POGS – Proximal Operator Graph Solver

Chris Fougner

June 4, 2014

1 Introduction

As data sets continue to increase in size, it becomes increasingly important to design algorithms that scale with the size of the data. Typical interior point solvers are able to handle a wide range of problems and produce solutions to high accuracy. However, they do not scale to the problem sizes that one encounters in “Big Data” settings. Beyond a certain point, only parallel and distributed algorithms remain competitive. The Alternating Direction Method of Multipliers (ADMM) is a method to solve optimization problems in a distributed fashion. Although it generally doesn’t achieve the same accuracy as an interior point solver, it can be applied to much larger problems.

We have created an ADMM-based solver targeted at a class of optimization problems, for which the proximal operators of all terms in the objective are known. We have implemented both CPU and GPU versions, with wrappers for MATLAB.

2 Problem Formulation

The problem we wish to solve is

\[
\begin{align*}
\text{minimize} & \quad f(y) + g(x) \\
\text{subject to} & \quad y = Ax,
\end{align*}
\]

where \( x \in \mathbb{R}^n \) and \( y \in \mathbb{R}^m \) are the optimization variables. The terms \( A \in \mathbb{R}^{m \times n}, f : \mathbb{R}^m \to \mathbb{R} \) and \( g : \mathbb{R}^n \to \mathbb{R} \) encode the problem data. We further require that \( f \) and \( g \) be convex and separable, meaning that it must be possible to express \( f \) (and \( g \)) as \( f(y) = \sum_{i=1}^m f_i(y_i) \), where \( y_i \in \mathbb{R} \), and \( f_i : \mathbb{R} \to \mathbb{R} \cup \{\infty\} \). This formulation is known as graph form [PB13], since \((x,y)\) are constrained to lie in the graph \( \{(x,y) \in \mathbb{R}^{m+n} \mid y = Ax\} \) of \( A \).
Using ADMM [BPC+11] we can formulate an iteration for solving this problem:

\[
\begin{align*}
    x^{k+1/2} & := \text{prox}_g(x^k - \tilde{x}^k) \\
    y^{k+1/2} & := \text{prox}_f(y^k - \tilde{y}^k) \\
    (x^{k+1}, y^{k+1}) & := \Pi_A(x^{k+1/2} + \tilde{x}^k, y^{k+1/2} + \tilde{y}^k) \\
    \tilde{x}^{k+1} & := \tilde{x}^k + x^{k+1/2} - x^{k+1} \\
    \tilde{y}^{k+1} & := \tilde{y}^k + y^{k+1/2} - y^{k+1},
\end{align*}
\]

where \(x^k\) and \(y^k\) are the \(k\)th iterate of the variables above, \(x^{k+1/2}\) and \(y^{k+1/2}\) are consensus variables and \(\tilde{x}^k \in \mathbb{R}^n\), resp. \(\tilde{y}^k \in \mathbb{R}^m\) are dual variables associated with the constraints \(x^{k+1/2} = x^{k+1}\) and \(y^{k+1/2} = y^{k+1}\). The operator \(\Pi_A\) is the orthogonal projection onto the graph of \(A\) and the proximal operator of a function \(f\) at a point \(v\) is defined as

\[
\text{prox}_f(v) = \arg\min_x \left( f(x) + \frac{\rho}{2} \|x - v\|^2 \right),
\]

where \(\rho\) is a penalty parameter for deviating from \(v\). Projecting \((x, y)\) onto \(A\) is equivalent to a least squares problem, whose solution is given by the KKT system

\[
\begin{bmatrix}
    I & A^T \\
    A & -I
\end{bmatrix}
\begin{bmatrix}
    x \\
    y
\end{bmatrix} =
\begin{bmatrix}
    c + A^Td \\
    0
\end{bmatrix}.
\]

Since we require that \(f\) and \(g\) be separable, the proximal step can be done component-wise and, therefore, in parallel. As a result, the main computational challenge is to solve this system of equations.

### 3 Diagonal Preconditioning

Diagonal preconditioning and an adaptive updating of \(\rho\) is necessary to ensure stability and robustness. A three step procedure was employed

1. Use the Sinkhorn-Knopp [SK+67] algorithm to generate diagonal matrices \(D\) and \(E\), such that \(DAE\) is column-stochastic, and row-stochastic up to a factor of \(\sqrt{m/n}\). The matrix \(\tilde{A}\) is the element-wise absolute value of \(A\).

2. Let \(\alpha\) be the mean of the diagonal of \((DAE)^T(DAE)\). Scale \(\tilde{D} = \alpha^{-1/2}D\) and \(\tilde{E} = \alpha^{-1/2}E\) and finally apply the ADMM iteration to \(DAE\).

3. In each iteration update \(\rho\) according to

\[
\rho^{k+1} := \begin{cases} 
    \rho^k \tau & \text{if } \|z^{k+1/2} - z^k\|_2 > \mu \|z^{k+1} - z^k\|_2 \\
    \rho^k / \tau & \text{if } \|z^{k+1} - z^k\|_2 > \mu \|z^{k+1/2} - z^k\|_2 \\
    \rho^k & \text{otherwise},
\end{cases}
\]

where \(\mu = \tau = 10\) was used. This is similar to the update (3.13) in [BPC+11].

For details on the method, refer to [TJ14].
4 Software Design

To guide software design decisions, we focused on simplicity and performance.

Solver Interface. The interface to the the solver is a single struct, which fully specifies the inputs, outputs and parameters

```cpp
template<typename T>
struct AdmmData {
    // Input
    std::vector<FunctionObj<T>> f, g;
    const T *A;
    size_t m, n;

    // Output
    T *x, *y, optval;

    // Parameters
    T rho; // default: 1.0
    unsigned int max_iter; // default: 1000
    T rel_tol, abs_tol; // default: (1e-2, 1e-4)
    bool quiet; // default: true
};
```

The FunctionObj is a struct, which parameterizes a family of functions (see next section on Proximal Operator Library for details). All inputs and outputs must be specified by the user and all parameters are given a default value by the constructor.

Proximal Operator Library. To make the solver as general as possible, each term \((f_1, \ldots, f_m, g_1, \ldots, g_n)\) in the objective must be specified separately. The functions \(f_i\) (resp. \(g_j\)) are each described by a set of five parameters \(a, b, c, d\) and \(h\), which together define the function

\[
f_i(x) = c h(a x - b) + d x,
\]

where \(a, b, d \in \mathbb{R}\) and \(c \in \mathbb{R}_+\) are parameters, \(x \in \mathbb{R}\) is the variable and \(h : \mathbb{R} \rightarrow \mathbb{R} \cup \{\infty\}\) is one of 12 convex functions

\[
\begin{align*}
    h(x) &= |x|, & h(x) &= \text{huber}(x), & h(x) &= x, & h(x) &= I(0 \leq x \leq 1), \\
    h(x) &= I(x = 0), & h(x) &= I(0 \leq x), & h(x) &= -\log x, & h(x) &= \log(1 + e^x), \\
    h(x) &= \max(0, x), & h(x) &= (1/2)x^2, & h(x) &= e^x, & h(x) &= 1/x,
\end{align*}
\]

where \(I(x \in C)\) is the indicator function of the convex set \(C\). By knowing the proximal operator of \(h\), we can compute the proximal operator of \(f_i\), using the formula

\[
\text{prox}_{f_i, \rho}(v) = \frac{1}{a} \left( \text{prox}_{h, \rho/(ca^2)}(a(v - d/\rho) - b) + b \right).
\]
The advantage of this parametric formulation is that the use of proximal operators is hidden from the user. We encapsulate these five parameters in a FunctionObj, which we define as

```cpp
template<typename T>
struct FunctionObj {
    Function h;  // enum { kAbs, kHuber, kIndEq0, ..}
    T a, b, c, d; // default: (1.0, 0.0, 1.0, 0.0)
};
```

MATLAB Wrapper. The MATLAB wrapper is a single MEX-function, with the signature

```matlab
[x, y, optval] = pogs(A, f, g, params)
```

where A is a matrix, f, g, and params are structs, x and y are vectors, and optval is a scalar. The structs f and g have fields h, a, b, c and d, each of which must either be a vector of dimension size(A,1) (resp. size(A,2)) or a scalar. If a scalar is specified, then it is assumed that the scalar should be repeated size(A,1) (resp. size(A,2)) times. All fields except h are optional. The params struct has fields rel_tol, abs_tol, rho, max_iter and quiet. Specifying params is optional.

Solving KKT System. The KKT system can be solved by forming the normal equations and performing a Cholesky solve [PB13]. If A is fat, then this can be expressed as

\[
y := (I + AA^T)^{-1}(Ac + AA^Td)
\]

\[
x := c + A^T(d - y).
\]

If A is skinny, it is more efficient to compute

\[
x := (I + A^TA)^{-1}(c + A^Td)
\]

\[
y := Ax.
\]

Alternatively, we could use the QR decomposition of either \([A^T \ I]\) or \([A^T \ I]^T\) (depending on whether A is fat or skinny) to find x and y. Both methods have a computational complexity \(O((m + n)\ min(m, n)^2)\) [Dem97]. The QR decomposition will be better conditioned, but since we are adding I to AA^T (resp. A^TA), the condition number of the matrix that we are inverting is bounded by \(\sigma^2_{\max}(A)\), which can be controlled by scaling A appropriately. Either method is likely to perform well, so we have chosen to use the Cholesky decomposition, because it is slightly simpler. All matrices are independent of the iteration \(k\), and, as a result, the factorization can be computed once and cached for subsequent iterations.

Coding Style. We follow the Google C++ Style Guide [WSE^+].
5 Implementations

The CPU and GPU implementations along with MATLAB wrappers have been completed. All code is open source and can be found on GitHub:

https://github.com/foges/pogs

**CPU Implementation.** For the CPU version we used a mixture of OpenMP and a tuned BLAS library (currently the Apple Accelerate Framework) for parallelism. The GNU Scientific Library (GSL) was used as a BLAS interface.

**GPU Implementation.** The GPU code is almost identical to the CPU code. Internally, the CUDA Thrust library is used to evaluate proximal operators. We wrote our own parallel block Cholesky routine as well as templated wrappers around the necessary cuBLAS functions. The code can be found in the `pogs/cml` directory and we are considering spinning it off as a separate project. The GPU implementation has support for both single precision and double precision matrices.

**MATLAB Wrapper.** The MATLAB wrapper can be linked to either the CPU or GPU implementation. When linking to the CPU implementation, MATLAB’s native BLAS library is used.

**Compiling.** To compile the the C++ code, open terminal and execute

```
$ make [gpu]
```

Alternatively, the MATLAB wrapper can compiled and installed by executing

```
>> pogs_setup [gpu]
```

in the MATLAB console. Specifying `gpu` links to the GPU rather than CPU backend.

6 Numerical Example

**Lasso Problem.** Consider the Lasso problem

\[
\text{minimize} \quad (1/2) \|Ax - b\|_2^2 + \lambda \|x\|_1,
\]

with \( A \in \mathbb{R}^{m \times n} \), \( b \in \mathbb{R}^m \), \( \lambda \in \mathbb{R}_+ \), which is equivalent to the graph form problem

\[
\text{minimize} \quad (1/2) \|y - b\|_2^2 + \lambda \|x\|_1
\]

subject to \( y = Ax \).

We generate entries of \( A \) as independent samples from a normal distribution with standard deviation \( 1/n \), then compute \( b = A\tilde{x} + v \), where each entry in \( \tilde{x} \) was drawn
from a standard normal distribution with probability 0.2 and is identical to zero with probability 0.8. The vector $v$ is Gaussian noise with standard deviation $1/2$.

Using the MATLAB interface, this problem can be specified as

```matlab
f.h = kSquare;
f.b = b;
g.h = kAbs;
g.c = lambda;
[x, y, optval] = pogs(A, f, g);
```

**Results.** In this numerical example, $n$ was fixed and $m$ was varied.

For large values of $m$, the CPU implementation was up to 300 times faster than CVX. Switching from CPU to GPU implementation gave an additional improvement of 5x to 8x when $n = 500$. For $n = 10,000$, the improvement factor was between 13x to 30x.

**7 Conclusion and Future Work**

We have created an open source ADMM-based solver. The advantage of such a solver over interior point based solvers has been shown empirically. Next steps include a warm start feature, wrappers for Julia, Python, and R, and sparse matrix support.

**Acknowledgments**

We would like to thank Stephen Boyd, Neal Parikh, Madeleine Udell, and Pontus Giselsson for many insightful discussions. Notation and math has been borrowed from [PB13].
References


