Final Document: AMSC/CMSC 663 and 664

The Alternating Direction Method of Multipliers
An ADMM Software Library

Peter Sutor, Jr.
psutor@umd.edu

Project Advisor

Dr. Tom Goldstein
tomg@cs.umd.edu

Assistant Professor
Department of Computer Science
University of Maryland

May 15, 2016

Abstract

The Alternating Direction Method of Multipliers (ADMM) is a method that solves convex optimization problems of the form $\min(f(x) + g(z))$ subject to $Ax + Bz = c$, where A and B are suitable matrices and c is a vector, for optimal points (x_{opt}, z_{opt}). It is commonly used for distributed convex minimization on large scale data-sets. However, it can be technically difficult to implement and there is no known way to automatically choose an optimal step size for ADMM. Our goal in this project is to simplify the use of ADMM by making a robust, easy-to-use software library for all ADMM-related needs, with the ability to adaptively select step-sizes on every iteration. The library will contain a general ADMM method, as well as solvers for common problems that ADMM is used for. It also tries to implement adaptive step-size selection, have support for parallel computing and have user-friendly options and features.

Introduction

The generalization of ADMM's usage is in solving convex optimization problems where the data can be arbitrarily large. That is, we wish to find $x_{opt} \in X$ such that:

$$f(x_{opt}) = \min\{f(x) : x \in X\}, \tag{1}$$

given some constraint $Ax = b$, where $X \subset \mathbb{R}^n$ is called the *feasible set*, $f(x) : \mathbb{R}^n \longmapsto \mathbb{R}$ is the *objective function*, X and f are convex, matrix $A \in \mathbb{R}^{m \times n}$ and vector $b \in \mathbb{R}^m$. Our input x here may have a huge amount of variables/dimensions, or an associated data matrix A for it can simply be hundreds of millions of entries long. In such extreme cases, the traditional techniques for minimization may be too slow, despite how fast they may be on normal sized problems.

Generally, such issues are solved by using parallel versions of algorithms to distribute the workload across multiple processors, thus speeding up the optimization. But our traditional optimization algorithms are not suitable for parallel computing, so we must use a method that is. Such a method would have to decentralize the optimization; one good way to do this is to use the *Alternating Direction Method of Multipliers* (**ADMM**). This convex optimization algorithm is robust and splits the problem into smaller pieces that can be optimized in parallel.

We will first give some background on ADMM, then describe how it works, how it is used to solve problems in practice, and how it was implemented. Then, we discuss common problems that ADMM is used to solve and how they were implemented as general solvers in the ADMM library. Finally, we discuss our results on adaptive step-size selection and draw conclusions on the potential for this to be used in practice. We end by discussing potential future work on this solver library and adaptive step-sizes.

Background

In the following sections, we briefly describe the general optimization strategy ADMM uses, and the two algorithms ADMM is a hybrid of. For more information, refer to [1].

The Dual Problem

Consider the following equality-constrained convex optimization problem:

$$\min_x(f(x)) \text{ subject to } Ax = b \tag{2}$$

This is referred to as the *primal problem* (for a *primal function* f) and x is referred to as the *primal variable*. To help us solve this, we formulate a different problem using the Lagrangian and solve that. The *Lagrangian* is defined as

$$L(x, y) = f(x) + y^T(Ax - b). \tag{3}$$

We call the *dual function* $g(y) = \inf_x(L(x, y))$ and the *dual problem* $\max_y(g(y))$, where y is the *dual variable*. With this formulation, we can recover $x_{opt} = \arg\min_x(L(x, y_{opt}))$; f's minimizer.

One method that gives us this solution is the *Dual Ascent Method* (**DAM**), characterized at iteration k by computing until convergence:

1. $x^{(k+1)} := \arg\min_x(L(x, y^{(k)}))$ (minimization for $f(x)$ on x)
2. $y^{(k+1)} := y^{(k)} + \alpha^{(k)}(Ax^{(k+1)} - b)$ (update y for next iteration)

Here, $\alpha^{(k)}$ is a step size for the iteration k and we note that $\nabla g(y^{(k)}) = Ax_{opt} - b$, and $x_{opt} = \arg\min_x(L(x, y^{(k)}))$. If g is differentiable, this algorithm strictly converges and seeks out the gradient of g. If g is not differentiable, then we do not have monotone convergence and the algorithm seeks out the negative of a sub-gradient of $-g$. Note that the term $y^T(Ax - b)$ acts as a penalty function that guarantees minimization occurs on the given constraint.

Dual Decomposition

It's important to realize that for high-dimensional input we may want to parallelize DAM for better performance. The technique for this is described in this section. Suppose that our objective is *separable*; i.e. $f(x) = f_1(x_1) + \cdots + f_n(x_n)$, and $x = (x_1, ..., x_n)^T$. Then we can say the same for the Lagrangian. From (3), we have: $L(x, y) = L_1(x_1, y) + \cdots + L_n(x_n, y) - y^T b$, where $L_i = f(x_i) + y^T A_i x_i$. Thus, our x-minimization step in the DAM is split into n separate minimizations that can be carried out in parallel:

$$x_i^{(k+1)} := \arg\min_{x_i} \left(L_i(x_i, y^{(k)}) \right).$$

This leads to a good plan for parallelization: disperse $y^{(k)}$, update x_i in parallel then add up the $A_i x_i^{(k+1)}$. This is called *Dual Decomposition* (**DD**), and was originally proposed by Dantzig

and Wolfe [2, 3], and by Benders [4]. However, Dual Decomposition's general idea is primarily due to Everett [5]. The algorithm computes the above x-minimization step for $i = 1, ..., n$, in parallel, then coordinates to update the *dual variable*:

$$y^{(k+1)} := y^{(k)} + \alpha^{(k)} \left(\sum_{i=1}^{n} A_i x_i^{(k+1)} - b \right).$$

Initially, this seems great. But this algorithm requires several big assumptions (sufficiently smooth and decomposible f), and can be slow at times. We need a faster method.

Method of Multipliers

What if we want to make DAM more robust, with faster iterations and convergence? The *Method of Multipliers* (**MM**) can do this, as proposed by Hestenes [6, 7] and Powell [8]. Simply swap the Lagrangian for an *Augmented Lagrangian*:

$$L_\rho(x, y) = f(x) + y^T(Ax - b) + (\rho/2)||Ax - b||_2^2 \text{ , where } \rho > 0. \tag{4}$$

Note the addition of another penalty term that penalizes straying too far from the constraint during minimization over the length of ρ. Now our iteration computes until convergence:

1. $x^{(k+1)} := \arg\min_x(L_\rho(x, y^{(k)}))$ (minimization Lagrangian for x)
2. $y^{(k+1)} := y^{(k)} + \rho(Ax^{(k+1)} - b)$ (update y for next iteraton)

Here, ρ is the *dual update step length*, chosen to be the same as the penalty coefficient ρ in (4). This Augmented Lagrangian can be shown to be differentiable under mild conditions for the primal problem. According to [1], for a differentiable f, the *optimality conditions* are:

Primal Feasibility:	$Ax_{opt} - b = 0$
Dual Feasibility:	$\nabla f(x_{opt}) + A^T y_{opt} = 0$

At each iteration k, $x^{(k+1)}$ minimizes $L_\rho(x, y^{(k)})$, so:

$$\begin{aligned} \nabla_x L_\rho(x^{(k+1)}, y^{(k)}) &= \nabla_x f(x^{(k+1)}) + A^T(y^{(k)} + \rho(Ax^{(k+1)} - b)) \\ &= \nabla_x f(x^{(k+1)}) + A^T y^{(k+1)} = 0 \end{aligned}$$

Thus, our dual update $y^{(k+1)}$ makes $(x^{(k+1)}, y^{(k+1)})$ *dual feasible*; the *primal feasibility* is achieved as $(Ax^{(k+1)} - b) \to 0$ (convergence on constrained solution).

What does all this mean?

Generally, MM is faster, more robust (does not require a smooth f) and has more relaxed convergence conditions than DD. However, MM's quadratic penalty in the Augmented Lagrangian

prevents us from being able to parallelize the x-update like in DD. With this set-up, we cannot have the advantages of both MM and DD. This is where ADMM comes into play.

The Alternating Method Of Multipliers (ADMM)

Having covered the background of ADMM, we can now begin discussing the algorithm itself.

The General ADMM Algorithm

ADMM combines the advantages of DD and MM. It solves problems of the form:

$$\min\left(f(x)+g(z)\right) \text{ subject to } Ax+Bz=c, \tag{5}$$

where f and g are both convex. Note that the objective is separable into two sets of variables. ADMM defines and uses a special Augmented Lagrangian to allow for decomposition:

$$L_\rho(x,z,y)=f(x)+g(x)+y^T(Ax+Bz-c)+\frac{\rho}{2}||Ax+Bz-c||_2^2 \tag{6}$$

The ρ in (6) is the step length. The original ADMM algorithm was proposed by Gabay and Mercier [9], and Glowinski, and Marrocco [10]. Many further findings in ADMM were discovered by Eckstein and Bertsekas [11]. At iteration k, we minimize for x, then z, and finally, update y, keeping the other variables constant during each minimization. This gives us the ADMM algorithm shown in Algorithm 1.

Algorithm 1 The Alternating Direction Method of Multipliers (ADMM)

procedure ADMM(A, B, c, ρ, OBJECTIVE, ARGMINX, ARGMINZ, STOPCOND)
  Set x, z and y to some initial value.
  while STOPCOND(x, z, y, A, B, c) $\neq 1$ **do**
    $x :=$ ARGMINX(x,z,y,ρ) (minimize $f(x)$ for x)
    $z :=$ ARGMINZ(x,z,y,ρ) (minimize $g(z)$ for z)
    $y := y+\rho(Ax+Bz-c)$ (Dual variable update)
  return (x, z, OBJECTIVE(x, z))

There are a few things to note about this formulation of ADMM. The way the algorithm works does not require explicitly knowing the objective function $f(x)+g(z)$; it only requires the constraint variables (A, B, and c), minimizing functions ARGMINX and ARGMINZ, and a stopping condition function STOPCOND. The algorithm only cares about the objective function OBJECTIVE for evaluating the final, minimized value. This can even be left up to the user. Another detail to note is that the algorithm decouples the objective function on variables x and z and minimize on them independently. This formulation changes our optimality conditions a little.

ADMM's Optimality Conditions

Assume f and g are differentiable. We now have a second dual feasible condition due to the z-minimization step in ADMM that did not exist in MM. The other conditions are slightly altered from MM for the ADMM constraint:

Primal Feasibility:	$Ax+Bz-c=0$
Dual Feasibility:	$\nabla f(x)+A^Ty=0,$
	$\nabla g(z)+B^Ty=0$

Assume $z^{(k+1)}$ minimizes $L_\rho(x^{(k+1)}, z, y^{(k)})$; we want to show that the dual update makes $(x^{(k+1)}, z^{(k+1)}, y^{(k+1)})$ satisfy the dual feasible condition for g. We proceed using the same strategy as was done in MM, using ADMM's Augmented Lagrangian instead:

$$\begin{aligned} 0 &= \nabla g(z^{(k+1)}) + B^T y^{(k)} + \rho B^T (Ax^{(k+1)} + Bz^{(k+1)} - c) \\ &= \nabla g(z^{(k+1)}) + B^T y^{(k+1)} \end{aligned}$$

Thus, the dual update makes $(x^{(k+1)}, z^{(k+1)}, y^{(k+1)})$ satisfy the second dual feasible condition. As for the other dual and primal conditions, they are both achieved as $k \to \infty$.

Convergence of ADMM

What conditions need to be satisfied in order for ADMM to be guaranteed to converge? According to [1, 11], ADMM requires two assumptions:

1. The infinite domain of functions f and g must be closed, proper and convex. In Euclidean space, a set is *closed* if its complement is an *open set*, a generalization of an open interval on the reals in higher dimensions. A set is *convex* in Euclidean space if for every pair of points in the set, all points on the line segment between them lie in the set as well. A convex function f's domain is *proper* if its effective domain (all x such that $f(x) < \infty$) is nonempty and never reaches $-\infty$.
2. The Augmented Lagrangian in (6) has a saddle point for $\rho = 0$.

Assumption 1 essentially states that the subproblems solved by ADMM in the minimization steps for f and g must indeed be solvable, even if not uniquely. This condition allows for f and g to be non-differentiable and take on the value of positive infinity. It also guarantees that for assumption 2, the saddle point is finite.

Thus, the conditions are not to strict for convergence. These two assumptions guarantee that residuals between iterates converge to 0, the objective approaches the optimal value, and that the dual value also approaches the optimal value.

Scaled Dual Form of ADMM

By scaling the dual variable y in ADMM by $1/\rho$, we can rewrite the algorithm in a simpler way. Let *residual* $r = Ax + Bz - c$, then:

$$\begin{aligned} L_\rho(x, z, y) &= f(x) + g(z) + y^T r + (\rho/2)||r||_2^2 \\ &= f(x) + g(z) + (\rho/2)||r + (1/\rho)y||_2^2 - (1/2\rho)||y||_2^2 \\ &= f(x) + g(z) + (\rho/2)||r + u||_2^2 - constant_y \\ &= L_\rho(x, z, u), \end{aligned} \tag{7}$$

where $u = (1/\rho)y$. Now the ADMM can be written in a simpler fashion, as shown in Algorithm 2.

Algorithm 2 Scaled Dual ADMM

procedure ScaledADMM(A, B, c, ρ, objective, argminx, argminz, stopcond)
  Set x, z and u to some initial value.
  while stopcond(x, z, u, A, B, c) $\neq 1$ **do**
    $x := \text{argminx}(x, z, u, \rho)$ (minimize $f(x)$ for x)
    $z := \text{argminz}(x, z, u, \rho)$ (minimize $g(z)$ for z)
    $u := u + (Ax + Bz - c)$ (Dual variable update)
  return (x, z, objective(x, z))

Writing General Convex Problems in ADMM Form

What if we want to use ADMM for a general convex optimization problem; that is, the generic problem: $\min f(x)$, subject to $x \in \mathbb{S}$, with f and $\mathbb{S}$ convex? One way is to simply write: $\min (f(x) + g(z))$, subject to $x - z = 0$, hence $x = z$. The question is, what do we make g? A good idea is to let $g(z) = \mathbb{I}_{\mathbb{S}}(z)$, the indicator function that z is in the set $\mathbb{S}$ (i.e., $g(z) = 0$ if $x \in \mathbb{S}$, else $g(z) = 1$), as it does not impact the problem we are trying to solve, but enforces the solution belonging in S.

Notice that in this formulation, $B = -\text{I}$, so z-minimization boils down to

$$\arg\min (g(z) + (\rho/2)||z - v||_2^2) = \mathbf{prox}_{g,\rho}(v), \tag{8}$$

with $v = x^{(k+1)} + u^{(k)}$, where $\mathbf{prox}_g(v)$ is the *proximal operator* of v on function g. The proximal operator is defined as

$$\mathbf{prox}_g(v) = \arg\min_z (g(z) + \frac{1}{2}||z - v||_2^2) \tag{9}$$

We add an additional parameter ρ to our version of the proximal operator for the step size. Since we have the special case that $g(z)$ is the indicator function, we can compute the proximal operator by projecting v onto $\mathbb{S}$, which is clearly the solution. Note that the indicator function is only convex if the set S is convex. Therefore, this way of writing the problem is limited to convex solution spaces.

One other common scenario is the situation where $g(z) = \lambda||z||_1$, a proximal mapping of the l_1 norm. This is often referred to as a "regularization term", where λ is the regularization parameter. It penalizes the solution from having extreme parameter values, thus preventing "overfitting" (tending to describe noise instead of the observed relationship) and making the problem less ill-posed. A good way to interpret this is that regularization affects the type of solution we will get - how much does noise or randomness fit in to the model? Then, we can use a technique called *soft-thresholding*, described in [1]. This gives the individual components of z in the minimization by computing:

$$z_i^{(k+1)} := \text{sign}(v_i)(|v_i| - t)_+ \tag{10}$$

The value $t = \lambda/\rho$ in our case. In general, this value t is known as the thresholding parameter. We perform this over all the components at once for our z-minimization step. Notice that both the x and z-minimization steps in ADMM are proximal operators, by definition of the Augmented Lagrangian. Thus, in a general ADMM program, we can ask for functions that compute the proximal operators for both as input functions argminx and argminz for Algorithms 1 and 2.

This formulation allows us to solve problems of the form in (1). If there is already a constraint like the one in (2), we can still use this formulation to solve it via ADMM. However, the constraint $Ax = b$ cannot simply be ignored; the parameters A and b will be incorporated into the x update step; i.e., they are part of the minimization problem for x (the solutions for which are supplied by the user as a function in generalized ADMM). How to handle the x update is dependent on the problem, though there are solutions for general cases such as in Quadratic Programming.

What about inequality constrained problems, such as $Ax \leq b$? There are some tricks one can do to solve certain inequality constrained convex optimization problems. Using a slack variable z, we can write the problem as $Ax + z = b$, with $z \geq 0$. This is now in ADMM form, but with the additional constraint that $z \geq 0$. The additional constraint primarily affects the z update step in this case, as we need to ensure a non-negative z is chosen. Considering $g(z) = \lambda||z||_1$, which can be solved in general via (10), we can modify the solution to select positive values for z. For example, we can project (10) into the non-negative orthant by setting negative components to zero, for $v = Ax^{(k+1)} + u^{(k)}$.

Convergence Checking

The paper by He and Yuan in [12] constructs a special norm derived from a variational formulation of ADMM. Suppose you have a certain encoding of ADMM's iterates in the form of

$$w^i = \begin{bmatrix} x^i & z^i & \rho u^i \end{bmatrix}^T \tag{11}$$

where x^i, z^i, and u^i are iteration i's results for x, z, and scaled dual variable u, and ρ is the step size. Let the matrix H be defined as follows:

$$H = \begin{bmatrix} G & 0 & 0 \\ 0 & \rho B^T B & 0 \\ 0 & 0 & I_m/\rho \end{bmatrix} \tag{12}$$

where $B \in \mathbb{R}^{m \times n_2}$ is the same matrix as from the ADMM constraints, I_m is the identity of size m, and $G \in \mathbb{R}^{n_1 \times n_1}$ is a special matrix dependent on the variational problem ADMM is trying to solve. Then, as shown in [12], $\{||w^i - w^{i+1}||_H^2\}$ are monotonically decreasing for all iterations i:

$$||w^i - w^{i+1}||_H^2 \leq ||w^{i-1} - w^i||_H^2 \tag{13}$$

The matrix G is actually irrelevant in this computation; it ends up disappearing anyway in the end result. Thus, you do not need to know G to compute the H-norms. Since these norms must be monotonically decreasing for ADMM to converge, evaluating and checking these norms every iteration and checking the condition (13) is a good strategy to check that ADMM is actually converging. For example, say you are given the constraints for an ADMM problem and the proximal operators that correspond to them. If the proximal operators are incorrect, or the constraints do not match what the proximal operators compute, then it is not expected ADMM will actually converge. In such a case, checking (13) will tell you immediately if there's an issue, and the algorithm can be stopped, reporting an error. This avoids needlessly running what could be long and expensive operations that will not converge anyway.

Since the H-norms evaluations are not free (though they can be evaluated very quickly), this would likely be an option the user specifies when they are initially trying out proximal operators

for a problem. Also, as there is the concern of round-off error, the condition (13) would likely be checked in terms of relative error to some specified (or default) tolerance.

Stopping Conditions

By [1], we can define the primal (p) and dual (d) residuals in ADMM at step $k+1$ as:

$$p^{k+1} = Ax^{k+1} + Bz^{k+1} - c \tag{14}$$
$$d^{k+1} = \rho A^T B(z^{k+1} - z^k) \tag{15}$$

The primal residual is trivial. However the dual residual stems from the need to satisfy the first dual optimality condition $\nabla f(x) + A^T y = 0$. More generally, for subgradients ∂f and ∂g for f and g, and since x^{k+1} minimizes $L_\rho(x, z^k, y^k)$:

$$\begin{aligned}
0 &\in \partial f(x^{k+1}) + A^T y^k + \rho A^T(Ax^{k+1} + Bz^k - c) \\
&= \partial f(x^{k+1}) + A^T y^k + \rho A^T(r^{k+1} + Bz^k - Bz^{k+1}) \\
&= \partial f(x^{k+1}) + A^T y^k + \rho A^T r^{k+1} + \rho A^T B(z^k - z^{k+1}) \\
&= \partial f(x^{k+1}) + A^T y^k + \rho A^T B(z^k - z^{k+1})
\end{aligned}$$

So, one can say $d^{k+1} = \rho A^T B(z^{k+1} - z^k) \in \partial f(x^{k+1}) + A^T y^k$, and the first dual optimality condition is satisfied by (15). It is reasonable to say that the stopping criteria is based on some sort of primal and dual tolerances that can be recomputed every iteration (or they could be constant, but adaptive ones are generally better). I.e., $||p^k||_2 \leq \epsilon^{pri}$ and $||d^k||_2 \leq \epsilon^{dual}$. There are many ways to choose these tolerances. One common example, described in [1], where $p \in \mathbb{R}^{n_1}$ and $d \in \mathbb{R}^{n_2}$:

$$\epsilon^{pri} = \sqrt{n_1}\epsilon^{abs} + \epsilon^{rel} \max(||Ax^k||_2, ||Bz^k||_2, ||c||_2) \tag{16}$$
$$\epsilon^{dual} = \sqrt{n_2}\epsilon^{abs} + \epsilon^{rel}||A^T y^k||_2 \tag{17}$$

where ϵ^{abs} and ϵ^{rel} are chosen constants referred to as *absolute* and *relative* tolerance. In practice, the absolute tolerance specifies the precision of the result, while the relative tolerance specifies the accuracy of the dual problem in relation to the primal.

Another option for stopping conditions would be the H-norms used in convergence checking. The paper by He and Yuan in [12] also shows that the convergence rate of ADMM satisfies:

$$||w^k - w^{k+1}||_H^2 \leq \frac{1}{k+1}||w^0 - w^*||_H^2 \tag{18}$$

for all solutions w^* in the solution space of the problem. As a solution w^{k+1} for an ADMM problem must satisfy $||w^k - w^{k+1}||_H^2 = 0$ (an extra iteration produced no difference), this implies that

$$||w^k - w^{k+1}||_H^2 \leq \epsilon \tag{19}$$

for some small, positive value ϵ is a good stopping condition for ADMM as well.

Parallelizing ADMM

The advantage of ADMM over the methods discussed in the background section is that it has the desired robustness and speed, but doesn't sacrifice the ability to parallelize. But how exactly could a distributed ADMM work?

We can let $A = \text{I}$, $B = -\text{I}$ and $c = 0$ to set the constraint as $x = z$. As a result, with a separable f and x, we can minimize f_i and require each $x_i = z$ at the end. Thus, we optimize each x_i and aggregate their solutions to update our z, so our Augmented Lagrangian looks like:

$$L_\rho(x, z, y) = \sum_i \left(f_i(x_i) + y^T(x_i - z) + \frac{\rho}{2}||x_i - z||_2^2\right) \tag{20}$$

where each x_i is a decomposed vector from the original x.

In general, ADMM actually solves a single convex function, which is decomposed into $f(x)+g(z)$. The function $g(z)$ is ideally chosen as something artificial and easy to solve, like the proximal mapping of the l_1 norm with the solution in (10). Thus, the above distributed scheme works for many uses of ADMM. More complicated schemes can be developed should the need arise. See Algorithm 3 for a general distributed ADMM algorithm. Parallelization with ADMM, along with other similar MM methods is discussed in further detail in [3] and [13].

Algorithm 3 Distributed ADMM (Scaled and Unscaled Dual)

procedure DISTRIBUTEDADMM(A, B, c, ρ, OBJECTIVE, ARGMINX, ARGMINZ, STOPCOND)
  Set x, z and u to some initial value.
  while STOPCOND(x, z, u, A, B, c) $\neq 1$ **do**
    for parallel machine labeled by index i **do**
      $x_i :=$ ARGMINX$(f_i(x) + (y_i)^T(x - z) + \frac{\rho}{2}||x - z||_2^2)$ (distributed x update)
    $z :=$ ARGMINZ$((y_i)^T(x_i - z) + \frac{\rho}{2}||x_i - z||_2^2) = \frac{1}{n}\sum_{i=1}^n (x_i + \frac{1}{\rho}y_i) = \frac{1}{n}\sum_{i=1}^n (x_i + u_i)$
    for component index i **do**
      $y_i := y_i + \rho(x_i - z) = \frac{1}{\rho}y_i + x_i - z = u_i + x_i - z$
  return (x, z, OBJECTIVE(x, z))

Unwrapped ADMM with Transpose Reduction

Consider the problem: $\min(g(Dx))$, where g is convex and $D \in \mathbb{R}^{m \times n}$ is a large, distributed data matrix. In "unwrapped" ADMM form, described in [14], this can be written as:

$$\min(g(z)) \text{ subject to } Dx - z = 0 \tag{21}$$

The z update is typical, but a special x update can be used for distributed data:

$$x^{k+1} = D^+(z^k - u^k) \tag{22}$$

where $D^+ = (D^T D)^{-1} D^T$ (known as the pseudo-inverse). If g is a decomposable function, each component in z update is decoupled. Then, an analytical solution or look-up table is possible. As $D = [D_1^T, ..., D_n^T]^T$, x update can be rewritten as:

$$x^{k+1} = D^+(z^k - u^k) = W \sum_i D_i(z_i^k - u_i^k) \tag{23}$$

Note that $W = (\sum_i D_i^T D_i)^{-1}$. Each vector $D_i(z_i^k - u_i^k)$ can be computed locally, while only multiplication by W occurs on central server. This technique describes another way to approach parallelizing ADMM in an efficient way, for certain problems.

Additionally, one can combine the strategy of Unwrapped ADMM with that of Transpose Reduction. Consider the following problem:

$$\min_x \left(\frac{1}{2}||Dx - b||_2^2 + H(x) \right) \tag{24}$$

for some penalty term $H(x)$. We know that:

$$\frac{1}{2}||Dx - b||_2^2 = \frac{1}{2}x^T(D^T D)x - x^T D^T b + \frac{1}{2}||b||_2^2 \tag{25}$$

With this observation, a central server needs only $D^T D$ and $D^T b$. For tall, large D, $D^T D$ has much fewer entries. Furthermore, we can see that $D^T D = \sum_i D_i^T D_i$ and $D^T b = \sum_i D_i^T b_i$. Now, each server needs to only compute local components and aggregate on a central server. This is known as Transpose Reduction. Once the distributed servers compute $D^T D$ and $D^T b$, we can continue on and solve the problem with Unwrapped ADMM, as described before. This strategy applies to many problems; thus, many ADMM solvers for these problems can be optimized and written in an efficient, distributed method.

Fast ADMM

There exist faster converging strategies for ADMM, under certain conditions. The most interesting of these, due to their generality, are discussed in [15]. First, we will discuss *Fast ADMM* (**FADMM**), a technique to speed up convergence in the case of functions f and g being *strongly convex*. If a function h is strongly convex, then there exists a constant σ_h such that for every valid $x, y \in \mathbb{R}^n$ in its domain:

$$\langle p - q, x - y \rangle \geq \sigma_h ||x - y||^2 \tag{26}$$

where $p \in \partial h(x)$ and $q \in \partial h(y)$. According to [15], the convergence for this type of problem can benefit by using a predictor-corrector method where we predict a quadratically scaled progression of the z and u variables in the next iteration, using this for the x, z and u updates. The prediction takes advantage of the strong convexity, which guarantees that a function lies above its local quadratic approximation. This strategy is known as FADMM and the corresponding algorithm is shown in Algorithm 4.

As per the convergence proof in [15], FADMM's predictors are used in the dual optimality condition, and thus the dual residual for stopping conditions in (15) must swap z^k out for the predicted value $\hat{z}^{k+1}$. Thus:

$$d^{k+1} = \rho A^T B(z^{k+1} - \hat{z}^{k+1}) \tag{27}$$

Algorithm 4 Fast ADMM (FADMM)

procedure FADMM(A, B, c, ρ, OBJECTIVE, ARGMINX, ARGMINZ, STOPCOND)
  Set $x^{(0)}$, $z^{(0)}$ and $u^{(0)}$ to some initial value.
  Set $\hat{z}^{(0)} = z^{(0)}$, $\hat{u}^{(0)} = u^{(0)}$ and $\alpha_1 = 1$ (Predictors)
  while STOPCOND(x, z, u, A, B, c) $\neq 1$ **do**
    $x^{(k+1)} := \arg\min_x(L_\rho(x, \hat{z}^{(k)}, \hat{u}^{(k)}))$ (Update x with predicted values)
    $z^{(k+1)} := \arg\min_z(L_\rho(x^{(k+1)}, z, \hat{u}^{(k)}))$ (Update z with predictor $\hat{u}$)
    $u^{(k+1)} := \hat{u}^{(k)} + (Ax^{(k+1)} + Bz^{(k+1)} - c)$ (Update u with predictor $\hat{u}$)
    $\alpha_{k+1} := \frac{1+\sqrt{1+4\alpha_k^2}}{2}$ (Predict a quadratic scaling)
    $\hat{z}^{(k+1)} := z^{(k)} + \frac{\alpha_k - 1}{\alpha_{k+1}}(z^{(k)} - z^{(k-1)})$ (Predict by scaled quadratic differences)
    $\hat{u}^{(k+1)} := u^{(k)} + \frac{\alpha_k - 1}{\alpha_{k+1}}(u^{(k)} - u^{(k-1)})$ (Predict by scaled quadratic differences)
  return (x, z, OBJECTIVE(x, z))

Accelerated ADMM

Unfortunately, strong convexity in both functions f and g often means that the problem is solvable by hand (we can compute exact gradients), thus we don't actually need ADMM to solve it. If not that, then oftentimes it can be solved by other, faster means. Thus, FADMM is something of a toy algorithm, used on toy problems. However, [15] shows that FADMM can be generalized to work quite well on *weakly convex* problems, where f and g are at least weakly convex. A function h is weakly convex if for all valid $x, y \in \mathbb{R}$ in its domain, for all $a \in [0, 1]$:

$$f(ax + (1-a)y) \leq af(x) + (1-a)f(y) \tag{28}$$

The observation that allows this generalization is that weakly convex functions tend to have the properties of strongly convex functions over large portions of their domain, apart from where they fail to meet the definition of strongly convex. Thus, we can often use FADMM with no troubles, unless we hit one of trouble spots, where we will suddenly stop converging. If this happens, we can simply reset FADMM on the next iteration, and start the quadratic predictions over again. This gets us past the trouble area, at the cost of resetting the quadratic acceleration. To detect a lapse in convergence, we can define a parameter η that gives a minimum relative difference in residuals before we decide to restart due to slow convergence. A special residual is defined for this, by [15]:

$$d^k = 1/\rho||u^k - \hat{u}^k||^2 + \rho||B(z^k - \hat{z}^k)||^2 \tag{29}$$

This residual combines the primal (first term) and dual (second term) errors between the corrected and predicted u and z. The dual error is not multiplied by A^T as in (15) since it doesn't have any meaning here; we are focusing on the error contribution from corrected / predicted z only, which involves only the coefficient matrix B from the ADMM constraint.

We refer to the above strategy as *Accelerated ADMM* (**AADMM**). Note that ADMM always requires f and g to be at least weakly convex. Thus, we can always use AADMM. However, there are several drawbacks. Firstly, we need to select a proper η parameter; a value strictly between 0 and 1. Empirical evidence from [15] shows that $\eta = 0.999$ is a conservatively safe choice. Secondly, we have

Algorithm 5 Accelerated ADMM (AADMM)

procedure FADMM(A, B, c, ρ, OBJECTIVE, ARGMINX, ARGMINZ, η)
  Set $x^{(0)}$, $z^{(0)}$ and $u^{(0)}$ to some initial value.
  Set $\hat{z}^{(0)} = z^{(0)}$, $\hat{u}^{(0)} = u^{(0)}$ and $\alpha_1 = 1$ (Predictors)
  while $d_k < \epsilon^{reset} d_{k-1}$ **do** (Special stopping condition)
    (FADMM corrector updates)
    $x^{(k+1)} := \arg\min_x(L_\rho(x, \hat{z}^{(k)}, \hat{u}^{(k)}))$
    $z^{(k+1)} := \arg\min_z(L_\rho(x^{(k+1)}, z, \hat{u}^{(k)}))$
    $u^{(k+1)} := \hat{u}^{(k)} + (Ax^{(k+1)} + Bz^{(k+1)} - c)$
    $d_{k+1} = 1/\rho||u^{(k)} - \hat{u}^{(k)}||_2^2 + \rho||B(z^{(k)} - \hat{z}^{(k)}||_2^2$ (Residual for this iteration)
    if $d_k < \eta d_{k-1}$ **then** (Still converging; proceed with FADMM predictor updates)
      $\alpha_{k+1} := \frac{1+\sqrt{1+4\alpha_k^2}}{2}$
      $\hat{z}^{(k+1)} := z^{(k)} + \frac{\alpha_{k-1}-1}{\alpha_{k+1}}(z^{(k)} - z^{(k-1)})$
      $\hat{u}^{(k+1)} := u^{(k)} + \frac{\alpha_{k-1}-1}{\alpha_{k+1}}(u^{(k)} - u^{(k-1)})$
    else (Hit a trouble-spot; restart FADMM predictions)
      $\alpha_{k+1} := 1$
      $\hat{z}^{(k+1)} := z^{(k-1)}$
      $\hat{u} := u^{(k-1)}$
      $d_{k+1} := d_k/\eta$
  return (x, z, OBJECTIVE(x, z))

no provable convergence guarantees with AADMM; the problem and η specific causes of resetting are unpredictable - only empirical evidence shows that we always converge for reasonable η. Lastly, the reliance on resetting destroys the guarantees of the usual stopping conditions described earlier. We even lose the H-norm squared residual's monotonically decreasing property. Thus, convergence can only be reliably measured by (29). The AADMM algorithm is shown in 5.

Implementing A General ADMM Function

The first step in our solver library is to have a generalized ADMM solver, which will be used to solve more specific problems. Our goal was to make this solver as general as possible; i.e., it should only be given proximal operator functions for f and g, and any customization to the algorithm. It should support all of the functionality from the previous section on ADMM, in a seamless way. In this section, we describe how this functionality is implemented and tested.

General Functionality

The generalized ADMM function, implemented in MATLAB, accepts exactly 3 inputs: the proximal operator for f, called `xminf`, the same for g, called `zming`, and a struct called `options` which contains fields that specify custom options for the execution of ADMM. Likewise, the output is a struct `results` that contains all the results for the execution of ADMM. This includes the final, optimal values for x, z, and u, the values of these at each iteration (in matrices/higher dimensional arrays), the values of primal and dual residuals or H-norms at each iteration, predictors/residuals for Accelerated/Fast ADMM at each iteration, ρ values at each iteration for adapative step-sizes, etc. All this information is recorded, if necessary, or desired. See the user manual for more details on what is recorded.

Similarly, the user manual contains details on what fields can be set in `options`. These include starting values for x, z and u, rho, parameters for residuals and stopping conditions (including which ones to use), parameters for Accelerated/Fast ADMM, constraint data (A, B and c), the dimensions of these (rows m in c, columns nA and nB in A and B), and more. These options all have default values. Detection of existing fields and assignment of default values is done via a useful function `setopts` that selects fields and returns their values or defaults depending on if these fields exist in `options`.

Proximal Operators

Proximal operators `xminf` and `zming` are function handles that must accept 4 inputs. In vanilla ADMM, these inputs are as follows, respectively: x, z, u, and ρ. These are self-explanatory. Demanding these inputs for a proximal operator are somewhat counter-intuitive; typically a proximal operator is expressed as accepting a vector v and proximal parameter t. Additionally, proximal operators for a variable a normally don't even use the current value of a! The reasoning behind using 4 inputs is as follows.

- First of all, the user needs to know A, B and c (from ADMM's constraint) in advance, thus it is redundant to, for example, compute and pass $v = Bz - c + u$ to `xminf` for an x update; the user can do it themselves when they define `xminf` in their local function (B and c can be globally available in their local function). Perhaps the user doesn't even need to multiply B as it is equal to the identity, or c is the zero vector and doesn't need to be added. Thus, the inputs x, z and u are sufficient, but still allow the user to make efficient improvements to their proximal operators.
- Secondly, the parameter t often depends on ρ. If it does not, then it depends on some other information that is unknown to general ADMM, and is encapsulated in the functionality of the proximal operator function handle.

- Third, if we are to use adaptive step sizes, and oftentimes proximal operators need to know step size ρ, we need to pass the current iteration k's ρ_k to the proximal operators.
- Lastly, it's possible that certain proximal operators cannot be computed precisely, but are estimated or utilize a predictor/corrector scheme. Since the only unique information that ADMM knows but the proximal operator doesn't is x, z, and u, and estimation techniques for a proximal operator (say, for f) might be implicit (e.g., for f, they need the current x for an implicit definition), it makes sense to pass all this information.

Relaxation and Proximal Operators

There exists a common technique for improving convergence called *relaxation*, described in [1]. Relaxation replaces the term Ax^{k+1} in the updates for ADMM with:

$$Ax^{k+1} := \alpha Ax^{k+1} - (1-\alpha)(Bz^k - c) \tag{30}$$

where A, B and c are the constraint parameters for ADMM, and α is a user-provided parameter referred to as the relaxation parameter. If $\alpha > 1$, this is called *over-relaxation*, and if $\alpha < 1$, this is called *under-relaxation*. The value $\alpha \in (0, 2)$. Over-relaxation is the more common technique, with typical values of α being in the range [1.5, 1.8].

This technique interpolates Ax^{k+1} with $-(Bz^k - c)$, which by the optimality conditions would mean that you allow g to contribute or detract from f's contribution to the ADMM constraint, without breaking the constraint. Tying these values to each other in such a way may provide attractive convergence properties, especially with over-relaxation, according to [11] and numerical results from Eckstein in many other papers, mentioned in the over-relaxation section in [1].

This poses a drawback in proximal operators having the 4 inputs x, z, u, and ρ, as opposed to v and t, discussed in the prior section: we can't perform relaxation. To get around this, if the user consciously makes the decision to perform relaxation by specify a relax field in `options` (which holds the value of α), we require that their proximal operators be designed to accept Ax, z, u, and ρ. The value Ax here is the relaxed product Ax^{k+1}, given by (30).

A Model Problem For Testing

Since ADMM requires a problem to solve, we can only test it by using some kind of model problem, for which we can already find an analytic solution and for which the proximal operators are very easy to compute (no room for error). We choose the following model problem:

$$\min_x (1/2||Px - r||^2 + 1/2||Qx - s||^2) \tag{31}$$

where $P, Q \in \mathbb{R}^{m \times n}$ and $r, s \in \mathbb{R}^m$. Not only are the proximal operators for this problem easy to figure out, but one can solve this problem exactly by hand, allowing us to compare ADMM's solution to the exact solution. To compute the exact solution, we simply take the gradient of (31), set it equal to 0, and solve for x:

$$\begin{aligned}\nabla_x(1/2||Px-r||^2+1/2||Qx-s||^2) &:= 0\\ P^T(Px-r)+Q^T(Qx-s) &:= 0\\ P^TPx+Q^TQx-(P^Tr+Q^Ts) &:= 0\\ (P^TP+Q^TQ)x &:= P^Tr+Q^Ts\end{aligned}$$

We see that then the minimizing x is:

$$x=(P^TP+Q^TQ)^{-1}(P^Tr+Q^Ts) \tag{32}$$

To efficiently compute this, we perform a MATLAB system solve with the backslash operator, avoiding explicit computation of the inverse.

In ADMM form, we write problem (31) as:

$$\min(1/2||Px-r||^2+1/2||Qz-s||^2), \text{ subject to } x-z=0 \tag{33}$$

where $f(x)$ is the first term and $g(z)$ is the second. The Augmented Lagrangian is thus:

$$L_\rho(x,z,u)=(1/2||Px-r||^2)+(1/2||Qz-s||^2)+\rho/2||x-z+u||^2 \tag{34}$$

To find the proximal operator for f and g here, we proceed as before and use gradients. For f, we take the gradient of (34) for x:

$$\begin{aligned}\nabla_x L_\rho(x,z,u) &:= 0\\ P^T(Px-r)+\rho(x-z+u) &:= 0\\ P^TPx+\rho x-P^Tr-\rho(z-u) &:= 0\\ (P^TP+\rho\mathbb{I})x &:= P^Tr+\rho(z-u)\end{aligned}$$

Thus, our proximal operator for f is equal to computing:

$$x=(P^TP+\rho\mathbb{I})^{-1}(P^Tr+\rho(z-u)) \tag{35}$$

Following the same procedure for g, but with a gradient on z, we get the proximal operator:

$$z=(Q^TQ+\rho\mathbb{I})^{-1}(Q^Ts+\rho(x+u)) \tag{36}$$

We will describe how to compute (35) and (36) efficiently in a later section about implementing ADMM solvers for specific problems. The same goes for testing and validation results for the Model problem itself.

Implementing And Testing Convergence Checking

From (13), we can see that the following condition should hold for a converging ADMM:

$$||w^i - w^{i+1}||_H^2 - ||w^{i-1} - w^i||_H^2 \leq 0 \tag{37}$$

Taking machine precision into account, we can replace the RHS of the equality in (37) with some small, constant positive ϵ_w, on the order of machine error ϵ, e.g. $\epsilon_w = 10^{-16}$. If the difference in monotonically decreasing differences of w's between iterations is greater than such a value, then (37) shouldn't hold, even with error from floating point operations. This essentially detects a non-converging ADMM setup; which could stem from errors in proximal operators. If the user makes a mistake, they don't have to wait hundreds of unnecessary iterations or expensive evaluations of proximal operators - the lack of convergence would be detected immediately.

For every successive iteration k, $||w^{k-1} - w^k||_H^2$ should get smaller and smaller. See Figure 1.(a) for an example of these monotonically decreasing norm evaluations. This is output from the example script `hnormdemo.m`, that creates a demo problem to evaluate these norms on. The relative convergence threshold (relative difference of 10^{-6} between norm evaluations) occurs at the blue line, at which point the model solver terminates (around 530 iterations).

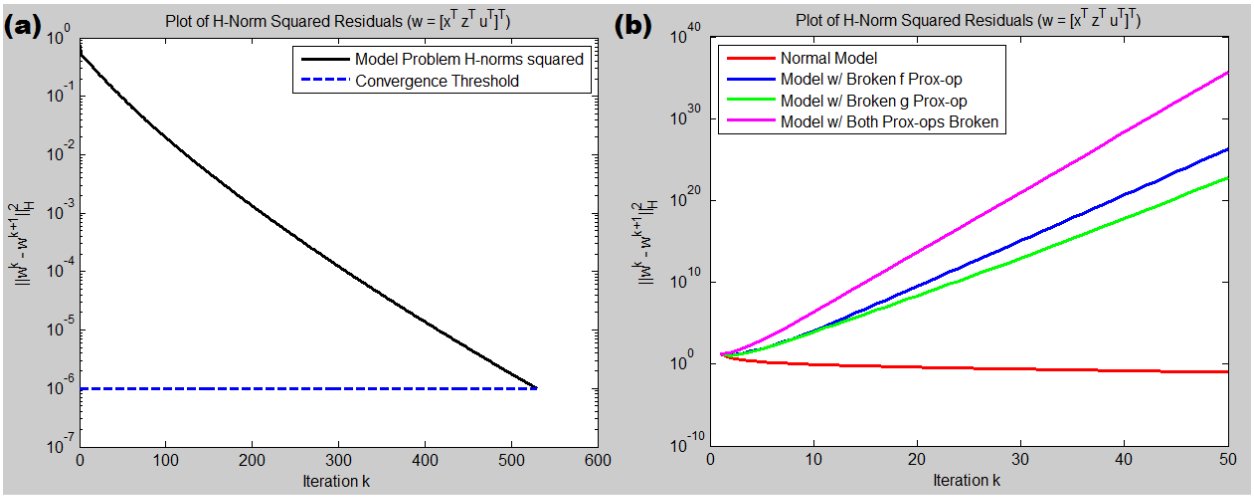


Figure 1: **(a)** H-norm squared values across differences in w for a random Model problem. **(b)** A comparison of H-norm squared values across differences in w for the model solver, the model solver with a broken f proximal operator, broken g prox-op, and both broken. **(c)** Output from ADMM with convergence checking on for each of the cases in (b). This shows the iteration ADMM stopped due to detecting non-convergence in H norm squared values.

To test the functionality of convergence checking, we use the example function `convergencechecking.m`, which creates a random problem for the Model solver to solve, and solves it using the normal Model solver, a solver with a "broken" proximal operator for g, the same for a broken proximal operator f, and a solver with both of these errors in f and g's proximal operators, outputting a graph comparing the H-norm squared evaluations between them. Figure 1.(b) shows the resulting graph for the first 50 iterations of when the botched prox-op for f is (35) with $(z+u)$ instead of $(z-u)$, and the botched prox-op for g is (36) with a negative $\rho(x+u)$ term instead of a positive one. Note that if convergence checking is not on, the botched cases would perform the default maximum number of iterations (1000) due to stopping conditions never being reached, as the botched cases do not converge.

We see that each of these cases have non monotonically convergent H-norm squared evaluations nearly immediately, for mistakes in the proximal operators that could have occurred through simple typos in the code. We expect from these results that ADMM will catch these problems near instantly. Figure 1.(c) shows the output for when convergence checking would stop ADMM's execution due to detecting non-convergence (single precision machine error 10^{-16} as the tolerance ϵ_w), in each case respective. For the botched g prox-op and the botched f prox-op cases, non-convergence is detected on the second convergence check (iteration 3, where checks start occurring at iteration 2). When both were botched, non-convergence was detected on the immediate first check. Thus, Figure 1.(b)'s broken cases would not have made it past iteration 3, on the second check, saving the user at least 997 needless iterations on a non-converging ADMM setup.

We test convergence checking in general ADMM by purposely botching proximal operators in this way, for the Model problem, in various ways. We also do this for other solvers and get similar results. The results indicate that not all erroneous changes cause non-convergence in ADMM. For example, botching the proximal operator for g in the model problem by adding a negative of the identity, instead of the positive, does not break convergence if the random data in the correct proximal operator for f happens to perform a larger step towards the correct solution for the x-update. This negatively impacts convergence, but convergence does occur. Thus, convergence checking will not catch ALL errors in proximal operators; only ones that break convergence. However, that is their purpose, so this limitation does not affect the usefulness of convergence checking in general. Convergence checking should always be used by a user who is programming and testing their own solver, as it is cheap and can save them from crashing MATLAB or waiting very long for non-converging executions of the solver. The only exception should be if they use Accelerated ADMM in their solver, as then we do not have the monotonically convergent guarantee.

Implementing Stopping Conditions

We have already outlined a couple of stopping conditions for ADMM in the section on Stopping Conditions. Namely, we have two strategies for general stopping conditions:

1. The primal and dual residuals (14) and (15), along with adaptive tolerances for each, (16) and (17). We require that both conditions $||p^k||_2 \leq \epsilon^{pri}$ and $||d^k||_2 \leq \epsilon^{dual}$ be true for ADMM to be considered converged.

2. The H norm squared evaluations from (19). We require the relative condition, for some ϵ_H, for iteration $i \geq 2$, that:

$$||w^{i-1} - w^i||_H^2 - ||w^{i-2} - w^{i-1}||_H^2 \leq \epsilon_H ||w^{i-2} - w^{i-1}||_H^2 \tag{38}$$

Tests of the convergence condition (19) have shown that the above formulation is the most convenient and general way of measuring relative error using H-norms.

In summary, the first strategy requires selection and tuning of ϵ^{pri} and ϵ^{dual} to get the precision and accuracy desired in the resulting solution. However, the condition can be evaluated at any iteration. The second condition is generally more conservative in measuring error, but requires choosing only one parameter, ϵ_H. However, it can't determine convergence until the second iteration due to the reliance on the w values from the previous two iterations (w^0 computed from starting values of x, z, and u).

We use the field `stopcond` in the options struct provided to ADMM for allowing the user to decide which stopping condition to use. This field can have the string values "standard", to use the first stopping condition, "hnorm" to use the second stopping condition, "both" to stop only when both conditions are met, and "either" to stop if either condition is met. The default value is "standard", if the user does not provide a choice for their stopping condition. For both conditions, we have additional fields for the parameters for each condition (`abstol` and `reltol` for ϵ^{pri} and ϵ^{daul}, and `Hnormtol` for ϵ_H). They have their own default values if these are not specified.

Stopping conditions are extensively tested first by the Model problem solver, for correctness in functionality. This is done by hand, changing values for fields in the options struct and observing the effect on ADMM's termination, to see if it matches the setting. Figure 1(a) shows an example with the Model problem of the second stopping condition in action, with the dashed line being the final value equal to (38) solved for ϵ_H that was required to break the condition. Once the stopping condition tests satisfied all cases in the Model problem, we expect that they will work for all other solvers. This is supported by validation of results in those solvers and extensive testing through tester functions for every solver. We will see more examples of the H-norm stopping condition, and the standard one, in further sections, primarily where we show results for ADMM solvers of other problems.

Implementing Parallel ADMM

Algorithm 3 gives a clear method of implementing a Distributed form of ADMM. For Parallel ADMM, we can simply use this strategy on local workers (cores and threads) on the machine via the MATLAB command `parfor`, which parallelizes a for loop's execution over a number of workers. For a parallel x-update, ADMM defines a function that runs a `parfor` loop much as in steps 4-5 of Algorithm 3. Likewise, there are such parallel functions for the z-update and u-update that can be executed.

For a user to enable Parallel ADMM, they need to fill in the `parallel` field of the options struct. The possible values are strings: "xminf", for parallelizing only the proximal operator for f; "zming", for parallelizing only the proximal operator for g; "both" for performing parallel versions of both, and "none" for not performing any parallel steps. The default value is "none", if the user does not populate the `parallel` field. The u update is always parallel if the user specifies any parallel functionality, as it would be a waste of resources to not distribute the workload of the u update across workers. If the user requires an alternate update for u (some parallelized problems that ADMM solves have very unique u-update steps), there also exists a field `altu`, which specifies a function handle for an alternate u update that the user creates themselves. Furthermore, is the user requires ADMM to perform some preprocessing of data before going into the main update loop, there exists a field `preprocess` that can be set to hold a function handle that will be executed with no inputs.

Parallel ADMM slices up the data in parallel update steps to distribute the workload across workers. By default, it will split up the rows of the problem to solve as equally as it can among the available workers in the parallel pool. However, in the case of a single parallel proximal operator, there exists a field `slices` in the options struct that the user can populate with a vector that contains the number of rows to assign worker j at component j. If the user provides a single scalar value, the data is sliced up rows of that size, to the degree possible. Since the slices among the updates can actually differ, then for the case of both proximal operators being parallel, the user can instead provide a cell containing two column vectors of slice sizes, for f and g, respectively, as the 1st and 2nd components of the cell. Each of these vectors follows the rules as stated for the single parallel proximal operator cases. In general, providing a scalar 0 instead of a vector for slices will revert Parallel ADMM to the default behavior of equally balancing the workload across the workers.

For Parallel ADMM, we require that proximal operators add an additional parameter i as input, which should relay to the proximal operator which step i of the `parfor` loop is being executed. Since `parfor` can execute the proximal operator steps in non-sequential ways, we also require that proximal operators make the slice of data needed to perform the parallel update available to the worker via parameter i, when using Parallel ADMM. If the user wishes to use `parfor` inside their proximal operators, as opposed to letting ADMM handle it, the user has the ability to do this in general. The aforementioned options struct field "parallel" can enable them to choose which proximal operator should be parallelized by ADMM, and which shouldn't be.

These ingredients in the options struct allow the user to fully customize their parallel execution of ADMM; either to be explicitly done by their own `parfor` loops inside of their proximal operators, or by ADMM, and gives the user control over how to distribute the data among the parallel workers. It also is built to require minimal input from the user to function; thus, a user can simply set `parallel` to "both" in the options struct and ADMM will automatically handle everything, if their proximal operators are built to handle the additional slice i parameter.

One must note that there is a general overhead to initializing the parallel pools in MATLAB. Depending on the machine and the number of cores, this can take a few seconds to half a minute. It is a one-time overhead, however, as parallel pools stay open for as long as needed in MATLAB, depending on the customizable timeout parameter. We test and validate a parallel LASSO solver in future sections to test and validate the functionality of Parallel ADMM, as we need a specific problem to solve. Additionally, the Unwrapped ADMM solver also uses parallel elements, and we test and validate the functionality of that later as well.

Implementing Fast/Accelerated ADMM

Algorithms 4 and 5 give a pretty roadmap of how to implement Fast and Accelerated ADMM. We simply need to weave it together with vanilla ADMM's execution in order to have a generalized ADMM that handles these faster convergent variants of ADMM. Note that Fast ADMM and Accelerated ADMM share the same basic predictor/corrector scheme and variables, but Accelerated ADMM requires a bit more for the reset stage, its special residual, and additional parameters. In our ADMM method, we allow the user to enable faster ADMM by the field `fast` in the options struct, which is a binary value. If true, ADMM will perform either Fast or Accelerated ADMM. The distinction between which of these is done by another field `fasttype`, which can hold the value "weak" or "strong", which refer to the convexity of the problem ADMM is trying to solve. By default, `fasttype` will always be "weak", as all problems solvable by ADMM are weakly convex. If

the user really wants to try a toy problem with Fast ADMM, however, they can explicitly perform it.

We implement the three varieties of ADMM (vanilla, fast, accelerated) by a case/if selection on the steps that differ between them. The x, z and u updates have these cases on them, to update according to the variant of ADMM being executed. The computation of the special residual in (29) is only performed and recorded if performing Accelerated ADMM, along with the reset steps. For Fast ADMM, we swap out the dual residual in standard stopping conditions for (27), and everything else stays the same, in terms of stopping conditions. However, for Accelerated ADMM, the stopping conditions are overrided and the stopping condition in Algorithm 5 is used instead. Thus, for Accelerated ADMM, we have additional fields `dvaltol` and `restart`, which specify ϵ^{reset} and η in Algorithm 5, respectively. By default, `restart` is set to 0.999, as recommended by [15].

Although we will not prove it here, the Model problem is strongly convex in both f and g. Thus, we can test FADMM and AADMM both using the Model problem solver. Furthermore, we can validate the results by comparing the convergences between the vanilla, accelerated and fast variants, which should be ranked from slowest to fastest converging, in this way. We show the results for this in a later section, where we describe how each solver works, specifically in the Model problem section.

General Structure of the ADMM Library

With the implementation of the general ADMM function out of the way, we are poised to describe the general structure of the ADMM library that was implemented in MATLAB. Describing this will make the testing and validation results for the solvers in the next major section easier to understand. The library is composed of 4 major parts; the main/root directory, the solvers sub-directory, the testers sub-directory, and the examples sub-directory. We describe the contents of each of these in the following sections. In general, the main/root directory has the most essential files, the solvers directory has all of the solvers for general ADMM problems, the testers directory has all the testing and validation code for each solver, and the examples directory has assorted examples of how to use the ADMM library, some of which also double for testing and validation of certain parts of general ADMM's functionality.

The Main Directory

In the top-most level of the software library, we have the central files necessary for the overall functioning of the library. This, of course, includes our generalized ADMM method described in the previous section, as a single file `admm.m`, as well as the sub-directories for solvers, testers and examples. The main directory also has two complementary files, `setuppaths.m` and `removepaths.m`, used to set up and remove paths.

The function `setuppaths` is used to set up local paths to the other directories in the library; the solvers, testers, and examples sub-directories, so that the user can run any of the functions in the library from any other directory. The user simply runs this function and the paths will be set up, assuming the names of ADMM's sub-directories have not been altered. The function can also be given a binary `quiet` parameter, which specifies whether or not it should suppress output of a message stating that the paths have been set up correctly. This function also defines and saves a global variable `setup`, which other functions in the library reference to see if the paths in the library have been set up. Paths are persistent until the polar opposite function `removepaths` is executed, which removes any paths that have been set up for the library. It also has a optional `quiet` parameter as `setuppaths`.

The main directory also contains a file `showresults.m`. This is a function, `showresults`, that will output information about a results struct returned by ADMM, or a test struct returned by a tester function. This function will also create plots about the execution of ADMM or a solver, that show salient information. This is, essentially, a convenience file for ADMM users to quickly visualize what happened when they executed `admm` or a solver.

Next, the main directory has a file `errorcheck.m`, which contains function `errorcheck`, a generalized error checker of inputs to other functions. The purpose of this is to avoid redundant error checks in solvers by simply calling this function to check if an input is valid or can be massaged into a usable form (which is given as output), or create an error message if something is wrong. This function accepts an argument to error check, a string containing the type of check to perform, another string containing the name of this argument (used in error messages), and an optional options struct that is used to customized the error checker in certain scenarios. The error-check function can check if something is a matrix (square, or fat, or skinny), or if something is a vector (a row vector or column vector, massaging it into the correct type if it isn't), or if something is a number (positive real, nonnegative real, integer), or if something is a struct, or even more specific

checks, such as if something matches the form of the `slices` parameter in ADMM, creating sliced data or information about slices from data in the options struct.

Finally, the main directory has the all-important file `getproxops.m`. The function `getproxops` accepts a string, describing what problem to set up proximal operators for, and an argument struct, that contains arguments needed to create the proximal operator. It returns the proximal operators for f and g, for the specified problem. This function, along with `admm`, is the work-horse of the ADMM library, setting up instances of problems to solve and providing proximal operators for them for all solvers and functions in the library. It consists of a massive switch statement that checks for what problem it should return a proximal operator for, then defines the prox-op functions within itself, to act as a global instance from which the proximal operators access data when called by `admm`'s updates. This function contains comments for each problem that describe it and how to solve it with ADMM efficiently, also explaining how to arrive at the proximal operators for the problem. The reason for such a set up is to conveniently have all proximal operators defined in one location (as some repeat or are very similar), and consolidate the code. It also can be used by users that want to write their own versions of the solvers or add new proximal operators to it for new problems (the function is built to scale up). A curious user can also simply look at how proximal operators are implemented or derived, in order to help them understand how the library works, keeping all this information in once place for them.

The Solvers Subdirectory

This directory contains all of the ADMM library's solvers. These are optimized functions that solve general problems ADMM is used for. There are 11 solvers so far in the library, some of which also contain multiple versions of the same algorithm or parallel versions of it. Each of these solvers will be discussed in the next major section, explaining the problem and how to arrive at a solution for it with ADMM. All testing and validation results are in this following section as well.

Generally, a solver accepts necessary inputs for its problem and an options struct that customizes both its execution and that of the calls to ADMM that the solver will perform. These solvers are designed to be easy to use for a user. Each one has a special operation that will automatically call `setuppaths` to setup paths for the ADMM library, if necessary. Assuming the names of the libraries folders are unchanged, the file name for `setuppaths` has not been changed, and the global variable `setup` created by `setuppaths` has not been altered, the paths will be silently set up for the user.

The structure of these solvers is generally as follows:

1. Setup paths if necessary.
2. Check if no arguments have been provided. It will then create a demo test of itself using the solver's associated testing function in the testers folder.
3. Check the user input for error.
4. Set up the parameters for `getproxops` from user input, and create the arguments struct to pass to `getproxops`
5. Call `getproxops` to instantiate the solver's proximal operators.
6. Call `admm` on the proximal operators and report the results struct from its execution to the user.

These functions are very short thanks to the work-horse efforts of `admm` and `getproxops`, along with the efforts of `errorcheck`. They demonstrate how quickly a user can write their own solver for an ADMM problem using the library.

The Testers Subdirectory

Complementing the each solver in the solvers library is a tester function for the solver, used to both test it and validate the results. They generate random problems of a certain size, which are designed to be solvable by the associated solver. They then run the solver on this problem, and validate the results against a "true" (or original) solution. These solvers generally accept a seed value for RNG (for repeatability and variety of tests), dimensions of the problem (rows and columns), an error tolerance that specifies how far the solver's solution can deviate from the original solution, a binary `quiet` variable that informs the tester whether or not to output and plot results of the test, and an options struct that customizes the solver's execution. The type of validation test for each solver will be discussed in the next major section, showing the testing and validation results as well.

The structure of these solvers is generally as follows:

1. Setup paths if necessary.
2. Check if no arguments have been provided. It will then create a completely random demo run of itself using default values.
3. Check user input for any error.
4. Set up a random problem to solve for the solver, generally with a precomputed solution.
5. Run the solver.
6. Validate the results of the solver against the expected results.
7. If the error from the true solution is too high (relative to the specified error tolerance), report the test as a failure, otherwise a success.
8. Output or plot any results.

The testers folder also contains an additional file `solvertester.m`. This is a function `solvertester` that will perform batches of tests over increasing scales for a solver. The purpose in doing this is to thoroughly test and validate the functionality of the solvers using the tester functions over a wide spread of random data. The data scales itself over a parameter i, which denotes the size of the data in the tests. At each scale, a certain number of trials are performed using the testers, recording the average run time, any failures, and etc. This batch tester accepts as input a string containing the name of the solver to test, a minimum scale value i_{min}, a maximum scale value i_{max}, the number of trials to perform at each scale, a binary value which denotes whether or not to show plots of the results, and an options struct customizing the execution of the solver. This options struct can also contain fields that choose the seed for the RNG for the batch tester, the error tolerance to use in the trials, the type of test to perform (e.g., scale over a square matrix, a fat matrix, a skinny matrix, etc.), a custom function handle `scaler` that accepts a scale i and outputs the problem sizes for that scale. The batch tester aggregates the results of all trials over all scales, and returns a results

struct that contains a matrix of all trials (columns) and scales (rows) of tests performed, whether they were successful or not, etc. It also reports whether there were any failures. Ideally, this batch tester is used over reasonable scales and should report that all trials successfully converged with the correct tolerance. This concludes the testing and validation for a solver.

The Examples Subdirectory

This subdirectory contain assorted examples of certain functionality of `admm`, and certain solvers. The purpose of this is to give examples of more concrete or complicated problems for solvers to solve, and to show to a user how to use the solvers and `admm`. For our immediate purposes, these serve as testing and validation of certain functionalities of `admm` and provide more testing and validation examples for certain solvers, to show that they are working as desired. The results of these functions will be shown in later sections, so we do not cover each file in detail here.

Solving, Testing and Validation of Solvers

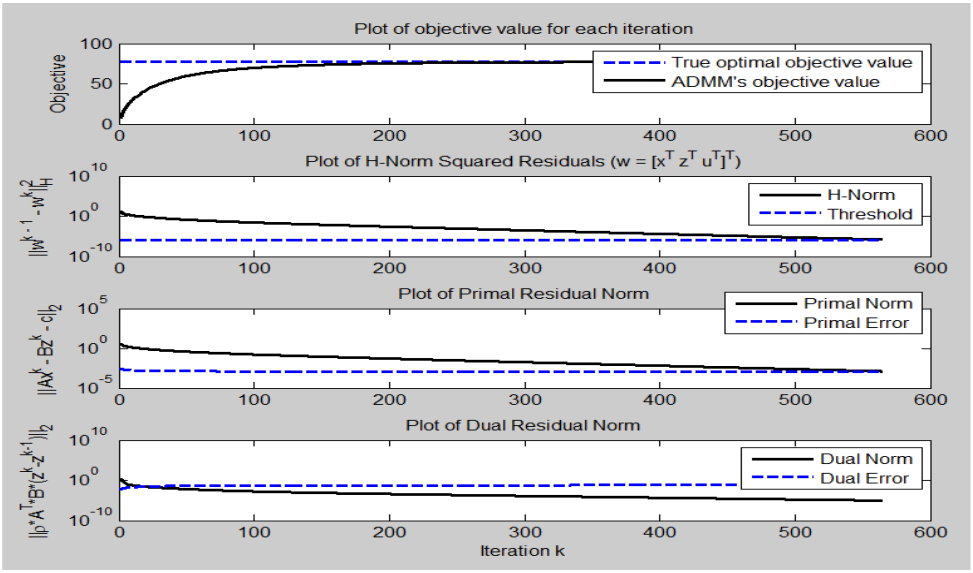
Here we discuss some problems ADMM can solve, how to solve them efficiently, and describe the implementations of solvers. We also present the basic results of our testers and batch-testing on these solvers; i.e., our testing and validation for them.

The Model Problem

Recall that the Model problem (31) is what we use to validate and test function `admm`, our primary work-horse for solving problems in ADMM. We already described how to solve it efficiently, and what the implementation would look like. We also described what it's tester function would do (generate normally distributed, random matrices P and Q and vectors r and s) to set up an example model problem, and what the true solution would be with (32). Here we present some testing and validation results on this problem to further demonstrate the correctness of `admm`.

A sample execution:

We show a sample execution of our Model problem solver in Figure 2, in order to describe the output and set the tone for what is returned by testers for other solvers.




```
>> model
No arguments detected; running a demo test of the model problem for random data of size m = n = 2^7:

ADMM EXECUTION AND TEST RESULTS ---
True optimal objective value: 77.4398
ADMM's optimal objective value for (x, z): 77.3957
Relative error in ADMM's objective: 2.0208e-06
Residual ||x_original - x_admm||: 0.00099767
Number of iteration steps performed: 565
Runtime of ADMM call: 0.2994 seconds.
Overall runtime of solver: 0.30146 seconds.
Test successful!
   ADMM's objective and x values match the true solution to specified relative error tolerance 0.001
```

Figure 2: The plot and output returned by running `modeltest.m`, the Model problem's tester.

The plots shown show the objective value converging to the true optimal objective value, and both the H-norms and primal/dual norms reach their stopping thresholds as well. This indicates that ADMM itself is functioning correctly. The text output shows a low relative error between the true objective value and ADMM's objective value, and a low value for the residual between the true minimizing x and the one ADMM computed. This validates that ADMM's results are correct in the context of the Model problem.

Relaxation revisited:

Recall the technique of relaxation (30). We can test that it works correctly here by running ADMM repeatedly on different values of the relaxation parameter α. The quadratic form of the model problem is one that would benefit from over-relaxation, so we expect to see that $\alpha > 1$ will improve the convergence. Figure 3 shows the changes in convergence of H-norm squared values with α and similar changes in the primal and dual residuals. These were obtained by running the examples folder script `relaxationexample.m`. We see that as the relaxation parameter increases, so does the convergence speed of the Model solver. The relaxation functionality seems to be working, and over-relaxation does benefit the convergence, as predicted, validating our results for relaxation.

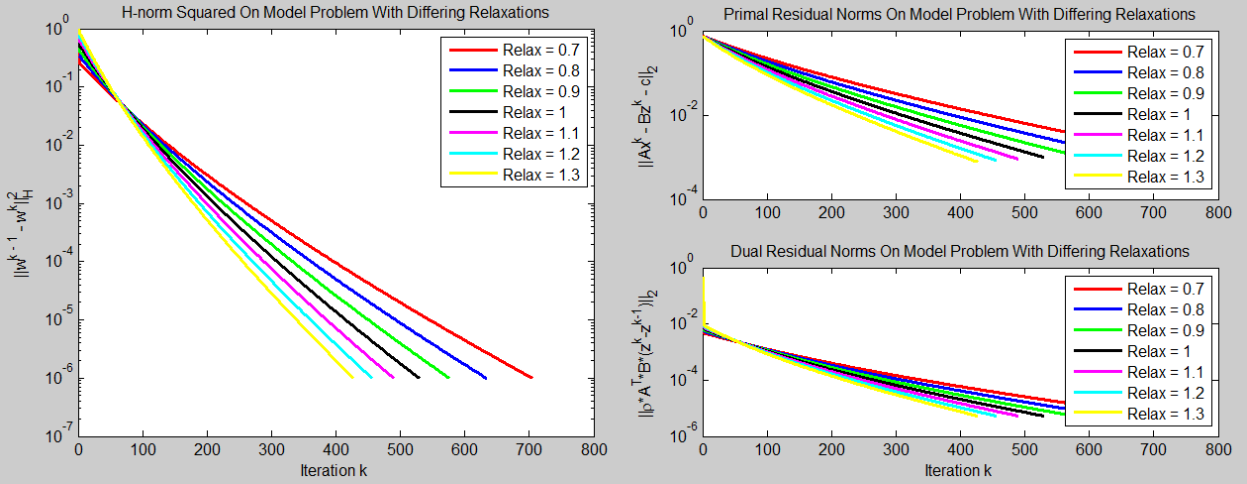


Figure 3: Differences in convergence of (left) H-norms squared values and (right) primal and dual residual norms in the Model problem, over varying relaxation values.

FADMM and AADMM Comparison:

We can additionally test and validate our FADMM and AADMM implementations on the Model problem. In general, FADMM is much faster than AADMM, as restarts in AADMM can cause a slow-down in convergence. However, Figure 4 shows an interesting scenario where this doesn't hold true. Here, normal ADMM doesn't converge sufficiently (1000 iterations is the cutoff), while Fast takes about 400 iterations and Accelerated beats all with 200 iterations. We note that up to around 150 iterations, AADMM perfectly mirrors FADMM (no restarts). Then, it suddenly converges sharply to a solution. We attribute this to the restart that happens at the same time (restarts are recorded whenever they occur by `admm`). By the quadratic nature of the Model problem, AADMM hits a good spot to begin making quadratic predictions anew and outperforms FADMM.

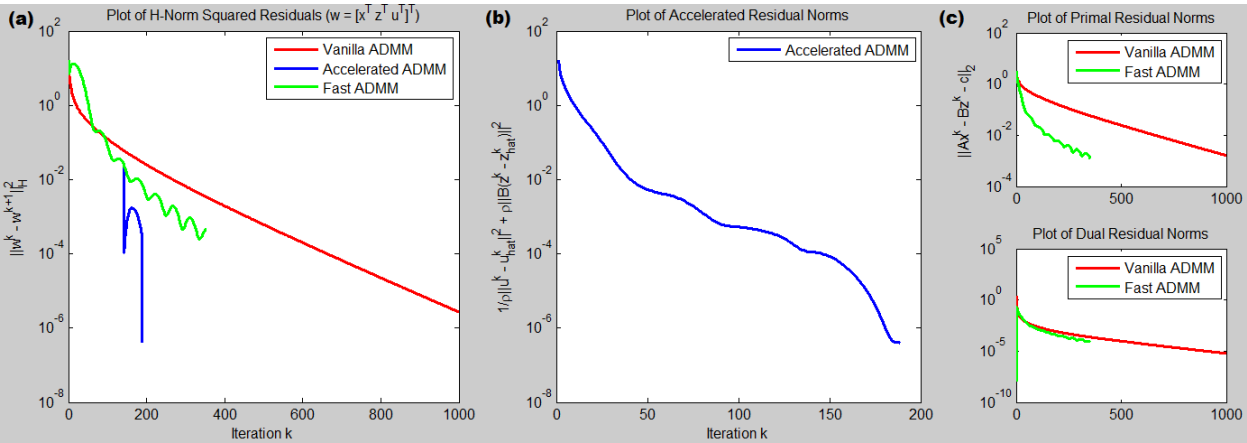


Figure 4: **(a)** H-norm squared values between the 3 ADMM variants **(b)** Accelerated ADMM's special residual norms **(c)** Primal and Dual residual norms for normal and Accelerated ADMM.

Additionally, note that H-norm squared values are no longer monotonically decreasing for AADMM and FADMM; this falls in line with the nature of these algorithms. We can clearly see the quadratic predictions occurring via the wobbling of the residuals - even the primal and dual in Figure 4.(c). These results are obtained by the examples subdirectory function `fasteradmmcomparison`.

The Basis Pursuit Problem

Basis Pursuit is the problem of minimizing for x the objective function:

$$obj(x) = ||x||_1, \text{ subject to } Dx = s \tag{39}$$

where D is a matrix and s is a column vector of appropriate length. This problem seeks to denoise a noisy signal x, for measurements s, using measurement data encoded into the rows of D. The x_{opt} signal that is returned is the denoised signal. This is usually used in an underdetermined system where we want the sparsest solution in the ℓ_1 sense. To write this problem into ADMM form, we note that the constraint $Dx = s$ implies we require an $x \in \{v : Dv = s\}$. This is an indicator problem, where we can let $f(x)$ be the indicator that x is not in this set. We let $g(z) = ||z||_1$, and then require $x - z = 0$, to tie x and z together - this is our ADMM constraint.

The proximal operator for g is easy; this is just soft-thresholding, from (10), with $t = 1/\rho$ and $v = x + u$. The proximal operator for f is a bit more involved. We note that D is expected to be underdetermined, and so we can't directly solve $Dx = s$. Instead, we use a few tricks. First we consider the matrix $P = I - D^T(DD^T)^{-1}D$. Note that this matrix is algebraically equal to 0. Also, we consider the vector $q = D^T(DD^T)^{-1}s$, which is algebraically equal to $D^{-1}s$. Then, a projection of a vector v onto the set $\{v : Dv = s\}$ is equivalent to $x = Pv + q$. For underdetermined D, DD^T is square and invertible. Thus, we cache DD^T and use system solves to avoid computing its inverse directly. This allows us to compute and cache P and q, which are used to compute the minimizing $x = Pv + q$. Recall that since we chose f to be the indicator function, it will return 0 (the minimum of f) for such an x. The vector v here is the parameter to the proximal operator. In general, for the proximal operator for f, in ADMM, $v = -(Bz - c + u)$. As $B = -I$ and $c = 0$ for Basis Pursuit in ADMM form, $v = z - u$.

By caching matrix P and vector q, we can quickly and efficiently compute $x = Pv + q$. Soft thresholding is very efficient as well, for g's proximal operator. The dominating overhead here will be the one-time system solves we perform to compute P and q. Our solver `basispursuit.m`, accepts the parameters for Basis Pursuit, D and s, and an options struct to customize the call to `admm`. To test and validate `basispursuit`, our tester function creates a normally distributed, random D. It then randomly chooses a sparse, normally distributed x. We then simply compute $s = Dx$. The resulting D and s are then passed to `basispursuit`. The tester expects the solver to minimize in the ℓ_1 sense, thus it expects the objective value to be much lower for x_{opt} the solver discovers. The testers checks the condition that the value of (39)'s *obj* function has decreased. It then checks that $||Dx_{opt} - s||$ is very small. This validates the results of the solver.

Figure 5 shows the results of a sample execution of `basispursuittest`, the tester for solver `basispursuit`. We see that denoised signal is indeed sparser in the ℓ_1 sense (less jumps than the original from the baseline). The objective values reported by ADMM decrease below the original signal immediately on the first step. Although not shown, the norm $||Dx_{opt} - s||$ here is 6.3601e-14, nearing machine precision. These results indicate that Basis Pursuit is working as intended.

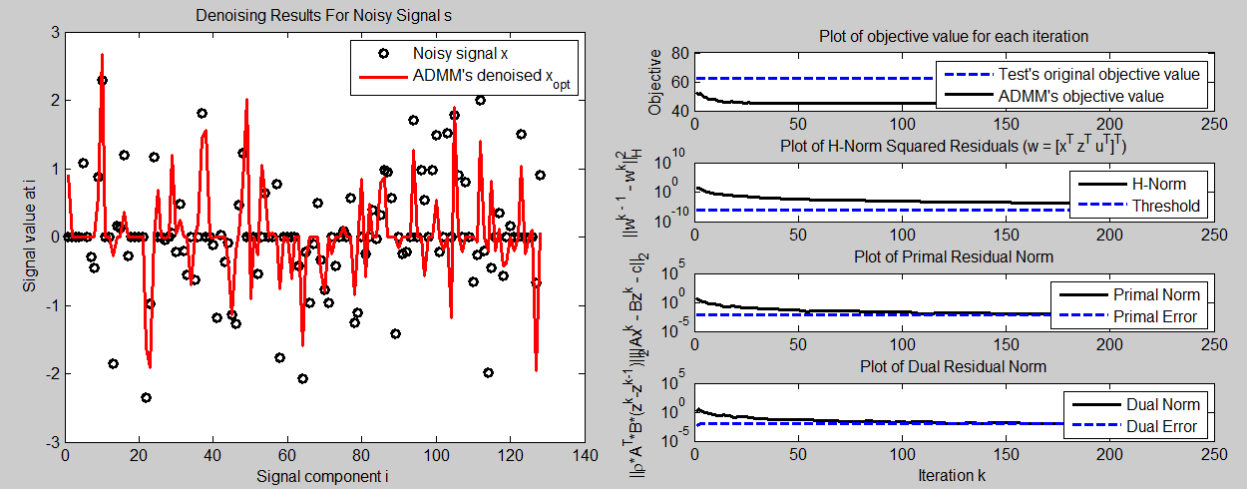


Figure 5: Results for a random test of our Basis Pursuit solver. We show the typical convergence information (right) and the original signal vs. the denoised one returned by the solver (left).

The Least Absolute Deviations Problem

We now look at similar problem of minimizing in the ℓ_1 sense. Least Absolute Deviations (LAD) is defined by the problem of minimizing the objective:

$$obj(x) = ||Dx - s||_1 \tag{40}$$

This problem differs slightly from Basis Pursuit in that we do not have the constraint $Dx = s$ anymore. We are given observation in the rows of D and corresponding noisy measurements s. Our goal here is to actually recover the original, minimal signal x, not denoise it, despite the noise

in s. We convert (40) trivially into ADMM form by letting $f(x) = 0$ and $g(z) = ||z||_1$. By the substitution $z = Dx - s$, we have $Dx - z = s$, which is a constraint in ADMM form.

The proximal operators for LAD are very easy with this formulation. The one for g is simply the soft-thresholding operator (10), once more, with $t = 1/\rho$ and $v = Dx - s + u$. Since $f = 0$, the gradient of the Augmented Lagrangian (7) collapses into:

$$\nabla L_\rho(x, z, u) = D^T(Dx - z - s + u)$$

Setting this equal to 0 and solving it for x gives our minimizing x to be the solution to:

$$(D^T D)x = D^T(z + s - u) \tag{41}$$

Instead of naively solving this system, we use a more efficient approach by finding the Cholesky decomposition $RR^T = D^T D$, where R and R^T are lower and upper triangular, respectively. We perform two easy system solves (due to R's triangularity) by first finding a y such that $Ry = D^T(z + s - u)$ and then finding our minimizing x such that $R^T x = y$.

To test and validate our solver, we want to set up a problem where we already know the true minimizing signal x, and design D and s around that. We know our solver works if it recovers the true signal x to some tolerance. Once more, our tester does this by creating a random but normally distributed matrix D, and a large scalar multiple of a normally distributed, random vector x. This is our true solution. We compute s by first letting $s = Dx$, but then introduce largely deviating, random noise into a random subset of indices of s, to create a noisy s. LAD is good at tolerating highly deviating outliers, thus we expect that it will still recover the true x to high precision.

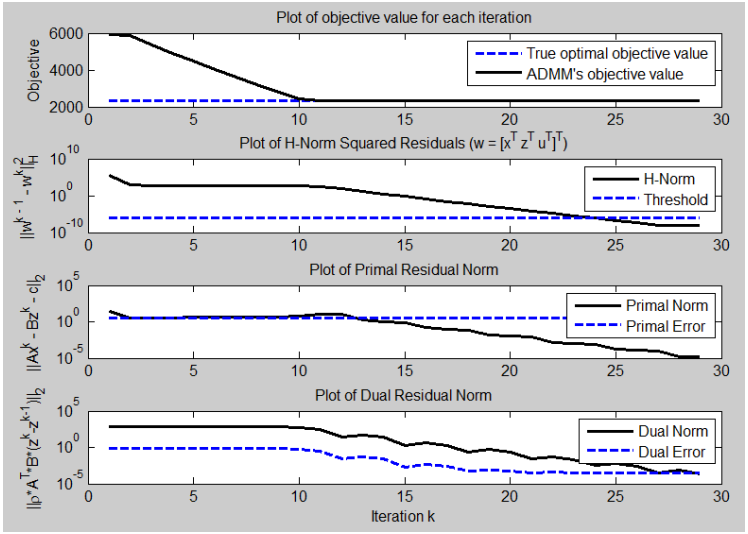


Figure 6: Convergence results for a random test of our LAD solver.

Figure 6 shows the convergence information for a random test of `lad.m` via its tester `ladtest.m`. We see that it quickly converges towards the optimal objective value of the true solution. ADMM's solution is within 1e-5 of the true signal x's objective value. The norm of the difference between

the true and recovered x is even lower, at 3.8234e-7 with an average error in a component even smaller, at 2.6604e-8. These results are as expected from LAD; it solved the problem accurately despite highly deviating noise in s.

The Huber Fitting Problem

The Huber function is defined as:

$$huber(a) = \begin{cases} a^2/2 & \text{if } |a| \leq 1 \\ |a| - 1/2 & \text{if } |a| > 1 \end{cases} \tag{42}$$

For vector arguments, this is applied over the components of the vector. Suppose that we wanted to fit (42) to row data matrix D and measurement vector s. This problem can be defined as minimizing for x the objective function:

$$obj(x) = 1/2 \sum(huber(Dx - s)) \tag{43}$$

We can write (43) in ADMM form with the same trick as for LAD; set $f(x) = 0$ and $g(z) = obj(z)$, such that $Dx - z = s$. The proximal operator for f is thus the same as LAD's (41) and we can use the same strategy to compute it efficiently. The proximal operator for g, however, is much more involved.

We see that (42) is smooth and its gradient can be computed piecewise as well. In the first case, the gradient of $a^2/2$ is simply a. Thus, in this case the gradient of the Augmented Lagrangian for component z_i is:

$$z_i - \rho([Dx]_i - z_i - s_i + u_i)$$

Setting this equal to zero and solving it for z_i gives:

$$z_i = \rho/(1+\rho)([Dx]_i - s_i + u_i) \tag{44}$$

For the other case, the gradient of $|z_i| - 1/2$ will be $z_i/|z_i| = \pm 1$, depending on the sign of z_i. Solving for z_i in this situation would give:

$$z_i = ([Dx]_i - s_i + u_i) + (\pm 1)/\rho$$

We can split these terms up into:

$$z_i = \rho/(1+\rho)([Dx]_i - s_i + u_i) + 1/(1+\rho)([Dx]_i - s_i + u_i + (\pm 1)(1+\rho)/\rho) \tag{45}$$

Comparing (44) and (45), we see the only difference is the second term in (45). If $|z_i| \leq 1$, then the second term is 0, otherwise it is there. If we let $v_i = [Dx]_i - s_i + u_i$ and $t = (1+\rho)/\rho$, then such behavior on the second term's $v_i + (\pm 1)(1+\rho)/\rho$ portion matches soft thresholding (10) for v and t. Thus, the z update can be written as:

$$z = \rho/(1+\rho)v + 1/(1+\rho)\mathbb{S}(v,t) \tag{46}$$

where $v = Dx - s + u$, $t = (1+\rho)/\rho$ and $\mathbb{S}$ is the soft-thresholding operation.

Huber fitting essentially replaces sharp, absolute-value like points with a smooth quadratic tip. Thus, this sort of fitting is resistant to spikes in data, much like with LAD, although without the need for sparsity.

The tester for `huberfit` sets up a normally distributed, random signal x. It then creates a normally distributed, random matrix D and normalizes the columns of it. Finally, a noisy vector s is created by computing Dx. We add sparse, but large noise to s. We expect that `huberfit` will not be affected by these large spikes. We omit a demonstration of Huber Fitting as it is somewhat similar to LAD.

The LASSO Problem

LASSO strives to minimize for x the objective:

$$obj(x) = 1/2||Dx - s||_2^2 + \lambda||x||_1, \tag{47}$$

LASSO fits a signal x using the ℓ_1 regularizer, but also introduces a smoother, quadratic term to the fit. We implement a serial and a parallel version of LASSO in the solver, to test some functionality of parallel ADMM as well. The LASSO problem is easily converted to ADMM form by making $f(x)$ the first term of (47), and $g(z)$ the second, with the ADMM constraint $x - z = 0$.

Serial LASSO:

Once more, we notice the proximal operator for g can be computed by soft-thresholding (10). The prox-op for f is quadratic and we can simply take the gradient of the Augmented Lagrangian, set it equal to zero and solve for x. This gives us the exact same proximal operator as (32) for the Model problem's f term. The tester for `lasso` works much the same as the `huberfit` one. We look for a reduction in the objective value from the original signal to the one LASSO provides.

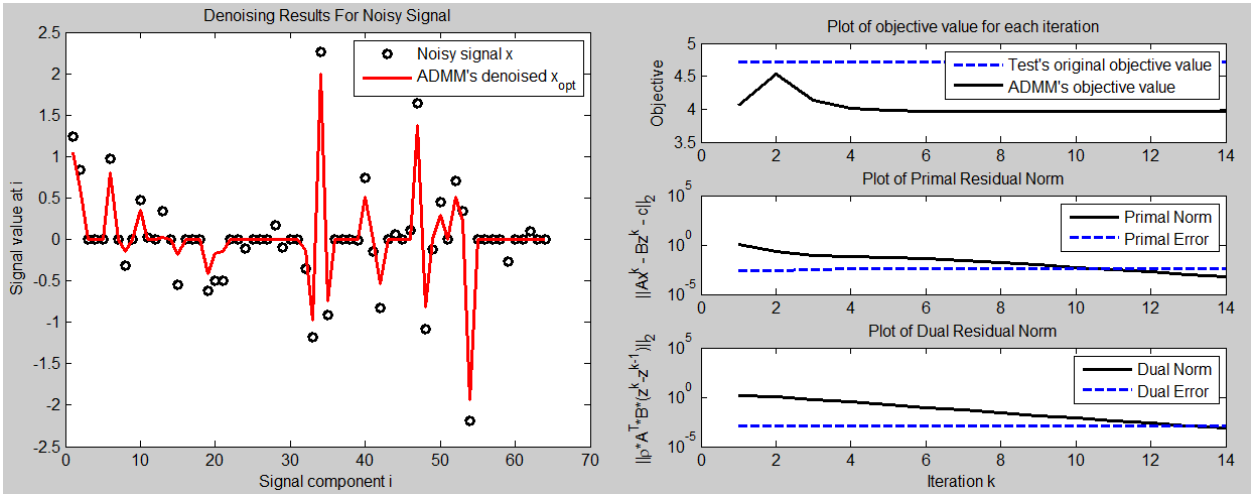


Figure 7: Results from a random trial of `lasso`

Figure 7 shows the results of LASSO denoising a signal. We first notice that the LASSO solver converges very fast; in just 14 iterations. The denoised signal in red hugs the center trend (normal distribution), as expected and spikes follow the sparse deviations in the original signal, though they are lower in amplitude (due to the denoising). It seems the LASSO solver is working as intended.

Parallel (Consensus) LASSO

We can parallelize LASSO by slicing up the data matrix D and signal measurement vector s into groups of rows. Each slice then becomes its own LASSO problem to solve, which is handled by a parallel process. Splitting up the data this way allows the x-update to be explicitly parallelized. Thus, each worker i in a parallel pool is given a slice D_i and s_i, and they perform the x update for serial LASSO by pretending their slice is a new LASSO problem for a variable x_i such that $x_i - z = 0$. When they perform their x_i updates, they report to a central server that then computes the average $\bar{x} = 1/n \sum_{i=1}^{n} x_i$. The central server itself performs the z update, using the newly computed $\bar{x}$ and the last $\bar{u}$, via soft-thresholding, as in serial LASSO, but with $v = z - \bar{u}$ and $t = \lambda/(\rho n)$ to reflect the division of labor across n workers. The workers, now knowing $\bar{z}$, can compute their own updates for u_i. Once again, the central server aggregates these u_i and computes $\bar{u}$. Figure 8 shows this workflow for a single iteration of ADMM.

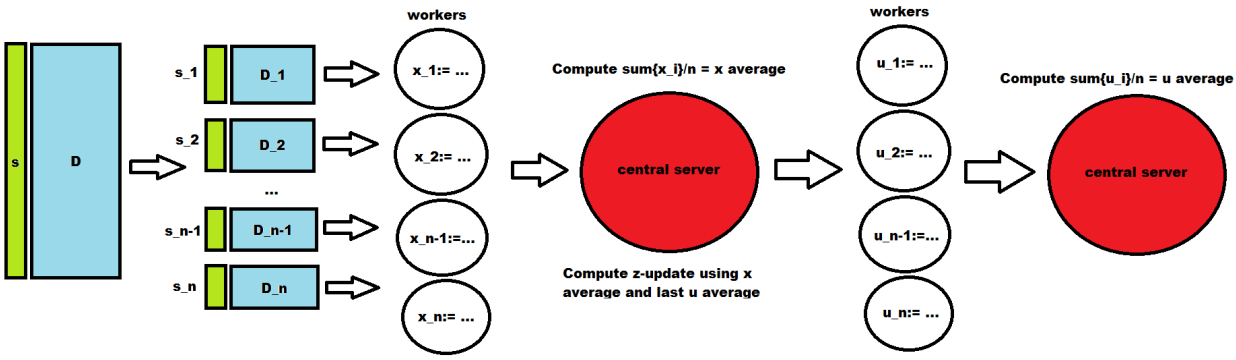


Figure 8: Work flow in parallel LASSO for a single iteration of the ADMM update loop.

We run the same trial as we did on serial LASSO and show the results in Figure 9. Interestingly, the results are quite similar; even convergence occurs at the same number of iterations. From empirical trials, it seems these two strategies of performing LASSO are largely equivalent. However, using `parfor` has consequences. Initializing a parallel pool takes a good number of seconds. There is also significant overhead to letting MATLAB control the worker's tasks. Figure 10 shows the results of running the `solvertester` function (a batch tester) on both serial and parallel LASSO, with the same seed and 10 trials per scale. The serial LASSO was run on a two core machine, and the parallel LASSO was allotted two cores for two workers. We see that there is a starting factor of nearly 100 times more overhead with `parpool`. This factor decreases as the problem size and run-times increase. It turns out that MATLAB's overhead in `parfor` does not scale with the size, but instead decreases - this is due to the fact that larger operations (on larger sized problems) begin to overtake the overhead of communication with workers in parallel LASSO. The last data point in the plots is likely an outlier, and with large enough problem sizes, even two-cores suffice for LASSO to be better than serial LASSO on a two-core machine in `Matlab`.

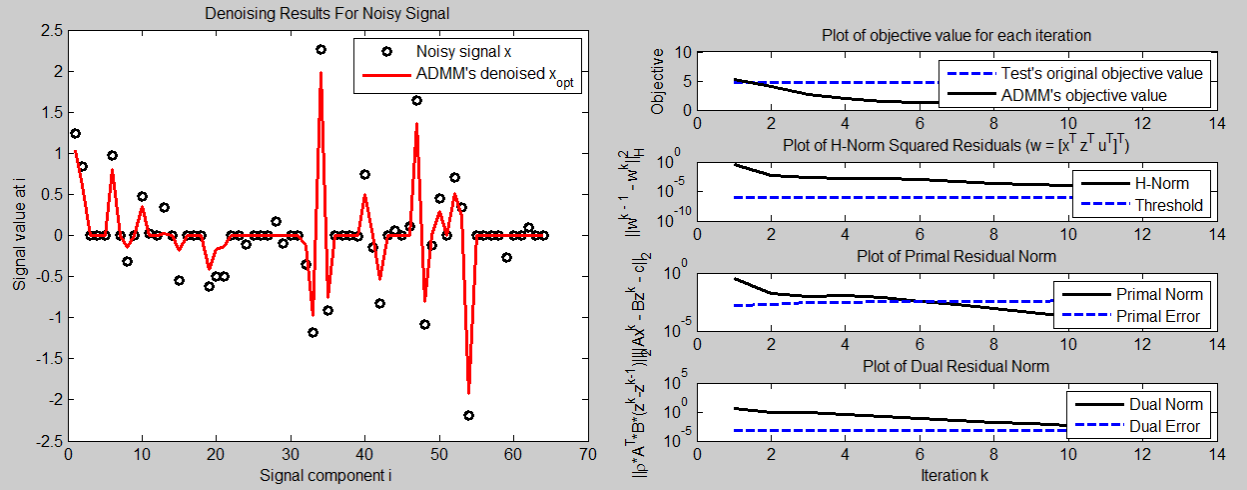


Figure 9: The same test on parallel LASSO yields similar results

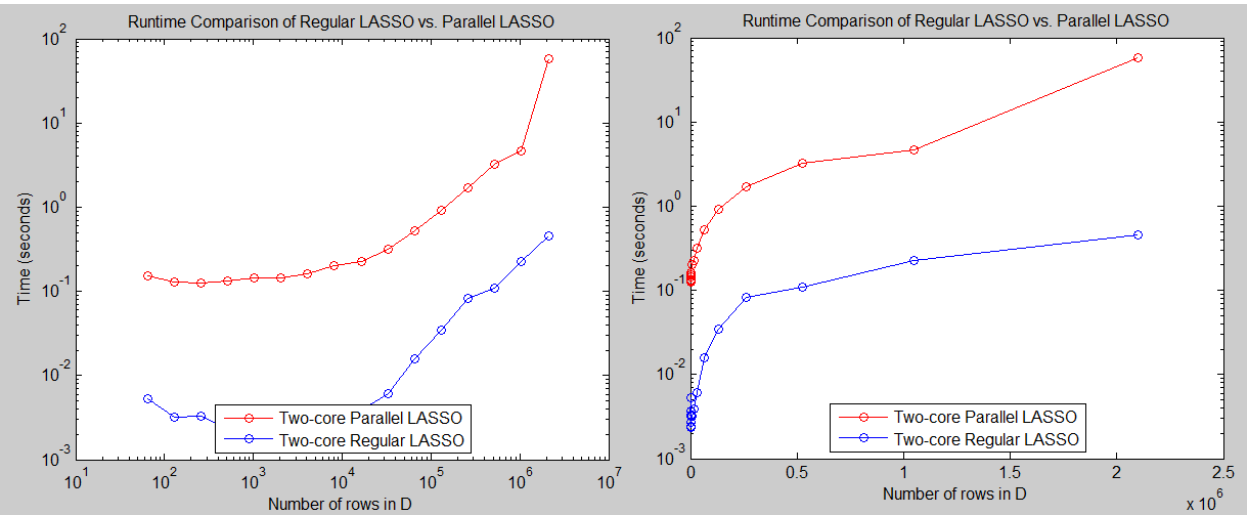


Figure 10: A comparison of average runtimes of serial vs parallel LASSO, over increasing problem sizes. On the left is a log-log scale of the results and on the right is a log-y axis scale of the results.

It's difficult to measure the overhead of `parfor` that we see in Figure 10, due to the design of anonymous functions and the way proximal operators are implemented; there is no way for a proximal operator to reliably convey timing on `parfor`'s execution. However, we can measure how long the proximal operators take to set up (slicing of data into cells), and it is confirmed that the contribution of that, performing Cholesky decompositions, matrix multiplication, and etc., are dwarfed by the overhead of `parfor`, even at large problem sizes. A casual survey of other `Matlab` users of `parfor` loops show that the overhead experienced in parallel LASSO is normal.

Communication between workers (cores) is what slows down `parfor`. It is really only meant to be used when someone has at least 4 cores and is working with large datasets.

Unwrapped ADMM And The Linear SVM Problem

To test our general Unwrapped ADMM solver, and by proxy, our Linear SVM solver, we solve the Linear SVM problem with Unwrapped ADMM. The Linear Support Vector Machine (SVM) problem strives to fit a loss function to a set of data D, such that it will generate a dividing hyperplane between classes, and thus create a general classifying hyperplane. In Unwrapped ADMM form, this problem can be formulated as:

$$obj(x) = 1/2||x||_2^2 + C \text{ loss}(Dx) \tag{48}$$

where loss is a loss function and C is a constant, regularization parameter (enforcing strictness of classification). The smaller C is, the less of a hard margin exists on the classifying hyperplane.

We've already discussed how Unwrapped ADMM works, and a serial version is implemented exactly as described. The only thing left to discuss for the serial case is the proximal operator for $g(z)$, which in this context is C loss(Dz). There are two types of common loss functions considered for Linear SVMs; the first is the 0-1 loss function, which is an indicator function of a component of z being incorrectly classified. The second is the Hinge loss function, $h(v) = \sum_i(\max(1 - \ell_i v_i, 0))$, where ℓ contains the training labels for classification. The Hinge loss function is a linear abstraction of the 0-1 (which is discontinuous), that is continuous and piece-wise differentiable. Thus, the Hinge loss is more used in practice. Its proximal operator in Unwrapped ADMM is:

$$z := (Dx + u) + \ell^T \max(\min(1 - \ell^T(Dx + u), C/\rho), 0) \tag{49}$$

In general, we cannot find a proximal operator for the 0-1 loss function. However, by transpose reduction (25) in Unwrapped ADMM (23), we find that it is possible, as this technique makes rows independent of each other. We test our Linear SVM solver by having it classify points above a line $x_1 = x_2$ as $+1$ (correctly labeled) and below it as -1 (incorrectly labeled). The points are scattered around this dividing line randomly. A point that is supposed to be positive labelled will be placed a random distance away from $x_1 = x_2$, but above it, and vice versa for negative labelled points. These points are then randomly moved back towards $x_1 = x_2$ based on a separability parameter, to create some overlap of points that is nontrivial to separate linearly. Figure 11 shows an example of this test with separability parameter $s = 0.2$ (points get moved back randomly with coefficient s on the random value between 0 and 1). We can visually see that the lines returned by both loss functions are close to the theoretically correct $x_1 = x_2$ line and each other.

We also have a harder testing and validation method for the Linear SVM solver; classification of the MNIST hand-drawn digits. These are not entirely linearly separable, but should be close enough such that a linear SVM will guess the correct digit with acceptable accuracy. A multi-class (more than 2) classifier with Linear SVMs can be trivially implemented by training the SVM on each digit class, to distinguish between that digit and all others. Thus, we simply run `linearsvm` 10 times; once for each digit. We encode the handdrawn digits into row vectors by stacking the pixel rows of the image horizontally. They are 20 by 20 pixel images, thus we get vectors of length 400. We have 60 thousand images at our disposal to choose training and testing subjects from, of what size we please. Generally, it's a good idea to have at least 5 times more training images than testing images.

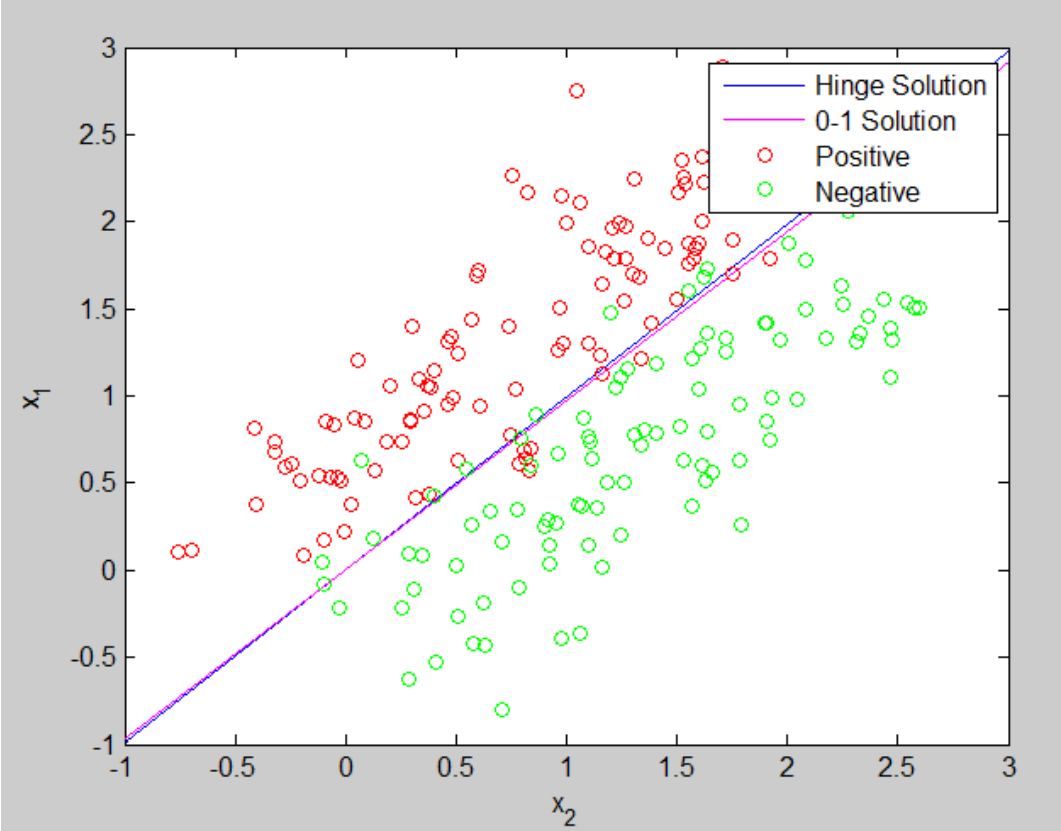


Figure 11: An example of a test of `linearsvm`. The dividing hyperplanes returned by the Hinge and 0-1 loss are both shown.

In Figure 12, we show the percentages of incorrectly guessed digits of each type over several varying classification runs with the Linear SVM solver on subsets of the MNIST data. We show the results for both types of loss functions. In general, it seems that the linear classifier does a good job falling under 10% for almost all digits in the training stage, on both types of loss functions. We see in the top-left corner, that having a small sample under-represents the data and gives bad testing results, despite the training results being seemingly good. The non-linear classification of these hand written digits is evident when even going out to 2000 iterations and thousand of training and testing samples doesn't improve the results. We also observe that the 0-1 loss and Hinge loss have different classification behaviors; sometimes the 0-1 loss does a much better job and sometimes it does not. All across the board, the digit 8 was the hardest to classify correctly in both the testing and training stages. This perhaps because of how similar hand-drawn 8's can look to other digits, such as to 3's, 0's, 6's, 9's and 5's. The Linear SVM clearly has trouble classifying 8's correctly during the training stage and suffers for it in the testing stage.

Additionally, we also implemented a parallel version of Unwrapped ADMM. This implements the strategy (23) for updating x and the methods described there. Note that for Unwrapped ADMM, the size of slices needs to be consistent across all sizes in order to build the matrix W. We tested how well parallel ADMM compares to its serial version in terms of performance using `parfor`.

250 Iterations

Error Percentages: 600 training, 100 test samples.

Digit	Hinge (Train)	0-1 (Train)	Hinge (Test)	0-1 (Test)
0	2.0000	2.0000	13.0000	13.0000
1	0.8333	0.8333	9.0000	9.0000
2	3.1667	3.3333	22.0000	20.0000
3	2.5000	2.1667	21.0000	22.0000
4	3.3333	3.1667	12.0000	12.0000
5	4.6667	4.6667	18.0000	18.0000
6	1.8333	1.8333	12.0000	12.0000
7	2.5000	2.5000	23.0000	23.0000
8	4.8333	4.6667	18.0000	20.0000
9	3.5000	3.6667	30.0000	28.0000

Elapsed time is 10.735045 seconds.

250 Iterations

Error Percentages: 6000 training, 1000 test samples.

Digit	Hinge (Train)	0-1 (Train)	Hinge (Test)	0-1 (Test)
0	2.4667	2.8167	4.3000	4.7000
1	2.0833	2.6000	4.3000	4.7000
2	3.9333	4.0333	8.3000	7.7000
3	5.1833	4.5500	9.0000	8.0000
4	3.6333	4.2667	7.8000	8.8000
5	5.5833	4.5833	9.9000	8.8000
6	2.4833	3.2833	5.2000	6.1000
7	3.0500	3.7000	5.6000	6.4000
8	13.2500	8.5833	16.5000	13.0000
9	8.2667	7.9333	12.7000	12.5000

Elapsed time is 102.358839 seconds.

250 Iterations

Error Percentages: 60000 training, 10000 test samples.

Digit	Hinge (Train)	0-1 (Train)	Hinge (Test)	0-1 (Test)
0	2.9850	3.1417	3.0100	3.2400
1	3.0050	3.5200	2.7400	3.2200
2	5.9383	5.1150	5.7300	5.2900
3	7.4017	6.3633	7.4500	6.6100
4	5.2450	5.3500	5.9700	6.2100
5	7.4867	6.0067	7.5000	6.0600
6	3.7483	3.8583	4.1300	4.2600
7	4.2467	4.4667	4.2800	4.6800
8	15.0200	10.5783	15.3800	11.3700
9	10.9150	10.0067	11.0600	10.1800

Elapsed time is 1016.946927 seconds.

2000 Iterations

Error Percentages: 12000 training, 2000 test samples.

Digit	Hinge (Train)	0-1 (Train)	Hinge (Test)	0-1 (Test)
0	2.1000	1.8750	3.6500	3.0000
1	1.5667	1.7583	2.8000	2.9000
2	5.1167	2.7583	5.9500	4.2000
3	7.1000	3.6833	7.1000	4.7500
4	4.1750	3.2333	6.1000	5.0500
5	5.8583	3.3583	6.9000	4.5000
6	2.4667	1.9000	4.0000	3.6000
7	3.2917	2.8833	4.2500	4.4500
8	13.5750	6.8000	16.2500	10.1000
9	9.0083	6.5750	10.1500	7.6000

Elapsed time is 1694.761875 seconds.

Figure 12: Several runs of the MNIST classifier, with varying iterations and training/testing subset sizes.

Figure 13 shows our results for a batch test testing the average run-time performance of serial vs. parallel Unwrapped ADMM, over increasingly larger problems to solve. Once again, we see the trend of the parallel version being roughly 100 times slower than the serial version for smaller problems. However, these seem to converge quicker towards each other than in the Parallel LASSO case, likely due to the Unwrapped ADMM technique being much efficient in parallel form. Before hitting problems of scale 1 million, Parallel ADMM is only 10 times slower. It's likely that for problems of sizes in the millions, it would actually be faster to use parallel Unwrapped ADMM via `parfor` loops on just two cores, than MATLAB's own multithreaded implementations. Given the tremendous overhead of `parfor`, this is actually somewhat impressive on the part of Unwrapped ADMM with Transpose Reduction.

Honorable Mentions Among Solvers

We did not describe all the solvers I implemented here, nor all the strategies, in the interest of space. I also implemented solvers for Quadratic and Linear Programming problems, which are very similar to each other in implementation, but are also not very exciting to demonstrate any results on. Furthermore, there is a solver for Sparse Inverse Covariance Selection, but I omitted discussion of that because (yet again) the results were not very interesting nor informative, but the problem, and the solution for it, were quite difficult to explain.

I also did not discuss the Total Variation Minimization (TVM) solver, which is a specific case of the generalization of the LASSO problem where $D = I$ and we apply a differencing scheme

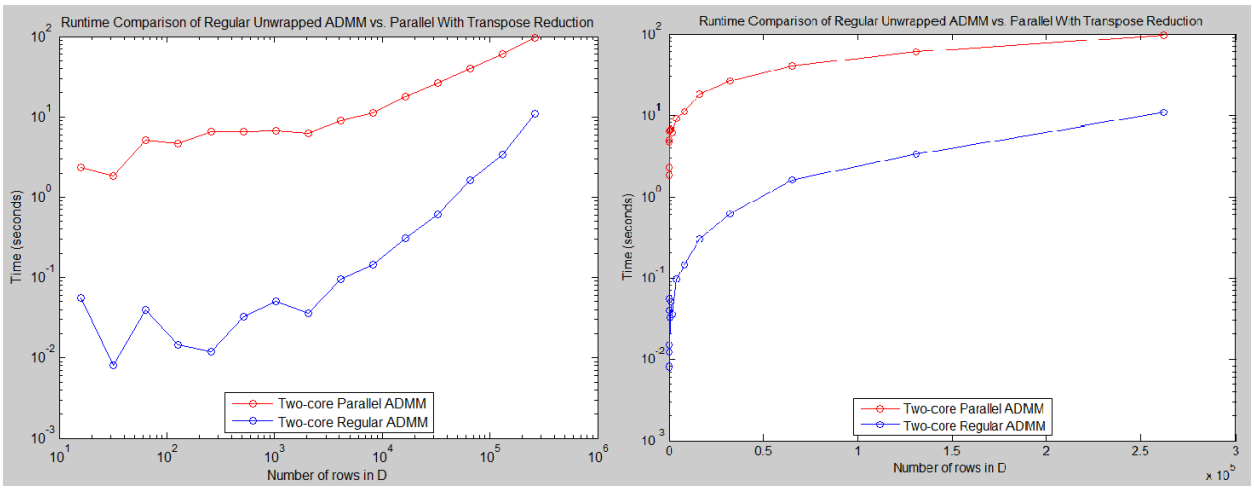


Figure 13: On the left, a log-log plot of 2-core Parallel Unwrapped ADMM's runtime vs. serial Unrapped ADMM's runtime. On the right, a semi-log graph of this, showing the trends better.

---

on the x in the ℓ_1 regularization term. Solving this problem can be done very quickly as we've already collapsed D into the identity and can use convolutions/FFTs to help compute the solution quickly, and the differencing scheme on the regularization term is sparse, and thus easier to deal with. However, I implemented a solver for Total Variation early in the first semester and discussed in presentations at least twice, as well as describing it in the first progress document last semester, so I omitted rehashing it again here.

Adaptive Stepsizes

Unfortunately, I was unable to get adaptive stepsizes to work, despite trying many things. For all the strategies I attempted to use, they all shared the same issue of exploding or disappearing step sizes. Perhaps the method worked for the first number of step sizes and seemed to speed up the convergence. However, then it would level out and stopping converging entirely due to uncontrolled step-size growth that would approach infinity. Sometimes, the opposite issue would happen, and the step sizes diminish drastically, to the point where we cease moving forward towards the solution as well. I summarize my attempts at adaptive step sizes in this section and discuss potential solutions that might still be worth pursuing.

Esser's Work On ADMM's Connections to the Douglas-Rachford Method

In this paper, Esser shows a connection between ADMM and Split-Bregman. More precisely, he demonstrates that ADMM is equivalent to the Douglas Rachford Splitting Method (DRSM), in the sense that ADMM is DRSM applied to the dual problem:

$$\max_{u\in\mathbb{R}^d}(\inf_{x,z\in\mathbb{R}^{m_1,m_2}}(L(x,z,u))) \tag{50}$$

The implication of this proof is that ADMM is equivalent to finding u such that $0 \in \psi(u)+\phi(u)$, where $\psi(u) = B\partial g^*(B^T u) - c$ and $\phi(u) = A\partial f^*(A^T u)$, where A, B, and c are the typical constraint variables in ADMM. One strategy we tried is to form the residuals equal to $\psi(u^k) + \phi(u^k)$, for each iteraton k, and interpolate with last residual $(k-1)$ over stepsize ρ_k. We would then like to minimize this for a new step-size parameter. That is, we would like to solve:

$$\frac{r-r^k}{\rho} = \frac{r^{k+1}-r^k}{\rho_k} \iff r = r^k + (r^{k+1}-r^k)\frac{\rho}{\rho_k} \iff \min_{\rho}(r^k + (r^{k+1}-r^k)\frac{\rho}{\rho_k}) \tag{51}$$

This is simply the least squares problem in one dimension, and thus can be solved explicitly. If we let $\hat{r} = r^{k-1} - r^k$, the solution is:

$$\rho_{k+1} = -\frac{\rho_k \hat{r}^T r^{k-1}}{\hat{r}^T \hat{r}} \tag{52}$$

The idea is that assuming this linear regression situation, we can find what a better ρ would have been for the last iteration - then use it in the next one. The hope would be that the change in optimal ρ would not be too different between just one iteration.

However, results show that optimal ρ values found by this method tend explode when seeded to a large ρ, or approach 0 for a small seed. Either situation produces too much round-off error. The solution also doesn't converge when the values are allowed to change so drastically. Figure 14 shows an example of these exploding values. One can see how these would pose an issue towards the convergence and numerical stability of ADMM.

If such drastic changes are not allowed, i.e., checking relative change in these values of ρ and requiring them to be within some realistic difference, setting them to the previous ρ if they are

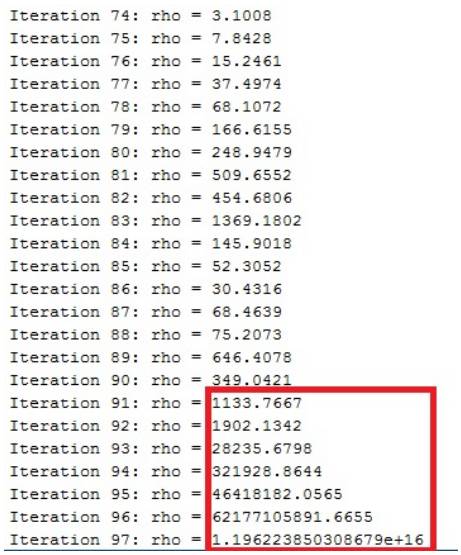


Figure 14: An example of the technique from (52) exhibiting the phenomenon of exploding step-sizes. This occurs 90 iterations into execution.

---

not, then adaptive ADMM seems to "work". However, it doesn't converge in less iterations as often as it should. Sometimes it will converge in the same number of iterations as regular ADMM. Sometimes slightly less; rarely, even significantly less. On some occasions, it actually performs a little worse (some low step-size values were chosen)! We would like to improve this performance to consistently give significant lower iteration counts than a static step-size. Most of my efforts on adaptive step-sizes involved trying to fix this issue.

Using H-Norm Squared Values To Predict Stepsizes

Recall the w encodings (11), which, with the H matrix (12), define monotonically decreasing residuals (13). These values provide some insight into the convergence of ADMM, as noted by [12], without needing to know anything more than the state of the ADMM iterates. These are a prime candidate for adaptive step sizes, since we can tell if ADMM is converging for a step-size if they still decrease. Furthermore, if a change in step size does not break convergence, odds are that the change aided convergence; especially if the differences between these values increase from it.

I spent a great deal of time just trying things out on various problems with this idea in mind. However, I kept running into the issue of inexplicably exploding or diminishing step sizes, much as in Figure 14. A future endeavor that might help with this is mapping out optimal step size paths through brute force means. A first step in this direction is the function `stepsizetesting`, that can be used to run multiple simultaneous executions of tester functions, on differing values of ρ. This helps to visualize the value of ρ actually affect convergence. For example, Figure 15 shows the effect of ρ on the convergence of the Total Variation Minimization solver. We see that decreasing ρ here will strictly increase the convergence speed, up till a certain point (step sizes must be positive). On the flip side, problems like Sparse Inverse Covariance Selection give more ambiguous results, as shown in Figure 16. The iterations do not increase with ρ; the trials converge out of order.

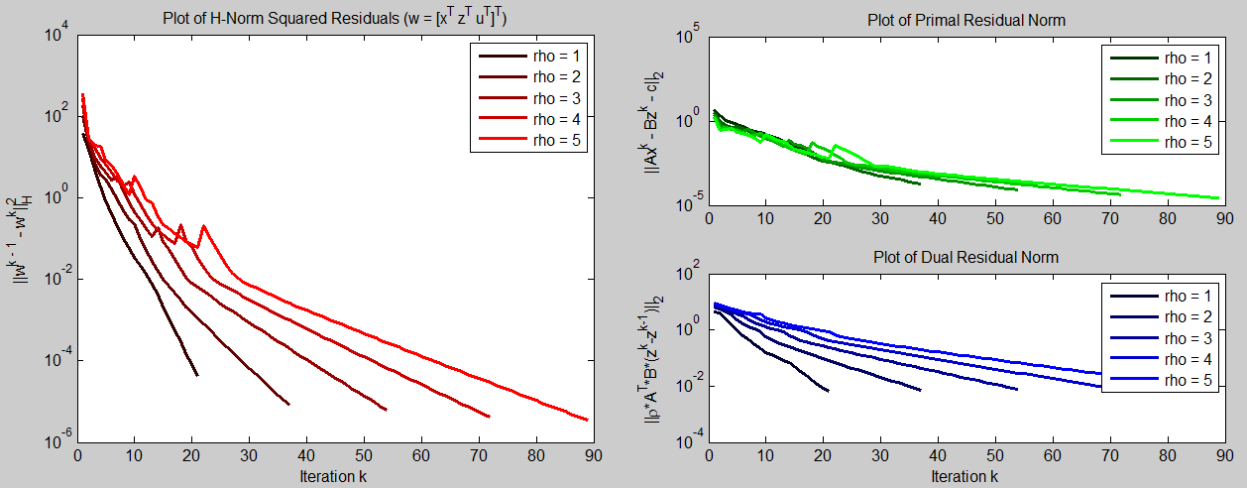


Figure 15: Output from function `stepsizetesting`, which allows you to test multiple stepsizes on a single solver and see the effect on it. This is for the TVM problem.

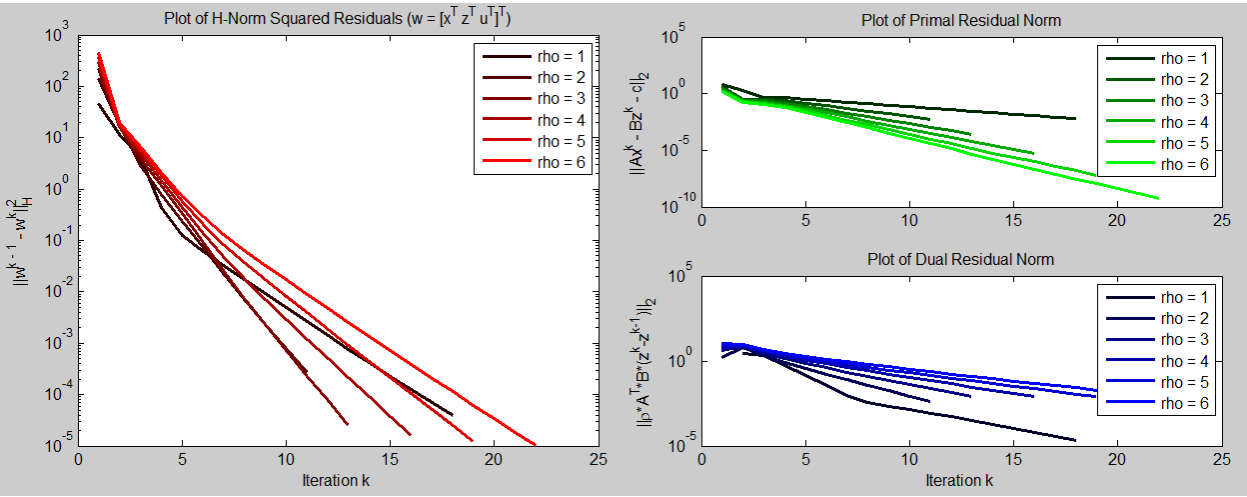


Figure 16: Output from function `stepsizetesting`, which allows you to test multiple stepsizes on a single solver and see the effect on it. This is for the Sparse Inverse Covariance Selection problem.

Future Work

There's a lot of directions one can go in to improve the ADMM library:

1. Generalize the solvers to larger problems, then create less general solvers that use these. For example, a general solver for functions that use regularization terms. This would then be used by LASSO, Basis Pursuit, etc. Or, a general consensus ADMM setup.

2. More work on adaptive step sizes. There are bits and pieces of potential ways to make adaptive step sizes that are actually somewhat useful. There's a lot of potential in the H-norm squared values, as they have properties tied directly to convergence that we could potentially abuse. There are also general schemes for choosing good values of ρ every iteration, but they rely on selection of other, sometimes multiple, parameters. Perhaps we can find a way to set these parameters (maybe even every iteration) such that the step sizes are selected appropriately.

3. Add more options to the customizability of `ADMM`. For example, warm starting, the act of using optimal ADMM solutions from a previous run of a solver to speed up convergence of a current run. This would require a way for users to save and retrieve prior optimal values to warm start whatever they would like for the next run. Also, options for disabling record-keeping of data during execution. I've noticed that batch testing really large problem sizes will quickly drain memory when you record everything that happens.

4. Generalize `admm.m` to higher dimensional input. The values of x, z and u can be matrices, for example. This is already done in the solver for Sparse Inverse Covariance Selection. However, constraint matrices A and B could be of higher dimension, such as 3-d matrices. Multiplying matrices would be an immediate issue.

5. Allow solvers to accept function handles instead of matrices, as well. Sometimes the constraints for a problem are too big to directly put into a matrix, or perhaps it is inefficient to do so.

6. Try to see if there is a better way to create locally parallelized ADMM. Eventually, it would be nice to scale up to distributed support for executing ADMM (e.g., message passing). However, MATLAB might not be the best choice of language for this, given the slowness of the `parfor loop`.

7. Implement a Python version, as was originally planned. This code should be able to reach those who haven't purchased MATLAB or obtained it by some other means, as well.

Conclusion

The work that has been accomplished so far is a great basis for a very usable piece of software. The ultimate goal is to make solving problems with ADMM very easy to do, and more attractive for researchers who may not have the programming skills or knowledge necessary to make their own, efficient implementations of proximal operators. Through my own efforts of programming these solvers, I've seen the usefulness of having a general `admm` function that can be customized to run as desired. Although the adaptive approach at step sizes has not yet produced much, it is a start to be able to at least sometimes achieve better convergence. In the future I hope to address the issue of exploding /disappearing step sizes and finally obtain a usable method for selecting good step sizes.

References

[1] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers", *Foundations and Trends in Machine Learning*, vol. 3, no.1, pp. 1-122, 2010.

[2] G. B. Dantzig, *Linear Programming and Extensions*. RAND Corporation, 1963.

[3] G. B. Dantzig and P. Wolfe, "Decomposition principle for linear programs", *Operations Research*, vol. 8, pp. 101-111, 1960.

[4] J. F. Benders, "Partitioning procedures for solving mixed-variables programming problems", *Numerische Mathematik*, vol. 4, pp. 238-252, 1962.

[5] H. Everett, "Generalized Lagrange multiplier method for solving problems of optimum allocation of resources", *Operations Research*, vol. 11, no. 3, pp. 399-417, 1963.

[6] M. R. Hestenes, "Multiplier and gradient methods", *Journal of Optimization Theory and Applications*, vol. 4, pp. 302-320, 1969.

[7] M. R. Hestenes, "Multiplier and gradient methods", in *Computing Methods in Optimization Problems*, (L. A. Zadeh, L. W. Neustadt, and A. V. Balakrishnan, eds.), Academic Press, 1969.

[8] M. J. D. Powell, "A method for nonlinear constraints in minimization problems", in *Optimization*, (R. Fletcher, ed.), Academic Press, 1969.

[9] D. Gabay and B. Mercier, "A dual algorithm for the solution of nonlinear variational problems via finite element approximations", *Computers and Mathematics with Applications*, vol. 2, pp. 17-40, 1976.

[10] R. Glowinski and A. Marrocco, "Sur l'approximation, par elements finis d'ordre un, et la resolution, par penalisation-dualit'e, d'une classe de problems de Dirichlet non lineares", *Revue Francaise d'Automatique, Informatique, et Recherche Operationelle*, vol. 9, pp. 41-76, 1975.

[11] J. Eckstein and D. P. Bertsekas, "On the Douglas-Rachford splitting method and the proximal point algorithm for maximal monotone operators", *Mathematical Programming*, vol. 55, pp. 293-318, 1992.

[12] B. He, X. Yuan, "On non-ergodic rate of Douglas-Rachford alternating direction method of multipliers," *Numerishe Mathematik*, vol. 130, iss. 3, pp. 567-577, 2014.

[13] A. Nedic and A. Ozdaglar, "Cooperative distributed multi-agent optimization", in *Convex Optimization in Signal Processing and Communications*, (D. P. Palomar and Y. C. Eldar, eds.), Cambridge University Press, 2010.

[14] T. Goldstein, G. Taylor, K. Barabin, and K. Sayre, "Unwrapping ADMM: Efficient Distributed Computing via Transpose Reduction", *CoRR*, vol. abs/1504.02147, 2015.

[15] T. Goldstein, B. O'Donoghue, S. Setzer and R. Baranuik, "Fast Alternating Direction Optimization Methods", *SIAM Journal on Imaging Sciences*, vol. 7, iss. 3, pp. 1588-1623, 2014.

[16] E. Esser, *Applications of Lagrangian-Based Alternating Direction Methods and Connections to Split Bregman*, April 2009.