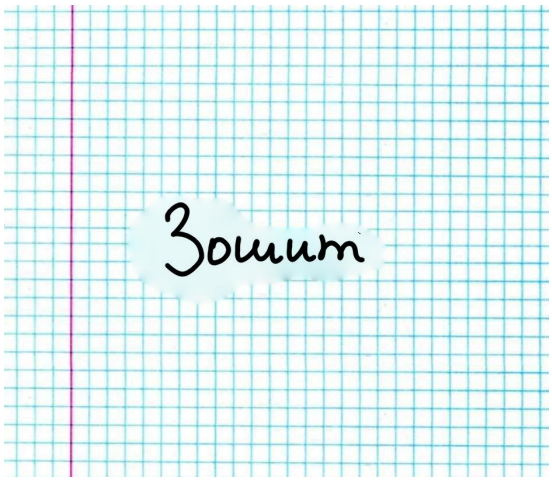
(c) Naive with greyscale

(d) Poisson with greyscale

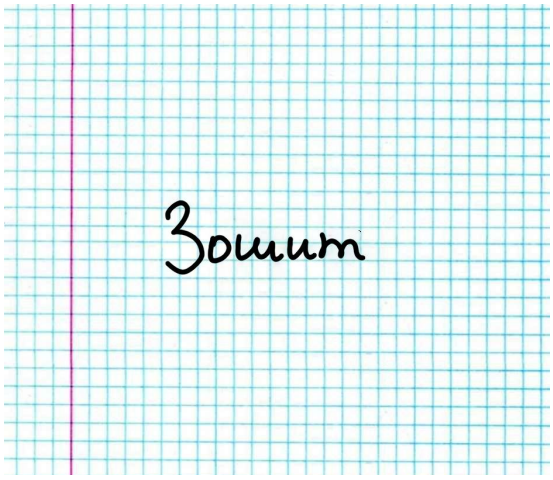
Figure 6.3: Example 3: texture swapping. Top row shows the result of the Poisson blending applied to the coloured image of the orange texture. It can be seen that a slight hint of orange colour is left in the result. It can be solved by converting the source image to greyscale before applying the algorithm as shown in the bottom row.



(a) Naive

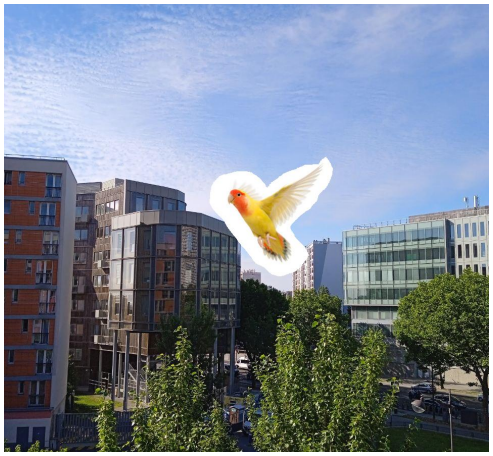


(b) Poisson

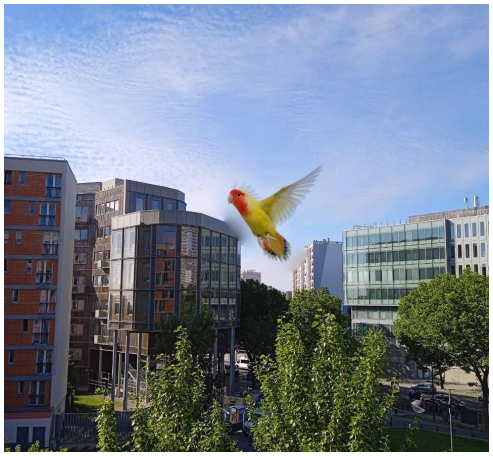


(c) Poisson with mixing

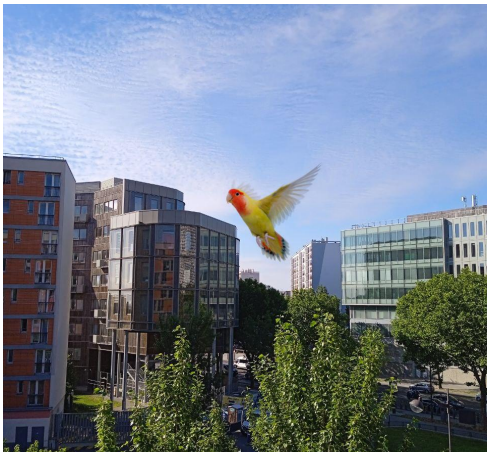
Figure 6.4: Example 4: cloning objects with holes on top of other objects in background. Here the "classic" guidance field (b) is obviously not enough: white background of the text causes background lines to become blurred. Replacing it with mixing gradients field gives close to perfect results as seen on image (c).



(a) Naive



(b) Poisson



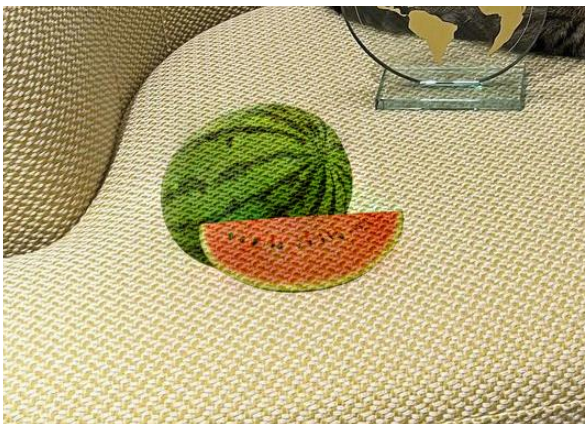
(c) Poisson with mixing

Figure 6.5: Example 5: cloned region is overlapping with background objects. Basically, we have the same situation as in previous example. Overall good results, the only possible downside is the colour of the parrot in (c): it seems to be a little too dark. However, this statement is subjective.

(a) Naive

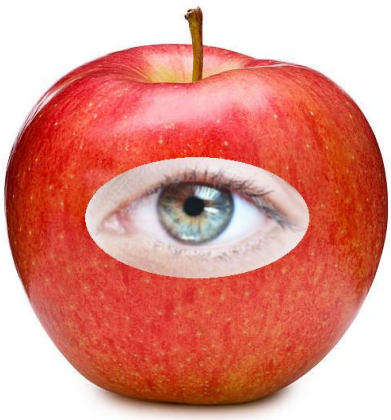
(b) Poisson

(c) Poisson with mixing

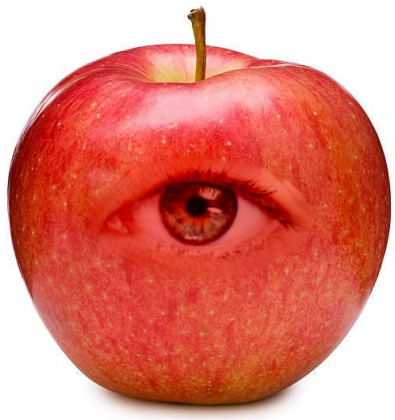


(d) Zoomed in

Figure 6.6: Example 6: texture blending. In image (b) we can observe a watermelon's background blurring the pattern on the sofa. Applying mixed gradients does not resolve this problem, but presents another interesting application. Image (d) gives an impression of the watermelon being "printed" over the textured sofa.



(a) Naive



(b) Poisson

Figure 6.7: Example 7: failure case. Attempting to clone an eye over apple results in "unnatural" (or rather not believable) colours for the eye: red whites and loss of colour of the iris.

(a) Naive (b) Poisson

Figure 6.8: Example 8: another failure case. Attempting to insert my white cat into a bright snowy landscape results in the selected region being too bright. This is most probably caused by the difference in the illumination of these two photos.

7 Conclusions

Multiresolution blending and Poisson editing both have their upsides and downsides. The first method is easy to implement and fast to use. However, it is not a very flexible instrument: it is limited to blending purposes, and even for them, it is often not enough. Now, the Poisson method is customizable thanks to guidance fields and in general tends to produce better results. But it still has weak spots like differences in illumination. Besides, for bigger regions it requires more computational resources and time than multiresolution blending. However, there are ways to improve the convergence speed that were not explored in this work: finding optimal initialization, computing optimal parameters for numerical methods etc. It is also worth mentioning that the conjugate gradient descent and the successive over-relaxation proved to be the best numerical methods for Poisson editing among those discussed. The most important source code can be found in Appendix A; the whole project will be published as a Github repository.

On a more personal side, this internship was a valuable experience and helped me revise some of my knowledge and discover new topics, both in mathematics and programming. It was a pleasure working in the lab, and I am grateful to everyone in MAP5 and UFR Math-Info who helped me through these months.

References

Aleksandar Botev, Guy Lever, and David Barber. Nesterov's accelerated gradient and momentum as approximations to regularised update descent. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 1899–1903, 2017.

Peter J. Burt and Edward H. Adelson. A multiresolution spline with application to image mosaics. *ACM Trans. Graph.*, 2(4):217–236, 1983.

Imre Bárány and Jozsef Solymosi. Gershgorin disks for multiple eigenvalues of non-negative matrices. *A Journey through Discrete Mathematics: A Tribute to Jiri Matousek*, pages 123–133, 2017.

K. R. James and W. Riha. Convergence criteria for successive overrelaxation. *SIAM Journal on Numerical Analysis*, 12(2):137–143, 1975.

Tony Lindeberg. Scale-space theory: A basic tool for analysing structures at different scales. *Journal of Applied Statistics*, 21:224–270, 1994.

David Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60:91–110, 2004.

Patrick Pérez, Michel Gangnet, and Andrew Blake. Poisson image editing. *ACM Trans. Graph.*, 22(3):313–318, 2003.

Lewis Fry Richardson and Richard Tetley Glazebrook. Ix. the approximate arithmetical solution by finite differences of physical problems involving differential equations, with an application to the stresses in a masonry dam. *Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character*, 210(459-470):307–357, 1911.

A Appendix: code

multiresolution.py

```
import numpy as np
import cv2 as cv
import matplotlib.pyplot as plt
import os


def clear_dir(path):
    if os.path.exists(path):
        for file_name in os.listdir(path):
            file = path + file_name
            if os.path.isfile(file):
                os.remove(file)
    else:
        os.makedirs(path)


def gaussian_pyramid(image, n, num, is_mask, filt_size=5, std=1):
    pyramid = [image]
    for i in range(1, n):
        size = (round(pyramid[i - 1].shape[1] * 0.5),
                round(pyramid[i - 1].shape[0] * 0.5))
        pyramid.append(cv.resize(
            cv.GaussianBlur(pyramid[i - 1], (filt_size, filt_size), std, std),
            size))
        if is_mask:
            plt.imsave(f'images/example{num}/gaussian␣pyramid/{i}.jpg',
                       (pyramid[i - 1] - pyramid[i - 1].min()) / (
                                   pyramid[i - 1].max() - pyramid[i - 1].min()),
                       cmap='gray')
    return pyramid


def laplacian_pyramid(image, n, num, filt_size=5, std=1):
    gauss_pyramid = gaussian_pyramid(image, n, num, False, filt_size=filt_size,
                                     std=std)
    pyramid = [gauss_pyramid[-1]]
    for i in range(1, n):
        pyramid.append(
            -cv.resize(gauss_pyramid[-i],
                       gauss_pyramid[-i - 1][..., 0].shape[::-1],
                       interpolation=cv.INTER_CUBIC) +
            gauss_pyramid[-i - 1])
        plt.imsave(f'images/example{num}/laplacian␣pyramid/{n␣-␣i}.jpg',
                   np.flip((pyramid[i] - pyramid[i].min()) / (
                               pyramid[i].max() - pyramid[i].min()),
                           axis=2))
    return list(reversed(pyramid))


# sum up all levels of a pyramid
def reconstruct(pyramid, num):
    size = (pyramid[0].shape[1], pyramid[0].shape[0])
```

```
53      result = np.zeros(pyramid[0].shape)
54      for i in range(len(pyramid)):
55          result += cv.resize(pyramid[-i - 1], size, interpolation=cv.INTER_CUBIC)
56          plt.imsave(f'images/example{num}/progress/{i}.jpg',
57                     np.flip(
58                         (result - result.min()) / (result.max() - result.min()),
59                         axis=2))
60      return result
61
62
63  def multires_blend(target, source, mask, num, n=10, filt_size=5, std=1,
64                     std_mask=100, filt_size_mask=15):
65      # compute gaussian pyramid for the mask
66      gauss_mask = gaussian_pyramid(mask, n, num, True, filt_size=filt_size_mask,
67                                    std=std_mask)
68      # compute laplacian pyramids for images
69      lapl_source = laplacian_pyramid(source, n, num, filt_size=filt_size,
70                                      std=std)
71      lapl_target = laplacian_pyramid(target, n, num, filt_size=filt_size,
72                                      std=std)
73      # compute laplacian pyramid of the resulting image
74      lapl_res = []
75      for i in range(n):
76          coeff = (gauss_mask[i] if len(target.shape) == 2 else gauss_mask[i][
77              ..., np.newaxis])
78          lapl_res.append(lapl_source[i] * coeff + (1 - coeff) * lapl_target[i])
79      return reconstruct(lapl_res, num)
80
81
82  def grad_mask(mask, filt_size):
83      std = (filt_size - 1) / 3
84      mask = cv.GaussianBlur(mask, (filt_size, filt_size), std, std)
85      return mask
86
87
88  def example(num, method, use_grad_mask=False, mask_filt_size=295, **kwargs):
89      clear_dir(f'images/example{num}/progress/')
90      clear_dir(f'images/example{num}/gaussian␣pyramid/')
91      clear_dir(f'images/example{num}/laplacian␣pyramid/')
92      source = cv.imread(f'images/example{num}/source.jpg') / 255
93      target = cv.imread(f'images/example{num}/target.jpg') / 255
94      if use_grad_mask:
95          mask = grad_mask(
96              cv.imread(f'images/example{num}/mask.png', cv.IMREAD_GRAYSCALE),
97              mask_filt_size) / 255
98          plt.imsave(f'images/example{num}/mask1.png', mask, cmap='gray')
99      else:
100         mask = cv.imread(f'images/example{num}/mask.png',
101                          cv.IMREAD_GRAYSCALE) / 255
102     result = method(target, source, mask, num, **kwargs)
103     plt.imsave(f'images/example{num}/result.jpg',
104                np.flip(np.clip(result, 0, 1), axis=2))
105
106
107 if __name__ == '__main__':
108     example(30, multires_blend, use_grad_mask=False, n=5, filt_size=15, std=10)
```

poisson.py

```
import numpy as np
import cv2 as cv
import matplotlib.pyplot as plt
from time import time
import iterative
from copy import deepcopy


def find_border(mask):
    border = cv.filter2D(mask, cv.CV_64F,
                         np.array([[0, 1, 0], [1, 0, 1], [0, 1, 0]]))
    border = np.where(border > 0, 1.0, 0)
    border = border - mask
    return border


def shift(f, n, axis):
    f = np.roll(f, n, axis=axis)
    if n == -1 and axis == 0:
        f[-1, :] = 0
    elif n == 1 and axis == 0:
        f[0, :] = 0
    elif n == -1 and axis == 1:
        f[:, -1] = 0
    elif n == 1 and axis == 1:
        f[:, 1] = 0
    return f


def compute_Np(shape):
    Np = np.full(shape, 4)
    Np[0, :] = 3
    Np[-1, :] = 3
    Np[:, 0] = 3
    Np[:, -1] = 3
    Np[[0, -1, 0, -1], [0, 0, -1, -1]] = 2
    return Np


def neighbours(mask):
    idx = np.nonzero(mask)
    mask_ind = np.zeros(mask.shape, dtype=int)
    mask_ind[idx] = np.arange(1, idx[0].size + 1, 1)
    bottom = shift(mask_ind, -1, 0)[idx]
    top = shift(mask_ind, 1, 0)[idx]
    left = shift(mask_ind, 1, 1)[idx]
    right = shift(mask_ind, -1, 1)[idx]
    return top, left, right, bottom


def shift_ind(f):
    left_x, left_y = np.arange(0, f.shape[0], 1), np.append(
        np.arange(1, f.shape[1], 1), f.shape[1] - 1)
    right_x, right_y = np.arange(0, f.shape[0], 1), np.insert(
```

```
        np.arange(0, f.shape[1] - 1, 1), 0, 0)
    bottom_x, bottom_y = np.append(np.arange(1, f.shape[0], 1),
                                   f.shape[0] - 1), np.arange(0, f.shape[1], 1)
    top_x, top_y = np.insert(np.arange(0, f.shape[0] - 1, 1), 0, 0), np.arange(
        0, f.shape[1], 1)
    return (np.repeat(left_x, f.shape[1]), np.tile(left_y, f.shape[0]),
            np.repeat(right_x, f.shape[1]), np.tile(right_y, f.shape[0]),
            np.repeat(bottom_x, f.shape[1]), np.tile(bottom_y, f.shape[0]),
            np.repeat(top_x, f.shape[1]), np.tile(top_y, f.shape[0]))


def poisson_blending(source, target, mask, n, eps=0.001, gamma=0.1, beta=0.9,
                     channel=1, save_step=100,
                     guidance='classic',
                     save_history=False, method='fixed-point', ref=None,
                     n_steps=0):
    ind_mask = np.nonzero(mask)
    Np = compute_Np(mask.shape)[ind_mask]
    result = deepcopy(target)
    border_mask = find_border(mask)
    target_border_masked = target * border_mask
    left_x, left_y, right_x, right_y, bottom_x, bottom_y, top_x, top_y = \
        shift_ind(
        mask)
    b1 = (target_border_masked[left_x, left_y] + target_border_masked[
        right_x, right_y] +
          target_border_masked[top_x, top_y] + target_border_masked[
              bottom_x, bottom_y]).reshape(mask.shape)[
        ind_mask]
    b2 = (source[left_x, left_y] + source[right_x, right_y] +
          source[top_x, top_y] + source[bottom_x, bottom_y]).reshape(
        mask.shape)[ind_mask]

    if guidance == 'classic':
        b = - b2 + b1 + Np * source[ind_mask]
    elif guidance == 'mix':
        mix_top = abs((target - target[top_x, top_y].reshape(mask.shape))[
                          ind_mask]) > 2 * abs(
            (source - source[top_x, top_y].reshape(mask.shape))[ind_mask])
        mix_bottom = abs(
            (target - target[bottom_x, bottom_y].reshape(mask.shape))[
                ind_mask]) > 2 * abs(
            (source - source[bottom_x, bottom_y].reshape(mask.shape))[ind_mask])
        mix_left = abs((target - target[left_x, left_y].reshape(mask.shape))[
                           ind_mask]) > 2 * abs(
            (source - source[left_x, left_y].reshape(mask.shape))[ind_mask])
        mix_right = abs((target - target[right_x, right_y].reshape(mask.shape))[
                            ind_mask]) > 2 * abs(
            (source - source[right_x, right_y].reshape(mask.shape))[ind_mask])
        b = b1 + (np.where(mix_top,
                           (target - target[top_x, top_y].reshape(mask.shape))[
                               ind_mask],
                           (source - source[top_x, top_y].reshape(mask.shape))[
                               ind_mask])
                  + np.where(mix_bottom, (
                           target - target[bottom_x, bottom_y].reshape(
```

```
                        mask.shape))[ind_mask],
                             (source - source[bottom_x, bottom_y].reshape(
                                 mask.shape))[ind_mask])
                    + np.where(mix_left, (target - target[left_x, left_y].reshape(
                      mask.shape))[ind_mask],
                             (source - source[left_x, left_y].reshape(
                                 mask.shape))[ind_mask])
                    + np.where(mix_right, (
                              target - target[right_x, right_y].reshape(
                        mask.shape))[ind_mask],
                             (source - source[right_x, right_y].reshape(
                                 mask.shape))[ind_mask]))
        mixing = np.zeros(mask.shape)
        mixing[
            ind_mask] = 1.0 * mix_top + 1.0 * mix_bottom + 1.0 * mix_right + \
                        1.0 * mix_left
        # plt.imshow(mixing)
        # plt.show()

    else:
        raise ValueError(
            'Wrong guidance field type! Available types \'classic\', \'mix\'')

    idx = neighbours(mask)
    N = b.size
    f = np.zeros(N + 1)
    # f[:-1] = source[ind_mask]
    delta = np.zeros(N + 1)
    delta[:-1] = iterative.iter_A(f, Np, idx)[:-1] - b
    # clear_dir(f'images/example{n}/progress/')
    if method == 'fixed-point':
        results, j = iterative.fixed_point(f, b, delta, eps, Np, idx, gamma,
                                           channel, target, source, mask, n,
                                           ref=ref,
                                           save_history=save_history,
                                           n_steps=n_steps)
    elif method == 'gauss-seidel':
        results, j = iterative.sor(f, b, delta, eps, Np, idx, channel, 1,
                                   target, source, mask, n, ref=ref,
                                   save_history=save_history, n_steps=n_steps)
    elif method == 'sor':
        results, j = iterative.sor(f, b, delta, eps, Np, idx, channel, gamma,
                                   target, source, mask, n, ref=ref,
                                   save_history=save_history, n_steps=n_steps)
    elif method == 'gradient-descent':
        results, j = iterative.gradient_descent(f, b, delta, eps, Np, idx,
                                                gamma, channel, target, source,
                                                mask, n,
                                                ref=ref,
                                                save_history=save_history,
                                                n_steps=n_steps)
    elif method == 'conjugated-gradient':
        results, j = iterative.conjugated_gradient(f, b, eps, Np, idx, channel,
                                                   target, source, mask, n,
                                                   ref=ref,
                                                   save_history=save_history,
```

```
                                                    n_steps=n_steps)
    elif method == 'gradient-descent-momentum':
        results, j = iterative.gradient_descent_momentum(f, b, delta, eps, Np,
                                                         idx, gamma, beta,
                                                         channel, target,
                                                         source,
                                                         mask, n, ref=ref,
                                                         save_history=save_history,
                                                         n_steps=n_steps)
    elif method == 'nag':
        results, j = iterative.nesterov_accelerated_grad(f, b, delta, eps, Np,
                                                         idx, gamma, beta,
                                                         channel, target,
                                                         source,
                                                         mask, n, ref=ref,
                                                         save_history=save_history,
                                                         n_steps=n_steps)
    else:
        raise ValueError(
            'Wrong method name! Available methods: \'fixed-point\', '
            '\'gauss-seidel\', \'sor\', \'conjugated-gradient\', '
            '\'gradient-descent\', \'gradient-descent-momentum\'')
    if save_history:
        result[ind_mask] = results[0][:-1]
        return result, results[1], results[2], results[3]
    else:
        result[ind_mask] = results[:-1]
        return result, j
```

iterative.py

```
import numpy as np
from numba import njit
from copy import deepcopy
import cv2 as cv


def roll_energy(f):
    f_rolled_vert = np.roll(f, -1, 1)
    f_rolled_vert[:, -1] = f_rolled_vert[:, -2]
    f_rolled_hor = np.roll(f, -1, 0)
    f_rolled_hor[-1, :] = f_rolled_hor[-2, :]
    return f_rolled_vert, f_rolled_hor


def compute_energy(f, source, mask):
    right_neighbour = cv.filter2D(mask, cv.CV_64F,
                                  np.array([[0, 0, 0], [0, 1, 1], [0, 0, 0]]),
                                  borderType=cv.BORDER_REFLECT)
    right_neighbour = np.where(right_neighbour > 0, 1., 0.)
    bottom_neighbour = cv.filter2D(mask, cv.CV_64F,
                                   np.array([[0, 0, 0], [0, 1, 0], [0, 1, 0]]),
                                   borderType=cv.BORDER_REFLECT)
    bottom_neighbour = np.where(bottom_neighbour > 0, 1., 0.)
    f_rolled_vert, f_rolled_hor = roll_energy(f)
    source_rolled_vert, source_rolled_hor = roll_energy(source)
```

```
    res = np.sum((
                          -f_rolled_vert * right_neighbour + right_neighbour *
                          f - source * right_neighbour + source_rolled_vert *
                          right_neighbour) ** 2 +
                 (
                          -f_rolled_hor * bottom_neighbour + bottom_neighbour
                          * f - source * bottom_neighbour + source_rolled_hor
                          * bottom_neighbour) ** 2)
    return res


@njit
def iter_A(f, Np, ind):
    result = np.zeros(f.shape)
    for i in range(f.shape[0] - 1):
        result[i] = Np[i] * f[i] - (
                     f[ind[0][i] - 1] + f[ind[1][i] - 1] + f[ind[2][i] - 1] + f[
                ind[3][i] - 1])
    return result


def A(f, Np, ind):
    result = np.zeros(f.shape)
    result[:-1] = Np * f[:-1] - (
                f[ind[0] - 1] + f[ind[1] - 1] + f[ind[2] - 1] + f[ind[3] - 1])
    return result


def update_history(temp_result, ind_mask, f, source, mask, ref_masked,
                   energy_arr, imgs):
    temp_result[ind_mask] = f[:-1]
    energy = compute_energy(temp_result, source, mask)
    norm = np.sum((f[:-1] - ref_masked) ** 2) ** 0.5
    energy_arr.append(energy)
    imgs.append(norm)


def fixed_point(f, b, delta, eps, Np, idx, gamma, channel, target, source, mask,
                n, ref=None, save_history=False,
                n_steps=0):
    j = 0
    ind_mask = np.nonzero(mask)
    if save_history:
        energy_arr = []
        nums = []
        imgs = []
        ref_masked = deepcopy(ref[ind_mask])
        temp_result = deepcopy(target)
    if not n_steps:
        while abs(delta).max() > eps:
            if save_history and j % 5 == 0:
                update_history(temp_result, ind_mask, f, source, mask,
                               ref_masked, energy_arr, imgs)
                nums.append(j)
            f = f - gamma * delta
            # print(f"channel{channel}, step{j}, del{abs(delta).max()}")
```

```
            delta[:-1] = iter_A(f, Np, idx)[:-1] - b
            j += 1
    else:
        for j in range(n_steps):
            if save_history and j % 5 == 0:
                update_history(temp_result, ind_mask, f, source, mask,
                               ref_masked, energy_arr, imgs)
                nums.append(j)
            f = f - gamma * delta
            # print(f"channel{channel}, step{j}, del{abs(delta).max()}")
            delta[:-1] = iter_A(f, Np, idx)[:-1] - b
    if save_history:
        update_history(temp_result, ind_mask, f, source, mask, ref_masked,
                       energy_arr, imgs)
        nums.append(j)
        return f, np.array(nums), np.array(energy_arr), np.array(imgs)
    else:
        return f, j


def gradient_descent_step(f, gamma, delta, Np, idx, channel, b, j):
    f = f - gamma * iter_A(delta, Np, idx)
    # print(f"channel{channel}, step{j}, del{abs(delta).max()}")
    delta[:-1] = iter_A(f, Np, idx)[:-1] - b
    return f, delta


def gradient_descent(f, b, delta, eps, Np, idx, gamma, channel, target, source,
                     mask, n, ref=None, save_history=False,
                     n_steps=0):
    j = 0
    ind_mask = np.nonzero(mask)
    if save_history:
        energy_arr = []
        nums = []
        imgs = []
        ref_masked = deepcopy(ref[ind_mask])
        temp_result = deepcopy(target)
    if not n_steps:
        while abs(delta).max() > eps:
            if save_history and j % 5 == 0:
                update_history(temp_result, ind_mask, f, source, mask,
                               ref_masked, energy_arr, imgs)
                nums.append(j)
            f, delta = gradient_descent_step(f, gamma, delta, Np, idx, channel,
                                             b, j)
            j += 1
    else:
        for j in range(n_steps):
            if save_history and j % 5 == 0:
                update_history(temp_result, ind_mask, f, source, mask,
                               ref_masked, energy_arr, imgs)
                nums.append(j)
            f, delta = gradient_descent_step(f, gamma, delta, Np, idx, channel,
                                             b, j)

```

```
    if save_history:
        update_history(temp_result, ind_mask, f, source, mask, ref_masked,
                       energy_arr, imgs)
        nums.append(j)
        return f, np.array(nums), np.array(energy_arr), np.array(imgs)
    else:
        return f, j


@njit
def iter_sor(f, b, idx, Np, omega):
    f_next = np.zeros(f.shape)
    for i in range(f.shape[0] - 1):
        f_next[i] = (1 - omega) * f[i] + omega * (
                b[i] + f_next[idx[0][i] - 1] + f_next[idx[1][i] - 1] + f[
            idx[2][i] - 1] + f[idx[3][i] - 1]) / Np[i]
    return f_next


def sor(f, b, delta, eps, Np, idx, channel, omega, target, source, mask, n,
        ref=None, save_history=False, n_steps=0):
    j = 0
    ind_mask = np.nonzero(mask)
    if save_history:
        energy_arr = []
        nums = []
        imgs = []
        ref_masked = deepcopy(ref[ind_mask])
        temp_result = deepcopy(target)
    if not n_steps:
        while abs(delta).max() > eps:
            if save_history and j % 5 == 0:
                update_history(temp_result, ind_mask, f, source, mask,
                               ref_masked, energy_arr, imgs)
                nums.append(j)
            # print(f"channel{channel}, step{j}, del{abs(delta).max()}")
            f = iter_sor(f, b, idx, Np, omega)
            delta[:-1] = iter_A(f, Np, idx)[:-1] - b
            j += 1

    else:
        for j in range(n_steps):
            if save_history and j % 5 == 0:
                update_history(temp_result, ind_mask, f, source, mask,
                               ref_masked, energy_arr, imgs)
                nums.append(j)
            # print(f"channel{channel}, step{j}, del{abs(delta).max()}")
            f = iter_sor(f, b, idx, Np, omega)
            delta[:-1] = iter_A(f, Np, idx)[:-1] - b
    if save_history:
        update_history(temp_result, ind_mask, f, source, mask, ref_masked,
                       energy_arr, imgs)
        nums.append(j)
        return f, np.array(nums), np.array(energy_arr), np.array(imgs)
    else:
        return f, j
```

```
194
195
196 def conjugated_gradient(f, b, eps, Np, idx, channel, target, source, mask, n,
197                          ref=None, save_history=False, n_steps=0):
198     j = 0
199     ind_mask = np.nonzero(mask)
200     if save_history:
201         energy_arr = []
202         nums = []
203         imgs = []
204         ref_masked = deepcopy(ref[ind_mask])
205         temp_result = deepcopy(target)
206
207     r = b - iter_A(f, Np, idx)[:-1]
208     p = np.zeros(f.shape)
209     p[:-1] = r
210
211     if not n_steps:
212         while abs(r).max() > eps:
213             if save_history and j % 5 == 0:
214                 update_history(temp_result, ind_mask, f, source, mask,
215                                ref_masked, energy_arr, imgs)
216                 nums.append(j)
217             # print(f"channel{channel}, step{j}, del{abs(r).max()}")
218             Ap = iter_A(p, Np, idx)[:-1]
219             alpha = np.sum(r ** 2) / np.sum(p[:-1] * Ap)
220             f[:-1] = f[:-1] + alpha * p[:-1]
221             r_next = r - alpha * Ap
222             beta = np.sum(r_next ** 2) / np.sum(r ** 2)
223             p[:-1] = r_next + beta * p[:-1]
224             r = deepcopy(r_next)
225             j += 1
226
227     else:
228         for j in range(n_steps):
229             if save_history and j % 5 == 0:
230                 update_history(temp_result, ind_mask, f, source, mask,
231                                ref_masked, energy_arr, imgs)
232                 nums.append(j)
233             # print(f"channel{channel}, step{j}, del{abs(r).max()}")
234             Ap = iter_A(p, Np, idx)[:-1]
235             alpha = np.sum(r ** 2) / np.sum(p[:-1] * Ap)
236             f[:-1] = f[:-1] + alpha * p[:-1]
237             r_next = r - alpha * Ap
238             beta = np.sum(r_next ** 2) / np.sum(r ** 2)
239             p[:-1] = r_next + beta * p[:-1]
240             r = deepcopy(r_next)
241             if np.sum(r) == 0:
242                 imgs.append([0 for i in range(0, n_steps - j - 1, 5)])
243                 break
244     if save_history:
245         update_history(temp_result, ind_mask, f, source, mask, ref_masked,
246                        energy_arr, imgs)
247         nums.append(j)
248         return f, np.array(nums), np.array(energy_arr), np.array(imgs)
249     else:
```

```
        return f, j


def gradient_descent_momentum(f, b, delta, eps, Np, idx, gamma, beta, channel,
                              target, source, mask, n, ref=None,
                              save_history=False, n_steps=0):
    j = 0
    ind_mask = np.nonzero(mask)
    if save_history:
        energy_arr = []
        nums = []
        imgs = []
        ref_masked = deepcopy(ref[ind_mask])
        temp_result = deepcopy(target)

    v_prev = - gamma * iter_A(delta, Np, idx)
    f = f + v_prev
    # print(f"channel{channel}, step{j}, del{abs(delta).max()}")
    delta[:-1] = iter_A(f, Np, idx)[:-1] - b
    j += 1

    if not n_steps:
        while abs(delta).max() > eps:
            if save_history and j % 5 == 0:
                update_history(temp_result, ind_mask, f, source, mask,
                               ref_masked, energy_arr, imgs)
                nums.append(j)
            v = beta * v_prev - gamma * iter_A(delta, Np, idx)
            f = f + v
            # print(f"channel{channel}, step{j}, del{abs(delta).max()}")
            delta[:-1] = iter_A(f, Np, idx)[:-1] - b
            v_prev = deepcopy(v)
            j += 1
    else:
        for j in range(n_steps):
            if save_history and j % 5 == 0:
                update_history(temp_result, ind_mask, f, source, mask,
                               ref_masked, energy_arr, imgs)
                nums.append(j)
            v = beta * v_prev - gamma * iter_A(delta, Np, idx)
            f = f + v
            # print(f"channel{channel}, step{j}, del{abs(delta).max()}")
            delta[:-1] = iter_A(f, Np, idx)[:-1] - b
            v_prev = deepcopy(v)

    if save_history:
        update_history(temp_result, ind_mask, f, source, mask, ref_masked,
                       energy_arr, imgs)
        nums.append(j)
        return f, np.array(nums), np.array(energy_arr), np.array(imgs)
    else:
        return f, j


def nesterov_loop(beta, v_prev, gamma, delta, Np, idx, f, j, channel, delta1,
                  b):
```

```
306     v = beta * v_prev - gamma * iter_A(delta, Np, idx)
307     f = f + v
308     # print(f"channel{channel}, step{j}, del{abs(delta1).max()}")
309     beta = 1 - 3 / (j + 1 + 5)
310     delta[:-1] = iter_A(f + beta * v_prev, Np, idx)[:-1] - b
311     delta1[:-1] = iter_A(f, Np, idx)[:-1] - b
312     v_prev = deepcopy(v)
313     return f, delta, delta1, v_prev, beta
314
315
316 def nesterov_accelerated_grad(f, b, delta, eps, Np, idx, gamma, beta, channel,
317                               target, source, mask, n, ref=None,
318                               save_history=False, n_steps=0):
319     j = 0
320
321     ind_mask = np.nonzero(mask)
322     if save_history:
323         energy_arr = []
324         nums = []
325         imgs = []
326         ref_masked = deepcopy(ref[ind_mask])
327         temp_result = deepcopy(target)
328
329     v_prev = - gamma * iter_A(delta, Np, idx)
330     f = f + v_prev
331     # print(f"channel{channel}, step{j}, del{abs(delta).max()}")
332     j += 1
333     beta = 1 - 3 / (j + 5)
334     delta[:-1] = iter_A(f + beta * v_prev, Np, idx)[:-1] - b
335     delta1 = np.zeros(delta.shape)
336     delta1[:-1] = iter_A(f, Np, idx)[:-1] - b
337
338     if not n_steps:
339         while abs(delta1).max() > eps:
340             if save_history and j % 5 == 0:
341                 update_history(temp_result, ind_mask, f, source, mask,
342                                ref_masked, energy_arr, imgs)
343                 nums.append(j)
344             f, delta, delta1, v_prev, beta = nesterov_loop(beta, v_prev, gamma,
345                                                            delta, Np, idx, f, j,
346                                                            channel, delta1,
347                                                            b)
348             j += 1
349     else:
350         for j in range(n_steps):
351             if save_history and j % 5 == 0:
352                 update_history(temp_result, ind_mask, f, source, mask,
353                                ref_masked, energy_arr, imgs)
354                 nums.append(j)
355             f, delta, delta1, v_prev, beta = nesterov_loop(beta, v_prev, gamma,
356                                                            delta, Np, idx, f, j,
357                                                            channel, delta1,
358                                                            b)
359
360     if save_history:
361         update_history(temp_result, ind_mask, f, source, mask, ref_masked,
```

```
362                         energy_arr, imgs)
363         nums.append(j)
364         return f, np.array(nums), np.array(energy_arr), np.array(imgs)
365     else:
366         return f, j
```