Efficient Handling of Dependence Analysis for Arrays

Robert Seater
Haverford College
Haverford, PA 19041
rseater@haverford.edu

Thesis Advisor: Dr. David G. Wonnacott
davew@cs.haverford.edu

Presented to the
Haverford College Computer Science Department
to fulfill the Undergraduate Computer Science Major

This work is supported by NSF grant CCR-9157384.

May 2, 2001

Abstract

Array dependence analysis provides valuable information for supercomputer compilers and parallel optimization. However, the worst-case time complexities for array dependence analysis techniques are either not well understood or alarmingly high. For example, the Omega Test uses a subset of the $2^{2^{O(n)}}$ language of Presburger Arithmetic for analysis of affine dependences. Even traditional data dependence analysis of affine dependences is equivalent to integer programming, and is thus NP-complete. Solving the systems of inequalities for feasibility which arise is equivalent to Linear Programming Satisfiability, and consequently exponential at worst. However, there exist techniques for bringing down this worst case both through clever algorithms and careful choice of efficient (but common) subdomains. By doing both of those things, equality and inequality constraints can be handled efficiently (low order polynomial time). However, disequalities are still exponential at worst.

In this paper, I discuss what array dependence analysis is, and how we can phrase questions about dependencies as questions about integer feasibility. I present the straight forward approaches to dealing with equalities, inequalities, and disequalities. I also give a sampling of the more subtle approaches to each; with a particular focus on improving Fourier's Method and on the L1(2)-unit subdomain. Furthermore, I present the workings of a new approach to reduce the exponential behavior of disequality analysis.
1 Introduction

Array dependence analysis is a valuable tool for supercomputer compilers and multiprocessing. However, the worst-case time complexities for array dependence analysis techniques are either not well understood or alarmingly high. For example, the Omega Test [Pug92] uses a subset of the $2^{2^{O(n)}}$ language of Presburger Arithmetic for analysis of affine dependences. Even traditional data dependence analysis of affine dependences is equivalent to integer programming, and is thus NP-complete. Solving the systems of inequalities which arise is equivalent to Linear Programming Satisfiability, and consequently exponential at worst. However, there exist techniques for bringing down this worst case both through clever algorithms and careful choice of efficient, but common, subdomains. By doing both of those, equality and inequality constraints can be handled efficiently (in low order polynomial time). However, disequalities are still exponential at worst.

In this paper, I discuss what array dependence analysis is, and how we can phrase questions about dependencies as questions about integer feasibility. I give a number of examples showing how this is done, and then move on to discuss more rigorous techniques. I present the straight forward approaches to dealing with equalities (using substitution), inequalities (using Fourier-Motzkin elimination), and disequalities (using disjunction and solving exponentially many subcases).

I then provide descriptions of several more subtle approaches to each. I introduce the LI(2), LI(2)-unit, and LI(2)-monotone subdomains, and give algorithms which take advantage of them, as well as some that work in general. I discuss Pugh’s use of modhat to reduce coefficient growth [Pug92], Pugh’s extension to Fourier’s Method [Pug92], and Nelson’s approach using chains and Helly’s Theorem [Nel78], as well as several other smaller techniques [HN94]. Furthermore, I present some work on a conservative preprocessing algorithm to reducing the exponential behavior of disequality analysis.

First, Section 1.1 provides a brief glossary of terms and variable names which I use consistently throughout the paper. In Section 2, I define in detail what dependence analysis is and the different types of dependencies. In Section 3, I then describe how to convert questions about dependencies into questions about integer feasibility. Section 4 describes the basic approach to dealing with equality constraints. Section 5 contains a detailed description of Fourier’s Method and shows how to use it to find real solutions to systems of inequalities. Section 6 gives a description of the exponential algorithm used for analyzing disequalities. In Section 7, I introduce the second part of the paper, where I go into more depth and discuss more subtle approaches to the problems presented in the previous sections. In Section 8, I describe several efficient subdomains for equality constraints, and also give an algorithm to reduce the growth of coefficient size and during equality substitution. In Section 9, I present the notion of a consequence and some related terminology. Section 10 revisits Fourier’s Method, but this time uses Pugh’s extension of that method to solve for integer solutions [Pug92]. Section 11, describes a different variation on Fourier’s Method, as described by Nelson [Nel78]. In Section 12, I describe a polynomial time preprocessing algorithm which attempts to find integer solutions by way of identifying fat polytopes. Section 13 presents new work on reducing the exponential behavior of disequality analysis. Section 14 concludes the thesis, and ties together the ideas presented here.

1.1 Terminology

Throughout this paper, I make an effort to use consistent and standardized terminology and variable names. Terms and variable names which I use only locally will be defined on a section-by-section basis. However, in the interest of clarity, I will provide a brief glossary of terms I use globally throughout this paper.

Dependence analysis is the study of finding and classifying restrictions on reordering the statements in program. This form of analysis is based on the idea that two statements which refer to the same location
in memory cannot be reordered without changing the output (and accuracy) of the program. In Section 2, I will provide some vocabulary which is more specific to dependence analysis.

As we will see in Section 3, questions about dependencies can be determined by analyzing a system of equations generated from the program. A system of equations is said to be feasible when it has solutions over the reals. The system is said to have integer feasibility if it has solutions over the integers. Some sources prefer to use the terms solvable and integer solvability instead.

The types of equations we will be dealing with in this paper will be linear inequalities. For consistency, all equations will appear in standard form, meaning that they will be rewritten so that the left hand side of the equation is zero. During dependence analysis, different types of inequalities are treated differently, so it is conventional to use the following means of clarifying:

An equality constraint is an equation of the form

\[ 0 = a_1 x_1 + a_2 x_2 \pm \ldots \pm a_n x_n \pm \text{constant} \]

An inequality constraint is one of the form

\[ 0 \geq a_1 x_1 + a_2 x_2 \pm \ldots \pm a_n x_n \pm \text{constant} \quad \text{or} \quad 0 > a_1 x_1 + a_2 x_2 \pm \ldots \pm a_n x_n \pm \text{constant} \]

In Section 7, we will see that in certain cases, all inequalities can be normalized into the first form.

A disequality is of the form

\[ 0 \neq a_1 x_1 + a_2 x_2 \pm \ldots \pm a_n x_n \pm \text{constant} \]

The density of an equation is the number of variables it contains. An equations is said to be LI(k) if it has density \( k \) or less. A system of equations is LI(k) if each equation in it is LI(k). More detail about these subdomains, as well as the LI(2)-unit subdomain, is given in Section 8.1.

If an equation is LI(2) and the two variables have different signs when written in standard form, then it is said to be a monotone constraint. These are equations of the form \( ax - by \pm c \) for positive constants \( a \), \( b \), and \( c \). We will see applications of this domain in Section 11.

Throughout this paper, I will hold to the conventions that

- \( E \) represents a system of equations,
- \( \alpha \) and \( \beta \) represent linear expressions,
- \( n \) represents the number of variables in \( E \),
- \( m \) represents the number of equations in \( E \), and
- \( e \) and \( d \) will refer either to number of equalities and disequalities in \( E \) (respectively), or as particular equalities and disequalities (as specified by context).

## 2 Dependence Analysis

Dependencies in a program represent restrictions on our ability to reorder the statements of the program without changing the program’s output (i.e. its accuracy). Thus it is important for a compiler to have dependence information, so that it knows which parts of the program can be reorganized to make it run more efficiently and which must be left in their current order. Specifically, data dependences arise from reads and writes which reference the same locations in memory, so observing the interrelation of variable assignments and references is the key to calculating dependences. For scalars, this problem has been well
solved [ASU86]. A fairly comprehensive list of scalar tests, including empirical results on several of the more important tests, can be found in [GKT91].

Classification of dependencies into flow, output, and anti-dependencies is discussed in Subsection 2.1. Some approaches are more specific in defining a dependence, and wish to avoid detecting a dependence which is killed by another dependence. The meaning of this sort of value based dependence analysis is discussed in Subsection 2.2.

When a program contains references to entries of a multi-dimensional matrix, where those references are nested inside several loops, it is generally not nearly as straight forward to identify dependencies. In the case of statements inside of loops, we are concerned with whether or not we can reorder the execution of those statements, as described in Section 2.2. Thus, when a compiler finds that there are no dependencies between iterations of a loop, it can allow those iterations to be run concurrently on different processors. In this manner, dependence analysis of arrays is closely linked to finding parallelism. When analyzing array dependencies, we also worry about which loop, if any, carries the dependence. This classification is discussed in Subsection 2.4.

2.1 Types of Dependencies

Based on the nature of how the flow of information prevents reordering, dependencies are classified as flow dependencies, output dependencies, or anti-dependencies. I will provide examples first for scalar dependencies (for simplicity) and then show how they generalize for arrays. In Section 3, there are more complicated examples which include array dependences.

A flow dependence is a dependence from a write to a subsequent read. This is perhaps the most intuitive kind of dependence - a reordering would change the nature of the read and consequently disrupt the accuracy of the program. This is also the only type of dependence which involves information actually flowing; the other types only represent the potential for information to flow. This notions will become important in Subsection 2.2. For now, consider an example:

\[
x = 5 \quad // \text{write to } x
\]
\[
\cdot
\]
\[
y = f(x) \quad // \text{read from } x
\]

Here, the final value of \( y \) depends on the fact that the write precedes the read.

An output dependence is one from a write to a later write. The first write is not used as far as those two statements are concerned, but were they to be reversed then the final value stored in memory would be different. Thus there is effectively a dependence between the statements.

\[
x = 5 \quad // \text{write to } x
\]
\[
\cdot
\]
\[
x = 10 \quad // \text{write to } x
\]

In this case, the final value of \( x \) differs based on the ordering of the statements.

An anti-dependence is a dependence from a read to a write. Offhand there does not appear to be a dependence, since the read is not affected by the write. However, the fact that the read is not affected by
the write is also a dependence; an improper reordering that places the write before the read would change
the result of the read.

\[ y = f(x) \quad \text{// read from } x \]

\[ x = 5 \quad \text{// write to } x \]

Any reordering that reverses these two lines would alter the value assigned to \( y \).

Sometimes it is useful to consider two reads to the same memory location as an actual dependence
[&GKT91]. In this case, we call it an \textbf{input} dependence.

We can summarize the three types of dependencies as follows:

<table>
<thead>
<tr>
<th>Dependence Type</th>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>flow dependence</td>
<td>write</td>
<td>read</td>
</tr>
<tr>
<td>output dependence</td>
<td>write</td>
<td>write</td>
</tr>
<tr>
<td>anti-dependence</td>
<td>read</td>
<td>write</td>
</tr>
<tr>
<td>input dependence</td>
<td>read</td>
<td>read</td>
</tr>
</tbody>
</table>

### 2.2 Value Based Dependence

Another issue that one might wish to account for is \textbf{value-based} dependence. In order to understand the
need for value based dependence, we need to be more specific about when we allow a reordering. We allow any reordering if it

1. Does not violate existing flow dependencies. It ok to violate other types of dependencies, since they do not represent actually flow of information - just the potential for a flow.

2. Does not create any new flow dependencies.

The idea behind value based dependence is that we are more careful about avoiding “false” dependencies. A dependence should not be counted if another dependence \textbf{kills} it. That is, if one write overwrite another write's information, then any flow dependence origination from the first write has been killed by the second write.

To see how this works, consider a program in which there are two assignments to a location in memory and two read from that location, as follows:

\[ x = 5 \quad \text{// first statement (write to } x) \]
\[ w = f(x) \quad \text{// second statement (read from } x) \]
\[ x = 10 \quad \text{// third statement (write to } x) \]
\[ z = g(x) \quad \text{// fourth statement (read from } x) \]
By the previous definitions, we have

<table>
<thead>
<tr>
<th>Dependence Type</th>
<th>From Line</th>
<th>To Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>A flow dependence</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>B output dependence</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>C flow dependence</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>D anti-dependence</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>E input dependence</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>F flow dependence</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

In this case, C is killed by A, since no information is actually getting from line 1 to line 4. If we do not calculate kills, then we would think that there are no reorderings possible. However, we could actually reorder the statements safely by placing the first two statements after the second two. Such a reordering will not have an effect on the output of the program, but it looks like it would both violate one flow dependence (C) and create another (from the current line 3 to the current line 2). However, those are false dependencies - we don’t want to be restrained by them. Also notice that those are exacting the flow dependencies which are killed.

Thus, not accounting for kills no only adds unnecessary restrictions, but it may actually add false ones. Of course, erring on the side of having too many dependencies will at worst prevent optimization, but will not actually cause any errors - it can only prevent reorderings that are actually ok to make. Furthermore, handling kills adds additional complexity to the system, as we will see in Section 3.1, and may not be worth dealing with in certain applications. More detailed treatments of this can be found in [Pug92], [Won95], and [PW98].

2.3 Array Dependence

When we are dealing with references to array entries, instead of just to scalars, then the situation is slightly different. There cannot be a dependence between a reference to A[2] and a reference to A[3], but there will certainly be a dependence between two references to A[5]. So, what we are really concerned with is whether the subscripts are equal or not. To a certain extent, each entry in an array can be thought of as a different scalar variable. As long as there are no loop variables in the array references, then this is an accurate representation.

However, if we have a reference to A[f(i)] and A[g(i)], for functions f and g, then we have to worry about if it is ever the case that f(i) = g(i). If so, then those references access the same location in memory and we ought to identify a dependence between them. As we will see in Section 3, determining if those subscripts are ever equal requires careful analysis.

2.4 Loop Carried Dependencies

If the dependence does not prevent the reordering of the loop iterations, but simply prevents the reordering of statements within each iteration, then the dependence is said to be loop independent. Otherwise, it is called loop dependent, and the dependence is said to be carried by the outermost loop which cannot be parallelized as a result of the dependence. Consider the following two examples:

For i = 1 to 10
   For j = 1 to 10
      x = 5

6
Here there is a value based dependence, but it is loop independent, and thus does not affect parallelism.

```
M[i,j] = x - 2
```

For $i = 1$ to 10
   For $j = 1$ to 10
      for $k = 1$ to 10
         $M[i,j] = k + j - 2$

Here there is a dependence carried by the $j$ loop, but that dependence is independent of the $k$ loop.

This distinction is important since it determines whether or not the loop can be parallelized. Loop dependent dependencies prevent parallelization of that loop, while loop independent dependencies merely prevent the compiler from restructuring the commands within the loop.

3 Generating Constraint Sets

In order to calculate dependencies among array references, one looks at an equivalent problem of analyzing a system of equations. By correctly translating loop bounds, memory references, and conditional statements (as described below), we can produce a system of linear equations whose feasibility corresponds exactly to the presence (or lack) of a dependence between two chosen memory references. Specifically, the following three statements are equivalent:

(a) There exist integer solutions to the system of equations that was generated.

(b) There exist references to the same location in memory (possibly by different iterations of the loop).

(c) The iterations of the loop in question cannot be parallelized, or reordered as the compiler sees fit.

For this reason, determining the solvability of those systems of equations is a vital step in optimizing a program to take advantage of multiple processors.

In order to understand how one generates and with such constraints, I will first provide some basic techniques. In Subsection 3.1, I describe how one creates a system of constraints from a question about dependence between two memory references. In Subsection 3.2, I list some techniques for visualizing the constraints generates, to provide intuition into why the methods I later present work the way they do. Subsection 3.3 describes how we can use extra variables to capture the fact that an array dependence can be loop carried.

3.1 Converting Dependence to Feasibility

In order for there to be a data dependence,
3.2 Graphical Representations of Constraints

Having a graphical intuition for what the generated systems of constraints looks like will be valuable for understanding the algorithms presented in this thesis. There will typically be a mix of equalities, inequalities, and disequations in the systems considered, so understanding how to picture their interplay is important.

Equalities do not need to be visualized, since they do not directly represent visual restrictions on solution points. Rather, they represent equivalences among the variables, and it is more valuable to think of them in that light. As we will see in Section 4, equalities will be quickly eliminated, and thus won't be present by the time geometric visualization is necessary.

Inequalities are \( n - 1 \)-dimensional hyperplanes which each define a half space of solutions, and thus collectively define a convex \( n \)-dimensional region (\( n \) is the number of variables). As we will see in Section 3.4, the coordinates of any integer solution point to that region correspond to the loop iterations that cause a dependence. Thus, the presence of any integers points within that convex region correlates exactly to the presence of a dependence - unless they are eliminated by disequations.

Disequations are \( n - 1 \)-dimensional hyperplanes which exclude any points they hit from being solutions. They are best visualized as lines and will be represented as dotted lines when drawn in \( \mathbb{R}^2 \). Thus, given the convex regions defined by the inequalities, we will be looking at whether or not the disequations slice through that region enough to hit (eliminate) all of the solutions points. If so, then there are no solutions.
to the system as a whole, and there is no dependence. If they don’t manage to eliminate all solutions, then there does exist a dependence.

3.3 Representing Multiple Iterations

The basic way that a system of equations can represent the fact that the same reads and writes are made over and over, but in different iterations, is by using two variables to represent each memory reference. Doing so will allow us to capture that dependencies can be loop carried.

To see an unpolluted example of this technique, consider the following program:

Loop I

For $i = 1$ to $10$
  $A[f(i)] = ...$
  ...
  $= A[g(i)]$

Our worry is that $f(i)$ on one iteration would have the same value as $g(i)$ on another (possibly different) iteration, since that would make the subscripts of the read and the write the same. That is, they would refer to the same location in memory and thus resist parallelization. Mathematically, we would write

$$i, i' \geq 1 \land$$
$$i, i' \leq 5 \land$$
$$f(i) = g(i')$$

Where $i$ represents the iterations in which the first statement is executed, and $i'$ represents the iterations in which the second statement is executed. That is, in order to catch inter-loop dependencies as well as intra-loop dependencies, we need to permit the variables from one memory reference to be different than those of the first. We could add the additional restriction $i \neq i'$ if we are only looking for loop-carried (loop dependent) conflicts.

On a brief note, one should be aware that for any loop variables $i$ and $j$, we know that $i$ and $j$ will never occur on the same side of an equation as $i'$ or $j'$. This fact follows directly from why we use a second (primed) set of variables, but is important to remember when producing systems of equations.

3.4 Sample Conversions

A sample loop that can be parallelized might look something like this:

Loop II

For $i = 1$ to $10$
  $A[i] = ...$  // write
  ...
  $= A[i+10]$  // read

That is, the write to $A[i]$ will never access the same place in memory as the read from $A[i + 10]$ for these loop bounds, so there is no reason not to run the different iterations on as many as 10 different processors in parallel. We can visualize this simple example in terms of the array $A$. 

9
Index 1 2 3 4 5 6 7 8 10 11 12 13 14 15 16 17 18 19 20 21
A   [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]

lr = lowest read
hr = highest read
lw = lowest write
hw = highest write

Clearly there is never a conflict in this example.

As far as equations go, we have

\[ 1 \leq i \land \\
   i' \leq 10 \land \\
   i = i' + 10 \]

Which, of course, has no solutions.

An analogous loop than cannot be parallelized is the following:

Loop III

For \( i = 1 \) to 10
   \( A[i] = \ldots \)  // write
   \ldots
   \ldots = A[i+2]  // read

In this case, the the read to \( A[i] \) will indeed access the same place in memory as the write to \( A[i + 2] \), so we cannot run the different iterations in parallel. We can visualize this example in terms of the indeces of the array \( A \).

Index 1 2 3 4 5 6 7 8 10 11 12 13 14 15 16 17 18 19 20 21
A   [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]

lr = lowest read
hr = highest read
lw = lowest write
hw = highest write

Clearly, accuracy will be sacrificed if the different iterations are not run in serial.

As far as equations go, we have

\[ i, i' \leq 10 \land \\
   1 \leq i, i' \land \\
   i = i' + 2 \]

There are several integer solutions to the system of equations, namely \( (i = 3, i' = 1) \). This solution corresponds to the fact that the write that happens on the \( i = 3 \) iteration accessed the same memory
location as the read that occurs on the $i = 1$ iteration. Thus those loop executions cannot be reordered and cannot be parallelized.

Now consider a simple example which produces a disequality.

Loop IV

for $i = 1$ to 10
  if($i=5$)
    $A[i] = ...$ // line 1
  else
    $A[i] = ...$ // line 2

Say we want to determine if there is a dependence from line 1 to line 2. We get the following mathematics:

$$1 \leq i, i' \leq 10 \land$$
$$i = i' \land$$
$$i = 5 \land$$
$$i' \neq 5$$

where $i$ is the variable for the iterations when the first line is executed, and $i'$ is used for the second.

That system of equations is unsatisfiable: Line 1 is only executed when $i = 5$, thus only write to $A[5]$. However, line 2 could only write to $A[5]$ and create a dependence if $i' = 5$, but that will never occur. Thus, there does not exists a dependence between those two lines.

In the last two examples, of course, one does not need a tool anywhere near as powerful as Fourier's method, but they serves to provide intuition for why we wish to solve more complicated systems. When there are no nested loop, dependencies are not hard to spot. Here is a more complicated example, for which basic examination and intuition probably fail to convince you one way or another.

Loop V

for $i = 1$ to 10
  for $j = 1$ to $i$
    $A[i - j] = ...$
      ...
      ...
      ...
    $... = A[i + j + 5]$\n
By hand you would probably just solve it by guessing solutions until you either found one or gave up, but solving it rigorously would prove difficult without.

As equations, we have

$$i, i' \leq 10 \land$$
$$1 \leq i, i' \land$$
$$1 \leq j, j' \land$$
$$j \leq i \land$$
$$j' \leq i' \land$$
$$i - j = i' + j' + 5$$

Combining multiple loops with all three types of constraints, we get a much more complicated example:
Loop VI

for i = 1 to 10
    for j = 1 to i+5
        if (i==j)
            M[i,j] = ...
        else
            M[i+1,j-1] = ...

Testing for dependence from the write in iteration i,j of the “then” to the write in iteration i',j' of the “else”, we get the following system of equations:

\[
\begin{align*}
    i, i', j, j' &\geq 1 & i = i' + 1 \\
    10 &\geq i, i' & j = j' - 1 \\
    5 + i &\geq j & i = j \\
    5 + i' &\geq j' & i' \neq j'
\end{align*}
\]

Solving this example, and the last one, requires more than simple observations, so we will have to examine more careful methods. The next four sections will provide basic techniques which will solve for integer feasibility, but not necessarily do so efficiently. The remainder of the paper then discusses more specific complications and more subtle solutions to the problem of determining satisfiability.

For more details and examples about the relation between dependences and systems of constraints, interested readers should consult [PW98] and [Pug92]. For an alternative approach, using graphs instead of systems of constraints, consult [MHL91].

4 Dealing with Equality Constraints

The equality constraints are the first type of constraints that we tackle, since they are both quick to deal with and doing so reduces the complexity of the problem significantly. If we are dealing with an arbitrary set of linear constraints, then we eliminate the equalities through simple substitution. That is, we consider each equality constraint in turn, solve it for one of its variables, and substitute out that variable anywhere else it occurs in the system. Thus, eliminating e equalities from a set of m constraints takes time proportional to em.

This process affects the complexity of the remaining problem in several ways

(a) Obviously, each substitution eliminates one equation from the system, thus overall reducing the total number of equations (m) by e.

(b) Each substitution reduces the number of variables in the system (n) by at least one. There is the potential to remove more than one variable if extra work is done to make clever choices of which variable to solve for, but it is not clear that doing so is worthwhile [Imb93]. Therefore, the process reduces the number total number of variables, n, by e or more.

(c) A single substitution can at worst square the size of the largest coefficient into the system. Thus, as a whole process increases the maximum coefficient size exponentially, raising it from some scalar c to \(c^{2e}\). Section 8.1 provides a more detailed treatment of that growth.
(d) Each time a substitution occurs, the maximum density could at worst go from $s$ to $2s-1$ (to a maximum of $n$). This means that, overall, the process may increase the maximum density of the system from $s$ to the minimum of $2^s$ and $n$.

In the case of solving for integer feasibility (as we are doing for dependence analysis), we are only dealing with integer coefficients and would like to maintain that consistency. After solving the equality constraints and substituting, the coefficients are only guaranteed to be rational, thus it may be necessary to multiply each equation through by an integer. That integer will be the coefficient of the variable solved for. That is, the equality $5 = 3y - 2x$ when solved for $x$ yields $x = \frac{3}{2}y - \frac{5}{2}$. After substituting, each of the equations that use to contain an $X$ has to be multiplied through by 2. This is straightforward enough, but important to keep track of during implementation. More subtle methods for dealing with equalities are discussed in Section 8.

5 Fourier's Test

Fourier's test determines the feasibility of a set of linear inequality constraints. It consists of performing successive projections, each of which eliminates one variable. When the problem has been reduced to just constants, it is trivial (and fast) to determine feasibility.

I will first present Fourier's method over the reals (Section 5.1) and given an example (Section 5.2). Fourier's Method can be extended to the integers with some extra work but without asymptotically increasing its time complexity. I discuss William Pugh's extension ([Pug92]) which is the classic way of doing that extension. Due to the mathematical complexity of that discussion, I leave it to Section 10, where I discuss several other subtiles of the algorithm. I will examine the complexity of Fourier's method over the reals in Section 5.3, and discuss its complexity in more specific situations in Section 10.4.

5.1 Fourier's Test for Real Solutions

Now we are faced with a set of $m$ inequality equations, with $n$ variables scattered amongst them, forming a convex polygon. In each of $n$ steps, we will eliminate one of the $n$ variables until the system is reduced to a trivial state. I will use the convention that on the $i^{th}$ step we will be eliminating the variable $x_i$, and hence transforming the previous system of equations into a the new system $E_i$. Specifically, the procedure is as follows:

1. Consider all inequalities that contain the variable $x_i$.
2. Divide them into those that are upper bounds (of the form $x_i \leq \beta$) and those that are lower bounds (of the form $\alpha \leq x_i$).
3. For each pairing of one lower bound on $x_i$ and one upper bound on $x_i$, we create a new equation $\alpha \leq \beta$ which no longer contains $x_i$. The new constraint is called "a resultant of $x_i$".
4. Add those new equations to the previous set, and eliminate the equations containing $x_i$.
5. Call this new system $E_i$ - it has the same feasibility as $E_{i-1}$, but has one fewer variables.

This process can be visualized as taking an $n$-dimensional hyper-polytope defined by the inequality constraints, and projecting it into $n-1$ dimensions by collapsing it along the $x_i^{th}$ axis. Every intersection of
two hyperplane constraints in \( n \) dimensions has now become an at most \( n - 1 \) dimensional constraint in the projected space. Figure 1 (in Section 5.2) shows a simple example of this for \( n = 2 \).

A lower level description of the above procedure will prove useful in understanding the later modifications to Fourier's method. For alternate explanations of the mathematics, interested readers should consult [Pug92] or [Cha93]. I will use \( z = x_i \) to make the equations easier to follow. Greek letters (\( \alpha \) and \( \beta \)) will represent expressions forming equations in \( E_i \). Lower case English letters (\( a \) and \( b \)) will represent positive integer constants. We wish to preserve the integrality of the coefficients in order to avoid complicating the system of equations.

Consider some lower bound on \( z \) of the form \( \beta \leq b z \) and some upper bound \( a z \leq \alpha \) (\( \alpha \) and \( \beta \) do not contain \( z \), since it has been solved for).

We can combine these constraints to get \( a \beta \leq ab z \leq b \alpha \) which gives us the new constraint (one of the resultants of \( z \)) \( a \beta \leq b \alpha \) in the real shadow (that is, the set of possible real solutions after the projection has been performed).

You will notice that each step has the potential to at worst square the number of equations in \( E \). Since we perform \( n \) such squarings, we will take at worst \( O(n^2) \) time to solve the system. A more precise bound on the complexity of Fourier's Method is presented in Section 5.3. If we are to apply Fourier's method practically, we must either

1. lower this worst case,
2. use a subdomain that lowers the worst case,
3. only apply it to small systems of equations, or
4. hope that we aren't unlucky.

The fourth option is not entirely humorous, since each step will not usually increase the size of each \( E_i \) by anything close to squaring it. However, the size will still increase exponentially over the course of then entire procedure, thus rendering it dangerous to apply to large systems of equations. As we will see later, in Section 10.4, this bound can be brought way down when we are working over the \( L(2) \)-unit domain.

### 5.2 Sample Application of Fourier's Method

Consider the following example of Fourier's method applied to real solutions.

Let the initial set of constraints be

\[
0 = z - w + x \geq -w + 1 \\
0 \geq -y - 2 \\
0 \geq y - 5 \\
0 \geq 3y - 5x - 9 \\
0 \geq w - x - 3
\]

First, substitution is used to remove the equality constraint \( w = x + 2 \), resulting in the following:

\[
0 \geq -z - 1 \\
0 \geq y - 5 \\
0 \geq 3y - 5x - 9 \\
0 \geq w - x - 3 \\
0 \geq z - x - 1
\]
Figure 1: Here, three inequalities define a triangular region of solutions. When projected along the y-axis, the system is collapsed into the dark line marked on the x-axis.

Now, we apply Fourier’s algorithm. On the first iteration, say that $z$ is arbitrarily chosen to be removed. Identifying upper and lower bounds on $z$ returns

$$
\begin{align*}
& z \geq -1 \land \\
& x + 1 \geq z
\end{align*}
$$

which combine to the resultant $x + 1 \geq -1$, thus leaving the set of equations as

$$
\begin{align*}
& 0 \geq -x - 1 \land \\
& 0 \geq y - 5 \land \\
& 0 \geq 3y - 5x - 9 \land \\
& 0 \geq w - x - 3
\end{align*}
$$

Next, assume that $y$ is arbitrarily chosen.

The situation at this state is shown in Figure 1. Note that $x$ would have been an easier choice, since there are only two constraints that involve $x$. However, the basic implementation of Fourier’s Test does not plan ahead about which variable to choose. Identifying upper and lower bound on $y$ returns

$$
\begin{align*}
& y \geq -2 \land \\
& 5 \geq y \land \\
& 3y \geq 5x + 9 \land \\
\end{align*}
$$

which combine into the following resultants:

$$
\begin{align*}
& 5 \geq -2 \land \\
& 25 \geq 5x + 9
\end{align*}
$$
leaving the main set of constraints as

\[
0 \geq -7 \land \\
0 \geq 5x - 16 \land \\
0 \geq -x - 2 \land
\]

Lastly, \(x\) is chosen to be eliminated. Identifying the upper and lower bounds on \(x\) returns

\[
16 \geq 5x \land \\
x \geq -2
\]

which combine to the \(x\)-resultant \(-10 \geq 16\), leaving the main set of constraints as

\[
0 \geq -7 \land \\
0 \geq -26
\]

Which is trivially correct. Thus, the original set of constraints was feasible, meaning that there does exist a dependence in the equivalent program.

### 5.3 Complexity of Fourier’s Test

It is an interesting exercise to determine more precisely the time complexity of Fourier's method on arbitrary input. It is not quite as simple (although equivalent asymptotically) as saying that each iteration, the number of constrains is squared, since some equations are also removed each step. That is, the size of \(E_i\) is only squared if every constraint contains \(x_i\). However, if that is the case, then all of the original equations of \(E_i\) are being thrown out. Thus, the worst case of squaring the size of \(E_i\) and the worst case of not eliminating any previous elements of \(E_i\) cannot coincide. We may benefit from a more detailed amortized analysis.

Thus, we can calculate the worst case size of \(E_i\) recursively \(S(i) = (S(i - 1)/2)^2 - S(i - 1)\) for \(S(0) = m\) over \(n\) iterations.

Asymptotically, the \(-S(i - 1)\) part is irrelevant, and the rest is roughly

\[
S(i) = \frac{S(i - 1)^2}{4}
\]

Thus, over \(m\) iterations, it is asymptotically

\[
\frac{m^{2n}}{2^n}
\]

where \(m \geq n\), making this still just our previous estimate \(O(m^{2n})\).

Applying Mathematica to the problem, we can solve that recursion more precisely and get

\[
\frac{M(n+1)}{4(2^n-1)}
\]

Unfortunately this is still badly exponential. However, doing this kind of careful analysis is not wasted. As we will see in Section 10.4, careful analysis pays off when we are operating on a less general domain. In fact, in Section 11, the key to getting a sub-exponential solutions is to not naively ignore amortized reduction of complexity.
6 Disequality Constraints

Given the convex region of solutions defined by some set of inequalities, we wish to know if the disequality constraints eliminate all of those solutions. Unfortunately, current techniques for dealing with disequalities are exponential [PW98], even for Ll(2)-unit, thus feasibility testing is also exponential as a whole.

The most straightforward way of dealing with disequalities is to treat them as a disjunction of inequalities. Given a disequality of the form $\alpha \neq \beta$ for expressions $\alpha$ and $\beta$, we replace it with a disjunction of the two inequalities $\alpha > \beta \lor \alpha < \beta$.

Introducing disjunction into the system means that we now have an exponential number of subproblems to solve. If there are $d$ disequalities in the system, we would have to run the inequality algorithm completely over again for each of the $2^d$ possibilities. So far, there does not seem to be a way to avoid the exponential approach entirely. However, we can still hope to recognize cases where we can run a faster algorithm, or at least reduce the size of the exponent when we do have to run the exponential one. A new approach to doing this is covered in detail in Section 13.

7 Subtle Techniques for Dependence Analysis

Now that we have examined the basic techniques used during array dependence analysis, I will move on to more subtle approaches. These techniques include both modifications to existing algorithms and restrictions to efficient, but common, subdomains. First I will present the formalization of normalization which is a necessary predecessor to several of the later methods. Then I address subtle methods for approaching equalities, inequalities, and lastly disequalities in Sections 8, 10, and 13 respectively.

7.1 Normalizing and Tightening Constraints

When we are determining the presence of integer solutions, then there is a useful technique that we can apply to simplify the system of constraints we are working with. A normalized constraint is one in which all the coefficients are integers and the greatest common divisor of those integers (other than the constant $a_0$) is 1. Thus $0 \geq 4x + 5y + 5$ and $0 \geq x - y + z - 4$ are normalized, but $0 \geq 3x + 3y + 5$ and $0 \geq 2x - 2y + 2$ are not. This is, of course, independent of whether the equation is an equality, inequality, or disequality.

If the initial set of constraints contains rational coefficients, then we can obtain integer coefficients by multiplying through by an appropriate constant.

To normalize a constraint, first calculate the greatest common divisor of the coefficients, not counting the constant $a_0$. Call this number $g$. If $g$ divides $a_0$, then divide through by $g$ to normalize it; we have simplified the equation without altering it. If $g$ does not divide $a_0$, then geometrically the line is slipping through all of the integer points in $\mathbb{R}^n$. What we do depends on the type of constraint that we are dealing with.

- If the constraint is an equality or disequality constraint, then the constraint is unsatisfiable over the integers. Finding an unsatisfiable equality means that the entire system is unsatisfiable, and we can stop there and not perform any more tests. Finding an unsatisfiable disequality means that we can eliminate that disequality from the system; it eliminates no solutions and thus has no impact on solvability.

- If the constraint is an inequality, then we go ahead and divide through by $g$. Dividing each of the coefficient by $g$ leaves them as integers, except for the constant term $a_0$. Instead, we take the $\text{floor}[a_0/g]$ so that it, too, becomes an integer.
Geometrically, this has the effect of shifting the inequality "inwards" until it contacts some integer points, as seen in figure 2. If it was previously a strict inequality (< or >), then it is not so any more (\leq or \geq). Doing this shift alters the feasibility over the reals, but not over the integers. However, it does have the effect of reducing the coefficient sizes and thus reduces the mathematical complexity of the problem.

Taking floors in the constant term has the effect of tightening inequalities; it moves them closer together without altering integer feasibility. Tightening and normalizing not only have the effect of reducing coefficient size, but they also guarantee us some facts which will prove useful in Section 8.2 when we examine Pugh's use of mod to reduce coefficient growth.

8 Subtle Approaches to Equalities

Recall that each equality substitution has the potential to at worst square the largest coefficient and at worst double the density of the system. By performing some preprocessing before substituting, we can reduce (but not remove) these drawbacks, as described in Subsection 8.2. If we restrict our domain to only include certain types of equations, then we can actually eliminate both of those problems, as described in Subsection 8.1.

8.1 Efficient Subdomains

There are two ways for coefficients to grow as a result of substitution.

a. Substituting "ax = ..." into "... + bx + ..." results in a coefficient of size ab, which can at worst square the largest coefficient.

b. Substituting "y = ax + ..." into "... + y + bx + ..." results in a coefficient size of a + b, which can at worst double the largest coefficient.

While the first form of increase is asymptotically worse, it is important to keep the second in mind when analyzing the usefulness of subdomains.

The Li-unit unit domain specifies that all variables have unit coefficients, and is at first appealing since it is very common in practice [SW00b]. This domain appears at first to be efficient, since it prevents the quadratic growth of coefficients at each substitution. Unfortunately, it does not prevent the second type of
growth (case b above), and thus is not closed under substitution. After a single substitution, the system of equations might now have a coefficient of size 2. Thus the system is no longer LI-unit and thus is once again vulnerable to quadratic growth per step, and exponential growth overall. So, the LI-unit domain is good while it lasts, but useless by itself.

Another domain that is extremely common in practice is LI(2) [SW00b]. The LI(2) subdomain specifies that each equation have a density of at most two. Recall from Section 4 that maximum worst case density grows from some integer s to 2s − 1 after each substitution. Conveniently, for s = 2 this involves no growth at all! Thus the LI(2) domain is closed under substitution and prevents the density growth that previously plagued the system. However, the coefficient growth is still exponential at worst. For more treatment of the LI(2) domain, interested readers should consult [Cha93].

Combining the first two subdomains proves to be very fruitful. The combined domain, LI(2)-unit, not only retains the high frequency in practice of the other two [SW00b], but also completely controls the coefficient and density growth. Density remains bounded by 2 for the same reason that it does for the LI(2) subdomain. As was the case for the LI-unit domain, the quadratic coefficient growth cannot happen (since \(1^2 = 1\)), but with only two variable per equation we also avert the additive growth. Thus the coefficient remain units, and the system of equations remains in LI(2)-unit after all of the equalities are substituted out. This is extremely valuable to know, since the LI(2)-unit domain make inequality and disequality analysis both much more efficient, as we will see in Section 10.4. Commonality combined with closure under substitution makes LI(2)-unit a very powerful domain.

It is important to notice that there is one more type of equations which can be safely included in the LI(2)-unit subdomain without harming its closure. An equation of the form \(2x \geq 0\) can actually be included in the LI(2)-unit, since when we look at it at \(x + x \geq 0\) we can see that it still fits the format. Even though it is, in some sense, not LI-unit, it does not disrupt the fact that LI(2)-unit is closed under substitution. Consequently, we allow it to be considered LI(2)-unit, thus slightly enlarging the size of the subdomain without any loss to its benefits.

### 8.2 Pugh's Modhat

Pugh [Pug92] offers a partial remedy to the problem of coefficient growth, by preprocessing each equality before substitution it into the remaining equations. He uses a variation of modulus division, denoted \(\text{mod}\) or simply "modhat", to reduce (although not actually prevent) the exponential growth of coefficient size.

First, consider the definition of \(\text{mod}\) as follows:

\[
a \mod b := a - b \ \text{floor}\left(\frac{a}{b} + \frac{1}{2}\right)
\]

More intuitively,

\[
a \mod b := \begin{cases} a \mod b & \text{ if } a \mod b \leq |(a \mod b) - b| \\ (a \mod b) - b & \text{ otherwise} \end{cases}
\]

that is, it is defined to be whichever of the two options is closer to zero. In other words, \(a \mod b\) minimizes the absolute value of \(a\), without changing its value when taken mod \(b\). Some numerical examples are provided in Figure 3.

To eliminate some equality \(e := \sum_{i \in \mathbb{N}} a_i x_i = 0\)
Figure 3: Here are a few numerical examples of \( \overline{\text{mod}} \). Notice that \((a \mod b)\) and \((a \overline{\text{mod}} b)\), by definition, will always be either equal or they will differ by exactly \( b \).

we first look for a \( j \) such that \( a_j = 1 \). If we can find one, then solve the equations for \( x_j \) and substitute \( e \) into the other equations in the system. Clearly, this won’t increase the coefficient size for \( x_j \) multiplicatively, though it may still increase another variable’s coefficient size additively. However, that is relatively minor growth compared to the worst case growth.

If there are no unit coefficients, then we have to be more clever. Notice that if we reach this point in the algorithm, then we know that \( a_k \neq 1 \) and \( a_k \neq 0 \). We then identify the non-zero coefficient with the smallest absolute value (there must be one); let \( a_k \) be that coefficient. Now, let \( M = |a_k| + 1 \).

**Lemma:** \( a_k \overline{\text{mod}} M = -\text{sign}(a_k) \)

**Proof:** If \( a_k > 0 \), then \( a_k \overline{\text{mod}} (|a_k| + 1) \) will be just \(-1\). This is because \( a_k \mod (|a_k| + 1) = 1 \) by definition of mod, and \((a_k \mod M) - M\) cannot be closer to 0 without actually equaling 0. It cannot equal 0 since \( M = |a_k| + 1 \) only equals 0 when \( a_k = 1 \). But if we had that \( a_k = 1 \) then we would not have reached this point in the algorithm. Thus if follows that \( a_k \overline{\text{mod}} (|a_k| + 1) = -1 \). Similarly, for \( a_k < 0 \) we get that \( a_k \overline{\text{mod}} (|a_k| + 1) = +1 \). Consequently we get that \( a_k \overline{\text{mod}} M = -\text{sign}(a_k) \), and the lemma is proven.

At this point, we introduce a new variable, \( \sigma \), and produce the constraint \( e' := \)

\[
\sigma = \sum_{i \in \mathbb{N}} (a_i \overline{\text{mod}} M)x_i
\]

Solving this new constraint for \( x_k \) gets us

\[
x_k = -\text{sign}(a_k)M\sigma + \sum_{i \in \mathbb{N}, i \neq k} \text{sign}(a_k)(a_i \overline{\text{mod}} M)x_i
\]

We then substitute this into all of the other constraints, including \( e \). Notice that this does not actually increase the total number of variables, although it changes what that set of variables is. Putting it into \( e \) produces

\[
-|a_k|M\sigma + \sum_{i \in \mathbb{N}, i \neq k} (a_i + |a_k|(a_i \overline{\text{mod}} M)x_i) = 0
\]
Since \(|a_k| = M - 1\), we can rewrite that as

\[-|a_k|M\sigma + \sum_{i \in \mathbb{N}, i \neq k} ((a_i - (a_i \mod M)) + M(a_i \mod M))x_i = 0\]

Since all the terms are divisible by \(M\), we can normalize the equations to the following:

\[-|a_k|\sigma + \sum_{i \in \mathbb{N}, i \neq k} (\lfloor \frac{a_i}{M} + \frac{1}{2} \rfloor + ((a_i \mod M))x_i = 0\]

which I will call \(e''\).

The absolute value of the coefficient of \(\sigma\) is the same as the absolute value as \(a_k\), the coefficient of \(x_k\). Each of the other coefficients has been reduced from \(a_k\) to \(\frac{a_k}{M}\), rounded up to the next integer. Notice that each \(a_k \geq 2\) and they each got divided by \(M \geq 3\). Thus, the most that they could be now is \(\frac{2}{3}\) of their original value, and they are most likely much lower. Therefore, repeated applications of this algorithm will eventually reduce one of the coefficients to a unit. Also notice that these repeated applications will not increase the total number of variables, even though each will replace an old variable with a new one (above, \(x\) is replaced by \(\sigma\)). That variable will then be identified and solved for, allowing us to eliminate the constraint through substitution.

In Subsection 8.4, I prove that the new equality, \(e''\) has integer solutions if and only if the only equality, \(e\), has integer solutions. Thus we can use the new one and keeping down the rate of coefficient growth without sacrificing accuracy.. Unfortunately, it is still exponential in the worst case worst [Pug92], just not nearly as badly so. At the end of Section 8.3, we will see a more detailed explanation of coefficient growth. One of the nice things about \(\text{LI}(2)\)-unit subdomain is that we can avoid this whole mess, since we will always be guaranteed an integer coefficient to solve for.

### 8.3 Sample Application of Modhat

I now given a sample application of the \(\mod\) algorithm, and in doing so give some intuition for why it works. I will give a formal proof later, in Subsection 8.4.

Consult Figure 4 for a sample application of Pugh's modhat algorithm. I will work through some of the mechanics of that example, to provide a better feel for how the algorithm functions.

The details for getting from the first entry to the second is as follows:

We decide to eliminate the first equality first, and choose \(7z\) as having the smallest coefficient. Thus, \(a_k = 7\) and \(M = 8\). We define the new variable \(\sigma\) by \(e' :=\)

\[8\sigma = (7 \mod 8)x + (12 \mod 8)y + (31 \mod 8)z + (-17 \mod 8)\]

which simplifies to

\[8\sigma(-1)x + (-4)y + (-1)z - 1\]

Next we solve for \(x\), giving us

\[x = -8\sigma - 4y - z - 1\]

which we substitute into all of the other equations.

Intuitively, we can see why Pugh's use of \(\mod\) helps as follows: By the way \(\mod\) is defined, the coefficients in the \(e'\) the equations are each answering the question "what is the minimum amount I can be changed
### Substitution

<table>
<thead>
<tr>
<th>Substitution</th>
<th>Resulting Constraints</th>
</tr>
</thead>
</table>
| **Original Problem** | \[7x + 12y + 31z = 17\]  
\[3x + 5y + 14z = 7\]  
\[1 \leq x \leq 40\]  
\[-50 \leq y \leq 50\] |
| \[x = -8\sigma - 4y - z - 1\] | \[-7\sigma - 2y + 3z = 3\]  
\[-24\sigma - 7y + 11z = 10\]  
\[1 \leq -8\sigma - 4y - z - 1 \leq 40\]  
\[-50 \leq y \leq 50\] |
| \[y = \sigma + 3\tau\] | \[-3\sigma - 2\tau + z = 1\]  
\[-31\sigma - 21\tau + 11z = 10\]  
\[1 \leq -1 - 12\sigma - 12\tau - z \leq 40\]  
\[-50 \leq \sigma + 3\tau \leq 50\] |
| \[z = 3\sigma + 2\tau + 1\] | \[2\sigma + \tau = -1\]  
\[1 \leq -2 - 15\sigma - 14\tau \leq 40\]  
\[-50 \leq \sigma + 3\tau \leq 50\] |
| \[\tau = -2\sigma - 1\] | \[1 \leq 12 + 13\sigma \leq 40\]  
\[-50 \leq -3 - 5\sigma \leq 50\] |
| **after normalization** | \[0 \leq \sigma \leq 2\] |

Figure 4: A sample application of Pugh's modhat algorithm.

by to make me divisible by 8?”. Consequently, substituting that equation back into \(e\) alters each of the coefficients of \(e\) in a minimal way so that they are now each divisible by 8. We can thus divide out by 8 without losing integer coefficients: \(\sigma\)'s coefficient returns to 7, and the other coefficients are reduced to an eighth of their previous value, rounded up.

Recall that \(e'\) is substituted in to *every* equation, not just \(e\). It is important that we get the minimum amount of change necessary, since that amount is how much additive coefficient growth there will be when we substitute \(e'\) into the other equations. However, that will always be less growth than we would get by not using \(\mod\); instead of growing from \(c\) to \(O(c^2)\) (\(e\) squarings of \(c\)), for \(e\) equalities, it only grows to \(O(c \cdot 2^e)\) (\(e\) doublings of \(c\)).

### 8.4 Proof of Equivalence

I will give a careful proof for the \(n = 2\) case so that it is easier to follow, and not cluttered up by summations. The proof I give can be easily generalized to work for any value of \(n\).

**Problem:** Consider a normalized equation \(e\) of the form

\[ax + by + c = 0\]

for some integers \(a, b, c\) (with \(a \neq b\) since \(e\) has been normalized). Without loss of generality, let \(a < b\).

Pugh's method directs us to define

1. the constant \(M := a + 1\), and
(2) the integer variable $s$ constrained by $e :=$

$$ms = (a \mod m)x + (b \mod m)y + (c \mod m)$$
$$\Rightarrow Ms = -x + (b \mod M)y + (c \mod M)$$
$$\Rightarrow x = -Ms + (b \mod M)y + (c \mod M)$$

The second statement follows from the lemma given in Subsection 8.2. The third is the result of solving for $x$.

Putting that resulting equation back into $e$ gives us

$$a(-Ms + (b \mod M)y + by + a(c \mod M)) + c = 0$$
$$\Rightarrow -aMs + a(b \mod M)y + by + a(c \mod M) + c = 0$$

Call this equation $e''$.

**Claim:** There exist integer solutions to $e$ if and only if there exist integer solutions to $e''$.

**Observation 1:** $e$ has integer solutions if and only if $by + c$ is divisible by $a$. This is obvious if you solve $e$ for $x$.

**Observation 2:** $e''$ has integer solutions if and only if

$$a(b \mod M)y + by + a(c \mod M) + c$$

is divisible by both $a$ and $M$, which is similarly obvious if you solve for $x$.

Thus it is sufficient to show that

$by + c$ is divisible by $a$

if and only if

$$a(b \mod M)y + by + a(c \mod M) + c$$

is divisible by both $a$ and $m$.

**Fact I:** $a(b \mod M) + b$ is divisible by $m$ for any integers $a, b, M = a + 1$. The same is thus true for $a(c \mod M) + c$. I will prove the general case of this fact later on in this section.

**Fact II:** $a(b \mod M) + b$ and $a(c \mod M) + c$ are both divisible by $a$. This follows from the fact that $a \mod b$ answers "how much a is from being divisible by b". Of course, regular $mod$ has that same property, but $a \mod b$ gives you the smallest needed change, while $mod$ only give the smallest positive change needed.

**Fact III:** From Fact II and Observation I, we can conclude the following:

$$a(b \mod M) + by + a(c \mod M) + c$$

is divisible by $a$ if and only if $by + c$ is divisible by $a$.

**Conclusion:**

Notice that, by Facts I, II, and III, $by + c$ is divisible by $a$ if and only if $a(b \mod M) + a(c \mod M) + by + c$ is divisible by $a$ and $M$. Thus the claim is proven as soon as Fact I is proven.

**Proof of Fact I:** We need to show that $a(b \mod M) + b$ is divisible by $M$ for any integers $a, b, M = a + 1$.

This is equivalent to proving that

$$a(b \mod (a + 1)) + b$$

is an integer. Call that expression $F$.

**Fact IV:** By the definitions of $mod$ and $div$, we have that

$$x = x \mod z + z(x \div z) \Rightarrow yx = y(x \mod z) + yz(x \div z)$$

$$\Rightarrow y(x \mod z) = yx - yz(x \div z)$$
Recall the alternate definition of Modhat

\[
x \mod y := \begin{cases} x \mod y & \text{ (or)} \\
(x \mod y) - y & \text{whichever is closer to zero.}
\end{cases}
\]

Applying Fact IV to those two cases, we get the following:

**Case 1:** \( x \mod y = x \mod y \)

With \( x = b, y = a, z = a + 1 \), we have

\[
F = (a(b \mod (a + 1)) + b) / (a + 1)
= (ab - a(a + 1)(b \div (a + 1)) + b) / (a + 1)
= (ab + b - a(b \div (a + 1))(a + 1)) / (a + 1)
= (b(a + 1) - a(b \div (a + 1))(a + 1)) / (a + 1)
= b - a(b \div (a + 1))
\]

which is an integer minus an integer, and is thus itself an integer.

**Case 2:** \( x \mod y = x \mod y - y \)

With \( x = b, y = a, z = a + 1 \), we have

\[
F = (a(b \mod (a + 1) - (a + 1)) + ab + b) / (a + 1)
= (ab - a(a + 1)(b \div (a + 1)) + b - (a + 1)) / (a + 1)
= (ab + b - a(b \div (a + 1))(a + 1) - (a + 1)) / (a + 1)
= (b(a + 1) - a(b \div (a + 1))(a + 1) - (a + 1)) / (a + 1)
= b - a(b \div (a + 1)) - 1
\]

which is an integer.

Thus, in either case, \( \frac{a(b \mod (a+1)) + b}{(a+1)} \) is an integer. So Fact I is proven, meaning that the theorem as a whole is correct.

### 9 The Notion of a Consequence

An important notion when dealing with feasibility is that of a consequence. Given a system of constraints, \( E \), and a constraint \( c \), we say that \( c \) is a **consequence** of \( E \) when \( x \in E \land c \) if and only if \( x \in E \). That is, the restrictions imposed by \( c \) about feasibility are redundant given the effects of \( E \). The importance of information about consequences is that it means we can safely ignore \( c \), and not add it to \( E \).

The most common way of determining if \( c \) is a consequence of \( E \) is to determine if adding \( \neg c \) to \( E \) makes the system infeasible. Since we already have algorithms for analyzing satisfiability, consequence is often defined in those terms. Intuitively, removing such constraints is a very useful thing to do during Fourier's method since the worst cast constraint growth is a factor of the total number of constraints, not the number of non-redundant constraints. Removing the redundant ones won't change the feasibility of the system, but will drastically reduce the growth of the problem size. However, identifying consequences in general is unacceptable slow, \( O(2^n) \), so that notion alone won't help us improve the speed of Fourier's algorithm.

Fortunately, many constraints that are consequences of a system of constraints turn out to be **direct consequences**. A constraint \( c \in E \) is a direct consequence of \( E \) if and only if there is a single constraint \( c' \in E - c \) such that \( c' \) alone makes \( c \) redundant. That is, \( c' \lor \neg c \) is infeasible.

Essentially, when we look for direct consequences we are ignoring transitivity. That is, \( x \leq y \) is a direct consequence of \( x \leq y - 1 \), but \( x \leq y \) is not a direct consequence of the conjunction \( x \leq z \land z \leq y - 1 \).
Direct consequences come from parallel redundancy; that is, \( c_1 \) is a direct consequence of \( c_2 \) if the two are parallel and \( c_2 \) is at least as strict. That means that we can just check to see if the coefficients on each variable are equal. This makes identifying direct consequences fast, \( O(m) \), since we only need to compare the coefficients of \( c \) to those of \( m \) other constraints. Many tests use this fact to remove the obviously redundant information without taking the time to worry about the less common indirect consequences.

In fact, this test can be made even faster than \( O(m) \) through the use of a hash table. If we keep a hash table to store the coefficients of each equation in \( E \), and update it as we go, then we can determine if a new equation, \( c \), is a direct consequence in \( O(1) \) time - unless we get very unlucky with the coefficients values [Pug92].

Filtering out the direct consequences, as we will see in Section 11.7, is quite useful when working on the LI(2)-unit subdomain. It is the key to lowering the complexity bound for Nelson's restriction to Fourier from \( m^{2^n} \) to a mere \( mn^{10^n} \) [Nel78]. Intuition and details for the specific complexity are given in Section 11.7.

Pugh's extension of Fourier's method to Integer solutions also benefits greatly from efficient consequence testing. When we are restricted to the LI(2)-unit subdomain, there are many more redundant constraints produced. Because of this sort of "sensitivity" that LI(2)-unit has to parallel redundancy, even just identifying direct consequences is enough to entail asymptotic reductions in the time complexity of Fourier's Method [Pug92]. A more detailed discussion of this is given in Section 10.4.

A minimally unsatisfiable set, is an infeasible set of constraints which can be made feasible by removing any one of its elements. This notion is central to Nelson's satisfiability algorithm ([Nel78]), given in Section 11.2. He shows that certain subproblems are minimally unsatisfiable if and only if the entire problem is infeasible, thus permitting a more efficient form of spotting indirect consequences (discussed in Section 11.6).

More discussion and alternate approaches to dealing with consequences can be found in [Nel78], [PW98], and [Won95], but for the topics covered in this thesis, the above level of detail is sufficient.

10 Fourier's Test for Integer Solutions

Pugh has extended Fourier's method to be applicable to finding integer solutions [Pug92], which allows us to use it for dependence analysis. The process is overall the same, but there are a number of subtle mathematical adjustments that must be made to retain accuracy.

Now, in the case of dependence analysis, we are only dealing with integer solutions. This makes Fourier's method a little bit more complicated, although asymptotically the same (see Section 10.4). Fortunately, as we will, restricting the domain to unit coefficients can make the added complexity much less intimidating.

10.1 Complications for Integers

There are several complications that provide serious obstacles to extending Fourier's method to work for integer solutions. We need to make sure that, in each iteration, there are solutions in the new constraint system if and only if there are solutions in the previous system. We have to be careful about dealing with narrow constraints, and multiple projected integer regions.

1 narrow constraints: If the constraints "slip through" the integer lattice points, then the projection may have integer solutions even when the original set of constraints does not. See Figure 5 for an example of this. As we will see below, this is an example of a light shadow projection. Fortunately, if we have normalized the constraints (Section 7.1), then this problem is averted.
Figure 5: Constraints which have not been normalized may “slip through” the integer points and complicate matters. The arrows represent integer solutions in the projection which do not correspond to integer solutions in the original region.

(2) **multiple projected regions**: If the constraints are evil enough, then there may be several disjoint regions of integer solutions. That is, there may be a place in the projection where there are three integer solutions in a row, where only the outer two of them were in the original region. Unfortunately, this can happen even if the constraints have been normalized.

See Figure 6 for an example of this. This problem introduces disjunction to the system and requires that we (1) detect the split, and (2) analyze each of the regions separately. As we will see below, this is an example of disjoint integer solutions. Fortunately, if the inequalities all have unit coefficients, then this cannot happen - another benefit of the LI(2)-unit subdomain.

After finding the real shadow from one of the projections, we now have to divide that shadow into two regions: the **dark shadow** and the **light shadow**. Intuitively, imagine a light shining directly down onto the polytope being projected.

The light that passes through enough thickness is darkened more than the light that does not, thus creating an inner “dark shadow” (as shown in Figure 7).

Mathematically, this means that if there are integer solutions to the dark shadow, then there must be integer solutions to the polytope as a whole. For example, anywhere which is at least one unit thick is in the dark shadow; for any integer solution in the projection, there must be an integer value above in (along the last projected axis), thus there must have been an integer solution to the previous set of constraints. As it turns out, we can be more specific than that: based on system of constraints, we can actually put a more specific bound on the depth of the dark shadow [Pug92]. In Section 10.2, we will see that that value is $ab - a - b + 1$, where $a$ and $b$ are the slopes of two of the inequalities.

Also, we know that if there are no integer solutions to either shadow, then there are no solutions to the polytope as a whole. However, in the case of integer solutions to the light shadow, more work has to be done (Section 10.3).
Figure 6: This is an example where the dark shadow is disjoint. The arrows represent regions in the projected dark shadow, while the X marks an integer point which is not. Situations like these complicate the extension of Fourier’s method to integer solutions.

Figure 7: Visualize the light and dark shadows as more or less light passing through the polytope.
10.2 The Dark Shadow

I will now provide a lower level mathematical analysis of the dark shadow cast by the two arbitrary constraints discussed above [Pug92]. Given two constraints

\[ \alpha \geq az \land \beta \]

So we want to consider the case where there is an integer solution to \( a\beta \leq b\alpha \), but there is no integer solution to \( a\beta \leq abz \leq b\alpha \). That is, where there are solutions to the projected constraints, but there is not multiple of \( ab \) between \( a\beta \) and \( b\alpha \).

Recall that \( a \) and \( b \) are positive integers. Define \( i := \text{floor}[\beta/b] \). It is then straightforward from the given constraints that

\[ abi < a\beta \leq b\alpha < ab(i + 1) \]

That is, we can bound the ranges of the projected bounds in terms of \( i \). We know that

1. \( ab(i + 1) - b\alpha \geq b \), and
2. \( a\beta - abi \geq a \), and
3. \( ab(i + 1) - abi = ab \).

By adding (1) + (2), we get

\[ ab(i + 1) - b\alpha + a\beta - abi \geq b + a \]

Subtract that result from the third equations, i.e: (3) \( - (1) \) \( - (2) \),

\[ ab(i + 1) - abi - ab(i + 1) + b\alpha - a\beta + abi \leq ab - a - b \]

Which simplifies to

\[ b\alpha - a\beta \leq ab - a - b \]

We will also find it useful to factor the right hand side of that inequality:

\[ ab - a - b + 1 = (a - 1)(b - 1) \]

Let \( X := b\alpha - a\beta \) and \( Y := ab - a - b \) meaning \( Y + 1 = b - a - b + 1 = (a - 1)(b - a) \).

In those terms, what we know is that \( X \leq Y \). Therefore, if we are told that \( X \geq Y + 1 \), then there is forced to be of sufficient depth above the projected region to ensure an integer solution. Since the projected region contains integer solutions, we know that there must be an integer solution to \( z \) in the previous set of constraints.

Translating this back into the original variables, we get the following fact: If we are given that

\[ b\alpha - a\beta \geq (a - 1)(b - a) \]

then we know that there must be an integer solution to \( z \).

This means that the Dark Shadow of

\[ \alpha \geq az \land bz \geq \beta \]

is just

\[ b\alpha - a\beta \geq (a - 1)(b - 1) \]

Notice that if either \( a = 1 \) or \( b = 1 \) then the equations for the real and dark shadows are identical. That is, there is no light shadow to deal with, and the projection is called **Exact**. This happens when the coefficients of the equations in \( E \) are units, making the \( \text{L}(2) \)-unit domain an appealing one to study.
10.3 The Real Shadow

Recall that we know that one of the following three situations will be the case:

(a) If there are no integer solutions to the real shadow at all (dark or light), then there are no solutions to the original set of constraints.

(b) If there are any integer solutions to the dark shadow, then there are definitely solutions to the original set of constraints.

(c) Otherwise, there are no (integer) solution to the dark shadow, and we have to examine the real shadow. The fact that there were no “easy” solution (i.e. those in the dark shadow) will prove very useful. It means that if there is a solution, then that solution is tightly nestled in between the upper and lower bounds [Pug92]. In fact, it is in a region between the upper and lower bounds separated by at least one unit of depth, otherwise the dark shadow test would have detected it. In fact, we can get a tighter bound than just “one unit of depth”, as we will see in the formal proof below. I will give a proof of that assertion in the following two subsections.

We consider a set of planes that are parallel to a lower bound and close to a lower bound. Any solution closely nestled between an upper bound and a lower bound must lie of one of these planes. I will now present the computational approach as described by Pugh in [Pug92].

At this point in the algorithm, if there is an integer solution to the original set of constraints, then there must exist a pair of constraints on $z$ of the form

$$\alpha \geq az \wedge bz \geq \beta$$

such that

$$ab - a - b \geq b\alpha - a\beta \geq 0 \wedge$$

$$b\alpha \geq abz \geq a\beta$$

That is to say, they are tight upper and lower bounds who are close enough to hide a solution. Specifically, they must be $ab - a - b + 1$ or less apart in order to contain an integer solutions but still fall within the real shadow. We thus get

$$ab - a - b + a\beta \geq abz \geq a\beta$$

from transitivity of the first two statements.

The only way to solve this is to determine the largest coefficient of $z$ in any upper bound on $z$ (i.e. $a$) and the largest coefficient $z$ in any lower bound of $z$ (i.e. $b$). Then test if there are integer solutions to the original problem plus $bz = \beta + i$ for each $i$ such that $(ab - a - b)/a \geq i \geq 0$.

This is a very expensive test, but it is necessary in the worst case. Fortunately, most of the time you won’t have to repeat the process too many times before either finding solutions in the dark shadow or showing that there are none in the entire shadow (options $a$ and $b$ from above). Thus, in this manner we can usually avoid the $m^2$ steps that the worst case presents us with.

In the spirit of Fourier’s test, we have eliminated a variable ($z$) from the system of equations but preserved feasibility.

29
10.4 Improved Complexity of Fourier’s Method

When running Fourier’s Method over the integers, the additional calculations slow down the process, but the asymptotic complexity doesn’t change. However, a lot of that added complexity can be avoided if we are working on the LI(2)-unit subdomain, since there will be no light shadow.

The quadratic growth of the number of constraints after each iteration of Fourier’s Method is not as bad or as intimidating as it appears when we take into account consequences (recall Section 9). After each step, it is likely that many of the constraints will be redundant, lowering the growth rate of the problem size. When solving in general, the number of the produced constraints which are redundant might manage to stay low, but if we are working in LI(2)-unit over the integers, then even the number of direct consequences will be significant. Intuitively, we can see why this is the case by the fact that there are only 8 possible non-redundant inequalities in any given plane. Of course, since LI(2)-unit guarantees a density of 2 (by definition), each such constraint is planar (in the plane formed by its two variables). Because of this “sensitivity” that LI(2)-unit has to parallel redundancy, even just identifying direct consequences is enough to entail asymptotic reductions in the time complexity [Pug92]. If we keep a hash table to store the coefficients of each equations in $E$, and update it as we go, then we can determine if a new equation $c$ is a direct consequence in $O(1)$ time - unless we get very unlucky with the coefficients in the system [Pug92].

That fact, plus the absence of a light shadow (recall Section 10.1), results in lowering the asymptotic bound for Fourier’s Test from $O(m^{2^r})$ to $O(n^3)$ [Pug92].

11 Nelson’s Approach

Another subdomain that algorithms sometimes restrict themselves to is that of monotone constraints. Recall from Section 1.1 that a monotone constraint is an LI(2) constraint for which the signs on the two variables are different when written in standard form. That is, they are of the form $ax - by \pm c$ for positive constants $a$, $b$, and $c$.

Nelson provides an algorithm which solves systems of monotone constraints in pseudo-polynomial time roughly proportional to $n^{\log n}$. His algorithm works by considering the chains of constraints that result from monotone constraints. For instance, $x < y$ and $y < z$ form the chain $x < y < z$ from which $x < z$ is a consequence. Nelson shows that we can identify inconsistent chains quickly, and that doing so for specifically chosen subproblems is sufficient to determine feasibility as a whole.

In Subsection 11.1, I discuss Nelson’s rephrasing on Fourier’s method, as a means to introduce some of his terminology and general manner of viewing the problem. This will also serve as a review of Fourier’s Method, given previously in Section 5.1, and as a different way to view the mechanics of that algorithm.

In Subsection 11.2, I present Nelson’s algorithm explicitly, and give a brief overview of the approaches he uses to prove its accuracy and time complexity. I then give a slightly more thorough overview of his proofs in Subsections 11.4 through 11.7. Nelson’s own paper already provides beautifully written descriptions and proofs, so I have not felt compelled to retype an already well written explanation here. I will instead provide a flavor of the clever techniques he uses and a general intuition for their accuracy. Interested readers should consult [Nel78] for the complete proofs.

11.1 Rephrasing Fourier’s Method

First we will consider Nelson’s rephrasing of Fourier’s method, which will also serve as a review of Fourier’s Method and as a different way of looking at it.

Definition of terms:
Let $S$ be a conjunction of linear inequalities.

Let $S_x$ be the conjunction of equations in $S$ which contain $x$.

Let $T_x$ be the conjunction of equations in $S$ which do not contain $x$.

Let $W_x$ be the conjunction of all $x$-resultants of pairs of constraints of $S$.

So, we have that $S = S_x \cup T_x$ and $S_x \cap T_x = \emptyset$.

Using that notation, we get the following three important facts.

I: Each constraint in a system of inequalities gives you either an upper or a lower bound on each variable, but never both. In either case, that bound is unique; changing the form of the equation won't affect what the corresponding bounds are.

II: For any chosen variable $x$, we can define an $x$-resultant $L \leq U$ for each lower bound $L \leq x$ and each upper bound $x \leq U$. The $x$-resultant does not contain $x$, but the union of all of $x$-resultants has the same effect on feasibility the original bounds did.

III: $S = S_x \land T_x$ is satisfiable if and only if $W_x \land T_x$ is satisfiable. That is to say, $W_x$ (the $x$-resultants) and $S_x$ (the equations with $x$) are equivalent when conjoined with $T_x$ (the equations without $x$). Each step of Fourier's Method preserves feasibility.

Fourier's Method is just a matter of applying Fact II once per variable. Fact I ensures that manipulating the equations in order to solve for the resultants does not alter feasibility. Fact III ensures that the (trivial) system of equations that results from eliminating all the variables in that manner has the same feasibility as the original one did.

11.2 Nelson's Satisfiability Algorithm

In the original statement of Fourier's Method, variables are selected and removed one at a time. The algorithm given below (directly taken from [Nel78]) is very similar to that approach, however, instead of forming resultants from one variable at each step, the algorithm forms resultants for each variable at every step. By doing so, the algorithm takes advantage of the lower amortized complexity of Fourier's test over monotone constraints - a more detailed explanation of this is given is Section 11.7. For now, consider the details of Nelson's satisfiability algorithm:

```plaintext
for i = 1 to ceiling(lg n) - 1
    let $S'$ be the set of all resultants of pairs of elements of $S$, then replace $S$ by $S'$ union $S$;
    for 1 <= i <= k <= n
        if $S_{-\{v_j,v_k\}}$ is unsatisfiable
            then return "unsatisfiable";
    filter $S$;
    return "satisfiable";
```

Given a set $S$ of LI(2) constraints among $n$ variables $v_1, v_2, ..., v_n$, this algorithm determines their satisfiability (feasibility).
11.3 Overview of Proof of Correctness

The algorithm is obviously correct if answers "unsatisfiable", since it does so only when Fact III (from Section 11.1) is not satisfied. To show that the algorithm is correct when it returns "satisfiable" takes a little more work. Specifically, we need to show explicitly that $\log(n) - 1$ iterations is enough to produce some $S_{v_1, v_k}$ unsatisfiable. So, what we need to show is that the algorithm will not need more than the $\log(n)$ steps it performs to identify a contradiction in any unsatisfiable set.

The basic approach to showing that $\log(n)$ steps are sufficient revolves around the interplay between the following two notions:

1. Helly's Theorem and some related facts about convex regions, and
2. Facts about chains of constraints and minimal unsatisfiability.

Helly's theorem will give us some facts about the intersections of convex regions, which is applicable since each constraint defines a half space (a convex region); thus the solutions to a system of inequalities is an intersection of convex regions. The notion of a chain of constraints will allow a quick test which will not always identify unsatisfiable cases, but it *will* do so in certain cases that Nelson's algorithm guarantees will occur [Nel78]. However, after applying the results of Helly's Theorem, we will see how to modify the chain test to still be asymptotically faster than Fourier but still return a completely accurate value.

11.4 Helly's Theorem

Helly gives us the following famous theorem:

**Helly's Theorem:** Let $F$ be a finite family of convex sets in $d$-dimensional space. Consider each subfamily of $d+1$ sets in $F$ - if all those subfamilies share a common point, then there is a point in common to all the sets in $F$.

Another way of phrasing this, is the following: If a convex region $R$ intersects $d+1$ mutually intersecting convex regions in $d$-dimensions, then $R$ is guaranteed to also intersect their mutual intersection.

In two dimensions ($d = 2$), this is geometrically straightforward. In Figure 8-a, we can see that if a fourth convex region intersects the three $(d + 1 = 3)$ regions $A \cup B$, $B \cup C$, and $C \cup A$, then that regions must intersect $A \cup B \cup C$. In Figure 8-b, we see an example where Helly's theorem does not apply here, since $d = 2$ but the $d + 1 = 3$ convex shapes depicted do not have a common intersection.

The importance of Helly's Theorem to this approach is the observation that each monotone constraint defines a half space which is (obviously convex). Thus, any minimally unsatisfiable set of constraints in $S$ will have at most $n + 1$ equations in it. That is, if $S$ is unsatisfiable, then some $n + 1$ sized subset of it is unsatisfiable. This fact will allow us to determining feasibility by checking only subsets of a certain size, rather than all possible subsets.

11.5 Chains of Inequalities

Nelson's approach depends heavily upon the notion of a chain of constraints. Intuitively, the monotone constraints

$$x \leq y \quad \text{and} \quad y \leq z$$

can be made into the chain $x \leq y \leq z$. The important fact is that the chain shortens to some equation $x \leq z$. If this shortened chain is inconsistent, then we know that the set of constraints containing the full
Figure 8: Figure (a) shows a simple two dimensional example of Helly’s Theorem. If a fourth convex regions intersected each of the existing three, then it would have to intersect their common point too. Figure (b) shows a simple example where it does not apply, since there exist three regions in two dimensions which with an empty mutual intersection.

A chain is not satisfiable. The clever thing about this method, is that we actually catch certain kinds of indirect consequences, allowing us to identify inconsistent chains with enough accuracy to spot infeasible system.

I will now give a more mathematically formal definition of a chain, and a means of classifying them based on overall shape. A sequence of two variable constraints $C_0, C_1, ..., C_n$ is a chain with internal variables $v_1, v_2, ..., v_n$ means the following:

For each $i$ (such that $1 \leq i \leq n$), $C_i - 1$ and $C_i$ have a $v_i$-resultant. Furthermore, $v_i$ does not occur in any of the other $C$’s. For example,

$$
\begin{align*}
5 &\leq x \\
 x &\leq y \\
y &\leq z + 2 \\
z &\leq 10
\end{align*}
$$

form a chain with internal variables $x$, $y$, and $z$.

$C_0$ may contain another variable besides $v_1$, in which case that variable is called the left end variable. The right end variable is defined analogously. If the right and left end variables are the same, then the chain is circular. Note that the end variables are not considered to be internal to the chain, and thus it is not a violation of the definition for them to be the same variable. Two chains intersect, if they contain a common internal variable.

Perhaps the most useful fact in proving thing about chains, is the following: The constraints outnumber the internal variables by exactly 1 in any chain. This fact will allow us to reduce the number of structures that are chains to 7, only 6 of which are non-trivial.

1. trivial case: $V$ is empty, and $S$ contains one constraint.
2. line case: There is a chain with no end variables which contains all the constraints in $S$.
3. circle case: There is a circular chain which contains all the constraints in $S$. 

33
Figure 9: These are structural diagrams of the 7 possible configurations of chains of constraints. The dots represent end variables, and the lines represent chains of inequalities.

Figure 10: These diagrams represent structures in which the inequality constraints do not outnumber the variables by exactly one, and thus are not possible configurations for chains.

(4) circle-lines case: $S$ can be partitioned into two non-intersecting chains - one with the end variable $u$, the other with $u$ as its only end variable.

(5) theta case: $S$ can be partitioned into three pairwise non-intersecting, non-circular chains - each of which have the same pair of end variables.

(6) eyeglasses case: $S$ can be partitioned into three pairwise, non-intersecting chains - two of which are circular chains with distinct end variables $u$ and $v$, and the third of which has $u$ and $v$ as end variables.

(7) figure eight case: $S$ can be partitioned into two non-intersecting circular chains which have a common end variable.

Each of these can be visualized diagrammatically, as depicted in Figure 9. The lines represent chains of constraints and the dots represent non-internal variables (end variables).

Proving that these structures, and no other similar structures, are chains requires the use of the fact that the number of constraints outnumbers the number of variables by exactly one. All other structures, such as
those depicted is figure 10, fail this criteria. Nelson proves this fact by examining a large number of cases of possible configurations, one by one, and proving that each fails the criteria of being a chain [Nel78].

11.6 Completing the Proof

To deal with the 6 non-trivial cases of chains, we will use the following lemma:

**Lemma:** Let $S$ be an unsatisfiable set of two-variable constraints, every proper subset of which is satisfiable. That is, $S$ is minimally unsatisfiable (recall Section 9). Let $V$ be the set of all variables appearing in constraints of $S$. Let $k$ be the number of variables in $V$. Let $S'$ be the set of all resultants of pairs of constraints in $S$.

**Claim:** There exists a subset $V'$ of $V$, containing no more than $\lfloor k/2 \rfloor + 1$ variable, such that $(S \cup S')_{V'}$ is unsatisfiable.

Intuitively, this lemma follows from the fact that, in an LI(2) domain (and even more so in a monotone domain) we are guaranteed to get long chains of constraints [Nel78].

Consider the facts that we have gathered:

1. The above lemma tells us that, for any unsatisfiable system of constraints $E$, there is a size $n + 1$ subset of $E$ which is unsatisfiable.
2. We can collapse a chain in logarithmic time, by halving the number of entries in it each step. This is done by taking resultant simultaneously, and lets us quickly identify if a chain is inconsistent.
3. Clearly, finding an inconsistent chain in any subset of a system of constraints proves that that system is unsatisfiable.

These boil down to saying the following: While not every unsatisfiable system contains an inconsistent chain, every unsatisfiable system has a minimally unsatisfiable subset with an inconsistent chain [Nel78]. Consequently, we can just look at the size $n + 1$ subsets and look for inconsistent chains in each of them. If we do find one, then the system is clearly unsatisfiable. If we fail to find one, then these facts give use that the system as a whole is satisfiable.

11.7 Time Complexity

Recall from Section 5.3 that the two worst cases of the ways in which Fourier's method can get complex cannot coincide. You cannot both square the number of constraints, and fail to remove any. Intuitively, Nelson's Algorithm takes advantage of that by lumping all of the complexity together into one step. In the full test, the amortized complexity only lowered the naive measurement to a better exponential, but on the monotone domain we actually lower it to sub-exponential time.

Specifically, on a monotone domain we know that $n \leq 2m$, since we are on a subset of the LI(2) domain. That means that if one variable appears in a lot of the equations, then the other variables necessarily occur in fewer of them. So, if calculating resultants is bad for one variable in a given system, then it is necessarily not so bad for the other variables in that system. Also, the more equations one variable appears is (the number we end up squaring at worst), the fewer total number of variables there can be in the system (the fewer steps it will take total). By calculating all of the resultants before altering the system of equations, we can take full advantage of those two facts. Specifically, this approach means that $\log(n)$ steps are needed to eliminate $n$ variables (instead of $n$ steps), and that those steps do not all square the number of constraints. As Nelson observes, the algorithm essentially removes the variables in parallel instead of in serial [Nel78].
As a result, we get the rough bound to be

\[ O(n^{lg} n) \]

and, with some fairly lengthy work, we can get the exact time bound to be

\[ O(m(n^{ceil(\lg n)}+3lg n) \]

Which is pseudo-polynomial time [Ne78]; it runs slower (asymptotically) than any polynomial, but faster than any exponential.

12 Identifying Fat Polytopes

A fast preprocessing stage that can be used before slower methods to identify easy cases of integer solutions is to identify fat polytopes [HN94]. If we can show that there is a unit hypercube contained within the real solutions, then we know that there is an integer solution within that region too. This is because every unit hypercube contains at least one integer point. Furthermore, if there is a hyper-sphere entirely within the region of real solutions which circumscribes a unit hypercube, then we know there is a unit hyper-cube within the real solutions. Consequently, the system would be feasible over the integers. So, we wish to find a contained sphere with radius \( r = \sqrt{k}/2 \) for \( k \) dimensions.

Consider what happens when every constraint is collapsed “inwards” by \( r \) - that is, in the direction which is indicated by the half-space it defines. Now, any real solution to the new set of constraints is the center of a sphere of radius \( r \) contained entirely within the original system, since it is at least \( r \) units away from each constraint’s original position. Therefore, finding real solutions in the collapsed set implies that there are integer solutions in the original set of constraints. This is convenient, since there exist fast solutions for feasibility over the reals. Furthermore, notice that if we round the real solution we found to the nearest integer point, then we will have found an integer solution to the original system. See Figure 11 for a 2-dimensional visualization of this process. Of course, not finding one does not mean anything, and we would then have to run a slower test.

Formally, shifting is accomplished by taking any constraint with a constant shift of \( c \) and generating \( \bar{c} \) as follows:

\[ \bar{c} = c - \frac{r}{\sqrt{\sum_{i=1}^{n} a_i^2}} \sum_{i=1}^{n} a_i \]

In the case of LI(2), the sums include at most 2 terms. The new set of inequality constraints, each with some constant \( c \) substituted by \( \bar{c} \), is also LI(2). Thus we can still run tests which work on LI(2) over the reals, and come up with an answer for the question of integer feasibility. Hochbaum and Naor show that this technique determines if there is a fat polytope or not in \( O(mn^2 lg m) \) time [HN94].

13 Subtle Approaches to Disequalities

Unfortunately, current techniques for dealing with disequalities are exponential [PW98], even for LI(2)-unit, thus feasibility testing is also exponential as a whole. Recall that, in the naive algorithm, we rewrite each disequality \( \alpha \neq \beta \) as a disjunction of two subproblems, each with one of the inequalities \( \alpha > \beta \lor \alpha < \beta \). This approach is clearly exponential in the number of disequalities. Without a way to avoid the exponential algorithm entirely, we can still hope to
Figure 11: (a) We are faced with a set of 5 inequality constraints in $\mathbb{R}^2$. (b) The constraints are shifted inward by $r$. (c) We identify a real solution to the collapsed system (solid dot). (d) Consider the relation of that solution to the original region. (d) We know that the original region contained a circle of radius $r$, centered at the solution. (f) Thus there is a unit square within the original region, which must contain an integer point (open dot). Furthermore, we can find that solution will be the nearest integer point to the center of the circle.
Figure 12: Case (a) shows a case where the inequalities define a region in which all disequalities are inert. Intuitively, a finite number of disequalities cannot eliminate all of the solutions of a region that is "splayed open" like this one is. Case (b) shows a case where the inequalities define a closed region, and all disequalities are ert. There could always just be one disequality per solutions point, thus a finite number can easily eliminate all of the solutions. Case (c) shows a case in which disequalities which are parallel to the two inequalities are inert, but those not parallel are inert. That is, a finite number of parallel disequalities can catch all of the solutions, but other types of disequalities cannot ever contribute to feasibility.

(a) Recognize the cases where we can avoid running the full exponential algorithm and not waste our time on it, or at least

(b) Reduce the size of the exponent when we do have to run the brute force algorithm.

In this section we provide an approach to catch some of the cases where we can get significant speed up by avoiding the exponential algorithm without sacrificing accuracy. In Subsection 13.1, I introduce the notions of inert and ert disequalities. In Subsection 13.2, I will give a quick conservative test to perform that classification, and provide proofs for its accuracy. While we do not yet have a test which will perform a complete classification, the test does catch the cases which (1) allow us to avoid the exponential algorithm and (2) our experience implies are commonly occurring in practice.

13.1 Inert (and Ert) Disequalities

Lassez [LM89] observed that, for constraints on real variables, disequalities are independent. That is, if no one disequality eliminates all solutions, there is no way for a finite number of disequalities to add up and together make the system unsatisfiable.

Unfortunately, it is in general possible for disequalities to add up for constraint systems with integer variables. Here we identify disequalities which cannot add up despite our use of integer variables.

In practice, disequalities usually do not add up, so a lot of time is wasted examining an exponential number of subcases. Thus there is need for a way to identify when the disequalities cannot add up, so that we can avoid unnecessary slowdown.

Formally, given a conjunction of integer inequalities $C$ and a disequality $d_0$, we say $d_0$ is inert in $C$ if and only if, for any finite set of disequalities $\{d_1, ..., d_k\}$, $d_0 \land C \land d_1 \land ... \land d_k$ is satisfiable $\iff$ both $d_0 \land C$ and $C \land d_1 \land ... \land d_k$ are satisfiable. Otherwise, we say that $d_0$ is ert in $C$. 38
Once the disequalities are classified in this manner, we can test the inert ones as follows: For each inert \(d_i\), test if \(d_i \land C\) is satisfiable. We do this by testing to see if either \(d_i^{U} \land C\) or \(d_i^{L} \land C\) is satisfiable, where \(d_i^{U}\) is \(d_i\) converted into an inequality upper bound, and \(d_i^{L}\) is \(d_i\) converted into an inequality lower bound. If either of those is satisfiable, then we know that \((d_i \land C)\) is satisfiable. If any of these disjunctions of tests produce false, then the original formula is infeasible. Otherwise, the full exponential algorithm must be run on the remaining (ert) disequalities. Even when exponential testing is still required, we will have reduced the exponent by the number of inert disequalities. For \(i\) inert disequalities and \(e\) ert disequalities, this reduces the number of satisfiability tests from \(2^{i+e}\) to \(2i + 2^e\), a factor of about \(2^i\).

**Claim:** \(d_0\) is inert in \(C \iff\) there exists a ray \(R\) in \(Z^n\) such that \(R\) is not parallel to \(d_0\) and there exist an infinite number of solutions in \(C\) along \(R\) (that is, there are an infinite number of solutions to \(R \land C\)). We conjecture that such a ray can be found in polynomial time if we are on the LI(2)-unit subdomain, and possibly also in general.

So the problem of determining if a disequality is inert has been reduced to identifying such rays. Of course, if we fail to find such a ray quickly, we can always just assume that none exist and perform the exponential algorithm. We do not yet have a complete general test, however I will present a quick, conservative test which our experience suggests will catch the common cases [SW00b].

### 13.2 A Quick Test for Inertness

We are given a disequality \(d\) of the form \(a_1x_1 + \ldots + a_n x_n \neq 0\) to classify as inert or ert. We perform the test in several stages.

**A.** If there are two parallel inequalities, one on each side of \(d\), then \(d\) is not inert. Mathematically, we look for a pair of inequalities \(a_1x_1 + \ldots + a_n x_n \leq c\) and \(a_1x_1 + \ldots + a_n x_n \geq k\) with \(k < c\). If they exist in the system of equations, then \(d\) is ert. Otherwise, we proceed to stage B. Recall that by using a hash table as described by Pugh, checking for parallel redundancy can be done in essentially constant time [Pug92].

**B.** If each variable in \(d\) is bounded above and below by constants, then \(d\) is ert. Otherwise, we proceed to stage C.

**C.** If any variable in \(d\) is unbounded, either above or below, then \(d\) is inert. If none of the variables are unbounded, then we examine each variable \(v\) in \(d\) and check if it is bounded in one direction by a single variable \(v'\). If \(v'\) is unbounded in the same direction then \(v\) is unbounded in that direction. If \(v'\) is bounded in the same direction by only a single variable \(v''\), then repeat the process until we find either a constant bound in that direction, a multi-variable bound in that direction, a variable previously encountered in the same chain, or unbounded. Notice that we will not ever be stopped by a multi-variable bound if we are working in LI(2).

**D.** If none of these classify the disequality, then we conservatively approximate it as ert. Of course, if only one equation is classified as ert, then it is classified as inert - there are no other ert disequalities for it to add up with.

**Proof of A:** In order to show that \(d\) is ert under a given criteria, it is only necessary to find an example.

Consider the system of equations depicted in Figure 13, \(x \geq 1 \land x \leq 4\). None of the disequalities \(x \neq 1 \land x \neq 2 \land x \neq 3 \land x \neq 4\) eliminate the solutions individually, but collectively they eliminate all solutions.
Proof of B: If each variable is bounded above and below, then the region in which the disequality lies could be bounded. A bounded region has only a finite number of integer solutions, so any inequality can add up, meaning \( d \) is ert.

Proof of C: If some variable \( u \) in \( d \) is directly unbounded, consider some solution \( s \) to the inequalities. Let \( v \) be the unit vector parallel to the \( u \) axis. We can construct a ray \( \vec{r} \) with initial point \( s \) which is parallel to the \( u \). \( \vec{r} \) has an integer solutions in \( C \) since \( u \) is unbounded, \( \vec{r} \) begins on a solutions point, and \( \vec{r} \) as a rational slope. \( \vec{r} \) cannot be parallel to \( d \) since the other variables are non-zero in \( d \) but zero in \( \vec{r} \).

If \( u \) is indirectly unbounded by way of other variables, then we will construct \( \vec{r} \) in a different manner. \( \vec{r} \) will still have an initial point at \( s \), but its slope will be harder to construct. Without loss of generality, let \( B = \{v = x_0, x_1, \ldots, x_p\} \) be the set of variables that we had to search through to find out that \( u \) was indirectly bounded. That is, \( \forall i \) such that \( 1 \leq i \leq p - 2 \), there is a constraint \( a_i x_i \leq a_{i+1} x_{i+1} \). We will construct \( \vec{r} \) as an \( n \)-dimensional vector in the following manner: if \( x_j \) is not in \( B \), then the \( j^{th} \) entry in the array is 0. If \( x_j \) is in \( B \), then the \( j^{th} \) entry will be \( \frac{a_{j+1}}{a_j} \). This construction ensures that \( \vec{r} \) is parallel to each of the relevant constraints. As before, \( \vec{r} \) has an integer solutions in \( C \) since \( u \) is unbounded, \( \vec{r} \) begins on a solutions point, and \( \vec{r} \) as a rational slope. \( \vec{r} \) cannot be parallel to \( d \) by construction; if it were.
Consider the following example where we apply the quick test:

\[
\begin{align*}
1 & \leq i \leq N \\
1 & \leq N \\
0 & \leq j \leq 7 \\
0 & \leq i + j \leq 10 \\
N & \text{ unbounded above} \\
M & \leq N + 1
\end{align*}
\]

Say we are classifying the disequalities

\[
\begin{align*}
i+j & \neq 0 \quad (d_0) \\
i-j & \neq 0 \quad (d_1) \\
i-j-x+y & \neq 0 \quad (d_2)
\end{align*}
\]

which I will refer to as $d_0$, $d_1$, and $d_2$ respectively.

- $d_0$ is classified as ert (by stage A) since it is parallel to two parallel inequalities.
- $d_1$ is classified as ert by default (incorrectly, but without loss of accuracy, just time).
- $d_2$ is like $d_0$, except that it contains the extraneous variables $x$ and $y$. Since these variables are not mentioned in any of the inequality constraints, they are unbounded and thus make the inequality inert (stage C).

Thus, the inert equation can be analyzed independently, but the two ert equations will have to be handled with the full exponential algorithm (in 4 subcases).

**Complexity (of Each Disequality Test):** Let $m$ be the number of inequalities. For a given disequality, $d$, let $z$ be the density of $d$ (the number of variables in $d$). Of course, $s$ is bounded above by $n$, the number of variables in the system.

A. Stage A takes $O(m)$ time to scan all of the inequalities to check if they are scalar shifts of $d$. With a hash table (like the one Pugh uses [Pug92]), this test can be done for LI(2)-unit constraints in essentially $O(1)$ time.

B. Stage B takes $O(mz)$ time to check each inequality against each variable in $d$.

C. Stage C requires $O(m^2s)$ time if implemented naively, since it potentially scans through all the inequalities and through all of them again at each inequality. However, with basic memoization to record if a variable is eventually unbounded, the complexity for stage C comes down to $O(mz)$ time. The information for stages A and B is very similar, so we can gather it all at once - which entails only $O(mz)$ time for both tests.

D. Stage D takes $O(1)$ time during the quick test, but will entail $O(2^e)$ time, for $e$ ert disequalities, when the full test is later run.

Therefore, we have a fast, conservative preprocessing test. When more empirical data has been gathered, we will be able to determine if the quick test does indeed catch the common cases, as intuition suggests. Also, we intend to derive a complete test for the general case, and an efficient complete test for the LI(2)-unit subdomain.
14 Conclusion

In this thesis, I have discussed the meaning of dependence analysis, the problems it presents, and solutions of varying degrees of subtleties and style. I have shown the mathematics behind using Fourier-Motzkin elimination, as well as the added mathematical complexity of Pugh's Extension to that method [Pug92]. I have given the key intuitions an overview of the proofs for Nelson's algorithm, which shows a different approach to making Fourier's Method more efficient [Nel78]. I have also given descriptions, proofs, and intuitions for a number of other relevant methods, notions, and techniques [HN94]. Each of these methods takes a different approach at how to avoid the exponential general case analysis that integer programming entails. In a number of places, I have also pointed to related work for interested readers to pursue and/or more detailed work on the subject. Furthermore, I have described the difficulties presented by disequality analysis, and have presented a fast, conservative preprocessing algorithm to reducing the exponential behavior during such analysis. I have pointed to future work in that area, both in the form of mathematical analysis and in the form of further empirical data.

15 Key Words

Array Dataflow Analysis, Constraint Manipulation, Data Dependence Analysis, Disequalities, Ent, Integer Feasibility/Solvability, Fourier, Fourier-Motzkin, Monotone Constraints, Omega Test, Parallelization, Presburger Arithmetic, Fourier, Inert, LI(2)-unit.

References


