Imperial College London

Creating a Parallel Alternating Direction Method of Multipliers Solver for Relaxations of Sparse Polynomial Optimisation Problems

Aron Teh · Giovanni Fantuzzi

Aeronautical Engineering Final Year Thesis

1 June 2021

Abstract

The search for efficient and accurate solutions to constrained and unconstrained polynomial optimisation problems spans across countless applications and industries. The non-linear nature of such polynomial optimisation problems make exact solutions very difficult to attain at reasonable computational costs, therefore such problems are commonly recast into convex optimisation problems through polynomial variable relaxation. Algorithms for solving convex optimisation problems already exist, however it is of utmost importance that these algorithms are efficient and scalable to very large problems. One promising solution is the alternating direction method of multipliers (ADMM), which solves the conic problem by updating two sets of variables of an augmented Lagrangian in alternating fashion. This paper explores the special case in which a polynomial optimisation problem is relaxed into a convex problem, resulting in a particularly sparse conic problem structure. We attempt to exploit the sparsity of the problem to create a more scalable, parallelisable algorithm for the solution to these convex optimisation problems. A detailed exploration of the algorithm was conducted, and a working prototype solver was coded using Python. The sparse ADMM solver was tested with convex problems of varying sizes and relaxation orders, and its performance and convergence characteristics were compared to that of a benchmark dense ADMM solver. Since the prototype sparse solver did not utilise multiprocessing and parallel computing, it was found that the original dense solver showed overall superior performance. However, it is argued that a optimised parallel, high performance implementation of the sparse solver still has good potential to surpass the benchmark solver, and several optimisation suggestions are proposed for further work and research.

Acknowledgements

The majority of my thanks must be given to Dr. Giovanni Fantuzzi for his patient guidance and support for me throughout the course of this project. His technical expertise on this subject motivated the existence and purpose for the project, and it is only through his invaluable advice that this project was made possible.

Additionally, I would like to thank Prof. Sergei Chernyshenko for his advice and feedback during and after the completion of this project. The encouraging support from my supervisors allowed for the successful completion of this challenging yet highly intellectually stimulating final year thesis.

Contents

1 Introduction

1.1 Context and Motivation

With the world's ever-evolving computational capabilities, we are becoming increasingly reliant on the use of data-driven methods in solving the most challenging problems of today. Data storage capacities increasing at exponential rates, and thus we are becoming capable of running statistical and machine learning algorithms on indeterminately massive datasets [4]. Many of such modern algorithms are related to some form of optimisation, the balance of advantages and disadvantages in a highly complex and nonlinear multivariate system. It is vital that optimisation algorithms developed for the world today are efficient and horizontally scalable to arbitrarily large applications.

Among the most common of optimisation problems is the polynomial optimisation problem, which minimises a polynomial constructed with a vector of independent variables. In the general case, the objective to be optimised will also be subject to a set of polynomial constraints; a constrained optimisation problem. It can be intuitively felt that at a small scale, such a problem with only a few independent variables can be solved very quickly. However, at the same time it can also be inferred that as the number of variables and cross couplings increase, the problem complexity increases exponentially, very quickly surpassing the practicalities of any brute force approach.

With direct analytical solutions out of reach for larger optimisation problems, it becomes necessary to search for alternative reformulations to the original problem. One common approach is to recast the problem into a convex optimisation problem through the careful introduction of relaxation variables. The reformulated problem is a simpler, linear, convex problem which is still capable of representing the characteristics of the original polynomial optimisation problem. This paper begins with the exploration of this convex optimisation problem and its existing algorithm solutions.

1.2 Project Objectives and Scope

The paper's overarching goal is to introduce and develop a new parallelisable algorithm to solve special cases of the convex optimisation problem where the formulation results in a problem with high sparsity. The mathematical background leading up to the end objective is quite vast, and thus it is very much necessary to explore this wealth of theory before proceeding to the final algorithm. This algorithm and its similar predecessors have been studied and developed for more than 60 years, and it is through the understanding of each major development that one can obtain the most appreciation for the purpose of this specific algorithm. This paper will contribute the following in order:

1. The generic polynomial optimisation problem is introduced, and it is briefly argued that a sufficiently large problem of this nature quickly becomes impossible to solve directly.

2. The process in which the problem is reformulated into a conic optimisation problem through polynomial variable relaxation is presented in detail, and the resulting formulation will make up the core problem to be solved.

3. Predecessors to the alternating direction method of multipliers solution is presented and discussed, alongside several previous implementations and theory for the ADMM algorithm.

4. The motivation for a new parallel solver for sparse polynomial optimisation problems is given and the resulting algorithm and approach is described in detail.

5. A practical implementation of the described algorithm is programmed using Python and used to solve numerous example problems of varying size. The algorithm's performance is then finally compared to that of a benchmark solver, and the results are discussed in detail.

6. The paper concludes with suggestions for further research and exploration, along with a short project summary and reflection.

2 Technical Background

The section below will provide the reader with all the necessary mathematical background for understanding the origins and purpose of the implemented ADMM algorithm. No prior mathematical knowledge beyond elementary linear algebra and matrix arithmetic will be required to follow this section.

2.1 The Polynomial Optimisation Problem

The presence of polynomial optimisation problems are truly far-reaching across a variety of industries. In finance, financial portfolio optimisation can be modelled as a polynomial optimisation problem [32]. In the field of engineering, optimisation problems more than regularly arise from the discretisation of nonlinear partial differential equations [9]. All of such problems take the following general form:

$$\begin{aligned} f^* = \min_{\vec{x}\in\mathbb{R}^n} \; & f(x) \\ s.t. \quad & g_i(x) \geq 0, \quad i = 1, ..., N_g \\ & h_j(x) = 0, \quad j = 1, ..., N_h \end{aligned} \tag{1}$$

In the above formulation, $\vec{x} \in \mathbb{R}^n$ represents a vector of independent variables to be optimised, $f(x)$ is the polynomial objective function whose value is to be minimised, and f^* is the optimal (minimum) value of $f(x)$, and is assumed to exist. The constraints $g_i(x)$ and $h_j(x)$ are a collection of polynomial inequality and equality constraints respectively that restrict the admissible set of the independent variables.

Numerous optimisation algorithms already exist that allow for the determination of the local minima of such problems. Examples include Newton's method, the gradient method, and various conjugate direction methods, all of which are well-studied and documented in research [3]. However, due to their nature and dependence on the gradient at each iteration, these methods are only capable of finding local minima for the problem, whereas one is usually far more interested in the global minimiser.

For sufficiently large problems, it quickly becomes impossible to check whether a determined local minimiser is indeed the global minimiser for the problem. This motivates a reformulation of the original problem into one that can hopefully allow for the computational determination of a global optimal solution.

2.2 Convex Optimisation Problem Reformulation

2.2.1 Variable Relaxation for Polynomial Optimisation Problems

It may be most useful to follow a practical example in order to best explain the process of variable relaxation and the introduction of slack variables. For the following section, consider the polynomial optimisation problem:

$$
\begin{aligned}
f^* = \min_{\vec{x}\in\mathbb{R}^2} \;& x_2 \\
s.t. \quad & x_1^2 + x_2^2 - 1 \geq 0 \\
& -x_1^2 + x_2 \geq 0
\end{aligned} \tag{2}
$$

As a small problem, the global solution is straightforward, and can be found using the gradient method, or simply by inspecting the graph of the inequality constraints as seen in Figure 1.

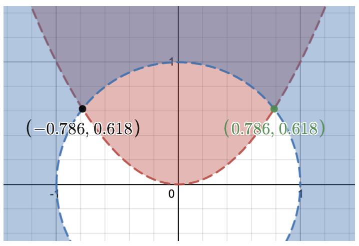


Figure 1: Example Problem

The constraints clearly represent the unit circle and a simple parabola. The optimal solutions are: $(\pm\sqrt{\frac{\sqrt{5}-1}{2}}, \frac{\sqrt{5}-1}{2})$, and the optimal value of the objective function is $f^* = \frac{\sqrt{5}-1}{2}$.

The objective function and constraints of the problem are all polynomials, each of which are made up of individual coupled monomials. The first step in the reformulation is to identify the highest order monomials present in the objective function, and generate an appropriate monomial vector containing the necessary monomials to build up the original problem [22]. In general, if the highest degree monomial is of order n, then it is necessary to form a vector of monomials with order up to and including $\omega = \frac{n}{2}$. The reason for this selection will soon become clear. In this example, the highest order monomial is of degree 2, therefore we select all possible monomials up to order 1 as entries for the monomial vector:

$$\vec{v}_1(x_1, x_2) = \begin{bmatrix} 1 & x_1 & x_2 \end{bmatrix}$$

Next, this monomial vector is multiplied with its transpose to form a moment matrix:

$$M_1(x_1, x_2) = \vec{v}_1\,\vec{v}_1^T = \begin{bmatrix} 1 & x_1 & x_2 \\ x_1 & x_1^2 & x_1x_2 \\ x_2 & x_1x_2 & x_2^2 \end{bmatrix}$$

The justification for the choice of monomial vector order from earlier is realised; multiplication by its own transpose has doubled the degree of the highest order monomial, and as a result the moment matrix will always contain all monomials included within the original polynomial optimisation problem.

The moment matrix clearly includes some non-linear terms, terms that we hope to remove in order to set up a linear convex optimisation problem. We simply introduce a new set of variables $\vec{y}$, and map each unique monomial to a new independent variable in y:

$$\begin{bmatrix} x_1 & x_2 & x_1^2 & x_1x_2 & x_2^2 \end{bmatrix} \longrightarrow \begin{bmatrix} y_0 & y_1 & y_2 & y_3 & y_4 \end{bmatrix}$$

The mapping converts the above moment matrix in terms of the original x_i variables into an equivalent matrix with the new slack variables y_i.

$$\begin{bmatrix} 1 & x_1 & x_2 \\ x_1 & x_1^2 & x_1x_2 \\ x_2 & x_1x_2 & x_2^2 \end{bmatrix} = \begin{bmatrix} 1 & y_0 & y_1 \\ y_0 & y_2 & y_3 \\ y_1 & y_3 & y_4 \end{bmatrix}$$

The exchange of variables clearly comes with the loss of some information about the problem; for example, the fact that $x_1 \cdot x_2 = x_1x_2 \rightarrow y_0 \cdot y_1 = y_3$ is no longer encoded. However, this symmetric moment matrix has an interesting property that

does encode some information about how these new slack variables related to one another. Specifically, this moment matrix is positive semidefinite, and this can be quickly seen by considering the following. If a matrix is positive semidefinite:
$\vec{z}^T \cdot M(\vec{x}) \cdot \vec{z} \geq 0$ for all nonzero $\vec{z}$. Rewriting the expression, $\vec{z}^T M(\vec{x}) \vec{z} = (\vec{z}^T \cdot \vec{v})(\vec{v}^T \cdot \vec{z}) = (\vec{v}^T \cdot \vec{z})^2 \geq 0$ [12]. Having shown this, the positive semidefiniteness of the moment matrix will form one of the constraints for the reformulated convex problem.

The next step is to encode the constraints of the polynomial optimisation problem using a similar approach. Consider the constraint $-x_1^2 + x_2 \geq 0$. If we perform an element-wise multiplication of this expression to the original moment matrix we create another moment matrix as seen below [22].

$$M_c(x_1, x_2) = (-x_1^2 + x_2) \cdot M_1(x_1, x_2) = \begin{bmatrix} -x_1^2 + x_2 & -x_1^3 + x_1 x_2 & -x_1^2 x_2 + x_2^2 \\ -x_1^3 + x_1 x_2 & -x_1^4 + x_1^2 x_2 & -x_1^3 x_2 + x_1 x_2^2 \\ -x_1^2 x_2 + x_2^2 & -x_1^3 x_2 + x_1 x_2^2 & -x_1^2 x_2^2 + x_2^3 \end{bmatrix} \succeq 0$$

The non-negativeness of the constraint and positive semidefiniteness of the original moment matrix combine implies that this new constraint-derived moment matrix must also be positive semidefinite. The original inequality constraint has therefore been encoded into a positive semidefinite constraint for the reformulated convex optimisation problem. More slack variables y_i would need to be introduced to cover all the corresponding monomials present in these constraints.

The encoding of equality constraints follows a similar intuition. Now, consider a separate problem with only a single independent variable x, and the equality constraint $(x-2)(x-3) = 0$. Also consider this vector of monomials with arbitrary size $\vec{v} = \begin{bmatrix} 1 \; x \; x^2 \; ... \; x^d \end{bmatrix} = \begin{bmatrix} 1 \; y_0 \; y_1 \; ... \; y_{d-1} \end{bmatrix}$. If d is chosen such that when the constraint expression is multiplied element-wise with $\vec{v}$, the resulting vector contains monomials of order no higher than those of the original moment matrix, we will have created a set of $d+1$ equality constraints that will constrain the new convex optimisation problem [12]. Following through with this example, the equality constraints will be:

$$(x-2)(x-3) \cdot \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^d \end{bmatrix} = \begin{bmatrix} x^2 - 5x + 6 \\ x^3 - 5x^2 + 6x \\ x^4 - 5x^3 + 6x^2 \\ \vdots \\ x^{d+2} - 5x^{d+1} + 6x^d \end{bmatrix} = \begin{bmatrix} y_1 - 5y_0 + 6 \\ y_2 - 5y_1 + 6y_0 \\ y_3 - 5y_2 + 6y_1 \\ \vdots \\ y_{d+1} - 5y_d + 6y_{d-1} \end{bmatrix} = 0$$

This set of inequalities can be rewritten into the matrix form $c - A^T y = 0$, and will encode information about the original single equality constraint in terms of these new y variables.

2.2.2 Higher Order Monomial Terms and the Lasserre Hierarchy

It was earlier mentioned that the size of the monomial vector must be selected such that the highest order monomial in the vector must be at least, but not exactly, half of the highest order monomial appearing in the objective function and constraints combined. This is to ensure the resulting moment matrix $M = \vec{v} \cdot \vec{v}^T$ will contain all monomials which appear in the problem. However, there is no restriction for how many terms one can choose to include in the monomial array, and it is now useful to mention the impact of including more or fewer terms in the array for the overall solution.

As a quick demonstration, take the same example problem from the above section. Although a monomial vector with order 1 monomials sufficed, one could just as easily have chosen to build the moment matrix using monomials of up to degree 2.

$$\vec{v}_2(x_1,x_2) = \begin{bmatrix} 1 & x_1 & x_2 & x_1^2 & x_1x_2 & x_2^2 \end{bmatrix}$$

Understandably, the resulting moment matrix will be considerably larger, as seen below.

$$M_2(x_1,x_2) = \vec{v}_2\,\vec{v}_2^T = \begin{bmatrix} 1 & x_1 & x_2 & x_1^2 & x_1x_2 & x_2^2 \\ x_1 & x_1^2 & x_1x_2 & x_1^3 & x_1^2x_2 & x_1x_2^2 \\ x_2 & x_1x_2 & x_2^2 & x_1^2x_2 & x_1x_2^2 & x_2^3 \\ x_1^2 & x_1^3 & x_1^2x_2 & x_1^4 & x_1^3x_2 & x_1^2x_2^2 \\ x_1x_2 & x_1^2x_2 & x_1x_2^2 & x_1^3x_2 & x_1^2x_2^2 & x_1x_2^3 \\ x_2^2 & x_1x_2^2 & x_2^3 & x_1^2x_2^2 & x_1x_2^3 & x_2^4 \end{bmatrix}$$

The rest of the problem formulation can follow from this point without issue. The new higher order terms simply couple together with the lower order terms to encode more information about the original problem. This systematic addition of monomial terms by increasing the order of the monomial vector is known as the Lasserre hierarchy for approximation algorithms [25].

2.2.3 Relation between Original and Reformulated Problem

It is clear that the reformulated problem is characteristically very different from the initial polynomial optimisation problem. The new slack variables which are to be optimised are not the original, and will never perfectly match up with the monomials after optimisation. For example, if a relaxation introduced the slack variables $x, x^2 \to y_0, y_1$, the y_i variables will be optimised without the knowledge that actually $y_1 = y_0^2$, and the final optimised solution will not satisfy this equality. The question is: how is the new linear convex problem related to the original, nonlinear polynomial optimisation problem?

In a brief statement, the reformulated convex problem solves for a lower bound of the objective function relative to the original problem. In other words, if the optimal minimiser for the polynomial optimisation problem is f^*, the convex formulation will find an $f_i \leq f^*$ that will generally be close to f^* [17]. The mathematical justification for this statement will not be provided here; however it is covered with detail in Lasserre's research paper at the following reference [17]. Moreover, the inclusion of higher order terms is shown, under certain conditions, to recover a solution that is closer and closer to the original global minimiser, i.e. $f_1 \leq f_2 \leq \ldots \leq f_\omega \leq f_{\omega+1} \leq \ldots \leq f^*$, where ω is the degree of the highest order term within the monomial vector [12].

It would be ideal for solution accuracy to choose a very high ω, however as seen in the above sections, the number of monomials and the size of the moment matrices increase at a polynomial rate with increasing monomial degree. Therefore, the rapidly increasing computational cost with increasing ω can quickly outdo the benefits of a very slightly more accurate solution, and thus it is important to gauge ones computational resources before attempting to solve a large problem of this nature.

2.2.4 Notes on Conic Problems

The fact that the reformulated problem is conic and therefore convex gives it several very useful properties, and it would be appropriate now to briefly discuss some of the unique characteristics of cones and the conic problem.

In a non-mathematical sense, one intuitively visualises the shape of a cone as that of an ice cream cone. Put into equations, this shape is represented by the following function [20].

$$\left\{x \in \mathbb{R}^3 : x_3 \geq \sqrt{x_1^2 + x_2^2}\right\} \tag{3}$$

Visually, one can already infer some of the conveniences of a conic problem structure. Such a cone, at every point on its surface, has a single gradient, all pointing directly towards a unique global minimum.

The mathematical definition of a cone is more complex. It is the set of points in Euclidean space with the property that: if a point $\vec{x}$ belongs to the cone, the point $\lambda\vec{x}$ also belongs to the cone, for any $\lambda \in \mathbb{R}_+$ [20]. This extends out to an arbitrary number of dimensions, and considering the arbitrary set $S \subset \mathbb{R}^n$, the subset

$$C(S) = \left\{(x, \lambda) \in \mathbb{R}^n \times \mathbb{R}_+ : x = \lambda x' \,\forall\, x' \in S\right\} \tag{4}$$

is a cone. A proper cone is a subset of the cone that takes a particular type of structure; it is convex, closed, pointed and full-dimensional [20]. There are several types of proper cones, but the types of concern for this problem are the zero cone, simply containing the single origin point $\mathcal{K} = \{0\}$; the non-negative orthant, a subset containing vectors with non-negative elements $\mathcal{K} = \{z \in \mathbb{R}^n : z_i \geq 0 \,\forall\, i \in \mathbb{Z}^+\}$; and most importantly the positive semidefinite cone, mathematically represented by the following expression [8].

$$\left\{X \in S^n : v^T X v \geq 0 \,(\forall v \in \mathbb{R}^n)\right\} \tag{5}$$

All constraints formulated using polynomial variable relaxation take one of the three conic forms. Since a general polynomial optimisation problem will have multiple of such cones within its constraints, in order to correctly formulate a conic problem we make the following important proposition. If we have several sets $\mathcal{K}_1 \subset \mathcal{Z}_1, \mathcal{K}_2 \subset \mathcal{Z}_2, \cdots, \mathcal{K}_s \subset \mathcal{Z}_s$, all of which are cones, then the set $\mathcal{K} = \mathcal{K}_1 \times \mathcal{K}_2 \times \cdots \mathcal{K}_s$ with tuples $(z_1, z_2, \cdots, z_s)$ such that $z_i \in \mathcal{K}_i$ is also a cone in the product space $\mathcal{Z}_1 \times \mathcal{Z}_2 \times \cdots \times \mathcal{Z}_s$. In other words, the product of multiple conic sets is also a conic sets, so any polynomial optimisation problem with arbitrary constraints can be reformulated into a convex optimisation problem.

2.3 Computational Implementation

2.3.1 Reformulated Problem Structure

All the necessary mathematical background has been covered up to this point, and next it is logical to discuss how such a problem can be computationally stored in code. The convex optimisation problem now takes the form:

$$\begin{aligned} \min_{y \in \mathbb{R}^m,\, z \in \mathbb{R}^n} \quad & -b^T y \\ s.t. \quad & c - A^T y = z \\ & z \in \mathcal{K} \end{aligned} \tag{6}$$

where $y \in \mathbb{R}^m$ is the vector of independent slack variables, $\mathcal{K}$ represents an arbitrary direct product of zero, non-negative orthant and positive semidefinite cones, $c - A^T y$ are the various constraints written in matrix form, and $-b^T y$ is the cost function to be minimised.

To capture the entire problem, it is of course necessary to store the matrix A and vectors b and c. However, we will need a consistent way of storing multiple types of constraints within the A matrix, as well as encoding the conic structure $\mathcal{K}$. We achieve this by storing each constraint in the following consistent order:

1. Equality constraints

2. Inequality constraints
3. Vectorised positive semidefinite constraints

Programmatically, we can store an extra `K` object with three attributes: `K.f`, the number of equality constraints; `K.l`, the number of inequality constraints; and `K.s`, a vector with each entry representing the size of each corresponding positive semidefinite constraint.

2.3.2 Storing Equality and Inequality Constraints

Once again, the best way to understand this implementation is through a detailed example, therefore consider the following polynomial optimisation problem. A slightly larger problem is intentionally chosen to provide a stronger intuition for how the problem is computationally set up.

$$\begin{aligned} f^* = \min_{\vec{x}\in\mathbb{R}^3} \quad & \vec{x}\cdot\vec{x} - \sum_{i=1}^{3} x_i \\ s.t. \quad & x_1 - x_2^2 = 0 \\ & x_2 x_3 = 0 \\ & 1 - x_2 - x_2^2 + x_3 = 0 \\ & 1 - 4x_3^2 \geq 0 \\ & 1 - x_1^2 - x_2^2 \geq 0 \\ & 1 - x_2^2 - x_3^2 \geq 0 \end{aligned} \tag{7}$$

From here, we introduce the monomial vector (chosen here with maximum degree 2) and corresponding slack variables. The mapping from the original monomials to the new will be:

$$\begin{bmatrix} x_2 \; x_3 \; x_1 \; x_2^2 \; x_2x_3 \; x_3^2 \; x_1x_2 \; x_1^2 \end{bmatrix} \rightarrow \begin{bmatrix} y_0 \; y_1 \; y_2 \; y_3 \; y_4 \; y_5 \; y_6 \; y_7 \end{bmatrix}$$

Note that the x_i monomials seem to be mapped with an unusual arrangement. The ordering of the monomials has been rearranged such that, when a Cholesky decomposition on the A matrix is performed in order to detect correlative sparsity (explained below), the resulting matrix will have the greatest sparsity, which is desirable for our current application [27]. Also note that the cross term x_1x_3 appears to be missing from the monomial array; the reason for this will also be covered shortly.

Rewriting the original constraints in terms of the slack variables y_i yields $y_2 - y_3 = 0$, $y_4 = 0$, $1 - y_0 - y_3 + y_1 = 0$, $1 - 4y_5 \geq 0$, $1 - y_7 - y_3 \geq 0$ and $1 - y_3 - y_5 \geq 0$. Following our established matrix storage convention, the three equality constraints will be stored in the first three rows of the A matrix, followed by another three rows capturing the inequality constraints. Using the same order in which the above equations are presented, the first six rows of the A matrix will be as follows. The vector of variables on the first row and dashed line separators are for visualisation purposes and of course will not appear in the actual computational storage.

$$A^T = \begin{bmatrix} y_0 & y_1 & y_2 & y_3 & y_4 & y_5 & y_6 & y_7 \\ - & - & - & - & - & - & - & - \\ 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ - & - & - & - & - & - & - & - \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \end{bmatrix}$$

Now, the moment matrix must be formed by introducing an appropriate vector of monomials, and the resulting positive semidefinite matrix constraint will form the third and final segment in the A matrix. Following the aforementioned procedure, we can define the following monomial vector of order 1:

$$\vec{v}_1(x_1, x_2, x_3) = \begin{bmatrix} 1 & x_1 & x_2 & x_3 \end{bmatrix}$$

and multiply by its own transpose to get:

$$\vec{v}_1 \cdot \vec{v}_1^T = M_1 = \begin{bmatrix} 1 & x_1 & x_2 & x_3 \\ x_1 & x_1^2 & x_1x_2 & x_1x_3 \\ x_2 & x_1x_2 & x_2^2 & x_2x_3 \\ x_3 & x_1x_3 & x_2x_3 & x_3^2 \end{bmatrix} = \begin{bmatrix} 1 & y_2 & y_0 & y_1 \\ y_2 & y_7 & y_6 & y_8 \\ y_0 & y_6 & y_3 & y_4 \\ y_1 & y_8 & y_4 & y_5 \end{bmatrix} \succeq 0$$

This 4 by 4 matrix could form the single positive semidefinite matrix constraint for the problem, as would certainly be a valid reformulation of the original problem. However, a closer look at the problem objective function and constraints reveals a further step which, if taken, can lead to a more compact and efficient formulation without any loss of generality.

2.3.3 Correlative Sparsity

Looking back at the example problem on the previous page, one can determine by inspection that the entire problem does not depend on the cross term x_1x_3. An algorithm for finding these terms programmatically is also not difficult to implement. One simply needs to check each monomial in the objective function and constraints, and track which terms are present. An efficient way of encoding this information is through a symmetric square matrix, where each nonzero entry at index (i, j) represents a codependency between variable i and j. Applied to the current example, the matrix of codependencies would take the form:

$$C = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

which encodes the information that all terms are present except for x_1x_3. For this case, we say that the problem has correlative sparsity with respect to x_1 and x_3 [30].

With this knowledge in hand, it is now possible to divide up this large single moment matrix into multiple, smaller components known as cliques. In the current example, it can be quickly determined by inspection that there are strong couplings only between x_1 and x_2, and between x_2 and x_3. Therefore, we divide the problem up into 2 cliques, defining two monomial vectors as seen below.

$$\vec{v}_1^1(x_1, x_2) = \begin{bmatrix} 1 & x_2 & x_1 \end{bmatrix} \qquad \vec{v}_1^2(x_2, x_3) = \begin{bmatrix} 1 & x_3 & x_2 \end{bmatrix}$$

Note that for these expressions, the superscript simply represents the index of the respective monomial vector, while the subscript states the order of the monomial vector.

Multiplying each monomial vector by its own transpose creates the following two symmetric positive semidefinite moment matrices.

$$M_1^1(x_1,x_2) = \begin{bmatrix} 1 & x_2 & x_1 \\ x_2 & x_2^2 & x_1x_2 \\ x_1 & x_1x_2 & x_1^2 \end{bmatrix} \succeq 0 \quad M_1^2(x_2,x_3) = \begin{bmatrix} 1 & x_3 & x_2 \\ x_3 & x_3^2 & x_2x_3 \\ x_2 & x_2x_3 & x_2^2 \end{bmatrix} \succeq 0$$

Computationally, the extraction of cliques from a matrix of codependencies can be performed with a Cholesky decomposition [9], and there are numerous packages available across many programming languages which perform this step efficiently. For Python implementations, the reader is encouraged to explore the scikit-sparse (sksparse) module [19].

When converted into the corresponding slack variables, the moment matrices now become

$$M_1^1(y) = \begin{bmatrix} 1 & y_0 & y_2 \\ y_0 & y_3 & y_6 \\ y_2 & y_6 & y_7 \end{bmatrix} \succeq 0 \quad M_1^2(y) = \begin{bmatrix} 1 & y_1 & y_0 \\ y_1 & y_5 & y_4 \\ y_0 & y_4 & y_3 \end{bmatrix} \succeq 0$$

and these will be the final PSD constraints which will be encoded and stored into the A matrix. The removal of unnecessary independent variables in this way can significantly improve the speed at which the problem can be solved, especially as the problem size increases.

2.3.4 Storing Positive Semidefinite Constraints

While an equality or inequality constraint can be stored easily in a single line, a PSD matrix clearly cannot. The solution here is to vectorise the matrix constraint by stacking up its entries into a long vector, where the terms at each matrix index will be stored in a separate row [8]. As example, a 3 by 3 matrix will be stored across 9 rows; the first row will store the terms at index $(1,1)$, the next row stores the $(1,2)$ index terms, and so on. For this application, matrices will be vectorised in row-major format.

Applying this to the two positive semidefinite constraints above, one can verify that we will arrive at the following matrix encodings.

$$A_{PSD_1}^T = \begin{bmatrix} \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix} \quad A_{PSD_2}^T = \begin{bmatrix} \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

Finally, stacking up all the above determined rows recovers the full A matrix seen below. The corresponding $\vec{b}$ and $\vec{c}$ vectors are also given, and for practice the reader is encouraged to follow and verify the encodings of these variables.

$$A^T = \begin{bmatrix} 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad c = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad b = \begin{bmatrix} -1 \\ -1 \\ -1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

Wrapping up the formulation, we populate a `K` object, as described above, with the following attributes:

- `K.f = 3`; we have 3 equality constraints
- `K.l = 3`; we have 3 inequality constraints
- `K.s = [3, 3]`; we have 2 positive semidefinite constraints, each with size 3 by 3 (ordered)

The conic space of this problem can be written as $\mathcal{K} = \{0\}^3 \times \mathbb{R}^3_+ \times \mathbb{S}^{3\times3}_+ \times \mathbb{S}^{3\times3}_+$, the product of a zero cone with length 3 (equalities), a non-negative orthant cone of length 3 (inequalities), and two PSD cones with size 3 by 3. Note that the resulting A matrix is particularly sparse; the formulation of the moment matrices causes this to be so.

2.3.5 Available Implementation Package in MATLAB

A MATLAB package which implements this convex optimisation reformulation is available at the following link [10]. The package, forked from the well-known YALMIP library, contains numerous algorithms and solvers for convex optimisation. To set up the convex formulation, one can define a symbolic objective function, a set of equality constraints, and another set of inequality constraints, then call the `compileparsemoment` function to execute the algorithm and retrieve the `A`, `b`, `c` and `K` variables as outputs.

3 Convex Optimisation Algorithms

With the convex optimisation problem formulation complete, discussion about the alternating direction method of multipliers algorithm can now follow. This section will first introduce two families of predecessors to the ADMM algorithm which motivated the development of ADMM, and will proceed to a detailed overview of ADMM, its advantages and its implementations.

3.1 Predecessors to ADMM

3.1.1 Dual Ascent

Consider the simple convex constrained optimisation problem:

$$\begin{aligned} &\min_{x \in \mathbb{R}^n} f(x) \\ &s.t. \quad Ax = b \end{aligned} \tag{8}$$

where $A \in \mathbb{R}^{m \times n}$ and $f : \mathbb{R}^n \to \mathbb{R}$. The Lagrangian for the problem is given by:

$$\mathcal{L}(x, y) = f(x) + y^T(Ax - b) \tag{9}$$

and the dual function by:

$$g(y) = \inf_x \mathcal{L}(x, y) = -f^*(-A^T y) - b^T y \tag{10}$$

where y is the Lagrange multiplier or dual variable, and f^* is the convex conjugate of f [4]. The convex conjugate is defined as the largest gap between the linear function yx and the objective function $f(x)$ [18]. For convex problems, the dual function has the key property that, given that certain conditions hold, the optimal solution which maximises the dual function also minimises the original function. Hence, we define the dual problem:

$$\max_{y \in \mathbb{R}^m} \; g(y)$$

and we will be able to retrieve the optimal solution x^* for the primal function from the dual optimal point y^*:

$$x^* = \arg\min_x \mathcal{L}(x, y^*)$$

With dual ascent, the problem is solved using the gradient ascent method, increasing the value of the dual function at each iteration until a dual maximum is achieved [4]. At each iteration, the x and y vectors are updated consecutively with the following equations:

$$x^{k+1} = \arg\min_x \mathcal{L}(x, y^*) \tag{11}$$

$$y^{k+1} = y^k + \alpha^k(Ax^{k+1} - b) \tag{12}$$

where α^k is a chosen step size. The x minimisation step and update is performed by noticing that the gradient of the Lagrangian is simply $\nabla g(y) = Ax^+ - b$. The y vector of Lagrange multipliers can be interpreted as a price vector which penalises each value for its deviation from the desired optimal value, and thus the second step is sometimes called a price update step.

The dual ascent method has the great advantage that it can be extended to a decentralised problem and algorithm, meaning very large problems can potentially be decomposed and solved in parallel using dual ascent [4]. Let's consider a large problem with a function that is separable, either into entirely independent variables, or subvectors of the global independent vector, i.e.

$$f(\vec{x}) = \sum_{i=1}^{N} f_i(\vec{x}_i) \qquad \vec{x} = [\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_N] \tag{13}$$

We can then partition the A matrix with $A = [A_1, A_2, \ldots, A_N]$ such that $A\vec{x} = \sum_{i=1}^{N} A_i\vec{x}_i$. The Lagrangian for the decentralised algorithm is then given by:

$$\mathcal{L}(x, y) = \sum_{i=1}^{N} \mathcal{L}_i(\vec{x}_i, y) = \sum_{i=1}^{N} f_i(\vec{x}_i) + y^T A_i \vec{x}_i - (\frac{1}{N})y^T b \tag{14}$$

Crucially, this Lagrangian is also separable in $\vec{x}$, and thus the x-minimisation step can be divided into N components and solved in parallel. Once the solution from each of the components are gathered, the dual variable update can occur as per usual. This decentralised approach to the dual ascent method is known as dual decomposition.

The greatest drawback to the dual ascent method is its lack of robustness to a wide range of applications. The algorithm is reliant on the condition that the primal function converges to an optimal solution as the dual function converges to another optimal dual point. However, the conditions that allow this to be true do not hold for many applications, and for those cases the dual ascent algorithm is simply infeasible.

3.1.2 Method of Multipliers with Augmented Lagrangians

An alternative algorithm for convex optimisation is the method of multipliers, which aims to improve on the robustness of the dual ascent method by introducing the augmented Lagrangian [4]. For the same optimisation problem statement above, the augmented Lagrangian is given by:

$$\mathcal{L}_\rho(x, y) = f(x) + y^T(Ax - b) + (\rho/2)\,\|Ax - b\|_2^2 \tag{15}$$

where the newly introduced variable $\rho > 0$ is known as the penalty parameter. The augmented Lagrangian can simply be considered as the non-augmented Lagrangian with the associated optimisation problem:

$$\begin{aligned} &\min_{x\in\mathbb{R}^n} f(x) + (\rho/2)\,\|Ax - b\|_2^2 \\ &s.t. \quad Ax = b \end{aligned} \tag{16}$$

and is equivalent to the original problem since the equality constraint, when satisfied, forces the second term to collapse to zero. Similarly, the dual function for this problem is $g_\rho(y) = \inf_x \mathcal{L}_\rho(x, y)$.

The iterative algorithm is very similar, and uses gradient ascent on the modified problem as seen below.

$$x^{k+1} = \arg\min_x \mathcal{L}_\rho(x, y^k) \tag{17}$$

$$y^{k+1} = y^k + \rho\,(Ax^{k+1} - b) \tag{18}$$

Note that in this algorithm, the penalty parameter has been chosen as the step size. It can be shown that this choice results in the problem being dual feasible [4]; the primal and dual functions will converge simultaneously to their respective minimum and maximum. This greatly increases the convergence properties of the method and thus yields a more robust algorithm.

Unfortunately, the introduction of the penalty parameter means the problem can no longer be decomposed into a decentralised form. The algorithm may be adept at handling smaller problem statements, but its inability to scale up to arbitrarily large problem sizes will ultimately limit its potential on modern world applications.

3.2 Alternating Direction Method of Multipliers

3.2.1 Algorithm Overview

The development of the alternating direction method of multipliers was motivated by the desire to harness both advantages of the dual ascent and method of multipliers methods, creating an algorithm that had both good convergence properties and the ability to be decomposed and solved in parallel [4]. The problem can take many slightly different forms; for the purpose of this introduction consider a problem of the form:

$$\begin{aligned} &\min_{x\in\mathbb{R}^n,\, z\in\mathbb{R}^m} f(x) + g(z) \\ &s.t. \quad Ax + Bz = c \end{aligned} \tag{19}$$

where $c \in \mathbb{R}^p$, $A \in \mathbb{R}^{p\times n}$ and $B \in \mathbb{R}^{p\times m}$. The clear single difference between this formulation and the previous one above is the splitting of the objective function into two independent functions. The augmented Lagrangian for this problem takes the following form:

$$\mathcal{L}_\rho(x, z, y) = f(x) + g(z) + y^T(Ax + Bz - c) + (\rho/2)\,\|Ax + Bz - c\|_2^2 \tag{20}$$

Each iteration in the algorithm consists of the following steps:

$$x^{(k+1)} := \arg\min_x \mathcal{L}_\rho(x, z^{(k)}, y^{(k)}) \tag{21}$$

$$z^{(k+1)} := \arg\min_z \mathcal{L}_\rho(x^{(k+1)}, z, y^{(k)}) \tag{22}$$

$$y^{(k+1)} := y^{(k)} + \rho\left(Ax^{(k+1)} + Bz^{(k+1)} - c\right) \tag{23}$$

where $\rho > 0$ is a selected penalty parameter. The method of multipliers would update the x and z vectors in a single step; $x^{(k+1)}, z^{(k+1)} := \arg\min_{x,z} \mathcal{L}_\rho(x, z, y^{(k)})$, however ADMM updates the variables one after the other in alternating fashion, hence its name. The separation of the objective function is what allows the problem to be decomposed when f or g are separable [4].

3.2.2 ADMM Convergence Properties

In is worth briefly discussing the convergence characteristics of the ADMM algorithm. For simplicity, we make the two conservative assumptions about the problem. Firstly, the functions f and g are assumed to be closed, proper and convex. This assumption ensures that an optimum x and z exist that minimise the augmented Lagrangian. The second is that the unaugmented Lagrangian of the problem has at least one saddle point. Since the algorithm attempts to simultaneously minimise x and maximise y, an optimiser for these 2 objectives will rest at a saddle point. Given these two assumptions, it is sufficient to claim that the ADMM iterations will have the following properties [4]:

1. Residual convergence; the residuals will converge to 0 as the iterations k increase towards infinity.
2. Objective convergence; the objective function will approach its optimal value: $f(x^{(k)}) + g(z^{(k)}) \to p^*$ as $k \to \infty$.
3. Dual variable convergence; The dual variable will converge to its dual optimum: $y^{(k)} \to y^*$ as $k \to \infty$.

In practice, the ADMM algorithm can converge to moderate accuracy very quickly, within 100 iterations. However, beyond moderate accuracy the algorithm is very slow to converge to very high accuracy, easily requiring several thousand iterations for slightly larger problems. Therefore, the ADMM algorithm may not be suitable for applications requiring such accuracy.

3.2.3 Existing Implementations and Previous Research

Research in the alternating direction method of multipliers for solving convex optimisation problems has gained popularity in the recent decade. Before introducing the sparse ADMM implementation used for this project, it would be appropriate to very briefly summarise a few notable pieces of previous work and research. The first standard form implementation of the ADMM algorithm for semidefinite programming problems (convex problems) was proposed by Zaiwen Wen in 2010 [31], and the results of their implementation show the algorithm to be strongly robust and efficient. A very similar implementation of their proposed algorithm was coded in this project and used as a benchmark comparison to our sparse ADMM implementation.

In similar spirit to that of this project, research has also extended to consider different types of ADMM implementations and convex problems with different specific structures. In 2012, a previously-known special type of ADMM algorithm known as the Douglas–Rachford ADMM [14] was extended in its robustness through the introduction of a Gaussian back-substitution procedure [16]. Later, in 2013, two highly efficient Gauss-Seidel type ADMM algorithms for convex optimisation were developed for special functions with smooth Lipschitz continuous gradients [15]. In 2016, the convergence behaviour of multi-block (more than two blocks; standard ADMM algorithms decompose problems into two blocks) convex problems was analysed, and it was concluded that such an extension of the ADMM algorithm is not necessarily convergent [5].

There are certainly more works on ADMM and convex optimisation beyond the aforementioned pieces. The possible areas for further research in the study of these problems and algorithms are practically endless, and this is without a doubt an exciting opportunity to provide yet another one of such contributions.

3.3 ADMM for Sparse Polynomial Optimisation Problems

Convex optimisation problems can arise from a wide range of different origins, and thus numerous different solvers have been created and optimised to tackle specific applications. When a convex optimisation problem is formulated from the relaxation of a polynomial optimisation problem, the nature of the formulation causes the resulting problem to take a particularly sparse structure.

Given this sparsity, it may be possible to decompose the convex problem into smaller components, and formulate a scalable, parallel algorithm that can be extended to problems of arbitrary size. The formulation of a performant solver of this nature is the primary focus of this project, and the following section provides a detailed explanation of the theory and implementation of this algorithm.

3.3.1 Problem Decomposition and Clique Detection

In similar fashion to that of detecting correlative sparsity described previously, the first step to the sparse formulation is to decompose the overall convex problem into smaller cliques. Once again, the clique detection logic is best understood by following a practical example. Consider the following convex optimisation problem:

$$\begin{aligned} \min_{y\in\mathbb{R}^m} \quad & -b^T y \\ s.t. \quad & c - A^T y \in \mathcal{K} \end{aligned} \tag{24}$$

and let a small example problem with only 3 independent variables be given by:

$$b = \begin{bmatrix} 0 \\ -1 \\ 1 \end{bmatrix} \qquad c = \begin{bmatrix} 2 \\ 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 2 \end{bmatrix} \qquad A^T = \begin{bmatrix} -1 & -2 & 0 \\ 0 & 2 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -3 \end{bmatrix}$$

with the cone $\mathcal{K} = \{0\}^1 \times \mathbb{R}^2_+ \times \mathbb{S}^{2\times 2}_+$. The reader can confirm without too much difficulty that the above variables encode the problem:

$$\begin{aligned} f^* = \min_{\vec{y}\in\mathbb{R}^3} \ & y_2 - y_3 \\ s.t. \quad & 2 + y_1 + 2y_2 = 0 \\ & y_3 - 2y_2 \geq 0 \\ & 1 - y_2 \geq 0 \\ & \begin{bmatrix} y_3 & 1-y_2 \\ 1-y_2 & 2+3y_3 \end{bmatrix} \succeq 0 \end{aligned} \tag{25}$$

By inspection, it can be observed that constraints 1 and 3 only depend on the variables x_1 and x_2, while constraints 2 and 4 only depend on x_2 and x_3. The problem is said to be sparse with respect to y with the following index sets: [11]

$$\begin{aligned} I_1 &= \{1, 2\} \\ I_2 &= \{2, 3\} \end{aligned} \tag{26}$$

In other words, we can identify and gather all the constraints which include x_1 and/or x_2 only into one clique, and gather the constraints which include x_2 and/or x_3 only into another clique. In this example, this means forming one clique with the first and third constraints, and another with the second and fourth constraints. This decomposition results in the following set of cliques:

$$\begin{bmatrix} 2 \\ 1 \end{bmatrix} - \begin{bmatrix} -1 & -2 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \in \{0\}^1 \times \mathbb{R}^1_+ \tag{27}$$

$$\begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 2 \end{bmatrix} - \begin{bmatrix} 2 & -1 \\ 1 & 0 \\ 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 0 & -3 \end{bmatrix} \begin{bmatrix} y_2 \\ y_3 \end{bmatrix} \in \mathbb{R}^2_+ \times \mathbb{S}^{2\times 2}_+ \tag{28}$$

Computationally, a matrix of codependencies can be formed in similar fashion to the detection of correlative sparsity in the polynomial optimisation problem, except in this application one must be careful to recognise the multi-line storage of the PSD constraints and collapse these constraints into a single row for codependency checks. From the codependency matrix, the maximal cliques can be determined. This is a well-known problem in graph theory, and there are various algorithms and implementations available for this. In this project, the Python library NetworkX was used [7], calling the `find_cliques` function on a NetworkX graph. This library uses an iterative version of the Bron-Kerbosch algorithm [29], and a recursive implementation is also available. It is worth noting that this algorithm is computationally very expensive, running in approximately $O(3^{n/3})$ time, where n is the number of vertices in the graph. Slightly more efficient algorithms of this nature do exist. For example, Tarjan and Trojanowski's algorithm [26] developed in 1977 runs in $O(3^{n/3})$ time, and the most efficient of such similar algorithms was developed by Robson in 1985 [24], and runs in $O(1.2108^n)$ time. Nevertheless, this exponential time complexity may have implications on this method's limitations for very large applications.

This clique detection process can be improved to at least polynomial time by taking advantage of the fact that the graph generated by the codependency matrix is a chordal graph. An arbitrary, potentially non-chordal graph can have exponentially many cliques scaling with its size, while the maximum number of possible cliques in a chordal graph increases only linearly. NetworkX provides a function with this implementation, `chordal_graph_cliques` [7], however no further details about the implemented algorithm are provided within the documentation. This function was tested in code, and unfortunately it was found that it performed significantly worse than the generic `find_cliques` function, and therefore was not implemented. There exists an even better algorithm for this purpose, which will be discussed in the final section. Considering the expensive nature of this pre-processing step, this is certainly a point of additional investigation in future works.

3.3.2 ADMM Problem Structure and Algorithm for Sparse Problems

3.3.2.1 Problem Reformulation for Sparse Applications

With the problem now decomposed into cliques, the new convex optimisation problem now takes the form:

$$\begin{aligned} f^* = \min_{y\in\mathbb{R}^m, s_i\in\mathbb{R}^{|I_i|}} \quad & -b^T y \\ s.t. \quad & c_i - A_i^T s_i \in \mathcal{K}_i \\ & s_i = P_{I_i} y \end{aligned} \tag{29}$$

where c_i, A_i^T and s_i are the local c, A^T and y variables respectively for each clique [11]. The variables P_{I_i} are simply a matrices of ones and zeroes which multiplies with the vector y to extract the desired components of the independent variable. With this reformulation, the cost function must also be revised to include the s_i terms, by equally decomposing the objective function and distributing its components across the cliques such that:

$$b^T y = \sum_{i=1}^{k} b_i^T s_i \tag{30}$$

We can choose to have the above expression represent the entire cost function, but it may be useful to keep some component of the global y vector, and alternatively use a weighted average of the two expressions. Depending on the application, λ can be adjusted to weight more heavily towards the global or local independent vectors, however as a starting point it is logical to simply select $\lambda = 0.5$. We can thus express the objective function in the form:

$$b^T y = \lambda b^T y + (1-\lambda) b^T y = \lambda b^T y + (1-\lambda)\sum_{i=1}^{k} b_i^T s_i \tag{31}$$

The problem statement has once again transformed, and after one more small modification we arrive at the following set of expressions for the sparse convex optimisation problem.

$$
\begin{aligned}
f^* = \min_{y\in\mathbb{R}^m,\ s_i\in\mathbb{R}^{|I_i|},\ z_i\in\mathbb{R}^{|J_i|}} \quad & -\lambda b^T y - (1-\lambda)\sum_{i=1}^{k} b_i^T s_i + \sum_{i=1}^{k}\delta_{\mathcal{K}_i}(z_i) \\
s.t. \quad & c_i - A_i^T s_i = z_i \\
& s_i = P_{I_i} y
\end{aligned} \tag{32}
$$

In order for the formulation to be programmatically feasible, we define an indicator function $\delta_{\mathcal{K}_i}(z_i)$ to enforce the conic constraint on the variable z_i. The indicator function is defined such that $\delta_{\mathcal{K}_i} = 0$ if $z_i \in \mathcal{K}_i$ and infinity otherwise [11].

3.3.2.2 The Augmented Lagrangian

Next, the augmented Lagrangian can be defined. With 2 sets of equality constraints and 2 sets of independent variables, we will in turn introduce 2 sets of Lagrange multipliers η_i, ζ_i, and 2 sets of penalty parameters σ_i, ρ_i. The augmented Lagrangian is given as the following:

$$
\begin{aligned}
\mathcal{L} := -\lambda b^T y - (1-\lambda)\sum_{i=1}^{k} b_i^T s_i + \sum_{i=1}^{k}\delta_{\mathcal{K}_i}(z_i) \\
+ \sum_{i=1}^{k}\eta_i^T(c_i - A_i^T s_i - z_i) + \sum_{i=1}^{k}\zeta_i\cdot(s_i - P_{I_i}y) \\
+ \sum_{i=1}^{k}\frac{\sigma_i}{2}\left\|c_i - A_i^T s_i - z_i\right\|^2 + \sum_{i=1}^{k}\frac{\rho_i}{2}\left\|s_i - P_{I_i}y\right\|^2
\end{aligned} \tag{33}
$$

As the augmented Lagrangian is quite cumbersome, it may be useful to break down the expression and remind the reader of each term's origin and purpose. The first three terms are simply an identical copy of the objective function, which by definition is included in all Lagrangian expressions. The $\delta_{\mathcal{K}_i}(z_i)$ function acts as a means of forcing z_i to belong to a conic set; the ADMM algorithm itself will ensure this is the case. There is one ζ_i and η_i vector for every clique in the problem, and these vectors can be of potentially different length (determined by the size of the clique). They are the Lagrange multipliers which represent the deviation of the current iteration from the equality constraints. For simplicity, they will be initialised to vectors of ones and are updated at each time step. The penalty parameters σ_i and ρ_i are scalars which weight the 'punishment' of deviating away from the equality constraints. Larger penalty parameters place implies a higher priority for the algorithm to satisfy the equality constraints, rather than minimising the objective function. They can also be unique to each clique, and possibly even updated at each time step depending on convergence characteristics. In this paper's implementation, both penalty parameters were kept at a constant value throughout all iterations.

3.3.2.3 The Sparse ADMM Algorithm

Finally, a description of the sparse parallelisable ADMM implementation can commence. In the same spirit of ADMM algorithms, the iterative process will follow three steps: [11]

1. Minimising $\mathcal{L}$ with respect to y, z_i
2. Minimising $\mathcal{L}$ with respect to s_i
3. Updating the Lagrange multipliers η_i and ζ_i

Of course, it will be necessary to attend to each of these steps with more detail. Note that for the following equations, superscripts will denote the time iteration for the variable of concern.

Minimising $\mathcal{L}$ with respect to y, z_i

Firstly, consider the minimisation of the Lagrangian with respect to the global independent vector y. This is achieved simply by setting the derivative of the Lagrangian with respect to y to 0: $\frac{\partial \mathcal{L}}{\partial y} = 0$. With a few steps of vector calculus and matrix manipulation, we arrive at the following expression for this minimisation step:

$$y^{(j+1)} = \left(\sum_{i=1}^{k} \rho_i P_{I_i}^T P_{I_i} \right)^{-1} \cdot \left(\lambda b + \sum_{i=1}^{k} P_{I_i}^T (\zeta_i^{(j)} + \rho_i s_i^{(j)}) \right) \tag{34}$$

One should not be alarmed by the presence of a matrix inverse in the first term. Upon closer inspection, one notices that since the rows in all P_{I_i} matrices are unit vectors, the first product in the expression produces a diagonal matrix, and its inverse is computationally trivial to calculate. Moreover, since the P_{I_i} matrices are simply a static parameter of the problem, if one chooses to keep ρ_i constant between all iterations, the entire first term can be pre-computed before the start of the iterative algorithm and retrieve when needed.

Next, we consider the minimisation with respect to z_i. Mathematically, this can be expressed by:

$$\begin{aligned} z_i^{(j+1)} &= \arg\min_{z_i} \mathcal{L} \\ &= \arg\min_{z_i} \left[\delta_{\mathcal{K}_i}(z_i) + \frac{\sigma_i}{2} \left\| c_i - A_i^T s_i^{(j)} - z_i + \frac{1}{\sigma_i} \eta_i^{(j)} \right\|^2 \right] \\ &= \arg\min_{z_i \in \mathcal{K}_i} \left\| \left(c_i - A_i^T s_i^{(j)} + \frac{1}{\sigma_i} \eta_i^{(j)} \right) - z_i \right\|^2 \end{aligned} \tag{35}$$

The above mathematical expression can be interpreted verbally as: find vectors z_i that belong in the cones $\mathcal{K}_i$ such that the vectors are as close as possible to $c_i - A_i^T s_i^{(j)} + \frac{1}{\sigma_i} \eta_i^{(j)}$. Mathematically, this is known as the projection of the vector $c_i - A_i^T s_i^{(j)} + \frac{1}{\sigma_i} \eta_i^{(j)}$ onto the cone $\mathcal{K}_i$.

For our problem statement, we have three different types of cones: the zero cone, the non-negative orthant cone, and the positive semidefinite cone. The projections for each conic type are performed with the following:

- Zero Cone: Set all elements in the vector to zero.
- Non-Negative Orthant Cone: Set all negative elements in the vector to zero.
- Positive Semidefinite Cone: This projection is the most computationally expensive. First, the eigenvectors and eigenvalues of the matrix are found; let U be the eigenvector matrix and D the eigenvalue array. The variables are then filtered for only the positive eigenvalues and corresponding eigenvectors; let's denote them as U_+ and D_+. Finally, the projected matrix is found as $S = U_+ D_+ U_+^T$. Note that, computationally, the PSD matrix will need to be converted into a matrix for the eigenvalue problem, and re-vectorised after its projection.

Lastly, it is useful to note that each of these projections are independent of one another and thus can be performed in parallel.

Minimising $\mathcal{L}$ with respect to s_i

This step follows a similar logic to the previous: differentiating the Lagrangian with respect to s_i and setting the result equal to 0: $\frac{\partial \mathcal{L}}{\partial s_i} = 0$. This will produce a linear system of equations given by:

$$\Big(\rho_i I + \sigma_i A_i A_i^T\Big) s_i^{(j+1)} = \rho_i P_{I_i} y^{(j+1)} + \sigma_i A_i \Big(c_i - z_i^{(j+1)} + \frac{1}{\sigma_i}\eta_i^{(j)}\Big) - \zeta_i^{(j)} + (1-\lambda) b_i \quad (36)$$

Any appropriate factorisation method can be considered for solving this linear system. On the other hand, attempting to invert the matrix is computationally very expensive and is not recommended. In this project's implementation, a Cholesky decomposition from the Python `scikit-sparse` library was used. If one chooses a constant ρ_i and σ_i, the matrix factorisation will only need to be performed once and can be re-used at each time step. This step, for non-constant ρ_i and σ_i, is expected to demand the greatest computational resources.

The reader should be cautious and note that the y and z_i variables are the ones from iteration $j+1$, updated from the previous step. The ordering of the two steps is essential, and thus they cannot be performed simultaneously. On the other hand, note that one linear system must be solved for each clique in the problem, and that those systems can be solved in parallel.

Updating the Lagrange multipliers η_i and ζ_i

This step is straightforwards and computationally inexpensive. The multipliers are updated with the following equations.

$$\eta_i^{(j+1)} = \eta_i^{(j)} + \sigma_i\Big(c_i - A_i^T s_i^{(j+1)} - z_i^{(j+1)}\Big) \quad (37)$$

$$\zeta_i^{(j+1)} = \zeta_i^{(j)} + \rho_i\Big(s_i^{(j+1)} - P_{I_i} y^{(j+1)}\Big) \quad (38)$$

These equations can be updated in parallel, however since the step is inherently cheap, this shouldn't be a priority. This concludes the sparse ADMM algorithm.

3.3.3 Residuals and Stopping Criterion

The ADMM algorithm described above will always have some form of asymptotic convergence properties, i.e. the iterations will approach, but never reach, the global optimiser. Therefore it is important to formulate a measure of how 'close' the algorithm is from the optimal solution, and discuss a suitable stopping criterion for when the iterations should terminate.

The above ADMM algorithm produces multiple residuals at each time step. Specifically, there are two sets of primal residuals and three sets of dual residuals that measure the algorithm's proximity to convergence.

Primal Residuals

The primal residuals are determined by setting the derivative of the augmented Lagrangian with respect to the dual variables η_i and ζ_i to zero. Mathematically, these quantities show how fast the Lagrange multipliers are changing between iterations, which intuitively reveal some information about how close the algorithm is to convergence. With some elementary vector calculus, we arrive at the following expressions for the primal residuals:

$$r_{primal_1} = \frac{\partial \mathcal{L}}{\partial \zeta_i} = s_i - P_{I_i} y = 0 \tag{39}$$

$$r_{primal_2} = \frac{\partial \mathcal{L}}{\partial \eta_i} = c_i - A_i^T s_i - z_i = 0 \tag{40}$$

Note that there will be 1 of each primal residual for every clique in the problem, for a total of $2n$ primal residuals for an n-clique problem. It makes sense to use take maximum residual of all cliques to represent the overall primal residual for the iteration. Computationally, it is logical to update these residuals either just before or just after the Lagrange multiplier update step, whichever is more convenient.

Dual Residuals

In the same spirit, the dual residuals are found by setting the derivative of the augmented Lagrangian with respect to the primal variables y, z_i and s_i to zero. These will show how fast the independent variables are changing, as well as how 'far' each conic projection is from its initial value. Conveniently, $\frac{\partial \mathcal{L}}{\partial s_i} = 0$ always for this application, since the enforcement of this equality takes place at the second step of the ADMM algorithm, and the final Lagrange multiplier update step does not affect this equality. Therefore, we are reduced to only 2 sets of dual residuals. The derivation for these residual expressions is slightly more involved, but after some algebraic manipulation and evaluating the derivatives at iteration $j+1$, the first dual residual is given by the following:

$$\begin{aligned} r_{dual_1} = \frac{\partial \mathcal{L}}{\partial y} &= -\lambda b - \sum_{i=1}^{k} P_{I_i}^T \zeta_i^{(j)} - \sum_{i=1}^{k} \rho_i P_{I_i}^T (s_i^{(j)} - P_{I_i} y^{(j+1)}) = 0 \\ \Rightarrow r_{dual_1} &= -\lambda b - \sum_{i=1}^{k} P_{I_i}^T \zeta_i^{(j+1)} = \sum_{i=1}^{k} \rho_i P_{I_i}^T (s_i^{(j)} - s_i^{(j+1)}) \end{aligned} \tag{41}$$

Importantly, both the left and right expressions are equivalent, and thus either can be used to calculate the dual residual. The choice of which expression to use is left to the programmer. For this project, the right expression was implemented; the expressions were updated separately for each clique just after the s_i variable update, and summed at a later step to calculate the total residual. Note that since this is a derivative with respect to the single global vector y, there will only be one dual residual of this type at each iteration.

For the second dual residual, again evaluating at iteration $j+1$, we get:

$$\begin{aligned} r_{dual_2} = \frac{\partial \mathcal{L}}{\partial z_i} &= \frac{\partial}{\partial z_i}(\delta(z_i)) - \eta_i^{(j)} - \sigma_i (c_i - A_i^T s_i^{(j)} - z_i^{(j+1)}) = 0 \\ \Rightarrow r_{dual_2} &= \frac{\partial}{\partial z_i}(\delta(z_i)) - \eta_i^{(j+1)} = \sigma_i A_i^T (s_i^{(j+1)} - s_i^{(j)}) \end{aligned} \tag{42}$$

Once again, both expressions are equivalent and valid representations of the residual. However, in this case we have an unusual $\frac{\partial}{\partial z_i}(\delta(z_i))$ term on the left-hand side, which we are unable to compute. The right-hand expression will therefore be the sole choice for this dual residual. It is recommended that this residual is calculated just after the s_i variables update, where one will have access to the s_i vectors at both the current and previous iteration. Similar to the primal residuals, there will be 1 of these dual residuals for every clique in the problem.

Stopping Criterion

With the primal and dual residuals defined, one has essentially free choice over the stopping criterion; selecting any desired ϵ for which the algorithm will terminate once both residuals fall below ϵ. Nevertheless, it may be useful to select a different ϵ for the primal and dual residuals, and determine some suitable ratio $\epsilon^{primal} : \epsilon^{dual}$ based on the properties of the problem. This will ensure that both residuals reach their respective stopping criterion at approximately the same time [4]. It is recommended that the choice of residuals follow these equations below.

$$\epsilon^{primal} = \max\left(\left[\sqrt{n}\,\epsilon^{abs} + \epsilon^{rel} \cdot \max\left(c_i, A_i^T s_i^{(j)}, z_i^{(j)}\right)\right], \left[\sqrt{m}\,\epsilon^{abs} + \epsilon^{rel} \cdot \max\left(s_i, P_{I_i} y^{(j)}\right)\right]\right) \tag{43}$$

$$\epsilon^{dual} = \max\left(\left[\sqrt{p}\,\epsilon^{abs} + \epsilon^{rel} \cdot P_i^T s_i^{(j)}\right], \left[\sqrt{n}\,\epsilon^{abs} + \epsilon^{rel} \cdot A_i^T s_i^{(j)}\right]\right) \tag{44}$$

where n is the length of vector c_i, m is the length of each respective s_i vector and p is the length of b. Intuitively, it makes sense to scale each ϵ by the length of the vectors it measures. As a starting point, it is recommended that we set $\epsilon^{rel} = 10^{-3}$ or 10^{-4}, and ϵ^{abs} can be tuned to the user's application needs.

4 Solver Implementation

The sections prior discuss existing theory and motivation for this project; the following will implement the established theory in a prototype sparse ADMM solver, and the properties and performance of the solver will be discussed in detail. The sparse solver will be compared to a benchmark solver which will be run on the same convex optimisation problems using a dense ADMM algorithm that does not implement any form of sparse decomposition.

4.1 Benchmark ADMM Solver

In order to create a means of comparison, a benchmark solver which implements the ADMM algorithm without clique splitting was programmed. Extra care was taken to ensure that similar steps within both algorithms were implemented with exactly the same code. This will ensure that any difference observed in the characteristics and performance of each solver will reflect the effects of the sparse decomposition only. In similar spirit to a previous implementation of the ADMM solver for convex problems [15], the benchmark solver will solve a convex optimisation problem in the form:

$$\begin{aligned} f^* = \min_{y \in \mathbb{R}^m} \quad & -b^T y \\ s.t. \quad & c - A^T y = z \\ & z \in \mathcal{K} \end{aligned} \tag{45}$$

and will follow these very similar steps at each iteration:

1. Minimising $\mathcal{L}$ with respect to y
2. Minimising $\mathcal{L}$ with respect to z
3. Updating the Lagrange multipliers η_i

4.2 Available Code Repository

For this project, the prototype solvers were coded using Python. Code for both solvers is publicly available at the following GitHub repository: `https://github.com/Ellipsoul/ADMM_Solver`. Several folders in the repository root folder can be ignored; they are earlier experimental implementations that were later refactored into the final version. The sparse ADMM solver can be found in the directory named `refactored_splitting`, and the benchmark solver is located in the `no-splitting` directory. A large set of randomly generated convex optimisation problems of different sizes and relaxation orders are also available in the `popData` directory, and the full set of results produced in this project's solver runs is available in the `results` directory. For more information regarding the code implementation, the reader is encouraged to visit the `README.md` file accessible from the repository homepage.

4.3 Test Problem Sets and Performance Metrics

The implemented solvers were run on a subset of the sample problems available in the repository. Due to computational limitations, runs were limited to a maximum of 10000 iterations, and problems of reasonable sizes were chosen such that both solvers could complete their respective algorithms in approximately 15 minutes or less. Altogether, 57 sets of runs were completed; 42 with relaxation order $\omega = 1$, 13 with $\omega = 2$, and just 2 runs with $\omega = 3$. As mentioned previously, increasing the relaxation order increases the number of independent variables with polynomial speed, and it was found that above a relaxation order of 3, the problem size would grow beyond the available computational capacity for this project.

Additional code was wrapped around both solvers in order to gather data on important performance metrics, as well as general properties about each algorithm. Below is a short summary of all properties which were measured; these will be processed into several different performance metrics, which will be discussed and evaluated in detail in the following section.

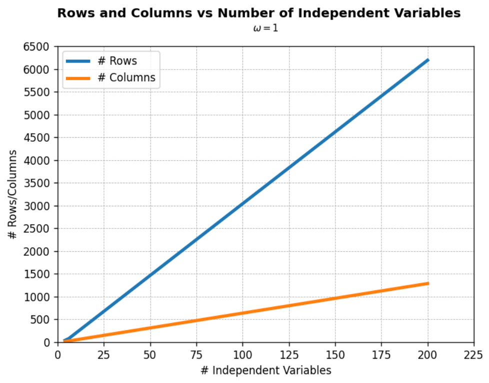
- The time taken for each step in both solvers, as well as the total time for each run. For the sparse ADMM implementation, the time taken to identify the cliques was also included. A breakdown of this metric would give insight into which aspects of the ADMM algorithms demand the highest computational resources, and would help direct further optimisation work. This would also give insight into how the required computational resources scale with the convex problem size.
- The objective cost, primal residuals and dual residuals per iteration. These would allow one to evaluate and compare the speed and behaviour of convergence for both solvers.

To ensure that both solvers were coded correctly and indeed produced the correct solution for each convex optimisation problem, each test problem was run with the well-known Mosek optimisation library [1] in MatLab, and each solution was verified and confirmed to coincide with that of the Mosek solver.

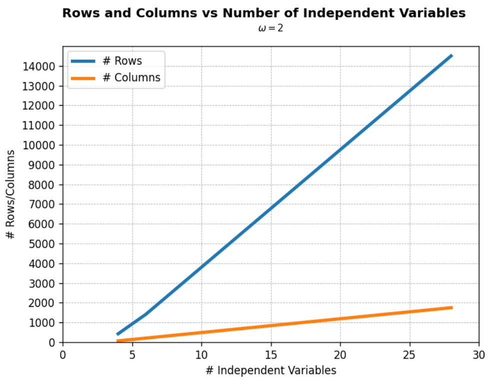
4.4 Results and Discussion

4.4.1 Polynomial Relaxation and Clique Detection Characteristics

Before diving directly into the solver performance metrics, it may be useful to consider a few properties of the problem's formulation. Specifically, it may be interesting to understand how the size of the reformulated convex optimisation problem scales with the number of independent variables in the original polynomial optimisation problem. Figure 2 below shows the relation between the number of rows and columns in the resulting A matrix as a function of the number of independent variables in the polynomial optimisation problem, for a relaxation order of 1 and 2.



(a) $\omega = 1$



(b) $\omega = 2$

Figure 2: Number of Rows and Columns vs Number of Independent Variables

A linear trendline for both the number of rows and columns implies a quadratic scaling in problem size with number of independent variables. This behaviour is to be expected for all ω due to the nature of the moment matrix formulation; introducing another independent variable quadratically increases the number of possible monomials that can be formed.

A quadratically growing problem size may be acceptable in terms of its potential to scale to larger problems, however, notice that the slopes of each line for $\omega = 2$ are significantly greater than those of $\omega = 1$. This trend indeed continues for higher ω, and problem sizes beyond $\omega = 3$ quickly become impractically large to solve with the CPU of a single computer. For example, for $\omega = 4$, a problem with just 10 independent variables will be relaxed into a convex problem with more than 250000 rows!

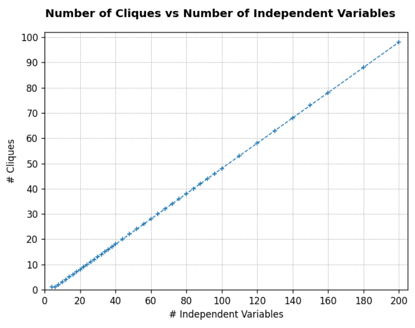
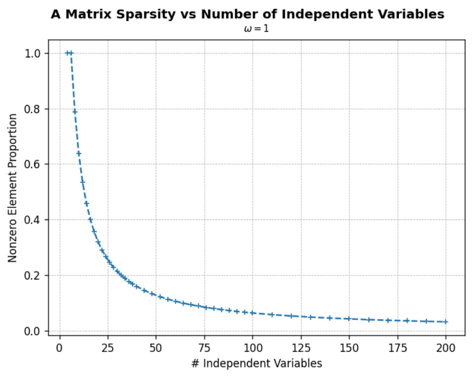


Figure 3: # Cliques vs # Elements

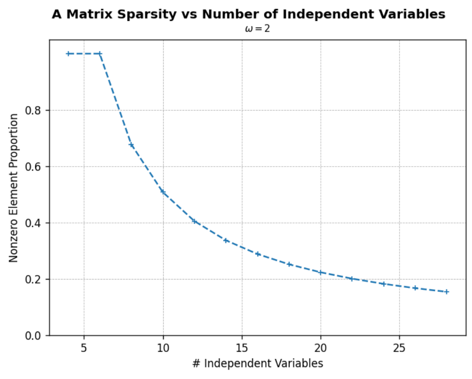
Additionally, it is useful to consider how the number of cliques found in each convex problem increases with the number of independent variables. From Figure 3 on the right, we observe that the number of detected cliques scales linearly with the problem size. Again, this behaviour is expected as the maximum possible number of cliques for a chordal graph is known to scale linearly with graph size, and is an essential insight which guarantees that the number of cliques will remain palatable for large scale applications. It is necessary to mention that the number of cliques detected follows an unusually perfect correlation with almost no variance; this may simply be attributed to the algorithm which produced the sample problems. In practice, the number of cliques should see some variance.

4.4.2 *A* Matrix Sparsity

It was mentioned in a previous theoretical section that the relaxation of a polynomial optimisation problem would lead to a convex optimisation problem with a particularly sparse structure. The implementation of this polynomial relaxation process described previously should already provide some intuition for this argument, however it may still be useful to verify and understand the extent of the problem's sparsity as a function of its size.



(a) $\omega = 1$



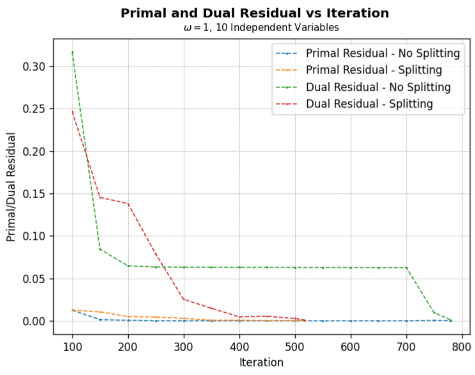
(b) $\omega = 2$

Figure 4: Sparsity of A Matrix vs Number of Independent Variables

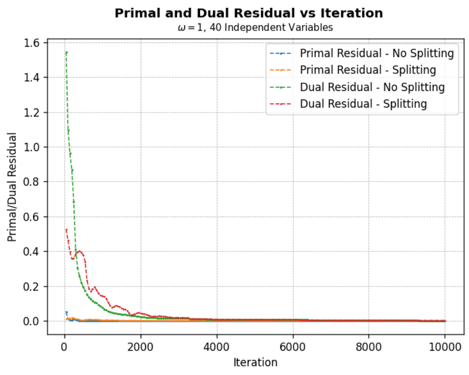
In Figure 4 depicting the proportion of nonzero elements in the A matrix as a function of the number of independent variables, the exponentially decaying density of the matrix can clearly be observed. Comparing the graphs for $\omega = 1$ and $\omega = 2$, we also notice that sparsity at higher ω also increases for the same number of variables, and therefore we can conclude the existence of a positive correlation between the problem size and its sparsity with this convex problem formulation.

4.4.3 Residual Convergence Performance

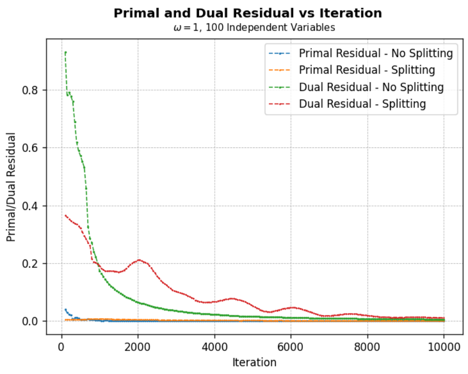
Next, we explore the convergence properties of the primal and dual residuals for various problem sizes and relaxation orders. In Figure 5 below, both residuals are plotted for 4 different problems of different size, and both the clique splitting and benchmark solver residuals are plotted on the same graph for each problem. Note that to improve graph readability, the residuals for the first iteration were removed from some plots, as they were too large and rendered the rest of the data unreadable. In all runs in this implementation, the residual stopping criterion was set at a static $\epsilon = 10^{-3}$ for both the primal and dual residuals.



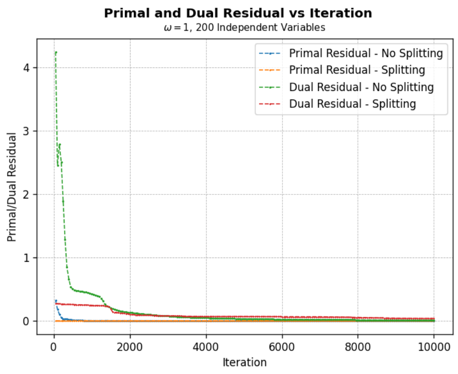
(a) $\omega = 1$, 10 Elements



(b) $\omega = 1$, 40 Elements



(c) $\omega = 1$, 100 Elements



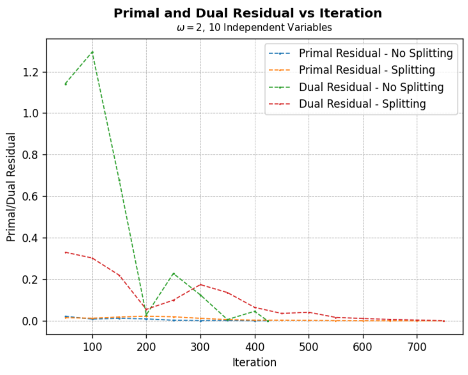
(d) $\omega = 1$, 200 Elements

Figure 5: Primal and Dual Residuals vs Iteration for $\omega = 1$

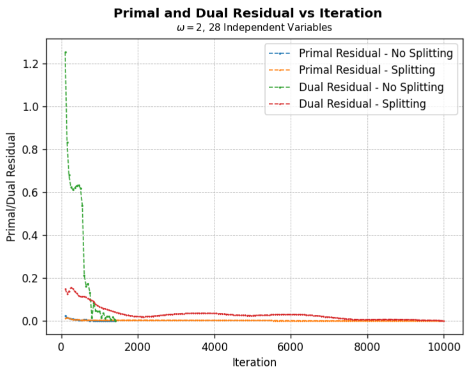
Several insights can already be drawn from this first set of graphs. Firstly, it is safe to conclude that convergence for smaller problems occurs significantly faster than that of larger problems. For a 10-variable problem, both algorithms converged within 800 iterations, while for most larger problems the algorithms ran for the full 10000 iterations without fully converging past the stopping criterion.

Evaluating the relative speeds of convergence between the sparse and benchmark solver, it appears that the sparse solver always starts with superior performance, but is quickly overtaken by the benchmark solver within a few hundreds iterations. This allows the sparse solver to converge faster for smaller problems, but fall short of the benchmark solver for larger problem sizes. Residuals for the sparse solver also appear to oscillate in decaying fashion, while the benchmark solver remains mostly monotonic and asymptotic in its residual convergence. For the sparse solver, this may be due to the global y vector and local s vectors competing for control over the objective minimisation.

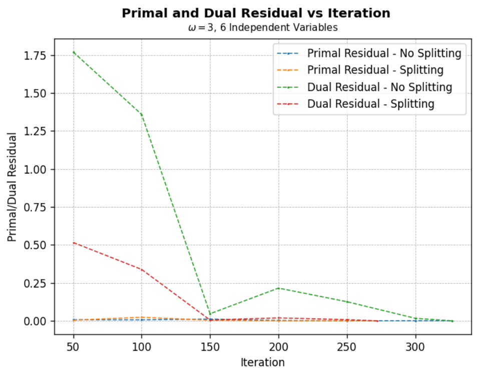
Extending our analysis to Figure 6 below, which depicts the residuals of three different problem sizes for higher ω, we see that the previous trends described above also hold for these examples. Additionally, we can observe that a greater ω does not directly imply that more iterations are required for convergence. The 10 independent variable problem with $\omega = 3$ formulates a larger convex problem than the 200 independent variable problem with $\omega = 1$, yet the former problem converges within 300 iterations while the latter terminates at 10000 iterations without converging. A reason for this observation could be the following. It is possible that a very large moment matrix with relatively few independent variables creates a very tightly coupled set of monomials that simultaneously converge with high efficiency, while a large moment matrix with many loosely couples independent variables are less likely to 'cooperate' with one another and converge together in a short time.



(a) $\omega = 2$, 10 Elements



(b) $\omega = 2$, 28 Elements



(c) $\omega = 3$, 10 Elements

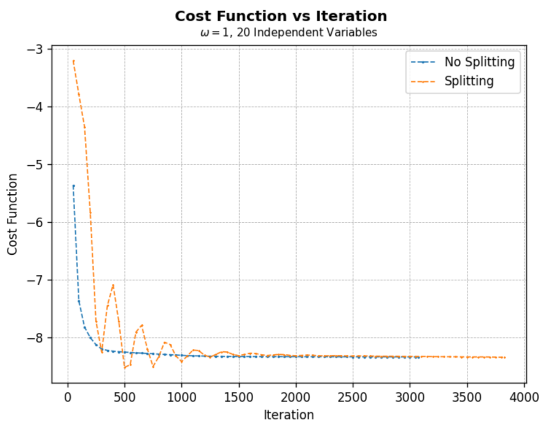
Figure 6: Primal and Dual Residuals vs Iteration for $\omega = 2, 3$

Although not visible graphically, it was observed during the runs that for lower numbers of independent variables and $\omega = 1$, the dual residual for the benchmark solver would occasionally become trapped at some constant value above the convergence criterion, while the primal residual would continue to decrease. The objective cost function would no longer change, however due to the solver's failure to meet the dual residual convergence criterion the algorithm would continue to the maximum allowed iterations. The same behaviour was not observed at all with the sparse solver. The most likely explanation for this is that at the point the algorithm became trapped, the objective function y had already attained its optimal value and hence was oscillating between two very similar points, but the z vector was still being optimised. The convergence criterion ϵ was kept at a constant value for these runs; perhaps implementing a scaled convergence criterion as previously described would allow the algorithm to acknowledge its convergence and prevent similar behaviour in the future.

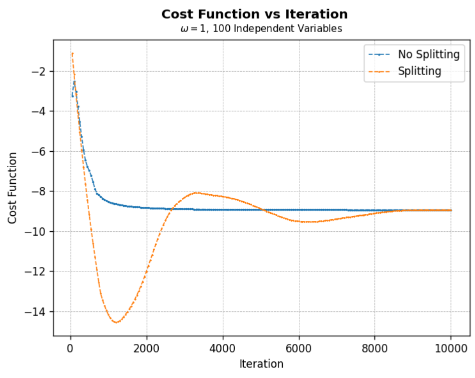
Many more insights are still to be uncovered from analysis the convergence of these residuals. If the reader were to further optimise these prototype solvers, or have access to more computational resources, it would be possible to run these solvers on larger problems, and potentially discover more trends.

4.4.4 Objective Function Convergence

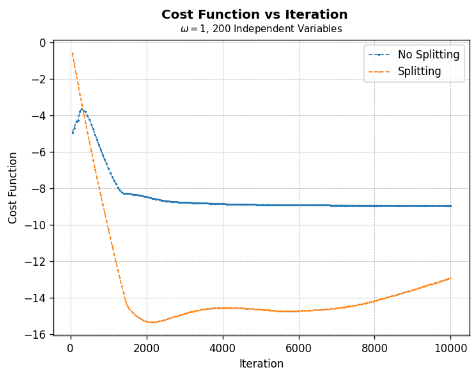
In the execution of each run, it was already confirmed with the Mosek library that both solvers converge towards the same, correct global optimiser, however it may still be useful to observe the objective function plotted against iteration number for a few runs. Three of such graphs are plotted in Figure 7 below, and a quick look at the plots reveals the reason for the oscillatory residual convergence behaviour observed in the previous section.



(a) $\omega = 1$, 20 Independent Variables



(b) $\omega = 1$, 50 Independent Variables



(c) $\omega = 1$, 200 Independent Variables

Figure 7: Objective Function vs Iteration

Specifically, we observe that the objective function for the sparse solver actually oscillates above and below the optimal minimiser in most cases, while the benchmark solver follows a more asymptotic behaviour. The oscillations occur less often as the problem size increases, and for larger problem sizes appears to disappear altogether. For all runs, data was collected every 50 iterations, so the volatile appearance of the oscillations in Figure 7a are actually smoother than they appear.

Figure 7c suggests a worrying trend for the sparse solver; the objective function appears to overshoot the optimal value by a significant margin and remain trapped far from the optimal value for several thousand iterations. Towards the end of the solution, the algorithm begins to correct slowly towards the optimal solution, but terminates before good convergence can be achieved. This behaviour is particularly pronounced for this example, but also manifested itself in other runs to a much more modest extent.

To determine the severity of this phenomena, it would be necessary to perform more runs with similar or larger problem sizes using both solvers. If it is found that this behaviour occurs frequently, more optimisation will need to be done to improve the performance of the sparse ADMM algorithm. In either case, it appears that the benchmark solver currently outperforms the sparse solver in most metrics, suggesting that that tuning of various parameters $(\lambda, \sigma_i, \rho_i)$ and other algorithm optimisations will be required before the sparse ADMM implementation can compete in performance with its benchmark.

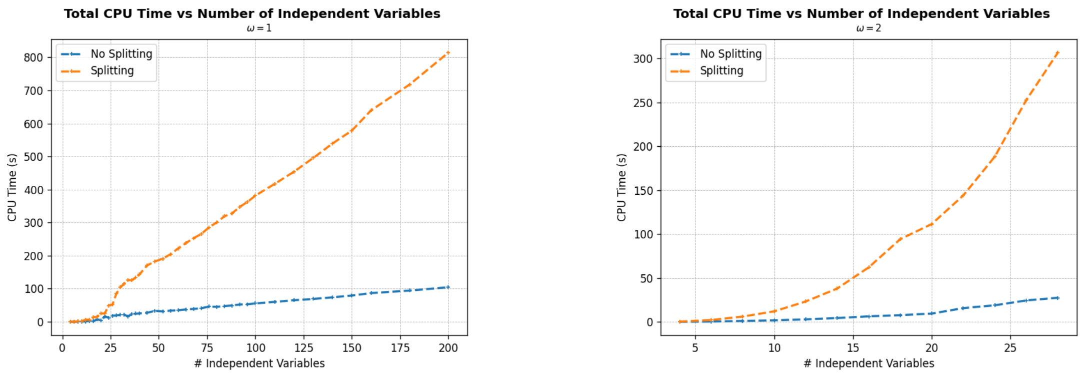
4.4.5 Solver Performance Breakdown by Time

Observing the convergence properties of the solvers is certainly useful, however, arguably the most important performance metric is the time required for each solver to complete its algorithms and solve the optimisation problem. The time taken for each solver to complete various steps within their respective ADMM algorithms were recorded and will be discussed in the following sections.

It is very important to note that as a prototype solver, multiprocessing and parallel computing have not been implemented in the sparse ADMM solver. At each parallelisable step in the algorithm (such as the s_i update step), simple `for` loops were used to execute the step in serial. If the solver is extended to perform all parallelisable steps in parallel, the performance of the parallel solver can increase by a very significant margin, proportional to the number of cliques within the problem. The exact improvement if parallel programming were to be implemented is very difficult to quantify, since the setup of multiple processes also requires significant overhead costs. Nevertheless, the following plots are still indicative of the total amount of computational work required to execute the algorithm, which is still interesting to explore.

4.4.5.1 Total CPU Time

Figure 8 below shows a comparison of the total CPU time required to complete the ADMM algorithm for each solver, for $\omega = 1, 2$. Understandably, with the sparse solver running parallelisable steps in serial, along with the additional steps of updating the s_i vectors and clique detection, the benchmark solver will complete its algorithm with significantly less time. This difference will become more pronounced as the problem size increases, since the number of cliques increases linearly with problem size.



(a) $\omega = 1$ **(b)** $\omega = 2$

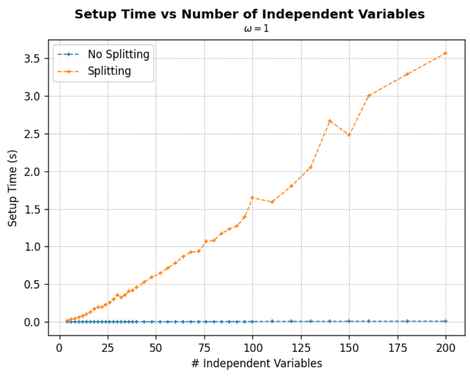
Figure 8: Total CPU Time vs Number of Independent Variables

It is also interesting to observe that the total CPU time for the sparse solver increases linearly with problem size when $\omega = 1$, but appears to increase quadratically when $\omega = 2$, while the benchmark solver shows a linear trend for both relaxation orders. Since an increase in relaxation order increases the size of the resulting cliques without affecting the number of cliques, this trend suggests that at least one step in the sparse ADMM algorithm becomes increasingly computationally costly as the size of the cliques increases.

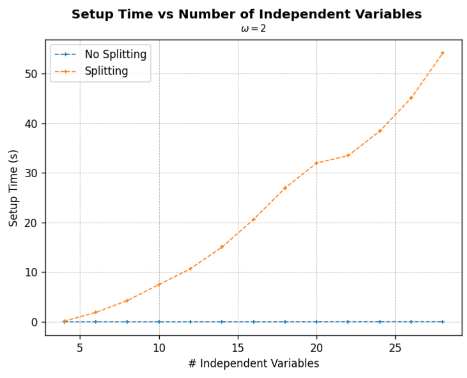
Finally, it is important to note that most runs reached the 10000 iteration cutoff before the convergence criterion could be satisfied, and the point at which the algorithm transitions from convergence to termination at 10000 iterations can be observed at the small chink in the sparse solver trendline. This implies that the actual time required for convergence for the larger problems could be larger than the graphs show.

4.4.5.2 Setup and Clique Detection Time

With the benchmark solver, setting up the algorithm programmatically only involves initialising a few class objects and variables for convenience and readability, and thus requires a negligible amount of time and computational resources. On the other hand, the setup for the sparse solver requires the currently very expensive step of detecting cliques from the matrix of codependencies. The entire setup process, including the distribution of cliques (once they are detected) into different component objects, takes negligible time relative to the single clique detection function. As mentioned in a previous section, the current implementation runs in exponential time, and this high cost is clearly visible from Figure 9 below.



(a) $\omega = 1$



(b) $\omega = 2$

Figure 9: Setup Time vs Number of Independent Variables

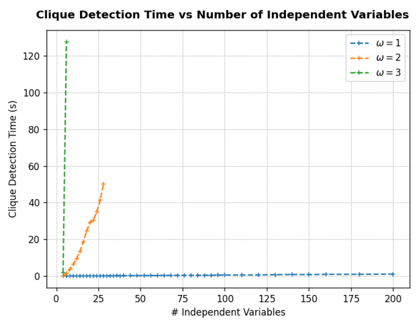


Figure 10: Clique Detection Time vs Number of Independent Variables

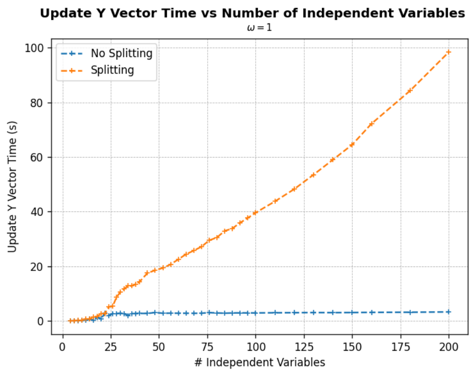
Interestingly again, the setup time for $\omega = 1$ appears to scale only linearly with problem size, with the setup time for $\omega = 2$ scaling approximately quadratically. It is possible that the NetworkX implementation has some hidden levels of optimisation for certain graph structures.

However, looking now at Figure 10 on the right, this implementation becomes far less promising. Plotting the above two trendlines from Figure 9 on the same plot, one can observe the extreme increase in processor time when the relaxation order increases. The almost vertical green line on the left of the plot shows the increase in clique detection time for $\omega = 3$ when the number of independent variables increases from just 4 to 6. At 8 independent variables, the current clique detection implementation already becomes computationally impossible to run.

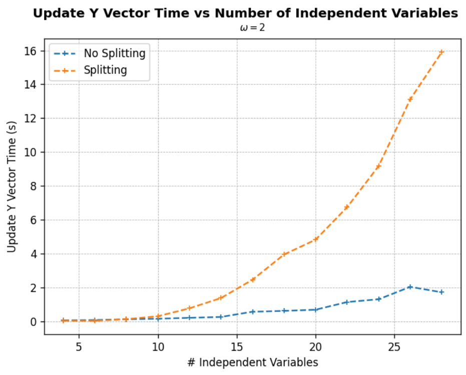
Theoretically, there already exists an algorithm that can determine the cliques of a chordal graph in linear time, with an available library that implements it. Unfortunately, difficulties were encountered when attempting to implement it for this project; the details of which will be described in the final section of this paper regarding further research.

4.4.5.3 Y Update Time

This is the first of three steps in the ADMM algorithm. Although this is an update of the global variable y, a glance back at the update formula for the sparse implementation reveals that it requires the summation of a few variables from each clique component. Therefore, it is also expected that the sparse solver running in serial will also require more and more time as the number of cliques increases, and this is verified by Figure 11 below. Since matrix and vector algebra is also involved, it is understandable that increasing the size of the cliques also has significant effect on the computational cost of this step, and this is seen by the quadratically increasing cost in Figure 11b.



(a) $\omega = 1$



(b) $\omega = 2$

Figure 11: Update Y Vector Time vs Number of Independent Variables

Although it appears from these plots that the sparse solver is completely dominated by its benchmark in terms of its performance, it is necessary to emphasize once more that the performance of the sparse solver can increase dramatically if a multiprocessing implementation was completed. For example, the problem with 200 independent variables has approximately 100 cliques, therefore parallelising the process with 100 worker nodes can increase the speed of this step by up to 100 times, only limited by the additional overhead costs of initialising and assigning tasks to pools of workers. Therefore, the concept of a parallel solver and this sparse ADMM approach should not be dismissed until at least a multiprocessing implementation is completed and tested.

4.4.5.4 s Update Time

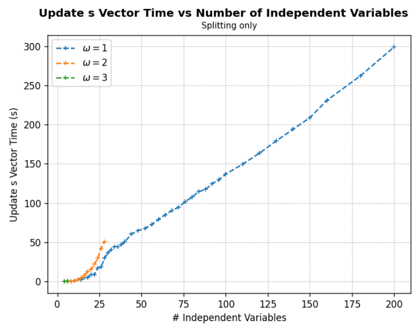


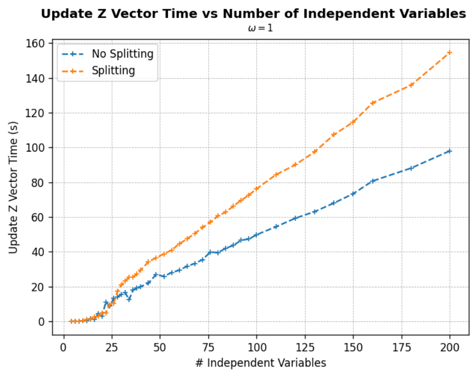
Figure 12: Update s Vector Time vs Number of Independent Variables

The local s_i vectors update is the only step unique to the sparse ADMM algorithm, and is therefore a full additional overhead cost relative to the benchmark implementation. In Figure 12 on the right, the chink in the trendlines that signify the transition between convergence and iteration limit cut-off is much clearer, occurring at around 32 independent variables for $\omega = 1$ and around 20 independent variables for $\omega = 2$. The trend before the transition indicates an approximately polynomial scaling with the number of variables. This may be acceptable for smaller problem sizes when multiprocessing is implemented, since this step is parallelisable with respect to the number of cliques, but could become more problematic when attempting to scale to arbitrarily large problems.

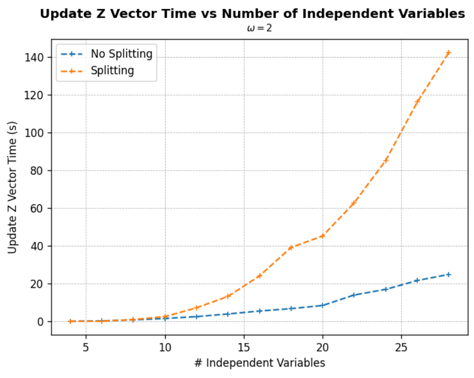
A revisit of the update step formula shows that the step is inherently computationally expensive, with many matrix and vector operations to perform. Since we are aware that a linear increase in the number of independent variables leads to a quadratic increase in the total number of elements in the A matrix, a quadratically scaling trend can be expected. Therefore, the only optimisation possible at this step would be an increase in the efficiency of the matrix and vector operations, perhaps by ensuring that they are all of the same matrix type, or by implementing the algorithm in a faster programming language such as `C++`.

4.4.5.5 z Update Time

This conic projection step, as seen in Figure 13 below, shows a smaller disparity between the sparse solver and benchmark solver, despite the sparse solver looping through each clique and performing these conic projections in serial. As seen from Figure 13 below, the performance between solvers for problems with only a few independent variables is almost identical for this step, understandably because only 1 clique would be detected for small problems. As the problem size increases, a small linear divergence can be observed for $\omega = 1$, with a larger quadratic-like divergence for $\omega = 2$. Given the nature of the clique detection process, where each constraint can be included into multiple cliques, the sparse implementation will simply be doing more conic projections than the benchmark solver as the number of cliques increases.



(a) $\omega = 1$



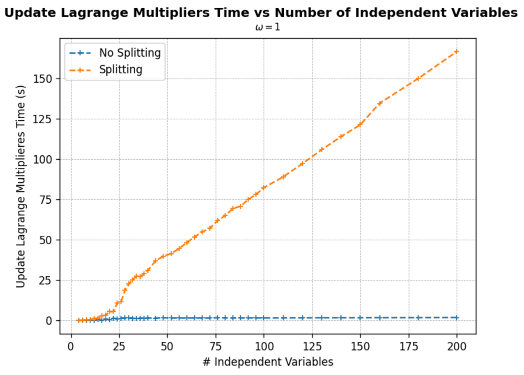
(b) $\omega = 2$

Figure 13: Update z Vector Time vs Number of Independent Variables

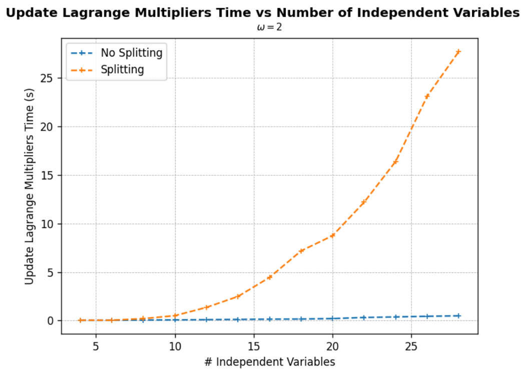
The z vector update step is the most parallelisable step in the ADMM algorithm. Not only can the projections for each clique run in parallel, each individual conic projection within the cliques can also be parallelised. This applies to the benchmark implementation as well. This implies that the efficiency of this step can scale almost infinitely with a large number of parallel processes and a high performance implementation.

4.4.5.6 Lagrange Multiplier Update Time

Lastly, both ADMM algorithms update their respective Lagrange multipliers before moving to the next iteration. For the benchmark implementation, having only one Lagrange multiplier, this occurs only once per iteration. However, the sparse implementation has two sets of Lagrange multipliers, and thus the solver computes two updates for every clique within the problem. The additional workload required for the sparse solver is clearly reflected in Figure 14 below.



(a) $\omega = 1$



(b) $\omega = 2$

Figure 14: Update Lagrange Multiplier Time vs Number of Independent Variables

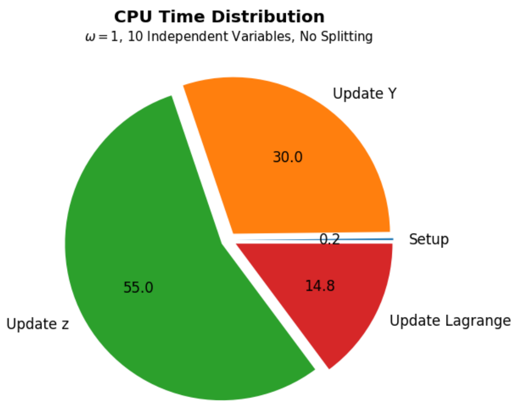
For this particular step, the benchmark solver completely dominates the sparse implementation by performance. Although a parallel sparse solver with many processes may be able to produce comparable performance, it would likely not compensate for the additional computational cost of running a large number of processes in parallel.

4.4.5.7 Distribution of Time between Steps

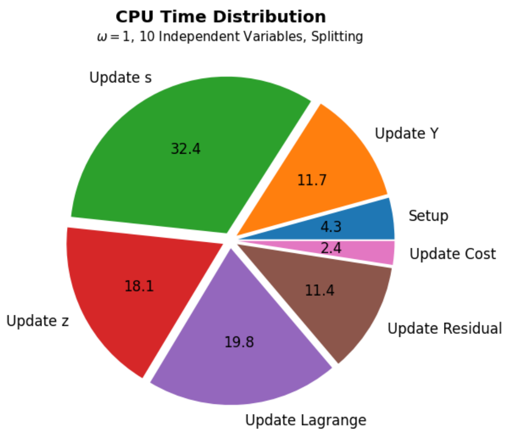
From the performance comparisons in the previous few sections, it is clear that the sparse ADMM implementation is in need of further optimisation before its performance can stand up to that of the benchmark solver. In order to facilitate this, it would be very useful to observe and determine which steps are demanding the highest computational resources for both algorithms. A percentage breakdown of the relative time taken for each step was plotted for various runs, and the results will be discussed below.

We start with an analysis of Figures 15, 16, 17 and 18 below, which plot the time distribution for both solvers for four problems of increasing size with $\omega = 1$. Observing first that for the benchmark solver, it is clear that the y vector update and Lagrange multiplier update steps remain relatively cheap, while the conic projection step demands an ever increasing proportion of the computational resources with increasing problem size. The setup time, having no cliques to detect, remains negligible throughout, along with the residual update calculation step (not displayed on plots). The high cost of the conic projection step is understandable; the projection of positive semidefinite cones requires solving the eigenvalue problem, a computationally expensive algorithm that runs with polynomial time complexity [23]. With that said, it is logical that any further optimisations to the benchmark ADMM solver should be focused on the conic projection step, and will likely involve finding the most efficient approach for solving many large eigenvalue problems in parallel.

Considering now the equivalent breakdown for the sparse solver, we observe that the time distribution for each step remains largely balanced for the different problem sizes. This implies that no particular step scales very well or very poorly relative to one another as the number of independent variables in the problem increases. The local s_i vector update step consistently remains as the most expensive, followed by the conic projection step. Efforts to improve the efficiency of the sparse implementation should be spread across all steps, probably with a small emphasis on the residual update component, which should be computationally cheap and is currently occupying an unusually large proportion of CPU time.

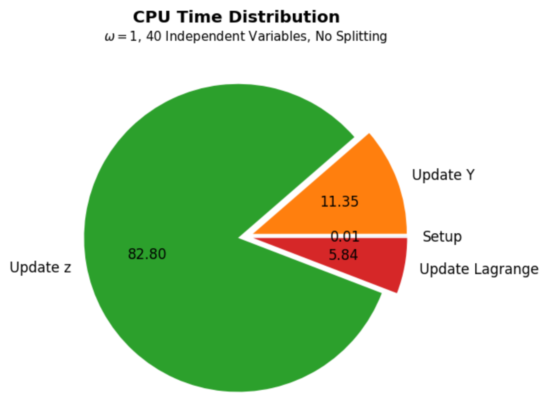


(a) No Splitting

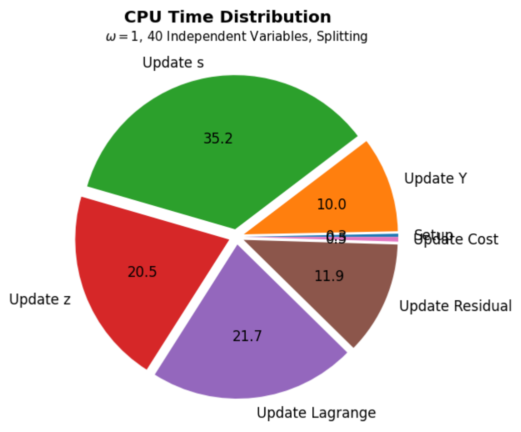


(b) Splitting

Figure 15: CPU Time Distribution for $\omega = 1$, 10 Independent Variables

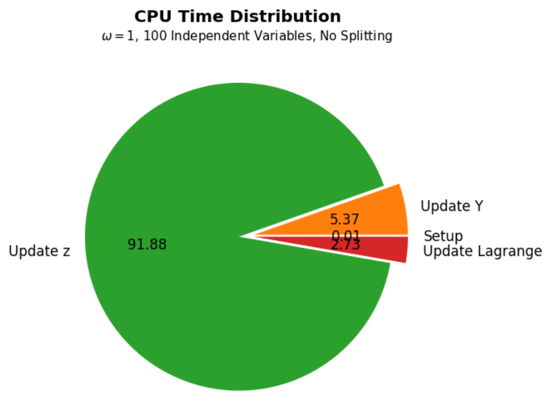


(a) No Splitting

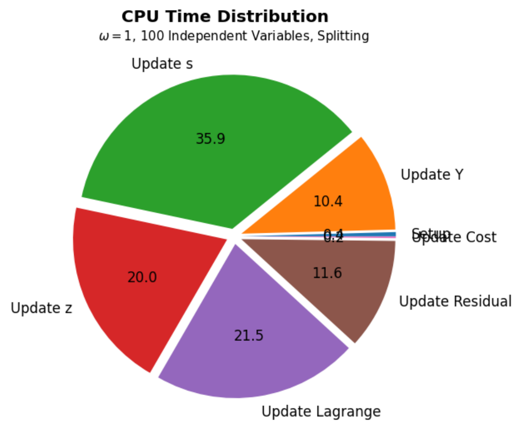


(b) Splitting

Figure 16: CPU Time Distribution for $\omega = 1$, 40 Independent Variables

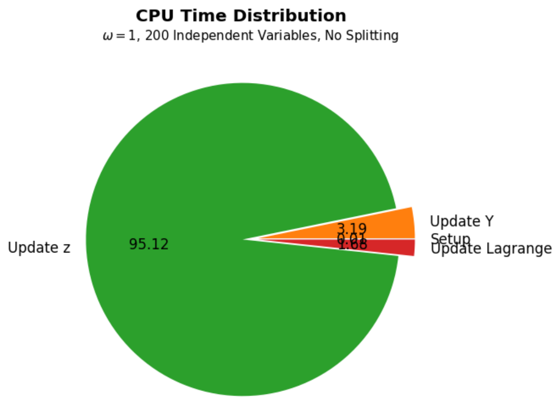


(a) No Splitting

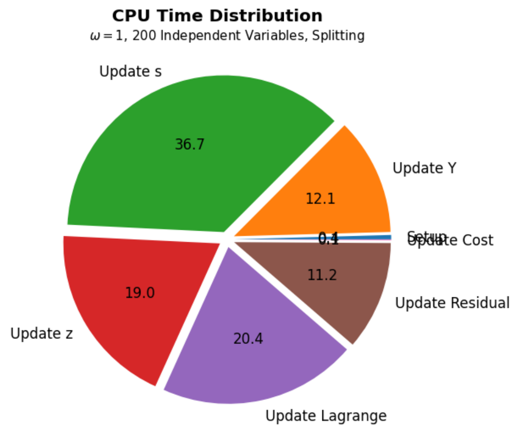


(b) Splitting

Figure 17: CPU Time Distribution for $\omega = 1$, 100 Independent Variables



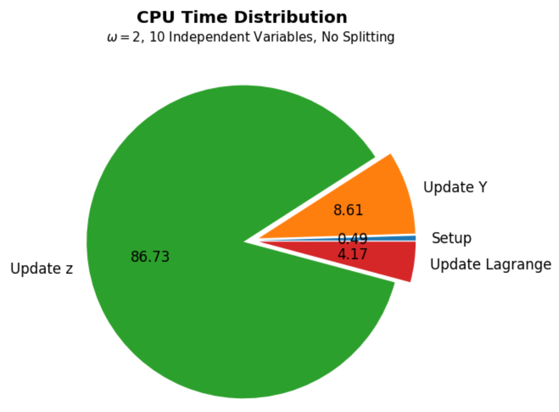
(a) No Splitting



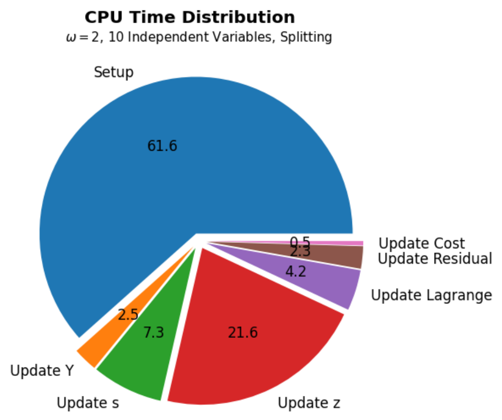
(b) Splitting

Figure 18: CPU Time Distribution for $\omega = 1$, 200 Independent Variables

Next, we consider the effect on these time distributions for problems with higher ω. Given the computational limitations for this project, far fewer runs were performed for $\omega > 1$, yet still enough were completed to derive a few interesting insights. Considering Figures 19, 20 and 21 below, a very similar trend for the benchmark algorithm can be observed, implying that increasing ω does not greatly influence the algorithm's resource distribution. On the other hand, a very significant distribution change can be seen for the sparse ADMM implementation. From Figures 19 and 20, we see that the conic projection step begins to demand a significantly greater proportion of resources with increasing number of independent variables, while at lower numbers of variables the setup time dominates the resource allocation. The latter effect is greatly amplified when ω is large with small numbers of variables, as clearly seen in Figure 21.

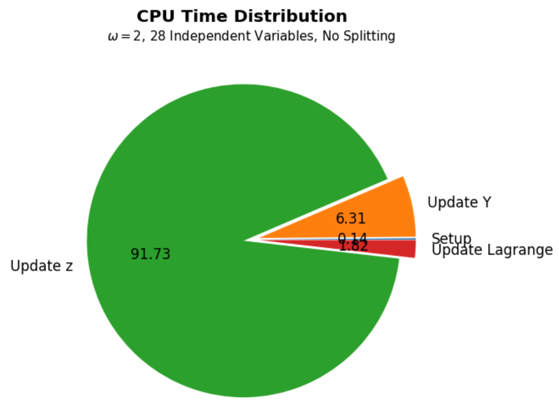


(a) No Splitting

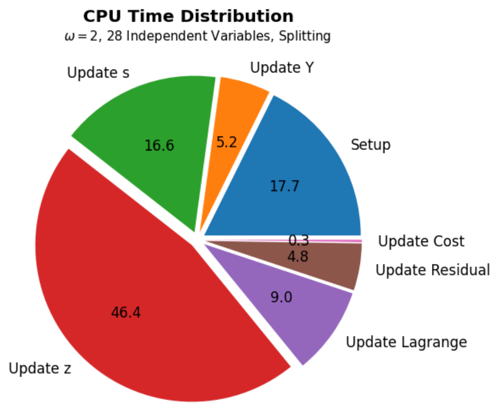


(b) Splitting

Figure 19: CPU Time Distribution for $\omega = 2$, 10 Independent Variables

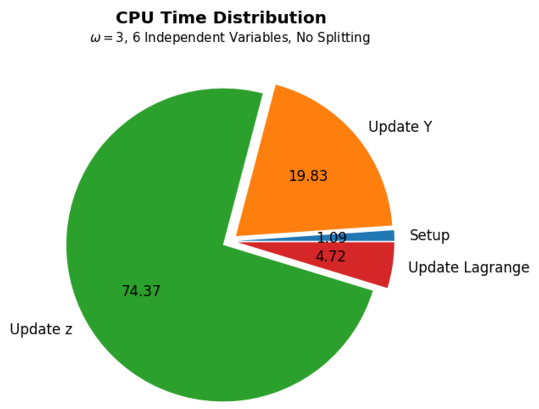


(a) No Splitting

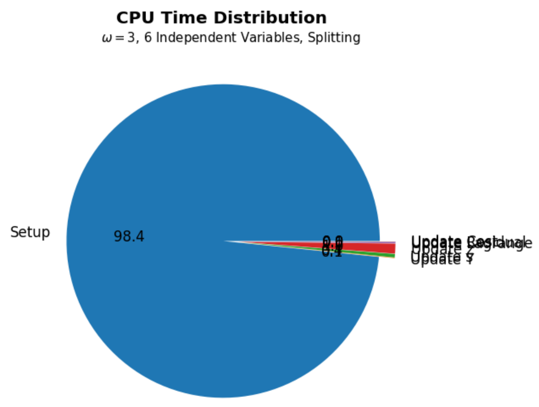


(b) Splitting

Figure 20: CPU Time Distribution for $\omega = 2$, 28 Independent Variables



(a) No Splitting

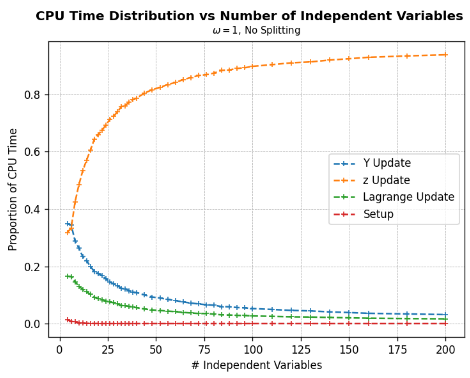


(b) Splitting

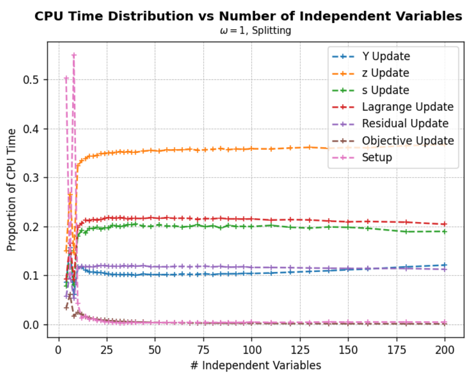
Figure 21: CPU Time Distribution for $\omega = 3$, 6 Independent Variables

We can attribute the increasing computational demand for the conic projection step to the nested `for` loops in the prototype solver, requiring the solver to iterate over all cliques and all constraints within each clique in order to perform each projection in serial. In Python, looping through nested `for` loops is a straightforwardly readable but inefficient implementation, unfortunately necessary for a prototype solver that does not make use of parallel computing. Particular attention to optimising this step is certainly recommended, although an efficient parallel implementation of the projections should very strongly contribute to improving its overall performance. With regard to the rapidly increasing clique detection time for higher ω, an improvement is suggested in the following section that, if implemented correctly, will run in linear time and very significantly improve the efficiency of this step.

The evolution of the total CPU time distribution between each step can be observed more clearly by plotting these step time proportions as a function of the number of variables in the problem, and this can be seen in Figures 22 and 23 below. As expected, we see an ever increasing dominance of computational resource demand for the conic projection step for the benchmark solver, while the sparse solver shows a more balanced distribution, also led by the conic projection step.

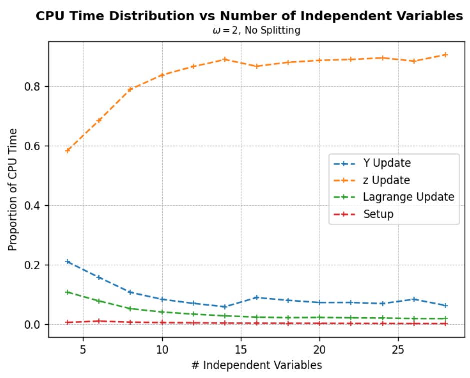


(a) No Splitting

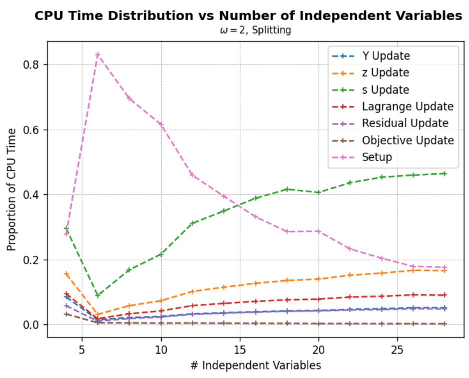


(b) Splitting

Figure 22: CPU Time Distribution vs Number of Independent Variables for $\omega = 1$



(a) No Splitting



(b) Splitting

Figure 23: CPU Time Distribution vs Number of Independent Variables for $\omega = 2$

For $\omega = 2$, the setup time for the sparse solver initially dominates the CPU time, before quickly giving way to the s vector update step. It was previously discussed that this step would be particularly expensive with this implementation due to the nested for loops through every conic projection of every clique, and this is once again reflected in Figure 23. This concludes our discussion of the sparse ADMM conic optimisation solver.

5 Conclusion and Further Work

5.1 Suggested Further Research

The scope of convex problem optimisation and the ADMM algorithm is far reaching. There are many components to the sparse implementation of this algorithm, several of which were not explored in this project. The following section hopes to provide a detailed list of suggestions for further work and research.

5.1.1 Optimised Clique Detection

As previously mentioned throughout previous sections, the currently implemented clique detection algorithm is extremely expensive for larger problems, running in exponential time. The results in the previous section have shown that this step quickly takes over the a large majority of the computational resources. Theoretically, the cliques of a chordal graph can be found in linear time [28], and the Python library `chompack` (Chordal Matrix Package) [2] offers this implementation.

To retrieve the clique sets from a codependency matrix using this library, a chompack `symb` object needs to be created from a `cvxopt` sparse matrix [21]. From there, the `cliques()` method of the `chompack` object returns the cliques of the chordal graph, and the inverse permutation matrix is given by the `ip()` method, if required. Detailed implementation instructions can be found on the package website.

An attempt was made to use the chompack library for this project. It was found that the `symb` object functioned the same as the NetworkX implementation and produced the same cliques for smaller problems. However, when same implementation was applied to larger problems, the `symb` object surprisingly produced a different set of cliques which resulted in an error further within the code. Due to the larger problem sizes, it was impossible to debug the reason for this difference during the course of this project, and the code was reverted to the original NetworkX implementation. If one were considering to use this recommended `chompack` implementation, it will be necessary to investigate the difference between the two algorithms and determine the cause of the disparity. The cliques of a chordal graph are unique, and thus the same set of cliques should be produced regardless of the algorithm used to find them.

5.1.2 Parallel Programming Implementation

The core motivation for the creation of the sparse ADMM algorithm is the ability to decompose the convex optimisation problem into smaller, parallelisable components. Therefore, it is logical that an important extension to this project would be to improve on the prototype solver such that the steps which can be executed in parallel are done so. In this project's prototype solver, these steps are clearly marked by `for` loops which currently iterate through and run the decomposed problem in serial. They have also been described in detail throughout Section 3 of this paper.

If one wishes to implement a parallel solver using Python, it is suggested that one explores the `multiprocessing` [13] and `joblib` [6] Python libraries. The `joblib` package provides a convenient wrapper around `multiprocessing` which allows for the convenient implementation of parallel `for` loops, while `multiprocessing` is a lower level but more robust library which allows one to initialise and assign tasks to pools of worker nodes to be executed in parallel. Both libraries are very well documented and all the necessary information can be found on their respective websites.

The reader is also free to explore implementations in other programming languages. Since computational efficiency is the ultimate objective for this application, `C++` may be a suitable language of choice. The `MPI` and `OpenMP` libraries in `C++` both support efficient and scalable implementations of parallel programming.

5.1.3 Varying λ, σ_i and ρ_i

Throughout this project's implementation of the sparse ADMM algorithm, the objective function weighting parameter λ as well as the penalty parameters σ_i and ρ_i were kept constant for all cliques and iterations. Finding a suitable formula for varying these parameters may result in an improvement in the algorithm's performance and convergence characteristics.

Firstly, it would be interesting to observe the impact of changing λ on the algorithm's behaviour. From earlier sections, it was seen that the algorithm had the objective function oscillate above and below the optimal value as it converged, and it was proposed that this could be due to the global y and local s_i vectors 'competing' for control over the objective function. It is very possible that increasing (or even decreasing) λ would give one side of the objective function more control over the algorithm and thus lead to a more uniform convergence behaviour, and would be very interesting to explore.

Also, varying the penalty parameters with each iteration can also have performance optimisation effects on the ADMM algorithm. In Boyd's paper on ADMM Optimisation [4], a formula is suggested for varying the penalty parameter such that the primal and dual residuals are drawn to remain within a certain factor of one another, and thus can converge to zero together with similar magnitude. Since the sparse ADMM algorithm consists of many primal and dual residual values, we suggest the following adaptation to Boyd's idea.

$$\rho_i^{(k+1)} = \begin{cases} \rho_i^{(k)} \cdot \tau^{incr} & \text{if } \left\|r_{i,primal_1}\right\|_2 > \mu \left\|r_{i,dual_1}\right\|_2 \\ \rho_i^{(k)}/\tau^{decr} & \text{if } \mu \left\|r_{i,primal_1}\right\|_2 < \left\|r_{i,dual_1}\right\|_2 \\ \rho_i^{(k)} & \text{otherwise} \end{cases} \tag{46}$$

$$\sigma_i^{(k+1)} = \begin{cases} \sigma_i^{(k)} \cdot \tau^{incr} & \text{if } \left\|r_{i,primal_2}\right\|_2 > \mu \left\|r_{i,dual_2}\right\|_2 \\ \sigma_i^{(k)}/\tau^{decr} & \text{if } \mu \left\|r_{i,primal_2}\right\|_2 < \left\|r_{i,dual_2}\right\|_2 \\ \sigma_i^{(k)} & \text{otherwise} \end{cases} \tag{47}$$

where $\mu, \tau^{incr}, \tau^{decr} > 1$. As a starting point, the values $\mu = 10$, $\tau^{incr} = 2$ and $\tau^{decr} = 2$ are recommended. In words, these formulae essentially doubles a penalty parameter if the corresponding primal residual exceeds its respective dual residual by more than 1 order of magnitude, and halves the parameter for the opposite condition. The introduction of this step ensures that cliques that are 'poorly' obeying their constraints are penalised more for doing so, while other cliques that more diligently obey their constraints are treated more 'leniently' in the coming iteration. This new penalty parameter update step is a constant time operation that, if performed in parallel across all cliques, should require negligible computational resources, and is a good starting point for adding a new layer of optimisation to the algorithm.

5.1.4 Uncovering More Trends with More Results

There are undoubtedly many more insights that are still to be discovered for the sparse ADMM algorithm. Due to computational limitations for this project and algorithm sub-optimality, the solver was not capable of running larger problems for $\omega > 1$. If one were to further optimise the solver and/or have access to additional computational resources, a more complete set of results could be produced. In particular, it would be interesting to discover the effects of increasing ω for the same number of independent variables. As previously seen from the results, a single increment of ω vastly increases the size and complexity of the resulting problem, so it would be very useful to quantify exactly how useful the additional relaxation order is for the convergence of the algorithm to the true polynomial optimisation problem minimiser. With this information, one can evaluate whether the increased accuracy is worth the additional computational resources for each particular application.

Even with the current set of results produced in this project, there are likely more insights still to be uncovered. The plots presented in the results section ultimately only represent a subset of all the possible plots that can be generated to reveal new insights. For the reader's convenience, all `csv` files containing the full set of results from every run have been uploaded to the GitHub repository in the `results` directory, along with a useful data processing file located in the `plotting` directory. The reader is welcome to explore, process and plot the available data in search for new trends.

5.2 Closing Remarks and Conclusion

The core purpose of this project was to create and test a sparse implementation of the ADMM algorithm to solve convex optimisation problems. This was motivated by the fact that the relaxation of a polynomial optimisation problem results in a conic problem with a particularly sparse structure. In order to decompose the conic problem, a codependency matrix of the problem's objective function and constraints could be formed and represented by a chordal graph. From there, a clique detection algorithm could be used to extract the cliques, and the problem could be decomposed based on each constraint's dependencies. A sparse ADMM algorithm was then proposed and a prototype solver was created using Python.

The solver was run for many sample sparse convex optimisation problems, and its performance was compared an equivalent dense solver also running the ADMM algorithm. Various data points were extracted throughout each run, and many insights were uncovered about the algorithm's behaviour, performance and convergence characteristics. Since the prototype sparse solver did not utilise multiprocessing and instead iterated through all parallelisable steps in serial using `for` loops, it was found that the benchmark solver outperformed the sparse solver in almost all regards. This included better and faster convergence characteristics, as well as superior efficiency and performance.

Despite falling short of the benchmark algorithm, it is proposed that the sparse implementation of ADMM should not be abandoned so hastily, as there are still numerous optimisations which can be made to the solver that would greatly improve its performance. For example, a linear time clique detection algorithm was recommended, along with several other suggestions for further research. If the aforementioned further optimisation suggestions were implemented into a new solver that also utilised multiprocessing and parallel computing, there is reason to believe that the sparse implementation still has the potential to surpass the performance of the benchmark solver.

Overall, all objectives within the scope of this project have largely been attained. A new algorithm for solving convex optimisation problems was motivated, and the idea was successfully implemented, tested and compared. Despite the prototype solver's inability to compete with an established benchmark solver in its current state, this project can still be considered as a successful exploration of a new implementation of the alternating direction method of multipliers for convex optimisation algorithms.

References

[1] Mosek. Available at: `https://www.mosek.com/`.

[2] Martin S. Andersen and Lieven Vandenberghe. Chordal Matrix Package, 2018. Available at: `https://chompack.readthedocs.io/en/latest/`.

[3] Alessandro Astolfi. The Art of Optimization. 2018.

[4] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers. Foundations and Trends in Machine Learning, 3(1):1–122, 2010.

[5] Caihua Chen, Bingsheng He, Yinyu Ye, and Xiaoming Yuan. The Direct Extension of ADMM for Multi-Block Convex Minimization Problems is not Necessarily Convergent. Mathematical Programming, 155(1-2):57–79, 2016.

[6] Joblib Developers. Joblib, 2018. Available at: `https://joblib.readthedocs.io/en/latest/parallel.html`.

[7] NetworkX Developers. Networkx, 2020. Available at: `https://networkx.org/documentation/stable/reference/algorithms`.

[8] Giovanni Fantuzzi. Conic Programs and their Implementation. 2020.

[9] Giovanni Fantuzzi. Parallelizable ADMM Solver for Relaxations of Sparse Polynomial Optimization Problems. 2020.

[10] Giovanni Fantuzzi. AeroImperial Yalmip, 2021. Available at: `https://github.com/aeroimperial-optimization/aeroimperial-yalmip`.

[11] Giovanni Fantuzzi. Decomposition of Sparse Conic Programs. 2021.

[12] Giovanni Fantuzzi. Examples of Polynomial Relaxation, 2021.

[13] Python Software Foundation. Python Multiprocessing, 2021. Available at `https://docs.python.org/3/library/multiprocessing.html`.

[14] Daniel Gabay and Bertrand Mercier. A Dual Algorithm for the Solution of Nonlinear Variational Problems via Finite Element Approximation. Computers & Mathematics with Applications, 2(1):17–40, 1976.

[15] Donald Goldfarb, Shiqian Ma, and Katya Scheinberg. Fast Alternating Linearization Methods for Minimizing the Sum of two Convex Functions. Mathematical Programming, 141(1-2):349–382, 2013.

[16] Bingsheng He, M I N Tao, and Xiaoming Yuan. Substitution for Separable Convex Programming. 22(2):313–340, 2012.

[17] Jean B. Lasserre. Semidefinite Programming vs. LP Relaxations for Polynomial Programming. Mathematics of Operations Research, 27(2):347–360, 2002.

[18] Niels Lauritzen. Convex Optimization. 2013.

[19] Antony Lee. Scikit-Sparse Cholesky Decomposition, 2016. Available at: `https://scikit-sparse.readthedocs.io/en/latest/cholmod.htm`.

[20] Adam N. Letchford and Andrew J. Parkes. A Guide to Conic Optimisation and its Applications. RAIRO - Operations Research, 52(4):1087–1106, 2018.

[21] Martin S. Andersen, Joachim Dahl and Lieven Vandenberghe. Python Software for Convex Optimization. Available at: `https://cvxopt.org/`.

[22] Masakazu Muramatsu and Masakazu Kojima. Aspects of SDP Relaxation for Polynomial Optimization. Available at: `https://www.mat.univie.ac.at/~neum/glopt/gicolag/talks/muramatsu.pdf`.

[23] Victor Y. Pan and Zhao Q. Chen. The Complexity of the Matrix Eigenproblem. Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing, 1999.

[24] J. M. Robson. Algorithms for Maximum Independent Sets. Journal of Algorithms, 7(3):425–440, 1986.

[25] Thomas Rothvos. The Lasserre Hierarchy in Approximation Algorithms. Lecture Notes for the MAPSP, pages 1–25, 2013.

[26] Robert Endre Tarjan and Anthony E. Trojanowski. Finding a Maximum Independent Set. SIAM Journal on Computing, 6(3):537–546, 1977.

[27] Sivan Toledo. Computing the Cholesky Factorization of Sparse Matrices. pages 1–25, 2007.

[28] Lieven Vandenberghe and Martin S. Andersen. Chordal Graphs and Semidefinite Optimization. Foundations and Trends in Optimization, 1(4):241–433, 2015.

[29] Sadanand Vishwas. Using Bron Kerbosch Algorithm to find Maximal Cliques. Available at: `https://iq.opengenus.org/bron-kerbosch-algorithm/`.

[30] Hayato Waki, Sunyoung Kim, and Masakazu Kojima. Sums of Squares and Semidefinite Program Relaxations for Polynomial Optimisation Problems. 17(1):218–242, 2006.

[31] Zaiwen Wen, Donald Goldfarb, and Wotao Yin. Alternating direction augmented Lagrangian methods for semidefinite programming. Mathematical Programming Computation, 2(3-4):203–230, 2010.

[32] Richard Y Zhang, Cédric Josz, and Somayeh Sojoudi. Conic Optimization with Applications to Machine Learning and Energy Systems. 2018.