Heat and Color Flow with Finite Elements

Obed Afram, Anissa El Keurti, Robert Seidel

November 2016

Contents

List of Tables

List of Figures

Chapter 1

Finite Elements and the Poisson Equation

1.1 Gaussian Quadrature

Gaussian quadrature is one of the popular integration schemes for computing integrals that are not possible to solve analytically. In one dimension the Gaussian quadrature takes the form

$$\int_{-1}^{1} g(x)\,\mathrm{d}x \approx \sum_{q=1}^{Nq} \rho_q g(z_q), \tag{1.1}$$

where N_q is the number of integration points, z_q are the Gaussian quadrature points and ρ_q are the associated Gaussian weights. This extends to higher dimensions by

$$\int_{\hat{\Omega}} g(x)\,\mathrm{d}x \approx \sum_{q=1}^{Nq} \rho_q g(z_q), \tag{1.2}$$

and specifying the vector quadrature points z_q as well as integrating over a suitable reference domain $\hat{\Omega}$, e.g. squares or triangles in 2D, tetrahedrons or cubes in 3D.

1.1.1 1D quadrature

We write a MATLAB function `I = quadrature1D(a,b,Nq,g)`[1] with the following arguments:

- $I \in \mathbb{R}$, value of the integral,
- $a \in \mathbb{R}$, integration start,
- $b \in \mathbb{R}$, integration end,
- $N_q \in \mathbb{N}$, number of integration points,
- $g : \mathbb{R} \to \mathbb{R}$ function pointer.

We then test the written function by comparing with the analytical solution of the integral

$$\int_{1}^{2} e^x\,\mathrm{d}x = (e-1)e \approx 4.6708 \tag{1.3}$$

which happens to be the same, as the number of integration points is increased.

[1] All functions mentioned in this report can be found in the code files.

1.1.2 2D quadrature

In higher dimensions, we often map to barycentric coordinates (or area coordinates, as they are known in 2D). The Gauss points are then given as triplets in this coordinate system.

We write a MATLAB function `I = quadrature2D(p1,p2,p3,Nq,g)` with the following arguments:

- $I \in \mathbb{R}$, value of the integral,
- $p1 \in \mathbb{R}^2$, first corner point of the triangle,
- $p2 \in \mathbb{R}^2$, second corner point of the triangle,
- $p3 \in \mathbb{R}^2$, third corner point of the triangle,
- $N_q \in \{1,3,4\}$, number of integration points,
- $g : \mathbb{R}^2 \to \mathbb{R}$, function pointer.

We then verify our function by comparing it with the analytical solution of the integral

$$\iint_{\Omega} \log(x+y)\,\mathrm{d}x\mathrm{d}y, \tag{1.4}$$

where Ω is the triangle defined by the corner points $(1,0)$, $(3,1)$ and $(3,2)$. We solve the integral analytically by mapping the coordinates to the barycentric coordinates and compare the results.

$$N_0(\xi,\eta) = 1-\xi-\eta \tag{1.5}$$
$$N_1(\xi,\eta) = \xi \tag{1.6}$$
$$N_2(\xi,\eta) = \eta \tag{1.7}$$

$$x = P(\xi,\eta) = x_0N_0 + x_1N_1 + x_2N_2 = 1+2\xi+2\eta \tag{1.8}$$
$$y = Q(\xi,\eta) = y_0N_0 + y_1N_1 + y_2N_2 = \xi+2\eta \tag{1.9}$$
$$|J| = 2 \tag{1.10}$$

$$\iint_{\Omega} \log(x+y)\,\mathrm{d}x\mathrm{d}y = \iint_{\Omega} \log(P(\xi,\eta),Q(\xi,\eta))\cdot|J|\,\mathrm{d}\xi\,\mathrm{d}\eta \tag{1.11}$$
$$= \int_0^1 \int_0^{1-\eta} \log(1+3\xi+4\eta)\cdot 2\,\mathrm{d}\xi\,\mathrm{d}\eta \approx 1.16542 \tag{1.12}$$

This is approximately the same as the results from our function, with a small error when the number of integration points is increased.

1.1.3 3D quadrature

We now extend the barycentric coordinates to 3 dimensions and tetrahedral elements. We then write a MATLAB function `I=quadrature3D(p1,p2,p3,p4,Nq,g)` with the following arguments:

- $I \in \mathbb{R}$, value of the integral,
- $p1 \in \mathbb{R}^3$, first corner point of the triangle,
- $p2 \in \mathbb{R}^3$, second corner point of the triangle,

- $p3 \in \mathbb{R}^3$, third corner point of the triangle,
- $p4 \in \mathbb{R}^3$, fourth corner point of the triangle,
- $N_q \in \{1,4,5\}$, number of integration points,
- $g : \mathbb{R}^3 \to \mathbb{R}$, function pointer.

We verify our function by comparing it with the analytical solution of the integral

$$\iiint_{\Omega} e^x \,\mathrm{d}x\mathrm{d}y\,\mathrm{d}z \tag{1.13}$$

where Ω is the tetrahedron defined by the corner points $(0,0,0)$, $(0,2,0)$, $(0,0,2)$ and $(2,0,0)$. We solve the integral analytically by mapping the coordinates to the barycentric coordinates and compare the results with that of the numerical function which happens to be approximately the same. However, there is a little margin of error.

$$N_0(\xi,\eta,\zeta) = 1-\xi-\eta-\zeta \tag{1.14}$$
$$N_1(\xi,\eta,\zeta) = \xi \tag{1.15}$$
$$N_2(\xi,\eta,\zeta) = \eta \tag{1.16}$$
$$N_3(\xi,\eta,\zeta) = \zeta \tag{1.17}$$

$$x = P(\xi,\eta,\zeta) = x_0N_0 + x_1N_1 + x_2N_2 + x_3N_3 = 2\xi \tag{1.18}$$
$$y = Q(\xi,\eta,\zeta) = y_0N_0 + y_1N_1 + y_2N_2 + y_3N_3 = 2\eta \tag{1.19}$$
$$z = K(\xi,\eta,\zeta) = z_0N_0 + z_1N_1 + z_2N_2 + z_3N_3 = 2\zeta \tag{1.20}$$
$$|J| = 6 \tag{1.21}$$

$$\iiint_{\Omega} e^x \,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z = \iiint_{\Omega} e^{P(\xi,\eta,\zeta),Q(\xi,\eta,\zeta),K(\xi,\eta,\zeta)} \cdot |J| \,\mathrm{d}\xi\,\mathrm{d}\eta\,\mathrm{d}\zeta \tag{1.22}$$
$$= \int_0^1 \int_0^{1-\xi} \int_0^{1-\xi-\eta} e^{2\xi} \cdot 6 \,\mathrm{d}\zeta\,\mathrm{d}\eta\,\mathrm{d}\xi \tag{1.23}$$
$$= \frac{3}{4}(e^2-5) \approx 1.79179 \tag{1.24}$$

1.2 Poisson in 2D

We solve the two-dimensional Poisson problem, given by

$$\nabla^2 u(x,y) = -f(x,y) \tag{1.25}$$
$$u(x,y)|_{\partial\Omega} = 0 \tag{1.26}$$

with f given by

$$f(x,y) = 16\pi^2 xy(x^2+y^2)\sin(2\pi(x^2+y^2)) - 24xy\pi\cos(2\pi(x^2+y^2)). \tag{1.27}$$

The domain Ω is given by a three-quarter slice of the unit disc. In polar coordinates, we get

$$\Omega = \{(r,\theta) : r \leq 1 \text{ and } \theta \in [0, 3\pi/2]\}. \tag{1.28}$$

1.2.1 Analytical solution

We verify that the expression

$$u(x,y) = xy\sin(2\pi(x^2+y^2)) \tag{1.29}$$

is a solution to the problem in (1.25-1.26):

$$\begin{aligned}
\nabla^2 u(x,y) =& \frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} && (1.30)\\
=& \frac{\partial}{\partial x}\{y\sin(2\pi(x^2+y^2)) + 4\pi x^2 y\cos(2\pi(x^2+y^2))\} && (1.31)\\
&+ \frac{\partial}{\partial y}\{x\sin(2\pi(x^2+y^2)) + 4\pi x y^2\cos(2\pi(x^2+y^2))\} && (1.32)\\
=& 12\pi xy\cos(2\pi(x^2+y^2)) - 16\pi^2 x^3 y\sin(2\pi(x^2+y^2)) && (1.33)\\
&+ 12\pi xy\cos(2\pi(x^2+y^2)) - 16\pi^2 x y^3\sin(2\pi(x^2+y^2)) && (1.34)\\
=& -16\pi^2 xy(x^2+y^2)\sin(2\pi(x^2+y^2)) + 24\pi xy\cos(2\pi(x^2+y^2)). && (1.35)
\end{aligned}$$

1.2.2 Weak formulation

We start by multiplying equation (1.25) by a test function v and integrate over the domain Ω:

$$-\iint_\Omega (\nabla^2 u)v\,\mathrm{d}x\,\mathrm{d}y = \iint_\Omega fv\,\mathrm{d}x\,\mathrm{d}y. \tag{1.36}$$

We apply Green's formula for the Laplacian to the first integral, with the purpose of eliminating the second derivative in order to impose a lower regularity on the solution. We find

$$-\iint_\Omega (\Delta u)v\,\mathrm{d}x\,\mathrm{d}y = \iint_\Omega \nabla u\cdot\nabla v\,\mathrm{d}x\,\mathrm{d}y - \iint_{\partial\Omega}\frac{\partial u}{\partial n}v\,\mathrm{d}\gamma \tag{1.37}$$

We can now consider only test functions which vanish at the boundary of the domain, hence the contribution of the boundary terms vanishes. In this way, the equation becomes

$$\iint_\Omega \nabla u\cdot\nabla v\,\mathrm{d}x\,\mathrm{d}y = \iint_\Omega fv\,\mathrm{d}x\,\mathrm{d}y. \tag{1.38}$$

We further reduce the equation to the form

$$a(u,v) = F(v), \forall v \in V. \tag{1.39}$$

with the bilinear functional a and the linear functional F given by

$$a(u,v) = \iint_\Omega \nabla u\cdot\nabla v\,\mathrm{d}x\,\mathrm{d}y, \tag{1.40}$$

$$F(v) = \iint_\Omega fv\,\mathrm{d}x\,\mathrm{d}y. \tag{1.41}$$

The space V of test functions must therefore be such that if $v \in V$ then $v|_{\partial\Omega} = 0$ and v must be a least once (weakly) differentiable. So, we choose $V = H_0^1(\Omega)$.

1.2.3 Galerkin projection

Instead of searching for a solution u in the entire space V, we look for a solution in a much smaller space $V_h \subset V$. We let Ω be discretized into M triangles such that our computational domain is the union of all of these, i.e. $\Omega = \bigcup_{k=1}^{M}$. Each triangle K_k is then defined by its three corner nodes x_i $(i = 1,2,3)$ and for each of these nodes exists a corresponding basis function. The space V_h is then defined by

$$V_h = \{v \in V : v|_{K_k} \in P_1(K_k),\ k = 1,...,M\} \tag{1.42}$$

for which $n \in \mathbb{N}$ basis functions $\{\varphi_i\}_{i=1}^n$ satisfy $V_h = \text{span}\{\varphi_i\}_{i=1}^n$, $\varphi_j(x_i) = \delta_{ij}$. By searching for a solution $u_h \in V_h$, it is then possible to write this as a weighted sum of the basis functions, i.e. $u_h = \sum_{i=1}^n u_h^i \varphi_i$. We now find $u_h \in V_h$ such that $a(u_h, v) = F(v)\ \forall v \in V_h$. We arrive at the equations

$$a\left(\sum_{i=1}^{n} u_h^i \varphi_i, \varphi_j\right) = F(\varphi_i), \tag{1.43}$$

$$\sum_{i=1}^{n} a(u_h^i \varphi_i, \varphi_j) = F(\varphi_i), \tag{1.44}$$

$$\sum_{i=1}^{n} a(\varphi_i, \varphi_j) u_h^i = F(\varphi_i), \tag{1.45}$$

which is equivalent to the linear system

$$Au = f \tag{1.46}$$

with

$$A = [A_{ij}] = [a(\varphi_i, \varphi_j)], \tag{1.47}$$

$$u = [u_h^i], \tag{1.48}$$

$$f = [f_i] = [F(\varphi_i)]. \tag{1.49}$$

1.2.4 Implementation

The function `getSlice` generates the domain Ω. We apply the Gaussian quadrature in the computation and realize the computational work in the function `poisson2D_dirichlet`. We compare here three meshes of different sizes (i.e. $n = 50, 500$ and 1000). We observe from figure A.1 that we have the image to be very close to the real solution with a very little margin of error as the number of elements is increased.

Stiffness matrix

The stiffness matrix represents the system of linear equations that must be solved in order to ascertain an approximate solution to the differential equation. In order to solve for u, we first choose a set of basis functions and then compute the integrals defining the stiffness matrix. We discretized the domain Ω by some form of mesh generation, wherein it is divided into non-overlapping triangles or quadrilaterals, which are generally referred to as elements. Each triangle is then visited once by the algorithm.

Without the inclusion of boundary conditions, the system (1.25-1.26) does not have a unique solution. Thus, the matrix A is singular as well.

Right hand side

In the same manner, we build the right side vector f, as we did with A.

Boundary conditions

We implement homogeneous Dirichlet boundary conditions by deleting the boundary nodes in A and f.

1.2.5 Verification

We solve the system (1.46) and then after verification with the analytical solution (1.29), we observe that the both the solutions are approximately the same.

1.3 Neumann boundary conditions

We now change the boundary conditions of our problem to

$$\nabla^2 u(x,y) = -f(x,y) \tag{1.50}$$
$$u(x,y)|_{\partial\Omega_D} = 0 \tag{1.51}$$
$$\frac{\partial u(x,y)}{\partial n}|_{\partial\Omega_N} = g(x,y) \tag{1.52}$$

with the source term f and exact solution u given as above, and g defined as

$$g(x,y) = \begin{cases} -x\sin(2\pi x^2), & y = 0 \text{ and } (x,y) \in \partial\Omega_N \\ +y\sin(2\pi y^2), & x = 0 \text{ and } (x,y) \in \partial\Omega_N \end{cases} \tag{1.53}$$

The Dirichlet boundary surface is defined in polar coordinates by $\partial\Omega_D = \{(r,\theta) : r = 1, \theta \in [0, 3\pi/2]\}$, and the Neumann boundary surface as $\partial\Omega_N = \{(r,\theta) : r \in [-1,0], \theta = 3\pi/2\} \cup \{(r,\theta) : r \in [0,1], \theta = 0\}$.

1.3.1 Boundary condition

We now verify that (1.53) is a solution to (1.50-1.52) at the boundary.

$$\frac{\partial u}{\partial n} = \nabla u \cdot \vec{n} \tag{1.54}$$
$$= \left.\frac{\partial u}{\partial x}\right|_{y=0} \cdot \begin{bmatrix} 0 \\ -1 \end{bmatrix} + \left.\frac{\partial u}{\partial y}\right|_{x=0} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{1.55}$$
$$= \begin{cases} -x\sin(2\pi x^2), & y = 0 \text{ and } (x,y) \in \partial\Omega_N \\ +y\sin(2\pi y^2), & x = 0 \text{ and } (x,y) \in \partial\Omega_N \end{cases} \tag{1.56}$$

1.3.2 Variational formulation

When we introduce a test function $v \in V = \{v \in H^1(\Omega) : v|_{\Omega_D} = 0\}$ to (1.50-1.52), integrate over Ω and then apply Green's formula, we end up with the equation

$$-\iint_\Omega (\Delta u) v \,\mathrm{d}x\,\mathrm{d}y = \iint_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x\,\mathrm{d}y - \iint_{\partial\Omega} \frac{\partial u}{\partial n} v \,\mathrm{d}\gamma = \iint_\Omega f v \,\mathrm{d}x\,\mathrm{d}y. \tag{1.57}$$

We now introduce both the Dirichlet and Neumann boundary conditions

$$\iint_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x\,\mathrm{d}y = \iint_\Omega f v \,\mathrm{d}x\,\mathrm{d}y + \iint_{\partial\Omega_D} \frac{\partial u}{\partial n} v \,\mathrm{d}\gamma + \iint_{\partial\Omega_N} \frac{\partial u}{\partial n} v \,\mathrm{d}\gamma. \tag{1.58}$$

By imposing that the test function v vanishes on Ω_D, the equation becomes

$$\iint_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x\,\mathrm{d}y = \iint_\Omega f v \,\mathrm{d}x\,\mathrm{d}y + \iint_{\partial\Omega_N} g v \,\mathrm{d}\gamma. \tag{1.59}$$

The problem takes the form:

$$\text{Find } u \in V \text{ such that } a(u,v) = F(v) \text{ for all } v \in V, \text{ where} \tag{1.60}$$

$$a(u,v) = \iint_{\Omega} \nabla u \cdot \nabla v \, \mathrm{d}x \, \mathrm{d}y, \tag{1.61}$$

$$F(v) = \iint_{\Omega} f v \, \mathrm{d}x \, \mathrm{d}y + \iint_{\partial\Omega_N} g v \, \mathrm{d}\gamma. \tag{1.62}$$

1.3.3 Gauss quadrature

The Neumann boundary condition is given as an integral and must be evaluated using Gaussian quadrature. We then modify our quadrature methods from the first section to solve line integrals in two dimensions, i.e. the method signature is `I = linequadrature1D(a,b,Nq,g)` where require $a \in \mathbb{R}^2$ and $b \in \mathbb{R}^2$.

1.3.4 Implementation

We change our code to solve the Neumann boundary value problem. The solution in the interior was the same as the analytical solution. Approximately, there is a very small error comparing the solution at the boundary, see figure A.2.

1.4 Poisson in 3D

We now solve the problem (1.25-1.26) in three dimensions. We then generate a mesh, using the function `getBox`, which is similar to the written function in the previous section with the only difference is that spatial coordinates have one more component. The elements now require one more index to be described. The codes from the previous section are modified to deal with the tetrahedral elements in three dimensions with a new source term f, defined by

$$f(x,y,z) = 12\pi^2 \sin(2\pi x)\sin(2\pi y)\sin(2\pi z) \tag{1.63}$$

and homogeneous Dirichlet boundary conditions ($u^D = 0$). The problem has the exact solution

$$u(x,y,z) = \sin(2\pi x)\sin(2\pi y)\sin(2\pi z). \tag{1.64}$$

Chapter 2

Anisotropic Diffusion using Finite Element Method

After applying the Finite Element Method to common and abstract cases, we chose to work on a diffusion equation which is applied in image processing. A typical problem in image processing is *denoising*. One way to remove noise in a picture is to blur noisy areas while leaving edges unchanged. The linear heat equation can be used to blur an image, as the flow of heat can be interpreted as a flow of color on a plate. However, the linear heat equation does not preserve edges. In this project, we study the effects of a nonlinear PDE in image processing: the Perona-Malik anisotropic diffusion equation, that slows down the diffusion on domains where an edge has been detected. We explain our Finite Element solver for that problem and display various numerical experiments.

2.1 What is a picture?

In this section, we briefly describe some foundations of mathematical image processing. For this report, we shall assume that a *continuous image* u is an element of $L^\infty(\mathbb{R}^2)$, where each point in $\mathrm{dom}(u)$ maps to its corresponding grayscale value. Images typically have a compact support. When we take a picture, we generate a discretization of that functions u which is an element of $\mathbb{R}^{n_1 \times n_2}$. Generally, we can take horizontal and vertical derivatives of images, when we assume $u \in L^\infty(\mathbb{R}^2) \cap C^1(\mathbb{R}^2)$. There derivatives are numerically approximated by first order differences and can be interpreted as *edges*, when they take relatively high values.

We also calculate also the *peak-signal-to-noise ratio* (PSNR) by taking the initial rescaled image without noise as a reference. The PSNR is a tool that allows us to measure the quality of our reconstruction.

2.2 The Perona-Malik Equation and Finite Elements

2.2.1 Diffusion Equation

The Perona-Malik equation (Perona and Malik, 1990) is a nonlinear diffusion equation that uses an inhomogeneous diffusivity coefficient,

$$\frac{\partial u}{\partial t} = \nabla \cdot (g(\|\nabla u\|)\nabla u), \tag{2.1}$$

where typical choices for g, often denoted as *transfer function*, are

$$g(\|\nabla u\|) = e^{-(\|\nabla u\|/K)^2} \tag{2.2}$$

or

$$g(\|\nabla u\|) = \frac{1}{1 + \left(\frac{\|\nabla u\|}{K}\right)^2}. \tag{2.3}$$

Apparently, image areas with a small gradient experience strong blurring while edge areas remain unchanged. The original formulation (2.1) can lead to local backwards diffusion and become an ill-posed problem. Thus, we employ the so called *regularized* Perona-Malik equation as proposed by Catté et al. (1992), where the edge indicator $\|\nabla u\|$ is replaced by $\|\nabla(G_\sigma * u)\|$, $G_\sigma \in C^\infty(\mathbb{R}^2)$ a smoothing kernel. The rest of this section relies on the definitions as given in Handlovičová et al. (2002). We begin with the regularized Perona-Malik equation

$$\frac{\partial u}{\partial t} = \nabla \cdot (g(\|\nabla(G_\sigma * u)\|)\nabla u). \tag{2.4}$$

The function $u = u(t,x)$ is defined in $I \times \Omega$. We assume $I = [0,T]$ and $\Omega \subseteq \mathbb{R}^2$ to a bounded rectangular domain. The parameter t can be seen as abstract scaling parameter (Handlovičová et al., 2002, p. 219). We equip (2.4) with Neumann boundary conditions

$$\frac{\partial u}{\partial \nu} = 0 \qquad \text{on } I \times \partial\Omega, \tag{2.5}$$

$$u(0,x) = u^0(x) \qquad \text{in } \Omega. \tag{2.6}$$

Further, we assume $g : \mathbb{R}^+ \to \mathbb{R}^+$ to be a smooth, nonincreasing function, $g(0) = 1$ and $\lim_{s\to\infty} g(s) = 0$. For the smoothing kernel, we assume $G_\sigma \in C^\infty(\mathbb{R}^2)$, $\int_{\mathbb{R}} G_\sigma \,\mathrm{d}x = 1$, $\int_{\mathbb{R}} |G_\sigma| \,\mathrm{d}x < \infty$ and that G_σ converges to the Dirac measure if $\sigma \to 0$. Finally, let $u^0 \in L^\infty(\Omega)$. As described in Handlovičová et al. (2002), we do not compute the convolution $G_\sigma * u$, but use isotropic diffusion, i.e. $g \equiv 1$, in order to obtain a blurred version $u_i so$ of the image. The PDE simplifies in this case to

$$\frac{\partial u_{\text{iso}}}{\partial t} = \Delta u_{\text{iso}}. \tag{2.7}$$

2.2.2 Finite Element Method

We do not present the full derivation of the Finite Element procedure in this report, but refer to the lecture and the paper from Handlovičová et al. (2002). The linear isotropic solution u_{iso} is used to compute the diffusion coefficients $g(\|\nabla(G_\sigma * u)\|)$. We assume these coefficients as fixed in scale, i.e. we numerically compute

$$\frac{\partial u_{\text{ani}}}{\partial t} = \underbrace{\nabla \cdot (g(\|\nabla \underbrace{(G_\sigma * u^0)}_{=u_{\text{iso}}}\|)}_{\text{const.}} \nabla u_{\text{ani}}). \tag{2.8}$$

We use implicit Euler in scale and obtain the weak formulations

$$\int_\Omega u_{\text{iso}} v \,\mathrm{d}x + \sigma \int_\Omega \nabla u_{\text{iso}} \nabla v \,\mathrm{d}x = \int_\Omega u^0 v \,\mathrm{d}x, \tag{2.9}$$

$$\int_\Omega u_{\text{ani}} v \,\mathrm{d}x + \tau \int_\Omega g(\|\nabla u_{\text{iso}}\|) \nabla u_{\text{ani}} \nabla v \,\mathrm{d}x = \int_\Omega u^0 v \,\mathrm{d}x. \tag{2.10}$$

We consider a triangulation $\mathscr{T}_h$ of Ω and a nodal basis of continuous, piecewise linear functions φ_j, satisfying $\varphi_j(x_i) = \delta_{ji}$ for all nodes x_i of $\mathscr{T}_h$, $i, j = 1, ..., M$. The initial value u^0 and the unknown functions u_{iso} and u_{ani} are represented by

$$u_h^0 = \sum_{i=1}^M u_i^0 \varphi_i, \tag{2.11}$$

$$u_{\text{iso},h} = \sum_{i=1}^M u_{\text{iso},i} \varphi_i, \tag{2.12}$$

$$u_{\text{ani},h} = \sum_{i=1}^M u_{\text{ani},i} \varphi_i, \tag{2.13}$$

with the unknown scalar coefficients $u_i^0, u_{\text{iso},i}, u_{\text{ani},i} \in \mathbb{R}$, $i = 1,...,M$. With the Ritz-Galerkin method, we obtain

$$\sum_{i=1}^{M} \left(\int_\Omega \varphi_i \varphi_j \,\mathrm{d}x + \sigma \int_\Omega \nabla\varphi_i \nabla\varphi_j \,\mathrm{d}x \right) u_{\text{iso},i} = \sum_{i=1}^{M} \left(\int_\Omega \varphi_i \varphi_j \,\mathrm{d}x \right) u_i^0, \tag{2.14}$$

$$\sum_{i=1}^{M} \left(\int_\Omega \varphi_i \varphi_j \,\mathrm{d}x + \tau \int_\Omega g(\|\nabla u_{\text{iso},h}\|)\nabla\varphi_i \nabla\varphi_j \,\mathrm{d}x \right) u_{\text{ani},i} = \sum_{i=1}^{M} \left(\int_\Omega \varphi_i \varphi_j \,\mathrm{d}x \right) u_i^0. \tag{2.15}$$

This can also be written as two linear systems

$$[M + \sigma A(1)] \begin{bmatrix} u_{\text{iso},1} \\ \vdots \\ u_{\text{iso},M} \end{bmatrix} = M \begin{bmatrix} u_1^0 \\ \vdots \\ u_M^0 \end{bmatrix}, \tag{2.16}$$

$$[M + \tau A(g(\|\nabla u_{\text{iso},h}\|))] \begin{bmatrix} u_{\text{ani},1} \\ \vdots \\ u_{\text{ani},M} \end{bmatrix} = M \begin{bmatrix} u_1^0 \\ \vdots \\ u_M^0 \end{bmatrix}, \tag{2.17}$$

where $M_{j,i} = \int_\Omega \varphi_i \varphi_j \,\mathrm{d}x$ is the so-called *mass matrix* and $A(w)_{j,i} = \int_\Omega w \nabla\varphi_i \nabla\varphi_j \,\mathrm{d}x$ is the *stiffness matrix*. The parameters σ and τ denote the scale parameter for the two diffusion processes.

2.3 Numerical experiments

2.3.1 First steps

We performed several numerical simulations of the Perona-Malik diffusion using the Finite Element Method on 2D images. Our first sample is the `cameraman` picture, see figure B.1. We just write $u^0 \in \mathbb{R}^{n_1 \times n_2}$ from now on, to denote the initial data, where $n_1, n_2 \in \mathbb{N}$ denote the image dimensions.

NOISE GENERATION

We use a noisy image (by adding a random noise of parameter σ_{noise}) and we choose the scale of the initial image. In the figure **??**, the first image in the left represents the initial image that has been rescaled and added noise. The second image shows the mesh grid. Finally the two last images use two different interpolation methods on the grid, that will be developed later.

Triangular Mesh A naive approach for grid generation is to identify each pixel in the image with a node on the grid. The obtained squares are crossed diagonally to obtain triangles. This procedure yields a grid with $\mathcal{O}(n_1 n_2)$ nodes and $\mathcal{O}(2n_1 n_2)$ elements. It turns out handy to use the matrix indices as spatial coordinates, i.e. we set $\Omega = [1, n_1] \times [1, n_2]$ and the triangulation nodes $x_{i,j} = (i, j)$ for $i = 1,...,n_1$, $j = 1,...,n_2$, see figure 2.1.

$(1,1)$	$\dots$	$(1,n_2)$
$\vdots$	$\ddots$	$\vdots$
$(n_1,1)$	$\dots$	(n_1,n_2)

Figure 2.1: Matching of grid points and pixels.

We realized this grid generation procedure with the function `getSquareTri` that takes the dimensions of the picture `nx` as parameters and returns `p`, a list of node coordinates, `tri`, a list of all triangles, denoted by their corner node ids, and `edge`, the set of all boundary line segments, denoted by the node ids of their corresponding start and ending points.

Algorithm The first program we tested is basically a modified version of Part 1 of the project. First, we compute the grid and pass the following arguments to `pm_tri`:

- `u`, the initial data,
- `tri, p, edge`, the output of `getSquareTri` or another grid generator (see section 2.3.3),
- the scale parameters `sigma` and `tau`,
- the transfer function `g` as a function handle.

The function returns the followings values:

- `u_ani`, the anisotropic solution,
- `A_iso`, the isotropic stiffness matrix,
- `M`, the mass matrix,
- `u_iso`, the isotropic solution,
- `g_ani`, the diffusivity on the elements,
- `A_ani`, the anisotropic stiffness matrix.

The whole procedure can be described in the following way:

1. **Grid generation (GG)**
2. **Isotropic diffusion (ID)**
   (a) Initialization of M, A_{iso}, N_q.
   (b) For each triangle (visited once):
   - Get the node id to identify points and basis functions.
   - Get the coordinates of the nodes `p1,p2,p3`.
   - Compute the functions φ_i ($i = 1,2,3$) that land in the triangle.
   - For $i, j = 1,2,3$:
     – Define $f_{\text{iso}} = \nabla\varphi_i \nabla\varphi_j$, a constant function.
     – Update the isotropic stiffness matrix.
     `A_iso(nodeID(i),nodeID(j)) += quadrature2D(p1,p2,p3,Nq,f_iso);`
     – Compute $f_{\text{mass}} = \varphi_i \varphi_j$, a quadratic function.
     – Update the mass matrix.
     `M(nodeID(i),nodeID(j)) += quadrature2D(p1,p2,p3,Nq,f_mass);`

   (c) Solve the linear system `u_iso = (M + sigma * A_iso) \ (M * u)`.
3. **Compute the diffusion coefficients (DC)**
   (a) Initialization of the gradient matrix d.
   (b) For each triangle `k` (visited once):
   - Get the node id to identify points and basis functions.
   - Get the coordinates of the nodes `p1,p2,p3`.
   - Compute the functions φ_i ($i = 1,2,3$) that land in the triangle.
   - For $i = 1,2,3$:
     – Compute the local gradient $z = u_{\text{iso},i} \nabla\varphi_i$.

- Update `d(k) += z`.

(c) Evaluate the transfer function: `g_ani = g(pointwise_norm(d))`.

4. **Anisotropic diffusion (AD)**

   (a) Initialization of A_{ani}.

   (b) For each triangle `k` (visited once):

   - Get the node id to identify points and basis functions.
   - Get the coordinates of the nodes `p1,p2,p3`.
   - Compute the functions φ_i ($i = 1,2,3$) that land in the triangle.
   - For $i, j = 1,2,3$:
     - Define $f_{\text{ani}} = g_k \nabla\varphi_i \nabla\varphi_j$, a constant function.
     - Update the anisotropic stiffness matrix.
       `A_ani(nodeID(i),nodeID(j)) += quadrature2D(p1,p2,p3,Nq,f_ani);`

   (c) Solve the linear system `u_ani = (M + tau * A_ani) \ (M * u)`.

Visualization We display our results by the following four pictures (in direction of reading):

1. The first picture is the initial picture.
2. The second picture visualizes the used grid.
3. The third picture is a visualization of the anisotropic solution u_{ani} on the grid using `trisurf`. We used the *interpolated shading* in MATLAB to give a smooth result.
4. The fourth picture represents the anisotropic solution u_{ani} interpolated on a regular meshgrid. We used the function `griddata` and the method `natural` for the interpolation.

The different approaches for the visualization of u_{ani} become important when we use unstructured grids, see section 2.3.3.

Results Our visual impression is that the image noise is removed while the edges are preserved. Table 2.1 summarizes the first numerical experiment.

Image data	`cameraman` (128×128 pixels)
Grid generation	`getSquareTri`
Num. of nodes	16384
Num. of elements	32258
Parameters	$\sigma = 0.5$, $\tau = 0.8$, $g(x) = e^{-(12x)^2}$
Runtime GG	0.43145s
Runtime ID	52.7717s
Runtime DC	0.68412s
Runtime AD	27.6838s
PSNR	29.0056
Visualization	See figure B.2.

Table 2.1: Results of first numerical experiment

2.3.2 Efficient implementation

Method In order to make a more efficient implementation, we took the following measures:

- Precomputation and storage of triangle areas, basis function coefficients and triangle midpoints.
- Enforce more use of vector notation.
- Removal of the function call of `quadrature2D` and an anonymous function call. Instead, we used the precomputed values to get the triangular integral directly.

Results The program obtained is faster, see table 2.2, were the precomputation time is denoted as *Runtime PR*. We shall use this version of the code for the further examples.

Image data	`cameraman` (128×128 pixels)
Grid generation	`getSquareTri`
Num. of nodes	16384
Num. of elements	32258
Parameters	$\sigma = 0.5$, $\tau = 0.8$, $g(x) = e^{-(12x)^2}$
Runtime GG	0.43546s
Runtime PR	0.77099s
Runtime ID	5.4611s
Runtime DC	0.1238s
Runtime AD	2.3429s
PSNR	28.6091
Visualization	See figure B.3.

Table 2.2: Results of efficiency-improved numerical experiment

2.3.3 Grid generation strategies

Diagonal swapping

Method In order to fit the edges of the image better, we decided to implement an improved grid generation strategy by swapping the diagonals of some triangles. For each square that shall be divided in two triangles by `getSquareTri`, we compare the terms $|u_{\text{top left}} - u_{\text{bottom right}}|$ and $|u_{\text{top right}} - u_{\text{bottom left}}|$. If the first term is bigger than the second term, we split the square along the line through $x_{\text{top right}}$ and $x_{\text{bottom left}}$. Otherwise, we split the square along the line through $x_{\text{top left}}$ and $x_{\text{bottom right}}$. This behavior is implemented in `getSquareTri_swp`.

Results When we compare the results with the first grid, we notice that the new grid seems to be more adapted to the image. Table 2.3 summarizes.

Using variance information

Method Intuitively speaking, it is not necessary to use a high resolution grid on image areas with few variation. However, the use of a fine grid is indicated in areas with high variation such as corner and edge regions. In order to use less elements in the smooth areas, and more elements in the edge areas we decided to implement a new grid generation strategy by using the *variance* (as known from statistics) as criterion for the local coarseness of the grid.

We implement a recursive procedure `getVarigrid` as follows: A rectangular image portion is considered to contain enough information for grid refinement if the variance of the image portion lies above a certain threshold. If this is the case, the center of the rectangle is added as new node to the triangulation and the four resulting rectangles are analyzed recursively. We start with the whole image as initial image portion and its corner nodes as

Image data	`rhombus` (40×40 pixels)	
Grid generation	`getSquareTri`	`getSquareTri_swp`
Num. of nodes	1600	
Num. of elements	3042	
Parameters	$\sigma = 0.5$, $\tau = 0.8$, $g(x) = e^{-(12x)^2}$	
Runtime GG	0.009271s	0.008785s
Runtime PR	0.078194s	0.07624s
Runtime ID	0.33618s	0.33155s
Runtime DC	0.013159s	0.012268s
Runtime AD	0.13707s	0.13583s
PSNR	27.0748	27.1408
Visualization	See figure B.4.	See figure B.5.

Table 2.3: Results of diagonal swapping

initial nodes for the mesh. The triangulation of the so generated point set is computed by MATLAB using built-in `delaunayTriangulation` procedure.

Further improvements As improvement, we added a *corner value criterion* to the procedure. It also allows to perform a split if the absolute difference of any of two corner values are above a certain threshold. Finally, we set a minimal depth and maximal depth for the number of recursive function calls in order to ensure a minimal and maximal grid fineness. Because we have to deal with noisy data, there is always a certain amount of noise-induced variance and the variance criterion can be misleading. Our suggestion to overcome this issue is to preprocess the image with a simple Gaussian filter, such as MATLAB's built-in `imgaussfilt` routine.

Results The result seems satisfying regarding the few numbers of nodes and elements that still yield a good fit of the image data. The performance of the algorithm is improved by the reduced number of nodes and elements. Table 2.4 summarizes this experiment.

Image data	`cameraman` (256×256 pixels)
Grid generation	`getVarigrid`
Num. of nodes	6470
Num. of elements	12866
Parameters	$\sigma = 0.5$, $\tau = 0.8$, $g(x) = e^{-(12x)^2}$, $\sigma_{\text{imgaussfilt}} = 2$, $\sigma^2_{\text{min}} = 0.0001$, $d_{\text{corner}} = 0.2$, $n_{\text{mindepth}} = 4$, $n_{\text{maxdepth}} = 50$
Runtime GG	0.47358s
Runtime PR	0.31019s
Runtime ID	2.1717s
Runtime DC	0.05034s
Runtime AD	0.95878s
PSNR	23.9248
Visualization	See figure B.6.

Table 2.4: Results of variance based grid generation

Gradient information and random node sampling

Method An other way to capture image areas with *relevant* information such as corners and edge is the use of gradient information. Intuitively speaking, image domains with a small gradient contain few relevant information

while image domains with a high gradient are important for the visual image perception. This idea motivates the following grid generation strategy `getGradgrid`: We use MATLAB's matrix convolution `conv2` in order to compute the vertical and horizontal image derivatives. We then normalize the norm of the obtained gradients to values between 0 and 1. These values are used as probabilities for each pixel of the image to become a node in the grid. The random sample of grid points is again transformed by `delaunayTriangulation` to a triangular mesh.

Further improvements In order to prevent image noise to bias the gradient values, we apply again Gaussian filtering as a preprocessing step to the image before the grid is generated. We further allow to use a transfer function to map the normalized gradient norms to probability values.

Results The use of `getGradgrid` creates a very smooth and *cartoon-like* image. We think this comes from the fact that gradient information is taken twice into account: once in the grid generation and once when solving the Perona-Malik equation. This sets a stronger focus on edge preservation and smoothing of already smooth areas. Again, a smaller number of nodes have to be used. Table 2.5 summarizes this experiment.

Image data	`cameraman` (256×256 pixels)	
Grid generation	`getGradgrid`	
Num. of nodes	6610	
Num. of elements	13158	
Parameters	$\sigma = 0.5$, $g(x) = e^{-(12x)^2}$	
	$\sigma_{\text{imgaussfilt}} = 2$, $g_{\text{transfer}}(x) = x$	
$\tau =$	0.8	1.4
Runtime GG	0.050293s	
Runtime PR	0.32198s	0.3245s
Runtime ID	2.3301s	2.3281s
Runtime DC	0.051472s	0.052429s
Runtime AD	1.0208s	1.0213s
PSNR	23.1113	23.399
Visualization	See figure B.7.	See figure B.8.

Table 2.5: Results of variance based grid generation

2.3.4 Parameter modification

Anisotropic scale τ

We want to see the diffusion on the picture for different values of τ. We compare the PNSR of different scales and present the results in table 2.6. Notably, we find an optimal PSNR around $\tau = 0.5$.

Image data	`cameraman` (256×256 pixels)									
Grid generation	`getVarigrid`									
Num. of nodes	6610									
Num. of elements	13158									
Parameters	$\sigma = 0.5$, $g(x) = e^{-(12x)^2}$,									
	$\sigma_{\text{imgaussfilt}} = 2$, $\sigma^2_{\text{min}} = 0.0001$, $d_{\text{corner}} = 0.2$,									
	$n_{\text{mindepth}} = 4$, $n_{\text{maxdepth}} = 50$									
$\tau =$	0	0.5	1	5	10	50	100	500	1000	5000
PSNR	23.9	24.1	24.0	23.4	23.0	21.3	20.4	18.2	17.1	14.2
Visualization in fig.	B.9	B.10	B.11	B.12	B.13	B.14	B.15	B.16	B.17	B.18

Table 2.6: Results for different values of τ

Transfer function

The modification of the transfer function g can change the way, the diffusion performs on the image. We tested several transfer functions, that satisfy the conditions stated in section 2.2.1. Namely, we parametrize

$$g(x) = g(x;K) = e^{-(Kx)^2}, \tag{2.18}$$

and present the results in table 2.7.

Image data	`cameraman` (256×256 pixels)			
Grid generation	`getVarigrid`			
Num. of nodes				
Num. of elements				
Parameters	$\sigma = 0.5$, $\tau = 1.5$, $g(x) = e^{-(Kx)^2}$, $\sigma_{\text{imgaussfilt}} = 2$, $\sigma^2_{\text{min}} = 0.0001$, $d_{\text{corner}} = 0.2$, $n_{\text{mindepth}} = 4$, $n_{\text{maxdepth}} = 50$			
$K =$	-100	-10	-1	-0.1
PSNR	24.35	23.20	21.32	21.30
Visualization	See figure B.19.			

Table 2.7: Results for different transfer functions g.

According to the comparison between the PSNR values, $K = 100$ seems to be the best choice, the picture keeps many details but is still noisy. The choice $K = 10$ seems to give a good result with a smooth image. Generally, the choice of the transfer function depends on the application (e.g. deleting the noise or keeping the details).

Interpolation method

We tried to modify the visualization method of our results by changing the interpolation method for the anisotropic solution, MATLAB offers `natural`, `nearest`, `linear`, `cubic`, and `v4`. There was only a slight difference between most of the method, but we found `natural` the most appropriate.

Parameters of grid generation

The procedures `getVarigrid` and `getGradgrid` procedures require a even broader set of parameter choices. We can modify the parameters e.g. in order to increase the numbers of nodes and elements. For the variance based grid generation, we decrease the required *minimum variance* for a split, such that more nodes can be selected, see table 2.8. For the gradient based grid generation, we modify the transfer function g_{transfer}, see table 2.9.

Image data	`cameraman` (256×256 pixels)		
Grid generation	`getVarigrid`		
Num. of nodes	2970	6510	12927
Num. of elements	5923	12945	25660
Parameters	$\sigma = 0.5$, $\tau = 1.5$, $g(x) = e^{-(12x)^2}$, $\sigma_{\text{imgaussfilt}} = 2$, $d_{\text{corner}} = 0.2$, $n_{\text{mindepth}} = 4$, $n_{\text{maxdepth}} = 50$		
$\sigma^2_{\text{min}} =$	10^{-3}	10^{-4}	10^{-5}
PSNR	22.23	23.58	24.49
Visualization	See figure B.20.		

Table 2.8: Results for different values of σ^2_{min} in `getVarigrid`

Image data	`cameraman` (256×256 pixels)		
Grid generation	`getGradgrid`		
Num. of nodes	6998	2410	1277
Num. of elements	13940	4808	2546
Parameters	$\sigma = 0.5$, $\tau = 1.5$, $g(x) = e^{-(12x)^2}$		
	$\sigma_{\text{imgaussfilt}} = 2$		
g_{transfer}	$x \mapsto x$	$x \mapsto x^2$	$x \mapsto x^3$
PSNR	23.26	21.11	19.17
Visualization	See figure B.21.		

Table 2.9: Results for different functions g_{transfer} in `getGradgrid`

2.3.5 Quadrilateral elements

Method Instead of using triangular elements, we thought, we could use square element and exploit the regular structure of image data. We implemented a Finite Element algorithm using squares instead of the triangles. We used analytical computations to make the code faster by computing:

- The basis functions φ_i and their value on the integration point.
- The integrations by the midpoint rule.

The precomputation is done by a MATLAB script that handles symbolic variables and auto-generates the correct MATLAB code for the Finite Element procedure. The visualization shows:

1. The initial noisy image
2. The isotropic solution
3. The anisotropic solution

Results We notice a *checkerboard effect* (see figure B.19) on the visualization of the anisotropic solution. Shukla et al. (2013) explain that the checkerboard effect is a common problem of the finite element method. One way to resolve it can be to use higher order Finite Element which leads to a computationally heavier program. Other methods like the *perimeter control technique*, or the *patch technique* are proposed in the paper. We did not continue to work with quadrilateral elements. We summarize this numerical experiment in table 2.10.

Image data	`cameraman` (128×128 pixels)
Grid generation	`getSquareTet`
Num. of nodes	16384
Num. of elements	16129
Parameters	$\sigma = 0.5$, $\tau = 1$, $g(x) = e^{-(12x)^2}$
Runtime GG	0.16881s
Runtime ID	6.6781s
Runtime DC	0.005166s
Runtime AD	1.929s
PSNR	–
Visualization	See figure B.22.

Table 2.10: Results for quadrilateral elements

2.4 Outlook

Our work can be applied in several fields that involve image processing. In the medical field, image processing is often used to exact information from e.g. ultrasound imaging or Magnetic resonance imaging (MRI). The goal is for example to distinguish the layers of different tissues in order to find a tumor. Our algorithm could help to denoise such medical images and facilitate their interpretation for human or machine vision. In future work, our algorithm could be applied on 3D images as well.

From the mathematical perspective, it could be useful to develop higher order Finite Element method and increase the accuracy. It should be taken into account that image data is often in high resolution and the application of Finite Elements can become costly. However, the usage of advanced grid generation strategies can help to reduce the computational complexity, as we have shown in this report.

The treatment of color images can be relevant as well. A naive approach here is to use our greyscale Perona-Malix implementation on the (typically three) color layers of the color image. The resulting color fringes can be reduced by employing different color encoding schemes such as the *hue-saturation-value* (HSV) color representation (Bredies and Lorenz, 2010). The use of multiple coupled PDEs can be another approach to treat color images (Bredies and Lorenz, 2010).

Chapter 3

Conclusion

In the first part of the report, we have examined the Poisson equation from various perspectives. In the second part of this report, we showed how to implement the Perona-Malik diffusion equation using Finite Element Methods with an efficient algorithm. We also showed how to reduce the numbers of elements and nodes (so the algorithm's speed) by focusing on the important image areas (edges for example) by using different methods (variance and gradient). The comparison between the different methods leads to choices depending on the application chosen (medical data, picture restoration, etc.), since different factors have to be taken into account (quality, speed, smoothness).

Bibliography

Dirk Bredies and Kristian Lorenz. *Mathematische Bildverarbeitung - Einführung in Grundlagen und moderne Theorie*. Springer-Verlag, Berlin Heidelberg New York, 2010.

Francine Catté, Pierre-Louis Lions, Jean-Michel Morel, and Tomeu Coll. Image Selective Smoothing and Edge Detection by Nonlinear Diffusion. *SIAM Journal on Numerical Analysis*, 29(1):182–193, 1992.

Angela Handlovičová, Karol Mikula, and Fiorella Sgallari. Variational Numerical Methods for Solving Nonlinear Diffusion Equations Arising in Image Processing. *Journal of Visual Communication and Image Representation*, 13 (1):217–237, 2002.

P. Perona and J. Malik. Scale-space and edge detection using anisotropic diffusion. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(7):629–639, Jul 1990.

Avinash Shukla, Anadi Misra, and Sunil Kumar. Checkerboard Problem in Finite Element Based Topology Optimization. *International Journal of Advances in Engineering & Technology*, 6(4):1769, 2013.

Appendix A

Figures for Part 1

Figure A.1: Solution of the 2D Poisson equation with homogeneous Dirichlet boundary conditions.

Figure A.2: Solution of the 2D Poisson equation with mixed boundary conditions.

Appendix B

Figures for Part 2

Figure B.1: The `cameraman` picture

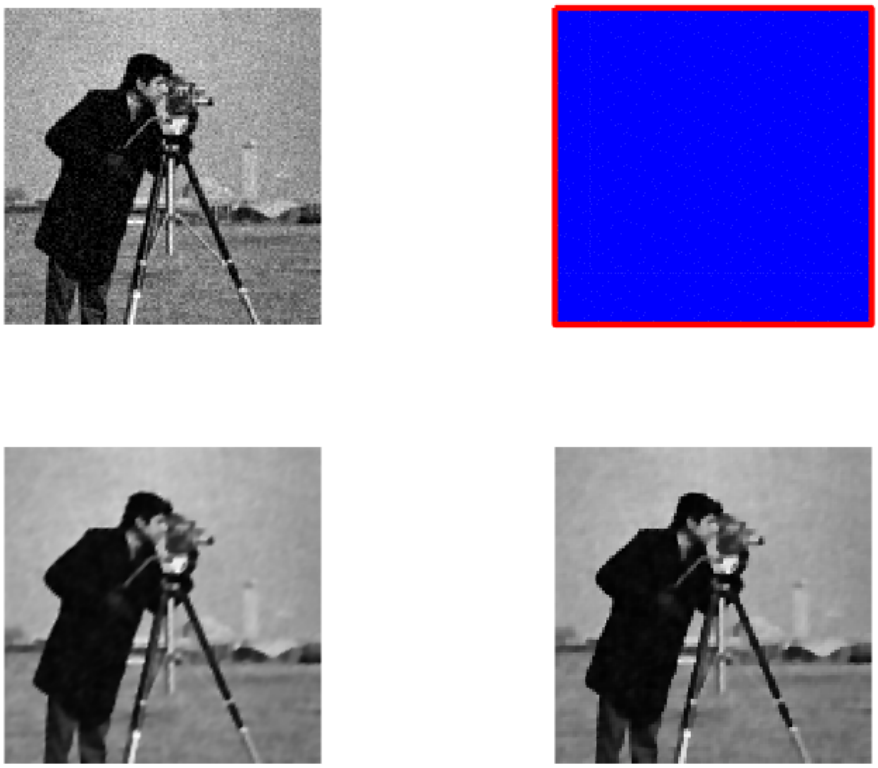

Figure B.2: Visualization of first numerical experiment

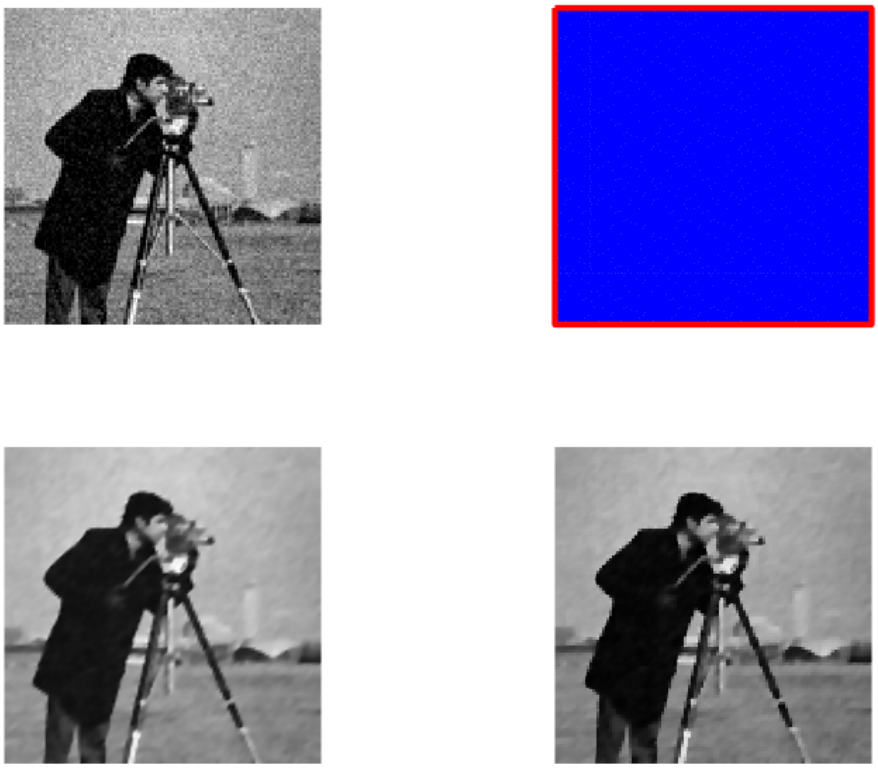

Figure B.3: Visualization of efficiency-improved numerical experiment

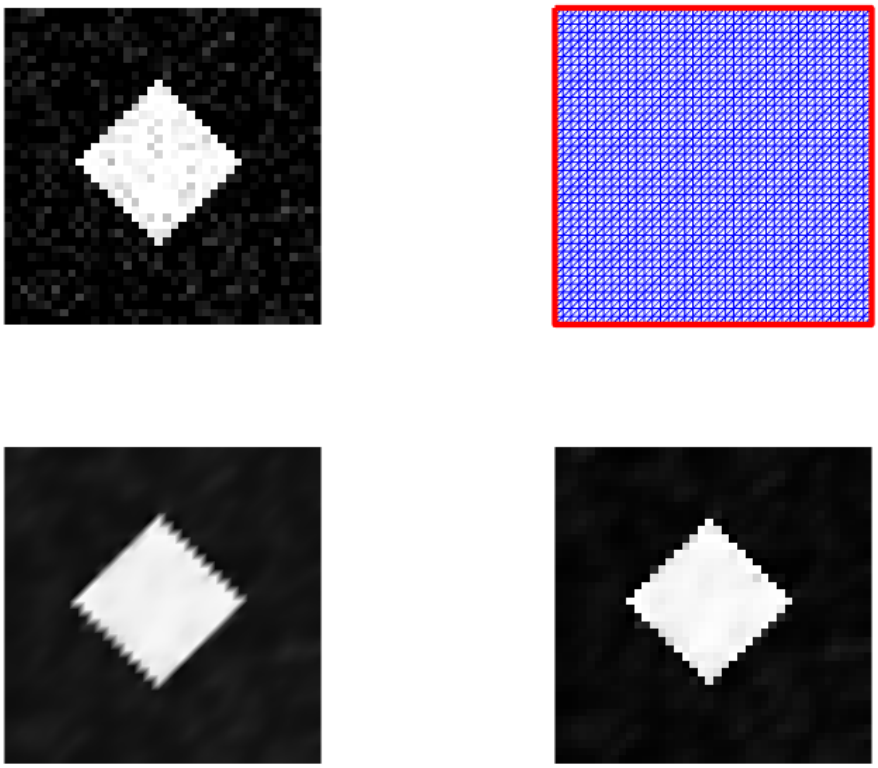

Figure B.4: Visualization of the *rhombus* denoising without diagonal swap

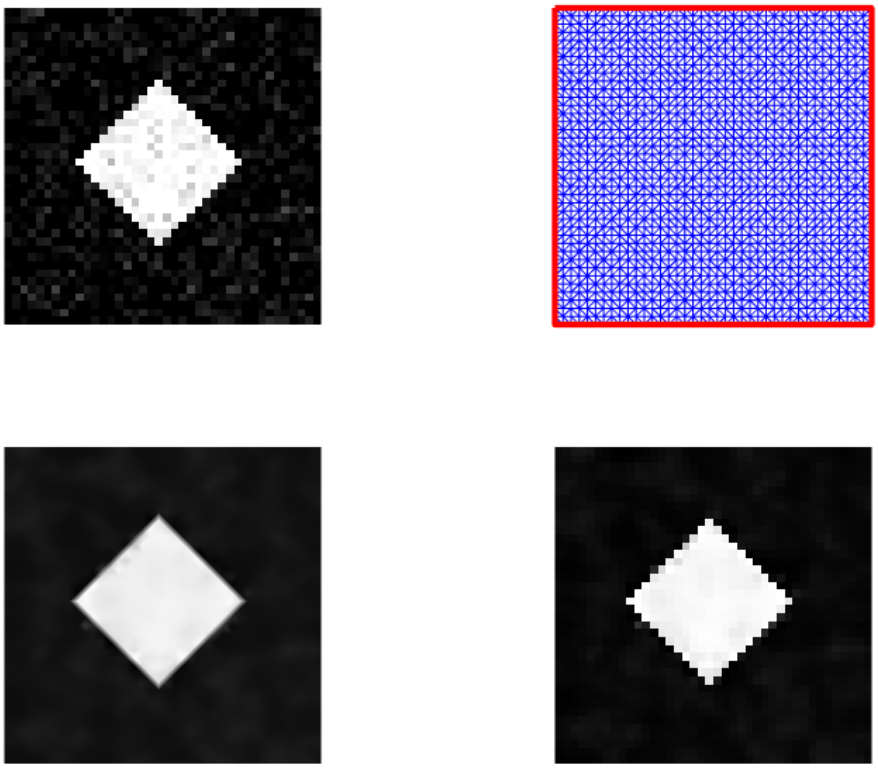

Figure B.5: Visualization of the *rhombus* denoising with diagonal swap

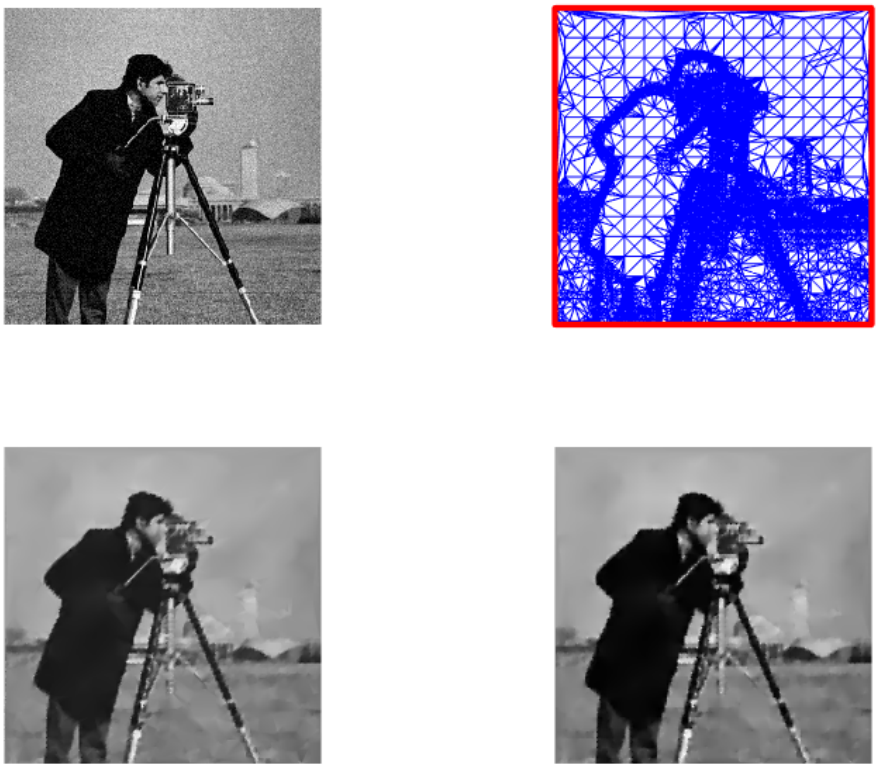

Figure B.6: Visualization of the *varigrid* experiment

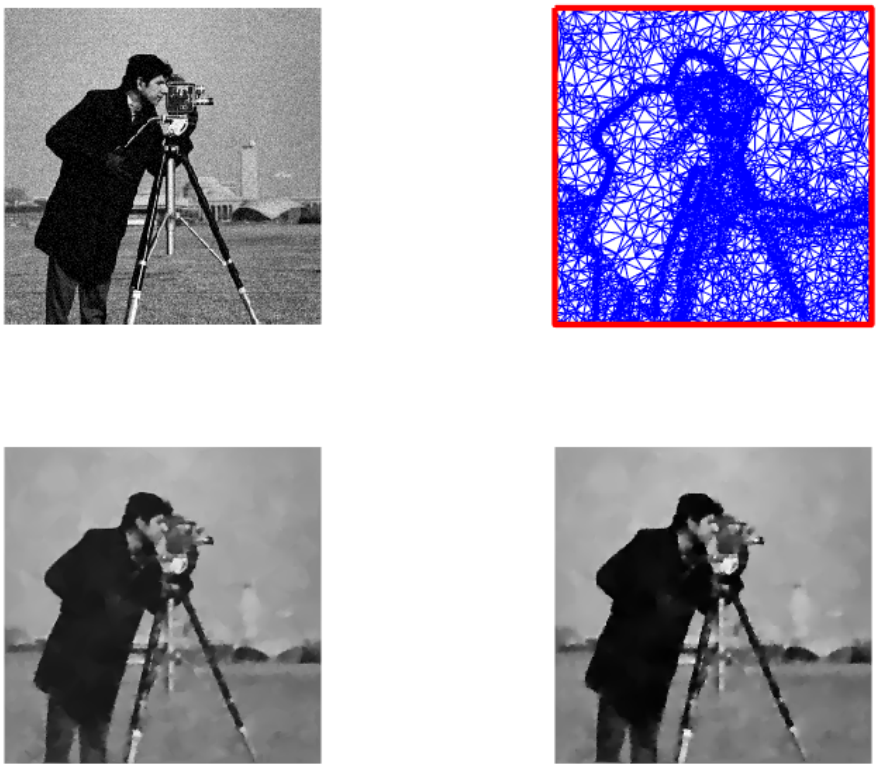

Figure B.7: Visualization of the *gradgrid* experiment with $\tau = 0.8$

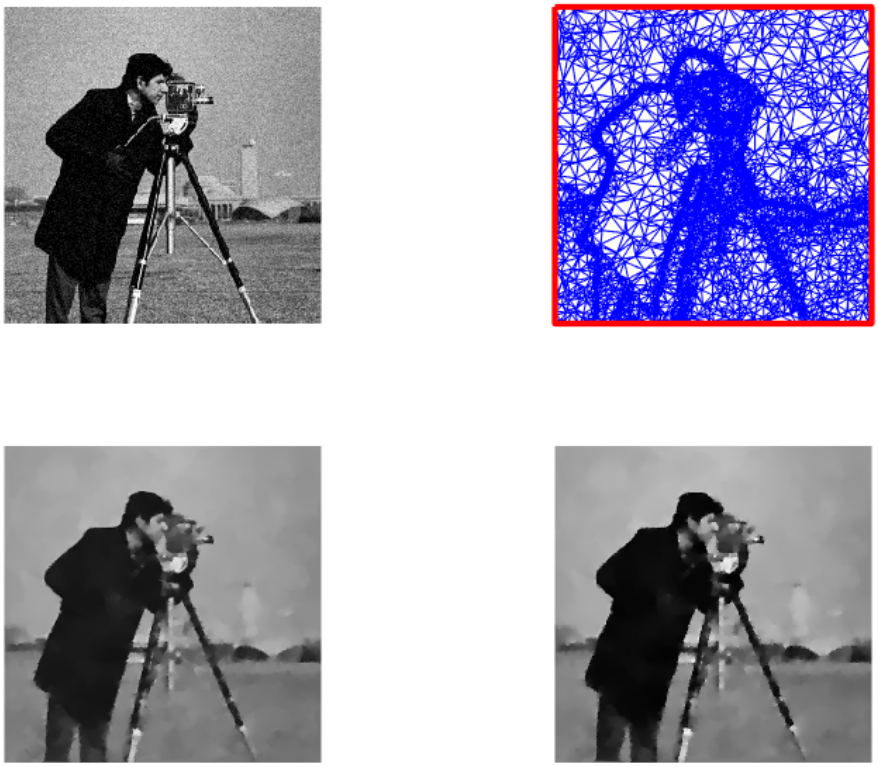

Figure B.8: Visualization of the *gradgrid* experiment with $\tau = 1.4$

Figure B.9: Variation of scale, here $\tau = 0$

Figure B.10: Variation of scale, here $\tau = 0.5$

Figure B.11: Variation of scale, here $\tau = 1$

Figure B.12: Variation of scale, here $\tau = 5$

Figure B.13: Variation of scale, here $\tau = 10$

Figure B.14: Variation of scale, here $\tau = 50$

Figure B.15: Variation of scale, here $\tau = 100$

Figure B.16: Variation of scale, here $\tau = 500$

Figure B.17: Variation of scale, here $\tau = 1000$

Figure B.18: Variation of scale, here $\tau = 5000$

Figure B.19: Results for different transfer functions g ($K = -100, -10, -1, -0.1$, in reading direction), see table 2.7

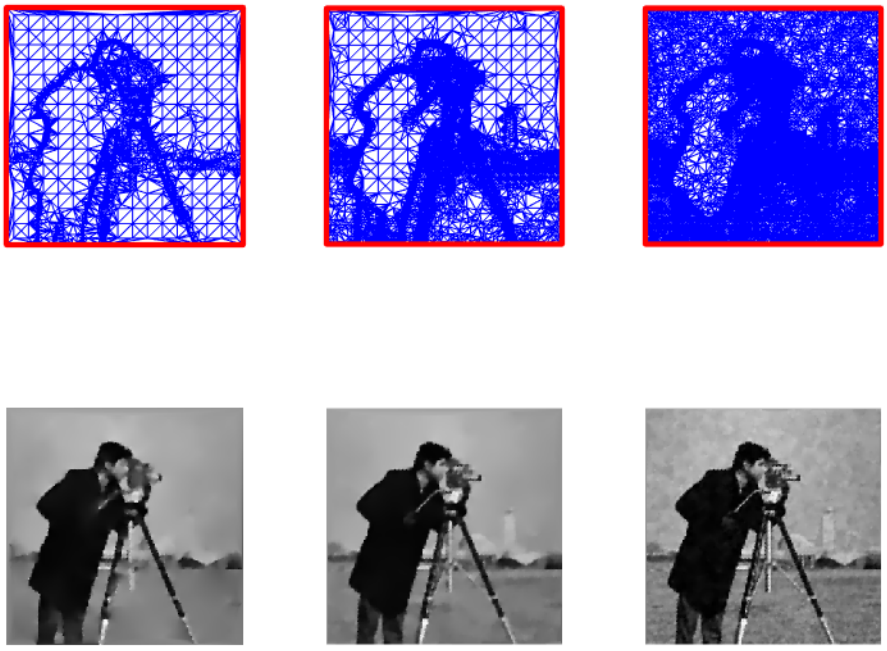

Figure B.20: Different minimal variance ($\sigma^2_{\text{min}} = 10^{-3}, 10^{-4}, 10^{-5}$, from left to right) in `getGradgrid` (top row) with corresponding solution (bottom row).

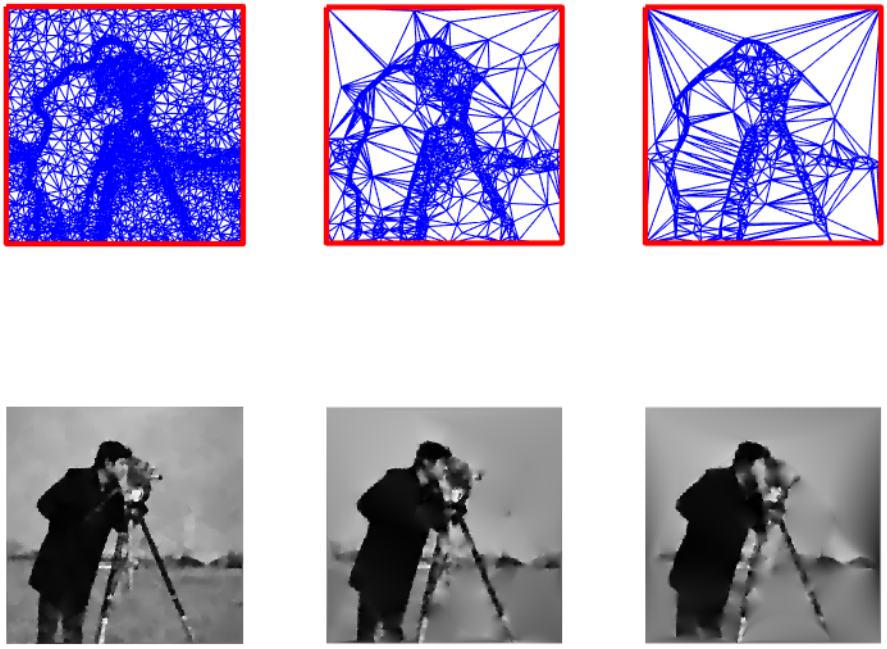

Figure B.21: Results for different functions g_{transfer} in `getGradgrid`, see table 2.9

Figure B.22: Results for quadrilateral elements with checkerboard effect