ProxImaL: Efficient Image Optimization using Proximal Algorithms

Felix Heide[1,2] Steven Diamond[1] Matthias Nießner[1] Jonathan Ragan-Kelley[1] Wolfgang Heidrich[3,2] Gordon Wetzstein[1]

[1]Stanford University [2]University of British Columbia [3]KAUST

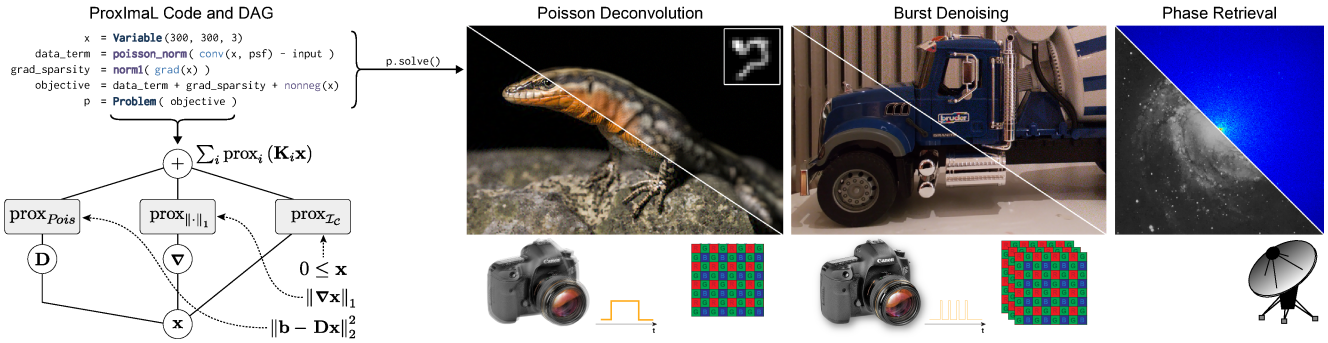


Figure 1: *As a domain-specific language, ProxImaL makes it easy to prototype a range of inverse problems in imaging. For example, we show ProxImaL code for deconvolution in the presence of Poisson-distributed shot noise and the corresponding directed acyclic graph (DAG, left) as well as the results generated by the compiled optimization algorithm (center left). The ProxImaL language makes it easy to prototype highly-efficient optimization routines for problems as diverse as denoising a stack of images (center right), or even nonlinear problems such as phase retrieval (right). For any of these applications, ProxImaL allows for different penalty functions, advanced image priors, and also different optimization algorithms to be rapidly evaluated. We demonstrate state-of-the-art results for these and other imaging applications.*

Abstract

Computational photography systems are becoming increasingly diverse, while computational resources—for example on mobile platforms—are rapidly increasing. As diverse as these camera systems may be, slightly different variants of the underlying image processing tasks, such as demosaicking, deconvolution, denoising, inpainting, image fusion, and alignment, are shared between all of these systems. Formal optimization methods have recently been demonstrated to achieve state-of-the-art quality for many of these applications. Unfortunately, different combinations of natural image priors and optimization algorithms may be optimal for different problems, and implementing and testing each combination is currently a time-consuming and error-prone process. ProxImaL is a domain-specific language and compiler for image optimization problems that makes it easy to experiment with different problem formulations and algorithm choices. The language uses proximal operators as the fundamental building blocks of a variety of linear and nonlinear image formation models and cost functions, advanced image priors, and noise models. The compiler intelligently chooses the best way to translate a problem formulation and choice of optimization algorithm into an efficient solver implementation. In applications to the image processing pipeline, deconvolution in the presence of Poisson-distributed shot noise, and burst denoising, we show that a few lines of ProxImaL code can generate highly efficient solvers that achieve state-of-the-art results. We also show applications to the nonlinear and nonconvex problem of phase retrieval.

Keywords: computational photography, digital image processing, optimization

Concepts: •**Computing methodologies** → *Computational photography; Regularization;* •**Mathematics of computing** → *Continuous optimization; Solvers;*

SIGGRAPH '16 Technical Paper,, July 24 - 28, 2016, Anaheim, CA,
ISBN: 978-1-4503-4279-7/16/07
DOI: http://dx.doi.org/10.1145/2897824.2925875

1 Introduction

Digital image processing is a research area with a wide variety of applications in computational photography, computer vision, robotics, scientific imaging, remote sensing, microscopy, and computer graphics. Traditionally, image processing algorithms have been tailored independently to each of these applications, with few techniques that generalize across fields. Researchers have recognized that all these applications share the fundamental task of recovering information from blurry, noisy, saturated, sparsely or indirectly sampled, or otherwise corrupted measurements. In particular, the information recovery task can be formulated as an optimization problem. Methods that approach image processing as solving an optimization problem have achieved state-of-the-art results in many classical applications, for example the demosaicking, deconvolution, and inpainting tasks of the image processing pipeline [Heide et al. 2014].

Optimization-based image processing generalizes across application areas, and thus, in theory, makes it easy to develop image processing techniques for new domains. In practice, however, developing image optimization methods can be difficult, because there are many ways to frame an image processing task as an optimization problem, and it is virtually impossible to predict which framing will yield the best results. This requires researchers to experiment with a wide variety of optimization approaches to see which works best for a given task.

For instance, consider the classic problem of deconvolution: we are given measurements $\mathbf{b}$ that approximately satisfy $\mathbf{b} = \mathbf{D}\mathbf{x}$, where $\mathbf{D}$ is a linear operator representing convolution with a known kernel and $\mathbf{x}$ is an unknown image. Our goal is to recover $\mathbf{x}$ given $\mathbf{b}$. The optimization-based approach to deconvolution is to say that $\mathbf{x}$ is the

solution to the optimization problem:

$$\text{minimize} \quad f(\mathbf{D}\mathbf{x} - \mathbf{b}) + r(\mathbf{x}), \tag{1}$$

where f is an error metric, and r is a penalty function that expresses prior knowledge about the image $\mathbf{x}$.

There are many reasonable choices for f and r. For instance, we might define f as a sum-of-squares error, a Huber loss, or a Poisson penalty. The penalty function r could be a constraint on the range of the values of $\mathbf{x}$, a sparsity-inducing penalty such as total-variation, a non-local patch prior as in the BM3D-based reconstruction shown in [Danielyan et al. 2012], or a combination of all these penalties.

Once we have chosen f and r, we must choose an algorithm to use for solving the optimization problem. Dozens of different optimization algorithms have been applied to image optimization problems, such as the alternating direction method of multipliers (ADMM) [Boyd et al. 2011], the primal-dual algorithm by Chambolle and Pock [2011], and half-quadratic splitting [Geman and Yang 1995].

Moreover, for each algorithm there may be many ways to translate Problem (1) into that algorithm's standard form. The only way to know which algorithm and translation into standard form works best for a problem is to try all of them.

Finding an effective image optimization method thus requires exploring a large space of problem formulations, algorithms, and translations between standard forms. Currently, researchers must develop a new solver implementation for each point they explore in the space, which is a time-consuming and error-prone process. Developing implementations is particularly challenging for image optimization problems since these problems typically involve millions of variables and can only be solved efficiently by exploiting problem structure.

In this paper, we address these challenges by introducing *ProxImaL*, a domain-specific language (DSL) for image optimization. The ProxImaL language allows users to describe image optimization problems in a few lines of code using an intuitive syntax that follows the math. Users write their problem using a fixed set of mathematical functions, whose structure can be exploited to generate an efficient solver. Most functions that occur in image optimization problems are included in the language, and it is easy to add support for more. Compositions of functions are limited by a set of simple rules that ensure the problems constructed by the user match our standard mathematical representation.

The ProxImaL compiler takes the user's problem description and choice of algorithm and automatically generates a solver implementation. The compiler considers a wide range of possible solver implementations and selects one based on expert knowledge about how to best formulate problems for the chosen solver algorithm. The user can also easily override the compiler's default choice to try out more implementations. The solver implementations generated by the compiler are highly efficient because we created optimized code for the core mathematical operations using Halide [Ragan-Kelley et al. 2013].

We demonstrate the utility of ProxImaL through applications to the image processing pipeline, burst photography and denoising, deconvolution, and phase retrieval. In many cases a few lines of ProxImaL code and the default solver implementation generated by the ProxImaL compiler achieves state-of-the-art results, often with a runtime under ten seconds.

We make the following contributions in this paper:

- We developed a simple language and mathematical representation for image optimization problems that captures the problem structure needed to generate an efficient solver.
- We built a compiler that takes the user's problem description and choice of solver algorithm and automatically generates an efficient solver, intelligently choosing from the many translations possible.
- We show that our framework can achieve state-of-the-art results on a variety of image optimization problems while also producing highly efficient solver implementations.

2 Related Work

Languages for Graphics and Image Processing Domain-specific languages for graphics and rendering have successfully made the transition from research to industry standard [Foley and Hanrahan 2011]. Today, general-purpose languages for GPU programming, such as CUDA, are popular for many applications beyond graphics. OpenCL extended this concept to heterogeneous computing platforms. Domain-specificity can be exploited to accelerate the execution of common tasks in a particular domain, for example in image processing [Ragan-Kelley et al. 2013], physical simulation [Bernstein et al. 2015], or multi-material 3D printing [Vidimice et al. 2013]. Most of these languages and systems focus on finding a domain-specific tradeoff between intuitive use and high-performance execution. ProxImaL follows this strategy but we build on formal optimization methods to develop a language and compiler for image optimization.

Optimization for Image Processing Over the past years, numerical optimization has become a standard tool for solving a number of classical restoration and reconstruction problems in computational photography. Examples include blind [Fergus et al. 2006] and non-blind [Krishnan and Fergus 2009; Joshi et al. 2009] deconvolution, image denoising [Zoran and Weiss 2011], and inpainting [Bertalmio et al. 2000]. Optimization has been successfully applied to image editing problems such as tonemapping [Fattal et al. 2002], Poisson-blending [Levin et al. 2004b] and colorization [Levin et al. 2004a]. Very efficient solvers have been developed for most of these problems [Krishnan and Szeliski 2011; Schmidt and Roth 2014]. Optimization techniques are also becoming increasingly popular solutions for scientific imaging problems such as x-ray tomography [Sidky and Pan 2008] and phase retrieval [Tian and Waller 2015]. Recently, it was shown that a large subset of low-level image processing problems can be solved through a single proximal algorithm framework [Heide et al. 2014].

Optimization and Optimization Languages The literature on algorithms for solving image optimization problems is extensive. A particularly fruitful line of research has focused on solving convex optimization problems using operator splitting methods and proximal algorithms [Parikh and Boyd 2013]. Prominent examples of such methods include the proximal point algorithm [Rockafellar 1976], forward-backward splitting [Bruck 1975], the Pock-Chambolle algorithm [Chambolle and Pock 2011; Pock et al. 2009], the split Bregman method [Goldstein and Osher 2009], ISTA and FISTA [Beck and Teboulle 2009], the alternating direction method of multipliers [Boyd et al. 2011], PDHG [Esser et al. 2010], and half-quadratic splitting [Geman and Yang 1995]. Recent work has applied these methods to nonconvex optimization problems and found conditions that guarantee convergence (though not necessarily to the global optimum); see, *e.g.*, [Attouch et al. 2011; Möllenhoff et al. 2015; Li and Pong 2015].

DSLs for optimization have a long history, going back to GAMS [Brooke et al. 1988] in the 1970s, and including DSLs specialized for convex optimization, such as CVX [Grant and Boyd 2014], YALMIP [Lofberg 2004], CVXPY [Diamond and Boyd 2016b], and Convex.jl

[Udell et al. 2014]. These approaches reliably solve modest size problems, with on the order of $10,000$s of variables, but for image optimization problems with millions of variables these solvers become infeasible due to their memory and computational cost. There have been several different approaches towards making an optimization DSL or framework that can handle large problems such as occur in image optimization. The approach in [Diamond and Boyd 2015] extends CVXPY to recognize and exploit fast linear transforms, such as convolution and the discrete Fourier transform. The Epsilon framework takes advantage of fast proximal operators for individual functions, transforming problems so they can be efficiently solved by a variant of ADMM [Wytock et al. 2015]. The TFOCS framework makes it easy to apply a variety of proximal and first order algorithms to optimization problems, and accommodates fast linear transforms [Becker et al. 2011]. None of these systems can compete with existing specialized solvers for individual image processing problems, however, and they are also limited to convex problems.

3 Representing Image Optimization Problems

We model an image optimization problem as a sum of penalties f_i on linear transforms $\mathbf{K}_i\mathbf{x}$ with $\mathbf{x} \in \mathbb{R}^n$ being the unknown:

$$\operatorname*{argmin}_{\mathbf{x}} \sum_{i=1}^{I} f_i\left(\mathbf{K}_i\mathbf{x}\right) \quad \text{with} \quad \mathbf{K} = \begin{bmatrix}\mathbf{K}_1 \\ \vdots \\ \mathbf{K}_I\end{bmatrix}, \tag{2}$$

where here $\mathbf{K} \in \mathbb{R}^{m\times n}$ is one large matrix that is composed of stacked linear operators $\mathbf{K}_1, \ldots, \mathbf{K}_I$. The linear operator $\mathbf{K}_i \in \mathbb{R}^{m_i\times n}$ selects a subset of m_i rows of $\mathbf{K}\mathbf{x}$. This subset of rows is then the input for the penalty functions $f_i : \mathbb{R}^{m_i} \to \mathbb{R}$.

Image optimization problems generally contain

- variables representing the image(s) to be reconstructed,
- a forward model of image formation in terms of linear operators,
- a penalty based on the difference of the results of this forward model from measured data,
- and priors and constraints on the the variables.

For example, consider a slightly more complex version of the deconvolution problem (1) where the convolved image $\mathbf{D}\mathbf{x}$ is subsampled by a known demosaicking pattern, which we represent with the linear operator $\mathbf{M}$. We formulate our problem using a sum-of-squares error metric, $f(\mathbf{x}) = \|\mathbf{M}\mathbf{D}\mathbf{x} - \mathbf{b}\|_2^2$, and the penalty function:

$$r(\mathbf{x}) = \mu\|\nabla\mathbf{x}\|_1 + (1-\mu)\|\nabla\mathbf{x}\|_2^2 + I_{[0,\infty)}(\mathbf{x}),$$

where $\mu \in [0,1]$, ∇ is the gradient operator, and:

$$I_{[0,\infty)}(\mathbf{x}) = \begin{cases} 0, & \text{if } \mathbf{x} \geq 0 \\ \infty, & \text{otherwise.}\end{cases}$$

The penalty function encodes the priors that many gradients are zero and the pixel values are nonnegative. Problem (3) shows the full optimization problem and how we represent it in the form of Problem (2).

$$\mathbf{x}_{\text{opt}} = \operatorname*{argmin}_{\mathbf{x}} \ \|\mathbf{M}\mathbf{D}\mathbf{x} - \mathbf{b}\|_2^2 + r(\mathbf{x}) \tag{3}$$

$$r(\mathbf{x}) = \mu\|\nabla\mathbf{x}\|_1 + (1-\mu)\|\nabla\mathbf{x}\|_2^2 + I_{[0,\infty)}(\mathbf{x}) \tag{4}$$

$$\text{model:} \quad \begin{aligned} f_1(\mathbf{v}) &= \|\mathbf{v} - \mathbf{b}\|_2^2, & \mathbf{K}_1 &= \mathbf{M}\mathbf{D} \\ f_2(\mathbf{v}) &= \mu\|\mathbf{v}\|_1, & \mathbf{K}_2 &= \nabla \\ f_3(\mathbf{v}) &= (1-\mu)\|\mathbf{v}\|_2^2, & \mathbf{K}_3 &= \nabla \\ f_4(\mathbf{v}) &= I_{[0,\infty)}(\mathbf{v}), & \mathbf{K}_4 &= \mathbf{I} \end{aligned} \tag{5}$$

Note that there are other ways to represent the problem in our standard form. For example, we could use:

$$f_1(\mathbf{v}) = \|\mathbf{M}\mathbf{v} - \mathbf{b}\|_2^2, \quad \mathbf{K}_1 = \mathbf{D}.$$

A key insight is that the choice of representation can drastically affect the performance of the solver algorithms. We take advantage of this fact and provide strategies to find an optimal reformulation.

The only assumption we make about the penalty functions $f_1, \ldots, f_I$ is that they provide a black box for evaluating the function's proximal operator. The proximal operator of a function f is defined as:

$$\mathbf{prox}_{\tau f}(\mathbf{v}) = \operatorname*{argmin}_{\mathbf{x}} \ \left(f(\mathbf{x}) + \frac{1}{2\tau}\|\mathbf{x} - \mathbf{v}\|_2^2\right),$$

where $\tau > 0$ and $\mathbf{v} \in \mathbb{R}^{m_i}$ [Parikh and Boyd 2013]. The proximal operator optimizes over the function in isolation, but incorporates a quadratic term that can be used to link the optimization with a broader algorithm. Many algorithms can be carried out using proximal operators that cannot be carried out using the traditional approach of interacting with functions by computing their gradients and Hessians [Parikh and Boyd 2013].

Similarly, the only assumption we make about each linear operator $\mathbf{K}_i$ is that it provides a black box for evaluating the forward operator $\mathbf{x} \to \mathbf{K}_i\mathbf{x}$ and the adjoint operator $\mathbf{z} \to \mathbf{K}_i^T\mathbf{z}$. This is a useful abstraction because many linear operators that arise in optimization problems from image processing are fast transforms, *i.e.*, they have methods for evaluating the forward and adjoint operator that are more efficient than standard multiplication by the operator represented as a dense or sparse matrix. Common fast transforms in image processing include the discrete Fourier transform (DFT), convolution, and wavelet transforms; see [Diamond and Boyd 2016a] for many more examples.

For simplicity, we assume that all linear operators are maps from a multidimensional real space $\mathbb{R}^{n_1\times\cdots\times n_k}$ to another multidimensional real space $\mathbb{R}^{m_1\times\cdots\times m_\ell}$. Complex-valued linear operators such as the DFT are represented as real valued operators using the standard embedding of a complex vector in $\mathbb{C}^{n_1\times\cdots\times n_k}$ as a real vector in $\mathbb{R}^{2n_1\times\cdots\times n_k}$.

We call algorithms that solve Problem (2) using only these black boxes proximal, matrix-free solvers. All solver algorithms in ProxImaL are proximal, matrix-free solvers. ProxImaL currently supports the Pock-Chambolle algorithm, ADMM, linearized ADMM, and half-quadratic splitting. See the supplement for detailed derivations showing that all of these methods fit into our framework from (2). These algorithms can solve Problem (2) when the functions $f_1, \ldots, f_I$ are convex.

Much state-of-the-art image optimization makes use of nonconvex penalty functions; however, in applications ranging from denoising and deconvolution to burst reconstruction and registration. Patch-based approaches and hard thresholding in particular have been very successful for image reconstruction problems [Krishnan and Fergus 2009; Danielyan et al. 2012; Heide et al. 2014].

Surprisingly, the same proximal, matrix-free solvers that work for convex problems yield good results for certain problems that include nonconvex penalty functions [Danielyan et al. 2012; Heide et al. 2014; Hallac et al. 2015]. There is often no guarantee that the algorithms will converge (see conditions in [Ochs et al. 2014] for exceptions). Furthermore, there is no guarantee that they find the optimal $\mathbf{x}$, but empirically for many problems with nonconvex penalties the algorithms do produce good results in a reasonable number of iterations.

`Problem(ProxFn, ...)`:	Defines a ProxImaL problem as an objective made up of the sum of a list of proxable functions.
`ProxFn(LinOp)`:	The base type of a proxable function, applied to a linear expression. Table 2 shows examples from the set of predefined proxable functions.
`LinOp(...)`:	The base type of a linear expression, each of which has zero or more linear expressions as children. Table 3 shows examples from the set of predefined proxable functions.
`Variable(w,h,...)`:	Defines a set of $w \times h \times \ldots$ unknowns as a single logical multi-dimensional array variable. A problem may use multiple logical variables of different size and shape, which together form its complete set of unknowns $\mathbf{x}$.

Table 1: *Core primitives in the ProxImaL language.*

We therefore allow the penalty functions $f_1, \ldots, f_I$ to be nonconvex, even though by doing so we sacrifice guarantees of optimality. For convex problems, we run the solver until convergence criteria are satisfied, but for nonconvex problems we follow the same approach as [Hallac et al. 2015] and return the iteration with the lowest objective value after a fixed number of iterations.

4 The ProxImaL Language

ProxImaL asks users to describe image optimization problems using a simple DSL, embedded in Python, which corresponds directly to the model in Problem (2). At the highest level, the user defines a `Problem` as a list of applications of proxable functions, or functions with a known proximal operator, instantiated as `ProxFn` objects. Each `ProxFn` term applies to a linear expression (`LinOp`), and each linear expression is an arbitrary sub-DAG of linear expressions, ultimately terminating in variable references. `Variable`s are defined as multidimensional arrays (e.g., $w \times h \times 3$ for a color image). A problem can use an arbitrary number of logical variables, each referenced arbitrarily within the linear expression of any of the proximal terms. Each logical variable refers to individual subcomponents of the vector of all stacked unknowns, *i.e.*, $\mathbf{x}$ from Problem (2). Variable references make up the leaves of the linear expression DAG, referring to individual components of the unknowns.

Operator overloading translates `alpha*expr` and `expr + expr` into the `LinOp`s `scale(alpha, expr)` and `sum(expr, expr)`, respectively. Here `alpha` is a scalar constant and `expr` may be any `ProxFn` term or linear expression, though `ProxFn` terms cannot be multiplied by negative constants.

For example, Problem (3) can be written in ProxImaL as:

```
        x = Variable(300, 300, 3)
data_term = sum_squares( subsample( conv(x, psf) ) - input )
grad_term =    mu  * norm1( grad(x) ) +
           (1-mu) * sum_squares( grad(x) )
objective = data_term + grad_term + nonneg(x)
        p = Problem( objective )
        x = Variable(300, 300, 3)
data_term = sum_squares( subsample( conv(x, psf) ) - input )
grad_term = mu  * norm1( grad(x) )
objective = data_term + grad_term
        p = Problem( objective )
```

Figure 2 shows how ProxImaL translates the above code into a DAG representation of the problem. The problem can be solved simply by calling `p.solve(solver="ADMM")`, where the solver keyword specifies what solver algorithm to use, in this case ADMM.

4.1 Proxable functions

ProxImaL provides a library of proxable functions that commonly occur in image optimization problems. Table 2 lists several examples

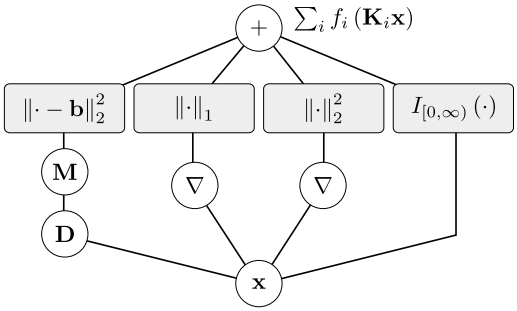


Figure 2: *The DAG representation of Problem* (3).

`sum_squares(e)`:	Defines the squared ℓ_2-norm $\|e\|_2^2$ for any linear expression `e`.
`norm1(e)`:	Defines an ℓ_1-norm $\|e\|_1$.
`poisson_norm(e, b)`:	Defines a maximum-likelihood denoiser that acts as a penalty function. Not a "formal" norm.
`patch_NLM(e, ...)`:	Patch prior for image self-similarity.
`group_norm1(e, dims)`:	Flattens the dimensions `dims` of `e` with an ℓ_2-norm, then computes an ℓ_1-norm of the result. Useful to describe sparse norms over vector-valued quantities.
`nonneg(e)`:	Defines a constraint that `e` is non-negative (0 cost if so, ∞ otherwise).

Table 2: *Example proxable functions provided by ProxImaL .*

from the library.

Every proxable function $f : \mathbb{R}^n \to \mathbb{R}$ defined in ProxImaL can be parametrized to express any function of the form:

$$g(\mathbf{x}) = \alpha f(\beta \mathbf{Q}\mathbf{x} - \mathbf{b}) + \langle \mathbf{c}, \mathbf{x} \rangle + \gamma \langle \mathbf{x}, \mathbf{x} \rangle,$$

where $\mathbf{x} \in \mathbb{R}^n$ is a variable, $\alpha > 0$, $\beta \in \mathbb{R}$, $\mathbf{b} \in \mathbb{R}^n$, $\mathbf{c} \in \mathbb{R}^n$ and $\gamma > 0$ are constants, and $\mathbf{Q} \in \mathbb{R}^{n \times n}$ is an orthogonal matrix.

The proximal operator of g can be evaluated using only the proximal operator of f. It is straightforward to show that:

$$\mathbf{prox}_{\tau g}(\mathbf{v}) = \mathbf{Q}^T \left(\mathbf{prox}_{\hat{\tau} f}(\hat{\mathbf{v}}) + \mathbf{b} \right) / \beta,$$

where $\hat{\tau} = \frac{\alpha \beta^2 \tau}{1 + 2\gamma\tau}$ and $\hat{\mathbf{v}} = \frac{\beta}{1+2\gamma\tau} \mathbf{Q}(\mathbf{v} - \tau \mathbf{c}) - \mathbf{b}$.

When f is separable, *i.e.*, $f(\mathbf{v}) = \sum_{i=1}^n f_i(\mathbf{v}_i)$ for proxable scalar functions $f_1, \ldots, f_n$, then we can replace β in the parametrization with a diagonal matrix $\Delta \in \mathbb{R}^{n \times n}$.

Proxable functions may also accept multidimensional inputs, *e.g.*, $f : \mathbb{R}^{n_1 \times \cdots \times n_k} \to \mathbb{R}$. In that case the parametrized form of the function is defined similarly, with $\mathbf{b}, \mathbf{c} \in \mathbb{R}^{n_1 \times \cdots \times n_k}$ and $\mathbf{Q}$ an orthogonal linear map.

In the ProxImaL language, each of these additional parameters is passed as an optional keyword argument to the proxable function constructor (e.g., `alpha=..., beta=...`). The ProxImaL compiler takes advantage of the parametrized form internally to rewrite optimization problems.

4.2 Linear operators

ProxImaL provides a library of linear operators that include standard operations like addition and multiplication by a constant, as well as common image processing operations. Table 3 lists several examples

`conv(e, k)`:	Convolves the subexpression `e` with the kernel `k`.
`subsample(e, steps)`:	Extracts every $\texttt{steps}_i$ pixel along axis `i`, starting with the pixel $\texttt{steps}_i$-1.
`mul_elemwise(weight, e)`:	Element-wise multiplication with a fixed constant weight array.
`scale(c,e)`:	Scale `e` by fixed constant scalar `c`.
`sum(e1,e2, ...)`:	Sums input expressions into a single linear expression.
`vstack(e1, e2, ...)`:	Vectorizes and stacks a list of input expressions into a single linear expression.
`grad(e)`:	Computes the gradients of `e`, by default across all n of its dimensions.
`warp(e, H)`:	Interprets `e` as a 2D image and warps it using the homography `H` with linear interpolation.
`mul_color(e, C)`:	Performs a blockwise 3×3 color transform using the color matrix `C`, or the predefined opponent (`opp`) and YUV (`yuv`) color spaces.
`x`:	Variable references make up the leaves of the linear expression DAG.

Table 3: *Example linear operators provided by ProxImaL .*

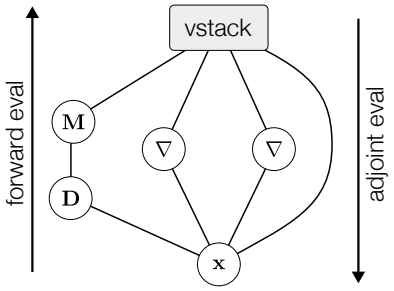


Figure 3: *The DAG for the stacked linear operators in Problem* (3).

from the library. Including linear operators for image processing operations like convolution and warp in the language, rather than having the user write these as multiplication by a dense or sparse matrix, is crucial to generating efficient solver implementations. The high-level descriptions of the linear operators can be exploited by a matrix-free solver to evaluate the operators efficiently, as discussed in [Diamond and Boyd 2015].

Compositions of linear operators are represented as expression DAGs. Figure 3 shows the DAG for the linear operators in Problem (3) stacked into a single operator K, as in Problem (2). The DAG structure makes it easy to evaluate the composition. We simply visit the nodes in topological order, reading the input to each node's linear operator from the node's incoming edges, applying the operator, and writing the outputs to the node's outgoing edges. The overall input for the composition is the input to the variable nodes, and the final output is the output of the root node.

For example, to evaluate the composition in Figure 3 on an input $\mathbf{x}$, we first evaluate the variable node by reading $\mathbf{x}$ as input and writing $\mathbf{x}$ to the node's outgoing edges. Next, we evaluate the $\mathbf{D}$ and ∇ nodes in any order. We can evaluate the $\mathbf{M}$ node any time after we evaluate the $\mathbf{D}$ node. We finish by evaluating the `vstack` node to get the final output.

We can evaluate the adjoint of the composition just as easily. We follow the same algorithm as for forward evaluation, but visit the nodes in reverse topological order (starting at the root and ending with the variable nodes), reading from the node's outgoing edges, evaluating the adjoint of the node's linear operator, and writing the result to the node's incoming edges.

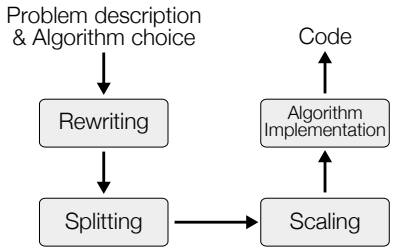


Figure 4: *The ProxImaL compiler pipeline.*

Even for the simple example in Figure 3, there are many possibilities for making the forward and adjoint evaluation algorithms more efficient. The $\mathbf{D}$ and ∇ nodes could be evaluated in parallel, and the graph could be rewritten so the ∇ operator is only evaluated once. See [Diamond and Boyd 2016a] for a detailed discussion of possible optimizations, which we are planning to add in our implementation.

4.3 Extensibility

ProxImaL also supports extending the set of proxable functions and linear operators beyond the built-ins provided. A linear operator extends the **`LinOp`** interface and must define `forward` and `adjoint` methods, which consume arrays of input values to produce arrays of output values. Linear operators may extend additional optional methods that provide information about how to invert the operator, as discussed in Section 6. A proxable function extends the **`ProxFn`** interface and must define the proxable operator `prox(tau, v)`.

5 Compiling Problems to Efficient Solvers

One of the main contributions in ProxImaL is a compiler that takes a problem specification and choice of algorithm and automatically generates an efficient solver implementation. In this section, we describe the stages of the compiler. Figure 4 shows an overview of the compiler pipeline. We use Problem (3) as a running example throughout our discussion.

5.1 Rewriting problems

The first stage of the compiler attempts to rewrite the optimization problem in a form better suited for the solver algorithm. The standard form in Problem (2) is not unique. Any given optimization problem written in the form of Problem (2) can be rewritten as many equivalent problems also in the same form. Our compiler considers two kinds of rewrites: absorbing linear operators into proxable functions and merging proxable functions.

Absorbing linear operators. Concretely, given a proxable function f composed with a composite linear operator $\mathbf{K}$ represented as an expression tree e, absorbing a linear operator means removing the root linear operator $\tilde{\mathbf{K}}$ of e and replacing f with the composition $f \circ \tilde{\mathbf{K}}$. Absorbing a linear operator is only possible when the root of the expression tree has exactly one child.

Whether absorbing a linear operator is a good idea depends on whether the composition $f \circ \tilde{\mathbf{K}}$ has an efficient proximal operator. For example, consider the proxable function $f_1(\mathbf{v}) = \|\mathbf{v} - \mathbf{b}\|_2^2$ and linear operator $\mathbf{K}_1 = \mathbf{MD}$ from Problem (3). We can absorb a linear operator by replacing f_1 with $\tilde{f} = f_1 \circ \mathbf{M}$ and $\mathbf{K}_1$ with $\tilde{\mathbf{K}} = \mathbf{D}$ Figure 5 shows the expression trees for $f_1 \circ \mathbf{K}_1$ and $\tilde{f} \circ \tilde{\mathbf{K}}$.

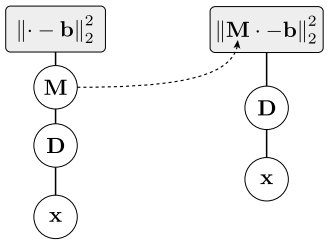


Figure 5: *Absorbing a linear operator.*

In this case, $\tilde{f}$ has the proximal operator

$$\mathbf{prox}_{\tau\tilde{f}}(\mathbf{v}) = \left(\frac{1}{2\tau}\mathbf{I} + \mathbf{M}^T\mathbf{M}\right)^{-1}(\mathbf{M}^T\mathbf{b} + \mathbf{v}/\sqrt{2\tau}),$$

which can be computed efficiently because $\mathbf{M}^T\mathbf{M}$ is diagonal.

We could absorb a linear operator again by replacing $\tilde{f}(\mathbf{v}) = \|\mathbf{M}\mathbf{v} - \mathbf{b}\|_2^2$ and $\tilde{\mathbf{K}} = \mathbf{D}$ with $\hat{f}(\mathbf{v}) = \|\mathbf{M}\mathbf{D}\mathbf{v} - \mathbf{b}\|_2^2$ and $\hat{\mathbf{K}} = \mathbf{I}$. In this case, the proximal operator:

$$\mathbf{prox}_{\tau\hat{f}}(\mathbf{v}) = \left(\frac{1}{2\tau}\mathbf{I} + \mathbf{M}^T\mathbf{D}^T\mathbf{D}\mathbf{M}\right)^{-1}(\mathbf{D}^T\mathbf{M}^T\mathbf{b} + \mathbf{v}/\sqrt{2\tau}),$$

is not as efficient. We could compute the proximal operator using an iterative method such as the conjugate gradient method (CG) [Hestenes and Stiefel 1952] or LSQR [Paige and Saunders 1982], which only interact with $\mathbf{D}$ and $\mathbf{M}$ by evaluating the linear operators and their adjoints. However, such iterative methods are more computationally expensive and less accurate than the methods for computing the proximal operator of f_1 or $\tilde{f}$.

Merging proxable functions. Merging proxable functions means replacing two proxable functions f_i and f_j that are composed with the same linear operator (*i.e.*, $\mathbf{K}_i = \mathbf{K}_j$) with a new function $g(\mathbf{v}) = f_i(\mathbf{v}) + f_j(\mathbf{v})$. In Problem (3), we can merge the proxable functions $f_2(\mathbf{v}) = \mu\|\mathbf{v}\|_1$ and $f_3(\mathbf{v}) = (1-\mu)\|\mathbf{v}\|_2^2$ because they are both composed with the linear operator $\mathbf{K}_2 = \mathbf{K}_3 = \nabla$. As with absorbing linear operators, merging proxable functions is only a good idea if the new function still has an efficient proximal operator. Note that our conditions for merging functions also include the common case when both functions are parametrized forms of the same function. We exploit here the general formulation of a proxable function from Section 4.1. For example, two functions $f_a(\mathbf{v}) = \mu\|\mathbf{v}\|_2$ and $f_b(\mathbf{v}) = \rho\|\mathbf{Q}\mathbf{v}\|_2^2$, where $\mu, \rho > 0$ and $\mathbf{Q}$ is an orthogonal matrix, turn out to be parametrizations of the ℓ_2-norm using the general definition. In addition, this example demonstrates that merging functions and absorbing operators work together symbiotically: the operator $\mathbf{Q}$ can be absorbed to make $\mathbf{K}_a = \mathbf{K}_b$. In general the parametrized form provides many opportunities to absorb linear operators and merge proxable functions into efficient compound proxable functions. In our example (3) the merged function $g(\mathbf{v}) = \mu\|\mathbf{v}\|_1 + (1-\mu)\|\mathbf{v}\|_2^2$ has an efficient proximal operator because it is a parametrized form of $\tilde{f}(\mathbf{v}) = \|\mathbf{v}\|_1$ discussed in Section 4.1.

Default choices. Absorbing linear operators and merging proxable functions generally simplifies the optimization problem and makes it easier to solve. At a minimum, absorbing linear operators makes multiplication by the overall linear operator $\mathbf{K}$ from Problem (2) more efficient.

Our compiler by default iterates over the proxable functions, and for each one repeatedly absorbs linear operators until doing so would substantially increase the cost of the proximal operator. The compiler then considers all pairs of proxable functions and greedily merges them whenever it can do so and still have an efficient proximal operator. In the context of Problem (3), our compiler would absorb the linear operator $\mathbf{M}$ but not the linear operator $\mathbf{D}$ and would merge $\alpha\|\nabla\mathbf{x}\|_1$ and $(1-\alpha)\|\nabla\mathbf{x}\|_2^2$.

5.2 Problem splitting

The second step in the compilation is problem splitting. The optimization algorithms in ProxImaL are operator splitting methods that solve problems in the standard form:

$$\begin{array}{ll}\text{minimize} & g(\mathbf{x}) + h(\mathbf{z}) \\ \text{subject to} & \mathbf{K}\mathbf{x} = \mathbf{z},\end{array} \tag{6}$$

where $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{z} \in \mathbb{R}^m$ are variables, $\mathbf{K} \in \mathbb{R}^{m\times n}$ is a known linear operator, and $g : \mathbb{R}^n \to \mathbb{R}$ and $h : \mathbb{R}^m \to \mathbb{R}$ have known proximal operators. Some of the algorithms in fact allow a more general standard form with a linear constraint of the form $\mathbf{K}\mathbf{x} + \mathbf{B}\mathbf{z} = \mathbf{c}$, but our compiler does not currently take advantage of this, though future, more sophisticated versions of the compiler will.

The compiler expresses g and h as:

$$g(\mathbf{x}) = \sum_{f_i\in\Omega} f_i(\mathbf{x}), \qquad h(\mathbf{z}) = \sum_{f_i\in\Psi} f_i(\mathbf{z}),$$

where Ω and Ψ are a partition of the set of functions $\{f_1, \ldots, f_I\}$ from Problem (2). Problem splitting means choosing Ω and Ψ.

The splitting $\Psi = \{f_1, \ldots, f_I\}$ is always valid for the algorithms in ProxImaL, so the compiler always has at least one choice of splitting. The functions $f_i \in \Omega$ must have the identity as their linear operator, *i.e.*, $\mathbf{K}_i = \mathbf{I}$. This may seem unduly restrictive, but recall that the compiler tries to absorb linear operators, which means that penalty functions that began with complex $\mathbf{K}_i$ may end up with $\mathbf{K}_i = \mathbf{I}$ by the problem splitting stage. Just because a function has the identity as its linear operator does not mean including it in Ω is a good idea. Each algorithm has its own logic for choosing when to include a function in Ω.

5.3 Problem scaling

The third stage of the compiler is problem scaling. Problem scaling replaces Problem (6) with the equivalent problem:

$$\begin{array}{ll}\text{minimize} & g(\mathbf{A}\hat{\mathbf{x}}) + h(\mathbf{B}^{-1}\hat{\mathbf{z}}) \\ \text{subject to} & \mathbf{B}\mathbf{K}\mathbf{A}\hat{\mathbf{x}} = \hat{\mathbf{z}},\end{array} \tag{7}$$

where $\mathbf{A} \in \mathbb{R}^{n\times n}$ and $\mathbf{B} \in \mathbb{R}^{m\times m}$ are invertible and the new variables $\hat{\mathbf{x}} \in \mathbb{R}^n$ and $\hat{\mathbf{z}} \in \mathbb{R}^m$ are related to $\mathbf{x}$ and $\mathbf{z}$ in Problem (6) via:

$$\mathbf{x} = \mathbf{A}\hat{\mathbf{x}}, \qquad \mathbf{z} = \mathbf{B}^{-1}\hat{\mathbf{z}}.$$

Problem scaling can substantially affect the number of iterations our solver algorithms take to converge [Pock and Chambolle 2011; Giselsson and Boyd 2014]. Our compiler by default sets:

$$\mathbf{A} = \frac{1}{\sqrt{\|\mathbf{K}\|_2}}\mathbf{I}, \qquad \mathbf{B} = \frac{1}{\sqrt{\|\mathbf{K}\|_2}}\mathbf{I},$$

where $\|\mathbf{K}\|_2$ is the spectral norm, or maximum singular value of $\mathbf{K}$. We use the implicitly restarted Arnoldi method to compute $\|\mathbf{K}\|_2$ using only multiplication by $\mathbf{K}$ and $\mathbf{K}^T$ [Lehoucq and Sorensen 1996].

5.4 Algorithm implementation

The final stage of our compiler is generating (or calling) an actual solver algorithm. ProxImaL currently provides the Pock-Chambolle algorithm, ADMM, linearized ADMM, and half-quadratic splitting. In this subsection, we show the algorithm implementations for Pock-Chambolle and ADMM as pseudo-code and how our compiler chooses default problem splittings and hyper-parameters. Please find an extended discussion of how the other algorithms fit into our framework from Problem (2) in the supplement.

Pock-Chambolle implementation. The pseudo-code for the Pock-Chambolle algorithm is given in Algorithm 1. Our compiler uses the default hyper-parameters $\tau = \sigma = 1/\|\mathbf{K}\|_2$, $\theta = 1$, $\mathbf{x}^0 = 0$, and $\mathbf{z}^0 = 0$. With the default problem scaling, we have $\|\mathbf{K}\|_2 = 1$.

Our compiler only allows at most one penalty function f_i to be included in Ω, and the operator $\mathbf{K}_i$ must be the identity. The restriction on Ω ensures that the algorithm can be carried out using known proximal operators. The compiler's default problem splitting is to include one penalty function in Ω whenever possible. For the example Problem (3), the penalty function $f_4(\mathbf{v}) = I_{[0,\infty)}(\mathbf{v})$ would be included in Ω and all other penalty functions would be in Ψ.

Algorithm 1 Pock-Chambolle to solve Problem (2)

1: Initialization: $\sigma\tau\|\mathbf{K}\|_2^2 < 1$, $\theta \in [0,1]$, $(\mathbf{x}^0, \mathbf{z}^0)$, $\bar{\mathbf{x}}^0 = \mathbf{x}^0$.
2: **for** $k = 1$ to V **do**
3: $\quad \mathbf{z}_j^{k+1/2} = \mathbf{z}_j^k + \sigma \mathbf{K}_j \bar{\mathbf{x}}^k \quad \forall j \in \Psi$
4: $\quad \mathbf{z}_j^{k+1} = \mathbf{z}_j^{k+1/2} - \sigma \mathbf{prox}_{f_j/\sigma}\left(\mathbf{z}_j^{k+1/2}/\sigma\right) \quad \forall j \in \Psi$
5: $\quad$ **if** $\Omega = \{f_i\}$ **then**
6: $\quad\quad \mathbf{x}^{k+1} = \mathbf{prox}_{\tau f_i}\left(\mathbf{x}^k - \tau \mathbf{K}^T \mathbf{z}^{k+1}\right)$
7: $\quad$ **else**
8: $\quad\quad \mathbf{x}^{k+1} = \mathbf{x}^k - \tau \mathbf{K}^T \mathbf{z}^{k+1}$
9: $\quad$ **end if**
10: $\quad \bar{\mathbf{x}}^{k+1} = \mathbf{x}^{k+1} + \theta\left(\mathbf{x}^{k+1} - \mathbf{x}^k\right)$
11: **end for**

Line 4 uses Moreau's Identity [Moreau 1965]. Note that the Pock-Chambolle algorithm and linearized ADMM from the supplement are equivalent to an augmented/preconditioned version of the ADMM method [Chambolle and Pock 2011; Zhu 2015], described next.

ADMM implementation. The pseudo-code for ADMM is given in Algorithm 2. Our compiler uses the default hyper-parameters $\rho = 1$, $\alpha = 1$, $\mathbf{x}^0 = 0$, $\mathbf{z}^0 = 0$, and $\lambda^0 = 0$. The choice of problem splitting in ADMM is more complex than in Pock-Chambolle, since computing step 3 requires more than the proximal operators of the $f_i \in \Omega$. Our compiler only allows quadratic functions to be included in Ω because then step 3 reduces to solving a least squares problem.

Algorithm 2 ADMM to solve Problem (2)

1: Initialization: $\rho > 0$, $\alpha \in (0,2)$, $(\mathbf{x}^0, \mathbf{z}^0, \lambda^0)$.
2: **for** $k = 1$ to V **do**
3: $\quad \mathbf{x}^{k+1} = \underset{\mathbf{x}}{\operatorname{argmin}} \ \sum_{i\in\Omega} f_i(\mathbf{x}) + \sum_{j\in\Psi} (\rho/2)\|\mathbf{K}_j\mathbf{x} - \mathbf{z}_j^k + \lambda_j^k\|_2^2$
4: $\quad \mathbf{z}_j^{k+1} = \mathbf{prox}_{\frac{f_j}{\rho}}(\mathbf{K}_j(\alpha\mathbf{x}_j^{k+1} + (1-\alpha)\mathbf{x}_j^k) + \lambda_j^k) \quad \forall j \in \Psi$
5: $\quad \lambda_j^{k+1} = \lambda_j^k + \mathbf{K}_j\left(\alpha\mathbf{x}_j^{k+1} + (1-\alpha)\mathbf{x}_j^k\right) - \mathbf{z}_j^{k+1} \quad \forall j \in \Psi$
6: **end for**

We can use iterative methods such as CG and LSQR to solve the least squares problem using only multiplication by the linear operators and their adjoints. We use several tricks to accelerate these iterative methods. First, we initialize the iterative methods with the previous solution $\mathbf{x}^k$. Second, we initially solve the least squares problem with low accuracy and increase the required accuracy each iteration. This approach keeps the number of linear operator evaluations in the iterative methods relatively constant across iterations, since the previous solution $\mathbf{x}^k$ is increasingly close to the next solution $\mathbf{x}^{k+1}$ [O'Donoghue et al. 2015].

Direct methods. Often the least squares problem can be solved using a simple direct method, which makes the ADMM implementation much faster and more reliable. For example, recall that after the rewriting stage our example problem 3 has the form:

$$\mathbf{x}_{\text{opt}} = \underset{\mathbf{x}}{\operatorname{argmin}} \ \|\mathbf{M}\mathbf{D}\mathbf{x} - \mathbf{b}\|_2^2 + r(\mathbf{x}) \tag{8}$$

$$r(\mathbf{x}) = \mu\|\nabla\mathbf{x}\|_1 + (1-\mu)\|\nabla\mathbf{x}\|_2^2 + I_{[0,\infty)}(\mathbf{x}) \tag{9}$$

$$\text{model:} \quad \begin{aligned} f_1(\mathbf{v}) &= \|\mathbf{M}\mathbf{v} - \mathbf{b}\|_2^2, & \mathbf{K}_1 &= \mathbf{D} \\ f_2(\mathbf{v}) &= \mu\|\mathbf{v}\|_1 + (1-\mu)\|\mathbf{v}\|_2^2, & \mathbf{K}_2 &= \nabla \\ f_3(\mathbf{v}) &= I_{[0,\infty)}(\mathbf{v}), & \mathbf{K}_3 &= \mathbf{I} \end{aligned} \tag{10}$$

If we choose the problem splitting:

$$\Omega = \emptyset, \qquad \Psi = \{f_1, f_2, f_3\},$$

then step 3 of ADMM is given by:

$$\begin{aligned} \mathbf{x}^{k+1} &= \underset{\mathbf{x}}{\operatorname{argmin}} \left\| \begin{bmatrix} \mathbf{D} \\ \nabla \\ \mathbf{I} \end{bmatrix} \mathbf{x} - \mathbf{z}^k + \lambda^k \right\|_2^2 \\ &= (\mathbf{D}^T\mathbf{D} + \nabla^T\nabla + \mathbf{I})^{-1}(\mathbf{D}^T + \nabla^T + \mathbf{I})(\mathbf{z}^k - \lambda^k). \end{aligned}$$

The linear operators $\mathbf{D}$, ∇, and $\mathbf{I}$ are convolutions and thus diagonal in the frequency domain. Note that arbitrary boundary conditions can be supported by padding and masking the observations [Almeida and Figueiredo 2013]. Given the convolutional operations, we can compute a diagonal matrix $\Delta \in \mathbb{C}^{n\times n}$ for which:

$$(\mathbf{D}^T\mathbf{D} + \nabla^T\nabla + \mathbf{I})^{-1} = \mathcal{F}^{-1}\Delta\mathcal{F},$$

where $\mathcal{F} \in \mathbb{C}^{n\times n}$ is the DFT matrix. The upshot is that we can solve the least squares problem exactly in $O(n \log n)$ operations using the FFT.

Our compiler includes a system for automatically detecting when a linear operator $\mathbf{K}$ has a Gram matrix $\mathbf{K}^T\mathbf{K}$ that is diagonal in the spatial or frequency domain and obtaining the Gram matrix's diagonal representation; see Section 6 for details. This allows the compiler to automatically exploit the fast direct method above for solving the least squares problem.

Default splitting. The compiler uses the following procedure to choose the problem splitting for ADMM. We consider each subset S of quadratic penalty functions in the problem. We check whether for $\Omega = S$ the least squares problem in step 3 has a Gram matrix that is diagonal in the spatial or frequency domain, in which case we can solve the problem with a fast direct method. We then set Ω to be a maximal cardinality subset for which the Gram matrix is diagonal. If there is no subset for which the Gram matrix is diagonal, we include all the quadratic penalty functions in Ω and solve the least squares problem using CG.

One might be concerned about the runtime of this method since the number of subsets is exponential in the number of quadratic penalty functions. However, hardly any problems will have more than a few

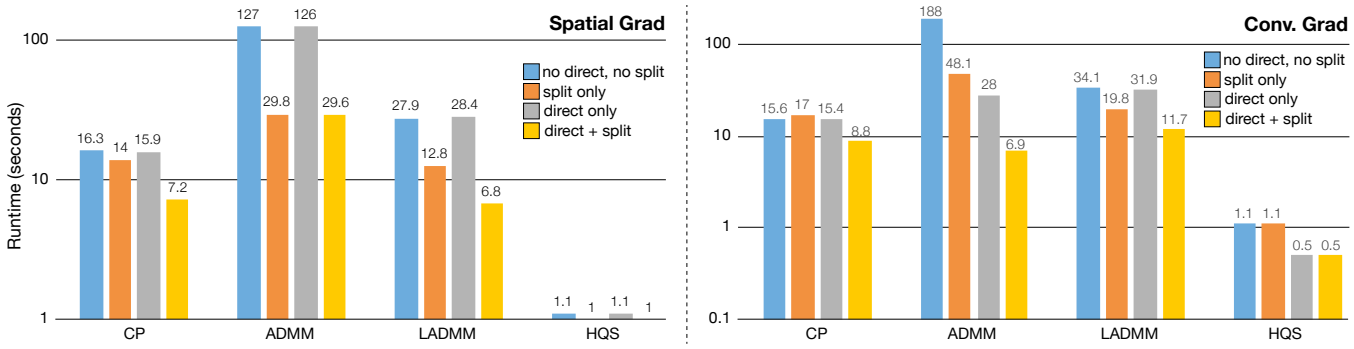


Figure 6: *Runtime (in seconds) of a TV-regularized deconvolution problem in ProxImaL. When the* direct *parameter is on, the ProxImaL compiler automatically replaces CG with a fast direct method. The* split *option further indicates whether the compiler's intelligent rewriting and splitting are used. We evaluate four different algorithms that are implemented in ProxImaL. Note that independent of the algorithm choice, our compiler choice improves runtime. We evaluate two implementations of the TV prior: a finite differences implementation in the spatial domain (left) and a convolutional implementation via Fourier multiplication (right).*

(often one or two) quadratic penalty functions, and our method for checking whether the Gram matrix is diagonal is linear time in the number of linear operators.

For the rewriting of the example Problem (3) chosen by our compiler, given in Equation (10), our method would choose $\Omega = \emptyset$ and solve the least squares problem directly. For the original, naive formulation in Equation (5) our method would choose $\Omega = \{f_1, f_3\}$ and use CG to solve the least squares problem. The choices made by our compiler can thus dramatically improve the performance of the ADMM implementation over a naive approach that always uses an iterative method or does not consider rewrites. Problem 3 may seem contrived, but solving the least squares problem with a direct method when possible is necessary for our system to be competitive with specialized solvers, and in fact was sufficient for state-of-the-art performance on a demosaicking problem; see Section 8.1 for details.

5.5 Manual options

We allow the user to override any of the default compiler choices. The user can manually rewrite the problem by changing the problem definition, choosing the problem splitting and least squares solver, and setting the problem scaling and algorithm hyper-parameters. An important example of where manual input is helpful is specifying the starting iterates in the solver algorithm (*e.g.*, $(\mathbf{x}^0, \mathbf{z}^0)$ for Pock-Chambolle). Starting from a good iterate can dramatically reduce the number of solver iterations needed and the quality of the solution for nonconvex problems [Heide et al. 2013].

5.6 Empirical validation

Figure 6 shows that the choices made by our compiler can dramatically reduce the solver runtime. We list the average runtime for a deconvolution problem regularized by total variation (TV) for each of the four algorithms in ProxImaL: Pock-Chambolle (CP), ADMM, linearized ADMM (LADMM), and half-quadratic splitting (HQS). Note that HQS does not have convergence guarantees for arbitrary convex problems fitting into Problem (2), but only for a subset [Robini and Zhu 2015], however, including common problems such as the considered TV-regularized deconvolution. Each of the four algorithm is tested with and without the compiler's intelligent rewritings and splitting, as well as with and without the compiler automatically replacing CG with a fast direct method by detecting diagonal matrices. Further, we evaluate implementations of the TV in the spatial domain (Figure 6 left) and as a multiplication in the Fourier domain (Figure 6 right). The latter case assumes circular boundary conditions but can sometimes be faster. Assuming circular boundary conditions slightly modifies the optimization problem being solved, but we evaluate all our results using the original objective, with non-circular boundary conditions. All algorithms are run until convergence to the exact same objective function value. The results show that the choices made by the compiler for all algorithms improve the runtime, often substantially. Please note that the absolute runtime depends in general on the algorithm-specific implementation and parameters, including effects of potentially adaptive parameter schedules as for example discussed in [Fougner and Boyd 2015]. However, Figure 6 shows that independent of the algorithm choice, the choice made by our compiler significantly improves runtime.

6 Analysis of Linear Systems

In this section, we explain our method for automatically detecting when a least squares problem:

$$\text{minimize} \quad \|\mathbf{K}\mathbf{x} - \mathbf{b}\|_2^2, \tag{11}$$

can be solved using a fast direct method. Specifically, our system detects when the Gram matrix $\mathbf{K}^T\mathbf{K}$ is diagonal in the spatial or frequency domain and computes its diagonal representation.

Diagonal $\mathbf{K}^T\mathbf{K}$. We first explain how we determine when $\mathbf{K}^T\mathbf{K}$ is diagonal. We use the subroutines `is_diag` and `is_gram_diag`. These subroutines take as argument an expression DAG e of linear operators. The subroutine `is_diag` returns true if the composite linear operator $\mathbf{K}_e$ defined by e is diagonal, and false otherwise. The subroutine `is_gram_diag` returns true if $\mathbf{K}_e^T\mathbf{K}_e$ is diagonal, which is always true if $\mathbf{K}_e$ is diagonal, but is also true for non-diagonal $\mathbf{K}_e$ such as:

$$\mathbf{K}_e = \begin{bmatrix} I \\ 0 \end{bmatrix},$$

which represents zero-padding.

The output of the subroutines depends on the type of e's root node n and the results of applying the subroutines to the subDAGs rooted at n's children. Table 4 gives examples of the logic for various linear operator types. Recall that leaves of an expression DAG are always variable nodes, so the behavior for variables is the base case.

We simply apply `is_gram_diag` to the expression DAG representing $\mathbf{K}$ in Problem (11) to determine whether $\mathbf{K}^T\mathbf{K}$ is diagonal. To find the value of $\mathbf{K}^T\mathbf{K}$, we use the `get_diag` subroutine, which takes as argument a variable node v and an expression DAG e for which

type	*subroutine logic*
x (variable)	`is_diag`: Always true.
	`is_gram_diag`: Always true.
`mul_elemwise`	`is_diag`: True if `is_diag` is true for children.
	`is_gram_diag`: True if `is_diag` is true for children.
`subsample`	`is_diag`: Always false.
	`is_gram_diag`: True if `is_diag` is true for children.
`sum`	`is_diag`: True if `is_diag` is true for children.
	`is_gram_diag`: True if `is_diag` is true for children.
`vstack`	`is_diag`: Always false.
	`is_gram_diag`: True if `is_gram_diag` is true for children.

Table 4: *Logic for the* `is_diag` *and* `is_gram_diag` *subroutines.*

`is_gram_diag`(e) returns true. The subroutine returns the diagonal of $(\mathbf{K}_{e,v}^T\mathbf{K}_{e,v})^{1/2}$, where $\mathbf{K}_{e,v}$ is the linear function of the variable v defined by e. The full diagonal of $(\mathbf{K}_e^T\mathbf{K}_e)^{1/2}$ is obtained by evaluating `get_diag`(v, e) for all variable nodes v in e and stacking the results into a single vector.

The output of `get_diag` depends on e's root node n and the results of applying `get_diag` to the subDAGs rooted at n's children. The parent call passes its variable node argument to the recursive calls. The output of `get_diag`(v, e) when e is a variable node is a vector of ones if $e = v$ and a vector of zeros otherwise. For other linear operators the logic of `get_diag` is straightforward and does not depend on the variable node argument.

Diagonal $\mathbf{K}^T\mathbf{K}$ in the frequency domain. We take an analogous approach to determine whether $\mathbf{K}^T\mathbf{K}$ is diagonal in the frequency domain and, if so, get its diagonal representation. We use the subroutines `is_fdiag`, `is_gram_fdiag`, and `get_fdiag`, which are the same as `is_diag`, `is_gram_diag`, and `get_diag`, respectively, except defined for the frequency domain.

Extensions. Our approach easily generalizes to linear operators $\mathbf{K}$ for which $\mathbf{K}^T\mathbf{K}$ is block diagonal in the spatial or frequency domain. For example, this includes color transformations in the image formation model. We would simply extend `is_diag`, `is_gram_diag`, and their counterparts for the frequency domain to track the dimensions of the blocks on the diagonal. When the blocks are small, it would be worthwhile to factor the blocks and find the solution $x^\star$ to Problem (11) directly via $x^\star = \mathbf{K}^\dagger\mathbf{b}$. A further extension we will consider is determining whether $\mathbf{K}^T\mathbf{K}$ is banded, or has overlapping diagonal blocks.

7 Implementation

We implemented ProxImaL as a Python library with syntax inspired by CVXPY [Diamond and Boyd 2016b]. Each solver algorithm is implemented as a Python driver that schedules and evaluates the series of proximal and linear operators which make up the objective, as split and scheduled by our compiler logic. Our implementation only evaluates one operator at a time; a more sophisticated approach would evaluate multiple operators simultaneously. The proximal and linear operators are evaluated using a combination of NumPy and Halide-generated parallel and vectorized x86 code. We are planning to extend our framework to compile into Halide-generated GPU code.

We applied traditional Halide scheduling techniques to optimize the proximal and linear operators and other kernels. Most are simply parallelized, vectorized, unrolled, and occasionally with loops reordered. A few (particularly those involving `warp`) also block and fuse one or two stages for locality. We paid particular attention to the common ℓ_2-norm and dot product, which we parallelized using two-phase reductions to expose both vector and multicore parallelism in the main phase. This improved performance by at least an order of magnitude over a basic serial implementation. The current implementation left some performance on the floor by conservatively generalizing all kernels to accept any input with either row- or column-major order for compatibility with arbitrary BLAS and Fortran code in NumPy. A number of kernels also still use simple NumPy implementations, without any Halide-generated code. Table 5 gives an overview of timings for several linear and proximal operators for NumPy and Halide implementations. Above all, there is significant opportunity to schedule and fuse the resulting pipelines *across* operators for each generated solver, thereby substantially improving on our initial implementation, which schedules each operator separately.

	ℓ_2-norm	dot product	subsample	subsample*	grad	grad*	convolution
Halide	41.6	15.6	72.6	72.6	94.8	237.4	121.4
NumPy	245.8	96.6	356.0	356.0	1188.0	713.1	7790.9

	convolution*	warp	warp*	norm1	group norm1	Poisson prox	FFT inversion
Halide	121.4	153.1	367.8	27.1	67.8	44.7	9.4
NumPy	7790.9	457.6	474.4	201.8	1036.6	265.2	23.4

Table 5: *Runtimes (in ms) for linear operators and some of the proximal functions implemented with NumPy and Halide. subsample*, grad*, etc., are the adjoint operators.*

8 Evaluation

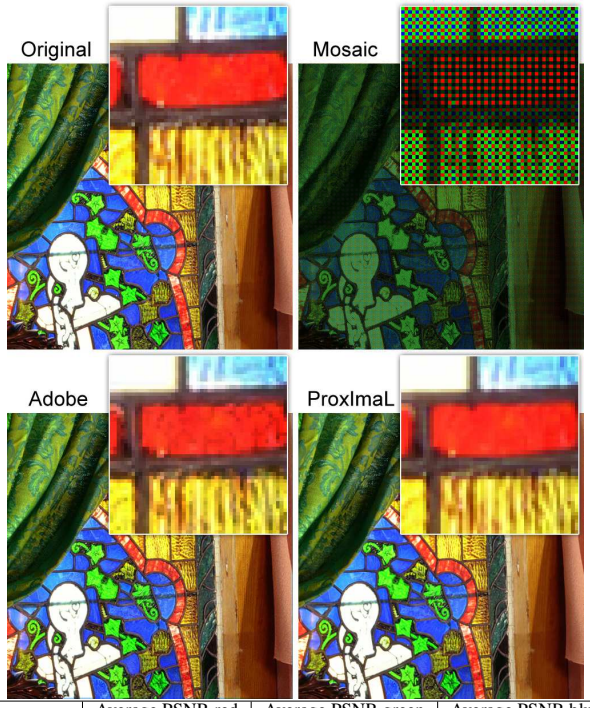
In this section, we evaluate ProxImaL for a range of inverse problems in imaging. In particular, we show that ProxImaL exploits problem-specific structure to improve quality and intuitiveness of what is considered the state-of-the-art image processing pipeline (FlexISP [Heide et al. 2014]). We also show state-of-the-art results for burst denoising and deconvolution in the presence of Poisson-distributed shot noise, which makes low-light photography so difficult. Lastly, we show applications to nonlinear image formations for the example of phase retrieval, which is an important problem in scientific imaging. For all applications shown in this section, ProxImaL allows for a very compact representation as shown in Table 6.

	Reference	ProxImaL
Burst	1020 (FlexISP)	6
Demosaicking	1020 (FlexISP)	6
IHdr	1020 (FlexISP)	6
Phase retrieval	300 (Matlab)	6
Poisson deconvolution	510 (Matlab)	6
ℓ_2 deconvolution	360 (Krishnan[2009])	6

Table 6: *Lines of code comparisons: We compare high-level (Matlab) code of reference methods with splitting done by hand to ProxImaL. For all example applications shown in this section, problems can be expressed in a very compact way using ProxImaL. Different splitting approaches that require large restructuring in the method can be expressed with just a few changed lines of ProxImaL code.*

8.1 Replacing Fixed-function ISPs

The core components of any image processing pipeline (ISP) include demosaicking, denoising, and deconvolution of a captured RAW image. Conventionally, ISPs are implemented as fixed-function blocks in the on-board signal processing chips on a camera. Recently, Heide et al. [2014] introduced FlexISP as the state-of-the-art ISP via image optimization. Their key insight was that formal optimization methods allow for all of the ISP problems to be solved very efficiently while allowing advanced natural image priors, such as self-similarity, to be incorporated. ProxImaL also builds on this idea, but provides a domain-specific language rather than a solver library. Working



	Average PSNR red	Average PSNR green	Average PSNR blue
FlexISP	38.50	41.84	36.50
ProxImaL	**38.54**	**41.90**	**36.53**

Table 7: *Demosaicking. A target color image (top left) is sampled by a monochrome sensor with a Bayer pattern (color-coded, top right). Commercial implementations, such as Adobe Camera RAW (bottom left), have been shown to be outperformed by image optimization in prior work (FlexISP). From only a few lines of code, ProxImaL automatically compiles the optimization routines for a solution that is qualitatively similar to the FlexISP approach (bottom right), but averaged over 12 test images quantitatively better (table and supplement) and also significantly faster (see text).*[1]

with solvers directly provides one of two choices: either one uses a generic solver, thereby giving up specificity that may be exploited, or one writes a dedicated solver for each problem, which requires a lot of time and parameter tuning. ProxImaL allows for a problem-specific solver to be automatically generated and makes it very easy to prototype new solvers or experiment with advanced image priors.

For example, consider the problem of demosaicking. Most digital sensors are inherently sensitive to all wavelengths throughout the visible range. Color filter arrays, such as the popular Bayer pattern, sample a color image via spatial multiplexing. The color reconstruction or *demosaicking* problem faces several challenges: subsampling of the channels, sensor noise, and optical blur by the point spread function (PSF) of the camera lens. Many commercially-available solutions exist, but none of them is without artifacts. A detailed comparison between many of them can be found in Heide et al. [2014]. The ProxImaL code for this problem is as simple as:

```
              x = Variable(300, 300, 3)
      data_term = sum_squares( subsample(x, bayer) - input )
patch_similarity = patch_BM3D( tonemap(x) )
   grad_sparsity = norm1( grad(x) )
       objective = data_term + patch_similarity + grad_sparsity
              p = Problem( objective )
```

ProxImaL automatically compiles this expression into the optimization routines required to solve this problem and it also detects the diagonal matrix structure of the subsampling operator, which can be inverted in closed form. As illustrated in Table 7 and in the supplemental material, a few lines of ProxImaL code are sufficient to

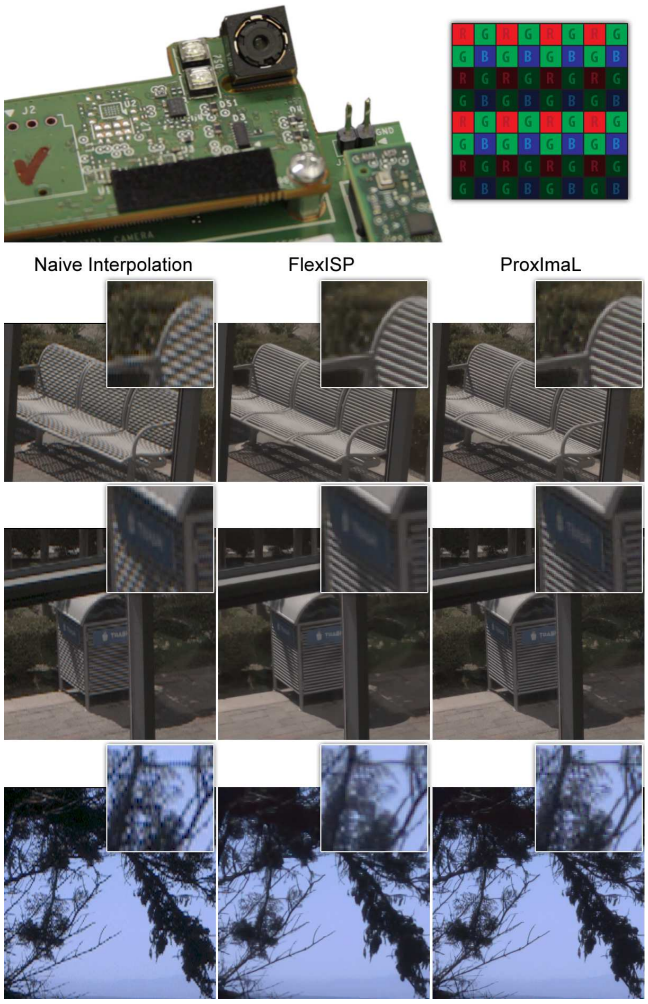


Figure 7: *High dynamic range from a single RAW image. The sensor captures interlaced exposures via a coded rolling shutter (top right), often found in moderns sensors (e.g., Aptina AR1331CP and Sony IMX135). FlexISP demonstrated state-of-the-art reconstructions for these types of problems, but ProxImaL further improves recovered image sharpness, runtime, and convenience.*[2]

generate the state-of-the-art ISP. Due to the fact that FlexISP or any generic solver is oblivious to problem-specific structure, we achieve faster runtimes because ProxImaL detects the closed-form solution of the subsampling operator and generates the most efficient solver. This makes the ProxImaL code faster and also slightly better, on average, than using a proximal operator based on iterative conjugate gradient (e.g., FlexISP). Other than the closed-form inverse of the subsampling step, both approaches are equivalent. We received the Matlab implementation of FlexISP from the authors.

The runtime of our splitting-based demosaicking is, for the most part, determined by two steps: the inversion of the subsampling and blur operators; and the BM3D denoiser. We improve upon the inversion step, which was previously implemented with the conjugate gradient method and takes 0.2 seconds per iteration for a 512×512 image and 9 seconds per iteration for a 16 megapixel image on our test computer. The closed-form inverse discovered by ProxImaL basically makes this step "free" and directly finds the optimal solution without any iterations. For the BM3D step, we use a Matlab implementation of BM3D since the code is closed source, which takes about 4 seconds per iteration for a 512×512 image; we

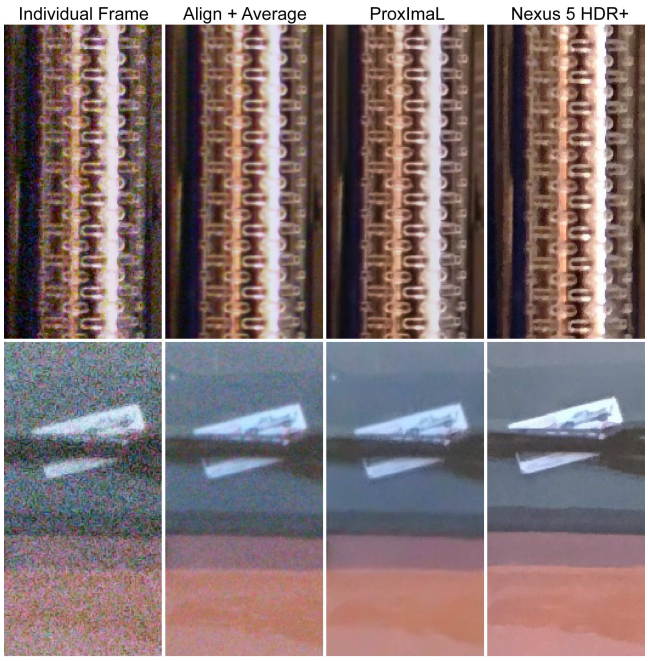


Figure 8: *Burst denoising and demosaicking for images captured with a Nexus 5 cellphone camera. Averaging and aligning several frames mitigates noise, but does not achieve optimal results. We combine automatic image alignment, demosaicking, and burst denoising into a single inverse problem that shows similar or better performance than the Nexus 5's HDR+ application.*

use 15 iterations in total. Tsai et al. [2014] demonstrated speedups of up to 1000× using GPU-optimized implementations.

Another common problem for image processing pipelines is the limited dynamic range offered by digital sensors. High dynamic range is often achieved using burst photography [Debevec and Malik 1997], but single-image approaches have also become very popular. For example, Gu et al. [2010] introduced the concept of coded rolling shutter, where different rows of the sensor image are read out with different exposure times or ISO gain settings. A common challenge for image reconstruction from coded rolling shutter images is missing image data from completely saturated regions. Combined with sensor noise, demosaicking, and PSF deconvolution, this inpainting problem often results in degraded image resolutions. We show in Figure 7 that ProxImaL achieves slightly sharper reconstructions than FlexISP while providing similar performance benefits as for the deconvolution.

8.2 Burst Denoising

High-quality photography in low-light conditions is one of the most challenging problems in computational photography due to the observed noise. The naive approach to mitigating image noise is to capture a stack of noisy images, align them, and then average them. Unfortunately, this simple approach does not result in optimal image quality, as seen in Figure 8. Using an additional non-local means (NLM) prior on the reconstructed image, one would formulate the burst denoising problem in ProxImaL as:

```
              x = Variable(300, 300, 3)
      data_term = sum_squares( vstack( warp(x, H_1),
                                       ...,
                                       warp(x, H_n) ) - input )
patch_similarity = patch_NLM( tonemap(x) )
      objective = data_term + patch_similarity
              p = Problem( objective )
```

Figure 9: *Comparison of image priors for burst denoising & demosaicking. For this simulation, a non-local means (NLM) prior outperforms the BM3D prior. The approximate NLM implemented in OpenCV is faster than, but not quite as good as, a full implementation.*[3]

In addition to the denoiser and the NLM prior, we include the image alignment as part of the optimization routine via the `warp` operator. This operator makes ProxImaL ideally suited for a range of burst photography applications—we simply specify that the global image alignment is part of the problem and the homography H is automatically estimated using [Evangelidis and Psarakis 2008]. Feature-based homography estimation also produced results of equal quality.

Figure 8 shows results for several scenes captured with the Nexus 5 rear-facing camera in low-light conditions. Compared to the reference output computed by the Android HDR+ app, our results are slightly sharper and less blocky. Improved image quality compared to simple image averaging in our implementation comes mostly from the NLM prior. Note that we do not tone map the image, which may be required for optimal color reproduction.

As discussed before, ProxImaL makes it very easy to evaluate different image priors for a particular reconstruction problem. For example, instead of NLM we can use BM3D, total variation, or other priors. Figure 9 evaluates several different priors for burst denoising in simulation. We compare BM3D, the fast approximation of NLM implemented by the OpenCV library, and a full NLM implementation. The latter is the slowest but also the best prior for this particular image. Some of the insights made in the presented experiments include the fact that different priors lead to vastly different results and implementing all of them separately takes a lot of time and effort. ProxImaL makes it very easy to evaluate all of them or combinations of them and it allows different priors to be combined with different solvers. We believe that this flexibility is one of the strongest benefits of a domain-specific language like ProxImaL.

8.3 Poisson Deconvolution

In low-light conditions, photographs are often not only noisy but also blurry. Especially in cellphone cameras, blur due to hand motion is almost unavoidable. In addition, the noise in images captured in low-light conditions are often dominated by Poisson-distributed shot noise. Instead of using a conventional ℓ_2-norm for the data fidelity term, the error metric should consider the Poisson-distributed nature of the noise. This can be done using the proximal operator implementing a maximum likelihood solution for Poisson noise, as discussed in the supplement and by Dupe et al. [2011]. With this proximal operator in hand, the deconvolution problem can additionally benefit from advanced image priors, such as self-similarity.

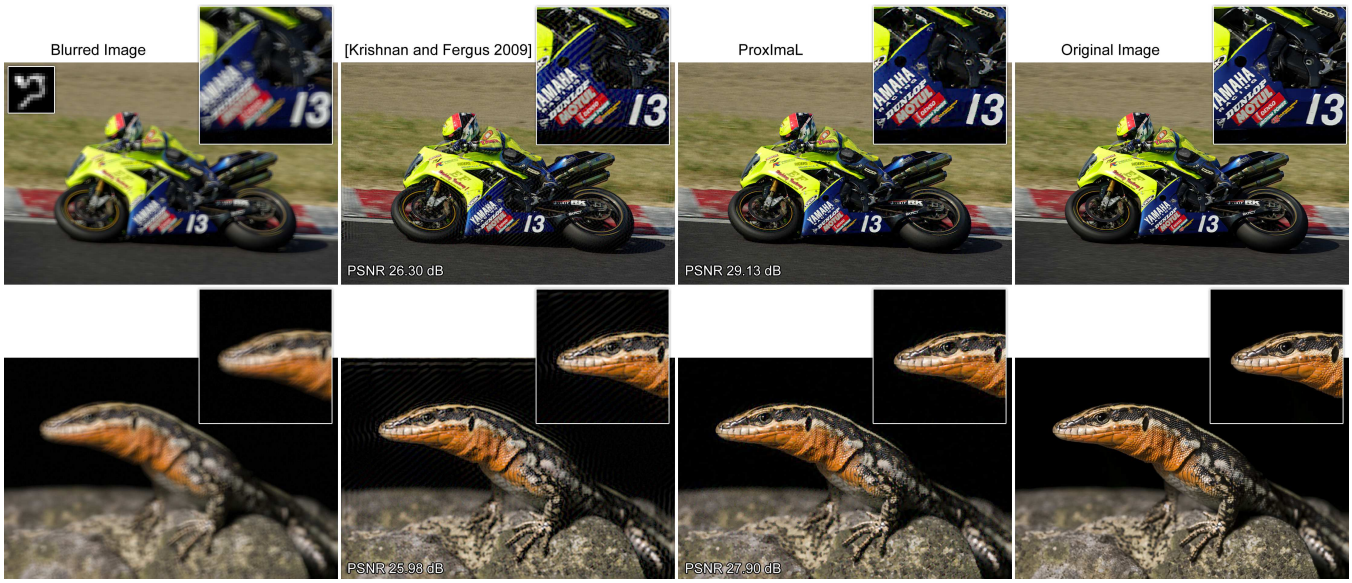


Figure 10: *Poisson deconvolution. We compare the deconvolution technique described in the text with the hyper-Laplacian prior described by Krishnan & Fergus [2009]. Although the latter method is efficient, it fails in the presence of Poisson-distributed shot noise, which often dominates image noise in low-light photography. An appropriate proximal operator for the noise term can mitigate ringing artifacts. This proximal operator is easily compiled with ProxImaL, but is not easily integrated into Krishnan & Fergus's method.*[4]

Usually, this would require the entire solver to be rewritten. Using ProxImaL, we simply add the appropriate proximal operator to the problem formulation as:

```
            x = Variable(300, 300, 3)
    data_term = poisson_norm( conv(x, psf) - input )
grad_sparsity = norm1( grad(x) )
    objective = data_term + grad_sparsity + nonneg(x)
            p = Problem( objective )
```

	Kernel 1	Kernel 2	Kernel 3	Kernel 4	Kernel 5
[Krishnan and Fergus 2009]	21.68	21.08	22.70	23.10	25.19
[Figueiredo and Jose 2010]	22.15	**22.02**	23.59	24.08	26.43
ProxImaL	**22.19**	21.87	**24.11**	**24.31**	**26.58**

Table 8: *Quantitative evaluation of Poisson deconvolution. We show average peak signal-to-noise-ratios (PSNRs) for 12 example images and for 5 different blur kernels. In almost all cases, ProxImaL reconstructs a higher-quality image compared to previous work. Please find individual PSNRs and the images in the supplement.*

The importance of the appropriate error metric is shown in Figure 10. The method proposed by Krishnan & Fergus [2009] is efficient, but it assumes Gaussian noise. It is not easily possible to reformulate this reconstruction method without proximal operators, which is precisely what ProxImaL does. The resulting deconvolved images exhibit significantly less ringing. In Table 8, we also show comparisons of peak signal-to-noise-ratios (PSNRs) averaged over 12 images and 5 different kernels (see supplement). We include an additional comparison to the method proposed by Figueiredo and Jose [2010]. For most cases, ProxImaL produces superior results. Note that the conceptual approach of Figueiredo and Jose is the same as ours, but we replace the conjugate gradient updates with the closed-form inverse of the diagonalized matrix. Yet again, exploiting problem-specific structure leads to better and faster solutions.

8.4 Phase Retrieval

All of the applications discussed above use a linear image formation model. Some imaging problems, however, have to deal with nonlinear and non-convex image formations. An example of such a problem is phase retrieval. As one of the most important problems in

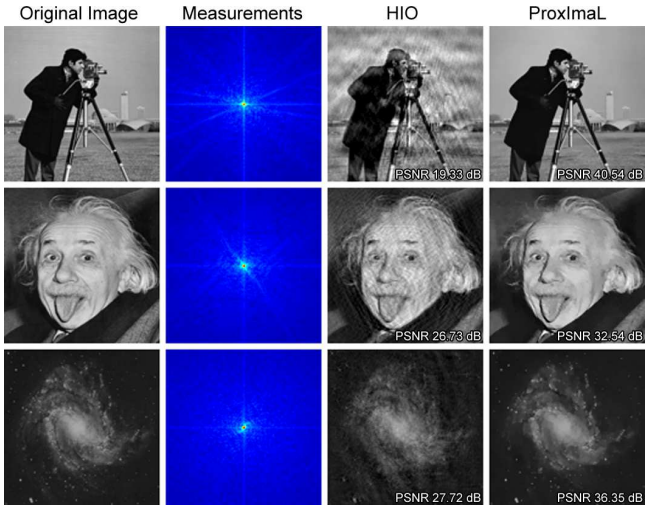


Figure 11: *Phase retrieval. In this nonlinear and nonconvex problem, we measure the Fourier magnitudes of the image (center left column). ProxImaL solves this problem with a nonlinear gradient descent update, which can be combined with any of the image priors previously discussed, simply by compiling their proximal operators into the generated solver.*[5]

electron microscopy, wavefront sensing, astronomy, crystallography, and other scientific imaging areas, it is interesting to evaluate ProxImaL for this challenging problem. The phase retrieval problem is the recovery of a real or complex-valued optical field or image $\mathbf{x}$ from measurements of its Fourier amplitudes:

$$\text{minimize} \quad f\left(|\mathcal{F}\mathbf{x}| - \mathbf{b}\right) + r(\mathbf{x}), \quad s.t. \quad 0 \leq \mathbf{x} \tag{12}$$

Here, $\mathcal{F}$ is the Fourier transform, f is the error metric, and r is an optional image prior. The most common approach to solving Problem (12) is the iterative hybrid input-output (HIO) algorithm [Fienup 1982]. In the spirit of HIO, we can approach this problem by adding

a simple total variation image prior as:

$$\text{minimize} \quad \left\| |\mathcal{F}\mathbf{x}| - \mathbf{b} \right\|_2^2 + \lambda \left\| \nabla \mathbf{x} \right\|_1 + \mathcal{I}_{[0,\infty)}\left(\mathbf{x}\right), \qquad (13)$$

where $\lambda > 0$, $\left\| \nabla \cdot \right\|_1$ is the total variation (ℓ_1-norm of the image gradients), and $\mathcal{I}_{[0,\infty)}\left(\cdot\right)$ is the indicator function that enforces the constraint that $\mathbf{x} \geq 0$. Proximal operators for both total variation and the indicator function are simple to implement, as discussed in the previous sections. We implement the proximal operator of the nonlinear data term as a simple iterative nonlinear gradient descent update that is inspired by HIO. The corresponding ProxImaL code is:

```
            x = Variable(300, 300)
    data_term = phase_ret(x, input)
grad_sparsity = norm1( grad(x) )
    objective = data_term + patch_similarity + nonneg(x)
            p = Problem( objective )
```

In addition to the phase retrieval example shown in Figure 1, we show two additional results in Figure 11. Whereas the HIO solution suffers from strong ringing artifacts, the TV-regularized solution computed by ProxImaL is almost free of artifacts. We do not claim that the ProxImaL implementation provides the state-of-the-art phase retrieval implementation, but it makes it convenient to evaluate modern image priors developed in the computer vision and computational photography communities for scientific imaging applications. We review the mathematical background of phase retrieval and derive its ADMM formulation in the supplement.

9 Discussion

Using a domain-specific language and compiler for image optimization, ProxImaL allows for rapid prototyping of inverse problems in imaging while providing high-performance execution. The performance not only comes from the fact that we can leverage extremely efficient implementations of linear and proximal operators in imaging problems, but also from the fact that ProxImaL intelligently compiles into many different solvers. Each of these solvers or optimization algorithms may have different benefits and limitations for different applications. However, the basic building blocks of all these solvers are proximal operators, which are used to describe linear and nonlinear image formations as well as advanced natural image priors, such as self-similarity. Implemented once, such a proximal operator can be easily re-used for any inverse problem that can be formulated in the ProxImaL language. We demonstrate state-of-the-art performance and quality for the image processing pipeline, burst denoising, Poisson deconvolution, phase retrieval, and other applications.

Limitations

Because ProxImaL's compiler strategies and reformulations exploit the structure of image optimization problems, they may not apply to other problem domains. For example, Natural Language Processing exhibits structure that differs from imaging problems. Even if the considered problem is an image optimization problem and fits into the generalized objective from Problem (2), it may contain non-separable, global penalties that cannot be decomposed into functions with efficient proximal operators. An example are penalties that assign cost based on global light transport simulations from their input. For very simple objectives, such as unconstrained unstructured least-squares problems, ProxImaL does not yield worse solvers than traditional approaches, but also does not improve on them (no structure in the image formation or objective can be exploited). For non-convex problems no convergence guarantees can be given. However, in practice, we found that for many non-convex problems ProxImaL actually can produce efficient solvers, such as the Phase Retrieval application discussed above. The biggest limitation of the proposed framework is the selection of objective and algorithm parameters. For the objective parameters, learning-based approaches have been proposed for denoising problems [Kunisch and Pock 2013], while adaptive schedules have shown to be good recipes for the algorithm parameters [Fougner and Boyd 2015]. Automatic parameter estimation is an exciting area for future research.

Future Work

A key extension that we plan to add is fully-automatic robust parameter estimation and problem scaling, beginning with diagonal scaling methods [Pock and Chambolle 2011; Giselsson and Boyd 2014]. There are also a number of other directions that are interesting to explore. Although we have not explicitly shown applications in multi-camera arrays, the current ProxImaL implementation contains all the necessary building blocks for image optimization with such arrays, for example to demosaick, super-resolve, and deconvolve images captured from slightly different perspectives. We plan to extend the set of proximal operators that are currently implemented (e.g., adding matrix and tensor factorizations). We hope to extend both the set of image formation models, and also the list of supported priors, including other patch-based methods and local low-rank models.

Conclusion

Image processing tasks have long been implemented as fixed function pipelines. A common argument is that available computational resources on cameras or cellphones are scarce, so a fixed function pipeline implemented in hardware is the most efficient solution. While this is true, the trend towards diverse computational photography systems, including multi-camera, light field, time-of-flight, multi-spectral, and 3D cameras along with a paradigm shift towards programmable image processing units (IPUs) in the industry is undeniable. Building on modern optimization techniques, ProxImaL provides an intuitive yet high-performance interface for a range of image optimization tasks in emerging camera systems.

Acknowledgements

We thank Paul Green for many fruitful discussions, and Algolux for providing image data for the burst application. This work was generously supported by the National Science Foundation under grants IIS 1553333 and DGE-114747, DARPA agreement FA8750-14-2-0009, the NSF/Intel Partnership on Visual and Experiential Computing (NSF IIS 1539120), the Intel Compressive Sensing Alliance, and the Stanford Pervasive Parallelism Lab (supported by Oracle, AMD, Intel, and NVIDIA).

References

ALMEIDA, M., AND FIGUEIREDO, M. 2013. Frame-based image deblurring with unknown boundary conditions using the alternating direction method of multipliers. In *Proc. ICIP*, 582–585.

ATTOUCH, H., BOLTE, J., AND SVAITER, B. F. 2011. Convergence of descent methods for semi-algebraic and tame problems: proximal algorithms, forward–backward splitting, and regularized Gauss–Seidel methods. *Mathematical Programming 137*, 1, 91–129.

BECK, A., AND TEBOULLE, M. 2009. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM Journal on Imaging Sciences 2*, 1, 183–202.

BECKER, S., CANDÈS, E., AND GRANT, M. 2011. Templates for convex cone problems with applications to sparse signal recovery. *Mathematical Programming Computation 3*, 3, 165–218.

BERNSTEIN, G. L., SHAH, C., LEMIRE, C., DEVITO, Z., FISHER, M., LEVIS, P., AND HANRAHAN, P. 2015. Ebb: A DSL for physical simluation on CPUs and GPUs. *arXiv e-Print 1506.07577*.

BERTALMIO, M., SAPIRO, G., CASELLES, V., AND BALLESTER, C. 2000. Image inpainting. In *Proc. SIGGRAPH*, 417–424.

BOYD, S., PARIKH, N., CHU, E., PELEATO, B., AND ECKSTEIN, J. 2011. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning 3*, 1, 1–122.

BROOKE, A., KENDRICK, D., MEERAUS, A., AND ROSENTHAL, R. 1988. *GAMS: A user's guide*. Course Technology.

BRUCK, R. 1975. An iterative solution of a variational inequality for certain monotone operators in Hilbert space. *Bulletin of the American Mathematical Society 81*, 5 (Sept.), 890–892.

CHAMBOLLE, A., AND POCK, T. 2011. A first-order primal-dual algorithm for convex problems with applications to imaging. *Journal of Mathematical Imaging and Vision 40*, 1, 120–145.

DANIELYAN, A., KATKOVNIK, V., AND EGIAZARIAN, K. 2012. BM3D frames and variational image deblurring. *IEEE Trans. Image Processing 21*, 4, 1715–1728.

DEBEVEC, P. E., AND MALIK, J. 1997. Recovering high dynamic range radiance maps from photographs. In *Proc. ACM SIGGRAPH*, 369–378.

DIAMOND, S., AND BOYD, S. 2015. Convex optimization with abstract linear operators. In *Proc. IEEE ICCV*.

DIAMOND, S., AND BOYD, S. 2016. Matrix-free convex optimization modeling. In *Optimization and Applications in Control and Data Sciences*. Springer. To appear.

DIAMOND, S., AND BOYD, S. 2016. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*. To appear.

DUPE, F.-X., FADILI, M., AND STARCK, J.-L. 2011. Inverse problems with Poisson noise: Primal and primal-dual splitting. In *Proc. ICIP*.

ESSER, E., ZHANG, X., AND CHAN, T. F. 2010. A general framework for a class of first order primal-dual algorithms for convex optimization in imaging science. *SIAM Journal on Imaging Sciences 3*, 4, 1015–1046.

EVANGELIDIS, G. D., AND PSARAKIS, E. Z. 2008. Parametric image alignment using enhanced correlation coefficient maximization. *Pattern Analysis and Machine Intelligence, IEEE Transactions on 30*, 10, 1858–1865.

FATTAL, R., LISCHINSKI, D., AND WERMAN, M. 2002. Gradient domain high dynamic range compression. In *ACM Trans. Graph.*, vol. 21, ACM, 249–256.

FERGUS, R., SINGH, B., HERTZMANN, A., ROWEIS, S. T., AND FREEMAN, W. T. 2006. Removing camera shake from a single photograph. *ACM Trans. Graph. 25*, 3, 787–794.

FIENUP, J. R. 1982. Phase retrieval algorithms: a comparison. *Applied Optics 21*, 15, 2758–2769.

FIGUEIREDO, M., AND BIOUCAS-DIAS, J. 2010. Restoration of Poissonian images using alternating direction optimization. *IEEE Trans. Image Processing 19*, 12, 3133–3145.

FOLEY, T., AND HANRAHAN, P. 2011. Spark: Modular, composable shaders for graphics hardware. *ACM Trans. Graph. (SIGGRAPH) 30*, 4.

FOUGNER, C., AND BOYD, S. 2015. Parameter selection and preconditioning for a graph form solver. *arXiv e-Print 1503.08366*.

GEMAN, D., AND YANG, C. 1995. Nonlinear image recovery with half-quadratic regularization. *IEEE Trans. Image Processing 4*, 7, 932–946.

GISELSSON, P., AND BOYD, S. 2014. Diagonal scaling in Douglas-Rachford splitting and ADMM. In *Proceedings of the 53rd IEEE Conference on Decision and Control*.

GOLDSTEIN, T., AND OSHER, S. 2009. The split Bregman method for ℓ_1-regularized problems. *SIAM Journal on Imaging Sciences 2*, 2, 323–343.

GRANT, M., AND BOYD, S., 2014. CVX: MATLAB software for disciplined convex programming, version 2.1. http://cvxr.com/cvx.

GU, J., HITOMI, Y., MITSUNAGA, T., AND NAYAR, S. 2010. Coded Rolling Shutter Photography: Flexible Space-Time Sampling. In *Proc. IEEE ICCP*.

HALLAC, D., LESKOVEC, J., AND BOYD, S. 2015. Network lasso: Clustering and optimization in large graphs. In *Proc. ACM SIGKDD*, 387–396.

HEIDE, F., ROUF, M., HULLIN, M. B., LABITZKE, B., HEIDRICH, W., AND KOLB, A. 2013. High-quality computational imaging through simple lenses. *ACM Trans. Graph. 32*, 5, 149.

HEIDE, F., STEINBERGER, M., TSAI, Y.-T., ROUF, M., PAJAK, D., REDDY, D., GALLO, O., LIU, J., HEIDRICH, W., EGIAZARIAN, K., KAUTZ, J., AND PULLI, K. 2014. FlexISP: A flexible camera image processing framework. *ACM Trans. Graph. (SIGGRAPH Asia) 33*, 6.

HESTENES, M., AND STIEFEL, E. 1952. Methods of conjugate gradients for solving linear systems. *J. Res. N.B.S. 49*, 6, 409–436.

JOSHI, N., ZITNICK, C. L., SZELISKI, R., AND KRIEGMAN, D. J. 2009. Image deblurring and denoising using color priors. In *Proc. IEEE CVPR*, 1550–1557.

KRISHNAN, D., AND FERGUS, R. 2009. Fast image deconvolution using hyper-Laplacian priors. In *Advances in Neural Information Processing Systems*, 1033–1041.

KRISHNAN, D., AND SZELISKI, R. 2011. Multigrid and multilevel preconditioners for computational photography. *ACM Trans. Graph. 30*, 6, 177.

KUNISCH, K., AND POCK, T. 2013. A bilevel optimization approach for parameter learning in variational models. *SIAM Journal on Imaging Sciences 6*, 2, 938–983.

LEHOUCQ, R., AND SORENSEN, D. 1996. Deflation techniques for an implicitly restarted Arnoldi iteration. *SIAM Journal on Matrix Analysis and Applications 17*, 4, 789–821.

LEVIN, A., LISCHINSKI, D., AND WEISS, Y. 2004. Colorization using optimization. In *ACM Trans. Graph.*, vol. 23, 689–694.

LEVIN, A., ZOMET, A., PELEG, S., AND WEISS, Y. 2004. Seamless image stitching in the gradient domain. In *Proc. ECCV*. 377–389.

LI, G., AND PONG, T. K. 2015. Global convergence of splitting methods for nonconvex composite optimization. *arXiv e-Print 1407.0753*.

LOFBERG, J. 2004. YALMIP: A toolbox for modeling and optimization in MATLAB. In *Proc. IEEE Int. Symp. Computed Aided Control Systems Design*, 294–289.

MÖLLENHOFF, T., STREKALOVSKIY, E., MOELLER, M., AND CREMERS, D. 2015. The primal-dual hybrid gradient method for semiconvex splittings. *SIAM Journal on Imaging Sciences 8*, 2, 827–857.

MOREAU, J.-J. 1965. Proximité et dualité dans un espace hilbertien. *Bulletin de la Société mathématique de France 93*, 273–299.

OCHS, P., CHEN, Y., BROX, T., AND POCK, T. 2014. iPiano: Inertial proximal algorithm for nonconvex optimization. *SIAM Journal on Imaging Sciences 7*, 2, 1388–1419.

O'DONOGHUE, B., CHU, E., PARIKH, N., AND BOYD, S. 2015. Operator splitting for conic optimization via homogeneous self-dual embedding. *arXiv e-Print 1312.3039*.

PAIGE, C., AND SAUNDERS, M. 1982. LSQR: An algorithm for sparse linear equations and sparse least squares. *ACM Trans. Mathematical Software 8*, 1, 43–71.

PARIKH, N., AND BOYD, S. 2013. Proximal algorithms. *Foundations and Trends in Optimization 1*, 3, 123–231.

POCK, T., AND CHAMBOLLE, A. 2011. Diagonal preconditioning for first order primal-dual algorithms in convex optimization. In *Proceedings of the IEEE International Conference on Computer Vision*, 1762–1769.

POCK, T., CREMERS, D., BISCHOF, H., AND A.CHAMBOLLE. 2009. An algorithm for minimizing the Mumford-Shah functional. In *Proceedings of the IEEE International Conference on Computer Vision*, 1133–1140.

RAGAN-KELLEY, J., BARNES, C., ADAMS, A., PARIS, S., DURAND, F., AND AMARASINGHE, S. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN 48*, 6, 519–530.

ROBINI, M. C., AND ZHU, Y. 2015. Generic half-quadratic optimization for image reconstruction. *SIAM Journal on Imaging Sciences 8*, 3, 1752–1797.

ROCKAFELLAR, R. 1976. Augmented Lagrangians and applications of the proximal point algorithm in convex programming. *Mathematics of Operations Research 1*, 2, 97–116.

SCHMIDT, U., AND ROTH, S. 2014. Shrinkage fields for effective image restoration. In *Proc. IEEE CVPR*, 2774–2781.

SIDKY, E. Y., AND PAN, X. 2008. Image reconstruction in circular cone-beam computed tomography by constrained, total-variation minimization. *Physics in medicine and biology 53*, 17, 4777.

TIAN, L., AND WALLER, L. 2015. 3D intensity and phase imaging from light field measurements in an LED array microscope. *Optica 2*, 2, 104–111.

TSAI, Y.-T., STEINBERGER, M., PAJAK, D., AND PULLI, K. 2014. Fast ANN for high-quality collaborative filtering. In *High Performance Graphics*.

UDELL, M., MOHAN, K., ZENG, D., HONG, J., DIAMOND, S., AND BOYD, S. 2014. Convex optimization in Julia. *Workshop on High Performance Technical Computing in Dynamic Languages*.

VIDIMICE, K., WANG, S.-P., RAGAN-KELLEY, J., AND MATUSIK, W. 2013. OpenFab: A programmable pipeline for multi-material fabrication. *ACM Trans. Graph. (SIGGRAPH) 32*, 4.

WYTOCK, M., WANG, P.-W., AND ZICO KOLTER, J. 2015. Convex programming with fast proximal and linear operators. *arXiv e-Print 1511.04815*.

ZHANG, L., WU, X., BUADES, A., AND LI, X. 2011. Color demosaicking by local directional interpolation and nonlocal adaptive thresholding. *Journal of Electronic Imaging 20*, 2, 023016–023016.

ZHU, Y. 2015. An augmented ADMM algorithm with application to the generalized lasso problem. *Journal of Computational and Graphical Statistics*, just-accepted.

ZORAN, D., AND WEISS, Y. 2011. From learning models of natural image patches to whole image restoration. In *Proc. IEEE ICCV*, 479–486.

Notes

1. Images are from the McMaster color image dataset [Zhang et al. 2011].
2. Images are from the FlexISP dataset.
3. Images courtesy of Flickr user susan402.
4. Images courtesy of Wikipedia users Laitche and Benny_Trapp.
5. Images from http://www.imageprocessingplace.com/root_files_V3/image_databases.htm.