GASPACHO: a Generic Automatic Solver using Proximal Algorithms for Convex Huge Optimization problems

Bart Goossens, Hiêp Luong and Wilfried Philips

Ghent University, Dept. of Telecommunications and Information Processing, imec-IPI-UGent, Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium

Keywords: convex optimization, automatic differentiation, inverse problems, proximal algorithms.

ABSTRACT

Many inverse problems (e.g., demosaicking, deblurring, denoising, image fusion, HDR synthesis) share various similarities: degradation operators are often modeled by a specific data fitting function while image prior knowledge (e.g., sparsity) is incorporated by additional regularization terms. In this paper, we investigate automatic algorithmic techniques for evaluating proximal operators. These algorithmic techniques also enable efficient calculation of adjoints from linear operators in a general matrix-free setting. In particular, we study the simultaneous-direction method of multipliers (SDMM) and the parallel proximal algorithm (PPXA) solvers and show that the automatically derived implementations are well suited for both single-GPU and multi-GPU processing. We demonstrate this approach for an Electron Microscopy (EM) deconvolution problem.

1. INTRODUCTION

In the past decade, convex optimization frameworks have become popular as a tool for solving various practical inverse problems (such as demosaicking, deblurring etc.). The success of these frameworks can be attributed to various factors, such as 1) the flexibility in modeling the degradation process and specifying image prior knowledge, 2) guarantees about the existence of a unique local minimum of the objective function and the convergence of the algorithms, 3) high quality restoration results and 4) increased computational efficiency of CPUs and GPUs. When applying the optimization tools in practical applications, many researchers can focus on the selection of the appropriate data fitting function (modeling the image degradation process) and regularization function(s) (incorporating domain-specific knowledge on the "ideal" or "desired" solution). Figure 1 displays such an image restoration framework: a degraded image is updated iteratively, using an iterative solver. This solver iteratively minimizes a cost function that weighs data fitting (related to the degradation model) and regularization functions (related to the image prior model).

However, there are also a few important issues with this approach: first, when solving an inverse problem, it is unsure in advance which combination of image priors and optimization methods will give optimal results (e.g., visually or in terms of an image quality metric, such as MSE, SSIM, etc.). Often, to achieve superior results, different regularization terms, such as wavelets and total variation, are used in combination with application-specific regularization (for example, range constraints). Some optimization algorithms may offer faster convergence for certain tasks than others. In general, it is difficult to predict in advance which combination of data fidelity function, regularization terms and optimization algorithm will give the best result for a certain application, and the most common solution is to try each combination one by one, which is - implementationwise - a time consuming task. In practice, many researchers therefore stick to just a few optimization algorithms (e.g., ADMM,[1] C-SALSA[2]) and adapt their algorithm to each problem at hand. However, this approach may be error-prone, because an incorrect implementation of a convex optimization algorithm may converge to a

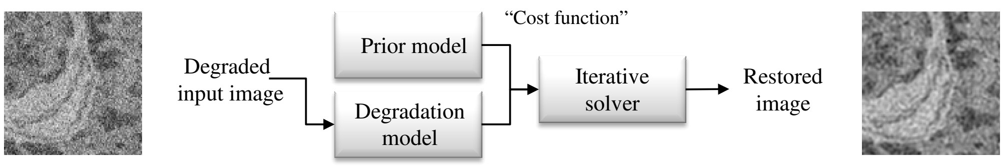


Figure 1. Convex optimization-based image restoration framework: degraded image, degradation and prior model.

suboptimal solution. Often it is difficult to detect these errors and they are only found when testing an large numbers of images.

Moreover, it becomes challenging to take advantage of the vast computational speedup offered by multi-core CPUs or graphical processing units (GPUs) for these tasks. Despite their high computational cost compared to suboptimal local filtering methods, convex solvers for inverse problems are embarrassingly parallel,[3] and hence they are well suited for implementation on a GPU. Also, in recent years in mobile device GPUs, the floating point performance per Watt has significantly increased, which makes it attractive to run convex optimization algorithms for image reconstruction/restoration directly on a mobile phone or tablet. But then, one faces issues with cross-platform development (requiring experienced professional programmers) and research prototypes rarely reach this stage.

To simplify the development of inverse methods for practical applications and on various platforms, there has been a recent interest in building optimized domain-specific languages (DSLs) and libraries. These languages have emerged from low- and mid-range convex solvers (i.e., that can deal with tens to thousands of unknown variables) noting that for images more advanced techniques are required, because the number of unknowns can be up to millions or even billions.

One example is ProxImal,[4] which is a domain-specific language based on the Pock-Chambolle and ADMM algorithms. Starting from a description of the cost function and regularization terms, ProxImal synthesizes an optimal large-scale convex solver. Under the hood, ProxImal uses the Halide[5] domain-specific language which has been designed for highly parallel computational imaging tasks. Some of the core ideas behind ProxImal are 1) to adopt a *matrix-free* approach (which allows easily synthesizing algorithms for huge problems involving millions of unknown variables) and 2) to exploit the structure in the underlying problem in order to synthesize specialized algorithms. One of the difficulties of ProxImal is that incorporating new problem formulations require implementation of new proxable functions and solvers; while it is desirable that new problem formulations can be defined within the DSL itself. Also, ProxImal relies on user input for proximal operators for which no direct implementation exists (in some cases, the user can then implement an iterative gradient descent solver). Another interesting missing feature for ProxImal is the ability to perform parameter estimation (e.g., estimation of a regularization parameter).

In this paper, we present GASPACHO, a general convex solver framework for inverse problems: 1) the ability to control the solution methods from within the DSL, 2) the possibility to extend the DSL from within the DSL itself, 3) support for parameter optimization/tuning. For this purpose, we also incorporate algorithms for algorithmic differentiation (AD) and algorithmic adjoint calculation (AAC): AD[6] is an efficient method to numerically evaluate the derivative or gradient of a linear or non-linear function specified by a computer program. AAC transforms a linear function into a function calculating its adjoint. AAC is also indispensable in solution methods for inverse problem that heavily rely on adjoint operators. AD and AAC relieve the user from the error-prone manual writing of respectively the derivatives and the adjoints of certain operators. Both AD and AAC fit within the framework of *matrix-free* computations: rather than working with matrix representations, the algorithms focus on the structure of the computations. Traditional AD algorithms are sequential in nature, therefore we investigate whether the AD algorithm synthesis can be performed such to enable generation of parallel code. In particular, we find that when the cost function can be calculated in parallel, then with some adjustments, the gradient of the cost function can also be calculated in parallel.

From a high-level programming point of view, the matrix-free computation approach also enables extracting the structure inherent in the operators more easily: for example, the DSL compiler may recognize pointwise operators, diagonal operators, etc. Because the analyzing characteristics of the DSL compiler can be limited, for this purpose we also introduce an *expression rewriting system* (ERS) to our DSL. The ERS allows the framework to transform certain expressions according to rules specified within the DSL. For example, the user can specify how to implement $\text{argmax}_x f(x)$ for some specific choices of f using a specialized solver, with a generic implementation as a fallback approach. This technique effectively allows the user to extend the DSL with custom solvers.

A pattern matching system will then select the most suited implementation to compute a (local) minimum of $f(x)$ with respect to x, given some specified constraints (for example, with respect to the structure of the involved matrices). This approach enables declarative programming: it is possible to describe the structure of the problem and to specify the desired solution, without the need for digging deep into the solution method. A generic solver may then rely on the gradient (e.g., using gradient descent, not-linear conjugate gradients, stochastic gradient descent), where the gradient is may be derived automatically, determined using the ERS, or a combination of both. Interesting is then the interaction between the DSL compiler and the user: the DSL compiler may give feedback for certain operations, and the user may control the algorithm synthesis by adding new or changing existing rewriting rules.

In this paper, we illustrate the main mechanisms of our approach and we therefore focus on convex ℓ_2 optimization problems with ℓ_1 regularization, of the following form:[*]

$$\min_{\mathbf{x}} \|\mathbf{y} - \mathbf{W}(\mathbf{x})\|_2^2 + \sum_{i=1}^{I} \lambda_i \left|\mathbf{S}_i \mathbf{x}\right|_1 \qquad (1)$$

where $\mathbf{W}$ is the degradation operator and where $\mathbf{S}_i$, $i = 1, ..., I$ are sparsity transforms (e.g., wavelets, total variation, shearlets). The class of simultaneous-direction method of multipliers (SDMM) and parallel proximal algorithm (PPXA) are convex solvers of interest for these problem. We describe how these solvers can be defined using the DSL and we discuss some of the algorithms that the DSL compiler uses internally. Both algorithms are particularly interesting, because their automatically generated implementations are well suited for multi-GPU processing: the optimization steps corresponding to each regularization term can be evaluated in parallel. Finally, as an application, we demonstrate this approach for an Electron Microscopy (EM) deconvolution problem, where the captured EM images are blurred, have a low signal-to-noise ratio and are corrupted with correlated noise.

For the underlying implementation of GASPACHO, we rely on Quasar,[7] which is an easy to use and MATLAB-like programming environment with compiler, runtime and tools for heterogeneous systems consisting of CPUs and GPUs.[†] The merit of Quasar for this work is that all algorithms are automatically parallelized and can be executed on a GPU, which gives a tremendous speedup compared to execution on a multicore CPU. Moreover, several internal code transformations from GASPACHO were borrowed from the Quasar compiler.

The remainder of this paper is as follows: in Section 2, we present the problem formulation in a DSL. We explain how the DSL helps modeling the problem and focusing on the algorithmic aspects rather then implementation details. We discuss two important aspects of the DSL: 1) expression rewriting through reductions, 2) modularity of the solution approach. In Section 3, we explain the use of algorithmic differentiation and adjoint calculation within the current framework. In Section 4 we present experimental results for our approach applied to EM. Finally, a conclusion follows in Section 5.

2. PROBLEM FORMULATION IN THE DOMAIN-SPECIFIC LANGUAGE

To illustrate the difference between the merits of implementation in a DSL such as GASPACHO compared to a "traditional" programming language, we refer to Figure 2. In the traditional implementation, the programmer combines routines from various libraries (e.g., numerically libraries), some of which are accelerated for CPU and/or GPU. Some additional routines are added that are not present in the libraries. The algorithm implementation then combines both. When multiple platforms are targeted (for example, one platform with a CPU, another platform with a CPU and a GPU), the algorithm need to switch between CPU and GPU implementations of the numerical routines. Although some numerical libraries do some effort to facilitate this, most of complexity (especially for the additional routines) is in the hand of the programmer, who often needs extensive knowledge of the platforms and needs to focus on low-level implementation details.

The DSL-based implementation (Figure 2(b)) permits implementation by an application specialist and the DSL compiler takes care of the multi-platform issues and the low-level implementation details. In general, because it is tailored to the application at hand, the description in the DSL is much easier to read and understand than an optimized implementation in a "traditional" program. Correspondingly, the DSL implementation is generally much more dense and contains fewer lines of code. Two challenges of DSL design are 1) ensuring the DSL is general enough that it does not restrict the user in its potential (for example, a DSL may allow the user to only solve linear problems; this may be too restricting) and 2) obtaining a performance efficiency of the generated code that is on a par with a specialist's implementation in a "traditional" programming language approach.

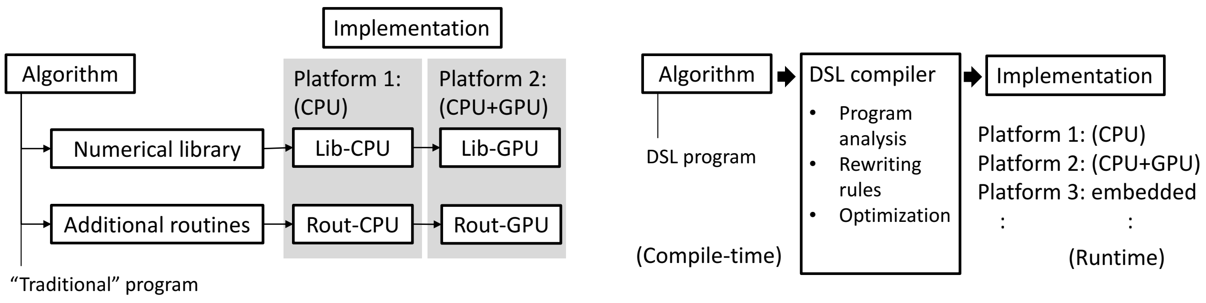
In this section, we focus on the design of GASPACHO for solving convex optimization problems and we illustrate how the above main challenges can be overcome.

2.1 Example problem specification

We illustrate our approach for an interpolation and deconvolution example. Assuming degraded data that is first convolved by a *known* filter with mask w, downsampled by a factor two and subsequently corrupted with Gaussian noise (see Figure

[*]Note that the presented framework is general enough to treat a much broader scope of problems, however this convex problem is of particular interest in image reconstruction/restoration.

[†]The Quasar software is available online for download on `http://gepura.io/quasar`



(a) (b)

Figure 2. "Traditional" implementation versus DSL implementation.

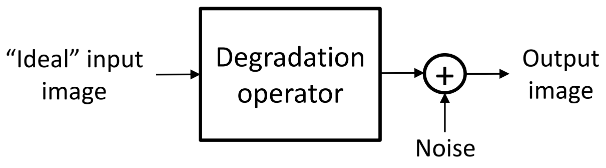


Figure 3. Degradation model: an ideal (noisy and blur-free) image is degraded using a degradation operator. Next, additive Gaussian noise is added, resulting in the observed output image.

3). In the presence of noise n, the relationship between the observed image y and the ideal undegraded image x is given by:

$$y(k,l) = (x \star w)(2k,2l) + n(k,l) \tag{2}$$

where $(k,l) \in \{0,\dots,M-1\} \times \{0,\dots,N-1\}$ are spatial coordinates. $n(k,l)$ is stationary additive Gaussian noise with mean zero and variance σ_n^2. [‡] To continue, we will switch to a vector notation rather than working with the functions $y(k,l)$ and $x(k,l)$. Therefore, let $\mathbf{x}$ and $\mathbf{y}$ be column-stacked versions of respectively $x(k,l)$ and $y(k,l)$, i.e., their components are given by $(\mathbf{x})_{2Nk'+l'} = x(k',l')$ for $k' = 0,\dots,2M-1$, $l' = 0,\dots,2N-1$ and $(\mathbf{y})_{Nk+l} = y(k,l)$ for $k = 0,\dots,M-1$, $l = 0,\dots,N-1$. To estimate the ideal undegraded image, we will use the following cost function with a variable number of regularization terms:

$$\min_{\mathbf{x}\in\mathbb{R}^{4MN}} \|\mathbf{y} - \mathbf{W}\mathbf{x}\|_2^2 + \sum_{i=0}^{I} \lambda_i |\mathbf{S}_i \mathbf{x}|_1 \tag{3}$$

where $\mathbf{W}$ denotes the degradation operator whose components are readily obtained from (2):

$$(\mathbf{W})_{Nk+l,2Nm+n} = w(2k-m,2l-n) \quad \text{with} \quad k = 0,...,N-1,\, l = 0,...,N-1. \tag{4}$$

The matrix $\mathbf{W}$ has dimensions $MN \times 4MN$, hence its total number of elements is $4M^2N^2$. Obviously it is desirable *not* to work with the matrix representation of $\mathbf{W}$ but instead directly exploit the structure given by (4). For this relatively simple example, this is evident, although we will see in Subsection 2.2 how this can be generalized to linear operators with arbitrary structure. A minimizer of (3) can be obtained using the simultaneous-direction method of multipliers (SDMM) algorithm and the parallel proximal algorithm (PPXA). For reference, the resulting algorithms are given in Algorithms 1 and 2. Both algorithms rely on the adjoint operators $\mathbf{W}^T$, $\mathbf{S}_1^T$ and $\mathbf{S}_2^T$, but also on the inverse $\left(\mathbf{W}^T\mathbf{W} + \mathbf{S}_1^T\mathbf{S}_1 + \mathbf{S}_2^T\mathbf{S}_2\right)^{-1}$. The adjoint degradation operator is easily expressed through its matrix form:

$$\left(\mathbf{W}^T\right)_{2Nm+n,Nk+l} = w(2k-m,2l-n) \quad \text{with} \quad k = 0,...,M-1,\, l = 0,...,N-1. \tag{5}$$

Again, we rely here on a matrix representation, which is not desirable. Instead, we like to obtain directly an algorithmic procedure for the operator applied to a vector: $\mathbf{W}^T\mathbf{x}$. This will be achieved using *algorithmic adjoint calculation* (see Section 3).

[‡]For simplicity, we assume here absense of spatial noise correlations, although when necessary they can easily be included in the model.

In the following, we will distinguish between *compile-time* which refers to the DSL compiler processing the DSL programming code written by the user and *runtime* which refers to the generated binary code being executed (see Figure 2(b)). We explain how the above formulation can be converted to a DSL program.

2.2 DSL specification

In GASPACHO, the user directly specifies the degradation operator using an imperative program. An example implementation of the degradation operator from (4) is given in Listing 2.2. Manually translating (5) to a similar implementation is straightforward, but for more complex linear degradation operators, it may be quite time-consuming and error-prone. Instead, the adjoint of an operator `op` can be derived automatically, using the `$adjoint(op)` meta-function.[§]

```
function [y : cube] = degradation_operator(x : cube, w : mat, center : vec2)
    y = zeros(size(x).*[2,2,1])
    for m=0..size(x,0)-1            % Image rows
        for n=0..size(x,1)-1        % Image columns
            for c=0..2              % Color components
                sum = 0.0
                for k=0..size(w,0)-1        % Filter kernel rows
                    for l=0..size(w,1)-1    % Filter kernel columns
                        sum += x[2*m+k-center[0],2*n+l-center[1],c] * w[k,l]
                    endfor
                endfor
                y[m,n,c] = sum
            endfor
        endfor
    endfor
endfunction
% Automatically calculate the adjoint of the degradation operator
adjoint_degradation_operator = $unapply($adjoint(degradation_operator(x,w,center),x),x)
```

Listing 1. Example blurring and subsampling filter, used in superresolution, specified in the Quasar language.[7]

To see how the degradation operator is used, consider first the special case of (3) for which the regularization factors are zero ($\lambda_1 = \lambda_2 = 0$). The solution that minimizes $\|\mathbf{x}\|_2$ during the optimization, is the pseudo-inverse solution:

$$\hat{\mathbf{x}} = (\mathbf{W}\mathbf{W}^T)^{-1}\mathbf{W}^T\mathbf{y}. \tag{6}$$

The above matrix inversion can be efficiently implemented using the conjugate gradients (CG) method given in Listing 2.2. CG relies on the implementation of the linear functions $\mathbf{Wx}$ and $\mathbf{W}^T\mathbf{x}$. Because the AAC calculates $\mathbf{W}^T\mathbf{x}$ automatically, it suffices to specify an algorithm to calculate $\mathbf{Wx}$ and the solution framework can calculate the pseudo-inverse (or an approximation to it) for any linear system. This allows us to define `conjugate_gradients` as a (higher-order) function of $\mathbf{W}$.

```
function x = conjugate_gradients(y, W, num_iterations=10, tol=1e-15)
    W_H = $unapply($adjoint(W(y), y), y)      % Algoritmic adjoint calculation
    x = 0; r = W_H(y); p = r; new_err = dotprod(r, r)

    for it=1..num_iterations
        err=new_err; A_p = W_H(W(p))
        alpha = err / dotprod(p, A_p)
        x = x + alpha * p
        r = r - alpha * A_p
        new_err = dotprod(r, r)
```

[§]In GASPACHO, meta-functions are functions that are evaluated at compile-time, by a function that takes a function implementation as input. The resulting function is therefore called a code transformation. For clarity, all meta-functions have the $-prefix.

```
        if (new_err/numel(x) < tol)
            break
        endif
        beta = new_err / err
        p = r + beta * p
    endfor
endfunction
```

Listing 2. Generic conjugate gradients algorithm in Quasar, calculating the pseudo-inverse solution from (6).

With this approach, complex optimization algorithms dealing with large-dimensional data vectors **x** and **y** can straightforwardly be implemented in a fairly generic way. Due to the modularity of the functions, a PPXA or SDMM algorithm can internally use a conjugate gradient solver to solve a subproblem.

2.3 Reductions

To specify which optimization algorithm tends to be useful for a given task, and to specify under which circumstances a given solver can be used, reductions (also known as rewriting rules[8]) are very useful.[¶] In GASPACHO, reductions are specified as follows:

```
reduction ($a_1$,$a_2$,...,$a_N$) → pattern($a_1$,$a_2$,...,$a_N$) = replacement($a_1$,$a_2$,...,$a_N$) where conditions(
    $a_1$,$a_2$,...,$a_N$)
```

A reduction consists of bound variables (`arg1`, `arg2`, ..., `argN`), a pattern expression describing the expression to be matched, a replacement expression and a condition expression. The DSL compiler matches every expression in an input program against all loaded reductions (this is done very efficiently using a tree-based algorithm that follows the abstract syntax tree) and evaluates the conditions corresponding to the obtained matches. If the condition cannot be evaluated statically at compile-time (which means that the result may depend on a parameter that is only known at runtime), the condition will be evaluated at runtime. Then, one of the patterns for which the conditions evaluate to true is selected as the final match, using a reduction resolution strategy (e.g., based on priority rules). Reductions have several applications:

- Reductions are used to optimize matrix/vector expressions with specialized routines, such as Basic Linear Algebra Subroutines (BLAS).[11]
- Reductions are also used to add domain-specific knowledge to the DSL that can be exploited by the compiler. For example, one may point out that an inverse discrete Fourier transform followed to the forward discrete Fourier transform amounts to an identity operation: `ifft(fft(x)) = x`. Another example: in combination with a function that determines positive-definiteness of a matrix, a reduction can be defined to calculate the singular value decomposition of a matrix using a symmetric eigenvalue decomposition.

The set of reductions that are of interest are highly domain and/or application-specific. Therefore it makes sense to leave it to the user to add reductions. Note however, that the DSL gives no guarantees that user-specified reductions are correct. As a simplistic example, it is possible to specify that $1+1=3$, obviously breaking code that evaluates at compile-time to $1+1$. One interesting line of research we are pursuing is to couple reductions to automatic proving systems; this would give feedback to the user in case reductions are known a priori to be incorrect.

Continuing our declarative programming approach, the conjugate gradient solver can be defined as a reduction:

```
reduction (x : mat, y : mat, W : [mat -> mat] : [mat -> mat]) →
    argmin(sum((W(x) - y).^2), x) = conjugate_gradients(y, W) where $islinear(W)
```

[¶]Rewriting rules are also used in various computer algebra systems, e.g., Mathematica,[9] Maple,[10] ...

where `.^` applies the power to every component of the vector and where `sum(x)` sums the components of the vector `x` (idem for `.*` in the following, which performs pointwise multiplication). `W(x)` is a function that maps a matrix onto a matrix (indicated by the type `[mat -> mat]`). The reduction may only be applied when the DSL compiler can ensure that `W` is a linear function (see Subsection 2.4). Using the above reduction, we can solve the degradation problem $\hat{\mathbf{x}} = \min_{\mathbf{x}\in\mathbb{R}^{4MN}} \|\mathbf{y}-\mathbf{W}\mathbf{x}\|_2^2$ as follows:

```
x_hat = argmin(sum((W(x) - y).^2), x)
```

The DSL compiler will replace this expression by `conjugate_gradients(y, W)`, which seems at first sight only to be a convenience feature. However, the approach becomes more powerful when additional reductions are applied. Similarly, the SDMM and PPXA algorithms from Algorithm 1 and Algorithm 2 can be added to the DSL:

```
reduction (x, y, W, ...λ, ...S) →
    argmin(sum((y - W(x)).^2) + sum([λ.*sum(abs(S(x)))]), x) = SDMM_solver(x0, y, W, S, λ)
reduction (x, y, W, ...λ, ...S) →
    argmin(sum((y - W(x)).^2) + sum([λ.*sum(abs(S(x)))]), x) = PPXA_solver(x0, y, W, S, λ)
```

Here we use a special variadic argument notation (`...λ` and `...S`), which allows expressions with a variable number of summation terms to be matched. In fact, the pattern `sum([λ.*sum(abs(S(x)))])` will match expressions as `λ₁.*abs(S_TV(x)) + λ₂.*abs(S_wavelet(x))`, the bound variable `S` is then an ordered set of functions. Here, `S_TV(x)` and `S_wavelet(x)` respectively implement the forward Total Variation and Shearlet transforms. Variadic arguments require specialized handling inside the DSL compiler, but greatly enhances the generality of the solvers. In this case, the solvers can be implemented using a variable number of regularization terms. Summarizing, the reduction system allows us to easily solve optimization problems like:

```
 x = argmin(sum((y - W(x)).^2) +
            (1.4e-6.*sum(abs(S_TV(x))) +
             1.8e-4.*sum(abs(S_wavelet(x)).^p)), x)
```

The SDMM and PPXA algorithms are also suited for multi-GPU implementation: in fact the equation groups ((3),(6),(9)), ((4),(7),(10)) and ((5),(8),(11)) in Algorithm 1 and similarly ((3),(5),(7)) and ((4),(6),(8)) in Algorithm 2 are independent from each other and eligible for multi-GPU execution. In this case, each GPU may compute a different subproblem. In GASPACHO, these dependencies are detected automatically at runtime and when the GASPACHO program is executed in multi-GPU mode, the available GPUs are used for the calculations.

2.4 Modularity

The numerical solver often consists of building blocks that are numerical solvers themselves. Many problems naturally decompose into subproblems that are easy to solve using "simple" efficient solvers and this should be reflected in the design of the framework. For example, for image deblurring where the blur is assumed to be a result of a circular convolution using L^2 regularization of the unknown image, the estimation of the noise-free image is best done in the discrete Fourier domain: the least squares formulation then leads to a point-wise Wiener filter. Alternatively, many sparse solvers (e.g., based on ADMM,[12] augmented Lagrangrian,[13] primal-dual techniques[14]) use splitting variables to split the problem into a set of subproblems that are more easy to solve. Then, depending on the structure of the involved matrices and the cost function, a particular solver can be used for each subproblem. An elegant example is the proximal framework[15] for convex optimization: the optimization algorithms decompose into a sequence of proximal operators. A proximal operator is an operator related to a convex function f defined by:

$$\operatorname{prox}_f(\mathbf{v}) = \arg\min_{\mathbf{x}} \left(f(\mathbf{x}) + \frac{1}{2}\|\mathbf{x}-\mathbf{v}\|_2^2 \right). \tag{7}$$

From the above definition it is apparent that the proximal operator defines a subproblem that can be solved again using convex optimization. With GASPACHO, we can use reductions, leading to specialized solvers. Listing 2.4 lists several proximal operators from ref. [15, Table 10.2].

```
rectify = $(y,\lambda) \to \begin{cases} \lambda y & \text{if } y \geq 0 \\ +\infty & \text{otherwise} \end{cases}$
pointwise_l2_inverse = $(y,\lambda) \to \frac{1}{\lambda+1} y$
pointwise_l1_inverse = $(y,\lambda) \to \text{sign}(y)\max(0,|y|-\lambda)$
pointwise_proxmax_inverse = $(y,\lambda) \to y\mathrm{I}[|y| < \lambda] + \text{sign}(y)\lambda\mathrm{I}[\lambda \leq |y| < 2\lambda] + \text{sign}(y)(|y|-\lambda)\mathrm{I}[|y| \geq 2\lambda]$
pointwise_proxrectify_inverse = $(y,\lambda) \to \max(0,y-\lambda)$

reduction (x, y, W)
    → argmin(sum((y − W(x)).^2), x) = conjugate_gradients(y,($x \to x$),W)
reduction (x, y, W)
    → argmin(sum((y − W(x)).^2), x) = conjugate_gradients(y,W)
reduction (x, y, λ)
    → argmin(sum((y − x).^2) + λ.*sum(x.^2), x) = pointwise_l2_inverse(y,λ)
reduction (x, y, λ)
    → argmin(sum((y − x).^2) + λ.*sum(abs(x)), x) = pointwise_l1_inverse(y,λ)
reduction (x, y, λ)
    → argmin(sum((y − x).^2) + λ.*max(abs(x)−λ,0),x) = pointwise_proxmax_inverse(y, λ)
reduction (x, y, λ)
    → argmin(sum((y − x).^2) + λ.* rectify(x),x) = pointwise_proxrectify_inverse(y, λ)
```

Listing 3. GASPACHO specification of the proximal operators (according to ref. [15, Table 10.2]).

In Listing 2.4, the first group of functions are implementations of proximal operators, the reductions specify under which conditions these proximal operators can be used. Modularity can be achieved by simply reverting on subsolvers, i.e., by using the `argmin` function inside the implementation of the proximal operator. The reduction system then determines the most suitable solver for this specific subproblem, giving feedback to the user. To illustrate this, an example implementation of the split-Bregman method is given in Listing 2.4. It can be noted that the implementation almost maps one-to-one onto its algorithmic description. Hence this also greatly reduces the chance of implementation errors and reduces the possibility of a gap between the implementation and the algorithm description in the paper.

```
function x = split_bregman_solver(y, W, Φ, max_iter, λ)
   b = d = 0
   μ = 0.01 % penalizer parameter
   for i=1..max_iter
       % Solve the L2−problem
       x = argmin(sum((y − W(x)).^2) + μ * sum(Phi(x).^2), x)

       % Solve the L1−problem
       d = argmin(sum((b − Φ(x)).^2) + $\frac{\lambda}{\mu}$ * sum(abs(b)), x)
       b += Φ(x) − d
   endfor
endfunction

reduction (x, y, W, Φ) → argmin(sum((y − W(x)).^2) + λ * sum(abs(Φ(x))), x) =
    split_bregman_solver(y, W, Φ, 10, λ)
```

Listing 4. Modular and generic implementation of a split-Bregman solver in GASPACHO.

Again, the calculation approach is *matrix*-free, the operator W is implemented as a function that is evaluated on the fly. For the modularity to be effective in practice, we have found that it is necessary 1) to be able to specify properties of operators and 2) to deduce certain derivatives of operators (for example, the adjoint and the derivative). To enjoy the benefits of automatic program analysis, the following set of 'convenience' meta-functions was therefore added to GASPACHO:

- `$subs(A=B, x)`: performs substitution `A=B` in the expression `x`.
- `$islinear(A(x), x)`: determines whether the operator `A` is linear in `x`. This is obtained by means of program analysis.
- `$ispointwise(A(x), x)`: determines whether `A(x)` is a pointwise operator (in `x`).
- `$adjoint(A(x), x)`: *(algorithmic adjoint calculation)* derives an algorithm for calculating the adjoint of the operator `A(x)` (assuming that the operator is linear in `x`). See Section 3.
- `$diff(f(x), x)`: *(algorithmic differentiation)* derives an algorithm for calculating the derivative of the function `f(x)` with respect to `x`. See Section 3.
- `$diffm(f(x), x, y)`: *(algorithmic differentiation)* derives an algorithm for calculating the inner product of the derivative of the function `f(x)` with respect to `x`, and a vector `y`: $\left(\frac{\partial f}{\partial \mathbf{x}}\right)^T \mathbf{y}$. See Section 3 for more details.

Alternatively, when program analysis is inconclusive, it is possible to "manually" specify some properties of operators. This can be done again via reductions: the following reduction indicates that the derivative of an orthogonal wavelet transform is obtained through its inverse transform:

```
reduction (x, y) → $diffm(S_wavelet(x), x, y) = S_wavelet_inv(y)
```

Obviously, this reduction can be generalized to arbitrary orthogonal operators, as follows:

```
reduction () → isorthogonal(S_wavelet) = true
reduction (x) → $adjoint(S_wavelet(x), x) = S_wavelet_inv(x)
reduction (x, y, S) → diffm(S(x), x, y) = $subs(x=y,$adjoint(S(x), x)) where
    isorthogonal(S)
```

Here, the first three lines specify the orthogonality property of `S_wavelet` and the inverse operator for `S_wavelet`, while the third line specifies how the derivative of a linear orthogonal operator needs to be calculated. Similarly, using the above mechanism to specify operator properties directly within the DSL, it becomes quite straightforward to implement solvers for quadratic problems:

```
reduction (x, y, A)   → argmin(sum((y − A(x)).^2), x) = conjugate_gradient(y,A)
reduction (x, y, A)   → argmin(sum((y − A(x)).^2), x) = pointwise_L2_solver(y,A) where
    $ispointwise(A(x),x)
reduction (x, y, A)   → argmin(sum((y − A(x)).^2), x) = circulant_L2_solver(y,A) where
    $iscirculant(A(x),x)
```

This way, the DSL compiler can choose between different optimization algorithms, depending on known properties for the operators (e.g., whether they are linear, pointwise, circulant, etc.). A library designer may implement a whole range of optimization methods in a generic way (such as Listing 2.4), specify the conditions of use in terms of a reduction and the end-user does not need to know most specifics of the optimization methods.

3. ALGORITHMIC DIFFERENTIATION/ADJOINT CALCULATION

Algorithmic differentiation (AD) (also known as *automatic* differentiation) is a means to numerically evaluate the derivative of a linear or non-linear function specified by a computer program.[6] AD exploits the fact that a computer program consists of a sequence of arithmetic operations, no matter how complicated the program is. In practice, AD is a better alternative for both symbolic differentiation and numerical differentiation. Symbolic differentiation faces the difficulty of translating a computer program to a single expression (often resulting in inefficient code generation) and numerical differentiation suffers from round-off errors in the discretization process. AD does not have any of these problems but the complexity is rather shifted to the algorithmic techniques used to calculate the derivative of the function. In GASPACHO, we use source-to-source transformations for implementing AD; one of the strong points is then that AD can be applied to computer programs containing control flow. AD often needs to deal with the abstract syntax tree of a programming language, which may involve incomplete information, e.g., the dimensions of vectors or matrices, the types of certain variables. Many existing AD toolboxes relax some of the complexity by disallowing certain programming constructs (e.g., pointers, user-defined types) and by generate sequential algorithms due to the use of the "tape device" (which sequentially records memory operations allowing these operations to be played back in reverse). Such sequential algorithms are nontrivial to parallelize by existing compilers, even though theoretically, a parallel implementation exists.

To work around the difficulties of implementing AD, several packages (e.g., Theano, TensorFlow) use a computational graph: in such a graph every node specifies a computation operation, while the edges represent the dependencies between the different computations. AD can then straightforwardly be implemented by traversing the computational graph.

Essentially, AD represents each statement of a program as a composition of functions with input and output variables and then applies the chain-rule to them to calculate the derivative with respect to the desired variable. GASPACHO uses two different modes of AD, which differ in the way that the chain rule is applied. Assume in the following that F is a vector function with a scalar output and G and H are vector functions with components respectively G_i and H_i.

- *Forward mode*: the chain-rule is evaluated in forward direction by substituting the derivatives of the inner functions:

$$\frac{\partial}{\partial \mathbf{x}} F(G(H(\mathbf{x}))) = \sum_i \frac{\partial F}{\partial G_i} \sum_j \left(\frac{\partial G_i}{\partial H_j} \frac{\partial H_j}{\partial \mathbf{x}} \right). \tag{8}$$

- *Backward mode*: the derivatives of the outer function are substituted during the application of the chain rule:

$$\frac{\partial}{\partial \mathbf{x}} F(G(H(\mathbf{x}))) = \sum_{i,j} \left(\frac{\partial F}{\partial H_i} \frac{\partial H_i}{\partial G_j} \right) \frac{\partial G_j}{\partial \mathbf{x}} = \sum_j \mathrm{Tr} \left(\left[\frac{\partial F}{\partial \mathbf{H}} \right]^T \frac{\partial \mathbf{H}}{\partial G_j} \right) \frac{\partial G_j}{\partial \mathbf{x}}. \tag{9}$$

with $\mathrm{Tr}(\cdot)$ the matrix trace.

At first sight, the difference between both modes seems to be a matter of taste. However, considering that $\mathbf{x}$ is a vector, the vector derivative with respect to $\mathbf{x}$ is only used in the last factor of (9), the calculation is slightly more simple in the backward mode. In general, Backward AD also generates code that is more easy parallelize (in the sense that less processing steps are required). In particular, backpropagation, as used in many machine learning techniques, is a special case of backward mode AD.[16] Also notice the occurence of $\frac{\partial G_i}{\partial H_j}$ in (8) compared to $\frac{\partial H_i}{\partial G_j}$ in (9). In case $\mathbf{H}(\mathbf{x}) = \mathbf{A}\mathbf{x}$ is a linear function, the former will depend on $\mathbf{A}$ while the latter will depend on the adjoint, i.e. $\mathbf{A}^T$. Therefore it is easy to see that an AD implementation, also gives algorithmic adjoint calculation for free: suppose we have a function $f(\mathbf{x},\mathbf{y}) = \frac{1}{2}||\mathbf{A}\mathbf{x} - \mathbf{y}||^2$, then AD gives the function $\frac{\partial}{\partial \mathbf{x}}(\mathbf{x},\mathbf{y}) = \mathbf{A}^T(\mathbf{A}\mathbf{x} - \mathbf{y})$. Correspondingly, $\mathbf{A}^T\mathbf{y}$ can be evaluated using $-\frac{\partial}{\partial \mathbf{x}}(0,\mathbf{y})$. An example of an automatically derived adjoint algorithm of Listing 2.2 is given in Listing 3.

```
function x:cube = degradation_operator_adjoint(y : cube, w : mat, center : vec2)
    x=zeros(size(y))
    for m=0..size(y,0)-1
        for n=0..size(y,1)-1
            for c=0..2
                sum=y[m,n,c]
                for k=0..size(f,0)-1
                    for l=0..size(f,1)-1
                        x[2*m+k-center[0],2*n+l-center[1],c]+=f[k,l]*sum
                    endfor
                endfor
            endfor
        endfor
    endfor
endfunction
```

Listing 5. Algorithmically derived adjoint program for Listing 2.2 - outcome of the algorithmic adjoint calculation.

Finally, AD is also a modular approach: when AD is applied to a function F, and the function depends on another function G for which the derivative is not known yet, the DSL compiler will also algorithmically differentiate G. To avoid the function from being differentiated multiple times (which would cause code duplication), a reduction (see Subsection 2.3) is generated for each differentiated function. The same happens during adjoint calculation: the DSL compiler may also determine adjoints of dependent functions.

3.1 Evaluation of proximal operators using AD

To find a minimizer of (7) when f is a differentiable function that does not have a closed-form proximal operator, the gradient descent method successively updates the current candidate minimizer $\mathbf{v}_k$ at iteration i:

$$\mathbf{v}_{i+1} = \mathbf{v}_i - \gamma \frac{\partial f}{\partial \mathbf{x}}(\mathbf{v}_i) + \gamma(\mathbf{x} - \mathbf{v}_i) \tag{10}$$

where γ is the gradient descent step size parameter. The partial derivative $\frac{\partial f}{\partial \mathbf{x}}(\mathbf{v}_i)$ can be calculated using AD. Using the DSL, the resulting proximal gradient descent algorithm can then easily be implemented (see Listing 6). Similarly, more advanced optimization methods, such as non-linear conjugate gradients methods, Newton methods and quasi-Newton methods (e.g., L-BFGS) can be defined.

```
function $v$ = prox_op_gradient_descent($f$, $v_0$, $x$, $\gamma$, $\varepsilon$)
    gradient = $unapply($diff($f(x)$, $x$), $x$)
    $v = v_0$ % Initial solution
    repeat
        $v^{\text{old}} = v$
        $v$ = $v^{\text{old}}$ - $\gamma$ gradient($v$) + $\gamma(x - v^{\text{old}})$
    until $\|v - v^{\text{old}}\|_1 < \varepsilon$ % stop condition
endfunction
```

Listing 6. Gradient descent-based proximal operator evaluation.

4. RESULTS AND DISCUSSION

To illustrate the proposed framework in a practical EM restoration application with non-trivial degradation conditions, we write the cost function from (1) in the GASPACHO DSL, based on a shearlet and total variation (TV) sparsity prior. This leads to the simple program from Listing 7. Because the captured EM images contain correlated noise, the noise power spectral density (PSD) was estimated in a separate step from the high frequencies (assuming the image noise is mostly contained in the lower frequencies) and was also included in the modeling.

```
% 1. opening input files and estimate the input noise PSD
im = tiffread(inputfile).data
$C_n^{\text{inv}}$ = estimate_noise_autocov(im)

% 2. setting up the shearlet and Total Variation (TV) transforms
```

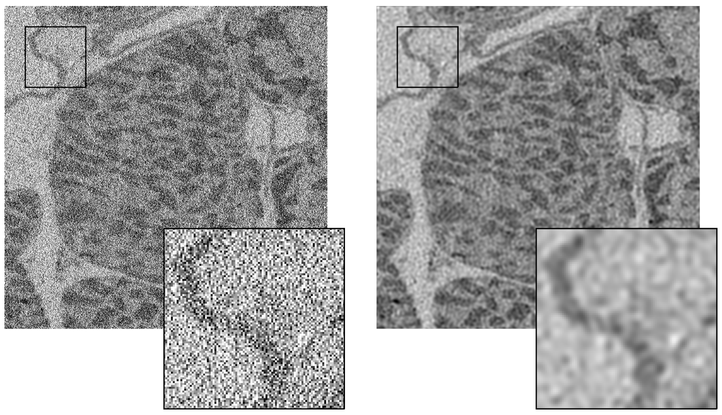

(a) (b)

Figure 4. A visual result for Electron Microscopy deconvolution. (a) Degraded input image, (b) restored image.

```
 [S1, S1_H] = build_dst2d(size(im,0..2), 8, 4)
 [S2, S2_H] = build_tvtransform()
 mask = gaussianfilter_kernel_2d([8,8],[2,2],pi/2)
 mask = mask / sum(mask)
W = x → degradation_operator(x, mask, [4, 4])

% 3. define the adjoint Shearlet and TV transforms
 reduction x → $adjoint(S1(x),x) = S1_H(x)
 reduction x → $adjoint(S2(x),x) = S2_H(x)

% 4. Calculate the result
 x = argmin(sum((C_n^inv(y - W(x))).^2) + λ1*sum(abs(S1(x))) + λ2*sum(abs(S2(x))), x)
```

Listing 7. GASPACHO image restoration program. Some of the functions were omitted because of the page limitation. In particular, `estimate_noise_autocov` estimates the input noise autocovariance, `build_dst2d` and `build_tvtransform` respectively construct a 2D shearlet transform and a Total Variation transform. `gaussianfilter_kernel_2d` calculates a 2D Gaussian filter of 8×8 taps with parameter $\sigma = [2,2]$.

By setting the parameters λ_1 and λ_2 we are able to trade-off the shearlet regularization versus the Total Variation regularization. In Listing 7, the merits of GASPACHO are clear: 1) the dense and readable code, 2) the easiness to include extra priors in the cost function, 3) the ERS that, via the `argmin` reduction, allows automatically solving the convex optimization problem, allowing the user to focus on the design aspects rather than implementation details.

The visual results (see Figure 4) indicate that despite the complex restoration task, the GASPACHO program allows to remove well the noise in the EM image while preserving the image details. We note that there are some model factors that are not taken into account in our modeling: the presence of mixed Gaussian and Poissonian noise, positivity of the ideal image and quantization noise. Taking these factors into account leads to a more extended GASPACHO program, but also potentially even better restoration results. GASPACHO is a convenient framework for experimenting with such image and degradation models. Thanks to the Quasar language and runtime underlying GASPACHO, the entire program was fully accelerated on GPU, leading in this case to a speed-up of about 15x compared to the equivalent multi-core CPU execution.

5. CONCLUSION

Many large scale inverse problems in image processing share many commonalities, and researchers often face the challenge of designing and implementing different cost functions, optimization algorithms, regularization terms hand-tailored to the application of interest. With GASPACHO, we separate the design and implementation problem, in a similar way as ProxImaL, allowing the user to experiment with the problem formulations, the necessary cost functions in a concise DSL specification (named GASPACHO), while the DSL compiler takes care of the hard and error-prone parts of the work. In this paper, we have discussed how two components of the DSL, namely, the expression rewriting system and the modularity, help to decompose convex optimization problems into subproblems, according to the proximal framework for convex optimization. Our approach is flexible and allows alternative optimization methods to be easily added without requiring changes to the DSL compiler. Underlying techniques rely on a matrix-free representation of the calculation and source-to-source mappings of the operations (e.g., algorithmic differentiation and adjoint calculation). Furthermore, the high-level characteristics of the DSL and the generated source code permit easy parallelization both on the fine grain level (GPU kernels) as well as on a coarse grain level (e.g., multi-GPU and/or distributed processing). We have successfully illustrated our approach for an EM deconvolution problem, in which the entire application knowledge could be described in just a few lines of code.

6. ACKNOWLEDGEMENTS

We would like to thank Saskia Lippens and Christopher J. Guérin of the Bio Imaging Core of the Flemish Institute for Biotechnology (VIB) for providing the EM data.

7. APPENDIX: SDMM AND PPXA ALGORITHMS

For reference, we provide the SDMM and PPXA algorithms (Algorithm 1 and Algorithm 2) from Combettes et al.[17] that were used in this paper.

Algorithm 1 The simultaneous-direction method of multipliers (SDMM).

```
Initialization: $\mathbf{x}^{(0)} = 0, \mathbf{s}_1^{(0)} = \mathbf{s}_2^{(0)} = \mathbf{s}_3^{(0)} = 0, \mathbf{d}_1^{(0)} = \mathbf{d}_2^{(0)} = \mathbf{d}_3^{(0)} = 0, \mathbf{z}_1^{(0)} = \mathbf{z}_2^{(0)} = \mathbf{z}_3^{(0)} = 0$. Fix $\lambda, \gamma$. $i = 0$.
1) do
2)      $\mathbf{x}^{(i+1)} = \left(\mathbf{W}^T\mathbf{W} + \mathbf{S}_1^T\mathbf{S}_1 + \mathbf{S}_2^T\mathbf{S}_2\right)^{-1}\left(\mathbf{W}^T\left(\mathbf{x}_1^{(i)} - \mathbf{z}_1^{(i)}\right) + \mathbf{S}_1^T\left(\mathbf{y}_2^{(i)} - \mathbf{z}_2^{(i)}\right) + \mathbf{S}_2^T\left(\mathbf{y}_3^{(i)} - \mathbf{z}_3^{(i)}\right)\right)$
3)      $\mathbf{s}_1^{(i+1)} = \mathbf{W}\mathbf{x}^{(i+1)}$
4)      $\mathbf{s}_2^{(i+1)} = \mathbf{S}_1\mathbf{x}^{(i+1)}$
5)      $\mathbf{s}_3^{(i+1)} = \mathbf{S}_2\mathbf{x}^{(i+1)}$
6)      $\mathbf{d}_1^{(i+1)} = \left(\gamma\mathbf{y} + \mathbf{s}_1^{(i+1)} + \mathbf{z}_1^{(i)}\right) / (\gamma + 1)$
7)      $\mathbf{d}_2^{(i+1)} = \text{softthresh}\left(\mathbf{s}_2^{(\mathrm{i}+1)} + \mathbf{z}_2^{(\mathrm{i})}, \gamma\lambda\right)$
8)      $\mathbf{d}_3^{(i+1)} = \text{softthresh}\left(\mathbf{s}_3^{(\mathrm{i}+1)} + \mathbf{z}_3^{(\mathrm{i})}, \gamma\lambda\right)$
9)      $\mathbf{z}_1^{(i+1)} = \mathbf{z}_1^{(i)} + \left(\mathbf{s}_1^{(i+1)} - y_1^{(i+1)}\right)$
10)     $\mathbf{z}_2^{(i+1)} = \mathbf{z}_2^{(i)} + \left(\mathbf{s}_2^{(i+1)} - y_2^{(i+1)}\right)$
11)     $\mathbf{z}_3^{(i+1)} = \mathbf{z}_3^{(i)} + \left(\mathbf{s}_3^{(i+1)} - y_3^{(i+1)}\right)$
12)     increment $i$
13) until $\left\|\mathbf{x}^{(i)} - \mathbf{x}^{(i-1)}\right\| \leq \varepsilon$
```

REFERENCES

1. Boyd, S., Parikh, N., Chu, E., and Peleato, B., "Distributed optimization via alternating direction method of multipliers," *Foundations and Trends in Machine Learning* **3**, 1–122 (2010).
2. Afonso, M. V., Bioucas-Dias, J. M., and Figueiredo, M. A., "An augmented lagrangian approach to the constrained optimization formulation of imaging inverse problems," *IEEE Trans. on Image Processing* **20**(3), 681–695 (2011).
3. Fox, G. C., Williams, R. D., and Messina, G. C., [*Parallel computing works!*], Morgan Kaufmann (2014).
4. Heide, F., Diamond, S., Nießner, M., Ragan-Kelley, J., Heidrich, W., and Wetzstein, G., "Proximal: Efficient image optimization using proximal algorithms," *ACM Transactions on Graphics (TOG)* **35**(4), 84 (2016).
5. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S., "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *ACM SIGPLAN Notices* **48**(6), 519–530 (2013).

Algorithm 2 Parallel proximal algorithm (PPXA).

```
Initialization: Fix $\lambda, \gamma > 0, \varepsilon \in ]0,1[$, $(\omega_1, \omega_2, \omega_3) \in ]0,1]^2$ such that $\omega_1 + \omega_2 + \omega_3 = 1$. Fix $\mathbf{x}_1^{(0)} \in \mathbb{R}^{4MN}$, $\mathbf{d}_1^{(0)}, \mathbf{d}_2^{(0)}, \mathbf{d}_3^{(0)}$. Set $i = 0$. Set $\mathbf{d}_1^{(0)} = \mathbf{d}_2^{(0)} = 0$, $\mathbf{b}_1^{(0)} = \mathbf{b}_2^{(0)} = 0$, $p_2^{(0)} = 0$.
Set $\mathbf{x}^{(0)} = \omega_1 \mathbf{x}_1^{(0)} + \omega_2 \mathbf{x}_2^{(0)} + \omega_3 \mathbf{x}_3^{(0)}$.
1) do
2)     $\mathbf{p}_1^{(i+1)} = \left(\frac{\gamma}{\omega_1}\mathbf{W}\mathbf{W}^T + I\right)^{-1}\left(\frac{\gamma}{\omega_1}\mathbf{W}^T\mathbf{y} + \mathbf{x}^{(i)}\right)$
3)     $\mathbf{d}_1^{(i+1)} = \text{softthresh}\left(\mathbf{S}_1\mathbf{p}_2^{(\mathrm{i})} + \mathbf{b}_1^{(\mathrm{i})}, \frac{\lambda\gamma}{\kappa\omega_2}\right)$
4)     $\mathbf{d}_2^{(i+1)} = \text{softthresh}\left(\mathbf{S}_2\mathbf{p}_3^{(\mathrm{i})} + \mathbf{b}_2^{(\mathrm{i})}, \frac{\lambda\gamma}{\kappa\omega_3}\right)$
5)     $\mathbf{p}_2^{(i+1)} = \left(I + \kappa\mathbf{S}_1\mathbf{S}_1^T\right)^{-1}\left(\mathbf{x}_2^{(i)} + \kappa\mathbf{S}_1^T\left(\mathbf{d}_1^{(i+1)} + \mathbf{b}_1^{(i+1)}\right)\right)$
6)     $\mathbf{p}_3^{(i+1)} = \left(I + \kappa\mathbf{S}_2\mathbf{S}_2^T\right)^{-1}\left(\mathbf{x}_3^{(i)} + \kappa\mathbf{S}_1^T\left(\mathbf{d}_2^{(i+1)} + \mathbf{b}_2^{(i+1)}\right)\right)$
7)     $\mathbf{b}_1^{(i+1)} = \mathbf{b}_1^{(i)} + \left(\mathbf{S}_1\mathbf{p}_2^{(i+1)} - \mathbf{d}_1^{(i)}\right)$
8)     $\mathbf{b}_2^{(i+1)} = \mathbf{b}_2^{(i)} + \left(\mathbf{S}_2\mathbf{p}_3^{(i+1)} - \mathbf{d}_2^{(i)}\right)$
9)     $\mathbf{p}^{(i+1)} = \omega_1\mathbf{p}_1^{(i+1)} + \omega_2\mathbf{p}_2^{(i+1)} + \omega_3\mathbf{p}_3^{(i+1)}$
10)    Select $\zeta_i$ such that $\varepsilon \le \zeta_i \le 2 - \varepsilon$
11)    $\mathbf{x}_j^{(i+1)} = \mathbf{x}_j^{(i)} + \zeta_i\left(2p^{(i+1)} - \mathbf{x}^{(i)} - p_j^{(i)}\right)$, $j \in \{1,2,3\}$
12)    $\mathbf{x}^{(i+1)} = \mathbf{x}^{(i)} + \zeta_i\left(p^{(i+1)} - \mathbf{x}^{(i)}\right)$
13)    increment $i$
14) until $\left\|\mathbf{x}^{(i)} - \mathbf{x}^{(i-1)}\right\| \le \varepsilon$
```

6. Pock, T., Pock, M., and Bischof, H., "Algorithmic differentiation: Application to variational problems in computer vision," *IEEE Transactions on Pattern Analysis and Machine Intelligence* **29**(7), 1180–1193 (2007).
7. Goossens, B., De Vylder, J., and Philips, W., "Quasar - a new heterogeneous programming framework for image and video processing algorithms on CPU and GPU," in [*Image Processing (ICIP), 2014 IEEE International Conference on*], 2183–2185, IEEE (2014).
8. Felgenhauer, B., Avanzini, M., and Sternagel, C., "A Haskell Library for Term Rewriting," *arXiv preprint arXiv:1307.2328* (2013).
9. Wolfram, S., [*Mathematica: a system for doing mathematics by computer*], Addison-Wesley (1991).
10. Armando, A. and Ballarin, C., "Maple's evaluation process as constraint contextual rewriting," in [*Proceedings of the 2001 international symposium on Symbolic and algebraic computation*], 32–37, ACM (2001).
11. Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh, F. T., "Basic linear algebra subprograms for fortran usage," *ACM Transactions on Mathematical Software (TOMS)* **5**(3), 308–323 (1979).
12. Boyd, S., Parikh, N., Chu, E., Peleato, B., and Eckstein, J., "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Foundations and Trends® in Machine Learning* **3**(1), 1–122 (2011).
13. Tai, X.-C. and Wu, C., "Augmented Lagrangian method, dual methods and split Bregman iteration for ROF model," *Scale space and variational methods in computer vision* , 502–513 (2009).
14. Chambolle, A. and Pock, T., "A first-order primal-dual algorithm for convex problems with applications to imaging," *Journal of mathematical imaging and vision* **40**(1), 120–145 (2011).
15. Combettes, P. L. and Pesquet, J.-C., "Proximal splitting methods in signal processing," in [*Fixed-point algorithms for inverse problems in science and engineering*], 185–212, Springer (2011).
16. Goodfellow, I., Bengio, Y., and Courville, A., [*Deep learning*], MIT press (2016).
17. Combettes, P. and Pesquet, J.-C., "A Douglas-Rachford splitting approach to nonsmooth convex variational signal recovery," *IEEE Journal of Selected Topics in Signal Processing* **1**, 1–12 (Dec. 2007).