Mid Year Review: AMSC/CMSC 663 and 664

The Alternating Direction Method of Multipliers
An Adaptive Step-size 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

December 14, 2015

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 an adaptive step-size selection algorithm to optimize performance on every iteration. The library will contain a general ADMM method, as well as solvers for common problems that ADMM is used for. It will also implement adaptive step-size selection, utilize the Message Passing Interface (MPI) 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, with a brief example of how it is used to solve problems in practice. Next, we will discuss some ideas for adaptive step-size selection that we will try to implement and investigate. At the end, we discuss the design of the software library in greater detail, as well as well as how the project will proceed.

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 [8, 6], and by Benders [7]. However, Dual Decomposition's general idea is primarily due to Everett [9]. 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 [11, 12] and Powell [13]. 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:

$$\left|\begin{array}{ll} \text{Primal Feasibility:} & Ax_{opt} - b = 0 \\ \text{Dual Feasibility:} & \nabla f(x_{opt}) + A^T y_{opt} = 0 \end{array}\right.$$

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:

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

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 [15], and Glowinski, and Marrocco [14]. Many further findings in ADMM were discovered by Eckstein and Bertsekas [16]. 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)

```
1: procedure ADMM($A$, $B$, $c$, $\rho$, OBJECTIVE, ARGMINX, ARGMINZ, STOPCOND)
2:     Set $x$, $z$ and $y$ to some initial value.
3:     while STOPCOND($x$, $z$, $y$, $A$, $B$, $c$) $\neq 1$ do
4:         $x :=$ ARGMINX($x, z, y, \rho$)                (minimize $f(x)$ for $x$)
5:         $z :=$ ARGMINZ($x, z, y, \rho$)                (minimize $g(z)$ for $z$)
6:         $y := y + \rho(Ax + Bz - c)$                   (Dual variable update)
7:     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:

$$\left|\begin{array}{ll} \text{Primal Feasibility:} & Ax + Bz - c = 0 \\ \text{Dual Feasibility:} & \nabla f(x) + A^T y = 0, \\ & \nabla g(z) + B^T y = 0 \end{array}\right.$$

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, 16], 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

```
1: procedure ScaledADMM($A$, $B$, $c$, $\rho$, objective, argminx, argminz, stopcond)
2:     Set $x$, $z$ and $u$ to some initial value.
3:     while stopcond($x$, $z$, $u$, $A$, $B$, $c$) $\neq 1$ do
4:         $x :=$ argminx($x, z, u, \rho$)                    (minimize $f(x)$ for $x$)
5:         $z :=$ argminz($x, z, u, \rho$)                    (minimize $g(z)$ for $z$)
6:         $u := u + (Ax + Bz - c)$                          (Dual variable update)
7:     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, $B = -\mathrm{I}$, 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:

$$z_i^{(k+1)} := (v_i - \lambda/\rho)_+ - (-v_i - \lambda/\rho)_+ \tag{10}$$

We can do 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 [4] 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 [4], $\{||w^i - w^{i+1}||_H^2\}$ 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 [4] 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 $A = \mathrm{I}$, $B = -\mathrm{I}$ 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:

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

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 [6] and [10].

Algorithm 3 Distributed ADMM (Scaled and Unscaled Dual)

```
1: procedure DistributedADMM($A$, $B$, $c$, $\rho$, objective, argminx, argminz, stopcond)
2:     Set $x$, $z$ and $u$ to some initial value.
3:     while stopcond($x$, $z$, $u$, $A$, $B$, $c$) $\neq 1$ do
4:         for parallel machine labeled by index $i$ do
5:             $x_i :=$ argminx$(f_i(x) + (y_i)^T(x-z) + \frac{\rho}{2}||x-z||_2^2)$        (distributed $x$ update)
6:         $z :=$ argminz$((y_i)^T(x_i - z) + \frac{\rho}{2}||x_i - z||_2^2) = \frac{1}{n}\sum_{i=1}^n (x_i + \frac{1}{\rho}y_i) = \frac{1}{n}\sum_{i=1}^n (x_i + u_i)$
7:         for component index $i$ do
8:             $y_i := y_i + \rho(x_i - z) = \frac{1}{\rho}y_i + x_i - z = u_i + x_i - z$
9:     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 [5], 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 $D^{+} = (D^TD)^{-1}D^T$ (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:

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

With this observation, a central server needs only D^TD and D^Tb. For tall, large D, D^TD has much fewer entries. Furthermore, we can see that $D^TD = \sum_i D_i^TD_i$ and $D^Tb = \sum_i D_i^Tb_i$. 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^TD and D^Tb, 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.

The Implementation and Results of ADMM and Solvers

ADMM is capable of solving many general convex optimization problems. To name a few, according to [1]:

- Basis Pursuit
- Sparse Inverse Covariance Selection
- Huber Fitting
- Intersection of Polyhedra
- Least Absolute Deviations
- Linear Programming
- ℓ_1 Regularized Logistic Regression
- Regressor Selection (nonconvex)
- Quadratic Programming
- Lasso Problem
- Support Vector Machines (SVMs)
- Total Variation Minimization (e.g., image denoising)

One can read about the problem statements for all of these optimization problems, as well as ADMM solutions for them in [1]. For the first semester, we focused on getting a working generalized ADMM implementation with adaptive step-size selection, and solvers for the last three problems on the list. Solvers for these problems will set the groundwork for the rest on the problems, which are solved in very similar ways, but with differences in input and proximal operators. However, we first start with describing the general implementation of ADMM that was designed this semester.

General ADMM

We already have described a setup and algorithm for solving a general ADMM problem. The minimal inputs for this algorithm are the constraint matrices A and B, the constraint vector c, such that $Ax + Bz = c$, and the proximal operators $\mathbf{prox}_f$ and $\mathbf{prox}_g$. However, there can be more inputs. For example, the step size ρ, the absolute and relative errors for the standard stopping conditions, maximum number of iterations to perform, etc. Thus, the implementation was designed with accepting exactly three inputs: the two proximal operators for f and g (in that order), and a structure `options` that contains additional parameters the user has set. Each parameter has a default state if unset, save for the constraint variables, which must be set.

These options can allow the user to customize almost every step of the ADMM algorithm. They are also exceedingly helpful in programming the solvers - as each solver will use the generalized ADMM function. Simply customize general ADMM to the optimized way for solving a specific problem and call the function.

For validation and testing on this generalized ADMM function, we used the following model problem:

$$\arg\min_x(||Ax-b||_2^2+||Cx-d||_2^2), \text{ with } A,\, C \in \mathbb{R}^{n\times n} \text{ and } b,\, d \in \mathbb{R}^n \tag{26}$$

The derivative of the function to minimize is:

$$\frac{\delta}{\delta x}(||Ax-b||_2^2+||Cx-d||_2^2) = 2A^T(Ax-b)+2C^T(Cx-d) \tag{27}$$

Thus, setting this equal to 0 and solving for x yields the following exact minimizer:

$$x := (A^TA+C^TC)^{-1}(A^Tb+C^Td) \tag{28}$$

Evaluating at this minimizer gives the exact objective value. Thus, if we solve this problem with ADMM, we can check if the solution matches the true solution. In ADMM form, with $f(x) = ||Ax-b||_2^2$, $g(z) = ||Cz-d||_2^2$, the problem can be written as:

$$\arg\min_x(||Ax-b||_2^2+||Cz-d||_2^2), \text{ subject to } x-z=0 \tag{29}$$

Our scaled dual augmented Lagrangian for this problem is:

$$L_\rho(x,z,u) = ||Ax-b||_2^2+||Cz-d||_2^2+\rho/2||x-z+u||_2^2 \tag{30}$$

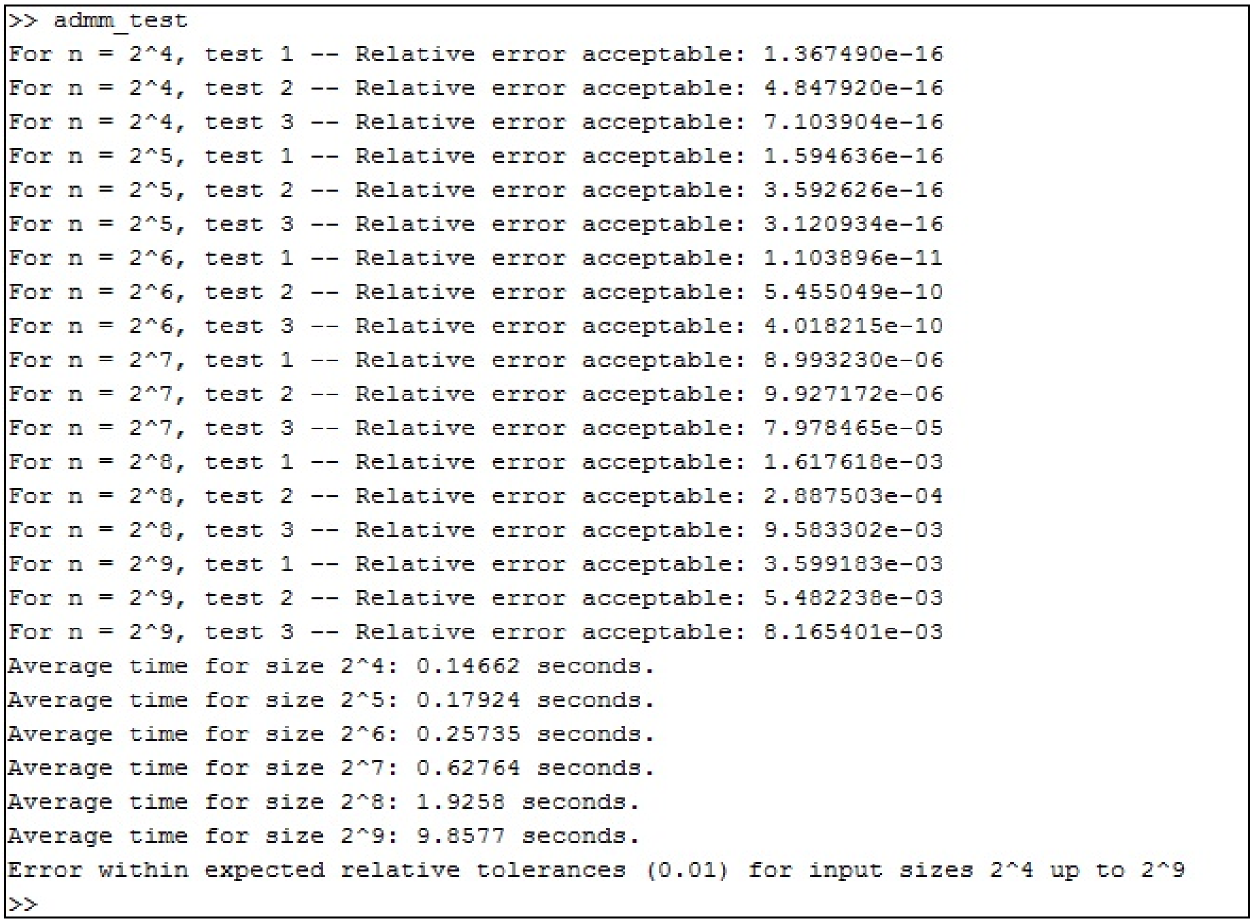
Thus, if we freeze the z and u variables at some iteration k, take the derivative for x, setting it to 0, and solving for x gives us the following expression for the x-minimization step (our $\mathbf{prox}_f$):

$$\mathbf{prox}_{f,\rho}(x,z^k,u^k) = (2A^TA+\rho I_n)^{-1}(2A^Tb+\rho(z^k-u^k)) \tag{31}$$

Likewise, for the z variable, with x frozen at iteration $k+1$ and u at k, we get the following minimizer:

$$\mathbf{prox}_{g,\rho}(x^{k+1},z,u^k) = (2C^TC+\rho I_n)^{-1}(2C^Td+\rho(x^{k+1}+u^k)) \tag{32}$$

Plugging these proximal operators in our algorithm, for some ρ, we can test to see if ADMM converges sufficiently to the true objective solution. The testing software generates random square matrices for A and C of size $n = 2^x$, with vectors c and d also generated randomly, for any range of positive integer x values specified. The objective and proximal operators are formulated from this random input. Then, the solution via ADMM is computed, as well as the true solution using (28). The ADMM result is checked to be within some predefined expected relative tolerance of the actual answer. These trials are run, timed, and checked for correctness multiple times for each size n, and the resulting times are averaged. Any results not with the expected tolerance are reported. Residual and objective plots are generated for the first trial of each size n, as well as a plot of the average run-times for each size.



```
>> admm_test
For n = 2^4, test 1 -- Relative error acceptable: 1.367490e-16
For n = 2^4, test 2 -- Relative error acceptable: 4.847920e-16
For n = 2^4, test 3 -- Relative error acceptable: 7.103904e-16
For n = 2^5, test 1 -- Relative error acceptable: 1.594636e-16
For n = 2^5, test 2 -- Relative error acceptable: 3.592626e-16
For n = 2^5, test 3 -- Relative error acceptable: 3.120934e-16
For n = 2^6, test 1 -- Relative error acceptable: 1.103896e-11
For n = 2^6, test 2 -- Relative error acceptable: 5.455049e-10
For n = 2^6, test 3 -- Relative error acceptable: 4.018215e-10
For n = 2^7, test 1 -- Relative error acceptable: 8.993230e-06
For n = 2^7, test 2 -- Relative error acceptable: 9.927172e-06
For n = 2^7, test 3 -- Relative error acceptable: 7.978465e-05
For n = 2^8, test 1 -- Relative error acceptable: 1.617618e-03
For n = 2^8, test 2 -- Relative error acceptable: 2.887503e-04
For n = 2^8, test 3 -- Relative error acceptable: 9.583302e-03
For n = 2^9, test 1 -- Relative error acceptable: 3.599183e-03
For n = 2^9, test 2 -- Relative error acceptable: 5.482238e-03
For n = 2^9, test 3 -- Relative error acceptable: 8.165401e-03
Average time for size 2^4: 0.14662 seconds.
Average time for size 2^5: 0.17924 seconds.
Average time for size 2^6: 0.25735 seconds.
Average time for size 2^7: 0.62764 seconds.
Average time for size 2^8: 1.9258 seconds.
Average time for size 2^9: 9.8577 seconds.
Error within expected relative tolerances (0.01) for input sizes 2^4 up to 2^9
>>
```

Figure 1: Results for a test of general ADMM on the model problem. Only 3 trials were performed for each size n, with $\rho = 5.0$, error tolerance 0.01 and convergence tolerance 0.001. Each trial performed 1000 iterations, unconditionally.

In Figure 1, we can see the results of one of these trials. Only 3 random trials were done per size of n so that the results could fit in the image. The test reports that all the results were within the expected tolerances and that all H-norms were monotonically decreasing with the default tolerance (relative error 0.001). However, we can see that the relative error for the true solution decreases drastically as the size of n increases. This means that the 1000 iterations performed for each trial are simply not enough for higher input sizes. The convergence rate tends to slow down.

Figure 2 shows the error norms and objective value for the first trial of size $n = 2^9$. The dashed lines are the relative/absolute tolerances for the primal and dual error norms. Once ADMM is below these dashed lines, it would have been considered to converge. As we can see, this convergence already happened around iteration 100. Furthermore, the objective value is hardly changing over the course of the algorithm. Thus, these results indicate that ADMM is slow to converge to precise values on this type of problem. Figure 3 shows the H-norm squared values between current and previous iterates. As expected, they are monotonically decreasing and tend to 0 as we get closer to the solution, demonstrating that they can be used for stopping conditions as well. If the proximal operators or the constraint matrices/vectors are changed at all in the model problem, the H-norm squared values were found to not be decreasing, and the algorithm terminated early, as desired. In official testing, the number of trials per each size n was set to 100, and the range of values were all power of 2 up to 2^9. The trials all gave positive results. Other features of ADMM were tested by changing the options arguments before running ADMM.

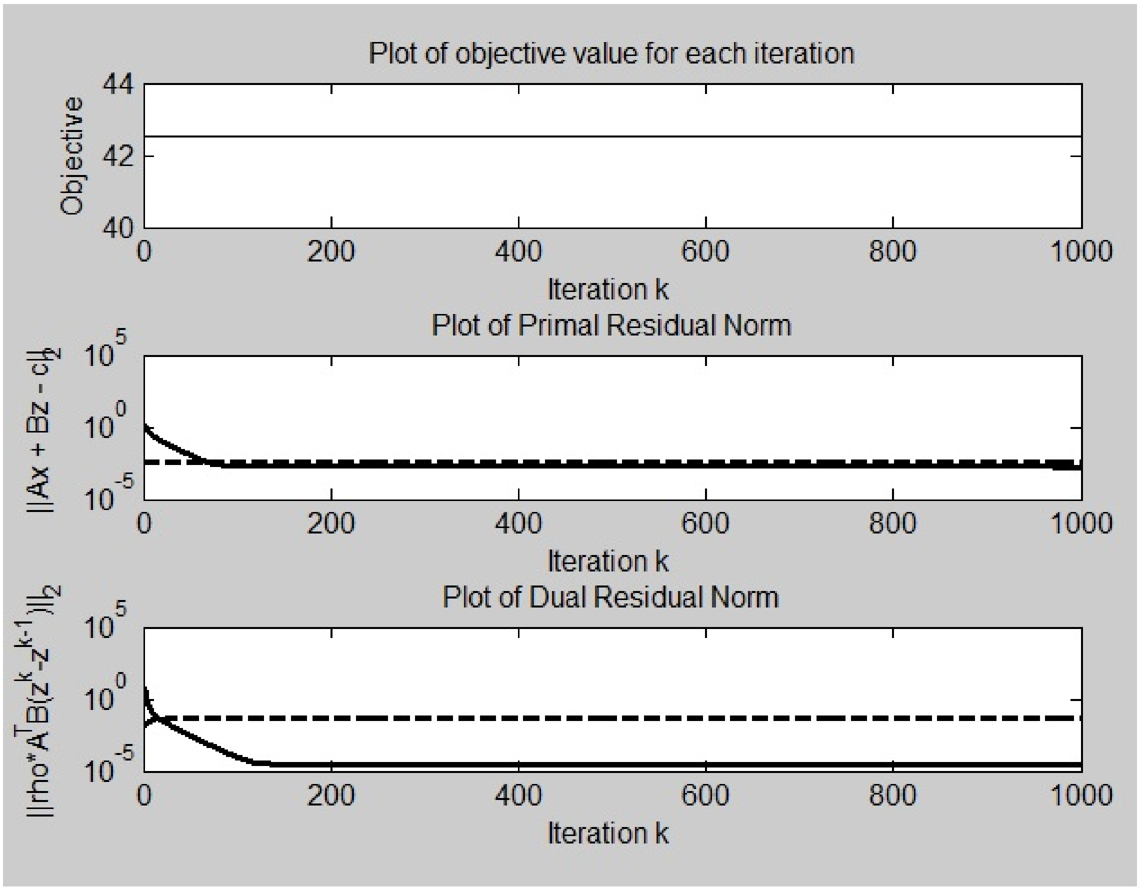


Figure 2: Results for a test of general ADMM on the model problem, for random input of size $n = 2^9$. The figure shows the progression of the objective value and the primal/dual error norms.

Total Variation Minimization

Total Variation (**TV**) measures the integral of the absolute value of the gradient of a signal x. Minimizing this integral eliminates small variations in the signal data but preserves large variations, thus removing noise from a noisy signal. Because of this, it has many applications. One example is image denoising in the 2-dimensional case.

In 1-dimension, TV is defined as $V(x) = \sum_i |x_{i+1} - x_i|$. We want to find a value close to a given signal component b_i that has smaller Total Variation. One way to measure such closeness is the sum of square errors: $E(x, b) = \frac{1}{2}||x - b||_2^2$. So our goal is to minimize for x, over a signal b:

$$E(x, b) + \lambda V(x) \tag{33}$$

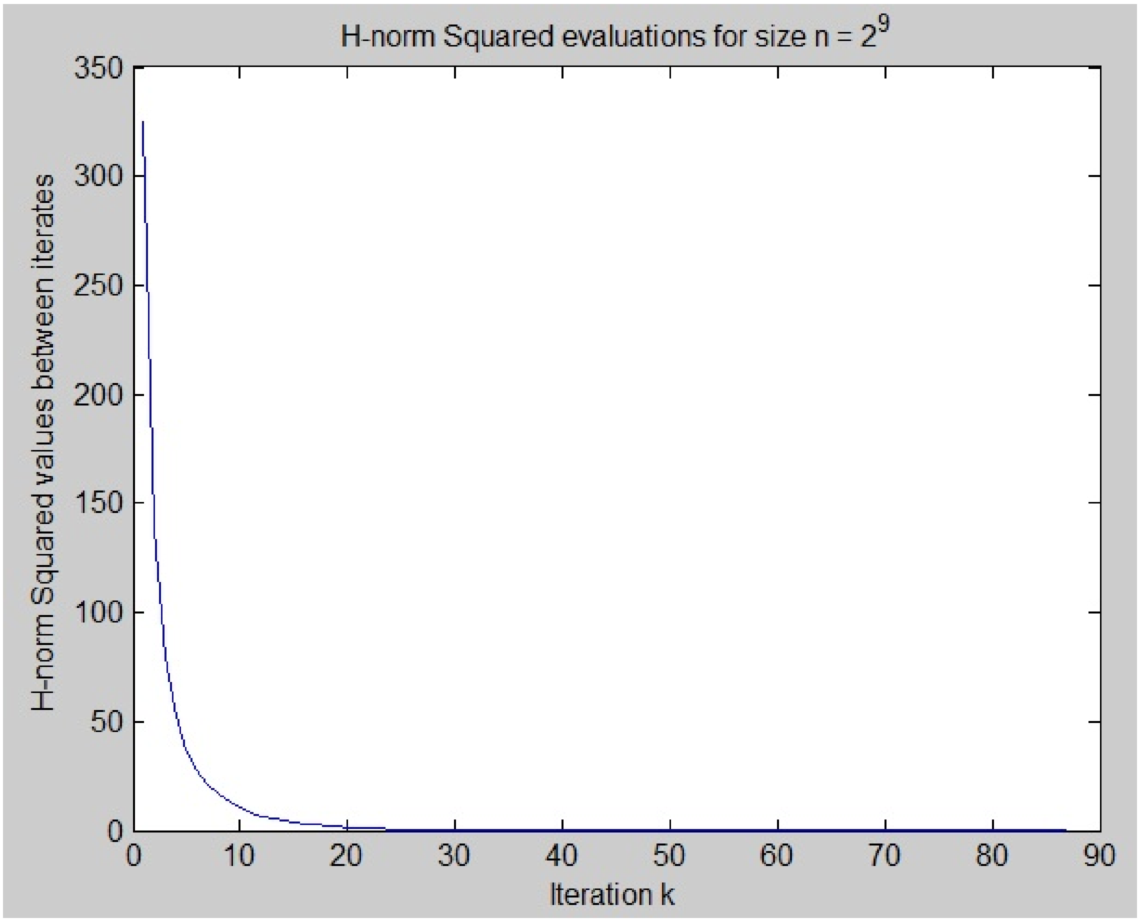


Figure 3: H-norm squared values for the same model random trial of size $n = 2^9$. We can see that these values are monotonically decreasing and tend to 0, as expected.

The λ here is called the *regularization parameter*, and it dictates how much of an impact the Total Variation has in the minimization by scaling it by λ. The problem, in ADMM form, can be written as:

$$\arg\min_x \left(E(x,b) + \lambda ||z||_1\right), \text{ subject to } Dx - z = 0 \tag{34}$$

First, let's figure our the x-minimization step. We see that the gradient can be computed by hand. So, using ADMM's Augmented Lagrangian we minimize x by:

$$\begin{aligned} \nabla_x L_\rho(x, z^{(k)}, u^{(k)}) &= \nabla_x E(x,b) + \nabla_x ||Dx - z^{(k)} + u^{(k)}||_2^2 \\ &= \nabla_x \frac{1}{2}||x-b||_2^2 + \rho D^T(Dx - z^{(k)} + u^{(k)}) \\ &= x - b + \rho D^T Dx - \rho D^T z^{(k)} + \rho D^T u^{(k)} = 0 \end{aligned} \tag{35}$$

We now solve (35) for x:

$$\begin{aligned} x - b + \rho D^T D x - \rho D^T z^{(k)} + \rho D^T u^{(k)} &= 0 \\ x + \rho D^T D x &= \rho D^T z^{(k)} - \rho D^T u^{(k)} + b \\ (\mathrm{I} + \rho D^T D)x &= \rho D^T (z^{(k)} - u^{(k)}) + b \\ x &= (\mathrm{I} + \rho D^T D)^{-1} (\rho D^T (z^{(k)} - u^{(k)}) + b) \end{aligned} \tag{36}$$

Thus, based off of the results in (36), our x-minimization step is:

$$x^{(k+1)} := (\mathrm{I} + \rho D^T D)^{-1} (\rho D^T (z^{(k)} - u^{(k)}) + b) \tag{37}$$

In real life, we would not form the inverse matrix in (37), but solve a system for x^{k+1}. The z and y minimization steps are as before for the general ADMM form, like in Algorithm 2. Thus, the algorithm for solving TV remains the same as Algorithm 2 but with (37) as the x-minimization step and soft-thresholding for the z step. This is shown in Algorithm 4

Algorithm 4 TV Minimization

```
1: procedure TVM($b$, $\rho$, $\lambda$, STOPCOND)
2:     Set $x$, $z$ and $u$ to some initial value.
3:     Form matrix $D$ using stencil $[1 - 1]$ (circular boundaries)
4:     while STOPCOND($x$, $z$, $u$, $D$ $b$) $\neq 1$ do
5:         $x := (\mathrm{I} + \rho D^T D)^{-1} (\rho D^T (z - u) + b)$
6:         $z := (v - \lambda/\rho)_+ - (-v - \lambda/\rho)_+$, with $v = -Dx - (u = y^{(k)}/\rho)$
7:         $u := u + Dx - z$
8:     return ($x$, $z$, Evaluate (16) for ($x$, $z$))
```

We tested this solver in the same way as the model problem. To figure out what the "true solution" is for a randomly generated data set, we used the Matlab constrained optimization solver `cvx`, which can be downloaded for free online. This solver can solve any linear constrained optimization problem (very slowly, however). Thus, we simply feed the problem to `cvx`, which gives us a good estimate for the true solution, and then we compare that to ADMM's solution in the same way as in the model problem. As in the model problem, all results came out positive for 100 trials at each size n.

Figure 4 shows the results of several trials of TV minimization on randomly generated signals (black). ADMM's results are shown in red. As we can see, the red signal is a denoised version of the original. The parameter λ affects how "closely" the denoised signal should resemble the original. Lower values dictate more lax conditions, while higher values require a more denoised result. The results also show that the step-size/penalty parameter ρ also has an effect on how closely the result should resemble the original signal.

LASSO Problem

The LASSO (Least Absolute Shrinkage and Selection Operator) method is a type of linear least squares minimization problem with which adds a least squares penalty via the ℓ_1 norm, as in TV minimization. One standard formulation is as follows:

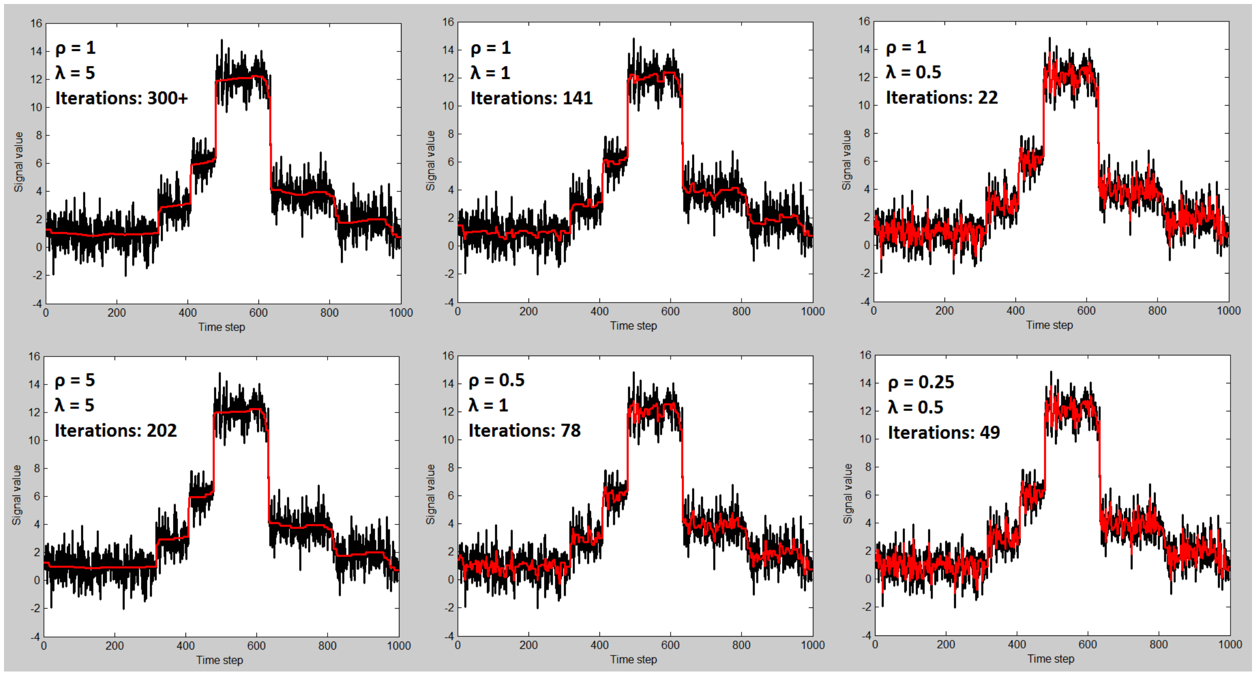


Figure 4: Results for a MATLAB implementation of TV Minimization using ADMM for various values of step-size ρ and λ.

$$\min_x \left(\frac{1}{2} ||Dx - b||_2^2 + \lambda ||x||_1 \right) \tag{38}$$

where $D \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$. The goal is to fit some data b to a linear model with the least amount of variance. The matrix D contains some kind of data by which we fit our variable x. This matrix could potentially be very tall (many training rows) or wide (many linear variables to fit).

Notice that the formulation of LASSO problem in (38) matches that of Transpose Reduction for Unwrapped ADMM. Thus, we can use this strategy to obtain the proximal operators for solving the LASSO problem. The solver is created directly from that. We test the LASSO solver using randomly generated D and b such that $b = D * x_0 + \epsilon$, where x_0 is a normally distributed, sparse, random vector, ϵ is a small perturbation vector which will generate something to minimize, and D is z-scored to fit with normal distribution. This way of constructing the random trials guarantees plausible linear solutions. Once again, we test the ADMM solution versus the solution computed via `cvx`, which we accept as the true solution. As for the model problem, we performed 100 trials for every power of 2 up to 2^7 (the size is limited as `cvx` is very slow to compute a true solution). Likewise, all trials had positive results.

Linear SVMs

Linear Support Vector Machines (SVMs) strive to classify linearly separable data via training data. That is, they seek a dividing hyperplane through some space of points such that all points are correctly classified, given some prior training examples of correct classification. If one does not want a dividing hyperplane that completely classifies everything, they can specify a regularization parameter that determines how "hard" the data should be classified. Such a general linear SVM can be formulated as solving the following minimization problem:

$$\min_x \left(\frac{1}{2}||x||^2 + Ch(Dx)\right) \tag{39}$$

for a given matrix D containing training data on each classification (each column is a category to classify into, each row a training sample). The parameter C here is a regularization parameter. If $C = 0$, then the goal is to get a completely dividing hyperplane for the data. The function h here is known as the Hinge-loss function, defined as:

$$h(w, (x, \ell)) = \max(0, 1 + \max_{\ell' \neq \ell}(w_{\ell'}x - w_{\ell}x)) := h(v) = \max_i(0, 1 - \hat{\ell}_i v_i) \tag{40}$$

for some dividing hyperplane specified by w, where x is some point to classify and ℓ is a set of labels on which we classify. The quantity $v = Dx$ encodes all the labeling information between w and x, so it can be written in a more concise, alternative form, as shown in (40) for a known $\hat{\ell}$ of labels where $+1$ denotes a positive classification and -1 a non-classification. These labels can be pre-computed from the training data. The Hinge-loss function is a differentiable, linear abstraction of the 0-1-loss function, which is a binary step function that returns only 0 (correctly classified) or 1 (incorrectly classified). Thus, being incorrectly classified returns a penalty of 1, and so minimizing over the 0-1 loss function minimizes the number of incorrect classifications. Similarly, the Hinge-loss attempts to do the same, but on what could be called a probability based scale.

Note that the formulation in (39) matches that of Unwrapped ADMM with Transpose reduction in (24). So, we can already determine how the proximal operator will look for this situation in the x variable. The z variable, which corresponds to the Hinge-loss function still needs to be figured out. However, notice that the formulation of Unwrapped ADMM decouples the minimization steps. It turns out that not only can minimization for the Hinge-loss be decoupled, but so can the minimization for the 0-1-loss function! In most SVMs, it is impossible to use the actual 0-1-loss function, so this is a unique feature of ADMM. For the Hinge-loss function, the minimization step is:

$$z = Dx + u + \ell \max(\min(1 - (\ell(Dx + u)), C/\rho), 0) \tag{41}$$

where the ℓ here is a vector of classification labels. For class i, $\ell(i) = \pm 1$, where a value of 1 indicates correct classification and -1 indicates incorrect classification for that class. For the 0-1-loss:

$$z = Dx + u + \ell \mathbb{I}_{01}(Dx + u, \rho/C) \tag{42}$$

where the function $\mathbb{I}_{01}(s, t)$ is vector indicator function that returns for every component of s the value of every case where $s \geq 1$ or $s < (1 - \sqrt{(2/t)})$. This essentially returns the component value in s wherever the label would be classified as incorrectly labeled. This is only possible because we've already decoupled for every unknown label the information in D by forming the pseudo inverse D^+. Thus, the z minimization steps only depend on row information in matrix D for each component.

The general strategy is to formulate the classes as columns and training samples as rows in the matrix D. Once the user has done that, they can run the SVM solver on their data D. ADMM will perform the training phase on each class from the training data, returning a vector x_{class} as a solution for each class (a one-vs-all scenario for each class). The user can then give testing samples which will be classified for the vector. This is done by computing the product between the test vector t and the computed x_{class}. Each component that is non-positive is considered to not belong in the class and each component that is positive is considered to be classified in that class.

For initial testing, we first tried classifying into two classes. The situation set up was to randomly generate points on the x-y plane such that half the points were above the line $y = x$, and half were below, with some points from the other class randomly entangled on the opposite side of the line. The solution to this problem would be if the SVM returned weights of 1 for both the x and y variables. Figure 5 shows a plot of one of the random results for $C = 100$. We can see that both the Hinge and 0-1 loss functions classify approximately along the identity function between the two variables, as expected.

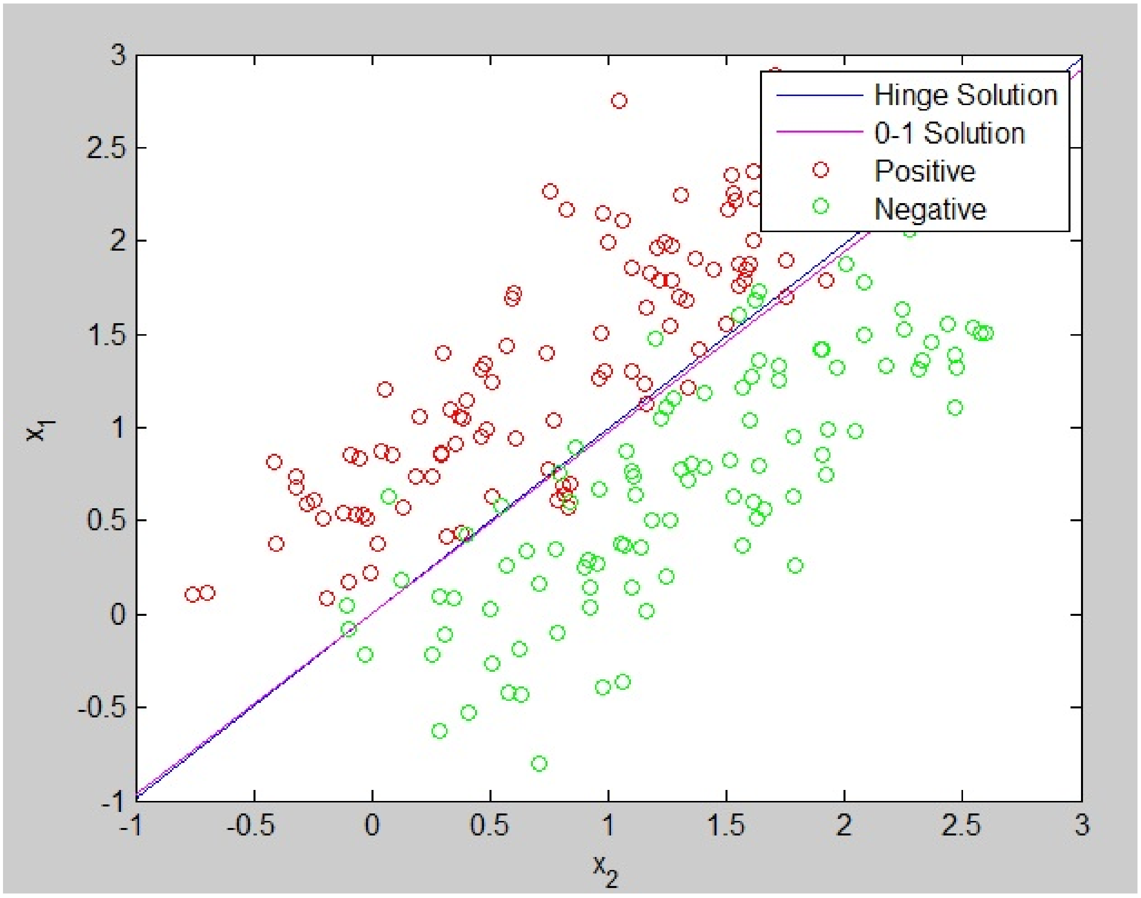
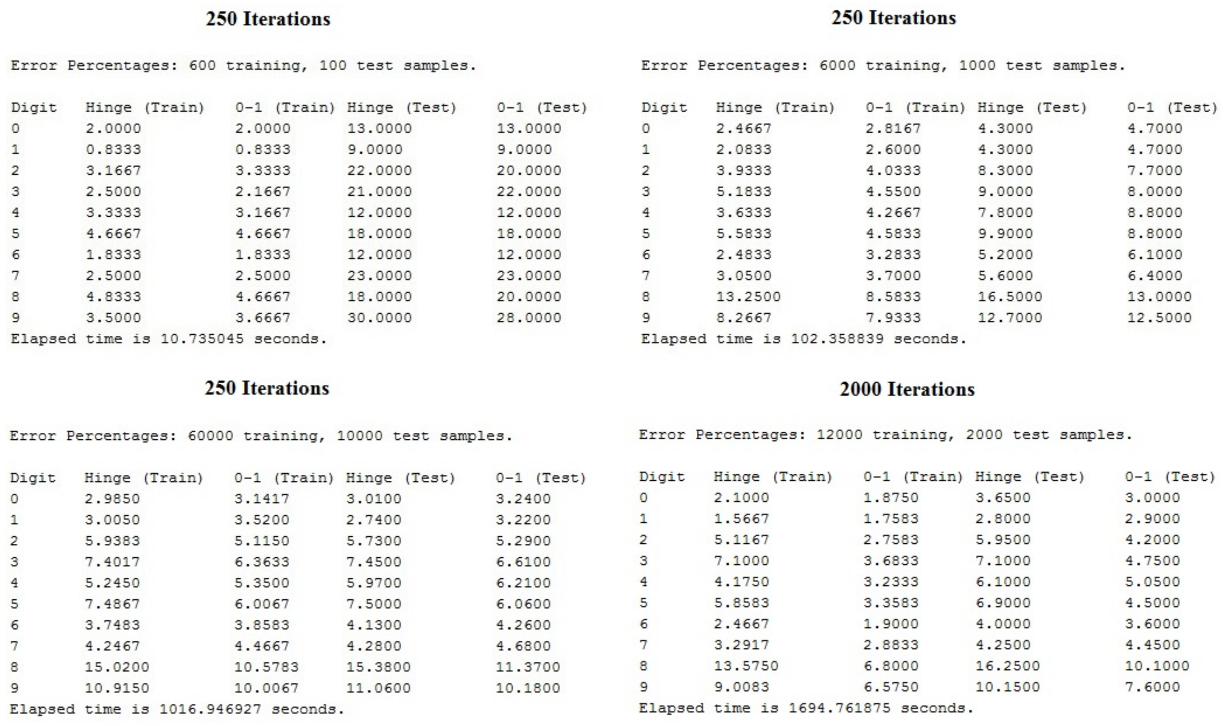


Figure 5: Results for binary classification with linear SVMs, with $C = 100$. Red and green dots are separate classes, with an approximate dividing line between most points of $x_1 = x_2$.

A more rigorous test of the linear SVM is to classify the digits in the MNIST handwritten database, which contains many hand-drawn binary images of the digits 0-9. This is a multi-classification problem. We assemble 10 columns in our D matrix which correspond to the the 10 digits, and try various subsets of the training and test data contained in the MNIST database. First we train a subset of the training data using ADMM, then we classify a subset of the testing data. We do this with both loss functions, and see how many were incorrectly classified for both functions, for both the testing and training data. Figure 6 shows results over various subsets of the data. Note that there are 60,000 training samples and 10,000 testing samples, with exactly 6,000 training samples and 1,000 testing samples per digit. We can see that for larger training samples, we tend to get lower percentages of incorrect results. If we increase the number of iterations, we can also improve the results. Note that there appears to be some difficulty classifying the digits 8 and 9. This is likely due to the fact that they resemble each other. We get decently fast run-times on this dataset and pretty good results, considering that the training and testing samples were randomly selected. It would be better to sample and train repeatedly for each digit using some sampling distribution as is often used to improve performance in machine learning. However, this was just a test to see if the SVM solver is indeed working correctly.



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 6: Percentages of incorrect results for linear SVM multi-classification of various subsets of testing and training samples of the MNIST database.

Adaptive ADMM

Adaptive ADMM would adaptively select a suitable step-size ρ each iteration, in an effort to decrease the number of iterations required to converge. Our strategy is based on results from Ernie Esser's paper in [2]. 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:

$$\max_{u\in\mathbb{R}^d}\left(\inf_{x,z\in\mathbb{R}^{m_1,m_2}}(L(x,z,u))\right) \tag{43}$$

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:

$$\frac{r - r^k}{\rho} = \frac{r^{k+1} - r^k}{\rho_k} \iff r = r^k + (r^{k+1} - r^k)\frac{\rho}{\rho_k} \iff \min_{\rho}(r^k + (r^{k+1} - r^k)\frac{\rho}{\rho_k}) \tag{44}$$

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:

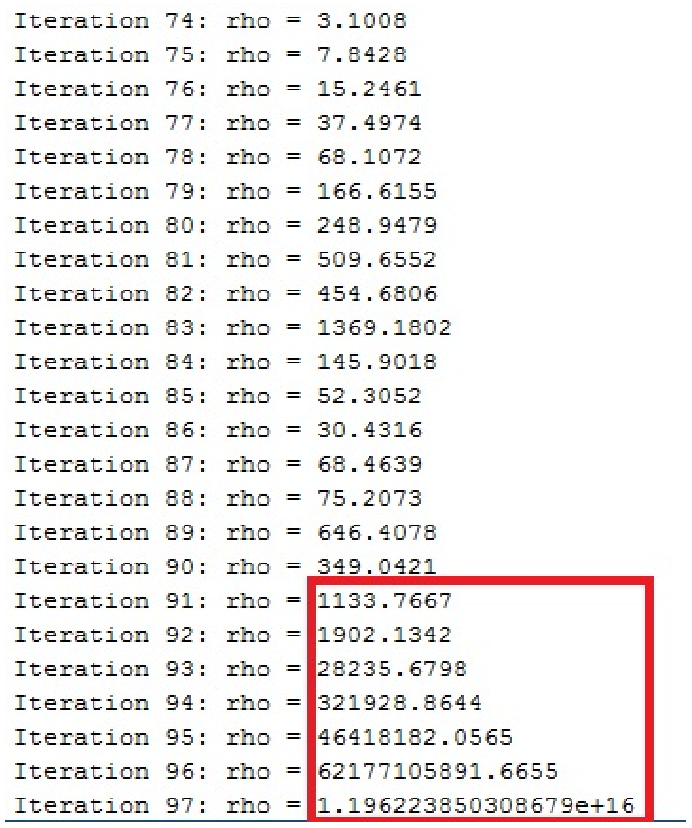
$$\rho_{k+1} = -\frac{\rho_k \hat{r}^T r^{k-1}}{\hat{r}^T \hat{r}} \tag{45}$$

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 7 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 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.

One option to try would be to select ρ_{k+1} in some clever way from previous values of ρ. Perhaps the linear model we minimize is insufficient and we should try to incorporate multiple values to interpolate through. However, this may be too costly an approach. Another option is to interpolate the H-norm evaluations. They are monotonic and decreasing, generally in a very smooth way, as shown in Figure 7. The interpolation step might give results that are less wild, avoiding the sudden explosion or disappearance of ρ values. Once more, a linear model might be insufficient here as well, since Figure 7 shows a convergence pattern of $O(1/k)$ that is described in [4].



```
Iteration 74: rho = 3.1008
Iteration 75: rho = 7.8428
Iteration 76: rho = 15.2461
Iteration 77: rho = 37.4974
Iteration 78: rho = 68.1072
Iteration 79: rho = 166.6155
Iteration 80: rho = 248.9479
Iteration 81: rho = 509.6552
Iteration 82: rho = 454.6806
Iteration 83: rho = 1369.1802
Iteration 84: rho = 145.9018
Iteration 85: rho = 52.3052
Iteration 86: rho = 30.4316
Iteration 87: rho = 68.4639
Iteration 88: rho = 75.2073
Iteration 89: rho = 646.4078
Iteration 90: rho = 349.0421
Iteration 91: rho = 1133.7667
Iteration 92: rho = 1902.1342
Iteration 93: rho = 28235.6798
Iteration 94: rho = 321928.8644
Iteration 95: rho = 46418182.0565
Iteration 96: rho = 62177105891.6655
Iteration 97: rho = 1.196223850308679e+16
```

Figure 7: An example of the explosion of ρ values generated via adaptive ADMM.

Project Schedule and Milestones

Below is the envisioned schedule for the project. The bold and italicized entries are milestones that can objectively measure the progress of the project.

- **Fall Semester Goals:**
  - **End of October:** ***Implement generic ADMM, solvers for the Lasso problem, TV Minimization, and SVMs.***
  - **Early November:** Implement scripts for general testing, convergence checking, and stopping condition strategies.
  - **End of November:** Find a working adaptive step-size selection implementation.
  - **Early December:** Finalize bells and whistles on ADMM options. ***Compile testing and validation data.***
- **Spring Semester Goals:**
  - **End of February:** ***Implement the full library of standard problem solvers.***
  - **End of March:** ***Finish implementing MPI in ADMM library.***
  - **End of April:** Finishing porting code to Python version.
  - **Early May:** ***Compile new testing/validation data.***

Apart from getting a working implementation of adaptive ADMM, all of the items on the checklist for the first semester have been finished. We plan to improve our results for adaptive ADMM to get something that works better during the Winter break, so that we do not fall behind schedule.

Deliverables

Below is a list of expected deliverables at the end of the project:

- **ADMM Library: Python and Matlab versions**
  - Contain general ADMM with adaptive step size routines and standard solvers for common problems ADMM solves.
  - Scripts for generating random test data and results.
  - Scripts for validating performance of adaptive ADMM to regular ADMM.
- Report on observed testing/validation results and on findings with adaptive ADMM - may lead to a paper eventually.
- Datasets used for testing the standard solvers (or references to where to obtain them, if they are too big).

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

[3] T. Goldstein, C. Studer and R. G. Baraniuk, "A Field Guide to Forward-Backward Splitting with a FASTA Implementation", *CoRR*, arXiv:1411.3406, Nov. 2014.

[4] 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.

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

[6] G. B. Dantzig and P. Wolfe, "Decomposition principle for linear programs", *Operations Research*, vol. 8, pp. 101111, 1960.

[7] J. F. Benders, "Partitioning procedures for solving mixed-variables programming problems", *Numerische Mathematik*, vol. 4, pp. 238252, 1962.

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

[9] H. Everett, "Generalized Lagrange multiplier method for solving problems of optimum allocation of resources", *Operations Research*, vol. 11, no. 3, pp. 399417, 1963.

[10] 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.

[11] M. R. Hestenes, "Multiplier and gradient methods", *Journal of Optimization Theory and Applications*, vol. 4, pp. 302320, 1969.

[12] 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.

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

[14] 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. 4176, 1975.

[15] 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. 1740, 1976.

[16] 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. 293318, 1992.