A 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 acquisiation, transmission and processing.y 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 aesthetical 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 [**?**] 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 $u_0 : \Omega \subset \mathbb{R} \to \mathbb{R}$ desribe 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 pictures 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:

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

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

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

$$\frac{1}{2}\|u - u_0\|_2^2 + \lambda \int_\Omega |\nabla u| \tag{1.3}$$

1.1.1 Edge preservation

A basic intuition why the L^1 norm in (1.3) is besser suited for conserving sharp discontinuities such as edges can be seen from the following plot.

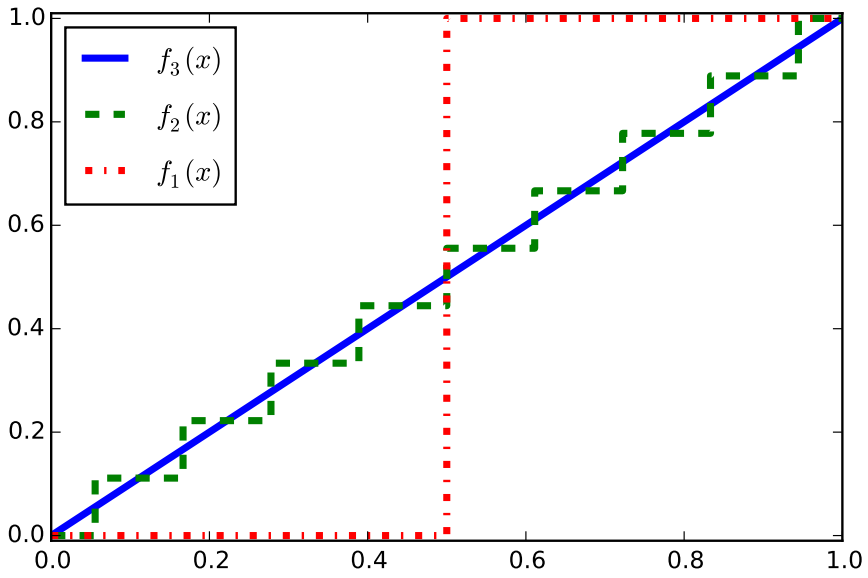

Figure 1.1: Plots of three functions with ($N = 1,\ 10,\ 100$) steps and a total variation equal to 1.0

Function	$\int_{[0,1]} \lvert\nabla f\rvert$	$\int_{[0,1]} \lvert\nabla f\rvert^2$
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 favors continuous transitions such as f_3 rather than the sudden jump in f_1 whereas the total variation is the same for all cases.

1.2 Color Images

The next step in the development of image denoising algorithms was their generalization to color images. From a mathemtical 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, Chang and Kang [**?**] showed the closest resemblance to human perception. In this case we can take $C = S^2 \times [0,1]$ such that the chromatictiy takes values on the sphere S^2 considered as a submanifold of the euclidian space $\mathbb{R}^3$, while the brightness is real-valued, as in the case of graysale 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 their matrices. This data arises in a variety of application 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 work

In this work 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 [**?**] as well as a proximal point algorithm by Weinmann et al [**?**]. We extend the implementation to 3D images cubes, the Grassmann manifold and also provides 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. After that the chapter 3 introduces the library itself in particular its capabilieties, 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 minimizers.

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 Riemannian Newton Method

2.2 Algorithms

2.2.1 IRLS

2.2.2 Proximal Point

2.3 Manifolds

2.3.1 Euclidian

2.3.2 Sphere S^n

2.3.3 Special Orthogonal Group SO(n)

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 [**?**]. This allows us to express the derivatives as combinations of simple Kronecker product of the arguments which also is very straightforward and compact to implement. The detailed derivations can be found in the appendix **??** while here we only represent the final results.
For the second derivative with respect to the first argument one readily arrives at

$$\frac{\partial^2 d^2(X,Y)}{\partial X \partial X} = -2\left[\left(\left(\log X^T Y\right)^T \otimes \mathbb{1}_n\right) + (\mathbb{1}_n \otimes X)\,\mathrm{D}\log(X^T Y)\left(Y^T \otimes \mathbb{1}_n\right) K_{nn}\right], \tag{2.1}$$

where K_{nn} denotes the commutator matrix which transforms the columnwise vectorization of a matrix A to the vectorization of its tranpose A^T.

The mixed derivative is given by

$$\frac{\partial^2 d^2(X,Y)}{\partial X \partial Y} = -2\,(\mathbb{1}_n \otimes X)\,\mathrm{D}\log(X^T Y)\left(\mathbb{1}_n \otimes X^T\right), \tag{2.2}$$

These expressions are quasi-analytical: Matrix logarithms, exponentials and the Frechet derivative of the logarithms need to be evaluated numerically. Details concerning the implementation of the latter are postponed to section **??**.

2.3.4 Symmetric Positive Definite Matrices SPD(n)

Second derivatives of the distance function

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

$$T_1 = \sqrt{X} \tag{2.3}$$
$$T_2 = T^{-1} \tag{2.4}$$
$$T_3 = T_2 Y T_2 \tag{2.5}$$
$$T_4 = \log(T_3) \tag{2.6}$$
$$\frac{\partial^2 d^2(X,Y)}{\partial X \partial X} = 2\left[\left(T_2 T_4^T \otimes \mathbb{1}\right) + \left(\mathbb{1} \otimes T_2 T_4\right) + \left(T_2 \otimes T_2\right) \mathrm{D}\log(T_3)\left(\left(T_2 Y \otimes \mathbb{1}\right) + \left(\mathbb{1} \otimes T_2 Y\right)\right)\right]\left(T_2 \otimes T_2\right) \mathrm{D}(\sqrt{(}X)) \tag{2.7}$$

and for the mixed derivatives

$$\frac{\partial^2 d^2(X,Y)}{\partial X \partial Y} = -2\left(X^{-\frac{1}{2}} \otimes X^{-\frac{1}{2}}\right) \mathrm{D}\log\left(X^{-\frac{1}{2}} Y X^{-\frac{1}{2}}\right)\left(X^{-\frac{1}{2}} \otimes X^{-\frac{1}{2}}\right) \tag{2.8}$$

2.3.5 Grassmanian Gr(N,P)

Distance function

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

$$g(X,Y) = \mathrm{Tr} X^T Y \tag{2.9}$$

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

$$d_g^2(X,Y) = \|\theta\|_2^2 = \sum_{i=1}^{p} \theta_i^2, \tag{2.10}$$

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

$$X^T Y = U \Sigma V^T = U \cos\Theta V^T \tag{2.11}$$
$$\Sigma = \mathrm{diag}(\sigma_1, \cdots, \sigma_p) \tag{2.12}$$
$$\Theta = \mathrm{diag}(\theta_1, \cdots, \theta_p) = \mathrm{diag}(\arccos\sigma_1, \cdots, \arccos\sigma_p) \tag{2.13}$$

The distance function (2.10) has the disadvantage that due to the occurence of the cosine no analytic derivatives can be computed.

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

$$d_P^2(X,Y) = \frac{1}{2}\|XX^T - YY^T\|_F^2 = \sum_{i=1}^{p} \sin^2\theta_i \tag{2.14}$$

First derivatives of the distance function

$$\frac{\partial d^2(X,Y)}{\partial X} = 2\left(XX^T - YY^T\right)X \tag{2.15}$$

Second derivatives of the distance function

$$\frac{\partial^2 d^2(X,Y)}{\partial X \partial X} = 2\left[\left(X^T X \otimes \mathbb{1}_n\right) + \left(\mathbb{1}_p \otimes \left(XX^T - YY^T\right)\right) + \left(X^T \otimes X\right) K_{np}\right] \tag{2.16}$$

The mixed derivative is given by

$$\frac{\partial^2 d^2(X,Y)}{\partial X \partial Y} = -2\left[\left(X^T Y \otimes \mathbb{1}_n\right) + \left(X^T \otimes Y\right) K_{np}\right] \tag{2.17}$$

2.4 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

$$f(X + E) - f(X) - L_f(X, E) = \wr(\|E\|). \tag{2.18}$$

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

$$L_{f\circ g}(X, E) = L_f(g(X)), L_g(X, E)) \tag{2.19}$$
$$L_f(X, L_{f^{-1}}(f(X), E)) = E \tag{2.20}$$

As we did in our formulation of the derivatives of the distance function, it can also be represented in the Kronecker form in which 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

$$\operatorname{vec}(L_f(X, E)) = K_f(X) \operatorname{vec}(E) \tag{2.21}$$

where $\operatorname{vec} : \mathbb{R}^{n\times n} \to \mathbb{R}^{n^2}$ denotes the columnwise vectorization operator.

2.4.1 Derivative of the matrix square root

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

$$L_{X^2}(X, E) = XE + EX. \tag{2.22}$$

Applying the inverse function theorem consequently leads to

$$L_{X^2}(X^{\frac{1}{2}}, L_{X^{\frac{1}{2}}}(X, E)) = X^{\frac{1}{2}} L_{X^{\frac{1}{2}}}(X, E) + L_{X^{\frac{1}{2}}}(X, E)X^{\frac{1}{2}} = E, \tag{2.23}$$

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

$$X^{\frac{1}{2}} L + LX^{\frac{1}{2}} = E, \quad L := L_{X^{\frac{1}{2}}}(X, E). \tag{2.24}$$

The Kronecker representation $K_{X^{\frac{1}{2}}}$ can now be obtained be using the vectorization operator on both sides of the equation and rearrange the term to the form (2.21) which leads to

$$K_{X^{\frac{1}{2}}}(X) = \left[\left(\mathbb{1} \otimes X^{\frac{1}{2}}\right) + \left(X^{\frac{1}{2}T} \otimes \mathbb{1}\right)\right]^{-1}. \tag{2.25}$$

However, this straightforward approach has the disadvantage that the inverse of a $n^2 \times n^2$ matrix need 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 (2.24), 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, tranposed Fréchet derivatives:

$$\left(K_{X^{\frac{1}{2}}}\right)_{in+j,\cdot} = \operatorname{vec}\left(L_{X^{\frac{1}{2}}}(X, E^{ij})^T\right) \tag{2.26}$$

2.4.2 Derivative of the matrix logarithm

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

$$\log(X) = 2\log(X^{\frac{1}{2}}) \tag{2.27}$$

is necessary.

Application of the chain rule leads to

$$L_{\log}(X, E_0) = 2\log\left(X^{\frac{1}{2}}, L_{X^{\frac{1}{2}}}(X, E_0)\right). \tag{2.28}$$

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

$$X^{\frac{1}{2}}E_1 + E_1X^{\frac{1}{2}} = E_0. \tag{2.29}$$

Repeating the procedure s times results in

$$L_{\log}(X, E_0) = 2^s L_{\log}\left(X^{\frac{1}{2^s}}, E_s\right) \tag{2.30}$$

$$X^{\frac{1}{2^i}}E_i + E_iX^{\frac{1}{2^i}} = E_{i-1}, \quad i = 1, \ldots, s \tag{2.31}$$

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

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

$$r_m(X) = \sum_{j=1}^{m} \alpha_j^{(m)}(\mathbb{1} + \beta_j^{(m)}X)^{-1}X \tag{2.32}$$

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

The derivative of (2.32) is then easily computed as

$$L_{r_m}(X, E) = \sum_{j=1}^{m} \alpha_j^{(m)}(\mathbb{1} + \beta_j^{(m)}X)^{-1}E(\mathbb{1} + \beta_j^{(m)}X)^{-1} \tag{2.33}$$

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

$$L_{\log}(X, E) \approx 2^s L_{r_m}\left(X^{\frac{1}{2^s}} - \mathbb{1}, E_s\right). \tag{2.34}$$

For the implementation of (2.34) we use algorithm 5.1 from [**?**] with fixed $m = 7$. The Kronecker representation is then constructed as in the square root case.

Chapter 3

The TVMT 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 Euclidian 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 $SPD(n) = \left\{S \in \mathbb{R}^{n\times n} : S = S^T, x^T Sx > 0 \; \forall x \in \mathbb{R}^n\right\}$
- Grasmann manifold $Gr(n,p) =$

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 [**?**], [**?**] and [**?**], one of the most important goals was of course a faster implementation with a smaller memory footprint. On the test platform with two hyperthreaded 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 smartphones 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 a 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 readibility and maintainability of the code since tensor product usualluay 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 represention, 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 a support for a new

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

For the TVMT 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. 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 specializaiton 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 requiere 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 the 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 [**?**].

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 multithreading 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 posible 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. At the time of writing, this unfortunately also includes 3D images, such that we implemented 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 to requires iteration over an image container. For most computations, in 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 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. This is illustrated by the following short listing 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 XD1 = img_type(without_last_col);
vpp::pixel_wise(XD1, weightsX_ | without_last_col, data_.img_ | without_last_col, ↘
    data_.img_ | without_first_col) | 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 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 arithmatic operation is simultaneously applied to all variables in the register such that theoretically the amount of floating

point operations is multiplied by the number of variables fitting in the registers.

In order to really achieve this speedup the data must be aligned in memory which means the adress of pixels in memory must always be multiple of the SIMD register size. Fortunately, that issue is handeld by the VPP and Eigen libraries enabling the compiler to perform the necessary vectorization optimizations.

3.2.3 C++ techniques

The TVMT 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.

Lambda functions

A lambda function is basically a locally defined function object, which is able to capture variable from the surrounding scope that can but need not to be named. The corresponding Matlab language construct is an anonymous function or funtion handle, usually defined using the operator. The following listing shows the basic definitions and uses 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 TVMT 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 introduce 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 usually implemented using some list type (in C va_list), which adds additional overhead, whereas in C++ it is realized via some special kind of template metaprogramming which is recursive in nature. The recursion, in turn, is resolved at compile-time an leads to code that is acutally 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 TVMTL are the implementation of the Karcher mean, needed for the proximal point implementation, and TVMTL's own version of the 3D pixel-wise kernels.

3.3 Components

3.3.1 Manifold classes

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 classes contains functionality necessary for TV minimization using either the IRLS or proximal point algorithm and furthermore some addtional operations that are needed for supporting tasks like interpolation and smoothing. The classes 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 classes in other projects requiring similiar 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 manifolds 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 dimension 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 postprocessing for interpolation is necessary.

Type definitions

To enable 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 expontial 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 expresion only, hence there is no need to use this more complicated construction on a general basis.
The convex_combinations method denotes a point z on the manifold by following a unit time geodesic connecting the points x and y for a time t.

Karcher means

```
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 eventuallys requires copying and is consequently slower but has an overloaded version which allows to compute a wighted 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 Euclidian 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 threedimensional 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 [**?**] 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 effciently 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 rowwise flattened, such that each line of the input file contains exactly one pixel. The order of the list is also rowwise for 2D or slice- than rowwise 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 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 parameters disc determines whether the isotropic or the anisotropic version is to be used. Please not that for thr proximal point algorithm only anistropic is available. Finally, the last parameter speciefies 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 sparse linear system that needs to 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 enumerated 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 2, 3 and 5 merely trigger a recomputation while the last three functions return references to these quantities.

3.3.4 TV Minimizer class

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

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 libary 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 full example to this 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.

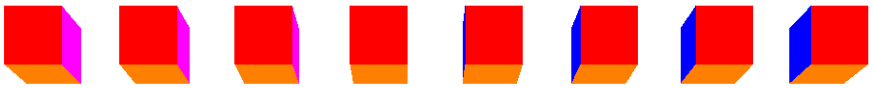

Figure 3.1: SO(3) Visualization as oriented, colored cubes

SPD(3) Visualization

For three-dimensional SPD 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 visualized by ellipsoids using three degerees of freedom for their orientation in space and the remaining three for length of its semi-axis.
Starting with the unit sphere centered at the origin again, we compute eigenvector and eigenvalues for every SPD matrix. Due to the SPD propery 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 diffusity 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 colorspace, such that clusters of similar orientations can be more easily visually distinguished.

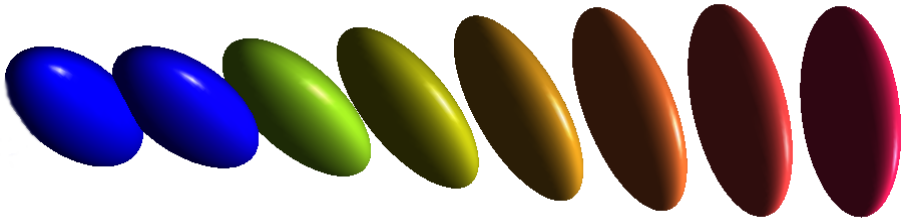

Figure 3.2: SPD(3) Visualization as oriented, colored ellipsoids

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.//

3D Volume image rendering

The volume image renderer just transforms the data to a 3d texture which is then mapped onto cube rotating about the z-axis. Plasticity is created by setting the alpha channel of each displayed voxel to the intesity value of the corresponding data voxel, such that dark areas are more transparent. The following 3.3 shows the rendered volume from different angles.

Figure 3.3: 3D Volume image using texture based rendering

3.3.6 Utility functions

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 requiered 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.

3.3.7 External Components

Maybe put somewhere else...

Linear algebra handler: Eigen

Data storage and processing: Video++

3.4 Using TVTML

3.4.1 Prerequisites

The main dependencies of TVTML are the *Eigen* C++ template library for linear algebra and the *Video++* video and image processing library. Those libraries, as well as TVMTL's core functionality are provided as header-only libraries. There are, however, some more recommended static libraries that are recommended to speed up the computation, enable easy I/O or 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, TVMTL also relies on the *CMake* installation tool for installation and compilation of user code using TVMTL.

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

- 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 sytem in the IRLS algorithm

3.4.2 Installation

3.4.3 Compilation of own projects using CMake

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 readabillity 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 usign 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 3.4 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 Euclidian $\mathbb{R}^3$ and a corresponding 2D image container

Listing 3.5 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 3.6 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 specified manifold and data class types need to be given as template parameter. The last template parameter, the dimension of the data, has default value 2 and can also be omitted in this case.

Listing 3.7 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 3.8 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 paramater 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 **??**.

Colorization using color inpainting

DT-MRI data denoising and visualization

Chapter 4

Numerical Experiments

4.1 Color image denoising

4.1.1 Grayscale

4.1.2 Color

4.1.3 Color inpainting

4.1.4 Volume images

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

4.2.1 Synthetic data

4.2.2 Fingerprint orientation data

4.2.3 Calculation of a dense flow field

4.3 SPD(3) image data

4.3.1 Diffusion Tensor MRI images

4.3.2 3D DT MRI data

4.4 Comparison IRLS and Proximal Point minimizers

4.5 Sensitivity to variations of the starting value

Chapter 5

Outlook

5.1 Extensions

- new manifols and functionals
- automatic differentiation for matrix valued functions
- distributed memory architecture

5.2 Recursive computation on subdomains

Bibliography