Archive of Numerical Software
vol. 1, no. S1, pages 1–13
© 2011

Using DUNE-ACFEM for Non-smooth Minimization of Bounded Variation Functions

Martin Alkämper and Andreas Langer[*]

Received: Jan 1, 1677; **final revision:** Jun 1, 1677; **published:** Jun 13, 1677.

Abstract: The utility of DUNE-ACFEM is demonstrated to work well for solving a non-smooth minimization problem over bounded variation functions by implementing a primal-dual algorithm. The implementation is based on the simplification provided by DUNE-ACFEM. Moreover, the convergence of the discrete minimizer to the continuous one is shown theoretically.

1 Introduction

The DUNE-framework [7] and in particular the discretization module DUNE-FEM [10] provides the means to handle discrete functions, operators and solvers on different grids. Still, implementing complicated partial differential equations (PDEs) and their solvers is cumbersome and tedious. The DUNE- module DUNE-ACFEM aims to simplify the usage of DUNE-FEM by defining expression templates for discrete functions and PDE models. This allows to linearly combine discrete functions and PDE models. Additionally it supports parallel and adaptive finite-element schemes on continuous discrete functions for the predefined and combined models. [12]
The flexibility of DUNE (and DUNE-FEM, DUNE-ACFEM) allows e.g. to exchange discrete spaces by a single line of code or to change the gridtype and linear solvers.
We will demonstrate the usefulness of DUNE-ACFEM by minimizing a non-smooth functional consisting of a combined L^1/L^2-data fidelity term and a total variation term. Such an optimization problem has been shown to effectively remove Gaussian and salt-and-pepper noise, see [13, 15]. In order to compute an approximate solution we use the primal-dual algorithm proposed in [9]. For the numerical implementation we discretize using finite-element spaces defined over locally refined conforming grids. Motivated by the works [3, 4, 5, 14], where the considered functional is composed solely of an L^2-data term and a total variation term, we refine the grid adaptively using an a priori criterion. Similar as in [3] we show for the considered minimization problem, that a minimizer over a finite element space converges to a minimizer in the space of functions of bounded variation as the mesh-size goes to 0.
In contrast to previous works [3, 4, 5, 14], we consider an additional non-smooth L^1-data term in the objective, which has to be treated carefully. Moreover due to the use of DUNE-ALUGRID [1] and the capabilities of DUNE-ACFEM the resulting algorithm is intrinsically parallelized by domain decomposition.

[*]Institute for Applied Analysis and Numerical Simulation, University of Stuttgart, Pfaffenwaldring 57, 70569 Stuttgart, Germany, `martin.alkaemper@mathematik.uni-stuttgart.de`, `andreas.langer@mathematik.uni-stuttgart.de`

2 Problem Formulation

We consider the following problem

$$\min_{v\in BV(\Omega)\cap L^2(\Omega)} J_{\alpha_1,\alpha_2}(v) := \alpha_1\|v-g\|_{L^1(\Omega)} + \alpha_2\|v-g\|^2_{L^2(\Omega)} + |Dv|(\Omega) \tag{1}$$

where $\Omega \subset \mathbb{R}^d$, $d \in \mathbb{N}$, is an open bounded set with Lipschitz boundary, $g \in L^2(\Omega)$ is a given datum, $\alpha_i \geq 0$ for $i = 1, 2$ with $\alpha_1 + \alpha_2 > 0$ and $BV(\Omega) \subset L^1(\Omega)$ denotes the space of functions with bounded variation. that is, $v \in BV(\Omega)$ if and only if

$$|Dv|(\Omega) := \sup\left\{\int_\Omega v \operatorname{div}\vec{\phi}dx,\ \vec{\phi}\in C_0^\infty(\Omega,\mathbb{R}^d),\ \|\,|\vec{\phi}|_2\|_{C^0(\Omega)} \leq 1\right\} \tag{2}$$

is finite, see [2, 11]. The space $BV(\Omega)$ endowed with the norm $\|v\|_{BV(\Omega)} = \|v\|_{L^1(\Omega)} + |Dv|(\Omega)$ is a Banach space [11]. For $\alpha_2 > 0$ the minimization problem (1) admits a unique solution owing to the strict convexity of the quadratic term [15]. Note, that if $\alpha_1 = 0$ in (1), then we obtain the functional used in [3, 4, 5, 14].

2.1 Discretization

Let $(\mathcal{T}_h)_{h>0}$ be a sequence of regular triangulations of Ω with (maximal) diameter $h = \max_{T\in\mathcal{T}_h} \operatorname{diam}(T)$. We define the following finite element spaces

$$\begin{aligned}\mathcal{L}^0(\mathcal{T}_h) &= \{q_h \in L^1(\Omega) : q_h|_T \text{ is constant for each } T \in \mathcal{T}_h\}\\ \mathcal{S}^1(\mathcal{T}_h) &= \{v_h \in C(\overline{\Omega}) : v_h|_T \text{ is affine for each } T \in \mathcal{T}_h\}.\end{aligned}$$

The nodal interpolant $\mathcal{I}_h v \in \mathcal{S}^1(\mathcal{T}_h)$ of a function $v \in W^{2,p}$, with $\frac{d}{2} < p \leq \infty$ or $p = 1$ if $d = 2$, satisfies

$$\|v - \mathcal{I}_h v\|_{L^p(\Omega)} + h\|\nabla(v - \mathcal{I}_h v)\|_{L^p(\Omega)} \leq c_{\mathcal{I}_h} h^2 \|D^2 v\|_{L^p(\Omega)},$$

where $c_{\mathcal{I}_h} > 0$; cf. [8].

We recall, that the space $BV(\Omega)$ is continuously embedded in $L^p(\Omega)$ for $1 \leq p \leq \frac{d}{d-1}$, i.e., there is a constant $c_{BV} > 0$ such that $\|v\|_{L^p(\Omega)} \leq c_{BV}\|v\|_{BV} = c_{BV}\left(\|v\|_{L^1(\Omega)} + |Dv|(\Omega)\right)$ for any $v \in BV(\Omega)$. For $1 \leq p < \frac{d}{d-1}$ this embedding is compact; cf. [2]. Smooth functions are dense in $BV(\Omega) \cap L^p(\Omega)$, $1 \leq p < \infty$, with respect to strict convergence in the sense that for $v \in BV(\Omega) \cap L^p(\Omega)$ and $\delta > 0$ there exists $\varepsilon_0 > 0$ and functions $(v_\varepsilon)_{\varepsilon>0} \subset C^\infty \cap BV(\Omega) \cap L^p(\Omega)$ such that for all $\varepsilon \leq \varepsilon_0$ we have

$$\|\nabla v_\varepsilon\|_{L^1(\Omega)} \leq |Dv|(\Omega) + c_0\delta, \tag{3}$$

$$\|v - v_\varepsilon\|_{L^p(\Omega)} \leq c_1\delta, \tag{4}$$

$$\|D^2 v_\varepsilon\|_{L^p(\Omega)} \leq c\varepsilon^{-2}\|v\|_{L^p(\Omega)}. \tag{5}$$

Now we are able to show the following convergence result, which follows similar ideas as the proof of [3, Theorem 3.1].

Theorem 2.1 *Let $u \in BV(\Omega)\cap L^2(\Omega)$ be a minimizer of the function J_{α_1,α_2} and $u_h \in \arg\min_{v\in\mathcal{S}^1(\mathcal{T}_h)} J_{\alpha_1,\alpha_2}(v)$. Then we have that $J_{\alpha_1,\alpha_2}(u_h) \to J_{\alpha_1,\alpha_2}(u)$ as $h \to 0$. If additionally $\alpha_2 > 0$, then $u_h \to u$ in $L^2(\Omega)$ as $h \to 0$.*

Proof 1 *By the optimality of u we have $J_{\alpha_1,\alpha_2}(u_h) - J_{\alpha_1,\alpha_2}(u) \geq 0$. For $\delta > 0$ let $u_\varepsilon \in C^\infty(\Omega) \cap L^2(\Omega)$ as above and $\mathcal{I}_h u_\varepsilon$ its nodal interpolant, i.e., $\mathcal{I}_h u_\varepsilon \in \mathcal{S}^1(\mathcal{T}_h)$. Then we deduce*

$$\begin{aligned}J_{\alpha_1,\alpha_2}(u_h) - J_{\alpha_1,\alpha_2}(u) &\leq J_{\alpha_1,\alpha_2}(\mathcal{I}_h u_\varepsilon) - J_{\alpha_1,\alpha_2}(u)\\ &= \|\nabla\mathcal{I}_h u_\varepsilon\|_{L^1(\Omega)} + \alpha_1\|\mathcal{I}_h u_\varepsilon - g\|_{L^1(\Omega)} + \alpha_2\|\mathcal{I}_h u_\varepsilon - g\|_{L^2(\Omega)}\\ &\qquad - |Du|(\Omega) - \alpha_1\|u-g\|_{L^1(\Omega)} - \alpha_2\|u-g\|_{L^2(\Omega)}.\end{aligned}$$

Using (3) *and* $\|\mathcal{I}_h u_\varepsilon - g\|^2_{L^2(\Omega)} - \|u-g\|^2_{L^2(\Omega)} = \int_\Omega (\mathcal{I}_h u\varepsilon - u)(\mathcal{I}_h u_\varepsilon + u - 2g)$ *we obtain*

$$\begin{aligned} J_{\alpha_1,\alpha_2}(u_h) - J_{\alpha_1,\alpha_2}(u) \le \|\nabla \mathcal{I}_h u_\varepsilon\|_{L^1(\Omega)} - \|\nabla u_\varepsilon\|_{L^1(\Omega)} + c_0\delta + \alpha_1\|\mathcal{I}_h u_\varepsilon - g\|_{L^1(\Omega)} \\ - \alpha_1\|u-g\|_{L^1(\Omega)} + \alpha_2 \int_\Omega (\mathcal{I}_h u\varepsilon - u)(\mathcal{I}_h u_\varepsilon + u - 2g). \end{aligned}$$

By the triangle-inequality and the Cauchy-Schwarz inequality we get

$$\begin{aligned} J_{\alpha_1,\alpha_2}(u_h) - J_{\alpha_1,\alpha_2}(u) &\le \|\nabla(\mathcal{I}_h u_\varepsilon - u_\varepsilon)\|_{L^1(\Omega)} + c_0\delta + \alpha_1\|\mathcal{I}_h u_\varepsilon - g - (u-g)\|_{L^1(\Omega)} \\ &\quad + \alpha_2\|\mathcal{I}_h u\varepsilon - u\|_{L^2(\Omega)}(\|\mathcal{I}_h u_\varepsilon\|_{L^2(\Omega)} + \|u\|_{L^2(\Omega)} + 2\|g\|_{L^2(\Omega)}) \\ &= \|\nabla(\mathcal{I}_h u_\varepsilon - u_\varepsilon)\|_{L^1(\Omega)} + c_0\delta + \alpha_1\|\mathcal{I}_h u_\varepsilon - u_\varepsilon + u_\varepsilon - u\|_{L^1(\Omega)} \\ &\quad + \alpha_2\|\mathcal{I}_h u_\varepsilon - u_\varepsilon + u_\varepsilon - u\|_{L^2(\Omega)}(\|\mathcal{I}_h u_\varepsilon\|_{L^2(\Omega)} + \|u\|_{L^2(\Omega)} + 2\|g\|_{L^2(\Omega)}) \\ &\le \|\nabla(\mathcal{I}_h u_\varepsilon - u_\varepsilon)\|_{L^1(\Omega)} + c_0\delta + \alpha_1\|\mathcal{I}_h u_\varepsilon - u_\varepsilon\|_{L^1(\Omega)} + \|u_\varepsilon - u\|_{L^1(\Omega)} \\ &\quad + \alpha_2(\|\mathcal{I}_h u_\varepsilon - u_\varepsilon\|_{L^2(\Omega)} + \|u_\varepsilon - u\|_{L^2(\Omega)})(\|\mathcal{I}_h u_\varepsilon\|_{L^2(\Omega)} + \|u\|_{L^2(\Omega)} + 2\|g\|_{L^2(\Omega)}) \end{aligned}$$

The bound $\|\mathcal{I}_h u_\varepsilon\|_{L^2(\Omega)} + \|u\|_{L^2(\Omega)} + 2\|g\|_{L^2(\Omega)} \le \tilde{c}$, *which holds provided that* $h \le c\varepsilon$, *and the nodal interpolant estimate yield*

$$\begin{aligned} J_{\alpha_1,\alpha_2}(u_h) - J_{\alpha_1,\alpha_2}(u) \le c_{\mathcal{I}_h} h\|D^2 u_\varepsilon\|_{L^1(\Omega)} + c_0\delta + c_{\mathcal{I}_h}\alpha_1 h^2\|D^2 u_\varepsilon\|_{L^1(\Omega)} + \alpha_1\|u_\varepsilon - u\|_{L^1(\Omega)} \\ + \tilde{c}\alpha_2(c_{\mathcal{I}_h} h^2\|D^2 u_\varepsilon\|_{L^2(\Omega)} + \|u_\varepsilon - u\|_{L^2(\Omega)}). \end{aligned}$$

Using (4), (5), *and the bound* $\|u\|_{L^2(\Omega)} \le \tilde{c}$ *we get*

$$J_{\alpha_1,\alpha_2}(u_h) - J_{\alpha_1,\alpha_2}(u) \le C(C_1\frac{h}{\varepsilon^2} + C_2\delta + C_3\frac{h^2}{\varepsilon^2} + C_4\delta + C_5\frac{h^2}{\varepsilon^2} + C_6\delta).$$

We choose $\delta = \frac{h}{\varepsilon^2}$ *and deduce for* $h \to 0$ *that* $\delta \to 0$ *and hence* $J_{\alpha_1,\alpha_2}(u_h) \to J_{\alpha_1,\alpha_2}(u)$.
If $\alpha_2 > 0$, *then* J_{α_1,α_2} *is strictly convex and it follows that*

$$J_{\alpha_1,\alpha_2}(u_h) - J_{\alpha_1,\alpha_2}(u) \ge \alpha_2\|u_h - u\|^2_{L^2(\Omega)},$$

cf. [15, *Lemma 3.8*]. *Hence* $u_h \to u$ *for* $h \to 0$. □

2.2 Steepest Descent Algorithm

By the definition of the total variation (2) an equivalent formulation of (1) reads

$$\min_{v\in BV(\Omega)\cap L^2(\Omega)} \max_{\vec{p}\in C_0^\infty(\Omega,\mathbb{R}^d),|\vec{p}|_2\le 1} \alpha_1\|v-g\|_{L^1(\Omega)} + \alpha_2\|v-g\|^2_{L^2(\Omega)} + \int_\Omega v \operatorname{div}\vec{p}. \tag{6}$$

We discretize using finite element spaces $\mathcal{U}$ and $\mathcal{P}$ (e.g. $\mathcal{S}^1(\mathcal{T}_h)$ or $\mathcal{L}^0(\mathcal{T}_h)$) for the primal variable u and the dual variable $\vec{p}$ respectively.

Following the ideas of Chambolle and Pock [9, Algorithm 1] we formulate our steepest-descent algorithm by identifying

$$F^*(\vec{p}) = \begin{cases} 0 & \text{if } |\vec{p}(x)|_2 \le 1 \\ \infty & \text{else} \end{cases}, \; G(v) = \alpha_1\|v-g\|_{L^1(\Omega)} + \alpha_2\|v-g\|^2_{L^2(\Omega)}, \text{ and } \langle Kv,\vec{p}\rangle = \int_\Omega v \operatorname{div}\vec{p},$$

where we assume that each element in $\mathcal{P}$ has a weak derivative. If this is not the case, we require it for $\mathcal{U}$ and define $\langle Kv,\vec{p}\rangle = \int_\Omega \nabla v\vec{p}dx$.

We assume, that the datum $g \in L^2(\Omega)$ and the cost parameters α_i are given. Then in our algorithm we initialize $u_0 = \bar{u}_0 \in \mathcal{U}$, e.g., $u_0 \equiv 0$ or $u_0 = \mathbb{P}g$, where $\mathbb{P} : L^2(\Omega) \to \mathcal{U}$ denotes the L^2-Projection onto the discrete space $\mathcal{U}$, and $\vec{p}_0 \in \mathcal{P}$ to be the zero function. As parameters we need the step sizes $\sigma, \tau > 0$, the coefficient $\beta = \frac{\tau\alpha_2}{1+\tau\alpha_1}$, and the overrelaxation parameter $\theta \in [0,1]$.
Then our steepest-descent algorithm iterates starting with $k = 0$ as follows:

1. Set $\bar{p}$ according to the descent direction with step size σ as
$$\bar{p} = \vec{p}_k + \sigma \nabla \bar{u}_k. \tag{7}$$
This is to be understood in the weak sense using test-functions from $\mathcal{P}$. As the algorithm is derived from formulation (6), we need $|p|_2 \leq 1$, so we have to project p using
$$\vec{p}_{k+1} = (I + \sigma \partial F^*)^{-1}(\bar{p}) \quad \Leftrightarrow \quad \vec{p}_{k+1} = \frac{\bar{p}}{\max(|\bar{p}|_2, 1)}. \tag{8}$$
2. Update u_k using
$$u_{k+1} = (I + \tau \partial G)^{-1}(u_k + \operatorname{div} \vec{p}_{k+1}) \quad \Leftrightarrow \quad u_{k+1} = \begin{cases} z - \beta & \text{if } z - \beta \geq \mathbb{P}g \\ z + \beta & \text{if } z + \beta \leq \mathbb{P}g \\ \mathbb{P}g & \text{else}, \end{cases} \tag{9}$$
where
$$z = \frac{1}{1 + \tau \alpha_1} (u_k + \tau \operatorname{div} \vec{p}_{k+1} + \tau \alpha_2 \mathbb{P}g) \in \mathcal{U} \tag{10}$$
in the weak sense with test-space $\mathcal{U}$. In equation (9) and (10) instead of using the projection $\mathbb{P}g$ we could use the real datum g, which is equivalent to the above formulation, as the update is understood in the weak sense.
3. As proposed in [9] we overrelaxate to speed up the algorithm:
$$\bar{u}_{k+1} = u_{k+1} + \theta(u_{k+1} - u_k) \tag{11}$$
4. If the stopping criterion
$$\frac{\|u_{k+1} - u_k\|_{L^2}}{\|u_k\|_{L^2}} < TOL \tag{12}$$
holds, we terminate the algorithm, otherwise we set $k \to k + 1$ and repeat.

Convergence of this algorithm to the discrete minimum of problem (1) is guaranteed, if $\theta = 1$ and $\sigma \tau B^2 < 1$, by using the standard convergence analysis from [9], where $B := \|\nabla\|_{L^2}$ is the operator norm of the gradient operator. This norm is bounded for discrete functions and of order $O(1/h)$, which implies that using finer meshes will always result in more steps of the steepest-descent algorithm. For such algorithms this is a typical and well-known behavior. Even using an adaptive mesh is not beneficial to the stepsize, as B depends on the minimum cell-width.

3 Implementation Details

The algorithm is implemented in the DUNE-project non–smooth–minization . This project is compatible with the 2.4-release and available on the website `http://www.ians.uni-stuttgart.de/nmh/downloads`. It requires the modules DUNE-ACFEM and all modules that are required by it, available at `gitlab.dune-project.org`. The installation works the standard DUNE-way. (see Appendix A)

3.1 Image Read-In and Noise Simulation

To read-in images we use The CIMG Library - C++ Template Image Processing Toolkit [16]. The single header file library CIMG is an open source project to provide easy handling and processing of images. It is capable of reading-in standard image formats (e.g. .png, .tif , .jpg) and of adding certain types of noise. Each pixel is accessible by its coordinates utilizing the data type image , whose constructor gets the filename containing the image location. Moreover, image provides a method to add noise to the data with a given noise level for different noise-types. In particular, Gaussian noise with noise level (variance) γ and mean 0 and salt-and-pepper noise S with noise level $s \in [0,1]$, which corrupts an original image $\hat{g}$ by

$$S\hat{g}(x) = \begin{cases} \min \hat{g} & \text{with probability } s/2 \\ \max \hat{g} & \text{with probability } s/2 \\ \hat{g}(x) & \text{with probability } 1-s \end{cases},$$

are available.
We define two adapter classes to convert the data type image and one adapter class to convert a algebraic expression into a discrete function within the header file src/datafunction.hh .

- The LocalDataAdapter constructs an $\mathcal{L}^0(\mathcal{T}_h)$ function that approximates an inserted algebraic expression.

- The LocalImageAdapter converts an instance of class image into an $\mathcal{L}^0(\mathcal{T}_h)$ function using the GridFunctionAdapter provided by DUNE-ACFEM.

- The LocalNoiseAdapter enhances an instance of class image by noise. More precisely it adds two independent types of noise, where every noise implemented by CIMG can be applied. Noise type and noise level are controlled by a set of parameters defined in the parameter file. For instance, setting nsm.noiseType1 to 0 0 leads to Gaussian noise and setting it to 2 corresponds to salt-and-pepper noise. Setting any noise level parameter to 0 disables the respective noise. For other noise types and more information consult the documentation of CIMG . After applying the noise the corrupted instance of the class image is then converted into an $\mathcal{L}^0(\mathcal{T}_h)$ as in the LocalImageAdapter .

The artificial addition of noise (LocalNoiseAdapter) is only needed for demonstration purposes as in real-world application the image is already corrupted by noise due to certain physical processes.
The most important method of the adapter classes is the init () method, that sets the value on each mesh cell. For example, using the LocalDataAdapter the following code results in an approximation of a circle of radius 0.3 and center (0.5, 0.5) with value 1 inside and 0 outside, see Figure 1a.

```
// Implement needed interface method for the adapter class
   void init(const EntityType& entity)
   {
   //the geometrical expression to be represented as an image
    const auto center =  entity.geometry().center();
    //circle center 0.5,0.5 radius 0.3
    if( (center(0) - 0.5) * (center(0) -0.5 ) +(center(1) - 0.5) * (center(1) -0.5 ) <= 0.09)
         result_ = 1;
    else result_ = 0;
   }
```

The adapters convert the image into discrete functions on the unit square $[0,1] \times [0,1]$, as we can scale any rectangular image accordingly. The image resolution and aspect ratio relate to the level of initial refinement and to the number of cells in each direction. So n uniform refinements with $l \times k$ initial cells lead to an image resolution of size $l2^n \times k2^n$ square pixels, where every square is composed of two triangles and l/k corresponds to the aspect ratio. This is done in the gridfile data/unitsquare2d.dgf which describes the unit square using the DUNE DGF – Interval [6] nomenclature.

3.2 Implementation of the steepest-descent algorithm

The steepest-descent algorithm from section 2.2 is implemented in the file src/nsm.cc , where some of its subroutines, which depend on the continuity of the discrete spaces $\mathcal{U}$ and $\mathcal{P}$, are outsourced into the file src/phc.hh . The template class ProjectionHelperClass uses partial template specialization to correctly define the methods calculateZ() and entitywiseProjection() . The main part of the steepest-descent algorithm reads:

```
//now the main part of the algorithm
 for(int i = 0; i< maxIt; ++i)
 {
   //update p
   phc.entitywiseProjection(uBar, p);
   //calculateZ
   phc.calculateZ(p, uOld, z);
   //update u
   dofwiseProjection(beta, z, projG, u);
   //Output and stopping criterion
   ...
   //update uBar
   uBar.assign(u);
   uBar *= 1+theta;
   uBar.axpy((-1. * theta), uOld);
   //update uOld
   uOld.assign(u);
 }
```

The method phc.entitywiseProjection() realizes equations (7) and (8). Solving eq. (7) is relatively easy, as all steps are provided by Dune-ACFem, e.g. for continuous $\mathcal{U}$ and $\mathcal{P}$:

```
void entitywiseProjection ( const ForwardDiscreteFunctionType & uBar, AdjointDiscreteFunctionType
    & p )
 {
   //the gradient model defines the weak gradient using continuous test-functions
   //from the space of p
   auto gradU = gradientModel(uBar);
   auto Dbc0 = dirichletZeroModel(p);
   auto mass = massModel(p);
   //we solve p = sigma * nabla u + p
   //with dirichlet zero boundary
   auto model = mass - sigma_ * gradU - p + Dbc0 ;
   //Testspace = AdjointDiscreteFunctionSpaceType = space of p
   typedef EllipticFemScheme<AdjointDiscreteFunctionType , decltype(model)> SchemeType;
   //p is the returned solution
   SchemeType scheme(p, model);
   scheme.solve();
```

Models (e.g. massModel , gradientModel , ...) of Dune-ACFem are objects that define the integral kernels to be multiplied with test-functions and thus describe the equation to be solved. Models can be linearly combined over the field $\mathbb{R}$ and discrete functions can be added (or subtracted). FemSchemes (e.g. EllipticFemScheme , ...) are objects that represent the complete finite-element method. A FemScheme always solves Model $= 0$. In the above case, for given p and $\bar{u}$ the equation

$$0 = \langle \bar{p}, \phi \rangle - \sigma \langle \bar{u}, -\operatorname{div} \phi \rangle - \langle p, \phi \rangle \quad \forall \phi \in \mathcal{P}$$

with dirichlet-zero boundary conditions is solved with respect to $\bar{p}$. The solution is written into the variable p.

Projecting p to be feasible (eq. (8)) is not in the features of Dune-ACFem. So we have to use the features of Dune-Fem directly. We choose to do the projection entity-wise, i.e. we iterate over all entities of the grid. On each entity we iterate over the degrees of freedom and if $|p|_2 > 1$ holds, we restrict the corresponding value to length 1 in the same direction. If the polynomial order of $\mathcal{P}$ is less or equal than 1, this implies the necessary condition $\| |p|_2 \|_{L^\infty} \leq 1$.

The calculation of z (eq. (10)) translates very nicely into code. We use the L2Projection of Dune-ACFem and its ability to linearly combine discrete functions. If $\mathcal{P}$ is continuous, the weak form of eq. (10) is

$$\langle z_k, \psi\rangle = \frac{1}{1+\tau\alpha_1}\langle u_k + \tau \operatorname{div} p_{k+1} + \tau\alpha_2 g, \psi\rangle \quad \forall\psi\in\mathcal{U}$$

and translates into code in the following way:

```
void calculateZ (const AdjointDiscreteFunctionType &p, const ForwardDiscreteFunctionType & uOld,
    ForwardDiscreteFunctionType& z)
  {
    auto divP  = divergence(p);
    L2Projection(1./(1.+tau_*lambda_2_) * ( tau_ * divP + uOld + tau_ * lambda_2_ * projG_), z);
  }
```

Dune-ACFem currently does not implement DG-methods. Hence, if $\mathcal{P}$ is not continuous, we have to use the weak divergence to shift the derivative to the test-space $\mathcal{U}$, which means that $\mathcal{U}$ has to be continuous. If neither $\mathcal{U}$ nor $\mathcal{P}$ is continuous, the program fails. As above we use the basic Models of Dune-ACFem, in particular the massModel and the weakDivergenceModel to implement eq. (10) for discontinuous $\mathcal{P}$ as follows:

```
void calculateZ (const AdjointDiscreteFunctionType &p, const ForwardDiscreteFunctionType & uOld,
    ForwardDiscreteFunctionType& z)
  {
    //get the mass model of the Forward space
    auto U_Phi = massModel(projG_.space());
    //calculate z
    auto weakDiv_P = weakDivergenceModel(p);
    auto projModel = U_Phi  -1./(1.+tau_*lambda_2_) * ( tau_ * weakDiv_P + uOld + tau_ * lambda_2_
        * projG_);
    typedef EllipticFemScheme<ForwardDiscreteFunctionType, decltype(projModel)> SchemeType;
    SchemeType scheme(z, projModel);
    scheme.solve();
  }
```

Note that in contrast to the implementation of method entitywiseProjection() we construct the massModel from a discrete space instead of a discrete function. This is explicitly allowed by Dune-ACFem as the object simply has to provide the data type for the test-function space.

3.3 Adaptive Refinement

Adaptive refinement is initiated by a positive parameter nsm.localRefine . This parameter denotes the number of additional local refinements to be done in the initialization phase in addition to the uniform refinements. If it is set to ≤ 0, no local refinement will be performed. The grid is locally refined at discontinuities of the piecewise constant datum $g \in \mathcal{L}^0(\mathcal{T}_h)$ in the following way: Given an Entity E and its neighbour N, if

$$|g_{|N} - g_{|E}| > TOL \tag{13}$$

holds, then E is marked for refinement, where the value of TOL is given by the parameter nsm.adaptTolerance . We do not refine during the iterations of the steepest-descent algorithm, but keep the mesh static. So the refinement increases the resolution of discontinuities and hence drastically improves the projection $\mathbb{P}g$. This is implemented in the method adaptGrid() in the file nsm.cc . We use the grid manager Dune-ALUGrid [1], which is among other things capable of handling the needed conforming, adaptive triangular grids.

4 Numerical Experiments

To reproduce the data of the experiments of this section, consult Appendix A and follow the described steps.

4.1 Comparison of Discrete Spaces

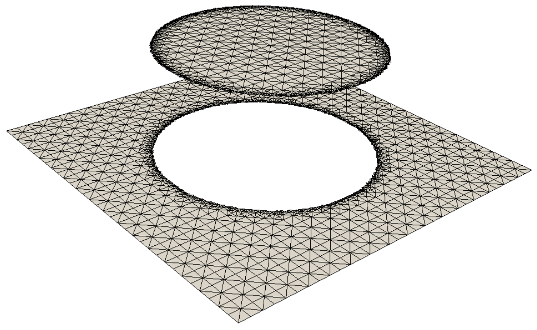
We investigate for different discrete spaces $\mathcal{U}$ and $\mathcal{P}$ the behaviour of our algorithm with respect to the non-smooth minimization problem (1) considering the following set-ups

$$\mathcal{U} = \mathcal{S}^1(\mathcal{T}_h) = \mathcal{P}, \quad \mathcal{U} = \mathcal{L}^0(\mathcal{T}_h), \mathcal{P} = \mathcal{S}^1(\mathcal{T}_h), \quad \mathcal{U} = \mathcal{S}^1(\mathcal{T}_h), \mathcal{P} = \mathcal{L}^0(\mathcal{T}_h).$$

For this set of experiments we choose a similar example as in [5], i.e., the observed data is 1 on a disk of radius 0.3 and 0 elsewhere. Hence, we choose the discrete datum g to be given as a piecewise constant approximation of the function

$$f(x) = \begin{cases} 1, & \text{if } |x - (0.5, 0.5)| \leq 0.3 \\ 0, & \text{else.} \end{cases}$$

The cost parameters are set to $\alpha_1 = 10$ and $\alpha_2 = 20$, the stopping tolerance to 10^{-5} and the adaptation tolerance to 0.1. We do 5 uniform refinements and 3 additional local refinements. As we use the unit square as our domain the operator norm of the gradient is $B = 2^8$. Consequently the step sizes are automatically set to $\tau = \sigma = 2^{-8}$. Then, in order to guarantee convergence, the overrelaxation parameter θ is chosen to be 1.


(a) $g = \mathbb{P}g$, as $\mathcal{U} = \mathcal{L}^0(\mathcal{T}_h)$

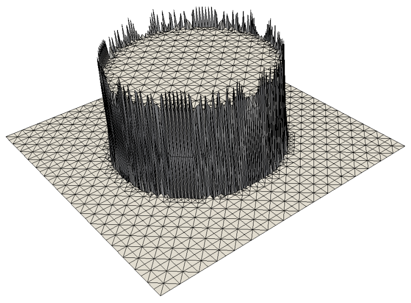
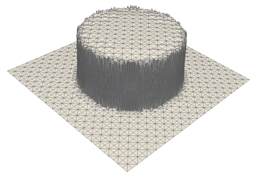
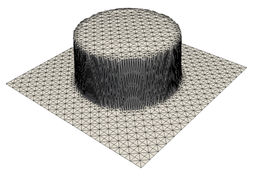

(b) $\mathbb{P}g$ for $\mathcal{U} = \mathcal{S}^1(\mathcal{T}_h)$

Figure 1: The Projection of the piecewise constant datum g onto the space $\mathcal{U}$.

By imposing all equations weakly the steepest-descent algorithm de facto uses the L^2-projection $\mathbb{P}g \in \mathcal{U}$ instead of the datum $g \in \mathcal{L}^0(\mathcal{T}_h)$. Figure 1 shows that in the case of $\mathcal{U} = \mathcal{S}^1(\mathcal{T}_h)$ by projecting a discontinuous function onto a continuous space we introduce an additional error in contrast to the case $\mathcal{U} = \mathcal{L}^0(\mathcal{T}_h)$. So intuitively it would be a good idea to choose $\mathcal{U} = \mathcal{L}^0(\mathcal{T}_h)$ to execute the algorithm, but we observe that the other setting performs better. This is due to the fact that in general, approximating functions of bounded variation is not possible with piecewise constant functions on a triangulation, since the length of discontinuities may not diminish over refinement [3]. The over- and undershoots of the continuous projected data do not pose a problem, because they are regularized over the course of the algorithm, as we can see in Figure 2.


(a) $\mathcal{U} = \mathcal{S}^1(\mathcal{T}_h) = \mathcal{P}$


(b) $\mathcal{U} = \mathcal{S}^1(\mathcal{T}_h), \mathcal{P} = \mathcal{L}^0$

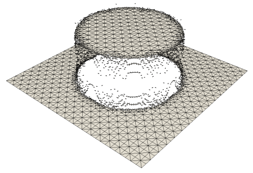
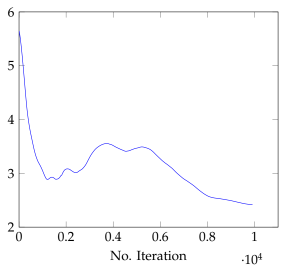
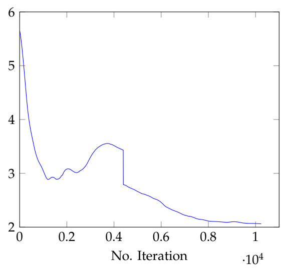

(c) $\mathcal{U} = \mathcal{L}^0(\mathcal{T}_h), \mathcal{P} = \mathcal{S}^1(\mathcal{T}_h)$

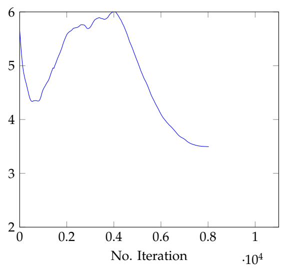
Figure 2: The discrete minima u^* of the functional $J_{10,20}$



(a) $\mathcal{U} = \mathcal{S}^1(\mathcal{T}_h) = \mathcal{P}$,
# Iterations: 9907,
$J_{\alpha_1,\alpha_2}(u^*) = 2.41776$



(b) $\mathcal{U} = \mathcal{S}^1(\mathcal{T}_h), \mathcal{P} = \mathcal{L}^0(\mathcal{T}_h)$,
# Iterations: 10234,
$J_{\alpha_1,\alpha_2}(u^*) = 2.06217$



(c) $\mathcal{U} = \mathcal{L}^0(\mathcal{T}_h), \mathcal{P} = \mathcal{S}^1(\mathcal{T}_h)$,
# Iterations: 8021,
$J_{\alpha_1,\alpha_2}(u^*) = 3.49582$

Figure 3: The value of the functional J_{α_1,α_2} over the course of the algorithm.

Figure 3 depicts the performance of the energy J_{α_1,α_2} of the algorithm during the iterations. We observe that the algorithm does not converge monotonically and also rather slowly, which is typical for steepest descent type methods. The combination $\mathcal{U} = \mathcal{S}^1(\mathcal{T}_h)$, $\mathcal{P} = \mathcal{L}^0(\mathcal{T}_h)$ yields the best result, while for $\mathcal{U} = \mathcal{L}^0(\mathcal{T}_h)$, $\mathcal{P} = \mathcal{S}^1(\mathcal{T}_h)$ the energy of the solution stays far from the minimum as already expected and also visible in Figure 2c, where one may still observe over- and undershoots.

4.2 Image Denoising

Here we demonstrate the denoising capability of the algorithm. Motivated by the above experiments we choose the spaces to be $\mathcal{U} = \mathcal{S}^1(\mathcal{T}_h)$, $\mathcal{P} = \mathcal{L}^0(\mathcal{T}_h)$ again. The considered image is first corrupted by Gaussian white noise with variance 0.1 and then salt-and-pepper noise with $s = 0.1$ is added. The obtained image is shown in Figure 4a. In order to reconstruct the image we choose $\alpha_1 = 250$, $\alpha_2 = 150$ in (1), and perform our steepest-descent algorithm with the tolerance set to 10^{-4}, $\theta = 1$ and $\sigma = \tau = 2^{-8}$. To resolve the 256×256 pixel-sized picture we apply 8 uniform refinements and no additional local refinement. In Figure 4b we depict the output of our algorithm. The result is reasonably smoothed due to the total variation regularization, while discontinuities are still preserved.

4.3 Strong Scaling

We do a strong scaling experiment on the same datum as in Section 4.1. The computation is executed on a 32 core shared memory system. The grid manager Dune-ALUGrid balances the computational load on the initial grid by partitioning it onto the different processors. So we cannot expect the algorithm to scale if the initial mesh is to coarse. E.g. if there are only two initial elements, only two processors can get partitions that are non-empty. Consequently we discretize

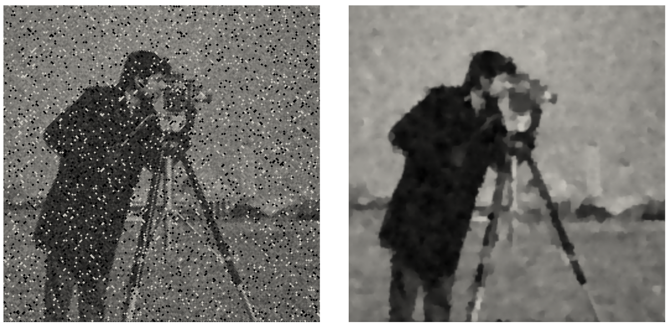


Figure 4: A corrupted image before and after the algorithm

the unit square by 8×8 elements, see `data/finecube_2d.dgf`. This results in a different operator norm on the initial grid, so we have to set the corresponding parameter `nsm.constantL` to 0.125 for the stepsizes τ, σ to be computed correctly. As the grid is already fine, we do not need as many uniform refinements to reach a good resolution, so we do 3 uniform and another 3 local refinements. The tolerance is set to 10^{-4} and $\alpha_1 = \alpha_2 = 150$ to reach a short runtime.

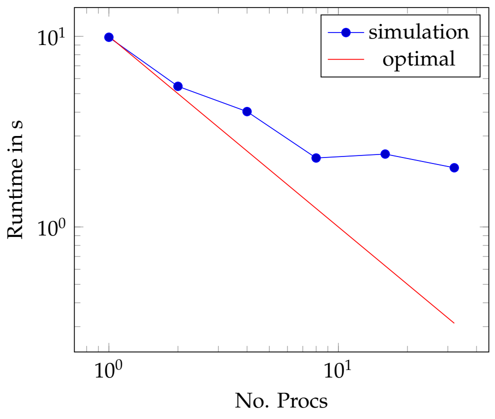


Figure 5: Strong scaling of the algorithm.

Figure 5 indicates that the strong scaling at least up to 8 cores is quite good and afterwards it stagnates, which is probably due to the small grid and not enough workload. We would have to increase the number of initial cells to improve the scaling further. The important part about this result is, that the parallelism is done inside Dune-ACFem and we spent almost no effort to parallelize the code. The only line of code needed initializes `MPI`.

5 Summary and Outlook

We have shown that the discrete minimizer converges to the continuous one. The steepest-descent algorithm used here is implemented very conveniently within Dune-ACFem. The flexibility of Dune-Fem allows us to easily exchange discrete spaces and easily set parameters. Also the algorithm is now intrinsically parallel by domain decomposition with a decent strong scaling. Future research includes implementing a semi-smooth newton method in DUNE-ACFEM to increase convergence speed even for highly adaptive grids.

Acknowledgements

The author Martin Alkämper would like to thank the German Research Foundation (DFG) for financial support of the project within the Cluster of Excellence in Simulation Technology (EXC 310/2) at the University of Stuttgart.

Appendix A - Installation Instructions

The program is designed to work in a Linux-environment. It may as well work on a unix machine. To install the program, first download the DUNE modules, DUNE-COMMON, DUNE-GEOMETRY, DUNE-GRID, DUNE-ISTL, DUNE-LOCALFUNCTIONS, DUNE-FEM, DUNE-ALUGRID and DUNE-ACFEM from `gitlab.dune-project.org` and `non−smooth−minization` from `http://www.ians.uni-stuttgart.de/nmh/downloads`. For the DUNE- modules checkout the branch `releases/2.4` . Put all the projects in a directory as direct subfolders. Then run

```
./dune=common/bin/dunecontrol −−opts=config.opts −−module=non−smooth−minization all
```

An example config file `example.opts` is provided in the main directory of the DUNE project `non−smooth−minization` .
Now DUNE is installed and the executable `nsm` from the subdirectory `src` should have been built within the `cmake` build directory. The scripts `example1.sh` to `example3.sh` produce the results presented in this paper. They require a `python` interpreter and `example3.sh` requires `mpi` . The `cmake` build directory is assumed to be named `build−cmake` . The scripts are simple so changes should be easy. The scripts basically consist of changing directories, compiling the executable `nsm` , executing it with the right set of parameters and parsing the output into suitable directories. There are two types of output. Console output from `std::out` is parsed into a file `output.graph` in a subdirectory of `src` , which describes the value of the functional and the value of all its parts with respect to both g and $\mathbb{P}g$ and a file `output.graph` , where the applied parameters and other output can be looked up. Console output from `std::err` is displayed on the console. The other type of output is in the `cmake` build directory located within the directory `output` in suitable subdirectories. These contain a set of `.vtu/.vtk` files (readable with `Paraview`), that contain the datum g (called "image"), the projection $\mathbb{P}g$ (called "projection of g"), the solution (called "u0") and the adjoint variable (componentwise "p0", "p1"). It will take some time to run the scripts. To get faster results, simply edit the `LOCREF` or `INITREF` variable of the bash scripts or lower the tolerance parameter `nsm.tolerance` .

Appendix B - Parameters

There are two types of parameters: Runtime parameters and Compile-time Parameters.

- Runtime parameters can be overloaded on the command line
  `./nsm nsm.parameter:value ...` .
  The default values are set inside the parameter file located at `data/parameter` . This file is copied into the `cmake` build directory at compile time. So parameters changed in the build directory will be overwritten when recompiling. Parameters with prefix fem, or istl belong to DUNE-FEM and DUNE-ISTL respectively, and are explained in their documentation. The specific parameters of this algorithm are prefixed nsm. The extensive list of runtime

parameters reads:

nsm.theta	overrelaxation parameter θ
nsm.maxIt	Maximum number of iterations of the algorithm
nsm.tolerance	The algorithm breaks if the tolerance is reached
nsm.outputStep	Output every nth step
nsm.constantL	Norm of the gradient on the initial mesh
nsm.tau	Stepsize τ
nsm.sigma	Stepsize σ
nsm.lambda_1	L^1-data term cost coefficient α_1
nsm.lambda_2	L^2-data term cost coefficient α_2
nsm.image	Filename of the image to be read-in
nsm.initialRefinements	Number of initial uniform refinements
nsm.localRefine	Number of initial adaptive refinements
nsm.adaptTolerance	Adaptive tolerance
nsm.noiseType1	The CIMG type of noise to be applied first
nsm.noiseLevel1	The CIMG noise level of the first noise
nsm.noiseType2	The CIMG type of noise to be applied second
nsm.noiseLevel2	The CIMG noise level of the second noise

- Compile-time parameters have to be declared when compiling. The location is in the file `CMakelists.txt`. As they are `cmake`-cache variables, they can be redefined in the usual way. (see e.g. `./example1.sh`) They determine what kind of problem to treat and which discrete spaces are to be used.

NSM_SET_STEPSIZE	If true, tau and sigma are set manually
NSM_USE_IMAGE	If true, image is used instead of geometric expression
NSM_USE_NOISE	If true, noised imaged is used, requires NSM_USE_IMAGE
NSM_U_POLORDER	The polynomial order of u (default 1)
NSM_P_POLORDER	The polynomial order of p (default 1)
NSM_U_DISCONT	If defined, $\mathcal{U}$ is discontinuous.
NSM_P_DISCONT	If defined, $\mathcal{P}$ is discontinuous.

References

[1] Martin Alkämper, Andreas Dedner, Robert Klöfkorn, and Martin Nolte. The dune-alugrid module. *Archive of Numerical Software*, 4(1):1–28, 2016.

[2] Luigi Ambrosio, Nicola Fusco, and Diego Pallara. *Functions of Bounded Variation and Free Discontinuity Problems*. Oxford Mathematical Monographs. The Clarendon Press, Oxford University Press, New York, 2000.

[3] Sören Bartels. Total variation minimization with finite elements: convergence and iterative solution. *SIAM Journal on Numerical Analysis*, 50(3):1162–1180, 2012.

[4] Sören Bartels. Broken sobolev space iteration for total variation regularized minimization problems. *IMA Journal of Numerical Analysis*, page drv023, 2015.

[5] Sören Bartels. Error control and adaptivity for a variational model problem defined on functions of bounded variation. *Mathematics of Computation*, 84(293):1217–1240, 2015.

[6] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, and O. Sander. A generic grid interface for parallel and adaptive scientific computing. II. Implementation and tests in DUNE. *Computing*, 82(2-3):121–138, 2008.

[7] P. Bastian, M. Blatt, A. Dedner, Ch. Engwer, R. Klöfkorn, S.P. Kuttanikkad, M. Ohlberger, and O. Sander. The Distributed and Unified Numerics Environment (DUNE). In *Proc. of the 19th Symposium on Simulation Technique in Hannover, Sep. 12 - 14*, 2006.

[8] Susanne C. Brenner and Ridgway Scott. *The Mathematical Theory of Finite Element Methods*, volume 15. Springer Science & Business Media, 2008.

[9] Antonin Chambolle and Thomas Pock. A first-order primal-dual algorithm for convex problems with applications to imaging. *Journal of Mathematical Imaging and Vision*, 40(1):120–145, 2011.

[10] A. Dedner, R. Klöfkorn, M. Nolte, and M. Ohlberger. A Generic Interface for Parallel and Adaptive Scientific Computing: Abstraction Principles and the DUNE-FEM Module. *Computing*, 90(3–4):165–196, 2010.

[11] Enrico Giusti. *Minimal Surfaces and Functions of Bounded Variation*, volume 80 of *Monographs in Mathematics*. Birkhäuser Verlag, Basel, 1984.

[12] Claus-Justus Heine. Dune-acfem documentation. *http://www.ians.uni-stuttgart.de/nmh/stage/documentation/software/dune-acfem/doxygen/*, 2014.

[13] Michael Hintermüller and Andreas Langer. Subspace correction methods for a class of nonsmooth and nonadditive convex variational problems with mixed L^1/L^2 data-fidelity in image processing. *SIAM J. Imaging Sci.*, 6(4):2134–2173, 2013.

[14] Michael Hintermüller and Monserrat Rincon-Camacho. An adaptive finite element method in L^2-TV-based image denoising. *Inverse Problems and Imaging*, 8(3):685–711, 2014.

[15] Andreas Langer. Automated parameter selection in the L^1-L^2-TV model for removing Gaussian plus impulse noise. *preprint*, 2016.

[16] David Tschumperlé. The cimg library - c++ template image processing toolkit. *http://www.cimg.eu/reference/*.