A Manifold Total Variation Minimization Template Library

Master Thesis

written by
Pascal Debus

supervised by
Markus Sprecher,
Prof. Dr. Philipp Grohs
Seminar for Applied Mathematics
ETH Zurich

October 1, 2015

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

Various forms of noise occur in many forms of data acquisition, transmission and processing. This noise needs to be removed in order to obtain a meaningful interpretation of the data, to enable further processing or, as in many image processing applications, just for aesthetic reasons. A common everyday example for a noisy image is taking a picture with a digital camera (e.g. integrated in a smart phone) in a weakly illuminated room: Especially the dark areas of the picture are not uniform in color and brightness but have small variations from pixel to pixel.

A noise removal algorithm needs to remove these small variations but at the same time not alter important features of the data. In the case of images important features are for example the edges separating areas of different colors and providing the necessary sharpness of the picture. These edges on the other hand are characterized by large variations. This distinction between small and large variations is also helpful in the task of inpainting, which tries to restore the picture at unknown or damaged regions.

The method of total variation(TV) noise removal, which has the above described capabilities, was first introduced by Rudin, Osher and Fatemi [23] in 1992 for the case of real-valued, that means grayscale images. Their method is briefly summarized in the following section.

1.1 Grayscale images

Let $u_0 : \mathbb{R} \supset \Omega \to \mathbb{R}$ describe the original, noise-free image, where the image domain Ω is usually a rectangular or cuboid subset of $\mathbb{R}^2$ or $\mathbb{R}^3$, respectively. Assuming the original picture is corrupted by Gaussian noise $n : \Omega \to \mathbb{R}$ with zero mean and variance σ^2 the noisy picture is given by $u : \Omega \to \mathbb{R}$, where $u = u_0 + n$. The edge preserving denoising of the picture is then equivalent to the solution $u^* : \Omega \to \mathbb{R}$ of the following constrained optimization problem:

$$u^* = \operatorname{argmin}_{f:\Omega\to\mathbb{R}} \int_\Omega |\nabla u| \, dx \quad \text{s.t.} \tag{1.1}$$

$$\int_\Omega (u - u_0) \, dx = 0, \quad \text{and} \int_\Omega (u - u_0)^2 \, dx = \sigma^2 \tag{1.2}$$

The first term $TV(u) = \int_\Omega |\nabla u| \, dx$ is called the total variation of u. Rudin, Osher and Fatemi then use a partial differential equation (PDE) approach to solve the corresponding Euler-Lagrange equation for (1.1). Later Chambolle and Lions [13] showed that (1.1) is equivalent to the minimization of the functional

$$J(u) = \frac{1}{2}\|u - u_0\|_2^2 + \lambda \int_\Omega |\nabla u| \, dx \tag{1.3}$$

1.1.1 Edge preservation

A basic intuition why the L^1 norm in (1.3) is better suited for conserving sharp discontinuities such as edges can be seen from the following plot, taken from [31]. The gradients in the integral expressions are to be understood in terms of their finite differences approximations with $\nabla f = f(x_{i+1}) - f(x_i)/h$ with some $h = 1/M < 1/N$, $x_i = 0$ and $x_{i+1} = x_i + h$.

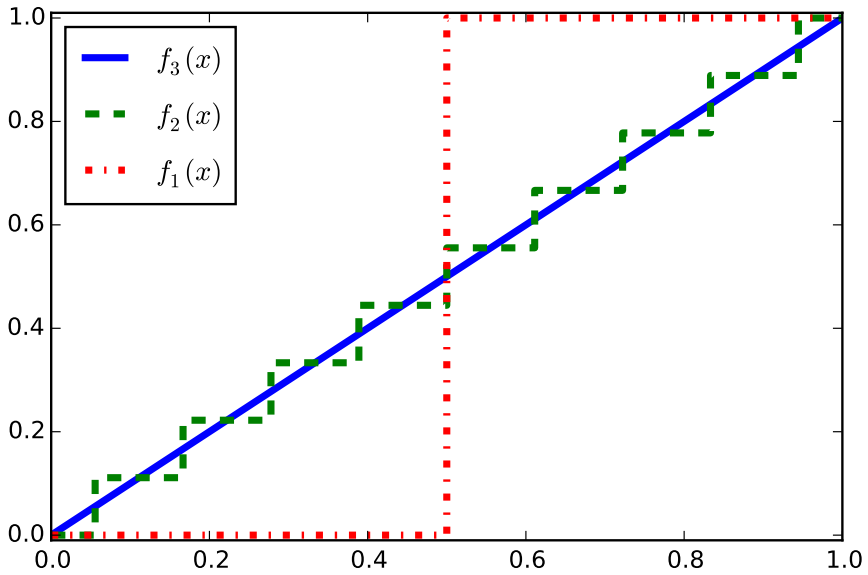


Figure 1.1: Plots of three functions with ($N = 1,\ 10,\ 100$) steps and a total variation equal to 1.0

Function	$\int_{[0,1]} \lvert\nabla f\rvert \, dx$	$\int_{[0,1]} \lvert\nabla f\rvert^2 \, dx$
f_1	1.0	1.0
f_2	1.0	0.1
f_3	1.0	0.01

One can see that the L^2 variation term favours continuous transitions, such as f_3, rather than the sudden jump in f_1 whereas the total variation is the same for all cases.

1.1.2 Discretization

For a grayscale image $u : \Omega \subset \mathbb{R}^2 \to \mathbb{R}$ we choose Ω as a discrete grid of pixels $\{1, 2, \ldots, m\} \times \{1, 2, \ldots, n\} \subset \mathbb{R}^2$. The picture u then takes the values $u_{i,j} := u((i,j)) \in [0,1]$ at the points $(i,j) \in \Omega$ and we can use a forward finite difference discretization of the gradient $\nabla u := (u_x, u_y)^T$ where

$$(u_x)_{i,j} = \begin{cases} u_{i,j+1} - u_{i,j}, & 1 \leq j < n \\ 0, & j = n \end{cases}$$
$$(u_y)_{i,j} = \begin{cases} u_{i+1,j} - u_{i,j}, & 1 \leq i < m \\ 0, & i = m \end{cases}. \tag{1.4}$$

We then arrive at the *isotropic* TV functional

$$TV_{iso}(u) = \sum_{i,j} \sqrt{(u_{i+1,j} - u_{i,j})^2 + (u_{i,j+1} - u_{i,j})^2}. \tag{1.5}$$

This TV term corresponds to the formulation originally proposed by Rudin et al[23], where $|\nabla u| = \sqrt{u_x^2 + u_y^2}$. Another possibility, which allows for more flexibility in the reweighting process described in 2.2.2 and which is even necessary for the proximal point algorithm described in 2.2.1, is the *anisotropic* version of the functional

$$TV_{aniso}(u) = \sum_{i,j} |u_{i+1,j} - u_{i,j}| + |u_{i,j_1} - u_{i,j}|. \tag{1.6}$$

For the total functional we only we have to add the pixel-wise differences to the original picture u_0

$$J(u) = \sum_{i,j} (u_{i,j} - (u_0)_{i,j})^2 + \lambda TV(u). \tag{1.7}$$

1.2 Color Images

The next step in the development of image denoising algorithms was their generalization to color images. From a mathematical perspective this just means considering pictures from $\Omega \to C \simeq \mathbb{R}^3$ where the form and additional properties of C depend on the chosen color model.
In the most simple case of linear models, like RGB for instance, one could choose C as $[0,1]^3$ and consider denoising each component individually (channel-by-channel model) or consider $\mathbb{R}^3$ as a normed vector space of tuples (x_R, x_G, x_B) (linear-vectorial model).
For the nonlinear models, especially the so-called chromaticity-brightness model, investigated by Chang and March [20], shows the closest resemblance to human perception. In this case we can take $C = S^2 \times [0,1]$ such that the chromaticitiy takes values on the sphere S^2 considered as a submanifold of the Euclidean space $\mathbb{R}^3$, while the brightness is real-valued, as in the case of grayscale images.

1.3 Manifold-valued Images

In the last section we have already seen that, depending on the chosen color model, pixels can take their values on a manifold and are usually represented by matrices. This data arises in a variety of applications such as Diffusion Tensor Magnetic Resonance Imaging (DTI-MRI), computer vision and robotics to name just a few.

1.4 Objective and Outline of this thesis

In this thesis we will introduce an extendable multi-threaded C++ template library for the purpose of TV minimization of manifold-valued images. So far the implemented minimization algorithms are based on the iteratively reweighted least squares (IRLS) adaption suggested by Sprecher and Grohs [18] as well as a proximal point algorithm by Weinmann et al [32]. We extend the implementation to 3D images cubes, the Grassmann manifold and also provide some quasi-analytic expressions for derivatives of the Riemannian distance function.

In the following Chapter 2 a short summary of the necessary theory, a description of the algorithms and relevant properties for each of the implemented manifolds is provided. After that, Chapter 3 introduces the library itself and in particular its capabilities, design concepts, structure, installation and usage in the form of some typical use cases. In Chapter 4 numerical experiments are conducted, showing various application of the library as well as convergence behavior and comparisons between IRLS and proximal point based algorithms.

Finally, chapter 5 concludes with possible extensions and adaptions of the library, in particular possibility of recursive splitting of the image domain into smaller subproblems and the transition to distributed architectures.

Chapter 2

Theory

2.1 Generalization of the functional

The functional defined in (1.7) needs a vector space structure for differences to make sense. Hence, the functional needs to be transformed to be still valid in the more general setting of manifold-valued pixels. Since on $\mathbb{R}$ we have $|x-y| = d(x,y)$ for the Euclidean distance, the generalization to metric spaces (X,d) is the appropriate way to proceed.

To also include the case of 3D pictures and shorten the notation we will use a graph G to specify over which pairs of pixels the sums in (1.7) are taken. Let V be an index set of all pixels in our picture. Denote by $E \subset V \times V$ the set of directed edges in G and by $n(i) := \{j \in V : (i,j) \in E\}$ the set of i's neighbors. Then

$$TV_{iso}(u) = \sum_{i \in V} \sqrt{\sum_{j \in n(i)} d^2(u_i, u_j)} \tag{2.1}$$

$$TV_{aniso}(u) = \sum_{(i,j \in E)} d2(u_i, u_j). \tag{2.2}$$

Since we used forward discretization, $n(i)$ just contains the next grid neighbors in each dimension, i.e. for $u_i = u((j,k,l))$, the neighbors are $n(i) = \{u((j+1,k,l)), u((j,k+1,l)), u((i,k,l+1))\}$. To also cover inpainting problems, let $V_k \subset V$ be the index set of pixels where the pixels of the (noisy) original image u_0 are known. Our generalized functional is given by

$$J(u) = \frac{1}{2} \sum_{i \in V_k} d^2(u_i, (u_0)_i) + \lambda TV(u). \tag{2.3}$$

2.1.1 The Riemannian distance

The metric d that is going to be used will of course depend on the manifold. Distances on Riemannian manifolds can be measured using smooth curves $\gamma : [a,b] \to M$ on the manifold. The length of this curve is

$$L(\gamma) = \int_a^b \sqrt{\langle \dot{\gamma}(t), \dot{\gamma}(t) \rangle_{\gamma(t)}}\, dt \tag{2.4}$$

and the distance can consequently be defined as

$$d : M \times M \to \mathbb{R}, \quad dist(x,y) = \inf_\Gamma L(\gamma) \tag{2.5}$$

where Γ denotes the set of smooth curves connecting $x, y \in M$ and $\langle \cdot, \cdot \rangle_x$ denotes the inner product on $T_x M$.

This rather general definition is used by [5] whereas for our case Pennec et al's [9] approach using Riemannian exponential and logarithm map is more convenient. Their definitions are based on *geodesics* because they are precisely the curves that realize the above minimum.

Let M be a connected, geodesically complete Riemannian manifold. The exponential mapping $\exp_x : T_xM \to M$ is defined by $\exp_x(\nu) := \gamma(1)$ where $\gamma : \mathbb{R} \to M$ is the unique geodesic with $\gamma(0) = x$ and $\dot{\gamma}(0) = \nu$. Thus, the function maps the tangent vector ν to the point y on the manifold reached after moving along γ for $t = 1$. The logarithm map is its inverse and hence provides us with the tangent vector ν that belongs to the geodesic connecting x and y.

A nice overview of Pennec's reinterpretation of vector space operations was given in [27].

Operation	Vector space	Riemannian manifold
Subtraction	$\nu = y - x$	$\nu = \log_x(y)$
Addition	$y = x + v$	$y = \exp_x(\nu)$
Distance	$\operatorname{dist}(x, y) = \|y - x\|$	$\operatorname{dist}(x, y) = \|\log_x(y)\|_x$

2.2 Algorithms

The next topic that must be addressed is the minimization of the above defined functional which brings about another challenge in the form of its non-differentiability. In the implementation, this problem is solved using two different methods. They are based on either working with a regularized version of the functional or by so called proximal mappings. The latter is used by the proximal point algorithm which will be shortly summarized in the next section before proceeding to the iteratively reweighted least squares (IRLS) algorithm.

2.2.1 Proximal Point

The proximal point algorithm for manifold valued data which is implemented in the MTVMT library is based on the work of Weinmann et al[32] and belongs to the class of proximal splitting methods. A survey on these methods for Euclidean space data is provided in [14]. The general scope in the real case are convex optimization problems of the form

$$\text{minimize}_{x \in \mathbb{R}^n} f_1(x) + \cdots + f_m(x) \tag{2.6}$$

where the $f_i : \mathbb{R}^n \to]-\infty, \infty]$ are convex functions but not necessarily differentiable. As we can see this is also true for the functional (2.3), even for the simple Euclidean case were the summands are given by $d(x, y) = |x - y|$.
Splitting means considering every summand f_i individually and minimize it using its proximal mapping

$$\operatorname{prox}_{f_i} x = \operatorname{argmin}_{y \in \mathbb{R}^n} \left(f_i(y) + \frac{1}{2}\|x - y\|_2^2 \right). \tag{2.7}$$

For the case of a differentiable function f the minimization problem (2.7) can be written in an explicit form and we see that $\operatorname{prox}_f x = x - \nabla f$, which can be interpreted as a gradient descent step.
In addition to that, one can show that the minimizers of f are exactly the fixed points of the proximal mapping (See [28], §2.3).

Application to manifolds

Let M be a Riemannian manifold and Ω as defined in 1.1.2. Due to the square root involved in the isotropic case, the method can only be applied to the anisotropic version of the functional (2.3) and

leads in the 2D case to the following splitting

$$J(u) = \sum_{i,j} F_{ij}(u) + \lambda \sum_{i,j} G_{ij}(u) + \lambda \sum_{i,j} H_{ij}(u) \tag{2.8}$$

$$F_{ij}(u) = d^2(u_{i,j}, (u_0)_{i,j}) \tag{2.9}$$

$$G_{ij}(u) = d(u_{i,j}, u_{i,j+1}) \tag{2.10}$$

$$H_{ij}(u) = d(u_{i,j}, u_{i+1,j}) \tag{2.11}$$

and proximal mappings of the form

$$\operatorname{prox}_{\lambda G_{ij}} x = \operatorname{argmin}_{y \in M^{m \times n}} \left(\lambda G_{ij}(y) + \frac{1}{2} d^2(x, y) \right). \tag{2.12}$$

We now only present the formula relevant for the implementation and finally the algorithm itself. For details on derivations and convergence and existence proofs consider [32].

The proximal mappings itself can be computed using unit speed geodesics. Here $[x, y]_t$ denotes the point reached by following the unit speed geodesic starting at x in direction y for a time t.

$$(\operatorname{prox}_{\lambda G_{ij}} u)_{ij} = [u_{i,j}, u_{i,j+1}]_{t_{TV}} \tag{2.13}$$

$$(\operatorname{prox}_{\lambda H_{ij}} u)_{ij} = [u_{i,j}, u_{i+1,j}]_{t_{TV}} \tag{2.14}$$

$$(\operatorname{prox}_{\lambda F_{ij}} u)_{ij} = [u_{i,j}, (u_0)_{i,j}]_{t_{l^2}} \tag{2.15}$$

Lastly, the times in the case of G_{ij} and F_{ij} are computed by

$$t_{TV} = \begin{cases} \mu, & \text{if } \mu < d(u_{i,j}, u_{i,j+1}) \\ \mu < d(u_{i,j}, u_{i,j+1}), & \text{else} \end{cases} \tag{2.16}$$

$$t_{l^2} = \frac{\mu}{1+\mu} d(u_{i,j}, (u_0)_{i,j}) \tag{2.17}$$

and for H_{ij} analogously.

In summary, the parallel version of the proximal algorithm now works by computing a proximal mapping for every pixel $u_{i,j}$ to its next neighbors on the grid and one to the original picture u_0, i.e. $[u_{i,j}, v]_t$ with $v \in \{u_{i-1,j}, u_{i+1,j}, u_{i,j-1}, u_{i,j+1}, (u_0)_{i,j}\}$. Next, the intrinsic (Karcher) mean [21] of these five mappings is computed and the pixel updated to a new value $u'_{i,j}$.

Finally, we can state the algorithm in 2.1.

2.2.2 IRLS

The IRLS approach to dealing with non-differentiable terms in the functional is by adding additional terms for regularization. In the continuous (and isotropic) case that means

$$TV_\epsilon = \int_\Omega \omega_\epsilon |\nabla u|^2 = \int_\Omega \frac{|\nabla u|^2}{\sqrt{|\nabla u|^2 + \epsilon^2}} \tag{2.18}$$

for a small $\epsilon > 0$. In this form, the functional becomes differentiable but directly minimizing a regularized functional J_ϵ will not work either.

The IRLS algorithm, described in more detail in [29], works by alternating between reweighting and minimization step. First, the weights are computed then the minimization is performed using the regularized functional with the weights considered constant. The steps for the isotropic functional

Algorithm 2.1 Parallel proximal point algorithm

Require: $\mu = (\mu_1, \mu_2, \ldots) \in l^2 \setminus l^1$
  $u \leftarrow u_0$
  for $r \leftarrow 1, 2, \ldots$ **do**
    for $i \leftarrow 1, 2, \ldots, m$; $j \leftarrow 1, 2, \ldots, n$; **do**
      $t \leftarrow t_{l^2} = \frac{\mu_r}{1+\mu_r} d(u_{i,j}, (u_0)_{i,j})$
      $z^{(1)} \leftarrow [u_{i,j}, (u_0)_{i,j}]_t$
      $t \leftarrow t_{TV}(\mu_r \lambda, u_{i,j}, u_{i,j+1})$
      $z^{(2)} \leftarrow [u_{i,j}, u_{i,j+1}]_t$
      $t \leftarrow t_{TV}(\mu_r \lambda, u_{i,j}, u_{i,j-1})$
      $z^{(3)} \leftarrow [u_{i,j}, u_{i,j-1}]_t$
      $t \leftarrow t_{TV}(\mu_r \lambda, u_{i,j}, u_{i,j+1})$
      $z^{(4)} \leftarrow [u_{i,j}, u_{i,j+1}]_t$
      $t \leftarrow t_{TV}(\mu_r \lambda, u_{i,j}, u_{i-1,j})$
      $z^{(5)} \leftarrow [u_{i,j}, u_{i-1,j}]_t$
      $u'_{i,j} \leftarrow \text{karchermean}(z^{(1)}, z^{(2)}, z^{(3)}, z^{(4)}, z^{(5)})$
    end for
    for $i \leftarrow 1, 2, \ldots, m$; $j \leftarrow 1, 2, \ldots, n$; **do**
      $u_{i,j} \leftarrow u'_{i,j}$
    end for
  end for

are as follows

$$w_i^{new} = W_{iso}^{\epsilon}(u)_i := \left(\sum_{j \in n(i)} d(u_i, u_j) + \epsilon^2 \right)^{-\frac{1}{2}} \quad \forall\, i \in V \tag{2.19}$$

$$u^{new} = U(w) := \text{argmin}_{u \in \Omega} \sum_{i \in V_k} d^2(u_i, (u_0)_i) + \lambda \sum_{i \in V} w_i \sum_{j \in n(i)} d^2(u_i, u_j). \tag{2.20}$$

The anisotropic steps are

$$w_{i,j}^{new} = W_{aniso}^{\epsilon}(u)_{i,j} := \left(d(u_i, u_j) + \epsilon^2 \right)^{-\frac{1}{2}}, \ \forall\, (i,j) \in E \tag{2.21}$$

$$u^{new} = U(w) := \text{argmin}_{u \in \Omega} \sum_{i \in V_k} d^2(u_i, (u_0)_i) + \lambda \sum_{(i,j) \in E} w_{i,j} d^2(u_i, u_j). \tag{2.22}$$

Further details on the derivation and proofs on convergence, existence and uniqueness of solutions for different manifold classes, such as Hadamard spaces or the sphere, can be found in [17]. The minimization can in principle be performed using various methods from smooth optimization theory. Due to its quadratic convergence rate, here the Newton method was chosen. Finally, the algorithm is given in 2.2.

2.3 Riemannian Newton method

2.3.1 Gradient

Let M be a Riemannian manifold and T_xM the tangent space at $x \in M$. Furthermore, let $f : M \to \mathbb{R}$ be a smooth function defined on M. We define the *Riemannian gradient* $\text{grad}\, f$ to be the unique tangent vector $\xi \in T_xM$ satisfying

$$\langle \text{grad}\, f(x), \xi \rangle_x = Df(x)[\xi] \tag{2.23}$$

where $Df(x)[\xi] = \xi[f]$, when $\xi : \mathcal{C}^\infty(M) \to \mathbb{R}$ is interpreted as derivation acting on f. The Riemannian gradient shares many properties of its Euclidean counterpart, in particular defining the direction of steepest ascent, such that first order methods like gradient descent could already

Algorithm 2.2 IRLS algorithm

Choose initial value $u^{(0)}$ by application of smoothing filter on u_0
$i \leftarrow 0$
repeat
  $W \leftarrow W^{\epsilon}(u^{(i)})$
  $u^{(i,0)} \leftarrow u^{(i)}$
  $k \leftarrow 0$
  repeat
    $u^{(i,k+1)} = newtonstep^{\lambda,u_0,V_k,V,E}(W, u^{(i,k)})$
    $k \leftarrow k + 1$
  until Stopping criteria (e.g. $k = 1, J(u^{(i,k+1)}) < J(u^{(i,0)}), d(u^{(i,k+1)}, u^{(i,k)})$
  $u^{(i+1)} \leftarrow u^{(i,k+1)}$
  $i \leftarrow i + 1$
until $d(u^{(i+1)}, u^{(i)}) < tol$
return $u^{(i+1)}$

be implemented at this point.

2.3.2 Hessian

For second order methods, such as the Newton algorithm, we also need a corresponding *Riemannian Hessian*. Following again the work of Absil et al [5], the Hessian is realized as linear endomorphism of the tangent space T_xM

$$\operatorname{Hess} f(x)[\xi] = \nabla_{\xi} \operatorname{grad} f(x) \tag{2.24}$$

where ∇ denotes the Riemannian connection on M.

2.3.3 Newton equation

Now let $x \in M$ and $\gamma : [a, b] \to M$ be a unique geodesic with $\gamma(0) = x$ and $\dot{\gamma}(0) = \nu \in T_xM$ passing through $y \in M$. Thus, we can set $y = \exp_x(\nu)$ and $\nu = \log_x(y)$ as suggested in 2.1.1. Using the definitions of gradient and Hessian above one can expand f to second order around x in the following way

$$\begin{aligned} f(\exp_x(\nu)) = f_x(\nu) = f(x) + \langle \operatorname{grad} f(x), \nu \rangle_x + \frac{1}{2} \langle \operatorname{Hess} f(x)[\nu], \nu \rangle_x + \mathcal{O}(\|\nu\|_x^3) \quad \Leftrightarrow \\ f(y) = f(x) + \langle \operatorname{grad} f(x), \log_x(y) \rangle_x + \frac{1}{2} \langle \operatorname{Hess} f(x)[\log_x(y)], \log_x(y) \rangle_x + \mathcal{O}(\|\log_x(y)\|_x^3). \end{aligned} \tag{2.25}$$

The first line of (2.25) suggests that, for given fixed x, we can also interpret f as a function of $\nu \in T_xM$ and perform the optimization f on the tangent space, on which grad f and Hess f have been defined. This finally leads to the following generalization of the *Newton equation* for the correction term $\eta_x \in T_xM$:

$$\operatorname{Hess} f(x)[\eta_x] = -\operatorname{grad} f(x) \tag{2.26}$$

The solution of the Newton equation η_x is a tangent vector of T_xM and, as (2.25) already implies, can be mapped back to the manifold M using the exponential map such that $y' = \exp_x(\eta_x)$. The whole iterative procedure, based on [5], Algorithm 5, is summarized in the following listing.

2.3.4 Newton equation for the TV functional

We now show the general form of the linear system defined by (2.26) in the case of manifold-valued 3D images $u, u_0 : \Omega \to M^{Z \times Y \times X}$ where $\Omega = \{1, \dots, Z\} \times \{1, \dots, Y\} \times \{1, \dots, X\}$ for $X, Y, Z \in \mathbb{N}$ and a corresponding functional $J : M^{Z \times Y \times X} \to \mathbb{R}$. Furthermore, the forward finite difference scheme described in 1.1.2 is used. To simplify notation, we also assume the regularization weights

Algorithm 2.3 Riemannian Newton method for real-valued functions

$x_0 \leftarrow x \in M$ (initial value)
for $k \leftarrow 0, 1, 2 \ldots$ **do**
  Solve newton equation $\operatorname{Hess} f(x_k)\eta_k = -\operatorname{grad} f(x_k)$ for $\eta_k \in T_{x_k}M$
  $x_{k+1} \leftarrow \exp_{x_k}(\eta_k)$
end for

obtained from the last IRLS reweighting step to be all equal to 1 and thus consider only the bare $d^2(\cdot,\cdot)$ terms:

$$J(u_{111}, u_{112}, \ldots, u_{ijk}, \ldots, u_{ZYX}) = \sum_{ijk} d^2(u_{ijk}, (u_0)_{ijk}) \tag{2.27}$$
$$+ \lambda \sum_{ijk}^{Z-1,Y,X} d^2(u_{ijk}, u_{i+1,j,k}) + \lambda \sum_{ijk}^{Z,Y-1,X} d^2(u_{ijk}, u_{i,j+1,k}) + \lambda \sum_{ijk}^{Z,Y,X-1} d^2(u_{ijk}, u_{i,j+1,k}).$$

We obtain the first derivatives

$$\begin{aligned}\frac{\partial J}{\partial u_{mnp}} = d_x^2(u_{mnp}, (u_0)_{mnp}) + \lambda\big(&d_x^2(u_{mnp}, u_{m+1,n,p}) + d_y^2(u_{m-1,n,p}, u_{mnp}) \\ &+ d_x^2(u_{mnp}, u_{m,n+1,p}) + d_y^2(u_{m,n-1,p}, u_{mnp}) \\ &+ d_x^2(u_{mnp}, u_{m,n,p+1}) + d_y^2(u_{m,n,p-1}, u_{mnp})\big),\end{aligned} \tag{2.28}$$

where $d_x^2((u,v)) := \frac{\partial}{\partial x}\, d^2(x,y)\big|_{(x,y)=(u,v)}$.

The second derivatives, considering only the terms in the first line of (2.28) containing the fidelity term and discrete gradient components in z direction, are then

$$\frac{\partial^2 J}{\partial u_{ijk}\partial u_{mnp}} = \delta_{nj}\delta_{kp}\delta_{mi} d_{xx}^2(u_{ijk}, (u_0)_{ijk}) \tag{2.29}$$
$$+ \lambda\delta_{nj}\delta_{kp}\Big[\delta_{mi}\Big(d_{xx}^2(u_{ijk}, u_{i+1,j,k}) + d_{yy}^2(u_{ijk}, u_{i+1,j,k})\Big) \tag{2.30}$$
$$+ \delta_{m-1,i} d_{xy}^2(u_{ijk}, u_{i+1,j,k}) + \delta_{m+1,i} d_{yy}^2(u_{i-1,j,k}, u_{ijk})\Big]. \tag{2.31}$$

Considering the pattern of the Kronecker deltas and reshaping the tensor of second partial derivatives into a single matrix, we can see that we obtain a block band matrix, where sums of non-mixed second derivatives of the squared distance function are located on the main diagonal, while all mixed derivatives belonging to the same gradient component (x, y or z) populate subdiagonals of equal distance to the main diagonal. In particular, discretizations in z-direction are on the first, in y-direction on the Zth and in x-direction on the $Z \times Y$th subdiagonals.

To illustrate this, consider the following schematic of the band structure

$$HJ = \begin{pmatrix} D & Z & & Y & & X & \\ Z & D & Z & & Y & & X \\ & Z & D & Z & & Y & \\ Y & & Z & D & Z & & Y \\ & Y & & Z & D & Z & \\ X & & Y & & Z & D & Z \\ & X & & Y & & Z & D \end{pmatrix}, \tag{2.32}$$

where D denotes a block, consisting of the sum of non-mixed second derivatives, according to the rules defined by the generalization of (2.29) to all discretization directions.

HJ is exactly the matrix representation of the Hessian operator $\operatorname{Hess} J(u)$, while the representation GJ of $\operatorname{grad} J(u)$ is just the vectorized version of the tensor of first derivatives (2.28). If the embedding space of the manifold is given by $\mathbb{R}^{n\times p}$ then the matrix HJ will be an element of $\mathbb{R}^{\tilde{D}\times\tilde{D}}$ with $\tilde{D} = npXYZ$. HJ is, however, also sparse with approximately $7\tilde{D}$ non-zero entries. Finally, solving the Newton equation (2.26) corresponds to solving the sparse linear system given by HJ and GJ, which is also the most computationally demanding part of the algorithm.

2.3.5 Tangent space restriction

We can choose a basis of the tangent space T_xM for each $x \in M$ and restrict the computed gradient and Hessian to the tangent space by performing a basis transformation. Since $\dim T_xM = \dim M$, this means that we can represent tangent vectors, such as the gradients of the squared distance function, or tangent space mappings like the Hessian using only $\dim M$ coefficients. We now have $D = \dim M(XYZ)$ such that the prefactor is now the intrinsic manifold dimension and not the dimension of the embedding space any more.

2.4 Manifolds

In this section we will present the relevant quantities needed to implement the IRLS and proximal point algorithm. These are the distance function and its derivatives, exponential and logarithm mapping and the tangent space basis transformation mapping. For the Grassmann manifold, due to its quotient manifold nature and because it was not covered in the original implementation based on [17] a more general introduction will be given.

2.4.1 Euclidean

The space in question is just $\mathbb{R}^n$, hence trivially a manifold and naturally a vector space such that all expressions can be calculated using basic multivariable calculus. The exponential and logarithm mappings are not to be understood in the sense of e^x but according to 2.1.1 such that they are just addition and subtraction in the vector space.

Exponential map

$$\exp_x(r) = x + r \tag{2.33}$$

Logarithm map

$$\log_x(y) = y - x \tag{2.34}$$

Squared distance function

$$d(x,y) = \|x-y\|^2 = \sum_{i=1}^{n}(x_i - y_i)^2 \tag{2.35}$$

First derivative of the squared distance function

$$\frac{\partial d^2(x,y)}{\partial y} = 2x \tag{2.36}$$

Second derivatives of the squared distance function

$$\frac{\partial^2 d^2(x,y)}{\partial x \partial y} = \frac{\partial^2 d^2(x,y)}{\partial x^2} = 2\mathbb{1}_n; \tag{2.37}$$

where here and in all following expression $\mathbb{1}_n$ denotes the identity matrix.

Tangent space restriction map

$$T_x = \mathbb{1}_{n^2} \tag{2.38}$$

2.4.2 Sphere S^n

We consider the manifold $S^n := \{x \in \mathbb{R}^{n+1} : \|x\| = 1\} \subset \mathbb{R}^{n+1}$ and express the maps in terms of vectors $x, y \in \mathbb{R}^{n+1}$ of the Euclidean embedding space.
The tangent space $T_x S^n$ at $x \in \mathbb{R}$ is, as basic intuition suggests, given by the tangent hyperplane to the sphere at x such that

$$T_x S^n := \{y \in \mathbb{R}^{n+1} : x^T y = 0\} \tag{2.39}$$

The standard Euclidean inner product $\langle r, s\rangle = r^T s$, restricted to the tangent space, turns S^n into a Riemannian manifold.

Exponential map

$$exp_x(r) = \cos(\|r\|_2)x + \frac{\sin(\|r\|_2)}{\|r\|_2} r \tag{2.40}$$

Logarithm map

$$\log_x(y) = \arccos(x^T y)\frac{y - x^T yx}{\|y - x^T yx\|_2} \tag{2.41}$$

Note that this is only well-defined for non-antipodal points $x, y \in S^n$.

Squared distance function

$$d(x, y) = \arccos(x^T y) \tag{2.42}$$

First derivative of the squared distance function

$$\frac{\partial d^2(x, y)}{\partial x} = \begin{cases} \frac{-2\arccos(x^T y)}{\sqrt{1-(x^T y)^2}} y, & x^T y \in (-1, 1) \\ -2y, & x^T y = 1 \end{cases} \tag{2.43}$$

Second derivatives of the squared distance function

$$\frac{\partial^2 d^2(x, y)}{\partial x^2} = \begin{cases} \left[\frac{2}{1-(x^T y)^2} - \frac{2\arccos(x^T)}{(1-(x^T y)^2)^{\frac{3}{2}})} x^T y\right] yy^T + 2x^T y\ \mathbb{1}_{n+1}, & x^T y \in (-1, 1) \\ \frac{2}{3} yy^T + 2x^T y \mathbb{1}_{n+1}, & x^T y = 1 \end{cases} \tag{2.44}$$

And the mixed derivative:

$$\frac{\partial^2 d^2(x, y)}{\partial x \partial y} = \begin{cases} \left[\frac{2}{1-(x^T y)^2} - \frac{2\arccos(x^T y)}{(1-(x^T y)^2)^{\frac{3}{2}}} x^T y\right] yx^T - \frac{-2\arccos(x^T y)}{\sqrt{1-(x^T y)^2}} y\ \mathbb{1}_{n+1}, & x^T y \in (-1, 1) \\ \frac{2}{3} yx^T - 2\ \mathbb{1}_{n+1}, & x^T y = 1 \end{cases} \tag{2.45}$$

Tangent space restriction map

As we can see from (2.39), the tangent space $T_x S^n$ at $x \in S^n$ is just the orthogonal complement of x in $\mathbb{R}^{n+1}$. Thus, constructing a basis of the tangent space amounts to constructing an orthonormal basis $\{b_0, b_1, \cdots, b_n$ of $\mathbb{R}^{n+1}\}$ with $b_0 = x$. Then $\mathcal{B} = \{b_i\}_{i=1}^n$ is the basis of the tangent space.
This can be done using the QR algorithm or, in the case of $S^2 \subset \mathbb{R}^3$, using the cross product. For our basis transformation mapping this means that

$$T_x : \mathbb{R}^{n+1} \to \mathbb{R}^n \tag{2.46}$$

$$T_x = \left(b_1 | b_2 | \cdots | b_n\right). \tag{2.47}$$

2.4.3 Special Orthogonal Group SO(n)

The special orthogonal group, considered as matrix group is defined as follows

$$SO(n) := \left\{ X\mathbb{R}^{n\times n} : X^T X = \mathbb{1}_n,\, \det(X) = 1 \right\}, \tag{2.48}$$

while its tangent space at $X \in SO(n)$ is

$$T_X SO(n) := \left\{ XS : S^T = -S \right\} = X[\mathrm{Skew}(n)]. \tag{2.49}$$

Here, the notation $X[\mathrm{Skew}(n)]$ means the set of all matrices that can be written as a product of X and a skew-symmetric matrix.
We equip $T_X SO(n)$ with the standard inner product $\langle R, S\rangle_X = \operatorname{tr} R^T S$ of its embedding space such that $(SO(n), \langle\cdot,\cdot\rangle)$ becomes a Riemannian manifold.

Exponential map

$$\exp_X R = X \exp(X^T R) \tag{2.50}$$

Note that the exponential map on the right hand side denotes the matrix exponential. The same holds for the logarithm on the right hand side of the following expression.

Logarithm map

$$\log_X(Y) = X \log(X^T Y) \tag{2.51}$$

First derivatives of the distance function

For the computation of the derivatives of the squared distance function, there exists a general analytic result by Karcher[21] that simplifies further computations considerably:

Theorem 2.1 (Karcher)**.** *Let M be a complete Riemannian manifold and $x, y \in M$ such that the geodesic connecting x and y is unique. Then the squared distance function to y is differentiable at x and we have*

$$\frac{\partial d^2(x,y)}{\partial x}[\cdot] = -2\left\langle \log_x(y), \cdot \right\rangle_x \tag{2.52}$$

where $\langle\cdot,\cdot\rangle$ is the Riemannian metric at $x \in M$.

Since with (2.51), we have a closed form expression for $\log_x(y)$ the first derivative can be expressed as

$$\frac{\partial d^2(X,Y)}{\partial X} = -2X \log(X^T Y). \tag{2.53}$$

Second derivatives of the distance function

For the computation of the second derivatives we can take the expression obtained using the above theorem as a starting point and follow the approach and notation of Magnus [26]. This allows us to express the derivatives as combinations of simple Kronecker products of the arguments, which is also very straightforward and compact to implement. The detailed derivations can be found in the appendix **??** while here we only present the final results.
For the second derivative with respect to the first argument one readily arrives at

$$\frac{\partial^2 d^2(X,Y)}{\partial X^2} = -2\left[\left(\left(\log X^T Y\right)^T \otimes \mathbb{1}_n \right) + (\mathbb{1}_n \otimes X)\, \mathrm{D}\log(X^T Y) \left(Y^T \otimes \mathbb{1}_n\right) K_{nn} \right], \tag{2.54}$$

where K_{nn} denotes the commutator matrix which transforms the column-wise vectorization of a matrix A to the vectorization of its transpose A^T.

The mixed derivative is given by

$$\frac{\partial^2 d^2(X,Y)}{\partial X \partial Y} = -2\,(\mathbb{1}_n \otimes X)\, \mathrm{D}\log(X^T Y) \left(\mathbb{1}_n \otimes X^T\right). \tag{2.55}$$

These expressions are quasi-analytic: Matrix logarithms and the Fréchet derivative of the matrix logarithm need to be evaluated numerically. Details concerning the implementation of the latter are postponed to Section 2.5.

Tangent space restriction map

The dimension of the tangent space $T_X SO(n)$ at $X \in SO(n)$ is $d = \frac{n(n-1)}{2}$. We define a basis for the space of skew-symmetric matrices Skew(n) in the following way. Let $K = \{(i,j) \in \mathbb{N}^2 : 1 \leq i < d,\ i < j \leq d\}$ and note that $|K| = d$. For all $k = (k_1, k_2) =\in K$ define the basis vector $B^{(k)}$ by

$$B_{ij}^{(k)} = \begin{cases} \frac{1}{\sqrt{2}}, & i = k_1 \text{ and } j = k_2 \\ -\frac{1}{\sqrt{2}}, & i = k_2 \text{ and } j = k_1 \\ 0, & \text{else} \end{cases}. \tag{2.56}$$

The basis of the tangent space is then $\mathcal{B}_{T_X SO(n)} = \{T^{(k)} = XB^{(k)}\}_{k \in K}$ and to obtain the basis transformation map we vectorize each basis matrix and define a matrix $T_X \in \mathbb{R}^{n^2 \times d}$ whose columns are given by the vectorized basis matrices.

$$T_X : \mathbb{R}^{n^2} \to \mathbb{R}^d \tag{2.57}$$

$$T_X = \left(\operatorname{vec} T^{(1)} | \operatorname{vec} T^{(2)} | \cdots | \operatorname{vec} T^{(d)}\right) \tag{2.58}$$

2.4.4 Symmetric Positive Definite Matrices SPD(n)

The cone of symmetric, positive definite matrices is defined as

$$SPD(n) := \left\{X \in \mathbb{R}^{n \times n} : X^T = X,\ y^T X y > 0\ \forall y \in \mathbb{R}^n \setminus \{0\}\right\}. \tag{2.59}$$

For $X \in SPD(n)$ the tangent space at X is isomorphic to the set of symmetric matrices

$$T_X SPD(n) := X[\text{Sym}(n)]. \tag{2.60}$$

The Riemannian metric defined on $T_X SPD(n)$ is given by

$$\langle R, S \rangle_X := \operatorname{tr}(X^{-1} R X^{-1} S). \tag{2.61}$$

Exponential map

$$\exp_X(R) = X^{\frac{1}{2}} \exp\left(X^{-\frac{1}{2}} R X^{-\frac{1}{2}}\right) X^{\frac{1}{2}} \tag{2.62}$$

Logarithm map

$$\log_X(Y) = X^{\frac{1}{2}} \log\left(X^{-\frac{1}{2}} Y X^{-\frac{1}{2}}\right) X^{\frac{1}{2}} \tag{2.63}$$

Squared distance function

As in the case of the special orthogonal group, the Riemannian metric (2.61) induces the distance function on SPD(n) such that

$$d^2(X,Y) = \| \log\left(X^{-\frac{1}{2}} Y X^{-\frac{1}{2}}\right) \|_F^2 \tag{2.64}$$

where $\| \cdot \|_F$ denotes the Frobenius norm on $\mathbb{R}^{n \times n}$.

First derivatives of the squared distance function

For the first derivatives we can apply theorem 2.1 again but some care must be taken in the computation this time since the Riemannian metric on $SPD(n)$ depends also on the base point X

of its tangent space $T_X SPD(n)$.

$$\frac{\partial d^2(X,Y)}{\partial X} = -2\left\langle \log_X(Y), \cdot \right\rangle_X \tag{2.65}$$

$$= -2\left\langle X^{\frac{1}{2}} \log\left(X^{-\frac{1}{2}} Y X^{-\frac{1}{2}}\right) X^{\frac{1}{2}}, \cdot \right\rangle_X \tag{2.66}$$

$$= -2\left\langle X^{-1} X^{\frac{1}{2}} \log\left(X^{-\frac{1}{2}} Y X^{-\frac{1}{2}}\right) X^{\frac{1}{2}} X^{-1}, \cdot \right\rangle_{\mathbb{1}_n} \tag{2.67}$$

$$= -2\left\langle X^{-\frac{1}{2}} \log\left(X^{-\frac{1}{2}} Y X^{-\frac{1}{2}}\right) X^{-\frac{1}{2}}, \cdot \right\rangle_{\mathbb{1}_n} \tag{2.68}$$

Second derivatives of the distance function

For the SPD matrices we proceed in the same way as for the orthogonal group and obtain

$$\frac{\partial^2 d^2(X,Y)}{\partial X^2} = 2\Bigg[\left(X^{-\frac{1}{2}} \log\left(X^{-\frac{1}{2}} Y X^{-\frac{1}{2}}\right)^T \otimes \mathbb{1}_n\right) + \left(\mathbb{1}_n \otimes X^{-\frac{1}{2}} \log\left(X^{-\frac{1}{2}} Y X^{-\frac{1}{2}}\right)\right) \tag{2.69}$$

$$+ \left(X^{-\frac{1}{2}} \otimes X^{-\frac{1}{2}}\right) \mathrm{D}\log(X^{-\frac{1}{2}} Y X^{-\frac{1}{2}}) \left(\left(X^{-\frac{1}{2}} Y \otimes \mathbb{1}_n\right) + \left(\mathbb{1}_n \otimes X^{-\frac{1}{2}} Y\right)\right) \Bigg] \times \cdots$$

$$\cdots \times \left(X^{-\frac{1}{2}} \otimes X^{-\frac{1}{2}}\right) \mathrm{D}(X^{\frac{1}{2}})$$

for the non-mixed derivatives and

$$\frac{\partial^2 d^2(X,Y)}{\partial X \partial Y} = -2\left(X^{-\frac{1}{2}} \otimes X^{-\frac{1}{2}}\right) \mathrm{D}\log\left(X^{-\frac{1}{2}} Y X^{-\frac{1}{2}}\right) \left(X^{-\frac{1}{2}} \otimes X^{-\frac{1}{2}}\right) \tag{2.70}$$

for the mixed ones.

Tangent space restriction map

For $SPD(n)$ we have that $d := \dim = \frac{n(n+1)}{2}$. In analogy to the previous manifold, we define a basis for the space of symmetric matrices Sym(n). In this case we have $K = K_1 \cup K_2 = \{(i,i) \in \mathbb{N}^2 : 1 \leq i \leq\} \cup \{(i,j) \in \mathbb{N}^2 : 1 \leq i < d,\ i < j \leq d\}$ such that again $|K| = d$. For all $k \in K$ define the basis vector $B^{(k)}$ by

$$B_{ij}^{(k)} = \begin{cases} 1, & i = k_1 \text{ and } j = k_2 \\ 1, & i = k_2 \text{ and } j = k_1 \\ 0, & \text{else} \end{cases}. \tag{2.71}$$

For $k \in K_1$, these are single-entry diagonal matrices. The basis of the tangent space is then $\mathcal{B}_{T_X SPD(n)} = \{T^{(k)} = X^{\frac{1}{2}} B^{(k)} X^{\frac{1}{2}}\}_{k \in K}$ and we proceed completely analogous to the $SO(n)$ case.

2.4.5 Grassmannian Gr(n,p)

The Grassmann manifold is special among the manifolds so far considered due to the fact that it is a quotient manifold. As such, there are different possibilities for choosing equivalence classes and representatives thereof.

For positive integers n and $p \leq n$ the Grassmann manifold is defined as the set of p-dimensional linear subspaces of $\mathbb{R}^n$. Since a linear subspace $\mathcal{Y} \in Gr(n,p)$ can be specified using a basis, we can arrange its basis vectors as columns of a matrix $Y \in \mathbb{R}^{n \times p}$ such that its column space spans $\mathcal{Y}$. The rank of Y must necessarily be full and equal to p because of the linear independence of its columns. Hence, elements of $Gr(n,p)$ can be represented using elements of the *non-compact Stiefel manifold*

$$\tilde{St}(n,p) := \left\{Y \in \mathbb{R}^{n \times p} :\ \operatorname{rank} Y = p\right\}. \tag{2.72}$$

Quotient representations

Observing now that post-multiplication by any invertible $G \in Gl(p)$ does not change the span of Y, we can form the equivalence classes

$$Y\,GL(p) := \{YG : G \in Gl(p)\} \tag{2.73}$$

consisting of all matrices having the same span as Y. These equivalence classes can be thought of as the distinct elements of the Grassmannian which leads to the following quotient manifold representation.

$$\tilde{G}r(n,p) := \tilde{S}t(n,p)/Gl(p) \tag{2.74}$$

This representation, used by Absil et al[4], is very general because only the rank is specified.

In the next steps of presenting the relevant quantities for the algorithm we will follow Absil's derivation and notation but choose the quotient representation used by Edelman et al [15] which is based on the orthogonal group. This will simplify most expressions and is also desirable from an algorithmic point of view as it removes more degrees of freedom in the choice of possibly unique representatives.

For the sake of completeness we also mention a completely different approach by Sato and Iwai [30] who choose $\mathbb{R}^{n\times n}$ as embedding space where elements of $Gr(n,p)$ are given by rank p orthogonal projection matrices. The presented applications are, however, mostly eigenvalue problems while in the case of image denoising the increased memory requirements are disadvantageous.

We denote by

$$St(n,p) = \left\{Y \in \mathbb{R}^{n\times p} :\ Y^TY = \mathbb{1}_p, \right\} \tag{2.75}$$

the *compact* Stiefel manifold.

The orthogonal group quotient representation of the Grassmann manifold, which is of course isomorphic to the previous representation, is given by

$$Gr(n,p) = St(n,p)/O(p) \tag{2.76}$$

The additional requirement is now that the basis spanning the subspace $\mathcal{Y}$ must be orthonormal.

Finally, we have the following canonical projection map to the quotient

$$\pi : St(n,p) \ni Y \mapsto \operatorname{span} Y = \mathcal{Y} \in Gr(n,p). \tag{2.77}$$

Locally unique representatives

From an algorithmic point of view it is desirable to work with representative as unique as possible for two reasons. Firstly, it provides us with the means to give well-defined expressions for various quantities we want to compute using arbitrary representatives and secondly, it allows us to find a parametrization of $Gr(n,p)$ in terms of $\mathbb{R}^{n\times p}$ matrices. This is necessary to construct a local basis of the tangent base and make the dimension of the sparse linear system a function of the intrinsic manifold dimension $(n-p)p$ instead of the embedding dimension np.

As we will see next, it is possible to obtain a set of locally unique representatives. By picking some other element $U \in St(n,p)$ and choose as representatives those elements who lie in the intersection of their equivalence classes and an affine cross section orthogonal to the equivalence class of U: Let $U \in St(n,p)$ and $\mathcal{U} := \operatorname{span}(U) \in Gr(n,p)$ and define the local affine cross section through U and orthogonal to the fiber $U[O(p)] = \pi^{-1}(\mathcal{U}) \subset St(n,p)$ by

$$S_U := \left\{V \in St(n,p) : U^T(V-U) = 0\right\} \subset St(n,p). \tag{2.78}$$

The equivalence class of $V \in St(n,p)$ is equal to $\pi^{-1}(\pi(V)) = V[O(p)]$ and to calculate its intersection with S_U we choose $R \in O(p)$ such that $VR \in V[O(p)]$ and obtain

$$VR \in S_U \;\Leftrightarrow\; U^T(VR - U) = 0 \;\Leftrightarrow\; R = (U^T V)^{-1} \tag{2.79}$$

which leads to the intersection

$$S_U \cap V[O(p)] = \left\{ VR = V(U^T V)^{-1} \right\}. \tag{2.80}$$

The intersection is empty if $U^T V$ is not invertible. Finally, we define a *cross-section mapping* σ_U restricted to the set

$$\mathcal{U}_U := \left\{ \mathcal{V} = \operatorname{span} V :\; U^T V \in GL(p) \right\} \tag{2.81}$$

by

$$\sigma_U : Gr(n,p) \supset \mathcal{U}_U \ni \mathcal{V} = \operatorname{span} V \mapsto V(U^T V)^{-1} \in S_U \subset St(n,p) \subset \mathbb{R}^{n \times p}. \tag{2.82}$$

σ_U is also a diffeomorphism providing the differentiable structure. The cross section map is illustrated in Figure 2.2.

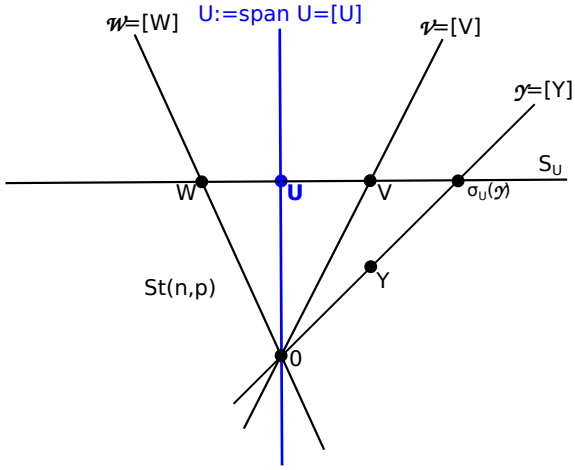


Figure 2.1: The cross section mapping is illustrated for the special case of $St(2,1) \subset \mathbb{R}^2$. Equivalence classes are all lines passing through the origin. We pick a $U \in St(2,1)$, then the cross section is given by all $V \in St(2,1)$ for which $U - V$ is orthogonal to U. This is true for the points V and W, representing $\mathcal{V}$ and $\mathcal{W}$, respectively but not for the representative Y of $\mathcal{Y}$. We can use the cross section mapping to obtain a representative $\sigma_U(\mathcal{Y})$ of $\mathcal{Y}$ which lies again on the cross section. Hence, the cross section is a tool for obtaining a set of locally unique representatives.

As an example for the application of the cross section map consider the calculation of averages on the Grassmann manifold.

Example 2.1 (Average)**.** *For the case of an average for, we can take representatives $Y_1, \ldots, Y_n \in St(n,p)$ for $\mathcal{Y}_1, \ldots, \mathcal{Y}_n \in Gr(n,p)$ and find a $U \in St(n,p)$ such that S_U has non-zero intersection with all the Y_i's equivalence classes, which is equivalent to $U^T Y_i \in Gl(p)$. The average $\mathcal{A}$ can then be written as*

$$\mathcal{A} := \pi\left(\sum_{i=1}^{n} \sigma_U(Y_i) \right) = \pi\left(\sum_{i=1}^{n} Y_i (U^T Y_i)^{-1} \right). \tag{2.83}$$

Tangent space

Due to the quotient structure which forces us to work with representatives we cannot just use the usual method for finding the tangent space by differentiating curves on the manifold. Instead we have to start with the "numerator" of the quotient $St(n,p)$. For the Grassmann manifold only tangent vectors of a special subspace of $T_Y St(n,p)$, the horizontal space, can modify the span of a

subspace and exactly those belong to the tangent space of $Gr(n,p)$. The notion of modifying and non-modifying tangent vectors can be best understood with the help of Figure **??**.

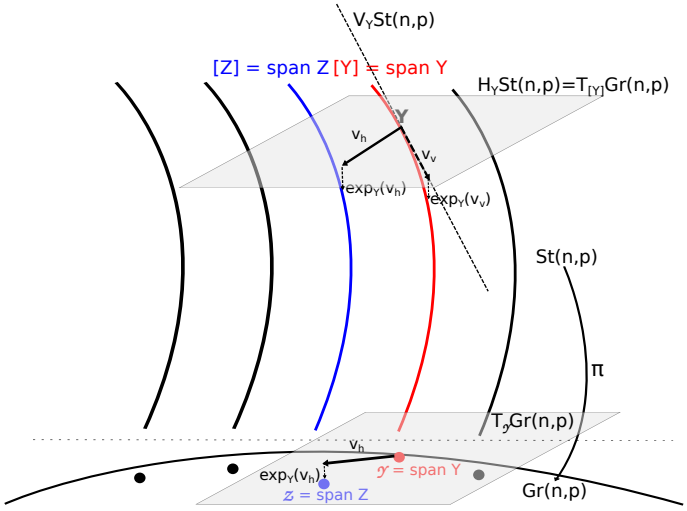


Figure 2.2

Let $Y \in St(n,p) \subset \mathbb{R}^{n\times p}$. Then the tangent space at Y ([5] for details of the derivation) to the compact Stiefel manifold is given by

$$\begin{aligned} T_Y St(n,p) &= \left\{ Z \in \mathbb{R}^{n\times p} : Y^T Z + Z^T Y = 0 \right\} \\ &= \left\{ Y\Omega + Y_\perp K : \Omega \in \mathrm{Skew}(p),\ K \in \mathbb{R}^{(n-p)\times p} \right\} \end{aligned} \tag{2.84}$$

where $Y_\perp \in \mathbb{R}^{n\times(n-p)}$ is such that $[Y, Y_\perp] \in O(n)$. The second representation of (2.84) already implies the decomposition into vertical and horizontal spaces we are going to perform next.
The vertical space at Y is by definition the tangent space to the fiber $\pi^{-1}(\pi(Y))$

$$V_Y = T_Y \pi^{-1}(\pi(Y)) = T_Y Y[O(p)] = Y[\mathrm{Skew}(p)], \tag{2.85}$$

while the horizontal space is defined as its orthogonal complement with respect to (2.84)

$$H_Y = V_Y^\perp = \left\{ H \in T_Y St(n,p) : Y^T H = 0 \right\} \simeq Y_\perp[\mathbb{R}^{(n-p)\times p}]. \tag{2.86}$$

Using this, the tangent space to $Gr(n,p)$ at $\pi(Y) = \mathcal{Y}$, along with its projector, is given by

$$T_{\mathcal{Y}} Gr(n,p) \simeq H_Y St(n,p) \simeq Y_\perp[\mathbb{R}^{(n-p)\times p}] \tag{2.87}$$

$$\pi_{Y_\perp} := \mathbb{1}_n - YY^T. \tag{2.88}$$

The problem that remains is to pick a unique representative for a tangent vector $\xi \in T_{\mathcal{Y}} Gr(n,p)$. This is resolved by demanding that the unique representative $\xi_{\diamond Y}$ should project to ξ via

$$d\pi(Y)\xi_{\diamond Y} = \xi \tag{2.89}$$

where $\pi : St(n,p) \to Gr(n,p)$ is the canonical quotient projection, such that $d\pi$ is a map between their tangent spaces. Using the cross section mapping (2.82), $\xi_{\diamond Y}$ can be computed by

$$\xi_{\diamond Y} = d\sigma_Y(\mathcal{Y})\xi. \tag{2.90}$$

$\xi_{\diamond Y}$ is called the *horizontal lift* of $\xi \in T_{\mathcal{Y}} Gr(n,p)$ at $Y \in St(n,p)$.

Finally, to obtain a basis for the tangent space we can choose $\{E_{ij}\}_{i=1,j=1}^{n-p,p}$, with the (i,j)th entry set to one and the rest zero, as a basis of $\mathbb{R}^{(n-p)\times p}$ and compute $Y_\perp$ using a QR decomposition

of Y. The orthogonal complement $Y_\perp$ is then just given by $Q_2 \in \mathbb{R}^{n\times(n-p)}$ which is part of the decomposition of the orthogonal matrix $Q = [Q_1, Q_2] \in \mathbb{R}^{n\times n}$.
For the basis of the tangent space we get

$$\{B_{ij}\}_{i=1,j=1}^{n,p} = \{Y_\perp E_{ij}\}_{i=1,j=1}^{n,p}\,. \tag{2.91}$$

Exponential map

Let X, R span $\mathcal{X}, \mathcal{R}$, respectively and let $U\Sigma V^T$ denote the thin singular value decomposition of R. Then

$$\mathrm{Exp}_{\mathcal{X}}(\mathcal{R}) = \mathrm{span}\left(XV\cos\Sigma V^T + U\sin\Sigma V^T\right). \tag{2.92}$$

Logarithm map

Let X, Y span $\mathcal{X}, \mathcal{Y}$, respectively and let $U\Sigma V^T$ denote the thin singular value decomposition of $Z = (\mathbb{1}_n - XX^T)Y(X^TY)^{-1}$.

$$\log_X(Y) = U\arctan\Sigma V^T \tag{2.93}$$

Note that using the notation introduced above, Z can also be written as $\pi_{X_\perp}\sigma_X(Y)$ which can be interpreted as picking a locally unique representative of Y with respect to the affine cross section defined by X and subsequently projecting it back to the tangent space $T_{\mathcal{X}}Gr(n,p)$ at $\mathcal{X}$.

Distance function

Using the exponential map, one can easily define a geodesic distance function on the Grassmann manifold which is induced by its Riemannian metric

$$g(X,Y) = \mathrm{Tr}X^TY \tag{2.94}$$

Then the distance function is given by the principal angles θ_i between the subspaces

$$d_g^2(X,Y) = \|\theta\|_2^2 = \sum_{i=1}^{p}\theta_i^2, \tag{2.95}$$

where the the principal angles can be obtained by computing the singular value decomposition of X^TY.

$$X^TY = U\Sigma V^T = U\cos\Theta V^T \tag{2.96}$$
$$\Sigma = \mathrm{diag}(\sigma_1,\cdots,\sigma_p) \tag{2.97}$$
$$\Theta = \mathrm{diag}(\theta_1,\cdots,\theta_p) = \mathrm{diag}(\arccos\sigma_1,\cdots,\arccos\sigma_p) \tag{2.98}$$

The distance function (2.95) has the disadvantage that due to the occurrence of the arccosine, analytic expression are much harder to obtain.

To avoid this problem, we follow Absil's [4] approach and choose an equivalent norm, the so-called projection Frobenius norm, given by

$$d_P^2(X,Y) = \frac{1}{2}\|XX^T - YY^T\|_F^2 = \sum_{i=1}^{p}\sin^2\theta_i \tag{2.99}$$

First derivatives of the distance function

$$\frac{\partial d^2(X,Y)}{\partial X} = 2\left(XX^T - YY^T\right)X \tag{2.100}$$

Second derivatives of the distance function

$$\frac{\partial^2 d^2(X,Y)}{\partial X \partial X} = 2\left[\left(X^T X \otimes \mathbb{1}_n\right) + \left(\mathbb{1}_p \otimes (XX^T - YY^T)\right) + \left(X^T \otimes X\right) K_{np}\right]. \tag{2.101}$$

The mixed derivative is given by

$$\frac{\partial^2 d^2(X,Y)}{\partial X \partial Y} = -2\left[\left(X^T Y \otimes \mathbb{1}_n\right) + \left(X^T \otimes Y\right) K_{np}\right]. \tag{2.102}$$

2.5 Fréchet derivatives of matrix logarithm and square root

To use the derivative expression computed above we need the so called Kronecker form of the Fréchet derivative. The Fréchet derivative of a matrix valued function $f : \mathbb{R}^{n\times n} \to \mathbb{R}^{n\times n}$ at a point $X \in \mathbb{R}^{n\times n}$ is a linear function mapping $E \in \mathbb{R}^{n\times n}$ to $L_f(X,E) \in \mathbb{R}^{n\times n}$ such that

$$f(X+E) - f(X) - L_f(X,E) = o(\|E\|). \tag{2.103}$$

Chain rule and inverse function theorem also hold for the Fréchet derivative:

$$L_{f\circ g}(X,E) = L_f(g(X)), L_g(X,E)) \tag{2.104}$$

$$L_f(X, L_{f^{-1}}(f(X),E)) = E \tag{2.105}$$

As we did in our formulation of the derivatives of the distance function, it can also be represented in the Kronecker form in which it is represented as map $K_f : \mathbb{R}^{n^2} \to \mathbb{R}^{n^2}$, such that $K_f(X) \in \mathbb{R}^{n^2\times n^2}$ is defined by

$$\operatorname{vec}(L_f(X,E)) = K_f(X)\operatorname{vec}(E). \tag{2.106}$$

Here $\operatorname{vec} : \mathbb{R}^{n\times n} \to \mathbb{R}^{n^2}$ denotes the column-wise vectorization operator.

2.5.1 Derivative of the matrix square root

We start by considering the Fréchet derivative of $f(X) = X^2$, which is given by

$$L_{X^2}(X,E) = XE + EX. \tag{2.107}$$

Applying the inverse function theorem consequently leads to

$$L_{X^2}(X^{\frac{1}{2}}, L_{X^{\frac{1}{2}}}(X,E)) = X^{\frac{1}{2}} L_{X^{\frac{1}{2}}}(X,E) + L_{X^{\frac{1}{2}}}(X,E) X^{\frac{1}{2}} = E, \tag{2.108}$$

where the last equality shows that the Fréchet derivative of the matrix square root $L_{X^{\frac{1}{2}}}(X,E)$ satisfies the Sylvester equation

$$X^{\frac{1}{2}} L + L X^{\frac{1}{2}} = E, \quad L := L_{X^{\frac{1}{2}}}(X,E). \tag{2.109}$$

The Kronecker representation $K_{X^{\frac{1}{2}}}$ can now be obtained by using the vectorization operator on both sides of the equation and rearrange the term to the form (2.106) which leads to

$$K_{X^{\frac{1}{2}}}(X) = \left[\left(\mathbb{1} \otimes X^{\frac{1}{2}}\right) + \left(X^{\frac{1}{2}T} \otimes \mathbb{1}\right)\right]^{-1}. \tag{2.110}$$

However, this straightforward approach has the disadvantage that the inverse of a $n^2 \times n^2$ matrix needs to be computed which has complexity $\mathcal{O}((n^2)^3) = \mathcal{O}(n^6)$. In addition to that, the inverse needs to be found explicitly which is not numerically stable in general.

The Sylvester equation (2.109), on the other hand, can be solved with $\mathcal{O}(n^3)$ operations via Schur transformation. We choose E^{ij}, the single-entry matrices having 1 at (i,j) and zero everywhere else, as a basis for $\mathbb{R}^{n\times n}$ and solve the Sylvester equation for each of the n^2 basis matrix elements. By that, the total complexity can be reduced to $n^2\mathcal{O}(n^3) = \mathcal{O}(n^5)$ and we avoid the potentially problematic explicit computation of inverses altogether.

We then obtain the final Kronecker form of the derivative by constructing its rows from the vectorized, transposed Fréchet derivatives:

$$\left(K_{X^{\frac{1}{2}}}\right)_{in+j,\cdot} = \operatorname{vec}\left(L_{X^{\frac{1}{2}}}(X, E^{ij})^T\right) \tag{2.111}$$

2.5.2 Derivative of the matrix logarithm

For the logarithm we follow the approach described by Al-Mohy et al [6] which is based on the differentiation of the Padé approximant to $\log(1+X)$. Since this is only applicable if the spectral radius of X is sufficiently small, the use of an inverse scaling and squaring technique based on the relation

$$\log(X) = 2\log(X^{\frac{1}{2}}) \tag{2.112}$$

is necessary.

Application of the chain rule leads to

$$L_{\log}(X, E_0) = 2\log\left(X^{\frac{1}{2}}, L_{X^{\frac{1}{2}}}(X, E_0)\right). \tag{2.113}$$

The second argument on the right hand side can again be written as solution $E_1 := L_{X^{\frac{1}{2}}}(A, E_0)$ of an Sylvester-type equation

$$X^{\frac{1}{2}}E_1 + E_1X^{\frac{1}{2}} = E_0. \tag{2.114}$$

Repeating the procedure s times results in

$$L_{\log}(X, E_0) = 2^s L_{\log}\left(X^{\frac{1}{2^s}}, E_s\right) \tag{2.115}$$

$$X^{\frac{1}{2^i}}E_i + E_iX^{\frac{1}{2^i}} = E_{i-1}, \quad i = 1, \ldots, s \tag{2.116}$$

where E_s is obtained by successively solving the set of Sylvester equations defined in the second line.

Finally, the Padé approximant of order m in its partial fraction form [19] is given by

$$r_m(X) = \sum_{j=1}^{m} \alpha_j^{(m)}(\mathbb{1} + \beta_j^{(m)}X)^{-1}X \tag{2.117}$$

where $\alpha_j^{(m)}, \beta_j^{(m)} \in (0,1)$ are the m-point Gauss-Legendre quadrature weights and nodes.

The derivative of (2.117) is then easily computed as

$$L_{r_m}(X, E) = \sum_{j=1}^{m} \alpha_j^{(m)}(\mathbb{1} + \beta_j^{(m)}X)^{-1}E(\mathbb{1} + \beta_j^{(m)}X)^{-1} \tag{2.118}$$

which leads to the final approximation of the matrix logarithm derivative,

$$L_{\log}(X, E) \approx 2^s L_{r_m}\left(X^{\frac{1}{2^s}} - \mathbb{1}, E_s\right). \tag{2.119}$$

For the implementation of (2.119) we use algorithm 5.1 from [6] with fixed $m = 7$. The Kronecker representation is then constructed as in the square root case.

Chapter 3

The MTVMT Library

The library which was developed in the course of this work is an easy-to-use, fast C++14 template library for TV minimization of manifold valued two- or three-dimensional images.

3.1 Capabilities

3.1.1 Supported Manifolds

- Real Euclidean space $\mathbb{R}^n$
- Sphere $S^n = \left\{x \in \mathbb{R}^{n+1} : \|x\| = 1\right\}$
- Special orthogonal group $SO(n) = \left\{R \in \mathbb{R}^{n\times n} : RR^T = \mathbb{1}, \det(R) = 1\right\}$
- Symmetric positive definite matrices $SPD(n) = \left\{S \in \mathbb{R}^{n\times n} : S = S^T, x^T Sx > 0 \; \forall x \in \mathbb{R}^n \setminus \{0\}\right\}$
- Grassmann manifold $Gr(n,p) = St(n,p)/O(p)$ (untested)

3.1.2 Data

- 2D and 3D images
- Input/Output via OpenCV integration supporting all common 2D image formats
- CSV input for matrix valued data
- Input methods for raw volume image data as well as the NIfTI [**?**] format for DT-MRI images
- Various methods to identify damaged areas for inpainting

3.1.3 Functionals

- isotropic(only possible for IRLS) or anisotropic TV functionals
- first order TV term
- weighting and inpainting possible

3.1.4 Minimizer

- IRLS
- PRPT

3.1.5 Visualizations

- OpenGL rotated cubes visualization for SO(3) images
- OpenGL ellipsoid visualization for SPD(3) images
- OpenGL volume renderer for 3D volume images

3.2 Design concepts

3.2.1 Goals

Performance

Since the core parts of the implementation are originally based on a Matlab prototype by Sprecher [17], [12] and [27], one of the most important goals was a faster implementation with a smaller memory footprint. On the test platform with two hyper-threaded 2.8GHz cores (Intel i5-2520) with AVX vector extensions the Matlab implementation froze for images larger than one Megapixel(MP). Hence, the C++ implementation should enable the algorithm to be tested in a much broader scope which is also closer to common picture sizes in image processing, especially since even smart phones today easily produce pictures in the Megapixel range. In additions to that, also other factors affecting the performance of the algorithm, such as cache locality and memory speed, can be investigated.

The main performance driver for this library is the multilevel-parallelization. Evidently, this does not include the formulation of the IRLS minimization algorithm itself, due to the fact that it is naturally an iterative method, but rather any possible subtask such as computation of the functional values, gradient and Hessian, for example. On top of that, we tried to maximize cache locality on the loop level and to free memory as soon as possible but keep data that is used very often and requires costly recomputation, like for example the IRLS weights.

In contrast to the original Matlab implementation, the computation of various quantities such as weights, first and second derivatives is not realized with tensor products any more. For Matlab, due the low speed of its internal loop constructs, the approach is justified but in a pure C++ implementation other factors are more important. One reason for the change is improved readability and maintainability of the code since tensor products usually tend to become very convoluted, especially for the manifolds with matrix representations. Also the modularization of the manifold class is not straightforward any more.
From a performance and parallelization perspective, contractions of tensor products are similar to matrix products and usually require some sort of blocking scheme for parallelization. In addition to that, because certain reshapes of the image container prior to the computation are necessary, the dimensions to be summed over are not necessarily contiguous in memory such that a high a cache utilization is more difficult to achieve.
Finally, in order to formulate certain operations as tensor products, temporary tensors of the correct dimensionality need to be created, which are actually not necessary.

Another measure that significantly reduces the memory footprint for the IRLS minimizer, especially for manifolds with matrix representations, is to only save gradient and Hessian in their local tangent space basis representation, such that the degrees of freedom correspond the intrinsic manifold dimension not the dimension of the embedding space. This also reduces the time to solve the sparse linear system.

Modularization and Extendability

In principle the programming paradigm in Matlab is still procedural resulting in a hierarchy of functions for the various tasks. Handling different types of manifolds then usually requires switch expressions in all functions that use manifold-specific functionality. Adding support for a new

manifold to the algorithm or modifying existing manifold functionality makes a modification of all switch cases necessary. There is no single point of change but many source files need to be edited.

For the MTVMT Library an object oriented and generic programming approach was chosen, which tries to model each variable component of the algorithm in a separate class, as independent of the other components as possible. For general information on C++ design paradigm see, for example [3] or [7]. Differences in each class are represented by specializations of their primary class template. The best example for that is the manifold class which has a specialization for every supported manifold type and due to the fact that the functions implemented in those class specializations are generally just functions of one or two elements of the manifold, they could also be used in other projects which require the same functionality.

Interfaces between classes are provided by giving classes higher in the hierarchy template parameters corresponding to lower classes: The class modelling the functional, for instance, has a manifold type template parameter, as described in more detail in Section 3.3. Like all other component classes, also the functional class can be extended by adding further specializations for other types of functionals, that include for example higher order terms or have different fidelity terms [31].

Those specializations also have the advantage that the code is just in one file, a single point of change to increase readability and maintainability.

3.2.2 Levels of parallelization

Parallelization takes place on two levels. The first one is shared memory multi-threading implemented with the OpenMP language extensions. In most cases this is realized using the so-called *pixel-wise kernels* of the VPP library, which makes it possible to map an arbitrary function on all pixels of a set of image containers: The function is called for each tuple of pixels having the same coordinates in their respective image. For the parallel execution each processor is assigned a batch of image rows. If the pixel-wise kernels are not applicable, for example if the needed subdomain of the image is too complicated, we use manual OpenMP loop parallelization. We implemented for example an own version of 3D pixel-wise kernels to keep the code compact.

The alternative to the above described tensor product implementation is to use pixel-wise kernels to parallelize any operation that requires iteration over an image container. For most computations, take for instance the case of computing derivatives, we only need the pixel and its next neighbor in a given dimension. For calculating the forward derivatives we just to have to call the pixel-wise kernel with two subimages of our current working image: One with the last slice (of the given dimension) missing and one with the first slice missing. At this point it must be noted that the concept of subimages does not involve any copies but just works by using different addressing schemes for the same data in the memory. The pixel-wise constructs are demonstrated in the following short listing 3.1 and further illustrated in 3.1:

Listing 3.1 Pixel-wise forward derivative computation

```
auto calc_first_arg_deriv = [&] (value_type& x, const weights_type& w, const ↘
    value_type& i, const value_type& n) { MANIFOLD::deriv1x_dist_squared(i, n, x); x *= ↘
    w; };

img_type YD1 = img_type(without_last_row);
vpp::pixel_wise(YD1, weightsY_ | without_last_row, data_.img_ | without_last_row, ↘
    data_.img_ | without_first_row) | calc_first_arg_deriv;
```

The advantage is that even though we evaluate some function for a pair of neighboring pixels which are not adjacent in memory, the parallel processing is still always row-wise. Since rows in row major languages like C++ are contiguous in memory we can avoid frequent memory access on distant locations and consequently avoid cache thrashing to a certain degree.

The second level of parallelization is instruction level parallelism, also known as *Single Instruction Multiple Data* (SIMD), which uses the processor's vector extensions (e.g. SSE, AVX, NEON). The

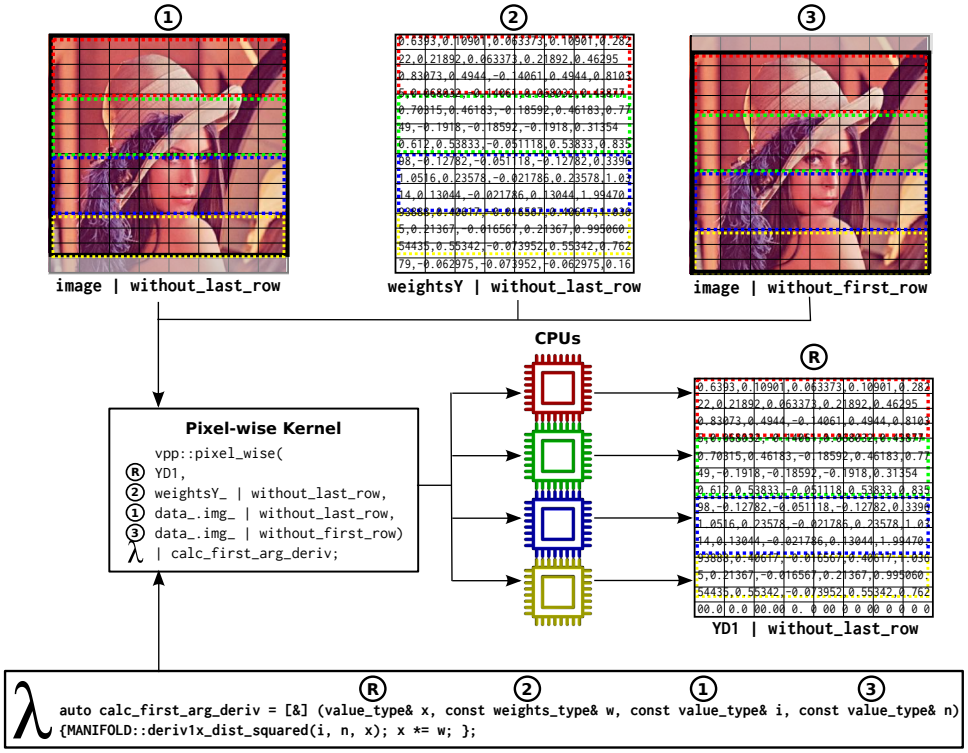


Figure 3.1: Parallel calculation of derivatives in y-direction and weighting using pixel-wise kernels. For each pixel position (i, j) in the three input pictures, ①, ② and ③, as well as the output picture ® the pixel-wise kernel creates a tuple $(R_{ij}, 2_{ij}, 1_{ij}, 3_{ij}) = (\mathrm{YD1}_{i,j}, \mathrm{weightY}_{i,j}, \mathrm{Image}_{i,j}, \mathrm{Image}_{i,j+1})$ which is than used to call the specified lambda function. Depending on the row number of the pixel, the calls are executed by different CPU cores.

CPU provides some additional special SIMD registers with increased size of usually 128 bits to 512 bits such that multiple integer or floating point variables fit inside. Then an arithmetic operation is simultaneously applied to all variables in the register (see figure 3.2) such that theoretically the amount of floating point operations is multiplied by the number of variables fitting in the registers.

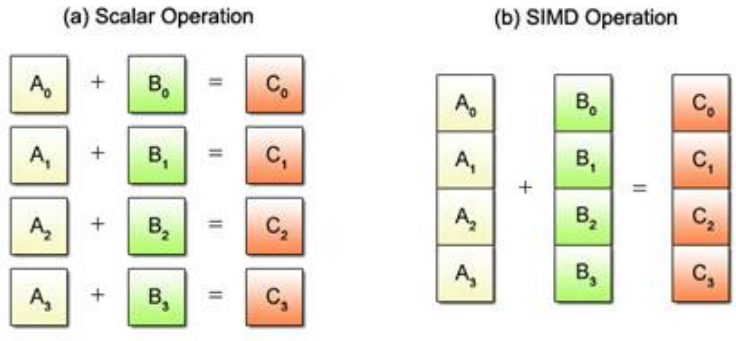


Figure 3.2: Instruction level parallelism using SIMD registers.

In order to really achieve this speedup the data must be aligned in memory which means the address of pixels in memory must always be multiple of the SIMD register size. Fortunately, that

issue is handled by the VPP and Eigen libraries enabling the compiler to perform the necessary vectorization optimizations.

3.2.3 C++ techniques

The MTVMT Library tries to take advantage of new C++11 and C++14 language features in order to speed up computations via compile-time optimizations and also make the code more compact and readable. The most important tools in that regard are lambda functions and variadic templates which are shortly described in the following section. For more details check for example [24].

Lambda functions

A lambda function is basically a locally defined function object, which is able to capture variables from the surrounding scope. The function can but needs not to be named. The corresponding Matlab language construct is an anonymous function or function handle, usually defined using the @ operator. The following listing shows the basic definitions and use cases of lambda functions:

Listing 3.2 Lambda functions

```
int init = 5;
std::vector<int> v {1, 2, 3, 4};

// C++11 lambda function for adding integers
// init is captured by reference
auto f = [&] (int a, int b) {return a + b + init;};

// C++14 generic argument lambda function
// init is captured by value
auto g = [=] (auto a, auto b) {return a + b + init;};

//Call named lambda functions
int d = f(8, 3);
double e = g(1.0f, 5);

// or directly pass anonymous lambda function as argument
std::transform(v.begin(), v.end(), v.begin(), [] (auto x) { ++x; });
```

In the MTVMT library lambda functions provide the connection between the static manifold methods and the pixel-wise kernels which apply them to the image containers. A typical case can be seen in the already introduced listing 3.1. Since lambda functions are only locally defined, in the scope where they are actually needed, one can avoid making the method list of the classes unnecessary long.

Variadic templates

With variadic templates it is possible to define functions which take a variable number of arguments. Obviously, this is also possible in other languages like Matlab or C with the most prominent example being the function printf. However, this is usually implemented using some list type (in C va_list), which adds additional overhead, whereas in C++ it is realized via a special kind of template metaprogramming technique which is recursive in nature. The recursion, in turn, is resolved at compile-time and leads to code that is actually equivalent to manually defining a function with the desired number of arguments and consequently, there is no additional runtime effort.

Listing 3.3 Variadic template example

```
// Recursion base case
template<typename T>
T sum(T v) {
    return v;
}

// Recursive template
template<typename T, typename... Args>
T sum(T first, Args... args) {
    return first + sum(args...);
}
```

The main application for this constructs in MTVMTL are the implementation of the Karcher mean, needed for the proximal point implementation, and MTVMTL's own version of the 3D pixel-wise kernels.

3.3 Components

In the following section we will briefly describe the different components of the library. For the manifold class this will be done in more detail to enable users to also use new or customized manifold classes. A general overview of all the components is provided in Figure 3.3.

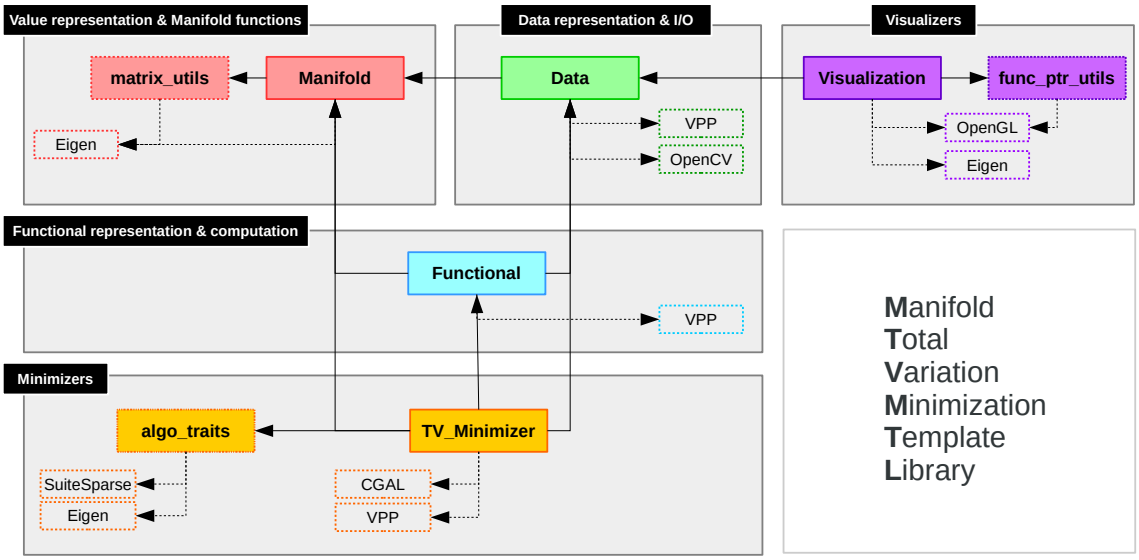


Figure 3.3: Overview of the class hierarchy and dependencies between the library components and also third party libraries

3.3.1 Manifold class

The manifold template class encapsulates all information and methods related to the differential geometric structure of the data. This enables the generic implementation of the functionality higher in the class hierarchy such as functional evaluations or minimizers. The primary template has the following parameters

```
// Primary Template
template <enum MANIFOLD_TYPE MF, int N, int P=0>
struct Manifold {
};
```

where MF is an enumeration constant to specify the type of the manifold, N denotes the dimension of the representation space and P the dimension of subspaces, as in the case of Grassmann manifolds. In order to add a new manifold one just has to implement a specialization of this primary template.

So far, the manifold class contains functionality necessary for TV minimization using either the IRLS or proximal point algorithm and furthermore some additional operations that are needed for supporting tasks like interpolation and smoothing. The class specializations are implemented using only static constants and methods: At no time is it necessary or desired to actually instantiate the class. The methods itself are usually unary or binary functions, with parameters and result all passed by reference to avoid copies. Since these methods are called very often, basically for every pair of neighboring pixels, they are all declared inline in order to support the compiler during the code optimization.

It is also possible to use these class in other projects requiring similar functionality like for instance when implementing a geodesic finite element solver.

In the following we will look at excerpts of the SPD implementation to illustrate which information and functionality a new manifold class need to provide and to give an overview of the available functions.

Static constants

```
static const MANIFOLD_TYPE MyType;        // SPD
static const int manifold_dim ;           // N*(N+1)/2
static const int value_dim;               // N*N

static const bool non_isometric_embedding;
```

The first constant just stores the manifold template template parameter introduced above, while manifold_dim and value_dim are the intrinsic dimensions of the manifold and of its embedding space, respectively. Finally, the Boolean constant is just a flag which tells the algorithm that special pre- and post-processing for interpolation is necessary.

Type definitions

To allow the generic formulation of the algorithms the manifold classes provide a mapping between the types of their values, derivatives, tangent bases and underlying scalar type and their actual representation as matrix and vector data types of the Eigen linear algebra library. Examples can be seen in the following listing:

```
// Scalar and value typedefs
typedef double scalar_type;
typedef double dist_type;
typedef Eigen::Matrix<scalar_type, N, N> value_type;
// ...

// Tangent space typedefs
typedef Eigen::Matrix <scalar_type, N*N, N*(N+1)/2> tm_base_type;
// ...

// Derivative Typedefs
typedef value_type deriv1_type;
typedef Eigen::sMatrix<scalar_type, N*N, N*N> deriv2_type;
typedef Eigen::Matrix<scalar_type, N*(N+1)/2, N*(N+1)/2> restricted_deriv2_type;
// ...
```

Static methods

Finally, the following methods are implemented for the manifold classes

Riemannian distance function and its derivatives

```
inline static dist_type dist_squared(cref_type x, cref_type y);
// First derivatives
inline static void deriv1x_dist_squared(cref_type x, cref_type y, deriv1_ref_type↘
    result);
inline static void deriv1y_dist_squared(cref_type x, cref_type y, deriv1_ref_type↘
    result);
// Second derivatives
inline static void deriv2xx_dist_squared(cref_type x, cref_type y, ↘
   deriv2_ref_type result);
inline static void deriv2xy_dist_squared(cref_type x, cref_type y, ↘
   deriv2_ref_type result);
inline static void deriv2yy_dist_squared(cref_type x, cref_type y, ↘
   deriv2_ref_type result);
```

Exponential and Logarithm map

```
template <typename DerivedX, typename DerivedY>
inline static void exp(const Eigen::MatrixBase<DerivedX>& x, const Eigen::↘
  MatrixBase<DerivedY>& y, Eigen::MatrixBase<DerivedX>& result);
inline static void log(cref_type x, cref_type y, ref_type result);

inline static void convex_combination(cref_type x, cref_type y, double t, ↘
  ref_type result);
```

The parameters of the exponential here are not the manifolds own typedefs but the base class of all Eigen matrix data types. The reason for using this construction is that the function can also be called with composite expressions (e.g. $XY + Z$) without a temporary copy. Most of the other functions are usually called with atomic expression only, hence there is no need to use this more complicated construction on a general basis.
The convex_combinations method computes the point z on the manifold reached by following a unit speed geodesic connecting the points x and y for a time t.

Karcher mean

```
inline static void karcher_mean(ref_type x, const value_list& v, double tol=1e↘
  -10, int maxit=15);
inline static void weighted_karcher_mean(ref_type x, const weight_list& w, const ↘
  value_list& v, double tol=1e-10, int maxit=15);

// Variadic templated version
template <typename V, class... Args>
inline static void karcher_mean(V& x, const Args&... args);
```

Implementations for finding the Karcher mean of an arbitrary number of points. The first version requires the points to be stored in a std::vector container while the second version is based on variadic templates and expects the arguments just as a comma separated list after the first argument, where the final result will be stored. Creating the list for the first version eventually requires copying and is consequently slower but has an overloaded version which allows to compute a weighted Karcher mean

Tangent plane basis, projector and interpolation

```
// Basis transformation for restriction to tangent space
inline static void tangent_plane_base(cref_type x, tm_base_ref_type result);
// Projection
inline static void projector(ref_type x);
// Interpolation pre- and postprocessing
inline static void interpolation_preprocessing(ref_type x);
inline static void interpolation_postprocessing(ref_type x);
```

The first function computes a basis of the tangent space at the point x and stores it in result as the columns of a matrix.
The projector, if defined for the given manifold, will project a point of the ambient embedding spacing onto the manifold. In the case of Euclidean space, where the embedding space and the manifold are identical, this function does nothing and will be optimized out by the compiler. Nevertheless, it must exist or programs will not compile.
Interpolation pre- and postprocessing is necessary for instance for the SPD manifold. Other manifolds must just provide an empty implementation.

3.3.2 Data class

The data class handles anything related to storage, input and output of two- or three-dimensional image data, as well as some support functions for detecting edges and damaged areas in a picture. In contrast to the manifold class, the data class needs to be instantiated such that a reference to the data object can be passed to any class which needs data access. In turn, in addition to the dimension of the picture, the data class takes a fully specialized manifold class type as a template parameter:

```
// Primary Template
template <typename MANIFOLD, int DIM >
class Data {
};
```

There are basically four multi-dimensional arrays stored in the data class: The original noisy image, the current working image and, if applicable, arrays storing the inpainting and edge weight information. For storage, the n-dimensional VPP [16] image container is used.

This image container class works very well together with the Eigen vector and matrix data types, provides a variety of expressive loop- and iterator constructs and also takes care of the alignment of the image data in memory, which is a prerequisite for the *Single Instruction Multiple Data* (SIMD) optimization and vectorization by the compiler. Since the memory management of the container is based on std::shared_pointer it is also very easy to efficiently access subimages or slices of an image without any copies.

The most common input method for 2D and 3D are summarized in the following code snippet:

```
 // 2D Input functions
void rgb_imread(const char* filename);  // for R^3
void rgb_readBrightness(const char* filename); // for R
void rgb_readChromaticity(const char* filename); // for S^2
void readMatrixDataFromCSV(const char* filename, const int nx, const int ny);

// Synthetic SO/SPD picture
void create_nonsmooth_son(const int ny, const int nx);
void create_nonsmooth_spd(const int ny, const int nx);

//3D Input functions
void rgb_slice_reader(const char* filename, int num_slides);
void readMatrixDataFromCSV(const char* filename, const int nz, const int ny, const int↘
    nx);
void readRawVolumeData(const char* filename, const int nz, const int ny, const int nx)↘
    ;
```

The purpose and usage of most of these methods is self-explanatory. The CSV readers expect the data to be a linear list of pixels, where the components of each pixel are comma-separated and row-wise flattened, such that each line of the input file contains exactly one pixel. The order of the list is also row-wise for 2D or slice- then row-wise for 3D images, respectively.
The slice reader reads a series of images, following the filename scheme filenameX.ext, where X is the number of the slice to be read into an image cube at z-coordinate X.

3.3.3 Functional class

In addition to fully specialized Manifold and Data class types (third and fourth template parameters), there are three further template parameters that must be specified by the library user. The first one is the order of the functional which refers to the order of the highest differential operator in the TV term of the functional. So far, only first order functionals are implemented which would correspond to setting ord=FIRSTORDER in the primary template shown below

```
//Primary Template
template <enum FUNCTIONAL_ORDER ord, enum FUNCTIONAL_DISC disc, class MANIFOLD, class ↘
    DATA, int DIM=2>
class Functional{
};
```

The second template parameter disc determines whether the isotropic or the anisotropic version is to be used. Please note that for the proximal point algorithm only anisotropic is available. Finally, the last parameter specifies the dimensionality of the data.
The main purpose of the functional class is to provide methods for the computation of all functional-related quantities, such as evaluation of the functional, its gradient, Hessian and construction of a local basis of the tangent spaces. That also means that in the IRLS case the functional class stores the sparse linear system that needs to be solved in each Newton step.

For users of the library, the most important methods are those for setting the λ and ϵ^2 parameters.

```
inline param_type getlambda() const { return lambda_; }
inline void setlambda(param_type lam) { lambda_=lam; }
inline param_type geteps2() const { return eps2_; }
inline void seteps2(param_type eps) { eps2_=eps; }
```

Should it be necessary, it is also possible to access some of the stored quantities directly using

```
// Evaluation functions
result_type evaluateJ();
void  evaluateDJ();
void  evaluateHJ();

void updateTMBase();

inline const gradient_type& getDJ() const { return DJ_; }
inline const sparse_hessian_type& getHJ() const { return HJ_; }
inline const tm_base_mat_type& getT() const { return T_; }
```

The functions in lines 3, 4 and 6 merely trigger a recomputation while the last three functions return references to these quantities. evaluateJ() returns the functional value and triggers the recomputation of the weights.

3.3.4 TV Minimizer class

For the TV Minimizer class it makes sense to consider the IRLS and proximal point implementation separately. The primary template is shown in the following code snippet.

```
//Primary Template
template <enum ALGORITHM AL, class FUNCTIONAL, class MANIFOLD, class DATA, enum ↘
    PARALLEL PAR=OMP, int DIM=2>
class TV_Minimizer{
};
```

As in the previous cases we have to provide fully specialized manifold, data and also functional types. Again, the last parameter specifies the dimension of the data. Of the remaining two PAR has the default value OMP which specifies the method of parallelization, in this case the OpenMP language extensions. Other methods, including just serial execution, could be added later. The remaining template parameter, AL specifies the minimizer to be used and can take the values IRLS or PRPT (Proximal point).

For IRLS, we have the following public class interface

```
void first_guess();                  // First guess for inpainting
void smoothening(int smooth_steps); // Simple averaging box filter
newton_error_type newton_step();     // perform one newton step
void minimize();                     // full minimization

// Getters and Setters for parameters
void setMax_runtime(int t) { max_runtime_ = t; }
void setMax_irls_steps(int n) { max_irls_steps_ = n; }
void setMax_newton_steps(int n) { max_newton_steps_ = n; }
void setTolerance(double t) {tolerance_ =t; }

int max_runtime(int t) const { return max_runtime_; }
int max_irls_steps(int n) const { return max_irls_steps_; }
int max_newton_steps(int n) const { return max_newton_steps_; }
int tolerance(double t) const { return tolerance_; }
```

while for proximal point we have

```
use_approximate_mean(bool u) { use_approximate_mean_ = u; } // turn mean approximation↘
    on/off
void first_guess();                                          // First guess for ↘
    inpainting

void updateFidelity(double muk);                             // Update Fidelity part
void updateTV(double muk, int dim, const weights_mat& W);   // Update TV part

void geod_mean();           // Calculate geodesic mean
void approx_mean();         // approximate mean using convex combinations
```

```
void prpt_step(double muk); // perform one proximal point step
void minimize();            // full minimization

// Getters and Setters for parameters
void setMax_runtime(int t) { max_runtime_ = t; }
void setMax_prpt_steps(int n) { max_prpt_steps_ = n; }

int max_runtime(int t) const { return max_runtime_; }
int max_prpt_steps(int n) const { return max_prpt_steps_; }
```

3.3.5 Visualization class

This class provides visualizations of 3D volume data and so far SO(3) and SPD(3) visualizations by cubes and ellipsoids. If these are to be used in user code it is necessary to link against OpenGL, GLUT and GLEW libraries, which is explained in more detail in section 3.4.3. The visualization classes have the following primary template.

```
//Primary Template
template <enum MANIFOLD_TYPE MF, int N, class DATA, int dim=2>
class Visualization{
};
```

The class methods that are relevant to users of the library are summarized here

```
void saveImage(std::string filename);
void GLInit(const char* windowname);

void paint_inpainted_pixel(bool setFlag);
```

The important function here is GLInit which initializes the rendering of the data. If one intends to also save the image, on has to specify a filename using saveImage *before* calling GLInit. Finally, paint_inpainted_pixel just sets a flag which decides whether inpainted pixels are not painted at all (setFlag = false, default value) or if they are visualized with the value they have at the time of rendering. Usually one wants to set this to true after the minimization to show the results.
A complete example is presented in section 3.4.4

SO(3) Visualization

For the visualization of SO(3) data we just consider a unit volume cube centered at the origin of $\mathbb{R}^3$ with its front face normal vector parallel to the y-axis. Then the rotation matrix representing the SO(3) element is applied to the cube. To break the $O_h \simeq S_4 \times S_2$ symmetry of the cube, we give a different color to every face.

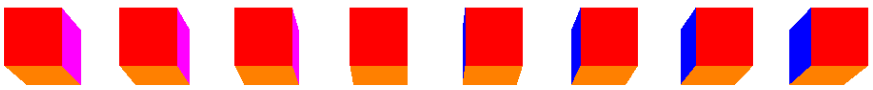


Figure 3.4: SO(3) Visualization as oriented, colored cubes

SPD(3) Visualization

For SPD(3) matrices we have six degrees of freedom, which in the case of DT-MRI pictures correspond to the diffusion coefficients in different directions. Those can be visualized by ellipsoids using three degrees of freedom for their orientation in space and the remaining three for the lengths of its semi-axis.
Starting with the unit sphere centered at the origin, we compute eigenvectors and eigenvalues for every SPD matrix. Due to the SPD property a full basis of eigenvectors with positive eigenvalues always exists. The diagonal matrix formed by the vector of eigenvalues is applied as a scaling transformation of the coordinate axis. The matrix whose columns are the computed eigenvectors can then be interpreted as a rotation (or principal axis transformation of the ellipsoid).

To avoid large size difference and overlaps between the ellipsoids we also normalize the eigenvalues using the mean diffusivity μ defined by

$$\mu = \frac{1}{3}\sum_{i=1}^{3}\lambda_i. \tag{3.1}$$

Finally, the color is defined by normalizing the largest eigenvector, called the principal direction, and mapping its coordinates to the RGB color space, such that clusters of similar orientations can be more easily visually distinguished.

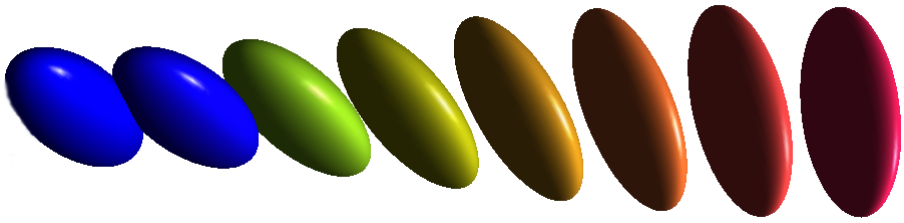
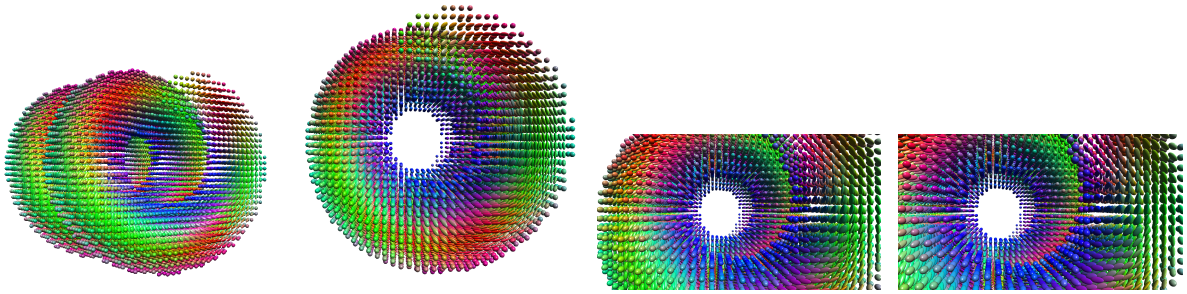


Figure 3.5: SPD(3) Visualization as oriented, colored ellipsoids

In the case of 3D SPD images the rendering window also provides some controls over the view. The up and down arrays keys can be used to zoom in and out of the picture, left and right keys pivot the camera and with the s key the image is saved using the filename specified before.



(a) Side View (b) Top View (c) Closeup (d) Inside

Figure 3.6: Example of the 3D data Visualization of SPD(3) images from different viewpoints.

3D Volume image rendering

The volume image renderer just transforms the data to a 3D texture which is then mapped onto a cube rotating about the z-axis. Plasticity is created by setting the alpha channel of each displayed voxel to the intensity value of the corresponding data voxel, such that dark areas are more transparent. The following 3.7 shows the rendered volume from different angles.

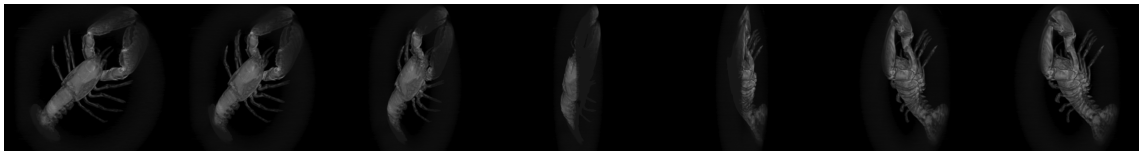


Figure 3.7: 3D Volume image using texture based rendering. The 'helix' synthetic tensor data set, produced with the tend program in the Teem toolkit[1]

3.3.6 Utility functions

Algorithm traits

The algorithm traits class contains standard values for the IRLS algorithm, like the number of IRLS iterations, number of Newton iterations and maximal runtime. It also contains the standard solver for the linear system. If the solver needs to be switched it must be changed in this file. All other parameters can be changed using the methods provided by the TV minimizer class.

Matrix functions

In the file matrix_utils.hpp additional matrix functions not included in the Eigen library are implemented. So far, these are only the methods for the computation of the Fréchet derivatives of matrix square root and logarithm, as well as their Kronecker representations.

Function pointers utilities

Located in the file func_ptr_utils.hpp are some auxiliary functions needed to transform pointers to class member functions to plain C function pointers. The latter are required by the OpenGL and GLUT library API.

3D pixel-wise kernels

The 3D version of the pixel-wise kernels along with useful tools based on them, for copying or filling 3D images.

3.4 Using MTVMTL

3.4.1 Prerequisites

The main dependencies of MTVMT are the *Eigen* C++ template library for linear algebra and the *Video++* video and image processing library. Those libraries, as well as MTVMTL's core functionality are provided as header-only libraries. There are, however, some additional static libraries that are recommended to speed up the computation, enable easy I/O or which are needed for visualization of the results. To administer all these different parts and because the header-only libraries require additional compiler flags for the code optimization, MTVMTL also relies on the *CMake* installation tool for installation and compilation of user code using MTVMTL.

The following lists shows the needed packages for the usage of MTVMTL:

- CMake ($\geq$ 2.8.0)
- g++ ($\geq$ 4.9.1), any C++14 compatible compiler should also be possible but is untested.
- Eigen ($\geq$ 3.2.5)
- Video++

Recommended are also the following packages. They are needed if any of the described extended functionality needs to be used.

- OpenCV ($\geq$ 2.4.9), for image input and output, edge detection for inpainting
- CGAL ($\geq$ 4.3), for first guess interpolation during inpainting
- OpenGL ($\geq$ 7.0), for visualizations of SPD, SO and any 3D data
- SuiteSparse ($\geq$ 4.2.1), faster parallel sparse solver for the linear system in the IRLS algorithm

Some care must be taken with the version recommendations. Usually there are no compatibility issues if only the minor software version changes but for major version updates it must be checked whether the new version's API is still backwards compatible.

3.4.2 Installation

3.4.3 Compilation of own projects using CMake

For the compilation of user code using CMake a file with the name CMakeLists.txt must be provided in the same directory as the user code. This file contains all the information about the locations of header files and external library code as well as compiler optimization flags. In listing 3.4 an example is provided for the compilation of a user application my_executable with a single source file mysource.cpp. The example is minimal in the sense that it is only for the compilation of a single executable and maximal in the sense that it links the executable against any possible external library used by MTVMTL.

The most important lines for the user are the last two. In the first of these we first add an executable by providing its name (my_executable) and the source file(s) it depends on (mysource.cpp). Next on must specify the external libraries our executable is linked against using the target_link_libraries() command. It expects the executable as the first parameter and then a space-separated list of all target libraries.

Listing 3.4 Example CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8)

list(APPEND CMAKE_MODULE_PATH "${CMAKE_CURRENT_SOURCE_DIR}/cmake")

find_package(OpenGL REQUIRED)
find_package(GLUT REQUIRED)
find_package(GLEW REQUIRED)
include_directories( ${OPENGL_INCLUDE_DIRS}  ${GLUT_INCLUDE_DIRS} )

find_package(OpenCV REQUIRED)
find_package(CGAL REQUIRED)
include(${CGAL_USE_FILE})

find_package(Cholmod REQUIRED)
find_package(SuperLU REQUIRED)

include_directories(${CMAKE_CURRENT_SOURCE_DIR}/.. /usr/include/superlu /usr/include/↘
    eigen3 $ENV{HOME}/projects/iod)
add_definitions(-std=c++14 -g -fopenmp)
add_definitions(-Ofast -march=native)
add_definitions(-DNDEBUG)

add_executable(my_executable_mysource.cpp)
target_link_libraries(my_executable  gomp ${OpenCV_LIBS} ${CGAL_LIBRARIES} ${↘
    CHOLMOD_LIBRARIES} ${SUPERLU_LIBRARIES} ${OPENGL_LIBRARIES} ${GLUT_LIBRARY} ${↘
    GLEW_LIBRARIES})
```

For the actual compilation, we create a separate directory called build and change to this directory.

```
mkdir build
cd build
```

Next we call cmake providing the source directory which also contains the CMakeLists.txt file as argument.

```
cmake ../src/
```

If everything was configured correctly, the output should look similar to the following.

```
-- The C compiler identification is GNU 4.9.2
-- The CXX compiler identification is GNU 4.9.2
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
```

```
-- Detecting CXX compile features - done
-- Found OpenGL: /usr/lib64/libGL.so
-- Found GLUT: /usr/lib64/libglut.so
-- Found GLEW: /usr/include
-- Build type: Release
-- USING CXXFLAGS = '-march=corei7 -mtune=native -O2 -pipe -msse3 -msse4 -mcx16 -msahf -mpopcnt -frounding-math -O3 -DNDEBUG'
-- USING EXEFLAGS = ' -Wl,-O1 -Wl,--as-needed '
-- Targetting Unix Makefiles
-- Using /usr/bin/c++ compiler.
-- Requested component: MPFR
-- Requested component: GMPXX
-- Requested component: GMP
-- Found CHOLMOD: /usr/include
-- Found SUPERLU: /usr/include/superlu
-- Configuring done
-- Generating done
-- Build files have been written to: [path-to-build-folder]/build
```

Finally, we can type

```
make my_executable
```

to build the program.

3.4.4 Tutorial and typical use cases

The basic process of using the library is to explicitly specify the necessary template parameters for all needed components. For the sake of compactness and readability this should be done using typedefs. In the next step one can then instantiate the classes and start implementing.

Image denoising, vectorial color model

As a first example we show the denoising of a simple color picture using the IRLS minimizer. In a first step the necessary classes need to be included. For the sake of shortening the code we also switch to the tvmtl namespace of the library.

Listing 3.5 Inclusion of library headers

```
#include "../core/algo_traits.hpp"
#include "../core/tvmin.hpp"

using namespace tvmtl;
```

Next, we specify the manifold type and data type we want to use, in this case Euclidean $\mathbb{R}^3$ and a corresponding 2D image container

Listing 3.6 Specification of manifold and data type

```
typedef Manifold< EUCLIDIAN, 3 > mf_t;
typedef Data< mf_t, 2> data_t;
```

Note that the data type must be specified using the *fully* specialized manifold class type we defined in the line before.
Our data type is now ready for work such that we can read the input data in the next few lines.

Listing 3.7 Initialization and input of image data

```
data_t myData=data_t();          // Creating the data object
myData.rgb_imread(filename);     // Reading an image file, filename is a const char*
```

After the data object is ready we must specify the functional we want to use, which we choose to be first order TV, isotropic and 2D. Again, also the fully specialized manifold and data class types need to be given as template parameters. The last template parameter, the dimension of the data, has default value 2 and can also be omitted in this case.

Listing 3.8 Defining the functional and setting parameters

```
typedef Functional<FIRSTORDER, ISO, mf_t, data_t, 2> func_t;

func_t myFunc(lambda, myData);   // Creation of the functional object
myFunc.seteps2(1e-10);           // Specify the epsilon parameter
```

For the instantiation of the functional we need to pass the λ for our functional as well as our newly created data object. The seteps2 method sets the value of ϵ^2 for the reweighting computation. In case of the proximal point algorithm it should be set to zero.

Listing 3.9 Choosing the minimizer, smoothing and minimization

```
typedef TV_Minimizer< IRLS, func_t, mf_t, data_t, OMP, 2> tvmin_t;

tvmin_t myTVMin(myFunc, myData);// Creation of minimizer object

myTVMin.smoothening(5);          // smoothing to obtain better starting value
myTVMin.minimize();              // Starts the minimization
```

Finally we choose the minimizer we want to use, in this case IRLS, and pass functional, manifold and data types as template parameters. The OMP parameter is not fully implemented yet and is supposed to provide choice between different parallelization schemes or also completely serial computation. The last parameter again has default value 2 and describes the dimension of the data. The complete listing of this example can be found in Appendix A.

Colorization using color inpainting

In the following we show a more complicated example: Recolorization of an image where most($\approx$99%) *color* information has been removed. This means that this problem is defined on the product manifold $S^2 \times \mathbb{R}$. Optimization, however, will only take place on S^2 while the $\mathbb{R}$ data part is only needed to obtain edge information. We also use three auxiliary functions (removeColor, DisplayImage, recombineAndShow) here that are not shown in the code snippets but will be included in the full listing in Appendix A. This time, we start by also obtaining some of the minimization parameters from the command line:

Listing 3.10 Include library files and read parameters from standard input

```
#include <iostream>
#include <string>
#include <cmath>

#include <opencv2/highgui/highgui.hpp>
#include "../core/algo_traits.hpp"
#include "../core/data.hpp"
#include "../core/functional.hpp"
#include "../core/tvmin.hpp"

#include <vpp/vpp.hh>
#include <vpp/utils/opencv_bridge.hh>

using namespace tvmtl;

int main(int argc, const char *argv[])
{
    if (argc < 3){
        std::cerr << "Usage : " << argv[0] << " image [lambda] [threshold]" << std::↘
            endl;
        return 1;
    }

    double lam=0.01;
    double threshold=0.01;

    if(argc == 4){
        lam=atof(argv[2]);
        threshold=atof(argv[3]);
    }

    std::string fname(argv[1]);

    //...
```

```
}
```

Here threshold defines the percentage of color information that remains in the picture.
In the next step, we again make the necessary type definitions for manifold and data classes and create our data objects.

Listing 3.11 Manifold and Data class type definitions and instantiation

```
// typedefs
typedef Manifold< SPHERE, 3 > spheremf_t;  // S^2
typedef Manifold< EUCLIDIAN, 1 > eucmf_t;  // R

typedef Data< spheremf_t, 2> chroma_t;  // Chromaticity part
typedef Data< eucmf_t, 2> bright_t;     // Brightness part

// Instantiation
chroma_t myChroma=chroma_t();
bright_t myBright=bright_t();
```

When the data containers are ready we need to read the input picture, extract color and brightness information and store it in the respective objects. This problem is basically a color inpainting problem but we do not want the reconstructed color to blur across edges in the picture. This can be solved by making use of the edge weights array that is stored together with the image. We will detect edges in the brightness part of the picture and and use those edges in the chromaticity denoising procedure.
Finally, we will remove the color in the following way: Create a random inpainting matrix where the probability a certain pixel is set to false is given by the threshold variable and then replace every RGB pixel by the mean of its three color components (those pixels are basically gray scale then).
The necessary steps are shown in the next listing

Listing 3.12 Color and brightness input, edge detection and color removal

```
myBright.rgb_readBrightness(argv[1]);    // Extract brightness from filename argv[1]
myBright.findEdgeWeights();              // Detect edges and store in matrix

myChroma.rgb_readChromaticity(argv[1]); // Extract chromaticity from filename argv[1]
myChroma.inpaint_=true;                  // Turn inpainting on
myChroma.setEdgeWeights(myBright.edge_weights_); // Initiliaze chromaticity part edges↘
    with brightness part edges
myChroma.createRandInpWeights(threshold);   // Create random inpainting matrix
removeColor(myChroma, myBright);            // Remove color

// Recombine chromaticity and brightness and show the colorless image
recombineAndShow(myChroma, myBright, "colorless_"+fname, "Colors removed Picture");
```

The next part works almost exactly as in the last example. We define functional, set its parameters, then define the minimizer. The only difference is that we have to run first_guess before the minimization.

Listing 3.13 Functional and minimizer definition, first guess and minimization

```
typedef Functional<FIRSTORDER, ISO, spheremf_t, chroma_t> cfunc_t;
typedef TV_Minimizer< IRLS, cfunc_t, spheremf_t, chroma_t, OMP > ctvmin_t;

cfunc_t cFunc(lam, myChroma);          // create functional object
cFunc.seteps2(1e-10);                  // set eps^2 parameter

ctvmin_t cTVMin(cFunc, myChroma);      // create minimizer object
cTVMin.first_guess();                  // first guess

std::cout << "Start TV minimization..." << std::endl;
cTVMin.minimize();

// Recombine Brightness and Chromaticity parts of recolored Picture
recombineAndShow(myChroma, myBright, "recolored_"+fname, "Recolored Picture");
```

Some visual results of the above code are also shown in Section 4.1.4.

3D DT-MRI data denoising and visualization

As a final example we choose a more complicated manifold, SPD(3) in this case, as well as 3D data to demonstrate the use of the visualization classes. Moreover, we use the proximal point algorithm in this example. The CSV reader just reads a list of pixels where the numerical values comprising the pixel are stored comma-separated, one pixel per line. The CSV file has no header such that we provide the dimensions as command line parameters.

Listing 3.14 Initialization

```
#include <iostream>
#include <cstdlib>
#include <string>
#include <sstream>

#include "../core/algo_traits.hpp"
#include "../core/data.hpp"
#include "../core/functional.hpp"
#include "../core/tvmin.hpp"
#include "../core/visualization.hpp"

int main(int argc, const char *argv[])
{
    int nz, ny, nx;
    nz = std::atoi(argv[2]);
    ny = std::atoi(argv[3]);
    nx = std::atoi(argv[4]);

    std::stringstream fname;
    std::string nfname;
    fname << "dti3d" << nz << "x" << ny << "x" << ny << ".png";
    nfname = "noisy_" + fname.str();

    // ...

    return 0;
}
```

Since the meaning of the individual components should be clear by now we make all the necessary type definitions at once in the next listing

Listing 3.15 Type definitions, Visualization type

```
using namespace tvmtl;

typedef Manifold< SPD, 3 > mf_t;
typedef Data< mf_t, 3> data_t;
typedef Functional<FIRSTORDER, ANISO, mf_t, data_t, 3> func_t;
typedef TV_Minimizer< PRPT, func_t, mf_t, data_t, OMP, 3 > tvmin_t;
typedef Visualization<SPD, 3, data_t, 3> visual_t;
```

The only innovation is the aforementioned Visualization class. The first 3 in its template parameter list is the embedding dimension of the manifold and the last 3 denotes the dimension of the data. Note that we needed to specify it for the functional and minimizer classes as well because the default value is 2. The remaining parameters specify the manifold type via an enumeration constant (in the same way one specifies it for the Manifold class) and the data type via a fully specialized data class type.
Before we start the minimization we want to display the original noisy data and eventually save it to a file once we have found a nice viewing angle in the rendering window. The necessary steps are as follows:

Listing 3.16 Data input and displaying the noisy data

```
data_t myData = data_t();    // Create data object
myData.readMatrixDataFromCSV(argv[1], nz, ny, nx); // Read from CSV file

visual_t myVisual(myData);  // Create visualization object
myVisual.saveImage(nfname); // Specify file name to save a screenshot

std::cout << "Starting OpenGL-Renderer..." << std::endl;
myVisual.GLInit("SPD(3) Ellipsoid Visualization"); // Start the Rendering
```

In the last step we create functional and minimizer class, perform the minimization and display the denoised data again.

Listing 3.17 Minimization and final rendering

```
double lam=0.7;
func_t myFunc(lam, myData); // Functional object
myFunc.seteps2(0);  // eps^2 should be 0 for PRPT

tvmin_t myTVMin(myFunc, myData); // Minimizer object

std::cout << "Start TV minimization.." << std::endl;
myTVMin.minimize();

std::string dfname = "denoised(prpt)_" + fname.str();
myVisual.saveImage(dfname); // Specify name for denoised image

std::cout << "Starting OpenGL-Renderer..." << std::endl;
myVisual.GLInit("SPD(3) Ellipsoid Visualization"); // Render
```

The resulting picture for this example are also shown in 4.3.3.

Chapter 4

Applications and Numerical Experiments

In the first part of the chapter we apply the algorithms to a variety of problems in image processing, computer vision, medical imaging and related fields. The second part will be a comparison of the performance of the two implemented minimizer and we close with an numerical experiment investigating the dependence of the solution on the changes in the original picture. This is a first step in extending the IRLS algorithm towards recursive splitting into subdomain.

As briefly mentioned in the previous section, the test platform is a Linux machine with two hyper-threaded 2.8GHz cores Intel i5-2520 (thus a total of four hardware threads) with AVX vector extensions and 8 GB RAM.

4.1 Image denoising

The very basic application of the algorithm is of course denoising of common 2D grayscale or color pictures. For grayscale pictures the TV minimization is performed over the Euclidean manifold $M = \mathbb{R}$, while for color pictures, as already explained in the introduction, we have either $M = \mathbb{R}^3$ for the linear-vectorial model or $M = S^2 \times \mathbb{R}$ for the non-linear chromaticity-brightness model.

4.1.1 Grayscale

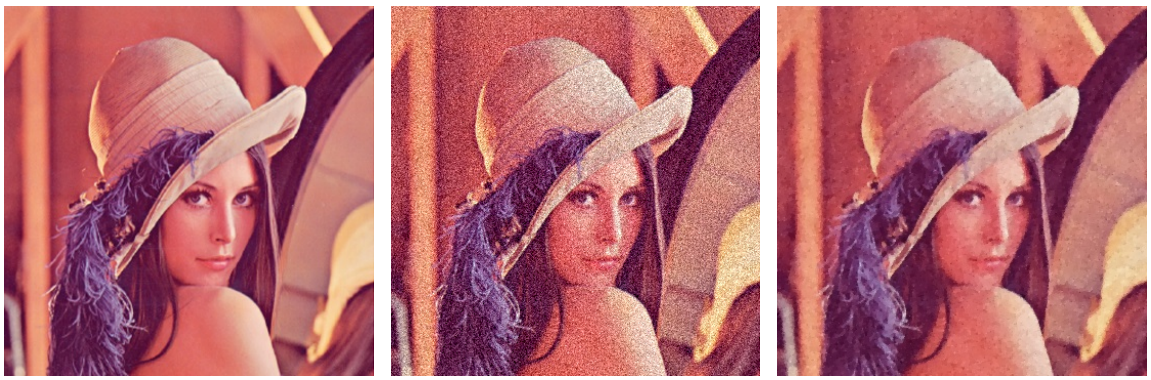
As introductory example and for the sake of completeness, we show in Figure 4.1 results from denoising a grayscale image.

(a) Original (b) Noisy (c) Denoised

Figure 4.1: Denoising of a grayscale image taking values in the manifold $\mathbb{R}$ (a) Original image "Cameraman.bmp", 256×256 px, 8 bit depth (b) Component-wise Gaussian noise $\mu = 0$, $\sigma = 0.01$ added (c) Denoised, IRLS with $\lambda = 0.09$, 5 IRLS steps, 1 newton steps per IRLS step

4.1.2 Color

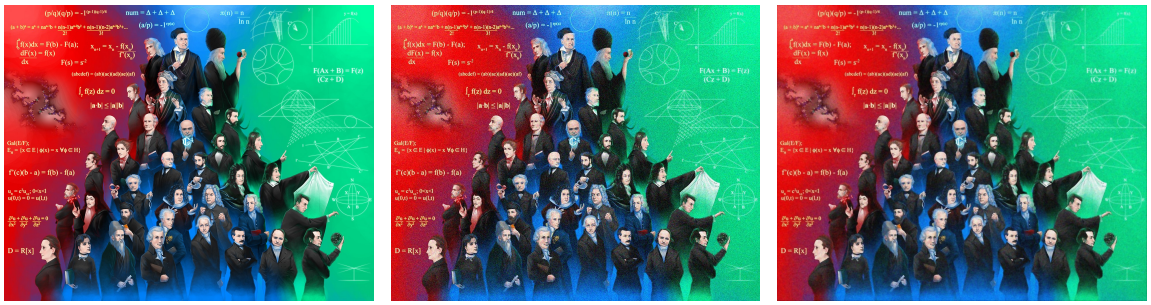
In this example we perform the TV minimization of color images using the two different color models. In Figure 4.2 we show results for among the image processing community well-known *Lena* picture, which is rather small in size. Minimization in the linear-vectorial color model using 5 IRLS iterations with one Newton step per reweighting is completed within 9.2 seconds.



(a) Original (b) Noisy (c) Denoised

Figure 4.2: Denoising of a color images using the linear vectorial color model which corresponds to the manifold $\mathbb{R}^3$ (a) Original image "Lena.jpg", 361×361 px, 8 bit color depth (b) Component-wise Gaussian noise $\mu = 0$, $\sigma =?$ added (c) Denoised, IRLS with $\lambda = 0.1$, 5 IRLS steps, 1 newton steps per IRLS step

Next, using the same model and parameters we denoise a different image with a size already in the megapixel range. The needed time, however, is with 272.6 seconds quite high. The result can be seen in Figure 4.3.



(a) Original (b) Noisy (c) Denoised

Figure 4.3: Denoising of color images using the linear vectorial color model which corresponds to the manifold $\mathbb{R}^3$ (a) Original image "mathematicians.jpg", 1280×1024 px, 8 bit color depth (b) Component-wise Gaussian noise $\mu = 0$, $\sigma =?$ added (c) Denoised, IRLS with $\lambda =?$, 5 IRLS steps, 1 newton steps per IRLS step

Finally, in Figure 4.4 we denoise a third picture using the chromaticity-brightness model. Here minimization over the product manifold $S^2 \times \mathbb{R}$ is performed by denoising the chromaticity(S^2) and the brightness($\mathbb{R}$) separately, which has the added advantage of more fine-grained control over the process because two λ parameters can be chosen separately for each part, too.

(a) Original (b) Noisy (c) Denoised

Figure 4.4: Denoising of a color images using the linear vectorial color model which corresponds to the manifold $S^2 \times \mathbb{R}$ (a) Original image "mathematicians.jpg", 284×177 px, 8 bit color depth (b) Component-wise Gaussian noise $\mu = 0$, $\sigma =?$ added (c) Denoised, IRLS with $\lambda_{\mathbb{R}} = 0.1$, 5 IRLS steps, 1 newton steps per IRLS step

4.1.3 Inpainting

We consider a damaged picture were a considerable part of the picture has been overpainted with blue color. In the first step we detect the damaged region which in this case is done via a simple color selector (e.g. all pixels with a blue value larger than 0.95). In principle many other selection methods known from common raster graphic editors could be implemented here as well.
Next, the a first guess is calculated using scattered linear interpolation and lastly the TV minimization itself is performed. The process is summarized in Figure 4.5.

(a) Original (b) Damaged (c) First guess (d) Restored

Figure 4.5: Inpainting of a color image using the linear vectorial color model which corresponds to the manifold $\mathbb{R}^3$ (a) Original image "Pepper.png", 359×361 px, 8 bit color depth (b) Damaged by overpainting with blue color (c) First guess via component-wise scattered interpolation (d) Restored, IRLS with $\lambda = 0.1$, 5 IRLS steps, 1 newton steps per IRLS step

4.1.4 Recolorization

Colorization, also known ss color inpainting, because it is basically just a special case of inpainting is performed in the next example. Here the picture is not necessarily noisy but we assume only the brightness of each pixel is known, while the chromaticity is known only for a low ratio $r = 0.01$ of all pixels. Note that this splitting implies that inpainting and TV minimization takes place only on S^2.

As in the previous example, we first have to detect all damaged, i.e. non-colored, pixels to inpaint. Again we use scattered interpolation to obtain the first guess which is depicted in Figure 4.5 (c). One can observe that the color runs over the edges of the leaves. To avoid this we need to detect the edges in the brightness part using the Canny edge detector [11], for example, and set the edge weights for the chromaticity part accordingly. As a result, we indeed obtain sharp and clear edges in the final result 4.6d (d).

(a) Original (b) Colors removed (c) First guess (d) Recolored

Figure 4.6: Recolorization using color inpainting in the Chromaticity-Brightness color model, corresponding to $S^2 \times \mathbb{R}$ (a) Original image "Basil.jpg", 300×179 px, 8 bit color depth (b) Image with a ratio of approximately 0.01 remaining colored pixels (c) First guess via component-wise scattered interpolation (d) Recolored, IRLS with $\lambda = 0.01$, 5 IRLS steps, 1 newton steps per IRLS step

4.1.5 Volume images

We conclude the picture section with an example of a 3D volume image as they might occur in medical imaging from magnetic resonance imaging (MRI) or computed tomography. In this demonstration, however, we chose the example of the so-called *Boston teapot*, taken from a volume image library [**?**] and added component-wise Gaussian noise. The image represents only intensity values, hence minimization is performed over $\mathbb{R}$. The results are shown in Figure 4.7a.

For this picture we used the proximal point algorithm, because the memory and computational requirements of the IRLS for a picture of this size are very high: In section 2.3.4 it was shown that the dimension of the sparse linear system is $\dim(M)XYZ$ which in this case amounts to 1.1×10^7, which is the length of the gradient while the Hessian will contain 7.7×10^7 non-zero-entries. The solution of a sparse linear system of that size is computationally very demanding while in comparison the geodesic averaging and Karcher mean calculations simplify to mostly vectorized addition and subtraction operations on a simple manifold like $\mathbb{R}$.

(a) Original (b) Noisy (c) Denoised

Figure 4.7: Denoising a 3D grayscale volume image (a) Original image "BostonTeapot.raw", $256 \times 256 \times 178$ px, 8 bit color depth (b) Component-wise Gaussian noise $\mu = 0$, $\sigma = 0.1$ added (c) Denoised, Proximal point with $\lambda = 0.1$, 50 PRPT steps

4.2 SO(2) and SO(3) images data

4.2.1 Synthetic data

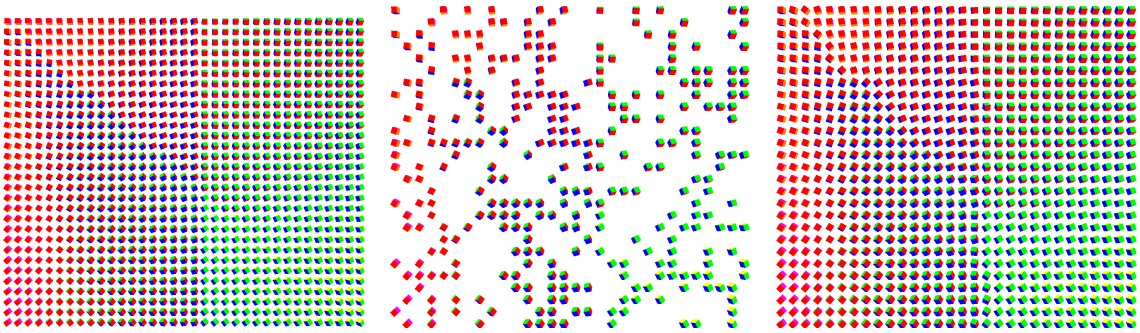
The following synthetic $SO(3)$ image is constructed in the following way. Let $\Omega = \{1, \dots, 30\}^2$ and define for every $(i, j) \in \Omega$ a rotation axis

$$v = \begin{cases} (2x, y, 0)^T, & x > 0.5 \\ (0, 2x, 0.5)^T, & \text{else} \end{cases}, \tag{4.1}$$

where $x = \frac{j}{30}, y = \frac{i}{30}$ and a rotation angle

$$\alpha = \begin{cases} x + y, & x > y \\ \frac{\pi}{2} + x - y, & \text{else} \end{cases}. \tag{4.2}$$

Then assign the corresponding $SO(3)$ element representing a rotation by α and about v. Noise is added component-wise and the noisy matrix is then projected back to $SO(3)$ using the projector $P_{SO(n)}(A) = UV^T$ where $A = U\Sigma V^T$ is the singular value decomposition of A.



(a) Original (b) Damaged (c) Reconstructed

Figure 4.8: Inpainting of synthetic SO(3) picture (a) Original image: Synthetic, non-smooth SO(3), 30×30 px (b) Threshold $p = 0.4$ (c) Denoised, IRLS with $\lambda = 0.1$, 5 IRLS steps, 1 Newton step per IRLS

4.2.2 Fingerprint orientation data

Fingerprint matching is based on extracting a set of particular features, called *minutiae*, which uniquely define the fingerprint. These features are usually ridge endpoints or ridge bifurcation points that are saved along with their position and orientation. This means that prior to minutia detection and extraction the calculation of an orientation field is necessary.

For pictures of fingerprints this is just a special form of edge detection which can be done by calculating discrete derivatives for every pixel using a Sobel or Scharr operator. The ridges in the original fingerprint, however, are usually too thick resulting in gradient values close to zero within the ridge and consequently ill-defined orientations. For that reason, we first employ the Zhang-Suen thinning algorithm [33] to obtain only the ridge skeleton for which the derivatives are then computed.

Depending on the quality and noise level of the picture the computed orientation field can be very noisy itself which is another application for our TV algorithms. An example is provided in Figure 4.9.

(a) Original (b) Ridge Skeleton (c) Computed Orientation field (d) Denoised Orientation field

Figure 4.9: Denoising a orientation field from a fingerprint, orientations represented by SO(2) elements (a) Original fingerprint (a) Ridge skeleton computed using a thinning algorithm (c) Orientation field computed using Scharr derivatives (d) Denoised, IRLS with $\lambda = 2.1$, 5 IRLS steps, 1 Newton steps per IRLS step

4.2.3 Reconstruction of a dense optical flow field

An optical flow is the pattern of apparent motion between two consecutive frame of a video sequence. This may be the result of either an actual movement of the depicted object or the result of a moving camera. Important applications are for example (abnormal) motion detection, crowd behavior analysis, surveillance, video compression or image segmentation.
A *dense* optical flow field can be interpreted as a vector field where each vector describes the displacement of a point from one frame to the next. If the set of points is restricted to only a few points of interest, a sparse feature set, we have *sparse* optical flow.

In the following example, we use a sparse feature set for tracking and flow computation in a short video sequence. The traffic scene was taken from a crowds/high density moving object data set provided by [8]. At first, we compute the sparse optical flow using the Lucas-Kanade algorithm [25] implemented in the OpenCV library.
For the set of tracked features $\mathcal{F}_1 := \{F_i^{(1)}\}_{i=1}^{400} \subset \Omega \subset \mathbb{R}^2$ in the first frame the algorithm tries to identify each feature in the second frame resulting in a set of identified features $\mathcal{F}_2 := \{F_i^{(2)}\}_{i=1}^{N<400} \subset \Omega \subset \mathbb{R}^2$ and corresponding displacement vectors $\mathcal{V}_{12} := \{V_i \,|\, V_i = F_i^{(2)} - F_i^{(1)}\}_{i=1}^{N}$.

We now assign to each pixel in our data an SO(2) element in the following way

$$\alpha_i = \arctan\left(\frac{V_i^y}{V_i^x}\right) \tag{4.3}$$

$$I(i,j) = \begin{cases} \begin{pmatrix} \cos\alpha_i & -\sin\alpha_i \\ \sin\alpha_i & \cos\alpha_i \end{pmatrix} & (i,j) \in \mathcal{F}_2 \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

Since we want to reconstruct the dense flow, this is an inpainting problem and we have to perform scattered interpolation before running the algorithm. The result can be seen in Figure 4.10.
Of course the optimization could have also been performed on S^1. Furthermore, there is also a more direct, variational approach for the calculation of the flow field which is also based on TV minimization but has a different fidelity term. This is one possibility for further extension of the library and is discussed in more detail in section 5.2

(a) Original (b) Computed Sparse Flow field (c) Reconstructed orientation field

Figure 4.10: Reconstructing a dense flow from sparse feature tracking, orientations represented by SO(2) elements (a) Frame of original video scene (b) Sparse features tracked using Lucas-Kanade (c) Reconstructed, IRLS with $\lambda = 0.05$, 5 IRLS steps, 1 Newton steps per IRLS step

4.3 SPD(3) image data

4.3.1 Synthetic data

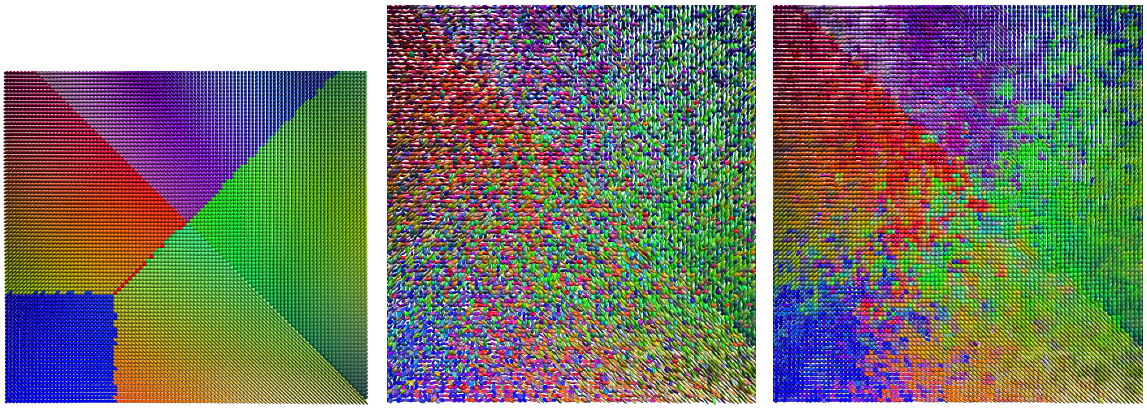
For the construction of the synthetic $SPD(3)$ image in Figure 4.11, let $\Omega = \{1, \dots, n\}^2$ and define for every $(i, j) \in \Omega$ a rotation axis

$$v = \begin{cases} (x, y, 2)^T, & x + y < 1 \\ (y, -x, 1)^T, & \text{else} \end{cases}, \tag{4.5}$$

where $x = \frac{j}{n}, y = \frac{i}{n}$ and a rotation angle

$$\alpha = \begin{cases} x + 2y, & x + y < 1 \\ y + 2x, & \text{else} \end{cases}. \tag{4.6}$$

Let R be the corresponding $SO(3)$ element representing a rotation by α and about v. Then define a diagonal matrix $D = \text{diag}(x + 0.2, y + 0.2, 0.5)$ and assign the matrix $A = R^T D R$ to the pixel. Noise is then added by taking the matrix logarithm of every pixel, adding Gaussian component-wise noise and applying the matrix exponential again.



(a) Original (b) Damaged (c) Denoised

Figure 4.11: Denoising of synthetic SPD(3) picture (a) Original image: Synthetic, non-smooth SPD(3), 100×100 px PLACEHOLDER (b) Threshold $p =?$ (c) Denoised, IRLS with $\lambda =?$, 5 IRLS steps, 1 Newton step per IRLS

4.3.2 Diffusion Tensor Magnetic Resonance Imaging

Diffusion Tensor Magnetic Resonance Imaging (DT-MRI) is a medical imaging method which is able to non-invasively measure diffusion coefficients of water molecules in living biological tissues. DT-MRI goes beyond CT or normal MR imaging methods which are only able to provide a single intensity value per voxel. Since water molecules can move easier along, for example axons, connecting the neurons in the brain, than they can move across it, the resulting anisotropic diffusion pattern can provide a lot of information about the structure of the brain.

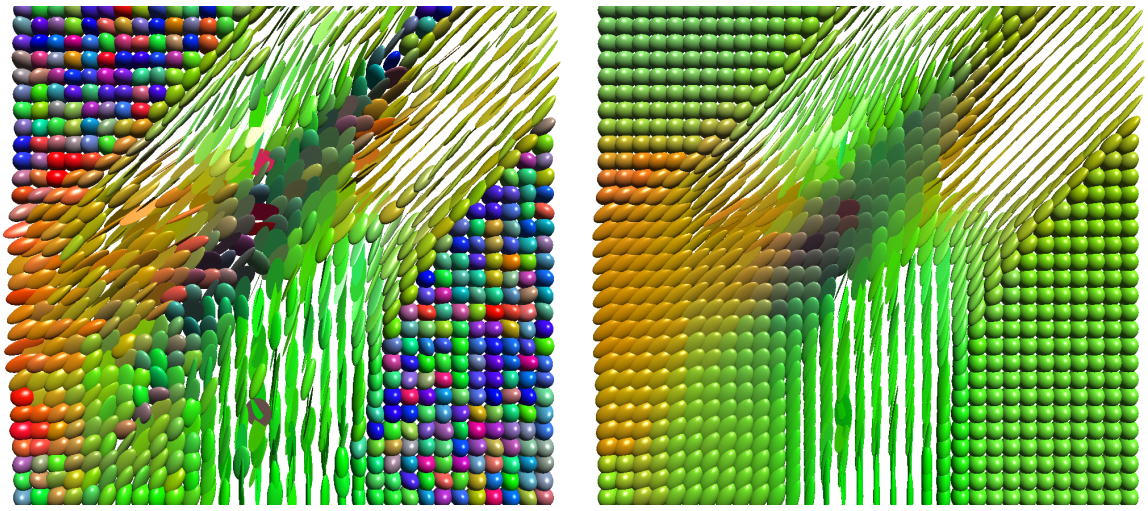
DTI data sets are usually calculated from a set of diffusion weighted magnetic resonance imaging (DW-MRI) pictures. The basic magnetic resonance imaging works by applying an external magnetic field along the z-axis such that the proton spins in the tissue align either parallel or anti-parallel to it while still preceding around the z-axis with the so-called Lamor frequency. An electromagnetic wave packet(HF-pulse) with that exact frequency leads to a collective state transition to a resonance state such that spin moments will be phase-synchronous before relaxing back to their original orientation with respect to the external field. The magnetic field created by having synchronized moments can be measured by a coil where an electric potential will be created. From the different relaxation times of different materials one can make conclusions about the structure of the tissue.

By applying an additional magnetic gradient field, the Lamor frequency of different layers of the probe can be modified such that only one layer of the material will resonate to the pulse. This provides an additional positional resolution of the imaging process.

The DTI image is finally computed using the *Stejskal-Tanner-equation* given by

$$A(\mathbf{g}) = A(0)\exp(-b\mathbf{g}^T\mathbf{D}\mathbf{g}) \tag{4.7}$$

where $A(\mathbf{g})$ denotes the signal strength, $\mathbf{g}$ the magnetic gradient field and b some measurement related parameters. Solving this equation for $\mathbf{D}$ finally leads to desired $SPD(3)$ matrix describing the diffusion coefficients and directions.



(a) Noisy (b) Denoised

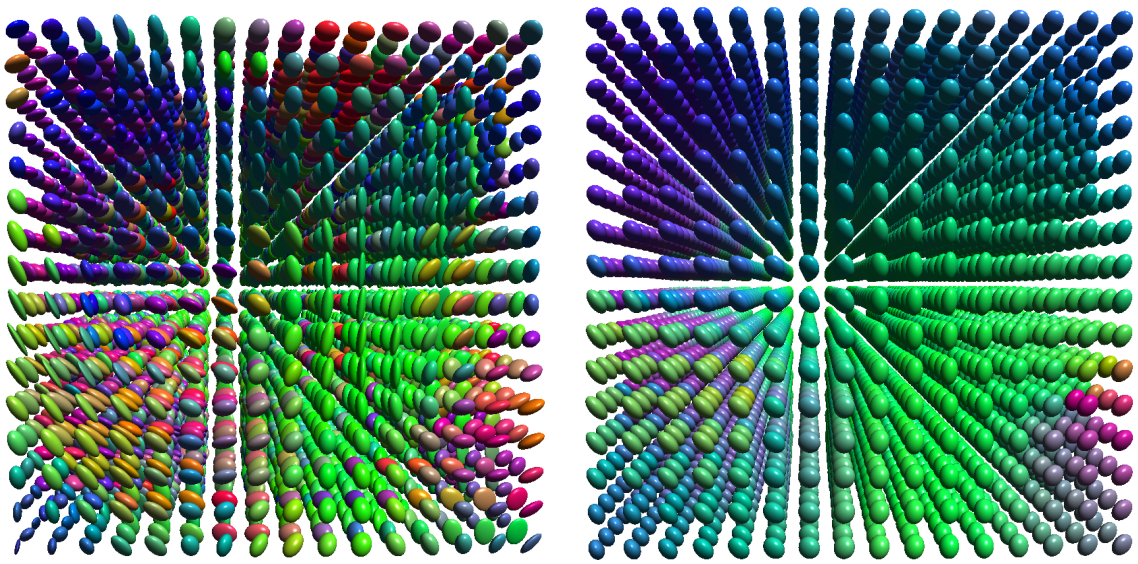
Figure 4.12: Denoising a DT-MRI image with pixel in SPD(3) (a) Original DTI data, 32x32 pixel (b) Denoised, IRLS with $\lambda = 0.7$, 5 IRLS steps, 1 newton steps per IRLS step

In Figure 4.12 the IRLS minimizer is applied to DTI data set provided by Barmpoutis [2]. One can clearly identify regions of high anisotropy in the picture, where the molecules are forced to diffuse in one preferred direction. The areas dominated mainly by green spheres correspond to approximately isotropic diffusion which means that there are no obstacles, like axons in the brain,

in the immediate proximity of the water molecules.

4.3.3 3D DT MRI data

Finally, in Figure 4.13a we show a 3D DTI image. Shown is a $16 \times 16 \times 16$ cube from a human brain scan and we used the proximal point algorithm for denoising. The brain data set is a courtesy of Gordon Kindlmann at the Scientific Computing and Imaging Institute, University of Utah, and Andrew Alexander, W. M. Keck Laboratory for Functional Brain Imaging and Behavior, University of Wisconsin-Madison.



(a) Original (b) Denoised

Figure 4.13: Denoising a 3D DT-MRI image with pixel in SPD(3) (a) Original, $16 \times 16 \times 16$ pixel (b) Denoised, Proximal point with $\lambda = 0.7$, 50 PRPT steps

4.4 Performance analysis of the library

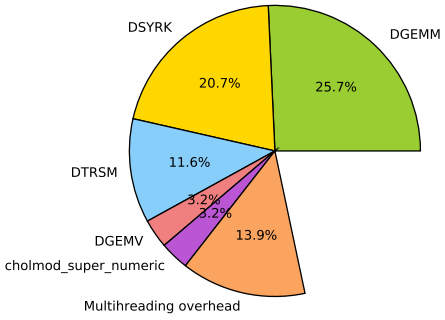
In this section we analyze the performance hotspots of the library for the case of the IRLS minimizer. This is done using the Linux tool *perf* which monitors a variety of different performance metrics during the execution of a program, like CPU cycles, cache or branch misses. To identify the hotspots, where most of the computation is spent, the number of CPU cycles is usually the most suitable metric.

Concerning the computational complexity of operations performed on single pixels the Euclidean manifold is certainly the least demanding, because exponential and logarithm map are just addition and subtraction and the second derivative of the distance function is just two times the identity matrix. At the other end of the spectrum is the SPD manifold where computations usually involve multiple matrix multiplications, exponentials, logarithms and derivatives thereof. For the analysis we thus choose these two representatives and compare different image sizes.

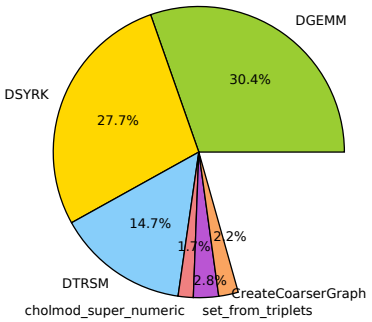
For the Euclidean manifold we choose the "Lena" and "Mathematicians" pictures already considered in the example above. Minimization in the first case took approximately 9 seconds and 270 seconds in the second case. The data is shown in Table 4.1 where the first column denotes the percentage of CPU cycles spent in the routine specified in the second column while the last column denotes the (external) library to which it belongs. Only the top six routines are shown since the individual

share of the others was in most cases less than 1 %.

We can observe that in both cases computation is dominated by the Basic Linear Algebra Subprograms (BLAS) Library. Those in turn are called by the CHOLMOD library which solves the sparse linear system using Cholesky factorization. The only contribution that does not belong to the linear system is the multi-threading overhead from the OpenMP (OMP) library in the smaller picture. We see, however, that for the increased system size the overhead becomes negligibly small such that for both problem sizes more than two thirds of the total computation time is for solving the linear system. For the larger picture this share even grows to more than 75% and could be expected to do so for yet larger images.



(a)



(b)

Share	Routine	Library
25.73	DGEMM(matrix matrix multiply)	BLAS
20.71	DSYRK(symmetric rank-k update)	BLAS
13.86	Multihreading overhead	OMP
11.57	DTRSM(solve triangular system)	BLAS
3.23	DGEMV(matrix vector multiply)	BLAS
3.22	cholmod_super_numeric	CHOLMOD

(a) 361×361

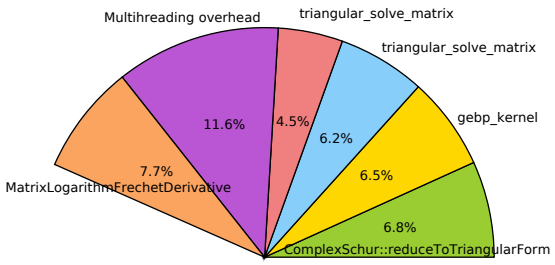
Share	Routine	Library
30.37	DGEMM(matrix matrix multiply)	BLAS
27.70	DSYRK(symmetric rank-k update)	BLAS
14.65	DTRSM(solve triangular sytem)	BLAS
2.79	set_rom_triplets(sparse initialization)	Eigen
2.18	CreateCoarserGraph	METIS
1.68	cholmod_super_numeric	CHOLMOD

(b) 1280×1024

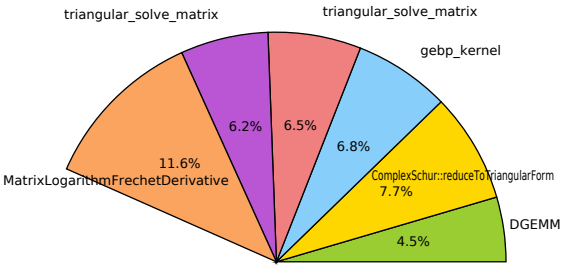
Table 4.1: Share of total CPU cycles for IRLS minimization over $M = \mathbb{R}^3$ (a)"Lena.jpg", 361×361 pixel (b)"Mathematicians.jpg", 1280×1024 pixel

For the $SPD(n)$ manifold the situation, shown in Table 4.2 and Table 4.3, looks a bit different at first. For the smallest problem size of 30×30 pixels, there are dominating parts. The largest contribution is the from multi-threading which is to be expected for such a small picture and which consequently vanishes for larger images. The next important routine is our implementation of the matrix logarithm Fréchet derivative while the remaining Eigen routines in the list are auxiliary functions for solving triangular matrix functions which are need for the computation of matrix square roots, logarithms and of course also their Fréchet derivatives. With increasing problem size, we can again observe how the BLAS routines move to the top of the list, even though their share only amounts to a fifth of all CPU cycles for the 300×300 pixel image.

We can conclude that solving the linear system is the most performance relevant aspect of the IRLS minimizer. The share is even higher for $SO(n)$ and S^n, where the SuperLU library is used, since the corresponding sparse Hessian is not symmetric, resulting in a further increased operations count. If a good preconditioner is found, iterative solvers might speed up the computation, the standard diagonal and incomplete LU preconditioner provided by the Eigen library, however, performed worse than the direct solvers from the SparseSuite library collection. Finally, for the matrix-valued manifolds, the implementation of the Fréchet derivative could potentially be improved.



(a)



(b)

Share	Routine	Library
11.6	Multithreading overhead	OMP
7.73	MatrixLogarithmFrechetDerivative	TVMTL
6.77	reduceToTriangularForm	Eigen
6.55	gebp_kernel(matrix blocking)	Eigen
6.18	triangular_solve_matrix	Eigen
4.55	triangular_solve_matrix	Eigen

(a) 30×30

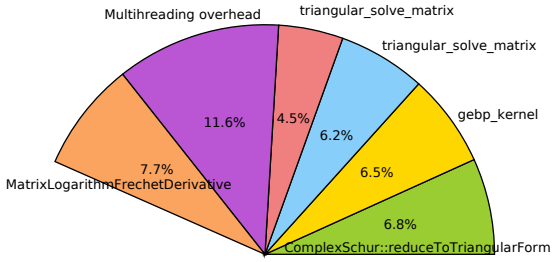
Share	Routine	Library
11.60	MatrixLogarithmFrechetDerivative	TVMTL
7.72	reduceToTriangularForm	Eigen
6.77	gebp_ kernel(matrix blocking)	Eigen
6.55	triangular_solve_matrix	Eigen
6.18	triangular_solve_matrix	Eigen
4.55	DGEMM(matrix matrix multiply)	BLAS

(b) 100×100

Table 4.2: Share of total CPU cycles for IRLS minimization over $M = SPD(3)$ (a) Synthetic $SPD(3)$, 30×30 (b) Synthetic $SPD(3)$, 100×100

Share	Routine	Library
11.31	DGEMM(matrix matrix multiply)	BLAS
9.73	MatrixLogarithmFrechetDerivative	TVMTL
9.19	DSYRK(symmetric rank-k update)	BLAS
6.60	reduceToTriangularForm	Eigen
5.73	gebp_kernel(matrix blocking)	Eigen
5.55	triangular_solve_matrix	Eigen

(a)

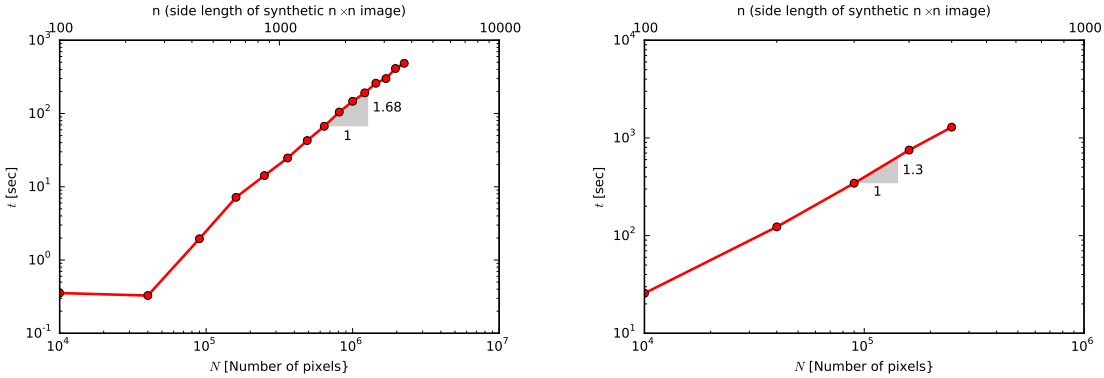


(b)

Table 4.3: Share of total CPU cycles for IRLS minimization of synthetic $SPD(3)$, 300×300 pixel, $M = SPD(3)$

4.4.1 Time Complexity

Finally, we will briefly take a look at the measurements of the time complexity obtained for the Euclidean $\mathbb{R}^3$ and $SPD(3)$. = For both cases the time complexity over the considered input size range, which was limited only due to the available RAM on the test platform, we see subquadratic time complexities. Despite the fact that based on the theory one would expect quasi-linear behavior, this deviation is to be expected. Handling large amounts of data, which in the above tests are in the GB range, naturally leads to additional cost for memory access. Memory is hierarchically structured with the CPU caches having the lowest access time but also the smallest size, whereas



(a) $\mathbb{R}^3$ Minimization time versus input size

(b) $SPD(3)$ Minimization time versus input size

Figure 4.14: Time complexity for the IRLS $\mathbb{R}^3$ and $SPD(3)$. In both cases, minimization was performed using 5 IRLS steps and 1 Newton step per reweighting (a) Denoising of a synthetic RGB picture of size $n \times n$ with $n = 100, 200, \ldots, 1500$ (b) Denoising of a synthetic $SPD(3)$ picture of size $n \times n$ with $n = 100, 200, \ldots, 500$

the RAM is comparably huge but has a much larger access time. At the bottom of the hierarchy is swap space on the hard drive. The cost of a so-called page-fault, which happens when a program tries to access a memory location that is not loaded into the main memory, can amount to more than 1000 CPU cycles. Since with increasing memory utilization the probability for page faults may also increase, performance will consequently decrease.

It is remarkable, however, that the complexity for the SPD manifold is lower than for Euclidean space. Taking into account the analysis of the previous section, where it was illustrated that most time is spent solving the linear system, a possible explanation might be that for the SPD manifold the share of computational effort of solving the linear system is smaller relative to the remaining operations, like calculation of the derivatives for example. Assuming that the latter perform indeed quasi-linear, this would suggest that, on the other hand, solving the system must have a complexity larger than $\mathcal{O}(\mathrm{NNZ}(HJ)) = \mathcal{O}(N)$.

Considering above remarks on the effects of memory access speed, we can support this claim by looking at other performance metrics like cache-misses and page-faults, where the BLAS routines are again at the top of the listings.

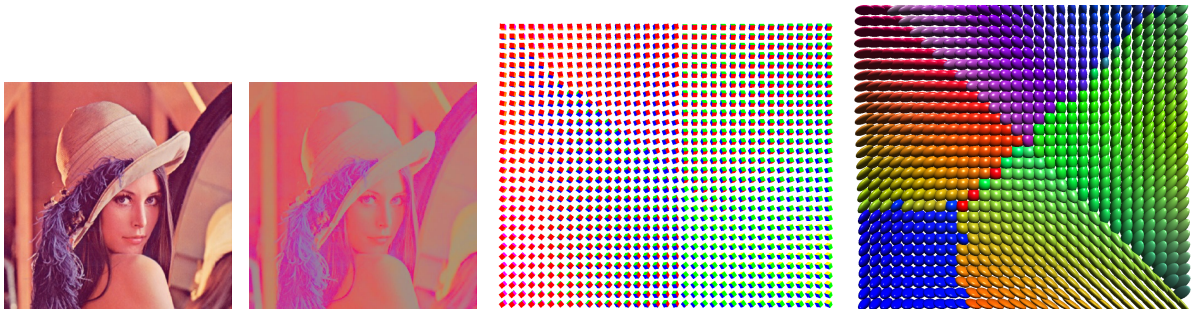
4.5 Comparison IRLS and Proximal Point minimizers

To make a comparison with the tests performed in [17], using the original Matlab implementation, to a certain degree possible, we choose almost the same set of test images (Figure 4.15) but additionally also some larger versions of the pictures for the more interesting case of the matrix manifolds $SO(3)$ and $SPD(3)$. Both, the IRLS and the proximal point minimizers are implemented using the same manifold classes and utilize the same pixel-wise parallelization techniques such that there is no obvious bias in this comparison.

The synthetic images are created using the formulas already described in the examples above. To each picture we add component-wise Gaussian noise with zero mean and standard deviation of $\sigma = 0.2$. With that noise level no smoothing is needed for the IRLS algorithm to converge. For each picture we compute the value of the functional after each iteration and the error relative to an approximate minimizer u^* which is computed using the IRLS algorithm with $\lambda = 0.2$, 20 IRLS steps and one newton step per reweighting. This error is defined as

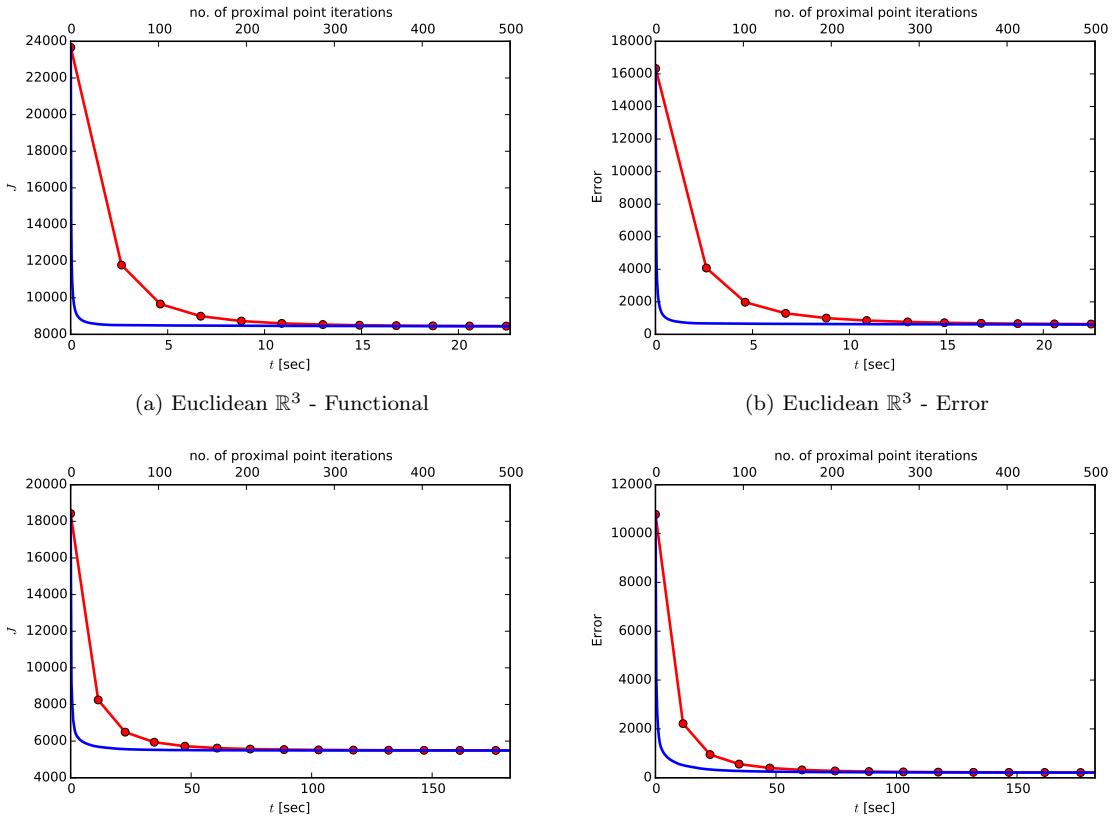
$$e^{(k)} = \sum_{i,j} d^2(u_{ij}^{(k)}, u_{ij}^*) \tag{4.8}$$

where $d^2(\cdot,\cdot)$ denotes the squared Riemannian distance function of the appropriate manifold. For the iterations itself we use 15 iterations for IRLS and 500 for proximal point with the sequence $\mu_k = 3k^{-0.95}$ (see [32] for details on this sequence) for all experiments.



(a) $\mathbb{R}^3$ (b) S^2 (c) $SO(3)$ (d) $SPD(3)$

Figure 4.15: Test images for the IRLS & PRPT comparison (a) "Lena.jpg", 361 × 361 px, $\mathbb{R}^3$-valued (b) Chromaticity part of "Lena.jpg", S^2-valued (c) Synthetic 30 × 30 and 100 × 100 image, $SO(3)$-valued (d) Synthetic 30 × 30 and 100 × 100 image, $SPD(3)$-valued

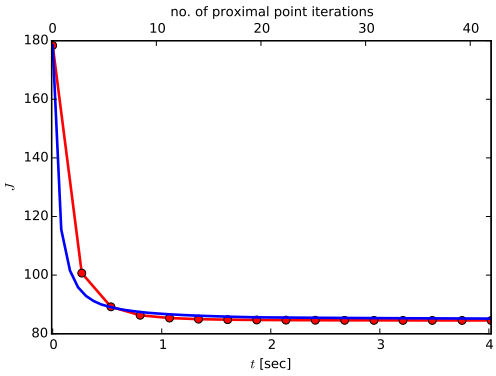


(a) Euclidean $\mathbb{R}^3$ - Functional (b) Euclidean $\mathbb{R}^3$ - Error

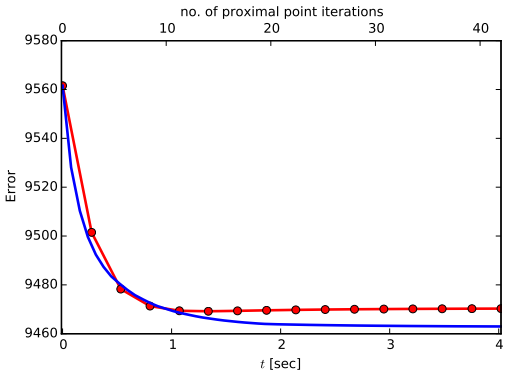
(c) Sphere S^2 - Functional (d) Sphere S^2 - Error

Figure 4.16: Comparison of functional values and errors for IRLS and PRPT minimizers. The red circles correspond to IRLS iterations while the line without any markers belongs to the proximal point iterations. (a) Functional values for "Lena" minimized over $\mathbb{R}^3$ (b) Errors relative to minimizer for "Lena" (c) Functional values for color part of "Lena" minimized over S^2 (d) Errors relative to minimizer for color part "Lena"

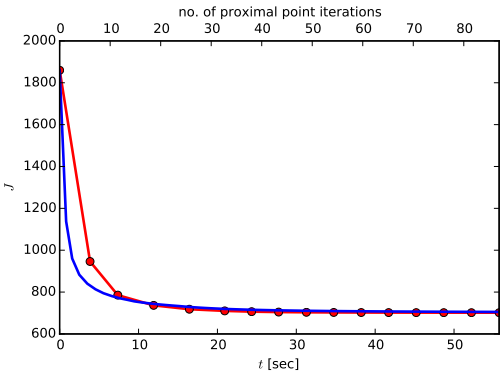
Figure 4.16 shows the results for Euclidean space $\mathbb{R}^3$ and the sphere S^2. IRLS needs only five iterations where proximal point needs more than 200 but nevertheless proximal wins in terms of speed. This result can be interpreted with respect to the performance analysis conducted in the previous section. Most manifold quantities can be computed using only addition and subtraction. This includes the geodesic interpolation that proximal point has to perform in every direction as well as the computation of derivatives for IRLS. The effort of doing these two task seems intuitively similar. At the end of that process, however, proximal point only has to compute simple arithmetic means while IRLS has to solve the sparse system which was shown to be the dominant part of the computation and thus probably more time consuming than the averaging procedure. For S^2, in comparison, one can already see IRLS catching up.



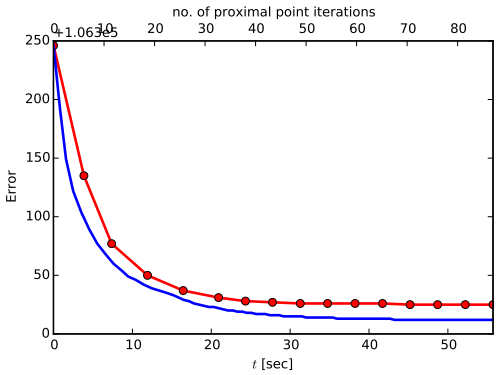
(a) $SO(3)$ (30×30) - Functional



(b) $SO(3)$ (30×30) - Error



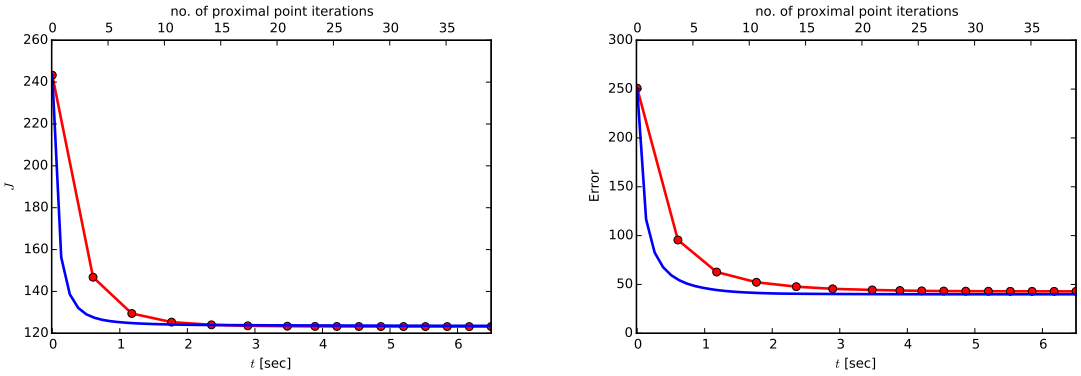
(c) $SO(3)$ (100×100) - Functional



(d) $SO(3)$ (100×100 - Error

Figure 4.17: $SO(3)$: Comparison of functional value and errors for IRLS and PRPT minimizers. The red circles correspond to IRLS iterations while the line without any markers belongs to the proximal point iterations. (a) Functional values for synthetic 30×30 $SO(3)$ (b) Errors relative to minimizer for synthetic 30×30 $SO(3)$ (c) Functional values for synthetic 100×100 $SO(3)$ (d) Errors relative to minimizer synthetic 100×100 $SO(3)$

For $SO(3)$, shown in 4.17, the difference again becomes much smaller, with both plots being very close and coinciding already after the second iteration. Next, in the case of $SPD(3)$, shown in 4.18, IRLS falls back to the niveau of S^2. One reason for that might be the particularly complicated form of the second derivative of the square distance function. This could be eventually optimized by improving the implementation of the Fréchet derivative. At this time it is based on the computation of a complex Schur decomposition which then, in turn, needs complex arithmetic. Switching to a block-based, but real Schur decomposition might provide an additional speed-up.



(a) $SPD(3)$ (30×30) - Functional

(b) $SPD(3)$ (30×30) - Error

Figure 4.18: $SPD(3)$: Comparison of functional value and errors for IRLS and PRPT minimizers. The red circles correspond to IRLS iterations while the line without any markers belongs to the proximal point iterations. (a) Functional values for synthetic 30×30 $SPD(3)$ image (b) Error relative to minimizer for synthetic 30×30 $SPD(3)$ image

4.6 Sensitivity to variations of the original data

In this section we perform a numerical experiment to see how changes of the noisy original picture influence the global solution of TV minimization. For that purpose we consider only the brightness part of the Lena picture. In this grayscale image we pick a single non-zero pixel far enough in the inside of the picture and set this pixel to zero, i.e. black color. This leads to two different original pictures

$$u_0, \hat{u}_0 : \Omega \to \mathbb{R} \tag{4.9}$$

$$(\hat{u}_0)_{ij} = \begin{cases} 0, & \text{if } i = i_0 := 100, j = j_0 := 100 \\ (u_0)_{ij}, & \text{else} \end{cases}. \tag{4.10}$$

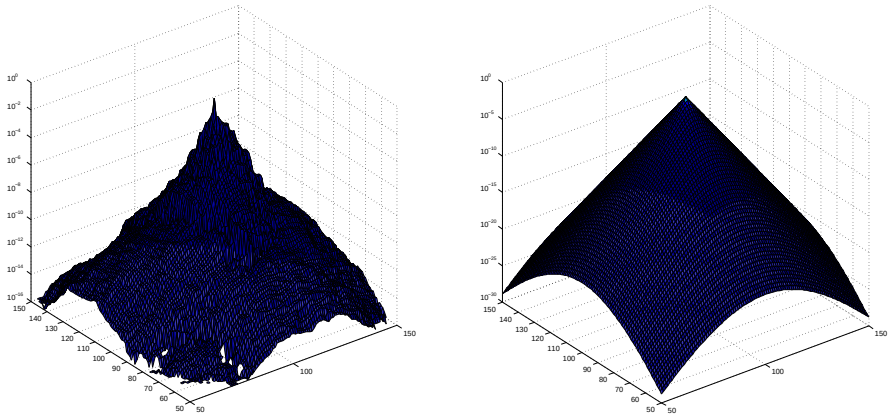
Then for the original u_0 and the modified image $\hat{u}_0$ an IRLS minimization with $\lambda = 0.1$, 5 iterations and one Newton step per reweighting is performed to compare the two the solutions u and $\hat{u}$. We take the absolute differences $e_{ij} = |u_{ij} - \hat{u}_{ij}|$ between the solutions which leads to the error cone shown in Figure 4.19a. This already suggests that the error caused by changing the original data decays exponentially with distance $r = \sqrt{(i - i_0)^2 + (j - j_0)^2}$.

After fitting a cone to the data, which is shown in 4.19b one finds that the aperture half-angle corresponds to a slope of $c = 1.XXX$ such that the error will decay as

$$e_{ij} \propto e^{-cr}. \tag{4.11}$$

The most important conclusion one can draw from this is that a pixel in the global minimizer depends practically only on a local neighborhood of that pixel in the original picture, not on pixels far away from that neighborhood. We will discuss possible applications of this in Section 5.3.

Of course these findings should be verified analytically by providing sufficiently precise error bounds, which is, however, unfortunately out of the scope of this thesis.



(a) Error cone

(b) Fitted cone

Figure 4.19: Sensitivity to change in original data (a) Logarithmic plot of absolute differences $e_{ij} = |u_{ij} - \hat{u}_{ij}|$ (b) Cone fitted to the data, resulting in aperture half-angle of $\alpha = XXX$

Chapter 5

Conclusion and Outlook

5.1 Summary

In this thesis we introduced a very versatile, multi-threaded C++ template library for manifold-valued data, which extends the original Matlab prototype in a variety of directions. This includes shared memory and SIMD parallelization for IRLS and proximal point algorithms, 3D images, the Grassmann manifold as well as supporting methods for numerical matrix function derivative computations and OpenGL visualization methods.

For the theoretical background we could provide some semi-analytic expressions for the derivatives of the squared distance functions using Kronecker product which allows a very compact implementation and gave a short overview about the relevant Grassmann manifold theory.

The third chapter can be taken as a high level documentation of the library and its structure and underlying concepts It gives more insights on the software engineering point of view on this thesis and also contains some basic tutorials on how to use or even extend the library.

Finally, we demonstrated the libraries capabilities on many different applications like standard grayscale and color images, medical DT-MRI data, synthetic $SPD(3)$ and $SO(3)$ data but also examples based on real applications like image orientation maps and optical flow computation. Aside from these more colorful demonstrations also a performance analysis of the library was done to investigate performance bottlenecks of the IRLS minimizer. As main result from this analysis we could identify the solution of the sparse linear system as most performance relevant component in the process, which does not seem to scale (quasi-)linearly. But also our implementation of the Matrix logarithm Fréchet derivative had some influence and has potential for further optimization.

Consequently, we found that IRLS performed slower compared to an also parallelized proximal point algorithm despite the much higher convergence rate of the first. Lastly, in the course of further optimization opportunities we investigated the sensitivity of the computed solution with respect to variations of the original data and found that resulting error in the solution is locally confined to a neighborhood around the varied pixel in the original and decays exponentially with the distance from the varied pixel. A possibility for an extension of the library exploiting this locality of the error is discussed in the last Section 5.3. Next, however, we suggest some more general possibilities.

5.2 Extensions and Improvements

5.2.1 Performance

Since the solution of the sparse linear system is the most performance critical component of the algorithm, new solution methods should be tested. In particular, iterative methods could perform

better if a good preconditioner can be found.

By making use of the special block-band structure of the Hessian, we avoided the temporary block diagonal matrix containing the tangent space basis transformation that was used in the Matlab prototype. The tangent space transformations could instead be applied directly to the pixels of the derivative containers. In a similar manner, it might also be possible to make the algorithm completely matrix-free by implementing a matrix-vector multiplication function for the Hessian matrix that can be used by iterative solver. This might save a lot of memory as well as computation time and bring the algorithm closer to the linear complexity regime.

5.2.2 Manifolds and Minimizers

The first thing that can easily be extended is the support for additional manifolds. Possibilities are, for instance, the Stiefel manifold that was briefly introduced in section 2.4.5 or an alternative implementation of the $SPD(n)$ manifold using the Log-Euclidean metric based on [10].

Also new minimizers could be added. It would be straightforward to utilize the gradient evaluation function already implemented in the functional to add some gradient descent based algorithm and compare again with IRLS-Newton and proximal point. For new functionals we provide a more detailed suggestion in the next section.

5.2.3 Functionals

So far isotropic and anisotropic first order TV functionals are implemented in the library. Extensions can be made with respect to the TV part or the fidelity part of the functional. In the first case this would mean to also include a second order TV term $\mu \int_\omega |\nabla^2 u|\, dx$ in the functional. This prevents the formation of numerical artifacts like the so-called staircasing effects, that might occur in first order TV. Of course also higher orders than second can be added to the functional. As long as also methods for the evaluation of the functionals gradient and Hessian are provided, the IRLS minimizer class will work without any changes.

The second possibility is the addition of new or different fidelity terms. The purpose of the fidelity term during the minimization is to penalize a TV regularization that moves too far away from the original picture. However, one can also utilize it for a direct calculation of a dense optical flow field, for example. This was also done in the application shown in 4.2.3 but it must be noted that the procedure we chose to compute the dense flow was rather complicated and indirect.

For the direct TV approach, as presented in [31], consider a 2D video sequence $I : \Omega \times [0, T] \to \mathbb{R}$. Let now $u : \Omega \to \mathbb{R}^2$ denote the displacement vector of the of the pixel $x \in \Omega$. The functional is then given by

$$J(u) = \int_\Omega \left| \frac{\partial I}{\partial t} + \nabla I \cdot u \right| dx + \lambda TV(u), \tag{5.1}$$

where the first term is the new data term, which implements the so-called optical flow constraint that could be interpreted as a continuity equation for pixels. Finally, as Lefévre and Baillet show in [22], the functional can be generalized also to flows on some classes of manifolds.

5.3 Recursive computation on subdomains

From the numerical experiment in Section 4.6 we can conclude that a local neighborhood of the original picture only affects the form of the minimizer of the function in its immediate neighborhood. The magnitude of the variation in the minimizer due to the variation of a single pixel in the original data seems to decay exponentially with the distance from that pixel.

This could in principle be exploited by dividing the image into subdomains with a specific small overlap, determined by the error boundaries, and solve each of the smaller subproblems individually.

Depending on the size of the necessary overlap this could also be employed recursively until a minimal subimage size is reached. Then, after all subproblems are solved the subpictures are recombined just by cropping all overlapping regions to arrive at the global solution. As a final example we take an extended version of the Lena picture and split it in the middle with $50px$ overlap, as shown in Figure 5.1.

(a) Noisy full picture (b) Noisy left half (c) Noisy right half

Figure 5.1: Splitting of the Lena picture into two domains (a) Original image "Lena.jpg", 1000×550 px, 8 bit color depth (b) Left half, 550×550 (c) Right half 550×550

Next, we run the algorithm on the full picture, which takes 72 seconds, as well as on the two halves, which takes 33 second for each run. Then we recombine the two halves after cropping away the overlap. The results, shown in Figure 5.2, look promising. Of course, at this point some more detailed numerical error analysis should be performed but just from visual inspection there are no serious problems, like a visible brightness or color gradient for example, of the splitting procedure evident.

(a) (b)

Figure 5.2: Comparison between the denoising over the full domain and denoising over two subdomains (a) Denoised full picture (b) Recombined, denoised picture halves

The advantages of this procedure are firstly, that even though there is some overhead from this procedure, a collection of smaller subproblems can be usually solved faster than one big problem and have a lower memory consumption. The speed up in the above example is not very large (66 versus 72 seconds), which is also to be expected from the subquadratic time complexity measured in Section 4.4.1. Nevertheless, the main point is not about the speed up of serial-solving the subproblems but the lower memory demand allowing yet larger images to be processed.

Secondly, this splitting scheme can also be used to introduce an additional layer of parallelism in the from of distributed memory, many core parallelization. Each subproblem can be assigned to a different node that in turn locally applies multi-threading. Building such a distributed computing architecture on top of the solver, a significant speed up can be reached, because except from splitting and recombination there is no need for communication between the nodes during the TV

minimization.

Appendix A

Listings

In this appendix chapter we provide full listings for the tutorial chapter.

Listing A.1: ./listings/tvmin˙test.cpp

```
#include <iostream>
#include <string>
#include <opencv2/highgui/highgui.hpp>

#include "../core/algo_traits.hpp"
#include "../core/data.hpp"
#include "../core/functional.hpp"
#include "../core/tvmin.hpp"

#include <vpp/vpp.hh>
#include <vpp/utils/opencv_bridge.hh>

using namespace tvmtl;

typedef Manifold< EUCLIDIAN, 3 > mf_t;
typedef Data< mf_t, 2> data_t;
typedef Functional<FIRSTORDER, ISO, mf_t, data_t> func_t;
typedef TV_Minimizer< IRLS, func_t, mf_t, data_t, OMP > tvmin_t;


void DisplayImage(const char* wname, const data_t::storage_type& img, vpp::↘
    image2d<vpp::vuchar3>& out){
        cv::namedWindow( wname, cv::WINDOW_NORMAL );

        // Convert Picture of double to uchar
        vpp::image2d<vpp::vuchar3> vucharimg(img.domain());
        vpp::pixel_wise(vucharimg, img) | [] (auto& i, auto& n) {
            mf_t::value_type v = n * (double) std::numeric_limits<unsigned char↘
                >::max();
            vpp::vuchar3 vu = vpp::vuchar3::Zero();
            vu[0]=(unsigned char) v[2];
            vu[1]=(unsigned char) v[1];
            vu[2]=(unsigned char) v[0];
            i = vu;
        };

        cv::imshow( wname, vpp::to_opencv(vucharimg));

        out =  vucharimg;
        cv::waitKey(0);
}


int main(int argc, const char *argv[])
{
```

```
        if (argc < 2){
            std::cerr << "Usage : " << argv[0] << " image [lambda]" << std::endl;
            return 1;
        }

        double lam=0.1;

        if(argc==3)
            lam=atof(argv[2]);

        data_t myData=data_t();
        myData.rgb_imread(argv[1]);

        func_t myFunc(lam, myData);
        myFunc.seteps2(1e-10);

        tvmin_t myTVMin(myFunc, myData);

        vpp::image2d<vpp::vuchar3> img;

        std::string fname(argv[1]);

        std::cout << "Smoothen picture to obtain initial state for Newton ↘
            iteration..." << std::endl;
        myTVMin.smoothening(5);
        DisplayImage("Smoothened", myData.img_, img);
        cv::imwrite("smoothened_" + fname, to_opencv(img));

        std::cout << "Start TV minimization..." << std::endl;
        myTVMin.minimize();


        DisplayImage("Denoised", myData.img_, img);
        cv::imwrite("denoised_" + fname, to_opencv(img));

        return 0;
}
```

Linear vectorial TV minimization

Listing A.2: ./listings/colorization˙test.cpp

```
#include <iostream>
#include <string>
#include <cmath>

#include <opencv2/highgui/highgui.hpp>

#include "../core/algo_traits.hpp"
#include "../core/data.hpp"
#include "../core/functional.hpp"
#include "../core/tvmin.hpp"

#include <vpp/vpp.hh>
#include <vpp/utils/opencv_bridge.hh>



using namespace tvmtl;
typedef Manifold< SPHERE, 3 > spheremf_t;
typedef Manifold< EUCLIDIAN, 1 > eucmf_t;

typedef Data< spheremf_t, 2> chroma_t;
typedef Data< eucmf_t, 2> bright_t;

typedef Functional<FIRSTORDER, ISO, spheremf_t, chroma_t> cfunc_t;
typedef Functional<FIRSTORDER, ISO, eucmf_t, bright_t> bfunc_t;

typedef TV_Minimizer< IRLS, cfunc_t, spheremf_t, chroma_t, OMP > ctvmin_t;
typedef TV_Minimizer< IRLS, bfunc_t, eucmf_t, bright_t, OMP > btvmin_t;

```

```
void removeColor(chroma_t& C,const bright_t& B){
    vpp::pixel_wise(C.img_, B.img_, C.inp_) | [&] (auto& c, const auto& b, const bool& i){
        if(i){
            c.setConstant(b[0]);

            if(b[0]!=0) c.normalize();
            else c.setConstant(1.0/256.0);
        }
        if(!std::isfinite(c(0))){
            std::cout << "NaN in RemoveColor" << std::endl;
            std::cout << b << std::endl;
        }
    };
}

void DisplayImage(const char* wname, const chroma_t& C){
        cv::namedWindow( wname, cv::WINDOW_NORMAL );

        // Convert Picture of double to uchar
        vpp::image2d<vpp::vuchar3> vucharimg(C.img_.domain());
        vpp::pixel_wise(vucharimg, C.img_) | [] (auto& i, auto& n) {
            spheremf_t::value_type v = n * (double) std::numeric_limits<unsigned char>::max();
            vpp::vuchar3 vu = vpp::vuchar3::Zero();
            vu[0]=(unsigned char) v[2];
            vu[1]=(unsigned char) v[1];
            vu[2]=(unsigned char) v[0];
            i = vu;
        };

        cv::imshow( wname, vpp::to_opencv(vucharimg));
        cv::waitKey(0);
}

void recombineAndShow(const chroma_t& C, const bright_t B, std::string fname, std::string wname){

        vpp::image2d<vpp::vuchar3> img(C.img_.domain());
        vpp::pixel_wise(img, C.img_, B.img_ ) | [] (auto& i, const auto& c, const auto& b) {
            vpp::vdouble3 v = c * b[0] * std::sqrt(3);

            double max = v.maxCoeff();
            if(max > 1.0) v /= max;

            v *= (double) std::numeric_limits<unsigned char>::max();


            vpp::vuchar3 vu = vpp::vuchar3::Zero();
            vu[0]=(unsigned char) v[2];
            vu[1]=(unsigned char) v[1];
            vu[2]=(unsigned char) v[0];
            i = vu;
        };
        cv::namedWindow( wname, cv::WINDOW_NORMAL );
        cv::imshow( wname, vpp::to_opencv(img));
        cv::waitKey(0);

        cv::imwrite(fname, to_opencv(img));

}

int main(int argc, const char *argv[])
{
        Eigen::initParallel();

        if (argc < 3){
```

```
        std::cerr << "Usage : " << argv[0] << " image [lambda] [threshold]" ↘
            << std::endl;
        return 1;
    }

    double lam=0.01;
    double threshold=0.01;

    if(argc==4){
        lam=atof(argv[2]);
        threshold=atof(argv[3]);
    }

    std::string fname(argv[1]);

    chroma_t myChroma=chroma_t();
    bright_t myBright=bright_t();

    myBright.rgb_readBrightness(argv[1]);
    myBright.findEdgeWeights();

    myChroma.rgb_readChromaticity(argv[1]);
    myChroma.inpaint_=true;
    myChroma.setEdgeWeights(myBright.edge_weights_);
    myChroma.createRandInpWeights(threshold);
    removeColor(myChroma, myBright);

    // Recombine Brightness and Chromaticity parts to view Picture with ↘
        colors removed
    recombineAndShow(myChroma, myBright, "colorless_"+fname, "Colors removed ↘
        Picture");

    cfunc_t cFunc(lam, myChroma);
    cFunc.seteps2(1e-10);

    ctvmin_t cTVMin(cFunc, myChroma);
    cTVMin.first_guess();
    recombineAndShow(myChroma, myBright, "recolored_fg_"+fname, "Recolor ↘
        First Guess");
    DisplayImage("Chromaticity First Guess", myChroma);

    std::cout << "\n\n---CHROMATICITY PART---" << std::endl;

    //std::cout << "Smooth picture to obtain initial state for Newton ↘
        iteration..." << std::endl;
    //cTVMin.smoothening(10);

    std::cout << "Start TV minimization..." << std::endl;
    cTVMin.minimize();


    // Recombine Brightness and Chromaticity parts of recolored Picture
    recombineAndShow(myChroma, myBright, "recolored_"+fname, "Recolored ↘
        Picture");



    return 0;
}
```

Colorization

Listing A.3: ./listings/dti˙tvmin˙prpt˙test3d.cpp

```
#include <iostream>
#include <cstdlib>
#include <string>
#include <sstream>

//#define TV_SPD_EXP_DEBUG
//#define TV_SPD_LOG_DEBUG
```

```
#define TV_DATA_DEBUG
//#define TV_FUNC_DEBUG
//#define TV_FUNC_DEBUG_VERBOSE
//#define TVMTL_TVMIN_DEBUG
//#define TVMTL_TVMIN_DEBUG_VERBOSE
//#define TV_VISUAL_DEBUG

#include "../core/algo_traits.hpp"
#include "../core/data.hpp"
#include "../core/functional.hpp"
#include "../core/tvmin.hpp"
#include "../core/visualization.hpp"

int main(int argc, const char *argv[])
{
        using namespace tvmtl;

        typedef Manifold< SPD, 3 > mf_t;
        typedef Data< mf_t, 3> data_t;
        typedef Functional<FIRSTORDER, ANISO, mf_t, data_t, 3> func_t;
        typedef TV_Minimizer< PRPT, func_t, mf_t, data_t, OMP, 3 > tvmin_t;
        typedef Visualization<SPD, 3, data_t, 3> visual_t;

        data_t myData = data_t();
        int nz, ny, nx;
        nz = std::atoi(argv[2]);
        ny = std::atoi(argv[3]);
        nx = std::atoi(argv[4]);
        myData.readMatrixDataFromCSV(argv[1], nz, ny, nx);


        visual_t myVisual(myData);
        std::stringstream fname;
        std::string nfname;
        fname << "dti3d" << nz << "x" << ny << "x" << ny << ".png";
        nfname = "noisy_" + fname.str();
        myVisual.saveImage(nfname);

        std::cout << "Starting OpenGL-Renderer..." << std::endl;
        myVisual.GLInit("SPD(3) Ellipsoid Visualization");
        std::cout << "Rendering finished." << std::endl;

        double lam=0.7;
        func_t myFunc(lam, myData);
        myFunc.seteps2(0);

        tvmin_t myTVMin(myFunc, myData);

        std::cout << "Start TV minimization..." << std::endl;
        myTVMin.minimize();

        std::string dfname = "denoised(prpt)_" + fname.str();
        myVisual.saveImage(dfname);

        std::cout << "Starting OpenGL-Renderer..." << std::endl;
        myVisual.GLInit("SPD(3) Ellipsoid Visualization");
        std::cout << "Rendering finished." << std::endl;

        return 0;
}
```

3D DTI TV minimization

Appendix B

Derivative Computations

Bibliography

[1] Teem toolkit. http://teem.sourceforge.net/. Accessed: 2015-09-20. (Cited on page 36.)

[2] A. Barmpoutis A., B. C. Vemuri, T. M. Shepherd, and J. R. Forder. Tensor splines for interpolation and approximation of dt-mri with applications to segmentation of isolated rat hippocampi. *IEEE Trans Med Imaging*, 26(11):1537–1546, 2007. (Cited on page 51.)

[3] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004. (Cited on page 27.)

[4] P. A. Absil, R. Mahony, and R. Sepulchre. *Riemannian Geometry of Grassmann Manifolds with a View on Algorithmic Computation*, volume 80. Springer Netherlands, January 2004. (Cited on pages 19 and 22.)

[5] P. A. Absil, R. Mahony, and R. Sepulchre. *Optimization algorithms on matrix manifolds*. Princeton University Press, 2009. (Cited on pages 9, 12 and 21.)

[6] Awad H. Al-Mohy, Nicholas J. Higham, and Samuel D. Relton. Computing the fréchet derivative of the matrix logarithm and estimating the condition number. *SIAM Journal on Scientific Computing*, 35(4):C394–C410, 2013. (Cited on page 24.)

[7] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. (Cited on page 27.)

[8] Saad Ali and Mubarak Shah. A lagrangian particle dynamics approach for crowd flow segmentation and stability analysis. In *CVPR*. IEEE Computer Society, 2007. (Cited on page 49.)

[9] V. Arsigny, P. Fillard, X. Pennec, and N. Ayache. Geometric means in a novel vector space structure on symmetric positive definite matrices. *SIAM Journal on Matrix Analysis and Applications*, 29(1):328–347, 2007. (Cited on page 9.)

[10] Vincent Arsigny, Pierre Fillard, Xavier Pennec, and Nicholas Ayache. Geometric means in a novel vector space structure on symmetric positive[U+2010]definite matrices. *SIAM Journal on Matrix Analysis and Applications*, 29(1):328–347, 2007. (Cited on page 61.)

[11] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6):679–698, 1986. (Cited on page 46.)

[12] M. Cavegn. Total variation regularization for geometric data. *Master thesis, ETH Zürich*, 2013. (Cited on page 26.)

[13] A. Chambolle and P.-L. Lions. Image recovery via total variation minimization and related problems. *Numerische Mathematik*, 76(2):167–188, 1997. (Cited on page 5.)

[14] PatrickL. Combettes and Jean-Christophe Pesquet. Proximal splitting methods in signal processing. In Heinz H. Bauschke, Regina S. Burachik, Patrick L. Combettes, Veit Elser, D. Russell Luke, and Henry Wolkowicz, editors, *Fixed-Point Algorithms for Inverse Problems in Science and Engineering*, volume 49 of *Springer Optimization and Its Applications*, pages 185–212. Springer New York, 2011. (Cited on page 9.)

[15] Alan Edelman, Tomás A. Arias, and Steven T. Smith. The geometry of algorithms with orthogonality constraints. *SIAM J. Matrix Anal. Appl.*, 20(2):303–353, April 1999. (Cited on page 19.)

[16] Matthieu Garrigues and Antoine Manzanera. Video++, a modern image and video processing C++ framework. Technical report, ENSTA-ParisTech, France, 2014. (Cited on page 33.)

[17] P. Grohs and M. Sprecher. Total variation regularization by iteratively reweighted least squares on hadamard spaces and the sphere. Technical Report 2014-39, Seminar for Applied Mathematics, ETH Zürich, Switzerland, 2014. (Cited on pages 11, 14, 26 and 55.)

[18] P. Grohs, M. Sprecher, and T. Yu. Scattered manifold-valued data approximation. Technical Report 23, Seminar for Applied Mathematics, ETH Zürich, Switzerland, 2014. (Cited on page 7.)

[19] Nicholas J. Higham. Evaluating padé approximants of the matrix logarithm. *SIAM Journal on Matrix Analysis and Applications*, 22(4):1126–1135, 2001. (Cited on page 24.)

[20] S. H. Kang and R. March. Variational models for image colorization via chromaticity and brightness decomposition. *IEEE Transactions on Image Processing*, 16(9):2251–2261, 2007. (Cited on page 7.)

[21] H. Karcher. Riemannian center of mass and mollifier smoothing. *Communications on Pure and Applied Mathematics*, 30(5):509–541, 1977. (Cited on pages 10 and 16.)

[22] Julien Lefevre and Sylvain Baillet. Optical flow and advection on 2-riemannian manifolds: A common framework. *IEEE Trans. Pattern Anal. Mach. Intell.*, 30(6):1081–1092, 2008. (Cited on page 61.)

[23] Stanley Osher Leonid I. Rudin and Emad Fatemi. Nonlinear total variation based noise removal algorithms. *Physica D*, 60:259–168, 1992. (Cited on pages 5 and 7.)

[24] Ray Lischner. *Exploring C++11*. Expert's voice in C++. Apress, New York, 2013. (Cited on page 29.)

[25] Bruce D. Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'81, pages 674–679, San Francisco, CA, USA, 1981. Morgan Kaufmann Publishers Inc. (Cited on page 49.)

[26] Jan R. Magnus and Heinz Neudecker. *Matrix differential calculus with applications in statistics and econometrics*. J. Wiley & Sons, Chichester, New York, Weinheim, 1999. (Cited on page 16.)

[27] M. Nägelin. Total variation regularization for tensor valued images. *Bachelor thesis, ETH Zürich*, 2014. (Cited on pages 9 and 26.)

[28] Neal Parikh and Stephen Boyd. Proximal algorithms. *Found. Trends Optim.*, 1(3):127–239, January 2014. (Cited on page 9.)

[29] P. Rodriguez and B. Wohlberg. An iteratively reweighted norm algorithm for minimization of total variation functionals. *IEEE Signal Processing Letters*, 14(12):948 – 951, 2007. (Cited on page 10.)

[30] Hiroyuki Sato and Toshihiro Iwai. Optimization algorithms on the grassmann manifold with application to matrix eigenvalue problems. *Japan Journal of Industrial and Applied Mathematics*, 31(2):355–400, 2014. (Cited on page 19.)

[31] Andreas Wedel and Daniel Cremers. *Stereo Scene Flow for 3D Motion Analysis.* Springer, 2011. (Cited on pages 6, 27 and 61.)

[32] A. Weinmann, L. Demaret, and M. Storath. Total variation regularization for manifold-valued data. *SIAM Journal on Imaging Sciences*, 7(4):2226–2257, 2014. (Cited on pages 7, 9, 10 and 56.)

[33] T. Y. Zhang and C. Y. Suen. A fast parallel algorithm for thinning digital patterns. *Commun. ACM*, 27(3):236–239, March 1984. (Cited on page 48.)