

# Investigating the Non-termination of Affine Loops

A THESIS PRESENTED IN PARTIAL FULFILMENT  
OF THE REQUIREMENTS FOR THE DEGREE OF  
MASTER OF SCIENCE  
AT THE UNIVERSITY OF STELLENBOSCH



By  
K. Durant  
October 2012

Supervised by: Prof. W. Visser

# Declaration

I the undersigned hereby declare that the work contained in this thesis is my own original work and has not previously in its entirety or in part been submitted at any university for a degree.

Signature: .....

Date: .....

# Summary

The search for non-terminating paths within a program is a crucial part of software verification, as the detection of an infinite path is often the only manner of falsifying program termination — the failure of a termination prover to verify termination does not necessarily imply that a program is non-terminating. This document describes the development and implementation of two focussed techniques for investigating the non-termination of affine loops. The developed techniques depend on the known non-termination concepts of recurrent sets and Jordan matrix decomposition respectively, and imply the decidability of single-variable and cyclic affine loops. Furthermore, the techniques prove to be practically capable methods for both the location of non-terminating paths, as well as the generation of preconditions for non-termination.

# Afrikaans summary

Sagtewareverifikasie vereis of die bewys van die beëindiging van 'n program, of die deteksie van oneindige uitvoerings. In hierdie tesis ontwikkel en implementeer ons twee tegnieke om oor die oneindige eienskap van affiene lusse te beslis. Die tegnieke wat ontwikkel word is gebaseer op konsepte soos Jordan matriksdekomposisie en herhaalde groepe wat al in die verlede gebruik is om die beëindiging van lusse te ondersoek. Die tegnieke kan gebruik word om die uitvoerbaarheid van beide een-veranderlike en sikliese affiene lusse te bepaal. Feitlik alle nie-eindige affiene lusse kan geïdentifiseer word en die toestande waaronder hierdie oneindige eienskap verskyn kan beskryf word.

# Acknowledgements

I would like to thank:

- Prof. Willem Visser, for providing me with both the opportunity and supervision to perform this work; and
- Prof. Stephan Wagner, for his innate ability to produce apt counter-examples.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Document outline . . . . .	3
<b>2</b>	<b>Background</b>	<b>4</b>
2.1	A brief review of software verification and falsification . . . . .	4
2.1.1	Safety and liveness properties . . . . .	5
2.1.2	State representation . . . . .	5
2.1.3	Invariants . . . . .	6
2.2	Problem description . . . . .	6
2.3	Related work . . . . .	8
2.3.1	Decidability . . . . .	8
2.3.2	Termination verification . . . . .	8
2.3.3	Conditional termination . . . . .	9
2.3.4	Termination falsification . . . . .	9
<b>3</b>	<b>Approach</b>	<b>11</b>
3.1	Affine loops . . . . .	11
3.2	Non-termination via recurrent sets . . . . .	14
3.2.1	Single-variable affine loops . . . . .	19
3.2.2	Cyclic affine loops . . . . .	21
3.2.3	Termination verification via recurrent sets . . . . .	23
3.2.4	Non-linear loops . . . . .	24
3.3	Non-termination via Jordan decomposition . . . . .	25
3.3.1	Diagonalisation . . . . .	26

3.3.2	Jordan decomposition . . . . .	29
3.3.3	Sums of exponential functions . . . . .	36
3.3.4	Proving non-termination . . . . .	42
3.3.5	Approximating the set of non-termination witnesses . . . . .	49
3.3.6	Constraining polynomials . . . . .	62
3.3.7	Deciding termination for simple affine loops . . . . .	64
3.3.8	Termination verification via sign permutations . . . . .	65
3.4	Summary . . . . .	67
<b>4</b>	<b>Implementation</b>	<b>68</b>
4.1	Detecting loops . . . . .	69
4.1.1	Detecting loop boundaries . . . . .	69
4.1.2	Constructing affine loops . . . . .	72
4.2	Non-termination algorithms . . . . .	75
4.3	Non-termination via recurrent sets . . . . .	78
4.4	Non-termination via Jordan decomposition . . . . .	80
4.5	Algorithm complexity . . . . .	84
<b>5</b>	<b>Evaluation</b>	<b>86</b>
5.1	Non-termination detection . . . . .	87
5.2	Conditional non-termination . . . . .	93
5.3	Summary . . . . .	96
<b>6</b>	<b>Conclusion</b>	<b>98</b>
6.1	Further work . . . . .	99
6.1.1	Termination verification and conditional termination . . . . .	99
6.1.2	Complex loop forms . . . . .	100
6.1.3	Test case generation . . . . .	100
<b>A</b>	<b>Supporting Concepts</b>	<b>101</b>
A.1	Simple program representation . . . . .	101
A.2	Complex arithmetic . . . . .	102
A.3	Complex eigenvalues and Lemma 4 . . . . .	102

<b>B Implementation Notes</b>	<b>105</b>
B.1 Algorithms . . . . .	105
B.2 Class structure . . . . .	107
B.3 Example loops . . . . .	108
<b>Bibliography</b>	<b>112</b>



# List of Tables

3.1	The maximal periods $K(n)$ of cyclic loops with few loop variables. . . . .	23
3.2	Several values of the function in Figure 3.16 at positive integer intervals. . . . .	41

# List of Figures

3.1	An unnested program loop. . . . .	12
3.2	The general form of an affine loop. . . . .	13
3.3	A periodically monotonic loop. . . . .	18
3.4	A general single-variable loop. . . . .	19
3.5	A non-terminating single-variable loop. . . . .	20
3.6	A loop which is not periodically monotonic. . . . .	21
3.7	A 2-cyclic non-terminating loop. . . . .	23
3.8	A 2-cyclic terminating loop. . . . .	24
3.9	A non-terminating, non-cyclic loop. . . . .	24
3.10	A loop for which termination can easily be verified. . . . .	24
3.11	A diagonalisable two-variable (excluding the auxiliary variable $x'$ ) loop. . . . .	28
3.12	A loop which is not diagonalisable. . . . .	34
3.13	A sampling of exponential functions. . . . .	36
3.14	Exponential sums with two positive coefficients. . . . .	37
3.15	Exponential sums with mixed (sign) coefficients. . . . .	38
3.16	An exponential sum in three parts. . . . .	39
3.17	An exponential sum of three terms, as a pair function and an exponential function. . . . .	40
3.18	A loop whose exponential sum is not dominated by the leading term. . . . .	46
3.19	A loop which engenders the eigenvalue zero. . . . .	53
3.20	A non-terminating loop. . . . .	60
3.21	A non-terminating loop which is detected by the techniques in Section 3.3.5, but not those of Section 3.2. . . . .	61
3.22	A positive polynomial with complex roots. . . . .	62

3.23	A polynomial for which the Upper Bound Theorem is incomplete. . . . .	63
3.24	A single-variable affine loop. . . . .	65
3.25	A terminating loop for which no linear ranking function exists. . . . .	66
4.1	A standard <i>while</i> loop and its generated (pseudo-)bytecode. . . . .	70
4.2	The bytecode generated by the guard condition $(x > 0 \wedge y > 0 \wedge z > 0)$ . . . . .	71
4.3	An affine loop. . . . .	72
4.4	A Flow diagram depicting the combined decision procedure of the techniques in Section 3. . . . .	77
4.5	A loop with 729 possible sign permutations, of which only 3 are considered. . . . .	85
5.1	Results of the application of the algorithms to single-constraint affine loops. . . . .	89
5.2	The application of the algorithms to 500 affine loops with two variables and constraints. . . . .	90
5.3	The application of the algorithms, excluding the PWC falsification heuristic, to larger affine loops. . . . .	93
5.4	A two-variable non-terminating loop [13]. . . . .	94
5.5	A three-variable non-terminating loop [13]. . . . .	94
5.6	A three-variable non-terminating loop. . . . .	95
5.7	A possibly non-terminating loop. . . . .	96

# Chapter 1

## Introduction

In 1928, Hilbert posed the decision problem: ‘does an algorithm exist which can validate (i.e., prove) a given mathematical assertion?’ The problem was proved impossible by, respectively, Church [10] and Turing [30], with Turing’s proof following from his own work on the undecidability of the halting problem<sup>1</sup>. The idea that program termination cannot be algorithmically approached has since been held by the computer science community at large — until recently. Due to focussed advances regarding the automation of certain termination verification techniques, termination provers now exist which are capable of acceptably, if not quite completely, verifying the termination of industrial programs [14]. Such verification is a necessary process, as the failure of software increasingly often leads to the failure of systems, and the standard methods of validating a constructed program — most prominently, testing — are unable to ensure that programs are entirely defect-free [19].

In addition to automated techniques of verification, two novel branches of termination inspection exist: the first is the proving of non-termination, that is, the detection of counter-examples to termination. This stance has been advocated by some as more valuable than correctness proving [17], due to the practical generation of valid counter-examples. The other problem which has been posed is that of conditional termination, which, instead of considering the universal validity of a program’s termination problem, seeks to provide a description of the circumstances under which the program terminates correctly. One might combine these perspectives, and consider that for a termination falsifier to locate a counter-example, it must construct some form of conditional description of non-termination which must be solved. The

---

<sup>1</sup>Turing’s halting problem asks whether a given Turing machine, on a specified input, will halt (terminate).

falsifier is thus designed to detect specific characterisations of non-termination. For certain simple programs, a subset of the entire non-termination condition is all that is required, as non-termination may always be recognised within a certain characterisation (this is the case for some linear programs [28]); however, in the situation where non-termination cannot be simplified to a finite set of characterisations, the ability of a falsifier to prove non-termination is directly comparable to the completeness of the conditions of non-termination it considers. From this perspective, the logical hybrid of non-termination and conditional termination — conditional non-termination — is a necessary consideration.

This exposition considers the non-termination of unnested affine program loops with integer variables. Such programs form part of basic programming arithmetic, and it is thus imperative that software model checkers are sufficiently able to evaluate their termination properties; although verification of their termination has been widely studied [11, 24, 6], their non-termination has as of yet only been mentioned [18] without a formal investigation. Unlike similar forms of loops in which variables are allowed to assume real or even rational values, the termination of affine loops over the integers has not been shown decidable. Since it is unlikely that this problem will be solved in the near future [8], a non-termination approach which is tailored to such loops seems pertinent. This approach should necessarily encapsulate as complete a description of affine loop non-termination as possible, since the reason for the failure to prove the termination of such loops decidable is the inability to characterise their non-termination using some specific condition.

To address affine loop non-termination, this text considers two methods: the first is based on the current non-termination technique of recurrent sets [18], and attempts to identify certain forms of affine loops which are guaranteed to display the periodic non-terminating behaviour towards which this technique is predisposed. The second applies concepts which have been used to prove the decidability of termination for other linear loop forms [28] to conditional non-termination; the structure of affine loops is decomposed and inspected, and with the aid of a few mathematical techniques, the mechanical non-terminating behaviour of these loops can be approximately described using linear constraints.

## 1.1 Document outline

To begin with, Chapter 2 introduces to the reader a few logical concepts with which software verification is concerned, thereby allowing the description of related approaches to termination verification and falsification.

The two techniques described in this text, in their entirety, are contained within Chapter 3; in fact, the majority of the section has been devoted to the development of what is termed the *positive weighted coefficient heuristic*, which provides a method of constraining an affine loop's variables to induce non-termination. This method is, unlike the currently known recurrent set approach, not concerned with periodic values of the loop.

Necessarily, the implementation of the developed techniques must be addressed; this discussion is found in Chapter 4. The goal of this implementation is to assess the issues which are encountered when attempting to practically utilise the techniques, and as such, the entire course of non-termination is followed — from the detection of loops within bytecode to the application of the heuristics to the structural decomposition of the interpreted affine loop.

Finally, experimental results, drawn from the application of the techniques to samples of random loops, are presented in Chapter 5. It is clear from these results that both the recurrent set and decomposition technique are useful. However, as far as the straightforward implementation described in this text is concerned, loops with more than a handful of variables and constraints place the selected satisfiability solver under computational strain.

## Chapter 2

# Background

### 2.1 A brief review of software verification and falsification

Software verification is the process of verifying that a constructed program agrees with its design specification. The topic has been developed logically since the middle of the twentieth century, when Alan Turing discussed methods of checking program routines [31]. An axiomatic approach to software verification began a few decades later, which, due to researchers such as Edsger Dijkstra, led to powerful methods of program reasoning [16]. These methods stimulated the desire to produce a procedure of automated program analysis, and there is currently a wealth of academic material which addresses the issue [20].

This section shall mention the primary terms associated with software verification and model checking, to the extent in which they relate to the current topic of program termination. The concepts which must be introduced are safety and liveness properties (termination being an example of the latter), enumerative and symbolic state representation, and invariants. Related approaches to the problem of termination shall be discussed in Section 2.3.

Relatively simple programs can easily be represented using mathematical logic; the standard approach to this representation [20] is given in Appendix A.1. Currently, it is sufficient to note the two foundational entities of such programs: states and locations. States are mappings of values to the program's variables, whereas locations are simply positions within the program's execution path. The manner in which locations are connected is described by a set of transitions. A location-state pair is called a configuration; a sequence of configurations is termed a

computation. This allows us to define the termination property, with which this text is concerned: a program is *terminating* if it contains no infinite computations, and *non-terminating* otherwise.

### 2.1.1 Safety and liveness properties

Although no complete solution to the problem of software verification can be developed [30], the task may be split into two sound, but incomplete, approaches regarding the errors within program code: the proof of their absence (verification), or their detection (falsification, or the search for program ‘bugs’). In general terms, verification attempts to prove that a superset abstraction of the program’s state space is free of errors, thereby allowing one to infer the same is true of the program’s state space itself. Falsification, on the other hand, searches a subset of the system’s reachable state space for errors; if an invalidity is found, the congruity of the program is disproved. Note how the failure of verification to prove the validity of the superset, and likewise that of falsification to locate an error within the subset, leads to an inconclusive result. Due to these limitations, the two approaches should be pursued in tandem in order to adequately verify constructed programs.

To more precisely describe the errors which are of concern, one may identify two forms of assertions regarding the program’s variables at particular locations within the program: safety properties prohibit erroneous program behaviour (e.g., ‘ $x$  is positive on the function’s return’ [20]), whereas liveness properties ensure the eventual achievement of desired actions (e.g., ‘each acquired lock will be released’ [14]). Such assertions can be expressed, for example, in terms of Büchi automata [2]. Termination is an example of a liveness property.

### 2.1.2 State representation

In practice, the states of a program are either represented in an enumerative manner, in which case each specific state is considered uniquely; or symbolically, where sets of states are represented by constraints, and actual states are not used [21]. The current text revolves around conditions upon program variables which attempt to describe a certain path of execution within the program, and symbolic state representation, therefore, is a natural accompaniment to the implementation of the techniques which shall be introduced.



Although alternative encodings exist, the most direct symbolic conditions make use of first-order logic, concisely representing infinite sets of program states, and creating the need for a close relationship between the software model checker and satisfiability modulo theory (SMT) constraint solvers [20]. The combination used for the purposes of this text is that of Symbolic Pathfinder (SPF) [25] and the CVC3 [5] constraint solver.

### 2.1.3 Invariants

An invariant of a program is any superset (or symbolic representation thereof) of the system's reachable state space which, like the reachable state space, is closed under program transitions [20]. Invariants were initially used in symbolic model checking to certify safety properties, since if an error location is not present in the invariant, it is also absent from the reachable state space. The concept of transition invariants generalises this technique to the verification of liveness properties, and it is subsequently shown that the verification of liveness properties in the presence of fairness assumptions can be reduced to the termination of unnested loops [23].

Initially, model checkers required the programmer to provide invariants, but much of the recent literature on the topic has addressed their automatic synthesis [11, 6, 32]. Linear invariant synthesis can in fact be reduced to the solving of non-linear arithmetic constraints [11], however non-linear constraint systems, although valuable (having also been used in practical solutions to termination falsification [18]), are generally less desirable than linear systems [13]. The synthesis of linear invariants is closely related to that of ranking functions for termination proofs. Ranking functions and other related topics shall be discussed presently, following a complete problem description.

## 2.2 Problem description

As mentioned in the previous chapter, termination discussions can adopt one of a few different views: attempting to verify termination, or to falsify termination by the location of a single non-terminating witness; and, more recently [13], the construction of a reasonable under-approximation of the witnesses to either termination or non-termination. This text targets the latter approach — an attempt to locate some set of non-terminating witnesses for a given affine loop. The reason for this aim is two-fold: firstly, the location of any non-terminating witness is

sufficient to falsify the termination property of a loop; and secondly, the more complete the set of witnesses obtained, the clearer the knowledge of the loop's non-termination properties. In searching for sets of non-terminating witnesses, the discussion must also touch on, and partly develop, termination verification techniques, since they are of near relation. The authors share the common view [18] that liveness properties should be investigated with the aid of both verification and falsification techniques.

This text is specifically concerned with affine loops — loops which contain only affine transformations on their variables; integer-valued variables will be considered in particular. The designed algorithm should attempt to reduce the non-termination of affine loops to sets of linear constraints on the loop variables, in the hope that current linear constraint solvers can successfully be applied to produce non-terminating witnesses.

Factors which fall outside of the scope of this document are concurrent programs, nested loops, loops which exhibit non-affine behaviour, and variables which are not restricted to the integers. With regard to integer-valued variables: the variables considered here are mathematical numbers, and are not restricted to a fixed width. In practice, most programs use fixed-width numbers, which can suffer from under- or overflow. These boundaries can affect the termination properties of a program, however the under- or overflow of a variable is most likely an undesired characteristic of a program, indicating unforeseen behaviour. Fixed-width variables are a particularly relevant issue to termination verifiers, since a loop which is terminating over the mathematical integers may be non-terminating over fixed-width integers [14]; a termination prover which overlooks this property may incorrectly verify a non-terminating loop. However, the issue is less pertinent for termination falsifiers: if a loop is non-terminating over the mathematical integers, but terminating over fixed-width integers (such as the simple loop in which an initially positive variable is continually incremented), a falsifier which considers mathematical numbers might return what it regards as a witness to non-termination, and, upon inspection of the witness, the programmer will likely discover undesired under- or overflow, possibly accompanied by a long execution path. If such behaviour was in fact of intelligent design, the falsifier has indeed failed to locate a programmatic error.

## 2.3 Related work

### 2.3.1 Decidability

A few remarkable results have already been obtained with regard to the termination of affine loops, primarily found during two outings into the field of linear algebra [28, 8]. Most importantly, the termination of affine loops is decidable when the loop variables range over the set of real numbers  $\mathbb{R}$ ; this result follows from the fact that, when concerned with non-termination, the positive real eigenvalues of the loop's matrix sufficiently describe the termination of the loop. Furthermore, the termination of affine loops is also a decidable problem over rational variables. However, these proof methods fail when loop variables are restricted to the set of integers  $\mathbb{Z}$ . The concepts used to arrive at these results rely on the decomposition of the loop's matrix representation, and assert that the non-termination of the affine loop in question implies the existence of a non-terminating witness which satisfies certain simple properties; witnesses which satisfy these properties can be found algorithmically, and so the absence of such a non-terminating initial state verifies the termination of the loop.

This text is founded upon the idea that algebraic concepts which have already yielded decidability results in this area are also of practical value — particularly, they can be adapted to approach both the falsification and non-termination precondition problems.

The following related methods, however, are not based on these algebraic concepts.

### 2.3.2 Termination verification

Currently, known techniques are able to capably verify the termination of many examples of affine loops [11, 24, 6, 14]. The most common approach to this verification is the search for so-called ranking functions, based on a suggestion by Turing [31]. These functions are mappings from the program variables into a well-ordered set, such that progression of the program engenders variable values which, when mapped, cause decreasing behaviour in this well-order; this progression cannot continue indefinitely, implying termination [14]. These ranking functions can be obtained automatically in a number of prominent ways:

- affine transformations may be represented as polyhedral cones, and ranking functions deduced from this representation [11];

- constraint template descriptions of ranking functions and supporting invariants may be defined, and solved to yield linear ranking functions [6]; in addition,
- a reduced set of linear inequalities has been provided, which, when solved, produce a linear ranking function [24]. This method is complete.

Once again, the limitation of current tools implies that the search for linear ranking functions is more valuable than non-linear functions, since termination provers are generally unable to verify non-linear ranking functions [14].

### 2.3.3 Conditional termination

While work has been done to synthesise linear ranking functions to assert the termination of program loops, few resources have been dedicated to the generation of termination preconditions, or approximations thereof. The methods devised for this purpose are themselves extensions of ranking function and invariant synthesis, and are often better suited to the synthesis of non-linear preconditions [13]. The most relevant, and practical, approach is to obtain candidate ranking functions; the conditions under which a candidate ranking function is in fact a ranking function are preconditions for the termination of the loop [13].

Within this text, the termination corollary presented in Section 3.3.8 provides an alternative to ranking function synthesis which is directly related to the termination preconditions of affine loops.

### 2.3.4 Termination falsification

Regarding related work, none is more relevant than that of termination falsification. Unfortunately, this topic has also received a surprisingly small amount of attention [18].

The current noteworthy approach to the falsification of loop termination is the search for recurrent sets which describe non-terminating behaviour [18, 32]. In this approach, the program is exhaustively (when possible) searched via symbolically execution; this search technique will systematically locate each possible program loop, also considering multiple iterations of a loop structure as possible loops. Upon the detection of a loop, which has a guard condition  $G(\mathbf{v})$  over the loop's variables  $\mathbf{v}$ , a template predicate  $R(\mathbf{v})$  is built directly from the loop's update

relation:

$$R(\mathbf{v}) = (G(\mathbf{v}) \wedge F(\mathbf{v})),$$

where  $F(\mathbf{v})$  is an arbitrary constraint. Constraint solvers are then employed, in a manner similar to that used for the synthesis of linear invariants [11], to generate an  $F(\mathbf{v})$  such that  $R(\mathbf{v})$  is recurrent with regard to the loop's update relation, that is, briefly:

$$R(\mathbf{v}) \neq \emptyset, \text{ and } R(\mathbf{v}) \Rightarrow R(\mathbf{v}'),$$

for any possible state  $\mathbf{v}'$  which can be reached from  $\mathbf{v}$ . Due to the presence of the loop's guard condition within  $R(\mathbf{v})$ , every  $v \in R(\mathbf{v})$  is a witness to the non-termination of the loop if  $R(\mathbf{v})$  is recurrent.

The method is incomplete, as infinite paths do not necessarily exhibit periodic behaviour (similar behaviour over a fixed number of iterations). In addition, although any element of a recurrent set is a witness to a loop's non-termination, the set does not necessarily contain initial values for every (periodic) non-terminating path.

Two welcome improvements to the algorithm would be a guarantee on the discovery of a recurrent set when a loop is non-terminating, and some theoretical bound on the number of iterations of a loop structure to consider. The candidate set construction proposed in Section 3.2 addresses these factors for particularly simple, but valuable, while loops.

## Chapter 3

# Approach

This chapter is primarily theoretical in nature, presenting two proposed solutions to the falsification of affine loop termination. In this regard, loops are presented mathematically, and applications to their appearance in programs is provided supplementarily where pertinent.

The chapter is structured as follows: after an introduction to affine loops, an adaptation of the known recurrent set method for falsification [18] is developed (Section 3.2). This adaptation does not rely on an abstract constraint template, and is shown to be complete for (able to decide the termination of) single-variable and cyclic affine loops.

Thereafter, the discussion proceeds to the primary result of this text — a termination falsification heuristic which is based on the Jordan decomposition of a loop’s transformation matrix (Section 3.3). This heuristic is both sound and able to provide reasonable preconditions for non-termination, but in general not complete for affine loops over integer variables. Unlike other practical approaches to affine loop termination, it is based on the concepts which yield the decidability of affine loops over real and rational variables [28, 8]. This decomposition also leads to a useful termination verification algorithm (Section 3.3.8).

### 3.1 Affine loops

Firstly: a *loop*, in the current context, refers to a standalone program loop without nesting. Such a loop consists of a guard condition and a body of update transformations; both regard the loop variables  $\mathbf{v} = [v_1 \dots v_n]^T$  — the finite array of variables which affect the termination of the loop, either directly, by appearing in the loop’s guard condition, or indirectly, by affecting

$$\begin{array}{l} \text{while } (\textit{guard condition}) \{ \\ \quad \textit{body} \\ \} \end{array}$$

**Figure 3.1:** An unnested program loop.

the update transformation of a variable which is present in the guard condition. If the variables are interpreted over a set  $\mathcal{X}$ , then each program state will be an array, i.e., an element of  $\mathcal{X}^n$ .<sup>1</sup> In this standalone form, any such state is a valid initial state for the loop.

Note that the syntactic *while* form used in Figure 3.1 is but one possible representation of the loop construct, chosen for its simplicity; alternative loop structures are mentioned in Chapter 4.

Secondly: A transformation  $t: \mathcal{X}^n \rightarrow \mathcal{X}$  is affine if it is of the form

$$t(x_1, \dots, x_n) = a_1x_1 + a_2x_2 + \dots + a_nx_n + c,$$

where  $a_1, \dots, a_n, c$  are scalars. Intuitively, an affine transformation is the combination of a linear transformation and a scalar shift, or translation.

As previously stated, this text is focussed on loops whose arithmetic form consists only of affine transformations, specifically over a set of integer-valued variables, and the scalar field of integers  $\mathbb{Z}$ ; hence the following definition:

**Definition 1 (affine loop).** An *affine loop* is a program loop without nesting in which:

- the guard condition contains a conjunction of linear inequalities,
- the loop body is made up of affine variable transformations,
- the loop variables are integer-valued, and
- all scalars are elements of  $\mathbb{Z}$ .

The general form of such a loop is shown in Figure 3.2.<sup>2</sup>

---

<sup>1</sup>A technical representation of affine loops as simple programs (as per Section A.1) can be achieved via the definition of loop locations, and the analogous relational expression of the affine transformations. Furthermore, the program states would then be mappings of the variables to elements of  $\mathcal{X}$ .

<sup>2</sup>In a simple program representation of a loop, the loop body would consist of a relational update:  $\mathbf{v}' = A\mathbf{v} + \mathbf{c}$ .

$$\begin{aligned} &\text{while } (G\mathbf{v} > \mathbf{b}) \{ \\ &\quad \mathbf{v} := A\mathbf{v} + \mathbf{c} \\ &\} \end{aligned}$$

**Figure 3.2:** The general form of an affine loop.

The loop depicted in Figure 3.2 is guarded by a combination of  $r$  linear inequalities over the  $n$  loop variables<sup>3</sup>  $\mathbf{v}$ . Each variable is iteratively updated, and thus each iteration of the loop performs  $n$  simultaneous affine transformations. For completeness, the elements of the loop are:

- $G$ , an  $r \times n$  integer matrix;
- $\mathbf{b}$ , an  $r \times 1$  integer matrix;
- $A$ , an  $n \times n$  integer matrix; and
- $\mathbf{c}$ , an  $n \times 1$  integer matrix.

Matrices are indexed using standard notation; e.g., the entry in the  $i$ th row and  $j$ th column of  $A$  is denoted  $a_{ij}$ . Furthermore,  $>$  defines a matrix relation, based on point-wise comparison:  $A > B$  if, and only if, both  $A$  and  $B$  are  $r \times s$  matrices and  $a_{ij} > b_{ij} \forall i = 1, \dots, r; j = 1, \dots, s$ .

Let  $\mathbf{v}^{[k]}$  ( $v_i^{[k]}$  for individual variables) depict the values of the loop variables after  $k$  iterations of the loop, such that the initial values are  $\mathbf{v}^{[0]}$  (or just  $\mathbf{v}$ ), and  $\mathbf{v}^{[k]} = A\mathbf{v}^{[k-1]} + \mathbf{c}$ . Then, due to the distributivity of matrix multiplication over addition,

$$\begin{aligned} \mathbf{v}^{[1]} &= A\mathbf{v} + \mathbf{c}, \\ \mathbf{v}^{[2]} &= A(A\mathbf{v} + \mathbf{c}) + \mathbf{c} = A^2\mathbf{v} + A\mathbf{c} + \mathbf{c}, \text{ and} \\ \mathbf{v}^{[k]} &= A^k\mathbf{v} + \sum_{l=0}^{k-1} A^l\mathbf{c}. \end{aligned}$$

This expression is somewhat unwieldy, hence the definition of the transformation matrix  $T$ , which stores the translation along with the linear transformation, in essence by augmenting

---

<sup>3</sup>Although  $\mathbf{v}$  is defined as a column vector, this notation shall occasionally be abused to refer to the set of loop variables within the vector.



the loop variable array with the constant 1:

$$\text{Let } \mathbf{v}_* = \begin{bmatrix} \mathbf{v} \\ 1 \end{bmatrix}, \text{ and } T = \begin{bmatrix} A & \mathbf{c} \\ 0 & 1 \end{bmatrix}.$$

The loop's update transformation can now be rewritten as  $\mathbf{v}_* := T\mathbf{v}_*$ , and thus

$$\mathbf{v}_*^{[k]} = T^k \mathbf{v}_*, \tag{3.1.1}$$

so that  $T^k$  captures  $k$  applications of the loop's update transformation. In addition, the guard matrix  $G$  can be augmented with a new column  $-\mathbf{b}$ :

$$\text{Let } G_* = \begin{bmatrix} G & -\mathbf{b} \end{bmatrix}, \tag{3.1.2}$$

so that the loop's guard condition can be written as  $(G_*\mathbf{v}_* > \mathbf{0})$ . An affine loop may now be defined succinctly as  $L = (G_*, T)$ .

The  $(n + 1) \times (n + 1)$  transformation matrix  $T$  is mathematically pleasing — allowing for the application of linear algebraic methods to affine loops; in fact it will be useful enough to warrant the omission of the subscript  $*$  in  $\mathbf{v}_*$  and  $G_*$ . Unless referring to the set of  $n$  loop variables, the variable vector  $\mathbf{v}$  will henceforth depict the array  $[v_1 \dots v_n \ 1]^T$ , and similarly  $G$  will denote the matrix defined in Equation 3.1.2. Let states of affine loops consist of  $n$  arbitrary integers and 1, so that each state is an element of  $\mathbb{Z}^{n+1}$ . The notation  $t(\mathbf{v}): \mathbb{Z}^{n+1} \rightarrow \mathbb{Z}$  will be used to depict individual affine variable updates, while  $T\mathbf{v}: \mathbb{Z}^{n+1} \rightarrow \mathbb{Z}^{n+1}$  represents the matrix multiplication of  $T$  with the array  $\mathbf{v}$ .

## 3.2 Non-termination via recurrent sets

The first manner of searching for infinite computations (in this context referring to infinite integer array sequences produced by a loop) is a localisation of a known method, which was mentioned in Section 2.3: the search for recurrent sets [18, 32]. The approach presented here differs from published techniques in several ways: it is only applied to affine loops, whereas recurrent sets can also be used to falsify the termination properties of non-linear loop constructs; the candidate set is explicitly defined here, as opposed to the use of a candidate template; and the present technique awards a form of completeness.

Put simply, this approach attempts to assert that whenever the values of the loop variables proceed away from their guard boundaries over a certain number of applied loop iterations  $k$ , this behaviour will continue indefinitely.

To begin with, a recurrent set for a loop with the guard condition ( $G\mathbf{v} > \mathbf{0}$ ) is defined as follows:

**Definition 2 (recurrent set).** Given an affine loop  $L = (G, T)$ , a set of integer arrays  $\mathcal{R} \subseteq \mathbb{Z}^{n+1}$  is *recurrent* under an affine transformation  $U: \mathbb{Z}^{n+1} \rightarrow \mathbb{Z}^{n+1}$  if

- $\forall \mathbf{r} \in \mathcal{R}: G\mathbf{r} > \mathbf{0}$ ,
- $\mathcal{R} \neq \emptyset$ , and
- $\forall \mathbf{r} \in \mathcal{R}: U\mathbf{r} \in \mathcal{R}$ .

Assume that for a given affine loop, a recurrent set  $\mathcal{R}$  under the loop's update transformation (that is, when  $U = T$ ) is known; then any  $\mathbf{r} \in \mathcal{R}$  ( $\mathcal{R}$  is not empty) is a witness to the loop's non-termination:  $\mathbf{r}$  satisfies the loop's guard condition, as does every state in the computation it generates. In general, a program is non-terminating if, and only if, a recurrent set for the program exists. Sufficiency is clear from the above, whilst necessity follows from the fact that the set of states visited by any infinite computation of a program is itself recurrent under the loop's update transformation.

The construction of a recurrent set from individual elements, as suggested by the previous argument for necessity, is implausible, as the infinite computation is not known; instead, candidate (potentially recurrent) sets must be described via constraints. Considering Definition 2, which might initially appear vague, one may view the first property as a preliminary constraint description of a candidate set's structure, and attempt to strengthen it in such a way as to induce the latter two properties. The additional strengthening constraint(s) can be automatically generated [18], or, as is shown here, explicitly defined in an effort to capture a specific form of non-terminating path.

In addition to the refinement of the candidate set, a further requirement for recurrent set construction is the specification of the transformation  $U$ , as it should encapsulate the loop's update transformation  $T$ , possibly multiple times. Hence, a search for a recurrent set might be performed under the compound update transformation for any number of loop iterations,  $T^k$ .

However, the consideration of compound iterations for all positive values of  $k$  is infeasible, and a theoretical restriction on the possible period lengths would be valuable.

Firstly, the consideration of compound update transformations implies a search for computations whose infinite behaviour can be described periodically. A simplified manifestation of periodic infinite behaviour is the continual shifting of the variables' values further from, or at least no nearer towards, their guard condition boundaries, drawing on the theory that transitions which leave the variables no closer to termination are of interest. For example, consider a loop with the guard condition  $(\mathbf{v} > 0)$ , along with the compound update transformation  $T^k$ : an initial state  $\mathbf{v}^{[0]}$  such that  $\mathbf{v}^{[(l+1)k]} \succeq_{\mathbf{v}} \mathbf{v}^{[lk]}$  for all  $l \geq 0$  might generate an infinite computation.

Stated generally, loops which exhibit a form of monotonicity over some period  $k$  are particularly interesting. This progressive variable behaviour over a period  $k$  can be defined more succinctly as follows:

**Definition 3 (periodic monotonicity).** An affine loop  $L = (G, T)$  is *periodically monotonic* over a period  $k$  if it possesses an update matrix  $T$  such that:

$$\forall \mathbf{v} \in \mathbb{Z}^{n+1} : GT^k \mathbf{v} \succeq_{\mathbf{v}} G\mathbf{v} \Rightarrow GT^{2k} \mathbf{v} \succeq_{\mathbf{v}} GT^k \mathbf{v},$$

where the relational operator  $\succeq_{\mathbf{v}}$  describes the  $r$  element-wise relationships between  $GT^k \mathbf{v}$  and  $G\mathbf{v}$  using the operators  $\{\leq, =, \geq\}$ . Periodic monotonicity (over period  $k$ ) thus requires this same configuration of relationships to hold under a further application of  $T^k$ . The concept of *strict periodic monotonicity* can be defined by replacing  $\succeq_{\mathbf{v}}$  of the previous definition with  $\succ_{\mathbf{v}}$ , which describes the  $r$  relationships using  $\{<, =, >\}$ .

A candidate set strengthening based on this property could be described by  $(G\mathbf{r} > \mathbf{0} \wedge GT^k \mathbf{r} \geq G\mathbf{r})$ , where comparisons occur element-wise, although for periodic monotonicity to imply non-termination, not only must every  $k$ th value of a witnessing computation conform to divergence, but, if the computation is viewed as a partition of  $k$  separate monotonic sequences:  $\{(\mathbf{v}^{[ik]}), (\mathbf{v}^{[ik+1]}), \dots, (\mathbf{v}^{[ik+(k-1)]})\}$ , then each sequence must proceed away from the guard condition's boundaries. Thus, a more apt definition of the candidate set must ensure the validity of each sequence's initial state, as well as its desired monotonicity. Accordingly, let the candidate set  $\mathcal{R}_k$  of a given affine loop be described by the constraint

$$Q_k(\mathbf{r}) = (GT^l \mathbf{r} > \mathbf{0} \wedge GT^{l+k} \mathbf{r} \geq GT^l \mathbf{r}; \forall l = 0, \dots, k-1).$$

It has already been stated that the existence of a recurrent set for a loop  $L$  implies the non-termination of  $L$ ; now, for the sake of formality:

**Lemma 1.** *If, for an affine loop  $L = (G, T)$ , the set of integer arrays  $\mathcal{R}_k$  which satisfy  $Q_k$  is recurrent under  $T^k$ , then  $L$  is non-terminating, and every  $\mathbf{r} \in \mathcal{R}_k$  is a witness to this non-termination.*

*Proof.* If  $\mathcal{R}_k$  is recurrent, any  $\mathbf{r}^{[0]} \in \mathcal{R}_k$  is a witness to the non-termination of the loop, as  $\mathbf{r}^{[ik]} \in \mathcal{R}_k$  for all  $i \geq 0$ , and  $\mathbf{r}^{[ik]} \in \mathcal{R}_k \Rightarrow G\mathbf{r}^{[ik+l]} > \mathbf{0}, \forall l = 0, \dots, k-1$ . Hence,  $G\mathbf{r}^{[l]} > \mathbf{0}$  for all  $l \geq 0$ . □

To display the validity of the given candidate set construction  $\mathcal{R}_k$ , one must show that the non-termination property of an affine loop which is monotonic over a period of  $k$  implies the candidate set's recurrence.

**Lemma 2.** *If  $\mathcal{R}_k$  is the candidate set derived from a periodically monotonic affine loop  $L = (G, T)$ , and  $L$  is non-terminating, then  $\mathcal{R}_k$  is recurrent under  $T^k$ . Furthermore, if  $L$  is in fact strictly periodically monotonic,  $\mathcal{R}_k$  describes exactly the set of witnesses to the non-termination of  $L$ .*

*Proof.* Assume  $L$  is periodically monotonic; that is,  $\forall \mathbf{v} \in \mathbb{Z}^{n+1}: GT^k\mathbf{v} \succeq_{\mathbf{v}} G\mathbf{v} \Rightarrow GT^{2k}\mathbf{v} \succeq_{\mathbf{v}} GT^k\mathbf{v}$ . Every  $\mathbf{r} \in \mathcal{R}_k$  satisfies the loop's guard condition —  $(G\mathbf{r} > \mathbf{0})$  — as it is subsumed by  $Q_k(\mathbf{r})$ .

To show that  $T^k\mathbf{r} \in \mathcal{R}_k$ , note that<sup>4</sup>  $GT^l(T^k\mathbf{r}) > \mathbf{0}, \forall l = 0, \dots, k-1$ , as  $Q_k(\mathbf{r})$  includes  $GT^l(T^k\mathbf{r}) \geq GT^l\mathbf{r} > \mathbf{0}$ . For the remainder of  $Q_k(T^k\mathbf{r})$ :  $GT^{l+k}(T^k\mathbf{r}) = GT^{2k}(T^l\mathbf{r}) \geq GT^k(T^l\mathbf{r}) = GT^l(T^k\mathbf{r}), \forall l = 0, \dots, k-1$ , by  $L$ 's periodic monotonicity and the assumption that  $Q_k(\mathbf{r})$  holds.

Finally, it must be shown that  $\mathcal{R}_k$  is not empty. Assume that  $\mathbf{s} \in \mathbb{Z}^{n+1}$  is a witness to  $L$ 's non-termination, but  $\mathbf{s} \notin \mathcal{R}_k$ . Then, for some  $l = 0, \dots, k-1$ , either  $GT^l\mathbf{s} \not> \mathbf{0}$  or  $GT^{l+k}\mathbf{s} \not\geq GT^l\mathbf{s}$ ; the former case cannot be true, as  $\mathbf{s}$  satisfies the guard condition of  $L$ , for all  $l \geq 0$ . Thus  $\mathbf{g}_j T^{l+k}\mathbf{s} < \mathbf{g}_j T^l\mathbf{s}$  for some row  $\mathbf{g}_j$  of  $G$  and a suitable  $l$ . By  $L$ 's periodic monotonicity, repeated

---

<sup>4</sup>Parentheses are used here only for clarity, as the combination of  $T^l$  and  $T^k$  is nothing more than matrix multiplication.

applications of  $T^k$  will cause this behaviour to continue:  $\mathbf{g}_j T^{ik}(T^{l+k}\mathbf{s}) \leq \mathbf{g}_j T^{ik}(T^l\mathbf{s})$ ,  $\forall i \geq 0$ . If  $L$  is in fact strictly periodically monotonic, the  $\leq$  in the previous relation may be replaced with  $<$ . Consider firstly the case where  $\mathbf{g}_j T^{t_j k}(T^{l+k}\mathbf{s}) = \mathbf{g}_j T^{t_j k}(T^l\mathbf{s})$ , for some  $t_j \geq 0$ : then this equality must hold for all  $i \geq t_j$ , or equivalently, periodically from iteration  $(t_j + k + l)$  onwards. Similar  $t$  values can be found for other inequalities which exhibit such decreasing behaviour, so that from the iteration induced by the maximum of these values  $t_{max}$  onwards, each of the  $r$  inequalities is non-decreasing under  $T^k$ . Letting  $\mathbf{s}_m = T^{l+t_{max}k}\mathbf{s}$ ,  $\mathbf{s}_m$  is a witness to  $L$ 's non-termination, and, because  $GT^j(\mathbf{s}_m) > \mathbf{0} \wedge GT^{j+k}\mathbf{s}_m \geq GT^j\mathbf{s}_m$ ,  $\forall j = 0, \dots, k-1$ ,  $\mathbf{s}_m \in \mathcal{R}_k$ , and  $\mathcal{R}_k \neq \emptyset$ . Lastly, consider the case where  $\mathbf{g}_j T^{ik}(T^{l+k}\mathbf{s}) < \mathbf{g}_j T^{ik}(T^l\mathbf{s})$  for all  $i \geq 0$ . At some point, the sequence will pass the boundaries of the guard condition:  $\exists i \geq 0: \mathbf{g}_j T^{l+ik}\mathbf{s} \leq \mathbf{0}$ ; this contradicts the infinite property of  $\mathbf{s}$ , so that any  $\mathbf{r} \notin \mathcal{R}_k$  does not generate an infinite computation. This, combined with the fact that every element of  $\mathcal{R}_k$  is a witness to  $L$ 's non-termination, implies that  $\mathcal{R}_k$  describes precisely the set of infinite computation generators when  $L$  is strictly periodically monotonic. □

$$\begin{array}{l} \text{while } (x > 0) \{ \\ \quad x := -x + 10 \\ \} \end{array}$$

**Figure 3.3:** A periodically monotonic loop.

Consider the example in Figure 3.3: in this case the compound update transformation over two iterations is  $T^2(x) = -(-x + 10) + 10 = x$ , so that the loop is non-terminating if, and only if,  $x > 0 \wedge -x + 10 > 0$ . Constructing a candidate set for  $k = 2$  yields the constraint

$$\begin{aligned} Q_2(x) &= (x > 0 \wedge T(x) > 0 \wedge T^2(x) \geq x \wedge T^3(x) \geq T(x)) \\ &= (x > 0 \wedge -x + 10 > 0 \wedge x \geq x \wedge -x + 10 \geq -x + 10) \\ &= (x > 0 \wedge -x + 10 > 0), \end{aligned}$$

This set is recurrent, since  $x \in \mathcal{R}_2(x) \Rightarrow x > 0$ ;  $\mathcal{R}_2(x) \neq \emptyset$ ; and  $x \in \mathcal{R}_2(x) \Rightarrow T^2(x) = x \in \mathcal{R}_2(x)$ . It can be seen that  $\mathcal{R}_2(x) = \{x: 1 \leq x \leq 9\}$  describes fully the non-terminating witnesses of the loop — the best-case scenario for the application of the recurrent set lemma.

Lemmas 1 and 2 prove the theoretical value of the explicit recurrent set construction, with regard to loops which exhibit periodically monotonic behaviour; specifically, termination is decided in the case of periodically monotonic loops, and the technique describes the non-termination completely when strictly periodically monotonic loops are considered. The relevance of the theory, however, rests on the the ability to recognise such loops, as well as the possible periods over which monotonicity is exhibited; it is implausible to attempt to check for recurrence over an arbitrary number of transformation periods. The first problem is relatively simple to solve: periodic monotonicity can be encoded as a satisfiability problem, and thus recognised; limiting the number of periods to consider, on the other hand, is not as simple. There are, fortunately, two forms of affine loops which are known to both exhibit periodic monotonicity and allow for a bounded iterative procedure.

### 3.2.1 Single-variable affine loops

Single-variable affine loops contain one loop variable<sup>5</sup>, and thus engender a  $2 \times 2$  transformation matrix. A common index-adjusting loop, (whose loop variable is either incremented or decremented during each update transformation) is an example of a single-variable loop, and thus conforms to the general description of Figure 3.4. Remarkably, all single-variable loops are periodically monotonic over a period of two, so that the recurrent set approach presented thus far is not only applicable, but also decides the termination of such loops.

$$\begin{array}{l} \text{while } (\mathbf{g}x > \mathbf{b}) \{ \\ \quad x := ax + c \\ \} \end{array}$$

**Figure 3.4:** A general single-variable loop.

The periodic monotonicity of single-variable loops is simple to see: given an affine transformation  $x := ax + c$ , consider the period  $k = 2$ :

$$x^{[2]} = a(ax + c) + c = a^2x + c(a + 1),$$

---

<sup>5</sup>Note that only those variables which affect the guard condition of a loop are considered loop variables. The body of a single-variable loop may still contain arbitrarily many instructions and variables.

which is in fact non-decreasing, as, if  $x_1 \leq x_2$ , then  $a^2x_1 \leq a^2x_2$ , and  $x_1^{[2]} \leq x_2^{[2]}$ . Each of the guard constraints within the guard condition is of the form  $gx - b > 0$ ; hence:

$$\begin{aligned}
gx^{[2]} - b \geq gx - b &\Leftrightarrow gx^{[2]} \geq gx \\
&\Rightarrow \begin{cases} x^{[2]} \geq x & \text{if } g \geq 0, \\ x^{[2]} \leq x & \text{if } g < 0. \end{cases} \\
&\Rightarrow \begin{cases} x^{[4]} \geq x^{[2]} & \text{if } g \geq 0, \\ x^{[4]} \leq x^{[2]} & \text{if } g < 0. \end{cases} \\
&\Rightarrow gx^{[4]} \geq gx^{[2]} \\
&\Leftrightarrow gx^{[4]} - b \geq gx^{[2]} - b.
\end{aligned}$$

A similar deduction holds for  $\leq$ , so that every single-variable loop is periodically monotonic over a period of 2.

By Lemma 2, the candidate set  $\mathcal{R}_2$  of a single-variable loop is recurrent if, and only if, the loop is non-terminating. The importance of this localisation is that the only period whose recurrence need be considered to decide the termination of such loops is  $k = 2$ .

```

while (x > 0) {
    x := -3x + 20
}

```

**Figure 3.5:** A non-terminating single-variable loop.

The example loop in Figure 3.5 is proved non-terminating by the recurrent candidate set

$$\begin{aligned}
\mathcal{R}_2(x) &= \{(x > 0) \wedge (-3x + 20 > 0) \wedge (9x - 40 \geq x) \wedge (-27x + 140 \geq -3x + 20)\} \\
&= \{x > 0 \wedge x < \frac{20}{3} \wedge x \geq 5 \wedge x \leq 5\} \\
&= \{5\}.
\end{aligned}$$

Note that  $x = 5$  produces a cyclic path through the loop, and is the only non-terminating witness.

One would hope that the completeness displayed by single-variable loops would extend to loops with a higher number of variables, however, even two-variable affine loops need not be

periodically monotonic over any number of iterations. Consider the example<sup>6</sup> loop in Figure 3.6.

$$\begin{aligned} &\text{while } (x > 0) \{ \\ &\quad x := 2y \\ &\quad y := -x \\ &\} \end{aligned}$$

**Figure 3.6:** A loop which is not periodically monotonic.

The first few iterations of the loop are

$$(x, y) \rightarrow (2y, -2y) \rightarrow (-4y, 4y) \rightarrow (8y, -8y) \rightarrow (-16y, 16y).$$

The loop cannot be periodically monotonic over an odd iteration  $k$ ; consider the counter-example  $\mathbf{v} = (-2^k - 1, -1)$  (considered as a column vector): this implies  $GT^k \mathbf{v} = (-2^k, 2^k) \geq (-2^k - 1, -1) = \mathbf{v}$ , however  $GT^{2k} \mathbf{v} = (2^{2k}, -2^{2k}) \not\geq (-2^k, 2^k)$ . Similarly, periodic monotonicity does not hold for even iterations  $k$ : consider  $\mathbf{v} = (-2^k - 1, 1)$ :  $GT^k \mathbf{v} = (-2^k, 2^k) \geq (-2^k - 1, 1) = \mathbf{v}$ , however  $GT^{2k} \mathbf{v} = (-2^{2k}, 2^{2k}) \not\geq (-2^k, 2^k)$ .

Another notable, though uncommon example of a periodically monotonic loop is an affine loop which always returns, or cycles, to its initial set of values over a period  $k$ .

### 3.2.2 Cyclic affine loops

A cyclic affine loop is necessarily periodically monotonic, as, by definition, a cyclic loop of period  $k$  (a so-called  $k$ -cyclic loop) is such that  $\mathbf{v}^{[k]} = \mathbf{v}$ , so that the transformation matrix  $T$  induces  $T^k = I$ , the identity matrix, and  $GT^{2k} \mathbf{v} = GT^k \mathbf{v} = G\mathbf{v}$ . By Lemmas 1 and 2 then, the set  $\mathcal{R}_k$  is non-empty if, and only if,  $L$  is non-terminating.

The issue faced when considering cyclic loops is similar to the caveat already mentioned in the case of periodically monotonic loops of a more general form — before strong conclusions can be drawn from the technique, it must be determined whether a loop possesses the properties for which these conclusions are veritable. In this case, one must be able to determine whether

---

<sup>6</sup>The update transformations present in example loops should be considered sequentially (as opposed to the inherent simultaneous nature of the matrix representation of an affine loop) unless otherwise stated.



a loop is cyclic over some period  $k$  or not; again though, this property can be encoded as a satisfiability constraint, so that each period  $k$  can be checked for cyclic behaviour<sup>7</sup>. The primary advantage of considering cyclic loops, similar to the case of single-variable loops, is that cyclic behaviour can only occur over a finite number of periods, dependent on the dimensions of the transformation matrix.

This result follows from a remarkable theorem in group theory, initially proved by Minkowski [22]. Considering the group  $GL(n, \mathbb{Z})$  of  $n \times n$  integer matrices whose inverses also have integer entries, the theorem states that  $GL(n, \mathbb{Z})$  has finitely many finite subgroups, up to isomorphism. In the current context, the transformation matrix  $T$  of a  $k$ -cyclic affine loop with  $(n - 1)$  loop variables is an element of  $GL(n, \mathbb{Z})$ , as  $T$  is an  $n \times n$  integer matrix, and  $T^{-1} = T^{k-1}$  also has integer entries. The cyclic subgroup formed by the  $k$  powers of  $T$  is finite, and by Minkowski's theorem there can be only finitely many such subgroups, so that there are only finitely many possible periods that an  $n \times n$  cyclic matrix  $T$  may possess. The application of this result is that there must be some maximal period  $K(n)$  for cyclic loops in  $(n - 1)$  variables, and thus only finitely many iterations need be checked to determine whether a given affine loop is cyclic or not; this bounds the recurrent set procedure.

The function  $K(n)$  is thus of particular interest: one need only check whether  $T^k = I$  for all  $k = 1, \dots, K(n)$  to decide whether an affine loop is cyclic or not. This function, unfortunately, grows quite rapidly, and it has been claimed that for larger  $n$ , the value  $n!2^n$  describes a maximal order [26]. The asymptotic behaviour of  $K(n)$  is of less interest to us than its value for small values of  $n$ , as an affine loop within program code is unlikely to depend on more than a handful of loop variables. With this in mind, consider Table 3.1, which depicts  $K(n)$  for small  $n$  [22]. It is a fact that  $K(2n) = K(2n + 1)$  for all positive  $n$ .

Once a loop is certified cyclic over a period  $k$ , its termination can be decided by checking the recurrence of  $\mathcal{R}_k$ . The 2-cyclic loop in Figure 3.7 is non-terminating, by the recurrent set  $\mathcal{R}_2 = \{x, y: x > 0 \wedge y > 0\}$ , whereas the 2-cyclic loop in Figure 3.8 is terminating, as  $\mathcal{R}_2 = \{x: x > 0 \wedge -x > 0\}$  is empty.

To return to the more general discussion involving periodically monotonic loops, note that, although an iterative bound (such as  $K(n)$  in the case of cyclic affine loops) is not known

---

<sup>7</sup>An alternative characteristic of a cyclic loop is that its transformation matrix  $T$  engenders eigenvalues which are roots of unity, since some power of its Jordan matrix (see Section 3.3.2) must be the identity matrix.

$n$	$K(n)$	$n$	$K(n)$
2	6	3	6
4	12	5	12
6	30	7	30
8	60	9	60
10	120	11	120

**Table 3.1:** The maximal periods  $K(n)$  of cyclic loops with few loop variables.

```

while ( $x > 0$ ) {
     $x := y$ 
     $y := x$ 
}

```

**Figure 3.7:** A 2-cyclic non-terminating loop.

for periodic monotonicity, the algorithm still proves useful, and must only be halted at some limit ( $K(n)$  is a suitable suggestion). As another example, consider Figure 3.9; this loop is not cyclic, and the recurrent set algorithm yields as a non-terminating witness  $(x, y) = (1, 2)$ , since  $(1, 2) \rightarrow (5, 2) \rightarrow (5, 2) \rightarrow \dots$ .

To conclude the exposition of the technique, recall that an explicit description of the candidate set was adopted in place of an automatic strengthening technique; the explicit approach is theoretically weaker, as it detects only specific forms of infinite paths, however, if the explicit candidate set is cleverly defined, it can be used to draw stronger conclusions than an abstract approach (in this case, decidability, and even completeness), as well as being simpler, and, considering that a constraint need not first be found, possibly more efficient to implement.

### 3.2.3 Termination verification via recurrent sets

Although it is not the concern of this text, the recurrence approach to termination falsification can also be adapted to termination verification: instead of attempting to identify periodically divergent behaviour over the set of loop variables, as in falsification, one need only show that

```

while ( $x > 0$ ) {
     $x := -x$ 
}

```

**Figure 3.8:** A 2-cyclic terminating loop.

```

while ( $x > 0$ ) {
     $x := 2y + 1$ 
     $y := 2y - 2$ 
}

```

**Figure 3.9:** A non-terminating, non-cyclic loop.

a single guard constraint is periodically converging towards its boundary. Formally, it must be shown, for some guard constraint  $\mathbf{g}\mathbf{v} > 0$ , that

$$\mathbf{g}\mathbf{v} > 0 \Rightarrow \mathbf{g}T^k\mathbf{v} < \mathbf{g}\mathbf{v}; \forall \mathbf{v} \in \mathbb{Z}^{n+1}.$$

For simple loop forms, such as those including variable decrements, this approach is sufficient for termination verification. The loop in Figure 3.10 is such a loop, as  $(x > 0 \wedge y > 0) \Rightarrow (x - y < x)$ .

```

while ( $x > 0 \wedge y > 0$ ) {
     $x := x - y$ 
     $y := y + 1$ 
}

```

**Figure 3.10:** A loop for which termination can easily be verified.

### 3.2.4 Non-linear loops

As a brief aside: the recurrent set technique is not restricted to affine loops; in fact, the concept of periodic monotonicity can be generalised to loops of a non-linear nature, since it involves nothing more than a relationship between periodic values generated by a loop. This topic

falls outside of the scope of this text, but it seems viable that periodic monotonicity might be practically useful when applied to other loop forms.

To conclude Section 3.2, recall that the concept of periodic monotonicity was introduced, and, combined with the upper bounds on their possible periods, yielded the decidability of single-variable and cyclic affine loops. As shall be shown in Chapter 5, this approach remains useful, if incomplete, when applied to more general affine loops of a multi-variable or non-cyclic nature.

### 3.3 Non-termination via Jordan decomposition

The search for recurrent sets in an attempt to prove non-termination is a valuable technique, prominently due to its practicality; however, this technique does not take advantage of the transparent mechanics of affine loops, and a more rigorous method, tailored specifically to the simple form of loop at hand, is desired. The set of infinite path generators returned by a proposed technique should under-approximate the complete set of witnesses as closely as possible, and, in the case of no obtainable witnesses, the ability to verify the loop's termination would be valuable. With this in mind, the affine loop is decomposed, in search of an explicit formula for iterative loop values, which can then be analysed independently. The following concepts, based on the Jordan decomposition of an affine loop, have previously been used to prove the decidability of the termination of affine loops over the set of real [28] and rational [8] numbers, as well as over the integers for a simplified form of loop (Section 3.3.4).

As an overview of the approach to follow: an affine loop is defined by a guard matrix  $G$  and transformation matrix  $T$ ; the powers of  $T$  can be used to express the values of the loop variables after any number of iterations. The Jordan decomposition of the transformation matrix allows the powers of  $T$  to be easily expressed in terms of the matrix's eigenvalues, and from this expression the values of the loop variables after a number of iterations  $k$  (and thus the linear combinations  $G\mathbf{v}$  within the guard condition) can also be explicitly expressed; non-termination of the loop can then be stated in terms of the positivity of these functional expressions. This deduction is described in Section 3.3.2.

The explicit functions which describe the iterative variable values are sums of exponential terms, and thus difficult to constrain in the desired positive manner. Section 3.3.3 examines

such functions in order to portray this difficulty and suggest an approximate solution. Section 3.3.4 then interjects the deduction in order to describe the known termination results for affine loops over integer variables — results which were obtained from the preceding concepts.

The suggestions of Section 3.3.3 can only be applied to exponential sums with positive bases, however, the functions generated by the decomposition of the loop may have negative and complex bases. Section 3.3.5 addresses this issue by abstracting the undesired terms, obtaining a lower bound for the function which is of the desired form. The chapter concludes with the presentation of a few algorithms (termed ‘heuristics’ due to their concern with the abstracted function) which constrain this lower bound function in such a manner as to engender the non-termination of the loop.

Firstly though, the concepts of diagonalisation and Jordan decomposition must be applied to the general form of an affine loop.

### 3.3.1 Diagonalisation

Consider again the algebraically manageable form of an affine loop  $L = (G, T)$ , first presented in Figure 3.2:

$$\begin{array}{l} \text{while } (G\mathbf{v} > \mathbf{0}) \{ \\ \quad \mathbf{v} := T\mathbf{v} \\ \} \end{array}$$

The update transformation  $\mathbf{v} := T\mathbf{v}$  can easily be decomposed when  $T$  is a diagonalisable  $(n + 1) \times (n + 1)$  integer matrix, and for now, such matrices are considered exclusively.  $T$  is diagonalisable if matrices  $P$  and diagonal  $D$  exist such that

$$T = PDP^{-1},$$

with  $P$  and  $D$  both  $(n + 1) \times (n + 1)$  complex matrices, and  $d_{ij} = \lambda_i$  if  $i = j$ , and 0 otherwise; the entries along the diagonal of  $D$  are eigenvalues of  $T$ , and an eigenvalue’s algebraic multiplicity specifies the number of such appearances. Affine loops with diagonalisable transformation

matrices shall be termed *diagonalisable affine loops*. Continuing:

$$\begin{aligned} T^k &= (PDP^{-1})^k \\ &= (PDP^{-1})(PDP^{-1}) \dots (PDP^{-1}) \\ &= PD^kP^{-1}. \end{aligned} \tag{3.3.1}$$

The powers of a diagonal matrix are easily calculated:  $D^k$  is itself a diagonal matrix with entries  $\lambda_i^k$ ,  $i = 1, \dots, n+1$ . Denoting the elements of  $P^{-1}$  as  $q_{ij}$  as to avoid confusion, and making use of Equation 3.3.1, one can obtain a formula for  $t_{il}^{[k]}$  — the  $(i, l)$ th entry of  $T^k$ . Firstly, note that the  $(j, l)$ th entry in  $D^kP^{-1}$  is  $\lambda_j^k q_{jl}$ , so that subsequently

$$t_{il}^{[k]} = \sum_{j=1}^{n+1} p_{ij} (\lambda_j^k q_{jl}). \tag{3.3.2}$$

Combining this with  $\mathbf{v}^{[k]} = T^k \mathbf{v}$  (Equation 3.1.1), each entry in  $\mathbf{v}^{[k]}$  can be written as the dot product of a row in  $T^k$  and the column vector  $\mathbf{v}^{[0]}$ :

$$\begin{aligned} v_i^{[k]} &= \sum_{l=1}^{n+1} t_{il}^{[k]} v_l \\ &= \sum_{l=1}^{n+1} \sum_{j=1}^{n+1} p_{ij} \lambda_j^k q_{jl} v_l \end{aligned}$$

and, reordering the elements to obtain a sum over the eigenvalues' powers,

$$v_i^{[k]} = \sum_{j=1}^{n+1} \left( p_{ij} \sum_{l=1}^{n+1} q_{jl} v_l \right) \lambda_j^k. \tag{3.3.3}$$

Intuitively: the value of  $v_i$  after  $k$  iterations of the loop  $L$  is a sum in  $(n+1)$  parts; each component is an exponential one<sup>8</sup>, whose base is an eigenvalue  $\lambda_j$ , exponent is  $k$ , and coefficient (within parentheses in Equation 3.3.3) is a linear combination  $p_{ij}(q_{j1}v_1 + \dots + q_{j(n+1)}v_{n+1})$  of the  $(n+1)$  loop variables over the scalar field of complex numbers  $\mathbb{C}$ .

One may note, as an interesting aside, that

$$v_i^{[0]} = \sum_{l=1}^{n+1} \left( \sum_{j=1}^{n+1} p_{ij} q_{jl} \right) v_l = v_i,$$

as it should, because  $PP^{-1} = I$  implies that  $\sum p_{ij}q_{jl} = 1$  when  $l = i$ , and 0 otherwise.

---

<sup>8</sup>Within this text, 'exponential function' shall refer to an expression in which a base is raised to some variable exponent, such as  $f(k) = c\lambda^k$ ; this should not be confused with the function  $e$ .

$$\begin{array}{l}
 \text{while } (x > 0) \{ \\
 \quad x' := 6x - 8y \\
 \quad y := x \\
 \quad x := x' \\
 \}
 \end{array}$$

**Figure 3.11:** A diagonalisable two-variable (excluding the auxiliary variable  $x'$ ) loop.

The example loop in Figure 3.11, whose transformation matrix can be written succinctly in terms of the variables  $x$  and  $y$ , is diagonalisable, since

$$\begin{aligned}
 T^k &= \begin{bmatrix} 6 & -8 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}^k = \begin{bmatrix} 4 & 2 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}^k \begin{bmatrix} \frac{1}{2} & -1 & 0 \\ -\frac{1}{2} & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix} = PD^kP^{-1} \\
 &= \begin{bmatrix} 2(4)^k - (2)^k & -4(4)^k + 4(2)^k & 0 \\ \frac{1}{2}(4)^k - \frac{1}{2}(2)^k & -(4)^k + 2(2)^k & 0 \\ 0 & 0 & 1 \end{bmatrix}.
 \end{aligned}$$

The explicit function for iterations of  $x$ , by Equation 3.3.3, is the first row of  $T^k[x \ y \ 1]^T$ :

$$x^{[k]} = (2x - 4y)4^k + (-x + 4y)2^k.$$

In the case of homogeneous affine loops such as that of Figure 3.11, where each variable  $v_i$  is constrained above or below some integer  $b_i$  by the guard condition, it follows that  $L$  is non-terminating on  $\mathbf{v}$  if, and only if, for each  $v_i$ , the right-hand side of Equation 3.3.3 is greater than (or less than, according to the constraint's relational operator) its relevant  $b_i$  for all  $k \geq 0$ . However, the current concern is the general affine loop, whose guard condition is  $(G\mathbf{v} > \mathbf{0})$ . Each of the  $r$  rows of  $G$  represents a linear inequality over the entries of  $\mathbf{v}$ , where the  $i$ th linear inequality is  $(g_{i1}v_1 + \dots + g_{in}v_n + g_{i(n+1)} > 0)$ . Thus,  $G\mathbf{v}^{[k]} > \mathbf{0}$  if, and only if,

$$\sum_{m=1}^{n+1} g_{im}v_m^{[k]} > 0, \quad \forall i = 1, \dots, r.$$

And, combining this with the explicit expression obtained in Equation 3.3.3:

$$\sum_{m=1}^{n+1} g_{im} \left( \sum_{j=1}^{n+1} p_{mj} \sum_{l=1}^{n+1} q_{jl}v_l \lambda_j^k \right) > 0, \quad \forall i = 1, \dots, r.$$

Once again grouping terms to obtain an outer sum over the transformation matrix's eigenvalues,  $L$  is non-terminating on  $\mathbf{v}$  if

$$\sum_{j=1}^{n+1} \left( \sum_{m=1}^{n+1} g_{im} p_{mj} \sum_{l=1}^{n+1} q_{jl} v_l \right) \lambda_j^k > 0, \quad \forall k \geq 0, \forall i = 1, \dots, r. \quad (3.3.4)$$

Clearly, an understanding of the exponential sum in Equation 3.3.4 might award an insight into the termination behaviour of diagonalisable affine loops; this is a sum over  $(n + 1)$  exponential functions, each of the form  $C_{ij}(\mathbf{v})\lambda_j^k$ , where

$$C_{ij}(\mathbf{v}) = \sum_{m=1}^{n+1} g_{im} p_{mj} \sum_{l=1}^{n+1} q_{jl} v_l. \quad (3.3.5)$$

$\sum g_{im} p_{mj}$  is a complex number, so that, similar to the coefficient of  $\lambda_j^k$  in Equation 3.3.3,  $C_{ij}(\mathbf{v})$  is no more than a linear combination of the entries in  $\mathbf{v}$ , and thus itself an element of  $\mathbb{C}$ .

For ease of reference<sup>9</sup>:

**Lemma 3.** *An affine loop  $L = (G, T)$ , such that  $T$  is diagonalisable, is non-terminating if, and only if, some  $\mathbf{v} \in \mathbb{Z}^{n+1}$  exists such that*

$$\sum_{j=1}^{n+1} C_{ij}(\mathbf{v})\lambda_j^k > 0, \quad \forall k \geq 0, \forall i = 1, \dots, r,$$

where  $G$  has  $r$  rows and  $C_{ij}(\mathbf{v})$  is as in Equation 3.3.5.

Before investigating inequalities of the previous form, it must be stated that these expressions relate only to diagonalisable affine loops, and as such, a similar result should first be obtained for loops whose transformation matrices are not diagonalisable (so called defective matrices).

### 3.3.2 Jordan decomposition

Although a given square matrix might not be diagonalisable, a similar (but slightly more complex) decomposition can always be performed; namely, the Jordan matrix decomposition. For any square matrix  $T$  there exist matrices  $P$  and  $J$  such that

$$T = PJP^{-1},$$

---

<sup>9</sup>The decompositional characterisation of affine loop termination, which Lemmas 3 and 4 describe, was first discussed in [28]. Section 3.3.4 outlines the known results drawn from this characterisation [28, 8], whereas Section 3.3.5 extends this characterisation to approximately describe the non-termination properties of an affine loop.



where  $J$  is a Jordan matrix, called the Jordan canonical/normal form of  $T$ .  $J$  is filled with 0 entries, except for its diagonal, which consists of Jordan blocks  $Y_1, \dots, Y_t$ , themselves matrices in which the diagonal is populated with some eigenvalue  $\lambda_i$  of  $T$  and the super-diagonal with 1. As in the case of diagonalisation, the algebraic multiplicity of an eigenvalue determines the number of appearances it makes along  $J$ 's diagonal, possibly divided among numerous Jordan blocks; an eigenvalue's geometric multiplicity denotes the number of Jordan blocks it engenders. Hence, if each eigenvalue's algebraic and geometric multiplicities are equal, every Jordan block has dimension 1, and  $T$  is diagonalisable. Affine loops which are not diagonalisable shall be termed *defective*. Visually, a matrix's Jordan matrix is of the following form:

$$J = \begin{bmatrix} Y_1 & 0 & \dots & 0 \\ 0 & Y_2 & \dots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & \dots & 0 & Y_t \end{bmatrix}, \text{ where } Y_i = \begin{bmatrix} \lambda_i & 1 & 0 & \dots & 0 \\ 0 & \lambda_i & 1 & \dots & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & \dots & 0 & \lambda_i & 1 \\ 0 & \dots & 0 & 0 & \lambda_i \end{bmatrix}$$

Again (similar to Equation 3.3.1),

$$T^k = PJ^kP^{-1}, \tag{3.3.6}$$

however, although a power of a diagonal matrix is nothing more than a matrix in which the entries along the diagonal have been raised to the given power, a Jordan matrix cannot be iterated quite as simply. Due to its diagonal block matrix form, if  $J$  is raised to the exponent  $k$ , each Jordan block is raised to the power of  $k$  individually. Let the starting index of  $Y_i$  in  $J$  be  $u_i$ , so that  $Y_i$  occupies the block from  $(u_i, u_i)$  to  $(u_{i+1}, u_{i+1})$  in  $J$ , and  $Y_i$  has dimension  $u_{i+1} - u_i$ ; define  $u_{t+1} = n + 1$ . The  $k$ th power of  $J$  is as follows:

$$J^k = \begin{bmatrix} Y_1^k & 0 & \dots & 0 \\ 0 & Y_2^k & \dots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & \dots & 0 & Y_t^k \end{bmatrix}, \text{ and } Y_i^k = \begin{bmatrix} \lambda_i^k & \binom{k}{1}\lambda_i^{k-1} & \dots & \binom{k}{(u_{i+1}-u_i)-1}\lambda_i^{k-((u_{i+1}-u_i)-1)} \\ 0 & \lambda_i^k & \dots & \binom{k}{(u_{i+1}-u_i)-2}\lambda_i^{k-((u_{i+1}-u_i)-2)} \\ \vdots & & \ddots & \vdots \\ 0 & \dots & 0 & \lambda_i^k \end{bmatrix}$$

Proceeding as in Section 3.3.1: the  $(j, l)$ th entry of  $J^kP^{-1}$  is the dot product of the  $j$ th row of  $J^k$  with the  $l$ th column of  $P^{-1}$ . If the  $j$ th row of  $J^k$  forms part of the  $d$ th Jordan block  $Y_d$ ,

then this dot product is a sum over the  $u_{d+1} - j$  non-zero entries in row  $j$  of  $J^k$ :

$$\sum_{s=0}^{u_{d+1}-(j+1)} \binom{k}{s} \lambda_d^{k-s} q_{(j+s)l}.$$

To describe an entry of  $T^k = PJ^kP^{-1}$  explicitly, one could simply sum over every column of  $P$ , as was done when handling diagonalisable matrices. However, this would enumerate every row of  $J^kP^{-1}$  individually, omitting the fact that Jordan blocks contain numerous appearances of a single eigenvalue. Instead, the sum is performed in two parts — firstly over Jordan blocks (by the index  $j$ ), and subsequently over each block's columns (index  $d$ ; note how  $\psi_j = u_{j+1} - u_j - 1$  will represent the size (minus 1) of  $Y_j$ ):

$$t_{il}^{[k]} = \sum_{j=1}^t \sum_{d=0}^{\psi_j=u_{j+1}-u_j-1} p_{i(u_j+d)} \left( \sum_{s=0}^{u_{j+1}-(u_j+d)-1} \binom{k}{s} \lambda_j^{k-s} q_{(u_j+d+s)l} \right),$$

and the formula  $\mathbf{v}^{[k]} = T^k \mathbf{v}$  yields

$$\begin{aligned} v_i^{[k]} &= \sum_{l=1}^{n+1} t_{il}^{[k]} v_l \\ &= \sum_{l=1}^{n+1} \left( \sum_{j=1}^t \sum_{d=0}^{\psi_j} \sum_{s=0}^{\psi_j-d} p_{i(u_j+d)} \binom{k}{s} \lambda_j^{k-s} q_{(u_j+d+s)l} \right) v_l \\ &= \sum_{j=1}^t \left( \sum_{l=1}^{n+1} \sum_{d=0}^{\psi_j} \sum_{s=0}^{\psi_j-d} \binom{k}{s} \lambda_j^{-s} p_{i(u_j+d)} q_{(u_j+d+s)l} v_l \right) \lambda_j^k. \end{aligned} \tag{3.3.7}$$

The parenthesised sums form the coefficients of an exponential sum; each coefficient is in fact a polynomial in  $k$ , however, before this can be perceived, the expansion of  $\binom{k}{s} = \frac{k!}{(k-s)!s!}$  must be investigated. Consider the following properties:

$$\begin{aligned} \binom{k}{0} &= 1, \quad \binom{k}{1} = k, \\ \binom{k}{2} &= \frac{k(k-1)}{2!} = \frac{1}{2}(k^2 - k) \\ \binom{k}{5} &= \frac{1}{5!}(k^5 - 10k^4 + 35k^3 - 50k^2 + 25k), \text{ and} \\ \binom{k}{s} &= \frac{1}{s!} \prod_{i=0}^{s-1} (k - i). \end{aligned}$$

In general, each binomial coefficient is a polynomial in  $k$  of degree  $s$ ; let the integer coefficient of  $k^i$  in  $\binom{k}{s}$  be given by  $I_{si}$ , so that

$$\binom{k}{s} = \sum_{i=0}^s I_{si} k^i.$$

As an example, consider the binomial coefficient  $\binom{k}{5} = \frac{1}{120}k(k-1)(k-2)(k-3)(k-4)$ : the coefficient of  $k^5$  —  $I_{55}$  — stems from the product of each linear polynomial's 'k' term, and is thus  $\frac{1}{120}1$ . Using similar logic, a term of degree four can be obtained by considering the constant term in one of the linear factors, so that  $I_{54} = \frac{1}{120}(-1-2-3-4) = -\frac{10}{120}$ . In general, explicit formulae for a few coefficients are easily obtainable:

$$\begin{aligned}
 I_{si} &= 0 \text{ if } i < 0 \text{ or } i > s \\
 I_{s0} &= 1 \text{ if } s = 0 \text{ and } 0 \text{ otherwise} \\
 I_{s1} &= (-1)^{s-1} \frac{1}{s!} \prod_{j=1}^{s-1} j = (-1)^{s-1} \frac{1}{s} \\
 I_{ss} &= \frac{1}{s!}, \text{ and} \\
 I_{s(s-1)} &= -\frac{1}{s!} \sum_{j=1}^s j.
 \end{aligned} \tag{3.3.8}$$

The general term  $I_{si}$  can be expressed as a recursive function, best explained combinatorially: firstly, consider that to obtain a term of degree  $i$  from  $\binom{k}{s}$ , the 'k' term from  $i$  factors must be chosen, while  $(s-i)$  of the  $(s-1)$  (non-zero) constant terms remain. There are thus  $\binom{s-1}{s-i}$  terms involving  $k^i$  which must be summed, and the coefficient of each individual term involves the product of any  $(s-i)$  elements of  $\{1, \dots, s-1\}$ .

Making further use of  $\binom{k}{5}$ , note that  $I_{52} = -\frac{1}{120}50$  is the sum of  $\binom{4}{3} = 4$  terms, namely:

$$-\frac{1}{120}(1 \times 2 \times 3), \quad -\frac{1}{120}(1 \times 2 \times 4), \quad -\frac{1}{120}(1 \times 3 \times 4), \quad \text{and} \quad -\frac{1}{120}(2 \times 3 \times 4).$$

Consider that the individual summands which constitute  $I_{si}$ , excluding those which involve the constant  $(s-1)$ , are employed in  $I_{(s-1)(i-1)}$ , albeit led by  $-\frac{1}{(s-1)!}$  instead of  $-\frac{1}{s!}$ . For example,

$$I_{41} = -\frac{1}{24}(1 \times 2 \times 3),$$

which includes the single product  $(1 \times 2 \times 3)$  from  $I_{52}$  which does not contain the integer  $5-1 = 4$ . Furthermore, each of the excluded terms is a product of  $(s-1)$  with some combination of  $(s-i-1)$  elements of  $\{1, \dots, s-2\}$  — the same combinations employed by  $I_{(s-1)i}$ . Consider that

$$I_{42} = \frac{1}{24}(1 \cdot 2 + 1 \cdot 3 + 2 \cdot 3)$$

contains the integer products of  $I_{52}$  which were not apparent in  $I_{41}$ , without the multiple 4.

Intuitively then:

$$I_{si} = \frac{1}{s} I_{(s-1)(i-1)} - \frac{s-1}{s} I_{(s-1)i}. \quad (3.3.9)$$

In a more practical sense, the coefficients  $I_{si}$  shall be calculated for all valid  $s$  and  $i$ , due to their appearances in Equation 3.3.7; as such, the recursive formula (Equation 3.3.9) is sufficient, and, in fact, more valuable than an explicit formula for the current approach. Returning to the exponential sum which represents the value of a given loop variable after  $k$  iterations, and adopting the notation  $\binom{k}{s} = \sum I_{se} k^e$ , Equation 3.3.7 begins to appear slightly daunting:

$$v_i^{[k]} = \sum_{j=1}^t \left( \sum_{l=1}^{n+1} \sum_{d=0}^{\psi_j} \sum_{s=0}^{\psi_j-d} \sum_{e=0}^s I_{se} k^e \lambda_j^{-s} p_{i(u_j+d)} q_{(u_j+d+s)l} v_l \right) \lambda_j^k$$

To represent the polynomial nature of the parenthesised coefficient of  $\lambda_j^k$  more clearly, recall that a binomial coefficient  $\binom{k}{s}$  is a polynomial of degree  $s$ , and that the  $k$ th power of a Jordan block  $Y_j$ , of square dimension  $(\psi_j + 1)$ , contains the binomial coefficients  $\binom{k}{0}, \binom{k}{1}, \dots, \binom{k}{\psi_j}$ . Thus the coefficient of a given  $\lambda_j^k$  is a polynomial in  $k$  of degree  $\psi_j$ . Because  $I_{se} = 0$  if  $e > s$ , the following polynomial sums are equivalent when  $\psi_j \geq s$ :

$$\sum_{e=0}^s I_{se} k^e = \sum_{e=0}^{\psi_j} I_{se} k^e.$$

Hence, because  $s$  never exceeds  $\psi_j$  for a given  $j$ ,

$$\begin{aligned} v_i^{[k]} &= \sum_{j=1}^t \left( \sum_{l=1}^{n+1} \sum_{d=0}^{\psi_j} \sum_{s=0}^{\psi_j-d} \sum_{e=0}^{\psi_j} I_{se} k^e \lambda_j^{-s} p_{i(u_j+d)} q_{(u_j+d+s)l} v_l \right) \lambda_j^k \\ &= \sum_{j=1}^t \left( \sum_{e=0}^{\psi_j} \left( \sum_{l=1}^{n+1} \sum_{d=0}^{\psi_j} \sum_{s=0}^{\psi_j-d} I_{se} \lambda_j^{-s} p_{i(u_j+d)} q_{(u_j+d+s)l} v_l \right) k^e \right) \lambda_j^k, \end{aligned} \quad (3.3.10)$$

and the polynomial coefficient can clearly be seen. Within each polynomial in  $k$ , the coefficient of  $k^e$  is a linear combination of the  $(n+1)$  loop variables over the scalar field  $\mathbb{C}$ .

Lastly:  $G\mathbf{v}^{[k]} > \mathbf{0}$  if, and only if,

$$\sum_{m=1}^{n+1} g_{im} v_m^{[k]} > 0, \quad \forall i = 1, \dots, r,$$

or, combined with Equation 3.3.10:

$$\sum_{m=1}^{n+1} g_{im} \left( \sum_{j=1}^t \left( \sum_{e=0}^{\psi_j} \sum_{l=1}^{n+1} \sum_{d=0}^{\psi_j} \sum_{s=0}^{\psi_j-d} I_{se} \lambda_j^{-s} p_{m(u_j+d)} q_{(u_j+d+s)l} v_l k^e \right) \lambda_j^k \right) > 0, \quad \forall i = 1, \dots, r.$$

A final rearrangement of the terms to elicit the exponential sum yields

$$\sum_{j=1}^t \left( \sum_{e=0}^{\psi_j} \left( \sum_{l=1}^{n+1} \sum_{m=1}^{n+1} \sum_{d=0}^{\psi_j} \sum_{s=0}^{\psi_j-d} g_{im} I_{se} \lambda_j^{-s} p_{m(u_j+d)} q_{(u_j+d+s)l} v_l \right) k^e \right) \lambda_j^k > 0, \quad \forall i = 1, \dots, r,$$

To demonstrate this decomposition, consider the loop in Figure 3.12, whose transformation matrix depends on the variables  $x$  and  $y$ :

$$\begin{aligned} &\text{while } (x + 2y + 1 > 0) \{ \\ &\quad x' := y \\ &\quad y := -x + 2y \\ &\quad x := x' \\ &\} \end{aligned}$$

**Figure 3.12:** A loop which is not diagonalisable.

In a similar manner to that of previous section's example, we obtain

$$\begin{aligned} T^k &= \begin{bmatrix} 0 & 1 & 0 \\ -1 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}^k \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 1 & 0 \end{bmatrix} = PJ^kP^{-1} \\ &= \begin{bmatrix} 0 & 1 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & k \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 1 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 1-k & k & 0 \\ -k & k+1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

Applying the guard matrix  $G = [1 \ 2 \ 1]$  to  $T^k \mathbf{v}$ , the exponential sum is obtained:

$$T^k \mathbf{v} = \begin{bmatrix} (1-k)x + ky \\ -kx + (1+k)y \\ 1 \end{bmatrix}$$

$$A(\mathbf{v}, k) = (1-3k)x + (2+3k)y + 1$$

$$= (x + 2y + 1) + (-3x + 3y)k.$$

This sum is particularly simple — a polynomial in  $k$  of degree one as a coefficient of the only eigenvalue 1. The process for matrices with multiple distinct eigenvalues is similar.

Now, the general form of Lemma 3 [28]:

**Lemma 4.** *An affine loop  $L = (G, T)$  is non-terminating if, and only if, some  $\mathbf{v} \in \mathbb{Z}^{n+1}$  exists such that<sup>10</sup>*

$$A_i(\mathbf{v}, k) = \sum_{j=1}^t C_{ij} \lambda_j^k > 0, \quad \forall k \geq 0, \forall i = 1, \dots, r,$$

where  $G$  has  $r$  rows, and

$$C_{ij} = \sum_{e=0}^{\psi_j} \left( \sum_{l=1}^{n+1} \sum_{m=1}^{n+1} \sum_{d=0}^{\psi_j} \sum_{s=0}^{\psi_j-d} g_{im} I_{se} \lambda_j^{-s} p_{m(u_j+d)} q_{(u_j+d+s)l} v_l \right) k^e.$$

Here  $t$  is the number of Jordan blocks  $Y_1, \dots, Y_t$  in the Jordan canonical form  $J$  of  $T$ ;  $\lambda_j$  is the eigenvalue which populates the diagonal of  $Y_j$ ;  $u_j$  is the index of the first row (and column) of  $Y_j$ ; and  $\psi_j = u_{j+1} - u_j - 1$ .

To relate the result in Lemma 4 to that of diagonalisable affine loops, observe that a diagonal matrix is a Jordan matrix in which each Jordan block has square dimension 1, so that  $t = n + 1$ , and for each  $j = 1, \dots, t$ :  $u_j = j$  and  $\psi_j = 0$ . Then  $C_{ij}$  can easily be reduced, as each polynomial is of degree zero, and is thus a linear combination of the loop variables:

$$C_{ij} = \sum_{e=0}^0 \left( \sum_{l=1}^{n+1} \sum_{m=1}^{n+1} \sum_{d=0}^0 \sum_{s=0}^0 g_{im} I_{00} \lambda_j^0 p_{m j} q_{j l} v_l \right) k^0$$

$$= \sum_{l=1}^{n+1} \sum_{m=1}^{n+1} g_{im} p_{m j} q_{j l} v_l,$$

---

<sup>10</sup>As indicated, the variable arguments of the exponential sum  $A_i$  are the loop variables  $\mathbf{v}$  and an iteration  $k$ ; technically the coefficient  $C_{ij}$  also depends on these arguments, however for the sake of legibility,  $C_{ij}$  is written in place of  $C_{ij}(\mathbf{v}, k)$ . A similar omission shall occasionally be used when inclusion of the arguments adversely affects the clarity of an expression.

that is, the exact equation reflected in Lemma 3.

Considering Lemma 4, it must be noted that each of the  $r$  inequalities can be investigated for non-terminating witnesses independently, and then a final solution obtained by intersecting the individual solutions. Each inequality involves a sum of exponential functions; the properties of this sum are determined by its coefficients, which are in turn deduced from the values of  $v_1, \dots, v_n$ . According to Lemma 4, these sums must remain positive to induce non-termination, however this is a more formidable task than one might realise; the nature of exponential sums is explored in the following section.

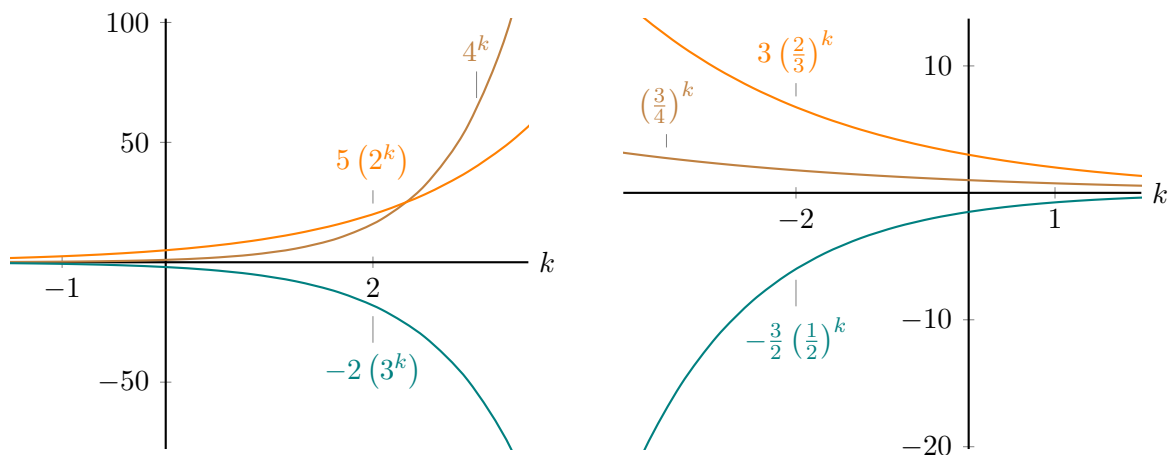
### 3.3.3 Sums of exponential functions

Exponential functions are a familiar sight, and can generally be handled comfortably with the aid of logarithms. However, in the case of exponential sums [27], which sum numerous exponential functions of different bases together, logarithms soon become inapplicable, and the properties of the sum elusive.

As an introduction, take the real-valued exponential function

$$e(k) = C\lambda^k,$$

where  $C, \lambda \in \mathbb{R}$ ,  $\lambda > 0$ , and  $k \in \mathbb{R}$ . Note that  $e(k)$  is monotonic — either constant (if  $\lambda = 1$ ),



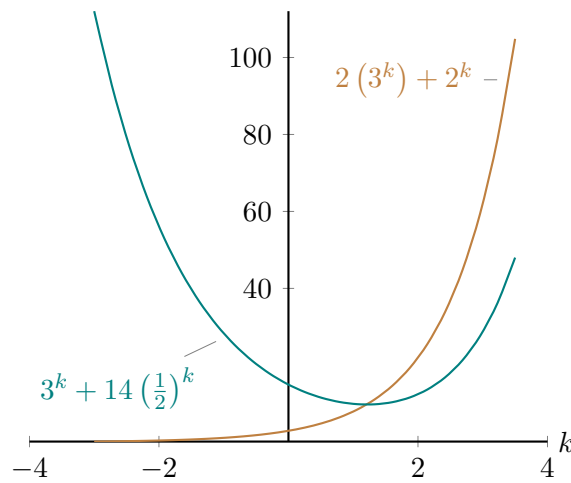
**Figure 3.13:** A sampling of exponential functions.

absolutely decreasing ( $\lambda < 1$ ), or absolutely increasing ( $\lambda > 1$ ). The coefficient term has an equal influence:  $e(k)$  is negative (if  $C < 0$ ), zero ( $C = 0$ ), or positive ( $C > 0$ ) for all  $k$ .

In lieu of Lemma 4, sums of exponential functions which are positive for  $k \geq 0$  are of particular interest; for the singular case, as has already been stated, this reduces to the positivity of the coefficient  $C$ . As a more substantial example of an exponential sum, consider the sum of two functions

$$R(k) = C_1 \lambda_1^k + C_2 \lambda_2^k,$$

where  $C_j, \lambda_j \in \mathbb{R} \setminus \{0\}$ ,  $\lambda_j > 0$ , and  $k \in \mathbb{R}$ ; assume also that  $\lambda_1 > \lambda_2$  (terms with equal bases may be grouped to form a single term).



**Figure 3.14:** Exponential sums with two positive coefficients.

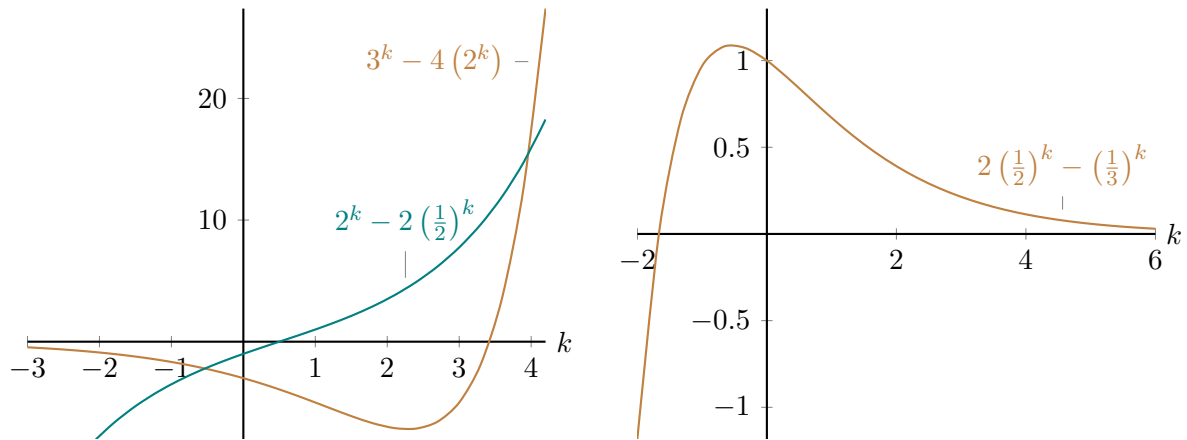
Let  $P(R)$  be the property of positivity for all non-negative values of  $k$ , with regard to an exponential sum  $R(k)$ ; that is:

$$P(R) = (R(k) > 0 \forall k \geq 0). \quad (3.3.11)$$

If  $C_1, C_2 > 0$  (as in Figure 3.14),  $R(k) > 0 \forall k$ , and  $P(R)$  holds trivially. To examine a sum  $R(k)$  which includes mixed signs, and state  $P(R)$  as a condition over the sum's coefficients, first note that  $\lambda_1 > \lambda_2$  implies the dominance of  $C_1 \lambda_1^k$  from some  $K \in \mathbb{R}$  onwards; in fact:

$$\begin{aligned} |C_1 \lambda_1^k| > |C_2 \lambda_2^k| &\Leftrightarrow |C_1| \lambda_1^k > |C_2| \lambda_2^k \\ &\Leftrightarrow \ln |C_1| + k \ln \lambda_1 > \ln |C_2| + k \ln \lambda_2 \\ &\Leftrightarrow k > \frac{\ln \left| \frac{C_2}{C_1} \right|}{\ln \frac{\lambda_1}{\lambda_2}} = K. \end{aligned}$$





**Figure 3.15:** Exponential sums with mixed (sign) coefficients.

$K$  denotes the value of  $k$  at which the terms are equally weighted; thus, if  $C_1 < 0$ ,  $R(k) < 0$  when  $k > K$ , and  $P(R)$  cannot hold. Similarly

$$C_1 > 0 \Rightarrow R(k) > 0, \forall k > K,$$

so that if  $C_1 > 0$  and  $C_2 < 0$ ,

$$R(k) > 0 \Leftrightarrow k > K,$$

and

$$P(R) \Leftrightarrow (K < 0).$$

In fact,  $\lambda_1 > \lambda_2 \Rightarrow \ln \frac{\lambda_1}{\lambda_2} > 0$ , so  $K$  is negative if, and only if,  $\left| \frac{C_2}{C_1} \right| < 1$ ; that is:  $|C_2| < |C_1|$ . Lemma 5 collects these descriptions of positivity.

**Lemma 5.** *Given an exponential sum containing two terms:  $R(k) = C_1\lambda_1^k + C_2\lambda_2^k$ , where  $C_j, \lambda_j \in \mathbb{R} \setminus \{0\}$ ,  $\lambda_1 > \lambda_2 > 0$ , and  $k \in \mathbb{R}$ ; then the positivity property  $P(R)$  holds if, and only if:*

- $(C_1 > 0)$ , and
- $(C_2 > 0 \vee |C_2| < |C_1|)$ , or, equivalently,  $(C_1 + C_2 > 0)$ .

*Proof.*  $R(k)$  is negative from  $K$  onwards if  $C_1 < 0$ . Assume then that  $C_1 > 0$ , and firstly that  $R(0) = C_1 + C_2 > 0$ ; then  $R(k) = C_1\lambda_1^k + C_2\lambda_2^k > (C_1 + C_2)\lambda_2^k > 0 \forall k > 0$ , and  $P(R)$  holds. Alternatively, if  $C_1 + C_2 \leq 0$ , then  $R(0) \leq 0$ , and  $P(R)$  does not hold.

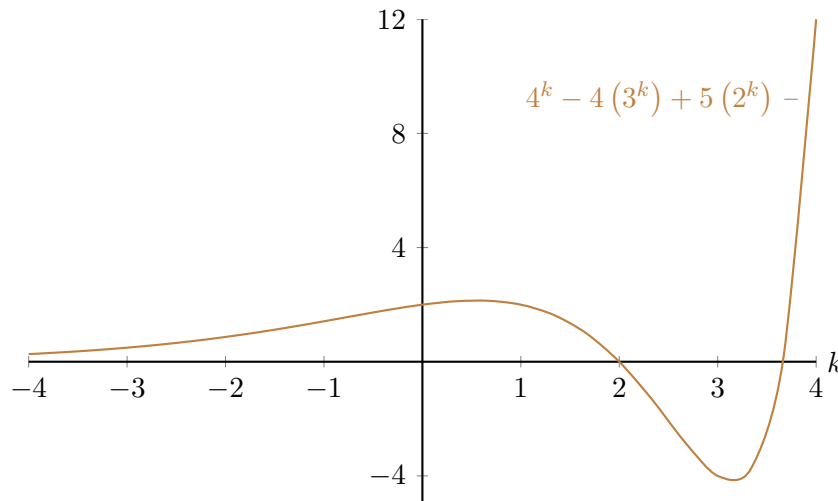
□

Setting aside sums in which both coefficients possess the same sign (as trivial), as well as those with a negative leading coefficient (as eventually negative), the focus is now limited to sums in which  $C_1 > 0$  and  $C_2 < 0$ . Due to the inherent link between the signs of the coefficients within an exponential sum and the sum's positivity, it will be convenient to explicitly represent the sign of each term from this point onwards: let  $D_1 = |C_1|$ ,  $D_2 = |C_2|$ , and

$$B(k) = D_1\lambda_1^k - D_2\lambda_2^k.$$

A sum of the form  $B(k)$  shall be termed a *pair function* [27]; though not explicitly discussed here, such sums are able to portray much of the behaviour present in larger exponential functions [27].

Proceeding to an exponential sum in three terms, it is intriguing to view the manner in which the previous techniques regarding positivity fail; let  $C(k) = 4^k - 4(3)^k + 5(2)^k$ . Notice



**Figure 3.16:** An exponential sum in three parts.

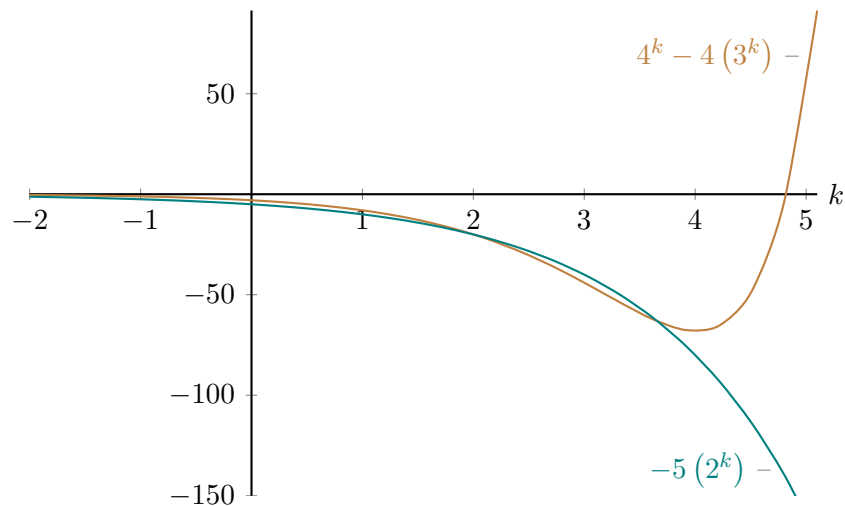
how, in Figure 3.16,  $C(k)$  is dominated at successive intervals by each of its terms  $5(2)^k$ ,  $-4(3)^k$ , and then  $4^k$ . Although, for a pair function, the precise point at which the leading term begins to dominate can be explicitly stated, the same reduction cannot be performed for more general sums, as the inequality (or a similar equality)

$$\left|4^k\right| > \left|-4(3)^k + 5(2)^k\right|$$

contains multiple terms on one side, and thus does not allow the application of logarithms. One could consider each pairing of terms: in this case note that  $|-4|(3)^k > |5|(2)^k$  from  $k \approx 0.55$

onwards, and  $|-4|(3)^k > 4^k$  until  $k \approx 4.82$ ; however, this approach is infeasible for large sums, and only awards regions which must be further inspected — although  $-4(3)^k$  is dominant between 0.55 and 4.82,  $C(k)$  is not negative at  $k = 1$  or  $k = 4$ .

As an alternative approach, one may consider that  $C(k)$  is merely a combination of a pair function and a single exponential function, and then reason that for the sum to remain positive, the pair function must exceed the negative of the singular function; this reasoning is plotted in Figure 3.17. Interpreting Figure 3.17, it is clear that the sum does not remain positive



**Figure 3.17:** An exponential sum of three terms, as a pair function and an exponential function.

for all  $k \geq 0$ ; in fact, the particular range for which positivity is violated is roughly  $[2, 3.7)$ . This interpretation is substantiated by Figure 3.16, as well as Table 3.2, which provides the first few values generated by the sum on positive integers. Such an approach, in which certain terms of the exponential sum are separated, can be useful if it allows each group of terms to be interpreted more easily; in this case, the behaviours of a pair function and an exponential function are more simply described than that of the original sum.

The preceding examples of compact exponential sums are sufficient to convey their intricacy, as well as to highlight the importance of the signs of a sum's coefficients. Knowledge of these signs leads to a representation of the general exponential sum as consecutive sums of explicitly positive and negative terms.

The general exponential sum containing  $m$  terms was initially written as a single sum over

$k$	$4^k - 4(3^k) + 5(2^k)$
0	2
1	2
2	0
3	-4
4	12
5	212

**Table 3.2:** Several values of the function in Figure 3.16 at positive integer intervals.

the coefficients  $C_1, \dots, C_m \in \mathbb{R} \setminus \{0\}$ ; alternatively, letting  $D_i = |C_i|$ , the sum is represented by alternating partial sums:

$$\begin{aligned}
 A(k) &= \sum_{i=1}^m C_i \lambda_i^k \\
 &= \sum_{i=1}^{p_1} D_i \lambda_1^k - \sum_{i=p_1+1}^{m_1} D_i \lambda_1^k + \sum_{i=m_1+1}^{p_2} D_i \lambda_2^k - \dots \\
 &\quad + \sum_{i=m_{s-1}+1}^{p_s} D_i \lambda_s^k - \sum_{i=p_s+1}^{m_s=m} D_i \lambda_s^k,
 \end{aligned} \tag{3.3.12}$$

where the bases  $\lambda_i$  are positive real numbers, and  $\lambda_1 > \lambda_2 > \dots > \lambda_s$ . Each base engenders both a positive and negative partial sum, one (but not both) of which may be empty.

To cause the function in Equation 3.3.12 to remain positive, one might attempt to induce the positive coefficients to outweigh their equally dominant negative counterparts. This idea is further discussed in Section 3.3.5, where the non-termination of an affine loop is reduced to exponential sums similar in form to that of Equation 3.3.12. Linear conditions for the non-termination of loops can in turn be inferred from such sums.

Before this reduction is performed, the known decidability results for the termination of affine loops with integer-valued variables shall be discussed, in an attempt to display the complexity of the problem, and the reason for the lack of a known solution.

### 3.3.4 Proving non-termination

Recall that an affine loop  $L = (G, T)$  in  $(n + 1)$  variables ( $n$  loop variables and a constant 1) is non-terminating if some  $\mathbf{v} \in \mathbb{Z}^{n+1}$  exists such that  $G\mathbf{v}^{[k]} > 0 \forall k \geq 0$ , and that  $G\mathbf{v}^{[k]}$  is reducible to a set of  $r$  exponential sums whose coefficients are determined from the value of  $\mathbf{v}$ . The initial goal, given a loop  $L$ , is to decide whether  $L$  is terminating or non-terminating; in this regard, the existence of some  $\mathbf{v}$  must be asserted, which yields coefficients within the exponential sums such that each of the  $r$  sums is positive for all  $k \geq 0$ .

Consider again Equation 3.3.7, the explicit representation of a loop variable  $v_i$  after  $k$  iterations. The results succeeding Equation 3.3.7 were obtained by expanding  $\binom{k}{s}$  into a polynomial in  $k$ ; however, for this section, it shall be prudent to leave the binomial coefficient in its compact form, so that the  $i$ th row of  $G\mathbf{v}^{[k]}$  may be written

$$A_i(\mathbf{v}, k) = \sum_{m=1}^{n+1} g_{im} \left( \sum_{j=1}^t \sum_{d=0}^{\psi_j} \sum_{s=0}^{\psi_j-d} \binom{k}{s} \lambda_j^{k-s} \sum_{l=1}^{n+1} p_{m(u_j+d)} q_{(u_j+d+s)l} v_l \right).$$

The cascading summand  $\sum_d \sum_s$  can be intuitively viewed as a two-dimensional set of index pairs  $(d, s)$ , summed over rows and columns respectively:

	$s = 0$	1	...	$\psi_j - 1$	$\psi_j$
$d = 0$	$(0, 0)$	$(0, 1)$	...	$(0, \psi_j - 1)$	$(0, \psi_j)$
1	$(1, 0)$	$(1, 1)$	...	$(1, \psi_j - 1)$	
$\vdots$	$\vdots$	$\vdots$	$\ddots$		
$\psi_j - 1$	$(\psi_j - 1, 0)$	$(\psi_j - 1, 1)$			
$\psi_j$	$(\psi_j, 0)$				

In a similar manner, the indices may be summed over columns and rows respectively, yielding an equivalent cascading summand  $\sum_s \sum_d$ ,

$$A_i(\mathbf{v}, k) = \sum_{j=1}^t \sum_{s=0}^{\psi_j} \binom{k}{s} \lambda_j^{k-s} \left( \sum_{d=0}^{\psi_j-s} \sum_{l=1}^{n+1} \sum_{m=1}^{n+1} g_{im} p_{m(u_j+d)} q_{(u_j+d+s)l} v_l \right).$$

Letting

$$D_{ijs}(\mathbf{v}) = \sum_{d=0}^{\psi_j-s} \sum_{l=1}^{n+1} \sum_{m=1}^{n+1} g_{im} p_{m(u_j+d)} q_{(u_j+d+s)l} v_l,$$

we have a sum in which each term is the product of a linear combination  $D_{ijs}(\mathbf{v})$  of the loop variables  $\mathbf{v}$ , a binomial coefficient, and a power of some eigenvalue. Recall that each of the  $j$  terms corresponds to a Jordan block in the Jordan canonical form of  $T$ , and assume then that the terms of  $A_i$  are ordered (renumbering indices where necessary) in a decreasing manner, firstly according to the absolute value of the eigenvalue —  $|\lambda_j|$  — and subsequently according to the dimension of the Jordan block —  $(\psi_j + 1)$ . Furthermore, terms which represent Jordan blocks of equal dimensions, and concerning the same eigenvalue, can be grouped; that is, if  $\lambda_a = \lambda_b$  and  $\psi_a = \psi_b$ , we have

$$\sum_{s=0}^{\psi_a} \binom{k}{s} \lambda_a^{k-s} D_{ias}(\mathbf{v}) + \sum_{s=0}^{\psi_b} \binom{k}{s} \lambda_b^{k-s} D_{ibs}(\mathbf{v}) = \sum_{s=0}^{\psi_a} \binom{k}{s} \lambda_a^{k-s} (D_{ias} + D_{ibs})(\mathbf{v}).$$

Then the exponential sum becomes

$$A_i(\mathbf{v}, k) = \sum_{j=1}^{\tau} \left( \sum_{s=0}^{\psi_j} \binom{k}{s} \lambda_j^{k-s} D_{ijs}(\mathbf{v}) \right), \quad (3.3.13)$$

where grouping results in  $\tau$  terms, and ordering implies that  $|\lambda_1| \geq |\lambda_2| \geq \dots \geq |\lambda_\tau|$ , and if  $|\lambda_a| = |\lambda_b|$ , then  $\psi_a > \psi_b$ .

The dominance hierarchy of terms within  $A_i(\mathbf{v}, k)$  is now apparent: the growth of each term as  $k$  increases is determined by both an exponential function  $\lambda_j^{k-s}$  and  $\binom{k}{s}$  — a polynomial in  $k$  of degree  $s$ , whose growth is in turn determined by the power function  $k^s$ . Exponential functions grow faster than power functions, so that if  $|\lambda_a| > |\lambda_b|$  for two indices  $a$  and  $b$ , the term  $\sum_s \binom{k}{s} \lambda_a^{k-s} D_{ias}(\mathbf{v})$  will eventually be greater in absolute value than  $\sum_s \binom{k}{s} \lambda_b^{k-s} D_{ibs}(\mathbf{v})$ , regardless of the relationship between  $\psi_a$  and  $\psi_b$ ; term  $a$  is then said to dominate term  $b$ .

Within a term  $\sum_s \binom{k}{s} \lambda_a^{k-s} D_{ias}(\mathbf{v})$  are products, generated by the elements of the polynomial  $\binom{k}{s}$  and  $\lambda_a^{k-s}$ . Note that for each  $s$ , the products  $\lambda_a^k, k\lambda_a^{k-1}, k^2\lambda_a^{k-2}, \dots, k^s\lambda_a^{k-s}$  are generated, so that, since  $s = 0, \dots, \psi_a$ , the  $a$ th term of  $A_i$  is a sum over the products  $\lambda_a^k, k\lambda_a^{k-1}, k^2\lambda_a^{k-2}, \dots, k^{\psi_a}\lambda_a^{k-\psi_a}$ . Since  $\lambda_a^{-s}$  is a constant, the product  $k^{\psi_a}\lambda_a^{k-\psi_a}$  grows quickest, so that the  $a$ th summand of  $A_i$  is itself dominated by the highest product which occurs when  $s = \psi_a$ , namely  $I_{\psi_a\psi_a} k^{\psi_a} \lambda_a^{k-\psi_a} D_{ia\psi_a}(\mathbf{v})$ . This also implies that if indices  $a, b$  are such that  $|\lambda_a| = |\lambda_b|$  and  $\psi_a > \psi_b$ , the product involving  $k^{\psi_a}\lambda_a^{k-\psi_a}$  will eventually dominate not only all of the other products  $k^s\lambda_a^{k-s}$  within  $\sum_s \binom{k}{s} \lambda_a^{k-s} D_{ias}(\mathbf{v})$ , but all of the products within  $\sum_s \binom{k}{s} \lambda_b^{k-s} D_{ibs}(\mathbf{v})$  as well.

This discussion allows us to identify the dominant term of  $A_i(\mathbf{v}, k)$ , namely, that which corresponds to the highest-dimension Jordan block of the eigenvalue with greatest absolute value. The exponential sum  $A_i$  is thus dominated by the product  $I_{\psi_1 \psi_1} k^{\psi_1} \lambda_j^{k-\psi_1} D_{ij\psi_1}(\mathbf{v})$ .

Before discussing the non-termination of  $A_i(\mathbf{v}, k)$ , one last property of the exponential sums should be highlighted: the properties possessed by a generated path from some iteration  $\ell$  onwards can be captured by another valid initial variable: since

$$\mathbf{v}^{[\ell+k]} = T^{\ell+k} \mathbf{v} = T^k \mathbf{v}^{[\ell]} = (\mathbf{v}^{[\ell]})^{[k]},$$

it follows that

$$\begin{aligned} A_i(\mathbf{v}, \ell + k) &= \sum_{m=1}^{n+1} g_{im} \mathbf{v}^{[\ell+k]} \\ &= \sum_{m=1}^{n+1} g_{im} (\mathbf{v}^{[\ell]})^{[k]} \\ &= A_i(\mathbf{v}^{[\ell]}, k), \end{aligned} \tag{3.3.14}$$

so then:

$$\sum_{j=1}^{\tau} \sum_{s=0}^{\psi_j} \binom{\ell+k}{s} \lambda_j^{\ell+k-s} D_{ijs}(\mathbf{v}) = \sum_{j=1}^{\tau} \sum_{s=0}^{\psi_j} \binom{k}{s} \lambda_j^{k-s} D_{ijs}(\mathbf{v}^{[\ell]}).$$

The preceding analytic tools are sufficient for the proof of the decidability of the termination of an affine loop when the variables range over the integers, and, crucially, the eigenvalues of the transformation matrix  $T$  are all positive real numbers. It shall be shown that a witness to the non-termination of a loop  $L$  must engender positive leading coefficients in the exponential sums; in fact, the existence of a  $\mathbf{v} \in \mathbb{Z}^{n+1}$  which creates a positive leading term in every exponential sum, regardless of the remaining coefficients, implies the existence of a witness to the loop's non-termination.

**Lemma 6.** *An affine loop  $L = (G, T)$ , such that the  $(n+1) \times (n+1)$  matrix  $T$  has only positive real eigenvalues, is non-terminating if, and only if, a  $\mathbf{v} \in \mathbb{Z}^{[n+1]}$  exists such that, in every  $A_i(\mathbf{v}, k)$ , the leading non-zero coefficient is positive; specifically, that is: for each sum  $A_i$  there exists an index tuple  $(j, s) = (\eta_i, \mu_i)$  such that*

1.  $D_{ijs}(\mathbf{v}) = 0 \quad \forall j < \eta_i,$
2.  $D_{i\eta_i s}(\mathbf{v}) = 0 \quad \forall s > \mu_i,$  and

3.  $D_{i\eta_i\mu_i}(\mathbf{v}) > 0$ .

*Proof.* Firstly, sufficiency is shown: assume that  $\mathbf{v} \in \mathbb{Z}^{n+1}$  satisfies all of the properties; it must be shown that  $L$  is non-terminating. This follows from the fact that  $A_i(\mathbf{v}, k)$  will be dominated from some point onwards by the positive term, and thus the vector at that point is a valid witness to the loop's non-termination. Formally: consider that

$$A_i(\mathbf{v}, k) = \binom{k}{\mu_i} \lambda_{\eta_i}^{k-\mu_i} D_{i\eta_i\mu_i}(\mathbf{v}) + \sum_{s=0}^{\mu_i-1} \binom{k}{s} \lambda_{\eta_i}^{k-s} D_{i\eta_i s}(\mathbf{v}) + \sum_{j=\eta_i+1}^{\tau} \left( \sum_{s=0}^{\psi_j} \binom{k}{s} \lambda_i^{k-s} D_{ijs}(\mathbf{v}) \right),$$

where  $D_{i\eta_i\mu_i}(\mathbf{v}) > 0$ . For large values of  $k$ , the leading term grows faster than the two sums (due to a higher degree polynomial and a greater eigenvalue, respectively); let  $K_i$  be a positive integer such that, for all  $k > K_i$ ,

$$\left| \binom{k}{\mu_i} \lambda_{\eta_i}^{k-\mu_i} D_{i\eta_i\mu_i}(\mathbf{v}) \right| > \left| \sum_{s=0}^{\mu_i-1} \binom{k}{s} \lambda_{\eta_i}^{k-s} D_{i\eta_i s}(\mathbf{v}) + \sum_{j=\eta_i+1}^{\tau} \sum_{s=0}^{\psi_j} \binom{k}{s} \lambda_i^{k-s} D_{ijs}(\mathbf{v}) \right|.$$

Letting  $K$  be the maximum of the  $K_i$  bounds generated by the exponential sums, it follows that  $A_i(\mathbf{v}, (K+1)+l) > 0$  for all  $l \geq 0$ , for every sum. Since  $A_i(\mathbf{v}, (K+1)+l) = A_i(\mathbf{v}^{[K+1]}, l)$ , the vector  $\mathbf{v}^{[K+1]}$  is a witness to the non-termination of  $L$ .

Contrarily, necessity also holds: assume that  $L$  is non-terminating, so that some witness to non-termination  $\mathbf{v} \in \mathbb{Z}^{n+1}$  exists. This implies that every  $A_i(\mathbf{v}, k) > 0 \forall k \geq 0$ ; take the (non-zero) highest degree summand of the largest eigenvalue, say  $(j, s) = (\eta_i, \mu_i)$ , so that  $D_{ijs}(\mathbf{v}) = 0$  for all  $j < \eta_i$ ,  $D_{i\eta_i s}(\mathbf{v}) = 0$  for all  $s > \mu_i$ , and  $D_{i\eta_i\mu_i}(\mathbf{v}) \neq 0$ . Then this term dominates  $A_i(\mathbf{v}, k)$  for large  $k$ , as in the argument for sufficiency. If  $D_{i\eta_i\mu_i}(\mathbf{v})$  were negative, the sum would eventually become negative, contradicting the loop's non-termination; thus  $D_{i\eta_i\mu_i}(\mathbf{v}) > 0$ , and the lemma's properties hold. □

It is important to note that the proof of sufficiency in Lemma 6 does not assert that a  $\mathbf{v}$  which satisfies the stated properties is itself a witness to non-termination; instead it is shown that some iteration of this vector, when used as an initial value, generates an infinite path within  $L$ .

The crucial assumption for Lemma 6 was that the eigenvalues of  $T$  were positive elements of  $\mathbb{R}$ . In fact, the lemma generalises to include the eigenvalue 0 rather naturally, since terms which



correspond to this eigenvalue are zero except possibly when the eigenvalue term  $\lambda^{k-s} = 0^{k-s}$  has a non-positive exponent. Since this can only happen for the first  $s$  iterations, the iteration bound  $K$  in Lemma 6 must be larger than  $s$ , in which case the terms generated by the zero eigenvalue are of no consequence.

It is not always the case that  $T$  initiates non-negative, or even real eigenvalues: complex eigenvalues are a common consequence of real matrices; however, when complex eigenvalues are allowed, the preceding proof approach can only be completed when the variables of  $L$  are allowed to adopt real, or at least rational, values. The complication introduced by negative and complex eigenvalues is that, for a non-terminating  $\mathbf{v}$ , the exponential sum  $A_i$  is no longer necessarily dominated from some point onwards; it is possible that each set of absolutely similar terms can remain positive without dominating the sum, for example, consider the loop in Figure 3.18.

```

while ( $x > 0$ ) {
     $x := 5x + y + z$ 
     $y' := 4y + 3z$ 
     $z := -3y + 4z$ 
     $y := y'$ 
}
    
```

**Figure 3.18:** A loop whose exponential sum is not dominated by the leading term.

This loop induces the transformation matrix

$$T = \begin{bmatrix} 5 & 1 & 1 \\ 0 & 4 & 3 \\ 0 & -3 & 4 \end{bmatrix}$$

along with the exponential sum

$$A(\mathbf{v}, k) = x^{[k]} = \left(x - \frac{1}{5}y + \frac{2}{5}z\right) 5^k + \left(\frac{1}{10}y - \frac{1}{5}yi - \frac{1}{5}z - \frac{1}{10}zi\right) (4 + 3i)^k + \left(\frac{1}{10}y + \frac{1}{5}yi - \frac{1}{5}z + \frac{1}{10}zi\right) (4 - 3i)^k,$$

in which each of the three terms carries an equal weight, since the eigenvalues all have an absolute value of 5. Interestingly enough, it was this loop, provided to the authors by a

colleague, which first proved that not all affine loops are periodically monotonic over some period. This is true since the complex number  $(4 + 3i)$ , as well as its conjugate, have irrational arguments, and thus are not equal to  $-5^k$  (when the value intersects the negative real axis with an argument of  $\pi$ ) for any power  $k$ ; however, powers of these eigenvalues do come arbitrarily close to  $-5^k$  over non-uniform periods. Notice how, when  $(x, y, z) = (2, 1, -2)$ ,

$$A(\mathbf{v}, k) = 5^k + \frac{1}{2}(4 + 3i)^k + \frac{1}{2}(4 + 3i)^k. \quad (3.3.15)$$

Since  $(4 + 3i)^k$  comes arbitrarily close to  $-5^k$  infinitely often, the sum often comes close to 0, without ever reaching it. A similar concept can alternatively be seen in the exponential sum  $4^k + (-4)^k + 3^k$ , which is dominated by the terms with absolute value 4 whenever  $k$  is even, but by  $3^k$  for odd  $k$ .

Thus the approach taken for loops with positive eigenvalues does not generalise. Instead, when faced with eigenvalues other than non-negative real numbers, it can be concluded that some witness  $\mathbf{w}$  exists which is non-terminating when only the positive real eigenvalues of  $T$  are considered; and then, using  $\mathbf{w}^{[K]}$  as an anchor, that some vector  $\mathbf{w}^{[K]} + \epsilon\mathbf{v} \in \mathbb{R}^{n+1}$  exists, near to  $\mathbf{w}^{[K]}$ , which does engender this dominating behaviour in, and is thus a witness to the non-termination of,  $L$  [8]. Note that only the existence of a real-valued witness is asserted; deciding the termination of an affine loop  $L$  over the integers is a much harder problem, and seems to require the ability to decide the positivity of an exponential sum  $A_i(\mathbf{v}, k)$  for all  $k \geq 0$ , for a given  $\mathbf{v} \in \mathbb{Z}^{n+1}$  [8]. It is true that termination is decidable over the rational numbers, and thus over the integers in the case where the loop's guard is bounded by zero (of the form  $G\mathbf{v} > 0$ , where  $G$  has not been augmented with a boundary column  $\mathbf{b}$ ), however the logic used to obtain a rational witness involves the investigation of loops of decreasing size, and is not subsumed by the techniques presented in Section 3.3.5. For interest's sake, note that if a rational witness to the non-termination of an affine loop with a zero-bounded guard condition is known, then an integer solution can be obtained by scaling the witness. Since the guard condition's boundary is  $\mathbf{0}$ , it is unaffected by the scaling, and thus the integer solution is a valid infinite path generator.

Although the decidability of affine loops with real and rational variables is known, these results are of lesser practical value without some form of application to affine loops over the integers, since non-integer values are not represented with infinite precision by computer systems. This implies that a real witness which lies arbitrarily close to some other real initial value may

not be representable by the system, unless  $\mathbb{R}$ , in its true sense, is available. In addition, one would surmise that the vast majority of affine loops are constructed with integer variables, in which case decidability is not yet known, other than in the positive eigenvalue case presented above.

The two ideas which can be taken from this discussion are: if an affine loop can be constructed such that its eigenvalues are non-negative, its termination can automatically be decided; and perhaps the coefficient-based approach to proving decidability can be adapted to a more general search for non-terminating witnesses, albeit without a guarantee of falsification even if a loop is non-terminating. The second concept is the basis of the main result of this text, and shall be discussed in the following section. With regard to the creation of loops whose eigenvalues are non-negative, it is interesting to note that a symmetric matrix is positive semi-definite if, and only if, all of its eigenvalues are non-negative. A real matrix  $A$  is symmetric if  $a_{ij} = a_{ji}$  for each of its entries  $a_{ij}$ , and is positive semi-definite if, for every non-zero real column vector  $\mathbf{z}$ , ( $\mathbf{z}^T A \mathbf{z} \geq 0$ ). In the case of single-variable loops, the transformation matrix is of the form

$$\begin{bmatrix} t_1 & t_2 \\ 0 & 1 \end{bmatrix},$$

and for such a matrix to be symmetric it must hold that  $t_2 = 0$ . Furthermore, since the entries along a diagonal matrix's diagonal are its eigenvalues, such a symmetric matrix is positive semi-definite if  $t_2 \geq 0$ ; this equates to an update transformation of the form  $v := t_2 v$ . Generally, however, it does not seem viable to attempt to create loops which engender only positive eigenvalues. Note that the inclusion of a constant term in any of the loop's update transformations prohibits the transformation matrix  $T$  from being symmetric.

As an aside, it should be noted that the known results regarding the decidability of affine loop termination exclude the possibility of loop preconditions — restrictions which may have been applied to the loop variables before the loop has been reached. Considering that deciding termination (over the reals and rationals) involves the assertion that although some vector  $\mathbf{v}$  might not be a witness to non-termination, its value after some number of iterations will be a witness, a loop which is declared non-terminating might be terminating in the presence of a well-contrived precondition. A loop which is verified as terminating by such an algorithm, however, remains so when preconditions are present: as if no non-terminating witnesses exist

for the ostricised loop, a subset of valid initial vectors cannot possess a witness.

Practically, preconditions are a common property of program loops, and since affine loop termination has not been shown decidable in their presence, a combination of verification and falsification techniques seems apt. The approach to falsification (that is, searching for witnesses to non-termination) presented in the following section continues naturally from the algebraic methods which have presently been discussed.

### 3.3.5 Approximating the set of non-termination witnesses

In this section, the general exponential sums  $A_i(\mathbf{v}, k)$  generated during the Jordan decomposition of a loop shall be abstracted to a form similar to that of the sums discussed in Section 3.3.3. This is done by obtaining a sum of the desired form which is always less than or equal to  $A_i(\mathbf{v}, k)$ ; this new sum allows for the simple extraction of linear constraints which describe its positivity, and thus imply non-termination.

Recalling Lemma 4, the goal is to find vectors  $\mathbf{v} \in \mathbb{Z}^{n+1}$  which force each of the exponential sums generated by the loop's guard condition  $G$  to remain positive for all iterations  $k \geq 0$ . Because these sum inequalities cannot be solved explicitly, even in the case of loops with diagonalisable transformation matrices, it remains to locate as many witnesses to non-termination as possible. Note that two descriptions of the relevant exponential sums have so far been given (firstly in Lemma 4, secondly as Equation 3.3.13):

$$\begin{aligned} A_i(\mathbf{v}, k) &= \sum_{j=1}^t C_{ij}(\mathbf{v}, k) \lambda_j^k \\ &= \sum_{j=1}^{\tau} \left( \sum_{s=0}^{\psi_j} \binom{k}{s} \lambda_j^{k-s} D_{ijs}(\mathbf{v}) \right) \end{aligned}$$

The former representation allows coefficients to be regarded elegantly, as polynomials, whereas the latter contains a secondary sum including binomial coefficients. Hence, for the remainder of this chapter, and specifically to devise a method of loop falsification, the first depiction of an exponential sum — introduced in Lemma 4 — will be adopted. This depiction is, intuitively, a sum over eigenvalue powers in which each coefficient  $C_{ij}(\mathbf{v}, k)$  is a polynomial in the iteration  $k$ , and the coefficients of the polynomial terms are themselves linear combinations of the loop variables  $\mathbf{v}$ . Similar to the simplification of Equation 3.3.13, the coefficients corresponding to Jordan blocks of equal eigenvalues can be grouped (renaming indices where necessary) by

summing the individual coefficients within the polynomial, yielding an exponential sum whose bases are unique and ordered according to decreasing absolute value:

$$A_i(\mathbf{v}, k) = \sum_{j=1}^{\rho} C_{ij}(\mathbf{v}, k) \lambda_j^k,$$

in which  $\lambda_j = \lambda_l \Leftrightarrow j = l$ , and  $|\lambda_1| \geq |\lambda_2| \geq \dots \geq |\lambda_\rho|$ ; complex eigenvalues are once again considered.

Since, as has already been stated, reducing the positivity of such a sum to complete conditions on the coefficients is not feasible (when  $\rho > 2$ ), a heuristic solution is proposed. This solution combines a set of coefficient properties which induce positivity, and subsequently produces loop variable constraints which can be solved to obtain non-terminating witnesses. Recall the representation of an arbitrary exponential sum  $A(k)$ , with real-valued coefficients and bases, as a sum of alternating partial sums (Equation 3.3.12):

$$A(k) = \sum_{i=1}^{p_1} D_i \lambda_i^k - \sum_{i=p_1+1}^{m_1} D_i \lambda_i^k + \sum_{i=m_1+1}^{p_2} D_i \lambda_i^k - \dots + \sum_{i=m_{s-1}+1}^{p_s} D_i \lambda_i^k - \sum_{i=p_s+1}^{m_s=m} D_i \lambda_i^k.$$

This representation is extremely useful, however, it relies on the explicit knowledge of coefficient signs, and, since  $A_i(\mathbf{v}, k)$  may contain complex coefficients, such a form cannot easily be obtained for the exponential functions at hand. For this reason, the exponential sum  $A_i(\mathbf{v}, k)$  shall be abstracted to a form which consists of only positive real eigenvalues combined with explicitly signed real coefficients. The abstraction rests upon the idea that terms corresponding to positive eigenvalues contribute positively to the sum (as long as the coefficient is also positive), whilst terms of negative or complex eigenvalues contribute negatively (regardless of the coefficient's sign). To deduce this, consider each case independently: firstly, let  $\lambda_p > 0$ , with a corresponding term  $C_{ip} \lambda_p^k$ . Note that every power of a positive eigenvalue is positive —  $\lambda_p^k > 0 \forall k$  — hence the sign of  $C_{ip} \lambda_p^k$  is solely determined by the sign of the coefficient  $C_{ip}$ , which is real since  $\lambda_p \in \mathbb{R}$  (Appendix A.3).

Secondly, let  $\lambda_n < 0$ , within the full term  $C_{in} \lambda_n^k$ . A power of a negative eigenvalue is positive for even exponents, and negative otherwise, and thus contributes negatively for infinitely many positive values of  $k$  whether  $C_{in}$  is positive (in which case the term is negative for odd  $k$ ) or negative (even  $k$ , respectively).

Lastly, if  $\lambda_c \in \mathbb{C} \setminus \mathbb{R}$ , consider the term  $C_{ic} \lambda_c^k$ : similarly to the case of negative eigenvalues, such a term will contribute negatively for infinitely many positive values of  $k$ . Recall that if

$\lambda_c$  has a complex argument  $\theta$  (in radians), then  $\lambda_c^k = |\lambda_c| e^{ik\theta}$ , and raising  $\lambda_c$  to a power is equivalent to a relevant rotation by  $\theta$  and scaling by  $|\lambda_c|$ . Thus,  $\text{Re}(\lambda_c^k)$  is negative for infinitely many positive  $k$ , and so is  $C_{ic}\lambda_c^k$  (if  $C_{ic} \neq 0$ ), as the complex coefficient  $C_{ic}$  always adjusts the argument of  $\lambda_c^k$  by a fixed amount.

The abstraction of  $A_i(\mathbf{v}, k)$  is in fact nothing but a lower bound, which captures the worst-case behaviour (negative contribution) of each term. As an example, note that for a negative eigenvalue  $\lambda_n$ ,

$$C_{in}\lambda_n^k \geq -|C_{in}||\lambda_n|^k.$$

However, the presence of the absolute value of a polynomial in  $k$  is undesired; assuming the sign of  $C_{in}$  is known, a lower bound can be obtained whilst avoiding the use of the coefficient's absolute value; let the function  $\Lambda$  define a lower bound for a term in an exponential sum whose base  $\lambda_n$  is negative:

$$\Lambda\left(C_{in}\lambda_n^k\right) = \begin{cases} -C_{in}|\lambda_n|^k & \text{if } C_{in} > 0, \\ C_{in}|\lambda_n|^k & \text{if } C_{in} \leq 0. \end{cases} \quad (3.3.16)$$

The inequality  $C_{in}\lambda_n^k \geq \Lambda(C_{in}\lambda_n^k)$  then holds for all  $k \geq 0$ . Simply put, if the coefficient is positive, its negation is used in the abstraction. The goal then, is to constrain the signs of the coefficients within an exponential sum, and for each permutation of signs, abstract the sum by making use of lower bounds. This yields an alternating exponential sum over real coefficients and bases, which can then be used to search for witnesses to the non-termination of the loop. Soundness of the heuristic follows from the fact that each abstraction is a lower bound, and thus if the abstracted sum is positive for all  $k \geq 0$ , positivity is also satisfied by the exponential sum.

Before the abstraction is complete, lower bounds must be obtained for complex eigenvalues. This case is slightly more complicated, the key tool in simplification being that complex eigenvalues appear in conjugate pairs. It is shown in Appendix A.3 that if  $\lambda_d = \overline{\lambda_c}$ , the pair of corresponding coefficients in  $A_i(\mathbf{v}, k)$  is also conjugate:  $C_{id} = \overline{C_{ic}}$ ; and subsequently that

$$C_{ic}\lambda_c^k + C_{id}\lambda_d^k = 2\text{Re}\left(C_{ic}\lambda_c^k\right).$$

Just as  $-|\lambda_n|^k$  is a lower bound for the powers of a negative eigenvalue  $\lambda_n$ , the real part of a complex number  $\lambda_c^k$  cannot exceed its absolute value, i.e.,  $\text{Re}(\lambda_c^k) \geq -|\lambda_c^k|$ , and similarly for

the imaginary part. Note now that

$$\operatorname{Re}\left(C_{ic}\lambda_c^k\right) = \operatorname{Re}\left(C_{ic}\right) \operatorname{Re}\left(\lambda_c^k\right) - \operatorname{Im}\left(C_{ic}\right) \operatorname{Im}\left(\lambda_c^k\right),$$

and

$$\begin{aligned} -|\lambda_c|^k &\leq \operatorname{Re}\left(\lambda_c^k\right) \leq |\lambda_c|^k, \\ -|\lambda_c|^k &\leq \operatorname{Im}\left(\lambda_c^k\right) \leq |\lambda_c|^k. \end{aligned}$$

These properties allow for the definition of lower bounds which again do not require the absolute value of the coefficient. In this case, the explicit sign of both the real and imaginary part of the coefficient must be known, so that the lower bound function  $\Lambda(2\operatorname{Re}(C_{ic}\lambda_c^k))$  can be extended to the complex terms of the sum as follows:

$$\Lambda\left(2\operatorname{Re}\left(C_{ic}\lambda_c^k\right)\right) = \begin{cases} 2(-\operatorname{Re}(C_{ic}) - \operatorname{Im}(C_{ic}))|\lambda_c|^k & \text{if } \operatorname{Re}(C_{ic}), \operatorname{Im}(C_{ic}) > 0, \\ 2(\operatorname{Re}(C_{ic}) + \operatorname{Im}(C_{ic}))|\lambda_c|^k & \text{if } \operatorname{Re}(C_{ic}), \operatorname{Im}(C_{ic}) \leq 0, \\ 2(-\operatorname{Re}(C_{ic}) + \operatorname{Im}(C_{ic}))|\lambda_c|^k & \text{if } \operatorname{Re}(C_{ic}) > 0, \operatorname{Im}(C_{ic}) \leq 0, \\ 2(\operatorname{Re}(C_{ic}) - \operatorname{Im}(C_{ic}))|\lambda_c|^k & \text{if } \operatorname{Re}(C_{ic}) \leq 0, \operatorname{Im}(C_{ic}) > 0. \end{cases} \quad (3.3.17)$$

so that  $2\operatorname{Re}(C_{ic}\lambda_c^k) \geq \Lambda(2\operatorname{Re}(C_{ic}\lambda_c^k))$  for all non-negative  $k$ , and all (sign-constrained) coefficients  $C_{ic}$ .

The last possible eigenvalue which must be discussed is an exception: that of the eigenvalue zero. Once again recalling the alternative representation of  $A_i(\mathbf{v}, k)$ , in which a Jordan block's term is of the form

$$\sum_{s=0}^{\psi_j} \binom{k}{s} D_{ijs}(\mathbf{v}) \lambda_j^{k-s},$$

note that if  $\lambda_j = 0$ , then  $k > s \Rightarrow \lambda_j^{k-s} = 0$ ; if  $k < s$ , the term is also zero, since  $s$  considers entries of the Jordan matrix power  $J^k$  which are removed from the diagonal, and these terms are zero when  $k < s$ . In the case where  $k = s$ ,  $\binom{s}{s} \lambda_j^0$  provides a value of 1, so that  $D_{ijs}(\mathbf{v})$  remains. Since  $s$  assumes  $(\psi_j + 1)$  different values, there are at most  $(\psi_j + 1)$  values of  $k$  for which the zero-eigenvalue term is non-zero. The minimum value of a term in which  $\lambda_j = 0$  is thus the minimum of the linear combinations  $\{D_{ijs}; s = 0, \dots, \psi_j\}$ . Unfortunately, since the minimum of these terms differs according to the values of the loop variables, a single minimum term cannot explicitly be chosen. Instead, it is sufficient to note that the term generated by a

Jordan block of dimension  $(\psi_j + 1)$  and eigenvalue zero is zero for all  $k > \psi_j$ , and to apply the lower bound abstraction to the exponential sum when  $k > \psi_j$ , whilst explicitly ensuring that the exponential sum is positive for  $0 \leq k \leq \psi_j$ . Figure 3.19 depicts a loop which generates the

$$\begin{aligned} &\text{while } (x > 0) \{ \\ &\quad x := x + y + 2 \\ &\quad y := -x \\ &\} \end{aligned}$$

**Figure 3.19:** A loop which engenders the eigenvalue zero.

eigenvalues 1 and 0, along with the Jordan matrix

$$J = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \Rightarrow J^k = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0^k & 0^{k-1} \\ 0 & 0 & 0^k \end{bmatrix}$$

The reader is asked to accept  $0^k$  as a shorthand depiction of Kronecker's delta function, which is 0 unless  $k = 0$ , in which case it is 1. The exponential sum generated by the loop at hand is

$$A(\mathbf{v}, k) = x^{[k]} = 2(1)^k + (x - 2)0^k + (x + y)0^{k-1},$$

so that  $x^{[k]} = 2$  for all  $k > 1$  (this is also directly inferable from Figure 3.19). Note how the dimension of the zero eigenvalue's Jordan block is 2, so that the corresponding  $\psi_2 = 1$  (recall that  $\psi_j$  was defined for Equation 3.3.7 as one less than the dimension of the Jordan block  $Y_j$ ), and by the previous discussion the terms  $x^{[0]}$  and  $x^{[1]}$  must be explicitly constrained, so that  $A(\mathbf{v}, k)$  can be considered from  $k = 2$  onwards, without the zero eigenvalue's terms.

Before continuing to the description of a lower bound for  $A(\mathbf{v}, k)$ , it is prudent to discuss the meaning of the expressions  $C_{in} > 0$  and  $\text{Re}(C_{ic}) > 0$ . Consider firstly  $C_{in}(\mathbf{v}, k)$ : in the case of diagonalisable loops, which inspired this approach, such a coefficient is a linear combination of the loop variables  $\mathbf{v}$ , independent of  $k$ , and the constraint is naturally defined as for real numbers. When the transformation matrix of the loop at hand is not diagonalisable, however,  $C_{in}(\mathbf{v}, k)$  is a polynomial in  $k$ , whose coefficients are themselves linear combinations of the entries in  $\mathbf{v}$ . In this case, the constraint  $C_{in} > 0$  is an abbreviated form of  $C_{in}(\mathbf{v}, k) > 0 \forall k \geq 0$ , or, if some  $\zeta$  is given,  $\forall k \geq \zeta$  (if a zero eigenvalue is present,  $A_i(\mathbf{v}, k)$  will only be



constrained from some  $k = \zeta$  onwards, where  $\zeta$  is the dimension of the largest Jordan block with a zero eigenvalue). The lower bound abstraction thus requires coefficients to remain either positive or non-positive, unfortunately excluding certain variable configurations. For example, if some assignment of values to a certain loop's variables engendered the polynomial coefficient  $(1 - 4k + 2k^2)$ , which acquires both positive and negative values (considering non-negative  $k$ , the polynomial is positive except for  $k = 1$ , when it is negative), this configuration of loop variables would not be incorporated by the abstracted sum. Fortunately, constraining polynomials in this way can still incorporate numerous configurations of  $\mathbf{v}$ , especially when polynomial coefficients are of a low degree. This approach was adopted because the signs of coefficients are vitally important to the behaviour of the exponential sum (the extraction of linear constraints which describe the sum's positivity require knowledge of the coefficients' signs); making use of a coefficient's absolute value to obtain a lower bound is not possible, since the absolute value function is not distributive, and thus a constraint involving the absolute value of a polynomial, or even a linear combination, cannot be reduced to simple constraints regarding the variables involved.

Aside from the real constraint  $C_{in} > 0$ , take the complex coefficient  $C_{ic}$ , whose lower bound is a reduction to inequalities over the real and imaginary parts of the complex value. If

$$C_{ic} = (\mathbf{c}_0 \mathbf{v}) + (\mathbf{c}_1 \mathbf{v})k + \cdots + (\mathbf{c}_s \mathbf{v})k^s,$$

then, using the fact that the dot product  $\mathbf{c}_j \mathbf{v}$  is a summation,

$$\begin{aligned} \operatorname{Re}(C_{ic}) &= \operatorname{Re}((\mathbf{c}_0 \mathbf{v})) + \operatorname{Re}((\mathbf{c}_1 \mathbf{v})k) + \cdots + \operatorname{Re}((\mathbf{c}_s \mathbf{v})k^s) \\ &= \operatorname{Re}(\mathbf{c}_0 \mathbf{v}) + \operatorname{Re}(\mathbf{c}_1 \mathbf{v})k + \cdots + \operatorname{Re}(\mathbf{c}_s \mathbf{v})k^s, \end{aligned}$$

and  $\operatorname{Re}(C_{ic})$ ,  $\operatorname{Im}(C_{ic})$  are themselves real-valued polynomials, so that inequalities of the form  $\operatorname{Re}(C_{ic}) > 0$  are no more than abbreviations of  $\operatorname{Re}(C_{ic}(\mathbf{v}, k)) > 0 \forall k \geq \zeta$ . Note that, unlike the absolute value function, the  $\operatorname{Re}$  and  $\operatorname{Im}$  operators are distributive.

The somewhat lengthy preceding discussion leads to the definition of a lower bound for an exponential sum  $A_i(\mathbf{v}, k)$ , assuming that the determination of the signs of the sum's coefficients is a solvable problem:

**Lemma 7.** *Given an exponential sum  $A_i(\mathbf{v}, k)$ , engendered by some affine loop  $L = (G, T)$ ; if the real coefficients, as well as the real and imaginary parts of every complex coefficient,*

maintain their signs for all non-negative  $j$ , then an alternating exponential sum of the form

$$B_i(\mathbf{v}, k) = \sum_{j=1}^{p_{i1}} \Lambda_j(\mathbf{v}, k) \lambda_1^k - \sum_{j=p_{i1}+1}^{m_{i1}} \Lambda_j(\mathbf{v}, k) \lambda_1^k + \dots + \sum_{j=m_{i(s_i-1)}+1}^{p_{is_i}} \Lambda_j(\mathbf{v}, k) \lambda_{s_i}^k - \sum_{j=p_{is_i}+1}^{m_{is_i}=m_i} \Lambda_j(\mathbf{v}, k) \lambda_{s_i}^k,$$

exists, such that  $A_i(\mathbf{v}, k) \geq B_i(\mathbf{v}, k) \forall k \geq \zeta$ , where  $\zeta$  is the dimension of the largest Jordan block corresponding to a zero eigenvalue; each  $\lambda > 0$ , and each  $\Lambda_j(\mathbf{v}, k)$  is a real-valued polynomial in  $k$  whose coefficients are linear combinations of the entries in  $\mathbf{v}$ , such that  $\Lambda_j(\mathbf{v}, k) > 0 \forall k \geq 0$ . In addition, the terms are ordered so that  $\lambda_1 > \dots > \lambda_{s_i}$ .

*Proof.* Let terms with zero-valued coefficients be ignored, complex terms within  $A_i(\mathbf{v}, k)$  be combined with their conjugate terms according to the equation  $C_{ic}\lambda_c^k + \overline{C_{ic}\lambda_c^k} = 2\operatorname{Re}(C_{ic}\lambda_c^k)$ , and terms with equal bases be grouped ( $C_{ia}\lambda_a^k + C_{ib}\lambda_a^k = (C_{ia} + C_{ib})\lambda_a^k$ ); then  $A_i(\mathbf{v}, k)$  becomes a sum of  $\rho_i$  terms. Then, renaming grouped coefficients where necessary, each term in the sum is of the form  $C_{il}\lambda_l^k$  or  $2\operatorname{Re}(C_{il}\lambda_l^k)$ , if  $\lambda_l \in \mathbb{R}$  or  $\lambda_l \in \mathbb{C} \setminus \mathbb{R}$  respectively. Since iterations  $k \geq \zeta$  are considered, terms with zero-valued eigenvalues may be ignored. For each term  $C_{il}\lambda_l^k$ , add (for positive eigenvalues with positive coefficients), or subtract (for terms with possibly negative contributions) a term  $\Lambda_j(\mathbf{v}, k) |\lambda_j|^k$  to/from  $B_i(\mathbf{v}, k)$ , where  $\Lambda_j(\mathbf{v}, k) = |C_{il}|$ . Similarly, for each term  $2\operatorname{Re}(C_{il}\lambda_l^k)$ , let the two terms described by  $\Lambda(2\operatorname{Re}(C_{il}\lambda_l^k))$  be subtracted from  $B_i(\mathbf{v}, k)$ , corresponding to the signs of the real and imaginary parts of  $C_{il}$ . For example, if a term  $2\operatorname{Re}(C_{il}\lambda_l^k)$  is such that  $\operatorname{Re}(C_{il}) \leq 0$  and  $\operatorname{Im}(C_{il}) > 0$ , then  $\Lambda(2\operatorname{Re}(C_{il}\lambda_l^k)) = (\operatorname{Re}(C_{il}) - \operatorname{Im}(C_{il})) |\lambda_l|^k$ , and the positive terms  $(-\operatorname{Re}(C_{il}) |\lambda_l|^k)$  and  $\operatorname{Im}(C_{il}) |\lambda_l|^k$  are subtracted from  $B_i(\mathbf{v}, k)$ . Then, noting that the bases in  $B_i(\mathbf{v}, k)$  are the absolute values of the bases in  $A_i(\mathbf{v}, k)$ , each of the  $s$  bases in  $B_i(\mathbf{v}, k)$  provide a sum of positive contributions (from positive eigenvalues) and a sum of negative contributions (from negative/complex eigenvalues), and  $B_i(\mathbf{v}, k)$  is a series of  $2s$  alternating partial sums of the desired form. Also, since each term (or pair of terms) in  $B_i(\mathbf{v}, k)$  is a lower bound for a term in  $A_i(\mathbf{v}, k)$ , and every term of  $A_i(\mathbf{v}, k)$  is so bounded, the result follows.  $\square$

As an example of the construction of a lower bound  $B_i(\mathbf{v}, k)$ , consider again the loop in

Figure 3.18, which engenders the exponential sum

$$\begin{aligned} A(\mathbf{v}, k) = x^{[k]} &= \left(x - \frac{1}{5}y + \frac{2}{5}z\right) 5^k + \left(\frac{1}{10}y - \frac{1}{5}yi - \frac{1}{5}z - \frac{1}{10}zi\right) (4 + 3i)^k \\ &\quad + \left(\frac{1}{10}y + \frac{1}{5}yi - \frac{1}{5}z + \frac{1}{10}zi\right) (4 - 3i)^k. \end{aligned}$$

Impose the restriction that each of the required coefficients be positive:

$$\begin{aligned} C_1(\mathbf{v}) &= \left(x - \frac{1}{5}y + \frac{2}{5}z\right) > 0; \\ \operatorname{Re}(C_2(\mathbf{v})) &= \left(\frac{1}{10}y - \frac{1}{5}z\right) > 0; \text{ and} \\ \operatorname{Im}(C_2(\mathbf{v})) &= \left(-\frac{1}{5}y - \frac{1}{10}z\right) > 0. \end{aligned}$$

Then the lower bounds are as follows:

$$\Lambda\left(C_1(\mathbf{v})5^k\right) = C_1(\mathbf{v}),$$

and

$$\begin{aligned} \Lambda\left(2 \operatorname{Re}\left(C_2(\mathbf{v})(4 + 3i)^k\right)\right) &= 2\left(-\operatorname{Re}(C_2(\mathbf{v})) - \operatorname{Im}(C_2(\mathbf{v}))\right) |4 + 3i|^k \\ &= \left(-\frac{1}{5}y + \frac{2}{5}z + \frac{2}{5}y + \frac{1}{5}z\right) 5^k \\ &= \left(\frac{1}{5}y + \frac{3}{5}z\right) 5^k. \end{aligned}$$

Then, for this particular permutation of coefficient signs, the lower bound sum for  $A(\mathbf{v}, k)$  is given by

$$\begin{aligned} B(\mathbf{v}, k) &= \left(x - \frac{1}{5}y + \frac{2}{5}z\right) 5^k + \left(\frac{1}{5}y + \frac{3}{5}z\right) 5^k \\ &= (x + z)5^k. \end{aligned} \tag{3.3.18}$$

In this particularly elegant example, notice how  $B(\mathbf{v}, k) > 0$  for all  $k \geq 0$  if, and only if  $x + z > 0$ . Take, for example, the values  $(x, y, z) = (2, 0, -1)$ , so that  $C_1(\mathbf{v}) > 0$ ,  $\operatorname{Re}(C_2(\mathbf{v})) > 0$ ,  $\operatorname{Im}(C_2(\mathbf{v})) > 0$ , as well as  $x + z = 1 > 0$ . The initial values  $(2, 0, -1)$  are thus a non-terminating witness for the loop<sup>11</sup>. Other permutations of the signs of  $C_1(\mathbf{v})$ ,  $\operatorname{Re}(C_2(\mathbf{v}))$ , and  $\operatorname{Im}(C_2(\mathbf{v}))$

<sup>11</sup>The reader might wish to review Equation 3.3.15 once more, which showed how  $(x, y, z) = (2, 1, -2)$  is also a non-terminating witness for the current loop. This witness, however, causes  $A(\mathbf{v}, k)$  to assume values arbitrarily close to zero; it also demonstrates the accuracy lost when obtaining a lower bound function: although this witness constrains the coefficients in a positive manner, so as to render the current  $B(\mathbf{v}, k)$  valid,  $B(\mathbf{v}, k)$  does not remain positive ( $x + z = 2 - 2 = 0$ ). This witness would thus not be detected.

may lead to functions as simple to constrain as Equation 3.3.18, and each of these conditions describes a different aspect of the loop's non-terminating behaviour.

The preceding lemma allows for the introduction of heuristics: values of  $\mathbf{v}$  are sought which allow  $A_i(\mathbf{v}, k)$  to be abstracted by some  $B_i(\mathbf{v}, k)$ , and which engender positivity in the sum  $B_i(\mathbf{v}, k)$  — a sum which consists of real-valued elements. If a vector  $\mathbf{v}$  exists for which  $GT^k \mathbf{v} > \mathbf{0} \forall k = 0, \dots, \zeta - 1$  and  $B_i(\mathbf{v}, k) > 0 \forall k \geq \zeta$ , for all of the loop's sums, then  $\mathbf{v}$  is necessarily a witness to the non-termination of the loop  $L$ . It is common for none of the loop's eigenvalues to be zero, in which case the first condition is not present, and the lower bound sum  $B_i(\mathbf{v}, k)$  must be positive for all  $k \geq 0$ . In the presence of a zero eigenvalue, however, a lower bound such as that for non-zero eigenvalues cannot easily be obtained, and instead the finite number ( $\zeta$ ) of iterations for which its term is non-zero are explicitly constrained. In this case, the signs of coefficient polynomials need only be constrained from some  $k = \zeta$  onwards.

Note that if, in each  $B_i(\mathbf{v}, k)$ ,  $p_{i1} = m_i$ , i.e., the sum consists only of positive summands, and no subtracted terms, then positivity holds for each sum, and  $\mathbf{v}$  is a witness to non-termination. However, because the worst-case negative contributions of terms with negative or complex eigenvalues are used to create  $B_i(\mathbf{v}, k)$ , each abstracted sum consists of only positive summands if, and only if, the coefficients of negative- and complex-based terms in every  $A_i(\mathbf{v}, k)$  are zero-valued; in fact, the conclusion of positivity is obvious in this case even without the abstracted sum  $B_i(\mathbf{v}, k)$ . The most basic heuristic (corresponding abstractly to the suggestion to force the positive contribution of each eigenvalue to outweigh its negative contribution, with which Section 3.3.3 concluded) is thus as follows:

**Heuristic 1 (positive coefficient).** Obtain all possible vectors  $\mathbf{v}$  such that, in every exponential sum  $A_i(\mathbf{v}, k)$  of a loop  $L$ :

- $C_{ij}(\mathbf{v}, k) > 0 \forall k \geq \zeta$ , for every coefficient  $C_{ij}$  whose corresponding base is a positive real eigenvalue;
- $C_{ij}(\mathbf{v}, k) = 0 \forall k \geq \zeta$ , for every coefficient  $C_{ij}$  preceding a negative or complex eigenvalue; and
- $GT^k \mathbf{v} > \mathbf{0} \forall k = 0, \dots, \zeta - 1$ , where  $\zeta$  is the dimension of the largest Jordan block corresponding to the eigenvalue zero.

The returned vectors (if any) are all witnesses to the non-termination of the loop  $L$ .

A more thorough heuristic allows the coefficients of negative and complex eigenvalues to assume non-zero values — as long as they are either positive or negative for all  $k \geq \zeta$ , and searches for vectors which cause the negative contributions in  $B_i(\mathbf{v}, k)$  to be outweighed by positive terms; note that these positive terms must then have bases of an equal, or greater, absolute value. Assume that the coefficients of  $A_i(\mathbf{v}, k)$  maintain their signs, so that  $B_i(\mathbf{v}, k)$  exists.  $B_i(\mathbf{v}, k)$  is a sum over  $s_i$  positive bases, each base engendering both a positive and a negative partial sum; the total sum can be written:

$$B_i(\mathbf{v}, k) = \left( \sum_{j=1}^{p_{i1}} \Lambda_j(\mathbf{v}, k) - \sum_{j=p_{i1}+1}^{m_{i1}} \Lambda_j(\mathbf{v}, k) \right) \lambda_1^k + \cdots \\ + \left( \sum_{j=m_{i(s_i-1)}+1}^{p_{is_i}} \Lambda_j(\mathbf{v}, k) - \sum_{j=p_{is_i}+1}^{m_{is_i}=m_i} \Lambda_j(\mathbf{v}, k) \right) \lambda_{s_i}^k.$$

As each pair of partial sums is itself a polynomial in  $k$ , this sum can be further simplified:

$$B_i(\mathbf{v}, k) = D_{i1}(\mathbf{v}, k)\lambda_1^k + D_{i2}(\mathbf{v}, k)\lambda_2^k + \cdots + D_{is}(\mathbf{v}, k)\lambda_{s_i}^k, \quad (3.3.19)$$

where  $D_{il}(\mathbf{v}, k)$  is a polynomial in  $k$  of the accustomed form, defined as

$$D_{il}(\mathbf{v}, k) = \sum_{j=m_{i(l-1)}+1}^{p_{il}} \Lambda_j(\mathbf{v}, k) - \sum_{j=p_{il}+1}^{m_{il}} \Lambda_j(\mathbf{v}, k).$$

To summarise what has been obtained: each exponential sum  $A_i(\mathbf{v}, k)$  has polynomial coefficients; when these coefficients are each either always positive or always negative, a simplified lower bound  $B_i(\mathbf{v}, k)$  can be obtained for  $A_i(\mathbf{v}, k)$  which is an exponential sum with polynomial coefficients and positive real bases. Thus any heuristic based on this concept will necessarily begin by obtaining vectors which cause every coefficient of  $A_i(\mathbf{v}, k)$  to remain either positive or negative, unfortunately restricting the set of possible witnesses to non-termination.

Considering Equation 3.3.19, one notices that an adaptation of the positive coefficient heuristic can be applied: obtain all  $\mathbf{v}$  such that  $D_{il}(\mathbf{v}, k) > 0 \forall k \geq \zeta$ , for all  $l = 1, \dots, s$ . Such a vector  $\mathbf{v}$  would induce positivity in  $B_i(\mathbf{v}, k)$ , and if  $\mathbf{v}$  does this for all of the exponential sums, and  $GT^k \mathbf{v} > \mathbf{0} \forall k = 0, \dots, \zeta - 1$ , it is a witness to the loop's non-termination.

An expansion of this idea leads to a final heuristic: if  $D_{i1}(\mathbf{v}, k)$  is sufficiently large,  $D_{i2}(\mathbf{v}, k)$  need not be positive for positivity of the sum to hold: since  $\lambda_1 > \lambda_2$ ,

$$D_{i1}(\mathbf{v}, k)\lambda_1^k + D_{i2}(\mathbf{v}, k)\lambda_2^k \geq D_{i1}(\mathbf{v}, k)\lambda_2^k + D_{i2}(\mathbf{v}, k)\lambda_2^k \\ = (D_{i1}(\mathbf{v}, k) + D_{i2}(\mathbf{v}, k)) \lambda_2^k,$$

and if  $D_{i1}(\mathbf{v}, k) > 0 \forall k \geq \zeta$ , one need not restrict  $D_{i2}(\mathbf{v}, k)$  to positive values, only  $(D_{i1}(\mathbf{v}, k) + D_{i2}(\mathbf{v}, k)) > 0$ . This technique can be extended to numerous terms, so that instead of enforcing the positivity of every coefficient  $D_{il}(\mathbf{v}, k)$ , one takes the residue of larger terms (which are also positively constrained) into account, ensuring that  $D_{i1}(\mathbf{v}, k) > 0$ ,  $(D_{i1}(\mathbf{v}, k) + D_{i2}(\mathbf{v}, k)) > 0$ ,  $(D_{i1}(\mathbf{v}, k) + D_{i2}(\mathbf{v}, k) + D_{i3}(\mathbf{v}, k)) > 0$ , etc. This approach yields Heuristic 2. Firstly, each coefficient  $C_{ij}(\mathbf{v}, 0)$  shall be constrained in three ways:  $(= 0), (> 0), (\leq 0) \forall k \geq \zeta$ . An assignment of one of these constraints to each coefficient in every sum is called a *sign permutation* of the loop. Then:

**Heuristic 2 (positive weighted coefficient).** Given a loop  $L$  with  $r$  exponential sums  $A_i(\mathbf{v}, k)$ ; for each sign permutation of  $L$ ,  $r$  abstraction sums  $B_i(\mathbf{v}, k)$  are obtained, each with non-zero coefficients  $D_{i1}(\mathbf{v}, k), \dots, D_{is_i}(\mathbf{v}, k)$ . Return the set of vectors  $\mathbf{v} \in \mathbb{Z}^{n+1}$  for which:

- $\sum_{l=1}^{\sigma} D_{il}(\mathbf{v}, k) > 0 \forall k \geq \zeta, \forall \sigma = 1, \dots, s_i, i = 1, \dots, r$ ; and
- $GT^k \mathbf{v} > \mathbf{0} \forall k = 0, \dots, \zeta - 1$ .  $\zeta$  is again the dimension of the largest Jordan block corresponding to the eigenvalue zero.

The three indices  $k, \sigma, i$  enumerate iterations of the polynomial, subsets of sum elements, and exponential sums respectively. The returned vectors (if any) are all witnesses to the non-termination of the loop  $L$ .

*Proof.* It shall be shown that any vector  $\mathbf{v}$  returned by the heuristic is a witness to the non-termination of  $L$ . Assume such a  $\mathbf{v}$  exists; then each sum  $B_i(\mathbf{v}, k)$  is such that  $D_{i1}\lambda_1^k > 0 \forall k \geq \zeta$ . Furthermore,

$$D_{i1} > 0 \Rightarrow D_{i1}\lambda_1^k > D_{i1}\lambda_2^k,$$

so that

$$D_{i1}\lambda_1^k + D_{i2}\lambda_2^k > (D_{i1} + D_{i2})\lambda_2^k > 0 \forall k \geq \zeta.$$

Inductively,

$$D_{i1} + D_{i2} > 0 \Rightarrow (D_{i1} + D_{i2})\lambda_2^k > (D_{i1} + D_{i2})\lambda_3^k,$$

so that

$$(D_{i1} + D_{i2})\lambda_2^k + D_{i3}\lambda_3^k > (D_{i1} + D_{i2} + D_{i3})\lambda_3^k > 0 \forall k \geq \zeta,$$

and, eventually,  $B_i(\mathbf{v}, k) > 0 \forall k \geq \zeta$ , and  $\mathbf{v}$  is a non-terminating witness. □

The positive weighted coefficient heuristic is the most thorough heuristic which shall be presented in this text. Note that heuristics such as these are simple to apply to the abstracted sums  $B_i(\mathbf{v}, k)$  — the difficulty lay in obtaining the abstraction. Hence, additional heuristics can be derived; for example, the terms of exponential sums can be synchronised at characteristic points such as abscissa intersections [27], and these synchronised points might be adjusted to imply non-termination.

It was mentioned previously that falsification techniques which could be adapted to include preconditions on the loop's variables were desired. The inclusion of preconditions in Heuristics 1 and 2 is simple, since the heuristics are constructed to generate conditions on the values of the loops variables, which, if satisfied, provide non-terminating witnesses. Preconditions are themselves conditions on the loop variables, and can thus augment those generated by the heuristic, resulting in the exclusion of values which do not satisfy the preconditions, as well as the validity of any values which satisfy the augmented heuristic conditions.

$$\begin{aligned} &\text{while } (x + 5 > 0) \{ \\ &\quad x = 2x - y \\ &\quad y = -2y - 2 \\ &\} \end{aligned}$$

**Figure 3.20:** A non-terminating loop.

Now that the heuristics which were the goal of this chapter have been obtained, another example seems fitting: the loop in Figure 3.20 produces the transformation and Jordan matrices

$$T = \begin{bmatrix} 2 & -1 & 0 \\ 0 & -2 & -2 \\ 0 & 0 & 1 \end{bmatrix} \text{ and } J = \begin{bmatrix} 2 & 0 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The exponential sum which must remain positive for the loop to be non-terminating is given by

$$A(\mathbf{v}, k) = x^{[k]} = \left(x - \frac{1}{4}y + \frac{1}{2}\right) 2^k + \left(\frac{1}{4}y + \frac{1}{6}\right) (-2)^k - \frac{2}{3}1^k.$$

To apply the positive weighted coefficient heuristic, a sign permutation must be selected. To this end, let

$$\left(x - \frac{1}{4}y + \frac{1}{2}\right) > 0 \text{ and } \left(\frac{1}{4}y + \frac{1}{6}\right) \leq 0,$$

so that the lower bound sum is

$$\begin{aligned} B(\mathbf{v}, k) &= \left(x - \frac{1}{4}y + \frac{1}{2}\right) 2^k + \left(\frac{1}{4}y + \frac{1}{6}\right) |2|^k - \frac{2}{3} \\ &= \left(x + \frac{2}{3}\right) 2^k - \frac{2}{3}, \end{aligned}$$

which is non-terminating according to Heuristic 2 if

$$\left(x + \frac{2}{3}\right) > 0 \text{ and } \left(x + \frac{2}{3} - \frac{2}{3}\right) = x > 0.$$

Combining this with the sign permutation's constraints, a precondition for the loop's non-termination is

$$\left(x > 0 \wedge y \leq -\frac{2}{3} \wedge x - \frac{1}{4}y + \frac{1}{2} > 0\right).$$

```

while (x + 5 > 0) {
    x' := 3x + 4y + 4z
    y' := 4x - 3y + 4z
    z := -3x - 3y - z
    x := x'
    y := y'
}

```

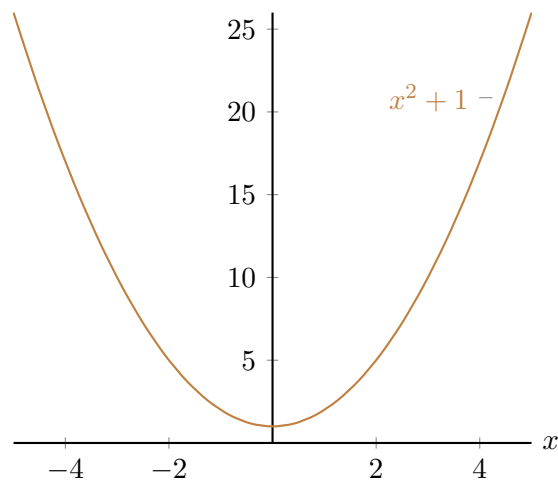
**Figure 3.21:** A non-terminating loop which is detected by the techniques in Section 3.3.5, but not those of Section 3.2.

As an additional example: the loop in Figure 3.21 is non-terminating, and a non-terminating witness can be found using both the positive coefficient and positive weighted coefficient heuristics. Attempting to prove a candidate set recurrent, however, as in Section 3.2, fails to yield a non-terminating witness. The loop produces the (abbreviated) eigenvalues  $-4.58916$ ,  $(1.79458 + 3.5001i)$ ,  $(1.79458 - 3.5001i)$ , and 1. The only non-terminating witness for this loop is  $(x, y, z) = (0, 0, 0)$ .



### 3.3.6 Constraining polynomials

Heuristics 1 and 2 both require the ability to constrain a polynomial coefficient  $D_{ij}(\mathbf{v}, k)$  to either the positive or non-positive domain, a problem in itself. Considering a polynomial  $C(x)$ , the terms *positive* and *non-positive* are, as before, taken to mean  $C(x) > 0 \forall x \geq 0$  and  $C(x) \leq 0 \forall x \geq \zeta$  respectively; the first is equivalent to showing that  $C(x)$  has no non-negative real roots, and then that  $C(0) > 0$ , whereas asserting non-positivity is equivalent to showing that the polynomial is zero, or each of the polynomial's positive roots has even multiplicity and  $C(x) < 0$  for some  $x \geq \zeta$ . The obvious solution to proving (non-)positivity, then, is to obtain



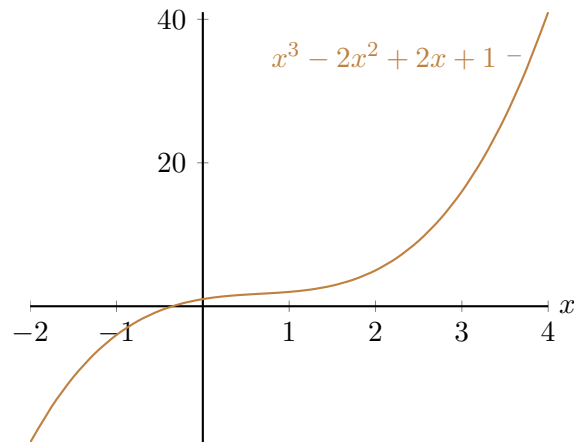
**Figure 3.22:** A positive polynomial with complex roots.

the roots of the polynomial in question, and assert the required nature of every real root. For polynomials of degree four and less, closed solutions for the values of roots exist, however by Abel's Impossibility Theorem [4], such solutions cannot be found for higher-degree polynomials. The alternative, in this regard, is the use of root-finding methods such as Newton's, Halley's, and Laguerre's Methods, or the Jenkins-Traub algorithm. Although practically effective, such methods are not necessarily guaranteed to locate all of a polynomial's roots; in addition, the polynomials with which this section is concerned are symbolic, and such algorithms rely on concrete values to search for roots.

An alternative solution, proposed here, is an application of the rather simple, but incomplete, Lower/Upper Bound Theorems, which can assert that a polynomial has no roots larger/smaller than a given value. Firstly, note that when a polynomial  $p(x)$  is divided by a

linear polynomial  $(x - c)$ , a quotient and remainder are obtained:  $p(x) = q(x)(x - c) + r$ ; the upper bound theorem states that if  $c > 0$ ,  $r > 0$ , and every coefficient of  $q(x)$  is positive, then  $c$  is an upper bound for the roots of  $p(x)$ . This theorem is relevant, since for  $C(x)$  to be positive from a point  $\zeta \geq 0$  onwards,  $x = \zeta$  must be an upper bound for the roots of  $C(x)$ . If this is the case, and both  $C(\zeta) > 0$  then  $C(x) > 0 \forall x \geq \zeta$ . In the common case where  $\zeta = 0$ , a value of  $c = 1$  must be used in the upper bound theorem, so that the roots of  $C(x)$  are less than or equal to 1; in addition, the two initial constraints  $C(0) > 0$  and  $C(1) > 0$  must be enforced. Constraints such as these can easily be encoded symbolically.

The strength of this approach is its simplicity and efficiency — it involves a few elementary arithmetic operations. Its weakness, however, lies in its incompleteness; a polynomial whose roots are bounded from above by some value  $c$  might not engender quotient polynomial  $q(x)$  with positive coefficients. As an example of this shortcoming, consider  $p(x) = x^3 - 2x^2 + 2x + 1$ , plotted in Figure 3.23, and assume  $\zeta = 0$ . One may perform the division of  $p(x)$  by  $(x - 1)$



**Figure 3.23:** A polynomial for which the Upper Bound Theorem is incomplete.

synthetically, by writing out the coefficients of  $p(x)$ , bringing the leading coefficient down, and then multiplying each value which is brought down by  $c = 1$ , adding it to the following coefficient, and bringing the obtained sum down:

$$\begin{array}{r|rrrr}
 1 & 1 & -2 & 2 & 1 \\
 & & 1 & -1 & 1 \\
 \hline
 & 1 & -1 & 1 & 2
 \end{array}$$

Then the lower row contains the quotient and remainder terms:  $p(x) = x^3 - 2x^2 + 2x + 1 = (x^2 - x + 1)(x - 1) + 2$ ; note that the coefficients of the quotient are not all positive, and thus the fact that  $x = 1$  is an upper bound for the roots of  $p(x)$  does not follow from the Upper Bound Theorem.

To assert the non-positivity of  $p(x)$ , the upper bound theorem is applied similarly to the negation of  $p(x)$ . Note that if, after obtaining a quotient and remainder by synthetic division, the remainder is 0, then the constant term  $c$  of the divisor is in fact a root of the dividend  $p(x)$ , as  $p(x) = q(x)(x - c)$ . In the current consideration, in which  $c = 1$  or  $c = \zeta$ , a remainder of 0 implies that  $p(c)$  is not positive. Thus, when checking for positivity, the remainder must be positive; however, when asserting non-positivity, a remainder of 0 in the division of  $-p(x)$  is acceptable: assuming that all of the quotient polynomial's coefficients are positive and the remainder 0,  $-p(x)$  must be non-positive, since adding any  $\epsilon > 0$  to the constant term of  $-p(x)$  will yield a positive remainder, implying by the upper bound theorem that  $-p(x) + \epsilon$  has no roots greater than or equal to  $c$ ; so either  $-p(x) \geq 0$  or  $-p(x) \leq 0 \forall x \geq c$ . To exclude the possibility of  $-p(x) \leq 0$ , it must be shown that  $p(x) > 0$  for some  $x \geq c$ : since each of the roots of  $p(x)$  which are greater than  $c$  have even multiplicity, there can be at most  $\frac{d}{2}$  distinct zeroes of  $p(x)$  which are greater than  $c$ , where  $d$  is the degree of  $p(x)$ . Hence, at least one of the  $\frac{d}{2} + 1$  integer mappings following  $x = c$  must be non-zero, and as such it is asserted that  $p(x) > 0$  for some  $c \leq x \leq c + \frac{d}{2}$ . Also adding the constraint  $p(0) \geq 0$  whenever  $\zeta = 0$ , the non-positivity constraint is complete<sup>12</sup>. Note that instead of negating the polynomial before the application of the upper bound theorem, if  $p(x) = q(x)(x - c) + r(x)$ , then  $-p(x) = -q(x)(x - c) - r(x)$ , so that the upper bound theorem can be applied to  $-p(x)$  by equivalently performing synthetic division on  $p(x)$  and asserting that each of the coefficients in  $q(x)$  are negative instead of positive, and similarly for the remainder.

### 3.3.7 Deciding termination for simple affine loops

It should be noted that the above heuristics were based on the fact that the positivity of exponential sums with more than two terms cannot be explicitly solved, and thus an alternative approach based on the signs of coefficients was devised. In the simplest case, however, an

<sup>12</sup>Although the presented non-positivity approach excludes the possibility of a zero polynomial, the issue is avoided since the zero case is handled separately.

exponential sum has no more than two terms, and can be explicitly solved: in particular, a single-variable affine loop generates a transformation matrix of dimension  $2 \times 2$ , and exponential sums of at most two terms, each with linear coefficients. The eigenvalue generated by the constant shift of the transformation matrix is always 1, and thus complex eigenvalues (which must appear in conjugate pairs) are not apparent. As shown in Lemma 5 of Section 3.3.3, the positivity of such functions is explicitly solvable for positive eigenvalues; a similar deduction can be performed in the case of a single negative eigenvalue. This leads to yet another method of deciding the termination of single-variable affine loops.

```

while ( $x > 0$ ) {
     $x := 3x - 20$ 
}

```

**Figure 3.24:** A single-variable affine loop.

The loop in Figure 3.24 generates the eigenvalues 3 and 1, and the exponential sum

$$A(x, k) = x^{[k]} = (x - 10)3^k + (10)1^k,$$

which can be explicitly solved:  $A(x, k) > 0 \forall k \geq 0$  if, by Lemma 5,  $x - 10 > 0$  and  $x - 10 + 10 = x > 0$ , i.e., if  $x > 10$ . Since Lemma 5 requires two non-zero coefficients, the case in which the leading coefficient is zero ( $x - 10 = 0$ ) must be augmented: in this case the sum also remains positive. The non-terminating condition for this loop is thus  $x \geq 10$ .

Two-variable affine loops which contain no integer translations (no constant terms in the affine combination) can also be represented by  $2 \times 2$  transformation matrices, and are thus decidable, since their exponential sums contain at most two terms.

In the case of two-variable affine loops which contain integer shifts, three terms may appear in the exponential sum, so that the sum might not be explicitly solvable.

### 3.3.8 Termination verification via sign permutations

Lastly, it is pertinent to elaborate on the importance of sign permutations: because a loop is generally dominated by its largest eigenvalue, certain conclusions can often be drawn from the structure of the exponential sums, without further inspection. For instance, if the largest

eigenvalue is not positive, and its coefficient is never zero, the loop will eventually terminate, regardless of the remaining coefficients. Although the focus here has been the search for non-termination via the examination of exponential sums, it seems logical that the approach could also prove valuable to the verification of affine loop termination, specifically in the approximation of termination conditions. The following lemma incorporates some simple observations of sign permutations into a verification tool which shall prove useful in the following chapter.

**Lemma 8.** *Given a loop  $L$  with  $r$  exponential sums  $A_i(\mathbf{v}, k)$ , if the  $m > 0$  largest (in terms of absolute value) eigenvalues of  $L$  are not positive, then the  $m$  leading coefficients of every exponential sum must simultaneously be zero for the loop to be non-terminating; i.e.: if no vectors  $\mathbf{v}$  exist such that  $C_{i1} = \dots = C_{im} = 0 \forall i = 1, \dots, r$ , then  $L$  is terminating.*

Note how the lemma specifically refers to the coefficients of the original exponential sums  $A_i(\mathbf{v}, k)$  — since  $B_i(\mathbf{v}, k) \leq A_i(\mathbf{v}, k)$ , the positivity of  $B_i(\mathbf{v}, k)$  implies the positivity of  $A_i(\mathbf{v}, k)$ , and thus termination can be falsified by applying heuristics to the former. However, termination cannot be verified by considering  $B_i(\mathbf{v}, k)$ , since the non-positivity of  $B_i(\mathbf{v}, k)$  does not necessarily imply the same in  $A_i(\mathbf{v}, k)$ .

This termination lemma is surprisingly useful: the most common manner of verifying termination for linear programs is the synthesis of a valid ranking function for the loop (a complete method exists for linear ranking functions [24]). The loop in Figure 3.25 is terminating, however no linear ranking function which proves this termination exists [24]. Decomposing the loop

$$\begin{array}{l} \text{while } (x \geq 0) \{ \\ \quad x := -2x + 10 \\ \} \end{array}$$

**Figure 3.25:** A terminating loop for which no linear ranking function exists.

yields the exponential sum  $A(x, k) = (x - \frac{10}{3})(-2)^k + \frac{7}{3}$ ; the coefficient of  $(-2)^k$ , the leading eigenvalue, can never be zero since  $x$  is an integer, and as such  $A(x, k)$  cannot be positive for all non-negative  $k$ . Hence the loop is terminating by Lemma 8.

### 3.4 Summary

This chapter deduced two approaches to termination falsification: the first based on periodic monotonicity and recurrent sets, the second relying on an algebraic decomposition of affine loops which reduces non-termination to the positivity of certain exponential sums.

The results obtained during the investigation of recurrent sets for affine loops yield decidability for single-variable and cyclic loops, and guarantee the existence of such a recurrent set in the case of periodically monotonic affine loops. Unfortunately, the maximal period lengths which provide a bound for the single-variable and cyclic procedures do not generalise to the standard periodically monotonic case, so that the termination of all periodically monotonic loops is not necessarily decidable.

Once the Jordan decomposition of the affine loop had been presented, a known proof for the decidability of affine loops which engender only positive eigenvalues was discussed. Two falsification heuristics were then developed by investigating the nature of the exponential sums generated by the decomposition. The first, positive coefficient heuristic attempts to force the contribution of positive eigenvalues to be positive, whilst excluding the contribution of non-positive eigenvalues; it is thus more applicable to affine loops with few non-positive eigenvalues. The last presented approach, and the primary heuristic obtained by this text, is the positive weighted coefficient heuristic; this heuristic does not exclude non-positive terms, but rather approximates them by way of lower bounds. Both of these approaches produce sets of non-terminating witnesses with aid from the known concepts introduced in Section 3.3.4.

The implementation of the above techniques shall be discussed in the forthcoming chapter, whilst a discussion of their applicability and usefulness awaits in Chapter 5.

## Chapter 4

# Implementation

Chapter 3 was primarily theoretical, concerned with finding some form of solution to the problem of affine loop termination falsification. Solving a problem theoretically, and attempting to apply a solution practically, however, each present unique issues, and require differing skills. This chapter then, provides insight into the implementation of the techniques discussed in the previous chapter, primarily to discuss the challenges of such a task; Chapter 5 remains to showcase, as well as evaluate, the functionality of the non-termination heuristics.

Program loops can appear in numerous forms: besides *for*, *while*, and *do while* constructs, an audacious programmer can engender repetitive behaviour by direct placements of the *goto* instruction, or even recursive function calls. Any of these semantic loop forms can lead to the general affine loop structure presented in Figure 3.2. The work described in this section is the creation of a program which, given bytecode generated from the Java [15] programming language, can detect and reconstruct affine loops, and apply the heuristics developed in the preceding section in an attempt to decide the termination of these loops. To reconstruct affine loops, one must be able to inspect the mathematical properties of a program loop — the manner of inequalities and transformations which are apparent — and detect the presence of such affine structure, without much regard for the loop’s syntax. The techniques of the previous chapter lend themselves to static verification — once the affine properties of a loop have been extracted, the program code need not be executed for the techniques to be applied.

The implementation described in this section has been constructed as an extension for the Java Pathfinder (JPF) [9] software model checker, as the software provides a solid foundation for the implementation’s requirements: it is highly extensible, provides interfaces for numerous

constraint solvers, includes a strong suite of symbolic utilities, and can be practically used for both static and dynamic verification. More specifically, the implementation requires the symbolic functionality of Symbolic Pathfinder (SPF) [25].

## 4.1 Detecting loops

The first task in implementing the termination techniques is the detection of loop constructs, and the inspection of their structure. Given the relevant source code along with a Java executable, JPF does provide basic functionality to handle the code, however, the presence of such source code is not guaranteed, as JPF is ultimately a bytecode-level model checker. Hence, the desired affine loop extension should include the ability to infer the required loop data directly from Java bytecode. Syntactically, the extension currently detects affine loops within *for* and *while* loops, which are encoded virtually identically within bytecode, however the extension is constructed in an object-oriented manner, and bytecode loops are first contained within an abstract *Loop* object, before this object is parsed to check whether it conforms to an affine structure. Thus, support for other syntactic loop forms or mathematical loop structures can be added to the implementation. Firstly, the process of recognising a *while* loop (or analogously, a *for* loop) in bytecode and constructing a generic *Loop* object will be briefly presented, followed by the method of checking for and deducing an affine loop object from such a description.

### 4.1.1 Detecting loop boundaries

A program loop consists of a guard condition and an iterable body, followed by a backwards jump. The loop body is encoded as a sequence of successive machine instructions, and is only of interest when investigating the loop's possible affine nature. The loop's guard condition, however, is a combination of variable constraints, each of which engenders an *if* jump instruction whose placement and arguments depend on the form of the constraint and guard condition. The bytecode body of a standard loop is succeeded by a *goto* instruction, which targets the beginning of the loop's guard condition. To interpret a Java bytecode loop, the extension must be able to recognise the loop's initial and final instructions, as well as the boundary of the guard condition and body, and be able to reconstruct the loop's guard condition from the stored sequence of jump instructions. As an example, consider the simple *while* loop, along



with its generated bytecode, depicted in Figure 4.1.

<pre> while (<math>x &gt; 0</math>) {     <math>x := x + 1</math> } </pre>	<pre> 1: push <math>x</math> 2: if <math>x \leq 0 \rightarrow 8</math> 3: push <math>x</math> 4: push 1 5: add 6: store <math>x</math> 7: goto 1 8: ... </pre>
----------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------

**Figure 4.1:** A standard *while* loop and its generated (pseudo-)bytecode.

Loops can engender bytecode which is far more complicated than that of Figure 4.1, however the unnested affine loops with which this text is concerned necessarily produce relatively simple bytecode. Although the goal of the implementation did not extend past the reconstruction of affine loops, support for the detection of nested loops and arbitrarily complex guard conditions was incorporated during its construction; these details are currently not relevant, and as such are omitted from the current discussion.

The following lemma sufficiently describes the detection of loop boundaries with regard to unnested *while* loops:

**Lemma 9.** *Consider a Java while loop which has neither parent or child loops. The bytecode generated by this loop block contains an if instruction whose target is preceded by a backwards jump, which in turn targets an instruction prior to the if instruction. Specifically, the backwards jump is the first instruction following the loop's body, whereas the target of the backwards jump is the initial instruction in the loop's guard condition.*

Any bytecode which contains the properties described in Lemma 9 engenders loop behaviour. In addition, the lemma provides both the indices of a loop's initial instruction as well as the first instruction following the loop (that is, the instruction succeeding the loop's *goto* instruction); these indices shall be named the *success* and *failure* indices of the guard condition respectively. The remaining instruction index to be located is that of the loop's first body instruction. Note that the final *if* instruction generated by a guard condition must target the

condition's failure index, thus, the instruction directly succeeding the last *if* instruction which targets the failure index must be the first instruction of the loop's body.

Regarding the guard condition of a bytecode loop: a condition is a set of variable constraints (constraints such as  $(x > 0)$ ,  $(2x + 3y \leq 4)$ , and  $(5x \neq -10y)$ ) which are combined at arbitrary depths by the operators *and* and *or*. The extension must parse a loop's guard condition and construct a symbolic representation thereof. This is done by identifying the individual constraints within the condition (constraint boundaries and operators can be identified by inspecting the targets of the bytecode jumps they generate), and executing the arithmetic instructions of each constraint using JPF's symbolic execution extension — SPF. This execution generates the desired symbolic representations of the constraints, which are then combined in a tree structure which incorporates the constraint operators. Instead of detailing the reconstruction of general guard conditions here, the algorithm which was designed to perform this task is given as Algorithm 2 in Appendix B.1. A simple example of the manner in which guard conditions appear within bytecode is provided in Figure 4.2.

```

1: push x
2: if x ≤ 0 → failInd
3: push y
4: if y ≤ 0 → failInd
5: push z
6: if z ≤ 0 → failInd
7: success
8: ...
9: failure

```

**Figure 4.2:** The bytecode generated by the guard condition  $(x > 0 \wedge y > 0 \wedge z > 0)$ .

The data gathered during the loop detection stage is stored as a new Java *Loop* object, containing:

- the indices of the initial loop instruction, the first body and guard instructions, and the first instruction succeeding the loop;
- a tree representation of the loop's guard condition; and

- the set of instructions which constitute the symbolic method in which the loop is found.

### 4.1.2 Constructing affine loops

Returning to the affine loops with which this text is concerned: once a loop has been identified within the bytecode of a Java program, and a generic object representation of this loop has been initialised, the guard condition and update transformations must be inspected to deduce whether only affine transformations are present. Once again, SPF is used, in this case to recognise linear operations of integer variables within the loop's body and thereby construct symbolic representations of affine transformations.

To explain the affine loop recognition procedure in more detail, recall the general form of an affine loop, replicated here as Figure 4.3, but originally presented in Section 3.1:

$$\begin{aligned} &\text{while } (G\mathbf{v} > \mathbf{0}) \{ \\ &\quad \mathbf{v} := T\mathbf{v} \\ &\} \end{aligned}$$

**Figure 4.3:** An affine loop.

Firstly, the guard condition of the *Loop* object is considered:  $(G\mathbf{v} > \mathbf{0})$  is an *and* condition in which each constraint is a singular homogenised affine inequality of the form

$$\mathbf{g}\mathbf{v} + b > 0.$$

Any general constraint

$$\mathbf{g}_1\mathbf{v} + b_1 \succ \mathbf{g}_2\mathbf{v} + b_2, \tag{4.1.1}$$

where  $\succ \in \{>, \geq, <, \leq, =\}$ , can be homogenised via a few simple transformations: firstly, the affine expressions can be grouped on one side of the inequality, yielding

$$\mathbf{g}\mathbf{v} + b \succ' 0,$$

where  $\succ' \in \{>, \geq, =\}$ . Since both the variables and coefficients range over  $\mathbb{Z}$ ,

$$\mathbf{g}\mathbf{v} + b \geq 0 \Leftrightarrow \mathbf{g}\mathbf{v} + (b + 1) > 0.$$

In addition,

$$\mathbf{g}\mathbf{v} + b = 0 \Leftrightarrow (\mathbf{g}\mathbf{v} + b > -1 \wedge -\mathbf{g}\mathbf{v} - b > 1).$$

Note that the relational operator  $\neq$  is not supported, since

$$\mathbf{g}\mathbf{v} + b \neq 0 \Leftrightarrow (\mathbf{g}\mathbf{v} + b > 0 \vee -\mathbf{g}\mathbf{v} - b > 0),$$

in which the homogenised inequalities form an *or* condition.

Using the above logic, the implementation processes constraints as in Equation 4.1.1 to verify the affine nature of a guard condition.

Whilst processing the constraints which constitute the affine loop's guard condition, a collection of the variables which appear within the constraints is maintained. These variables directly influence the termination properties of the loop, and are thus loop variables; this collection does not yet completely represent the set of loop variables, since any variable which appears in the update transformation of one of these preliminary loop variables also affects the loop's termination, albeit indirectly.

Following the inspection of the guard constraints, and making use of the preliminary loop variable collection thereby obtained, the loop body is inspected. Note that the instructions within the body of an affine *while* loop may be arbitrarily complex, as long as those which affect the loop variables in some way constitute affine transformations; that is, the transformations which influence termination must be identified and validated, whilst the remaining instructions are ignored. The SPF implementation accomplishes this by processing every update transformation applied to one of the program's symbolic variables, maintaining a hash map from each symbolic variable to its update expression. Subsequently, the update transformations of the preliminary loop variables are examined: if a non-affine transformation is encountered, the loop is non-affine; if the transformation is in fact affine, each of the variables which form part of the linear combination are added to the set of loop variables. This procedure continues until the update transformation of every variable present in the set of loop variables has been verified as affine.

At this stage of the affine loop reconstruction, the loop has been verified as affine, and the symbolic update transformation expressions of the loop variables have been obtained; the only remaining task involves the construction of the standardised matrices  $G$  and  $T$  — as in Figure 4.3 — which describe the loop. The construction of the guard matrix  $G$  is relatively

simple: each standardised affine inequality engenders a row in the matrix  $G$ , where the  $i$ th column entry in the row is the integer coefficient of the  $i$ th loop variable, and the final entry — in column  $(n + 1)$  — is the constant translation in the affine transformation. Terms within an affine transformation are combined by way of binary operations, and SPF's object-oriented representation of linear transformations is based on this principle. As such, extracting coefficients from a symbolic expression involves a depth- or breadth-first search of the hierarchical object structure; the homogenisation of the guard constraints is performed using the extracted integer arrays — a form more suited to the arithmetic task than the symbolic object tree.

The update transformation matrix  $T$  is constructed in a similar manner: since the loop variables and their respective update expressions are known, the coefficients of each update expression are extracted and stored in the respective row of  $T$ . A small complication is present regarding the construction of  $T$ : the matrix  $T$  is interpreted as simultaneous in Section 3.1, that is, each of the variable updates are performed at the same time, according to the values of the loop variables at the start of the current loop iteration. Programmed loops, however, contain sequential commands, and thus the variable values required by a certain update transformation might not be those which were present at the start of a loop iteration, if the update transformation of one of the variables in question appears at an earlier position in the loop's body. Not only is  $T$  simultaneous in nature, it also specifies the existence of  $n$  loop variables and  $n$  update transformations. Another consideration regarding the ordering of update expressions is thus the possibility of multiple affine updates to a single loop variable, which must be combined in the correct order. To resolve both of these issues, when a symbolic integer is loaded by the virtual machine, its current update expression is substituted in its place; this in essence retains any updates made within the loop body up until that point, and ultimately produces a set of variables which each have a single update expression, composed in terms of the variable values given at the start of the loop's body. This process translates analogously to Algorithm 1, which constructs  $T$  sequentially, recruiting previously stored expressions if applicable. The algorithm receives  $l$  sequential expressions regarding  $\mathbf{e}_1, \dots, \mathbf{e}_l$ , where each  $\mathbf{e}_i$  is a  $1 \times (n + 1)$  matrix detailing an affine transformation applied to a loop variable.

To summarise, the following is a brief presentation of the approach taken by the JPF extension regarding the detection and symbolic reconstruction of programmed affine loops:

1. Every invocation of a method is caught by the loop listener, which communicates with

---

**Algorithm 1** Simultaneous transformations

---

- 1: Let  $T := I_{n+1}$ , the  $(n + 1)$ -square identity matrix.
  - 2: **for each**  $e_i$  **do**
  - 3:     Set row  $i$  of  $T$  to  $e_i T$ .
  - 4: **end for**
- 

the JPF virtual machine. If the invoked method has been marked as symbolic by the user, its set of bytecode instructions are gathered.

2. The method's instructions are scanned; every *if* instruction is inspected, as it may denote the first constraint in the guard condition of a loop.
3. If a loop is located, it's boundaries are passed to an implementation of Algorithm 2 (Appendix B.1), which constructs a tree representation of the loop's guard condition; a generic *Loop* object can then be created.
4. If the *Loop* object conforms to the structure of an affine loop, the extracted symbolic representations of the loop's affine nature are used to create an *AffineLoop* object.
5. Once the method's instructions have been fully parsed for loops, execution is returned to the JPF virtual machine, and the symbolic execution of the method continued. If an instruction is encountered which signifies the start of a detected loop, the loop is retrieved, its termination investigated, and a decision made to either skip the loop, execute as normal, or place restrictions on the values of the loop variables.

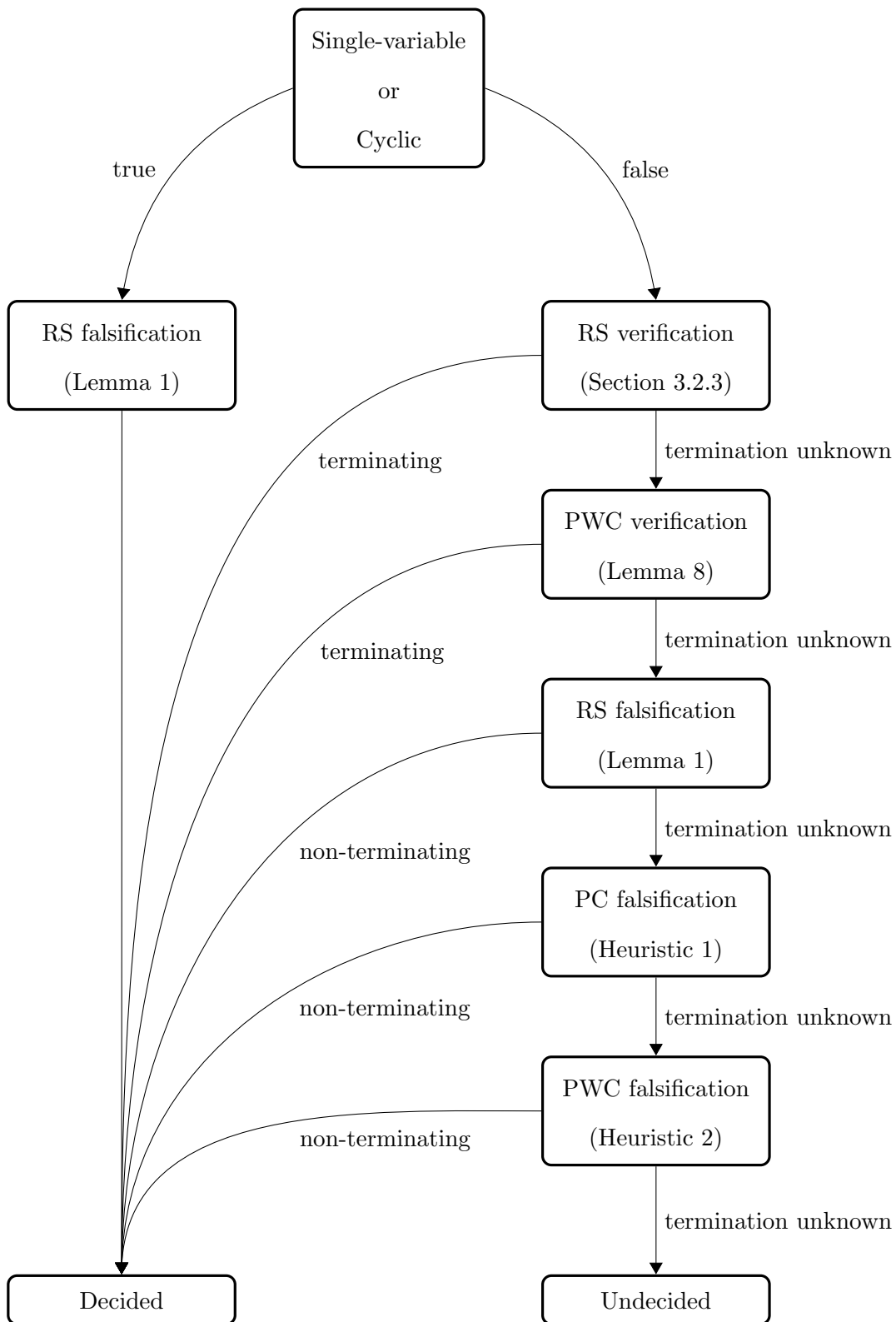
Presently, the detection and interpretation of program loops have been fully discussed; the remainder of the chapter shall be devoted to the implementation of the concepts developed in Chapter 3; as such, the loops considered may be assumed affine.

## 4.2 Non-termination algorithms

The translation of techniques from an algebraic to a programmatic domain is not without its own problems — apart from the resource limitations present in computer systems, which impose restrictions on the complexity of an algorithm, there are fundamental differences in the composition of the domains. A computer system is limited to finite approximations of rational

and real numbers, whereas algorithms may depend on the ability to define a real number with infinite precision. The linear algebraic nature of much of Chapter 3 implies that such finite representations of real numbers shall be rather problematic to the intended implementation of the proposed solutions. Fortunately, in this regard, there are utilities which extend the real number capabilities of the Java language; the two which have been employed in this implementation are the Apfloat [29] library for Java, and Wolfram's Mathematica [33] suite, along with its Java interface, JLink. The first tool — Apfloat — provides arbitrary-precision real and complex classes, along with an array of mathematical functions for use with these classes; the library performs more efficiently for very large numbers than the common *BigDecimal* native Java class. Wolfram's Mathematica, on the other hand, is an incredibly powerful tool, combining complex mathematics with computing power, and providing implementations of many mathematical procedures — particularly Jordan decomposition. The Jblas [7] linear algebra library, which is based on the LAPACK [3] matrix computation library, can be used as a substitute for Mathematica, but makes use of the *double* data type for floating-point numbers and, as of version 1.2.0, does not provide methods for the Jordan decomposition of an integer matrix. It does, however, provide the eigenvalues and eigenvectors of an integer matrix, and thus all that was required from the authors to enable support for the diagonalisation of such matrices was the implementation of a function which computes the inverse of a complex matrix (if it exists). Hence, if access to Mathematica software is unavailable, one may make use of Jblas, but in doing so be limited to diagonalisable transformation matrices and a maximum floating-point precision of 64 bits.

Before discussing the recurrent set and Jordan decomposition approaches individually, the order in which the techniques are applied shall be provided. Recall that the termination of single-variable and cyclic affine loops is decidable via a recurrent set algorithm based on Lemma 1 (single-variable loops can equally easily be decided via the positive weighted coefficient heuristic); these are the only loops which shall be considered decidable by the procedure. In the case that a loop is neither single-variable nor cyclic, the termination verification corollaries of the recurrent set and Jordan decomposition approaches respectively are applied, and thereafter, the loop is checked for recurrence, and the positive and positive weighted coefficient heuristics are applied. If any of these algorithms yield a decision on the loop's termination, the procedure may be exited. This process is more clearly presented in Figure 4.4.



**Figure 4.4:** A Flow diagram depicting the combined decision procedure of the techniques in Section 3.



### 4.3 Non-termination via recurrent sets

An algorithmic implementation of the recurrent solution presented in Section 3.2 involves two aspects: the construction of a candidate set  $\mathcal{R}_k$  for some period  $k$ ; and the check for  $\mathcal{R}_k$ 's possible recurrence.

Recall that the candidate set  $\mathcal{R}_k$  is defined by the condition

$$Q_k(\mathbf{r}) = (GT^l \mathbf{r} > \mathbf{0} \wedge GT^{l+k} \mathbf{r} \geq GT^l \mathbf{r}; \forall l = 0, \dots, k-1),$$

so that the construction required is that of a symbolic condition, which must be checked for recurrence. Since a certain number of periods are to be considered, say  $N$ , and the candidate conditions  $Q_k$  and  $Q_{k+1}$  are in certain ways similar, it is possible to construct each of the  $N$  candidate conditions in a dependent manner — the  $k$ th condition can be transformed into the condition concerned with the period  $(k+1)$ . Before the relationship between successive conditions is considered, note that the verification of candidate constraints shall be performed by satisfiability checking libraries; hence the recurrence of  $\mathcal{R}_k$  must first be translated into a form which can be understood by such libraries. In particular, the *CVC3* [5] satisfiability library is used; this solver was chosen as it is the only library currently integrated with JPF which supports linear combinations of integer variables over the scalar field  $\mathbb{R}$ . In practice, any satisfiability checker which supports linear combinations of this form may be used, however the majority of such solvers support only integer coefficients when integer-valued variables are present.

**Lemma 10.** *For a general affine loop  $L = (G, T)$ , the set of integer arrays  $\mathcal{R}_k$  which satisfy  $Q_k$  is recurrent under  $T^k$  if, and only if:*

1.  $\exists \mathbf{r} \in \mathbb{Z}^{n+1} : Q_k(\mathbf{r})$ , and
2.  $\{\mathbf{r} : Q_k(\mathbf{r}) \wedge \exists j : 0 \leq j < k \wedge GT^{j+2k} \mathbf{r} \not\leq GT^{j+k} \mathbf{r}\} = \emptyset$

*Proof.* Recurrence must be shown according to Definition 2. Firstly,  $\mathcal{R}_k$ 's non-emptiness is encoded by the lemma's first condition. Furthermore, any  $\mathbf{r} \in \mathcal{R}_k$  satisfies the loop's guard condition, as letting  $l = 0$  in  $Q_k$  reveals the inclusion of  $(G\mathbf{r} > \mathbf{0})$ . Lastly, the requirement that  $T^k \mathbf{r} \in \mathcal{R}_k$  for every  $\mathbf{r} \in \mathcal{R}_k$  is represented as a satisfiability check by the lemma's second

condition — note that

$$T^k \mathbf{r} \in \mathcal{R}_k \Rightarrow GT^{j+2k} \mathbf{r} \geq GT^{j+k} \mathbf{r} \quad \forall j = 0, \dots, k-1,$$

which, combined with the prerequisite  $Q_k(\mathbf{r})$ , implies condition 2. The converse holds similarly.  $\square$

By Lemma 10, the recurrence of the set  $\mathcal{R}_k$  is equivalent to the satisfaction of a constraint  $S_k$ ; the example constraints  $S_1$ ,  $S_2$ , and  $S_3$  follow:

$$S_1 = (G\mathbf{r} > \mathbf{0} \wedge G\mathbf{r} \geq G\mathbf{r}) \neq \emptyset$$

$$\wedge (G\mathbf{r} > \mathbf{0} \wedge G\mathbf{r} \geq G\mathbf{r} \wedge (GT^2 \mathbf{r} \not\geq G\mathbf{r})) = \emptyset;$$

$$S_2 = (G\mathbf{r} > \mathbf{0} \wedge G\mathbf{r} > \mathbf{0} \wedge GT^2 \mathbf{r} \geq G\mathbf{r} \wedge GT^3 \mathbf{r} \geq G\mathbf{r}) \neq \emptyset$$

$$\wedge (G\mathbf{r} > \mathbf{0} \wedge G\mathbf{r} > \mathbf{0} \wedge GT^2 \mathbf{r} \geq G\mathbf{r} \wedge GT^3 \mathbf{r} \geq G\mathbf{r} \wedge$$

$$(GT^4 \mathbf{r} \not\geq GT^2 \mathbf{r} \vee GT^5 \mathbf{r} \not\geq GT^3 \mathbf{r})) = \emptyset;$$

$$S_3 = (G\mathbf{r} > \mathbf{0} \wedge G\mathbf{r} > \mathbf{0} \wedge GT^2 \mathbf{r} > \mathbf{0} \wedge GT^3 \mathbf{r} \geq G\mathbf{r} \wedge GT^4 \mathbf{r} \geq G\mathbf{r} \wedge GT^5 \mathbf{r} \geq GT^2 \mathbf{r}) \neq \emptyset$$

$$\wedge (G\mathbf{r} > \mathbf{0} \wedge G\mathbf{r} > \mathbf{0} \wedge GT^2 \mathbf{r} > \mathbf{0} \wedge GT^3 \mathbf{r} \geq G\mathbf{r} \wedge GT^4 \mathbf{r} \geq G\mathbf{r} \wedge GT^5 \mathbf{r} \geq GT^2 \mathbf{r} \wedge$$

$$(GT^6 \mathbf{r} \not\geq GT^3 \mathbf{r} \vee GT^7 \mathbf{r} \not\geq GT^4 \mathbf{r} \vee GT^8 \mathbf{r} \not\geq GT^5 \mathbf{r})) = \emptyset.$$

Referring to constraints of the form  $(> \mathbf{0})$  as *positivity constraints*, those of the form  $(\geq GT^l \mathbf{r})$  as *increasing constraints*, and those of the form  $(\not\geq GT^l \mathbf{r})$  as *failure constraints* (since they cause recurrence to ‘fail’);  $S_{k+1}$  has one more positive constraint, increasing constraint, and failure constraint than  $S_k$ . To construct  $S_{k+1}$  from  $S_k$ :

- the positive constraint  $GT^k \mathbf{v} > \mathbf{0}$  is added to  $S_k$ ;
- $S_k$ ’s first increasing constraint —  $GT^k \mathbf{v} \geq G\mathbf{v}$  — is removed;
- the right-hand side of each of the  $k-1$  remaining increasing constraints is shifted to the neighbouring constraint of a higher degree (i.e.,  $GT^{k+l} \mathbf{v} \geq GT^l \mathbf{v} \rightarrow GT^{k+l} \geq GT^{l-1} \mathbf{v}$ );
- two new increasing constraints are added:  $GT^{2k} \mathbf{v} \geq GT^{2k-k-1} \mathbf{v}$  and  $GT^{2k+1} \mathbf{v} \geq GT^{2k-k}$ ;
- it is simplest to construct the set of failure constraints in its entirety for each period  $k$ .

Regarding Lemma 10: although the satisfaction of  $S_k$  implies the non-emptiness of  $\mathcal{R}_k$ , it is prudent to ensure this property (the first check for non-emptiness in  $S_k$ ) before the rest of  $S_k$  is checked, since the emptiness of  $\mathcal{R}_k$  is relatively easy to verify, and implies non-recurrence.

Practically, certain procedures associated with affine loop objects become invaluable during the implementation of this algorithm — particularly the ability to raise an affine loop’s transformation matrix to a given power, and to fetch the symbolic guard constraints of a given variable power. An overview of the implementation’s class structure is provided in Appendix B.2.

Regarding matrix powers: since the recurrent set technique requires the iteration of the loop’s transformation matrix, and entries of  $T^k$  may grow rapidly, the constructed constraints may contain large integer coefficients. For interest’s sake, the calculation of  $T^k$  is not performed by the repeated multiplication of  $T$ . Instead, the binary representation of  $k$  is used, thereby allowing  $T^k$  to be constructed from the product of matrices of the form  $T^{2^i}$ ; for example,  $T^{29} = T \cdot T^4 \cdot T^8 \cdot T^{16}$ , so that instead of performing 28 matrix multiplications, only 7 need be performed (the three operations present in the previous equation, as well as the 4 required to calculate the even powers up to  $T^{16}$ ).

The implementation of the recurrence-based termination verification algorithm incorporates a similar approach to the one detailed above.

## 4.4 Non-termination via Jordan decomposition

The non-termination check based on recurrent sets is comparatively simple, even elegant, when compared with the issues faced by the Jordan decomposition heuristics — these heuristics surpass the simple form of the affine loop used by recurrent sets, due to decomposition, and thus introduce further complications in the forms of complex numbers, arbitrary-precision floating-point numbers, polynomials and linear combinations. This is not to say the heuristics cannot successfully be implemented, only that the significant groundwork which was required when contriving the techniques does indeed translate to programmatic problem solving within the implementation.

Java classes have been created which allow for an elegant approach to most of this programming work — specifically the class representations of polynomials and linear combinations. These classes ease the transition of the heuristics from theoretical to practical, as far as possible. The remaining issue is that of arbitrary-precision real and complex numbers — a concept which can prove computationally awkward. As previously stated, the Apfloat library is used to handle real numbers of arbitrary precision. A high amount of precision (higher than that afforded by Java’s *double* data type) is sought, since the constraint objects which must be initialised should be done so with 64-bit accuracy. Before this is done, operations such as division shall be performed numerous times, and each operation may lead to the loss of some of a floating-point number’s precision.

To begin: the loop’s transformation matrix must be decomposed, making use of the Jordan decomposition procedure. It is during this procedure that floating-point numbers are introduced — the acquired matrices<sup>1</sup>  $P$ ,  $J$ , and  $P^{-1}$  may contain complex numbers with real and imaginary parts ranging over the real numbers, as opposed to the strictly integer entries of the matrices  $G$  and  $T$ . As previously mentioned, the Java linear algebra library Jblas has been used to enable the decomposition of diagonalisable matrices in the absence of Mathematica, but lacks support for defective matrices. In addition, the matrices yielded by Jblas’s decomposition are represented using the *double* data type, and thus limit the precision of the constructed constraints (after precision has been lost during arithmetic operations) to 32 bits. The Jblas library is a particularly useful tool, and an alternative when access to Wolfram’s Mathematica is not available, however the suite of tools offered by Mathematica is vast, and supports general Jordan decomposition with an arbitrary degree of floating-point precision; this software is thus the basis for the remainder of the discussion<sup>2</sup>. Making use of Wolfram’s own JLink library, which allows Mathematica queries to be called from within a Java program, results are returned as textual strings, and interpreted using the complex number class provided by the Apfloat

---

<sup>1</sup>The use of Section 3.3’s notation is continued here.

<sup>2</sup>As far as possible, the implementation attempts to remain independent of the vast array of mathematical procedures provided by Mathematica, so that in the absence of the software, the extension can still be applied competently to loops with diagonalisable matrices. Thus the implementation contains methods which might have been integrated more elegantly by means of a Mathematica query, but have instead been implemented using native Java libraries and the arithmetic operations provided by the Apfloat library, which is a constant feature of the implementation regardless of whether Jblas or Mathematica is used as the matrix decomposer.

library to form the matrices  $P$ ,  $J$ , and  $P^{-1}$ .

Before progressing to the topic of heuristics, the current discussion must be interjected with a note on the approximation of results: since precision is lost during arithmetic operations, a situation may arise when two values which should be equal are in fact slightly (according to the precision used) different. The relevant cases are, firstly, that in which two differing operations should yield matching results, and that in which a series of arithmetic operations involving non-integer values should result in a relatively neat rational value. Due to the importance of accurate values (such as the expectation that final results be integer valued) in the solutions presented in Section 3.3, an algorithm which recognises the intended form of a value, with a minimal margin of error, forms an vital part of the implementation. Specifically, the rounding of floating-point numbers considers the scientific form of the number, for example:  $3.99999 \times 10^{-1}$ , or  $124.9999 \times 10^{-2}$ ; the last few trailing digits of a given number are considered fallible — that is, they may be incorrect due to previously performed arithmetic. The given exponential form is compared to its neighbouring integers (in the case of  $3.99999 \times 10^{-1}$ : 3 and 4); if the difference, disregarding the few least significant digits, is zero, the neighbouring integer is assumed as the intended value of the given floating-point number. Clearly, a greater precision implies a narrower chance of false rounding.

Now, the implementation of the positive coefficient and positive weighted coefficient heuristics shall be addressed: clearly, some sets of polynomial coefficients must be created, according to the forms presented in the heuristics; subsequently, conditions must be constructed which constrain these coefficients in the desired manner, according to Heuristics 1 and 2. Each of the constraints detailed by these heuristics are done so in terms of linear combinations of the loop variables, and thus the final conditions which must be checked by the satisfiability library are unions of linear variable constraints — the reader may recall that the reduction of affine loop non-termination to linear constraints was an original goal of this text (Section 2.2).

Numerous forms of coefficients were defined in Section 3.3, as the discussion proceeded towards the non-termination heuristics. The coefficients implemented practically correspond to those introduced in Equation 3.3.19 — the coefficients used in the definition of the positive weighted coefficient heuristic. Since the exposition of these coefficients was previously rather extended, a more concise presentation of their construction shall be given; the practical initialisation of these coefficients is performed according to this presentation.

The final coefficients sought are those of the form  $D_{ij}(\mathbf{v}, k)$  (Equation 3.3.19). Given a loop  $L$ , with  $r$  guard constraints, and thus  $r$  exponential sums  $A_i(\mathbf{v}, k)$ :  $r$  abstraction sums  $B_i(\mathbf{v}, k)$  are to be constructed. Each sum  $B_i(\mathbf{v}, k)$  is a sum over  $s_i$  unique, positive bases, such that

$$B_i(\mathbf{v}, k) = \sum_{j=1}^s D_{ij}(\mathbf{v}, k) \lambda_j^k.$$

The bases  $\lambda_1, \dots, \lambda_s$  are the absolute values of the non-zero eigenvalues of the loop's transformation matrix  $T$ ; a coefficient  $D_{ij}(\mathbf{v}, k)$  is the sum of the lower bounds obtained for coefficients in  $A_i(\mathbf{v}, k)$  which correspond to eigenvalues whose absolute value is  $\lambda_j$ :

$$D_{ij}(\mathbf{v}, k) = \sum_{l=1}^{a_{ij}} \Lambda_{ijl}(\mathbf{v}, k) - \sum_{l=a_{ij}+1}^{b_{ij}} \Lambda_{ijl}(\mathbf{v}, k),$$

where  $a_{ij}$  and  $b_{ij} - a_{ij}$  count the number of such coefficients of positive eigenvalues and other eigenvalues respectively. Continuing: the lower bounds can be calculated from the ungrouped coefficients  $C_{im}(\mathbf{v}, k)$  in  $A_i(\mathbf{v}, k)$  — each corresponding to a Jordan block — as defined in Lemma 4:

$$C_{im} = \sum_{e=0}^{\psi_m} \left( \sum_{a=1}^{n+1} \sum_{b=1}^{n+1} \sum_{d=0}^{\psi_m} \sum_{s=0}^{\psi_m-d} g_{ib} I_{se} k_m^{-s} p_{b(u_m+d)} q_{(u_m+d+s)a} v_a \right) k^e.$$

Recall that the lower bound of such a coefficient depends on whether the coefficient is real or complex; in the case of a complex coefficient, the signs of the real and imaginary parts of the coefficient are of interest. Hence, practically, for each Jordan block corresponding to a real eigenvalue, a single real polynomial is stored; whereas for each Jordan block corresponding to a complex eigenvalue, two real polynomials are stored — corresponding to the real and imaginary parts of the coefficient; such a block's conjugate block can then be ignored, since it offers no new information for the lower bound process. With the positive coefficient heuristic, each of these polynomials (either  $C_{im}$ ,  $\text{Re}(C_{im})$ , or  $\text{Im}(C_{im})$ ) is constrained to positive or zero values, whereas the positive weighted coefficient heuristic may also require polynomials to be non-positive, and places additional conditions on their sums.

The positive weighted coefficient heuristic does engender its own unique caveats: since the implementation of this heuristic often requires a linear combination to be constrained to zero, the presence of imprecise coefficients can cause the technique to fail to detect a non-terminating

witness. Consider a loop with the guard and transformation matrices

$$G = [-4 \ -2 \ -2] \text{ and } T = \begin{bmatrix} -4 & -2 & -8 \\ -5 & -10 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

for which the non-terminating witness  $(x, y) = (-2, 1)$  engenders the cyclic path  $(-2, 1) \rightarrow (-2, 1) \rightarrow \dots$ , detected by the implementation of the recurrent set heuristic. The guard constraint in this case produces  $(-4x - 2y - 2 = 4 > 0)$ . The decomposition of this loop yields the exponential sum (in which coefficients have been abbreviated for legibility)

$$A(\mathbf{v}, k) = (-1.77x - 2.60y - 0.93)(-11.35)^k + (-2.22x + 0.60y - 5.06)(-2.64)^k + 4,$$

which is clearly only positive when the two leading exponential terms are zero. However, due to the loss of floating-point precision, the generated coefficients are not entirely accurate (regardless of the precision used), and the constraint solver cannot produce integer values for  $x$  and  $y$  which yield zero coefficients, and thus the decomposition heuristics fail to detect the non-terminating witness. This is the primary cause of the inaccuracies evident in the following chapter, regarding this technique.

Finally: in contrast with the implementation of the recurrent set technique, the coefficients within the generated constraints remain relatively small, since the loop need not be iterated to construct the constraints required by the heuristic.

## 4.5 Algorithm complexity

Proving non-termination by way of the recurrent set algorithm involves a certain number of recurrence checks. If the maximum period considered is  $N$ , the number of constraints present in condition 2 of Lemma 10 is  $3N$ , however a constraint which compares column vectors —  $(GT^{l+k}\mathbf{r} \geq GT^l\mathbf{r})$ , for example — engenders a constraint for each row. Thus the period  $N$  in fact engenders  $3Nr$  constraints, where  $r$  is the number of guard constraints in the loop's guard condition, as well as the row dimension of  $G$ .

The positive weighted coefficient heuristic is rather more complex to compute: every permutation of these coefficients is considered, and for each permutation, lower bounds are calculated,

and themselves constrained. Not only are positively and non-positively signed polynomials considered, but also zero-valued polynomials. If a loop engenders  $\ell \leq (n + 1)$  distinct eigenvalues and  $r$  guard constraints, the number of coefficients is  $(r \times \ell)$ , hence there are  $3^{r\ell} \leq 3^{r(n+1)}$  permutations — a rapidly growing function. To attempt to counteract the checking of large numbers of permutations, a polynomial's possible sign constraints are evaluated upon the polynomial's initialisation; if a certain sign constraint is not satisfiable, every sign permutation with such a configuration is not viable, and the number of possible permutations is reduced by a third. A similar reduction is yielded by ignoring permutations which are clearly terminating — such as those for which the leading term is strictly negative. In practice, these pre-emptive observations can award a marked reduction in the number of viable permutations. For example, the loop in Figure 4.5 engenders three distinct eigenvalues, and thus has a total of  $3^6 = 729$  possible permutations. In practice, however, the vast majority of these permutations are excluded — only 3 of them are considered 'viable' by the implementation, and used to create conditions which are checked by the satisfiability solver (a non-terminating witness for the loop is  $(x, y) = (2, 1)$ ).

```

while  $(x > 1 \wedge y > 0)$  {
     $x := x + y$ 
     $y := x - y$ 
}

```

**Figure 4.5:** A loop with 729 possible sign permutations, of which only 3 are considered.

Each permutation leads to the satisfiability check of a generated condition; a condition includes  $r\ell$  polynomial sign constraints, each of which engenders  $d$  constraints for a polynomial of degree  $(d - 1)$  (by the Upper/Lower Bound Theorem). Considering the Jordan loops matrix, recall that  $d\ell \leq (n + 1)$ . Lastly, the positive weighted coefficient heuristic requires  $r\ell$  sum constraints, so that each of the  $3^{r\ell}$  permutation conditions consists of at most  $(r\ell + r\ell d) \leq r(\ell + (n + 1))$  constraints.



## Chapter 5

# Evaluation

The assessment of the techniques discussed in the previous chapters is not a particularly simple matter: one wishes ideally to quantify in some way the percentage of loops for which a technique is successful, yielding insight into the extent of the developed techniques' error-detection capabilities. Such a quantification would rely on prior knowledge of the termination of the tested loops — problematic considering the pursuit of such knowledge is the reason for the development of the current techniques. A common method of evaluation with regard to termination is the application of derived techniques to large examples of program code, obtained, for instance, from open-source projects. Although this approach is warranted, and particularly practical, it fails to provide an adequate view of the completeness of a technique; instead, it highlights the existence of the errors for which the technique was developed in industrial projects. Numerous examples of loops whose non-termination can (or cannot) be detected by the given heuristics have been provided in the preceding sections, and additional examples await the reader in Section 5.2 as well as Appendix B.3; however, a focussed examination of the heuristics is also desired.

The proposition then, since affine loops are represented by integer matrices, is to gather termination results on a large number of different loop matrices; the number of loops whose termination is decided is not guaranteed to be the same as, or even very near to, the total number of examined loops. For this reason, each random loop whose termination is undecided shall be examined by the execution of many initial variable values — the termination of every one of these random paths indicates either a terminating loop, or one whose non-terminating behaviour is difficult to detect. Such loops shall be deemed 'likely' terminating, in an attempt

to provide a clearer view of the extent of each technique's non-termination detection.

Note that if one were to consider techniques which target more general loop forms — other than those of an affine nature — an automated evaluation such as the one adopted here might not be possible, since example loops might not be restricted to a form as particular as an integer matrix (this is the case regarding the general recurrent set technique [18], which attempts to detect and falsify any cycle within a program).

Extending further than the detection of non-termination, one of the goals of this text has been to gather as many witnesses to non-termination as possible. An evaluation of such sets of witnesses is a more formidable task than the former assessment: such an evaluation must compare the returned set to the set of all non-terminating witnesses, and thus necessarily requires concrete knowledge not only of a loop's possible non-termination, but also the specific conditions under which non-termination occurs. In lieu of this, this section of the discussion shall be limited to the application of the algorithms to a few example loops, unfortunately without the intention of drawing anything more than empirical conclusions.

## 5.1 Non-termination detection

Consider now the attempted falsification of numerous random affine loops. The implementation of such an assessment was eased by the objective nature of the JPF-symbc classes described in the previous chapter — an *AffineLoop* object can simply be initialised from integer matrices describing the guard and transformation matrices respectively, and the non-termination algorithms applied to this object as if it had been obtained from a symbolic execution of program code. Loops of between one and five loop variables have been considered, with guard and transformation matrices containing random integer entries ranging from  $-100$  to  $100$ . These loops are completely arbitrary: many may be trivially terminating (while  $(x > 0) \{x := -x\}$ , for example), even though guard constraints are guaranteed to be non-empty ; the goal is not to determine a ratio between terminating/non-terminating loops, but to quantify the ratio decided by each technique. In the case of an undecided loop, a search for non-terminating paths has been performed: 500 random sets of initial variable values have been generated, each with entries between  $-1000$  and  $1000$ , and executed to 200 iterations. If all of these paths terminate within the tested number of iterations, the loop may be terminating, and is deemed 'likely' to

terminate.

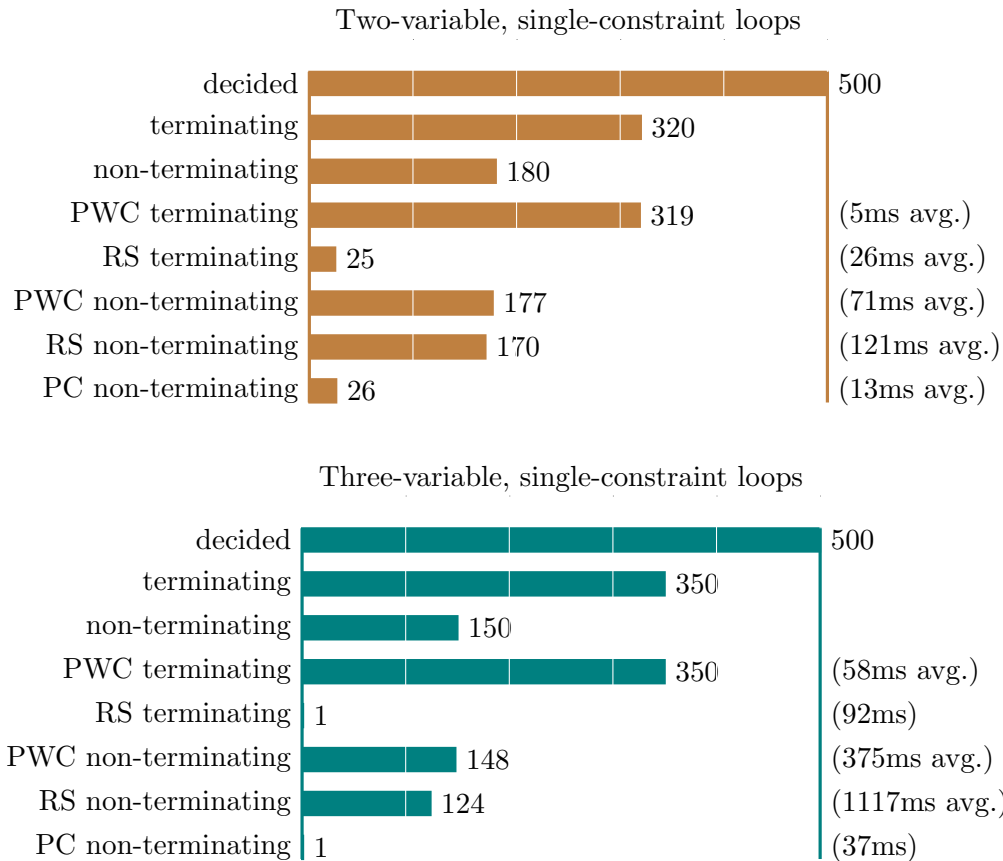
For two-variable loops with a single guard constraint, the application of the set of algorithms generally takes less than one second. The recurrent set algorithm scales to five variables, or fewer variables with additional guard constraints capably, its constraints being solved within a few seconds unless an exceptional case is located which seems to trouble CVC3. The positive weighted coefficient heuristic does not scale as comfortably (requiring lengths of time for various five-variable loops), since the complexity of the constraints grows exponentially — recall that a loop can engender  $3^{c(n+1)}$  sign permutations, where  $c$  is the number of guard constraints and  $n$  the number of variables.

In the figures displaying the results of the evaluations, certain abbreviations are used: the success of the recurrent set falsification algorithm of Lemma 1 is denoted by ‘RS non-terminating’, that of the positive coefficient and positive weighted coefficient heuristics (Heuristics 1 and 2) by ‘PC non-terminating’ and ‘PWC non-terminating’ respectively, and the recurrent set and sign permutation verification corollaries (Sections 3.2.3 and 3.3.8) by ‘RS terminating’ and ‘PWC terminating’ respectively. The anagrams RS, PC, and PWC shall be used throughout the chapter.

Before noting the results of the two-variable evaluation, recall that both heuristics can completely decide single-variable affine loops; in practice, this decision can be done quickly, and as such, these loops are omitted from this chapter with the assurance that either of the approaches is more than capable of handling them.

Consider then the results of the single-constraint tests in Figure 5.1: loops are either termed decided or undecided (regarding termination); decided loops are either terminating or non-terminating, and in each case the success rate of each algorithm is deducible according to the right-aligned vertical bar, which indicates the total number of tested loops (500). Fortunately, in the case of the sets of 500 randomised two- and three-variable affine loops, every loop was decidable by the algorithms developed in this text. Since the PC falsification heuristic produces low success rates in all of the evaluations, it shall be omitted from the rest of this discussion in favour of the remaining approaches. Notably, of the samples of 500 tested loops, the number of cyclic loops was negligible in both cases, unfortunately indicating that the decidability of cyclic loops is not of high practical importance.

Regarding execution time: the values provided alongside the bar charts in Figure 5.1 depict



**Figure 5.1:** Results of the application of the algorithms to single-constraint affine loops.

the average execution time (in milliseconds) of each algorithm. The recorded execution times exclude the initialisation of the JPF virtual machine (approximately 150ms) as well as the decomposition of each transformation matrix (roughly 100ms and 200ms for two- and three-variable loops respectively).

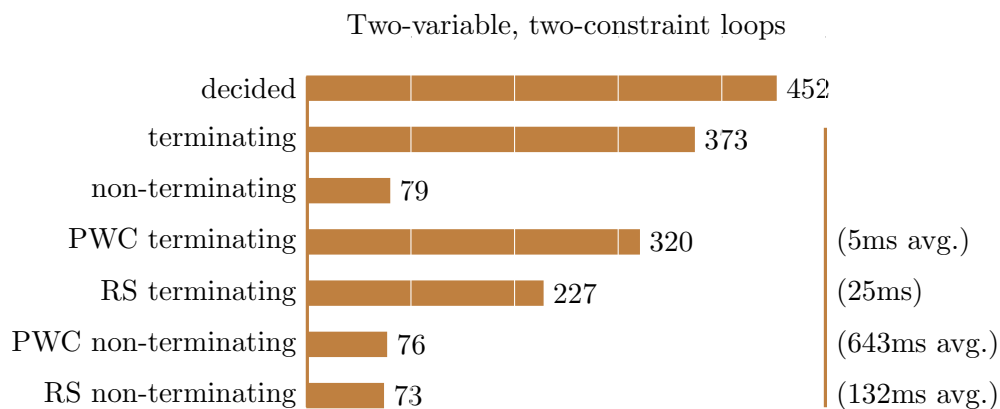
All of the algorithms completed in a small amount of time, save a few loops which caused the RS falsification algorithm to exceed the CVC3 time limit of thirty seconds. In the two-variable case, only one loop caused this heuristic to exceed the time limit, and once the period which generated this constraint was abandoned, the non-termination of the loop was proved over a larger period; the failure of CVC3 to decide one constraint did not, in this case, lead to the failure of the algorithm to locate a non-terminating witness. This outlier, which completed in 30184ms, was omitted from the dataset which produced the average time for the RS falsification heuristic. In the three-variable case, twenty loops caused CVC3 to exceed its time limit; again

all of these loops were proved non-terminating over larger periods, in total times of roughly 30 seconds, and were omitted from the average execution time dataset.

Furthermore, it may be interesting to note that none of the other techniques caused CVC3 time-outs, however a single outlier which took 21352ms to verify with the PWC terminating heuristic (presented in the latter portion of Appendix B.3) did raise this technique’s average time from 16ms to 58ms. Hence, the CVC3 time limit caused no adverse effect on the single-constraint loop evaluations, implying that the shortcomings of the techniques are theoretical, or, in unfortunate cases, due to implementation issues such as imprecisions in the representations of floating-point numbers.

Concerning the results depicted in Figure 5.1: note how the RS verification technique is unable to detect the majority of terminating loops; the primary recurrent set approach — the RS falsification technique — detects fewer non-terminating loops as the variable count is increased, implying a decrease in periodic non-terminating behaviour. The primary aspects of these single-costraint results, however, are the following: termination of the vast majority of the loops was decided, and for simple loops the PWC verification and falsification techniques perform particularly well — failing to decide only 6 of the total sample of 1000 loops.

Single-constraint loops provide a challenge, however the algorithms are faced with new issues when additional guard constraints are introduced:



**Figure 5.2:** The application of the algorithms to 500 affine loops with two variables and constraints.

As depicted in Figure 5.2, the majority of the tested loops were decidable by the developed

techniques. With the addition of a guard constraint, the RS verification algorithm became far more effective, and even though the PWC termination checker proved the termination of 73 more loops, it was the combination of both algorithms which led to the final tally of 373. With regard to falsification, both the RS and PWC approaches fared equally well, although the sets of loops falsified by each approach differ, since neither approach managed to falsify all 79 non-terminating loops. The 48 undecided loops were all marked ‘likely’ terminating; if these loops are in fact terminating, the weakness of the current techniques lies with the termination verification techniques — this is possible, as each was little more than a corollary, and their performance is surprisingly impressive — and not with the developed falsification heuristics, however it is also likely that non-terminating loops went undetected, as neither of the techniques are complete.

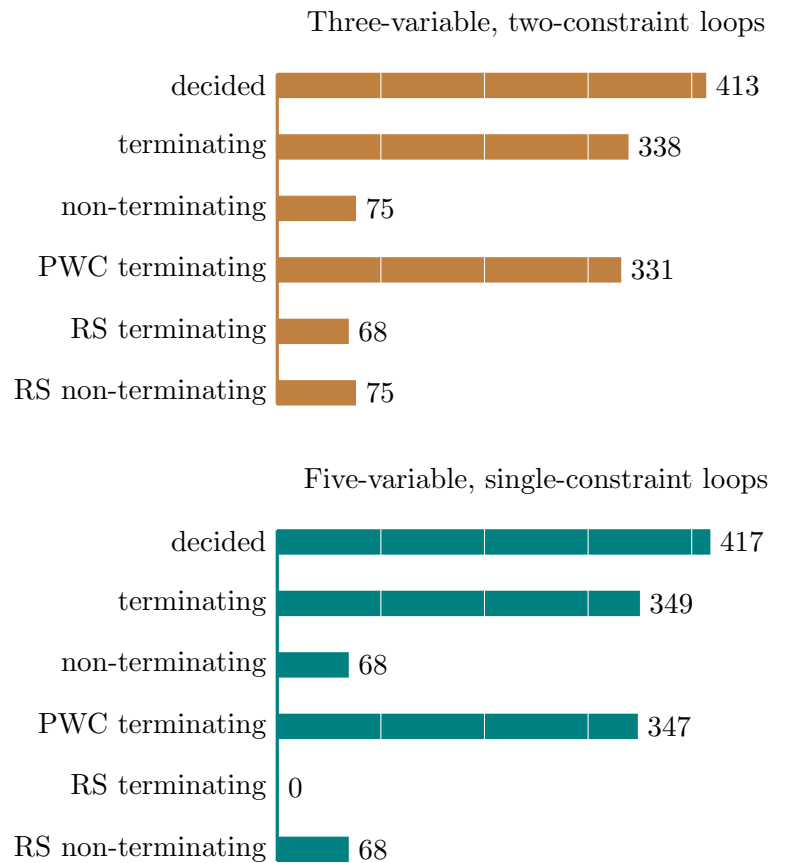
Regarding the execution times related to Figure 5.2, the expected growth caused by more complex constraints is apparent. Significantly, none of the RS falsification applications caused CVC3 to reach its time limit, suggesting that the addition of a constraint leads to more decisive candidate sets. Contrarily, however, the presence of multiple constraints has an adverse effect on the PWC falsification technique: 43 loops caused a sign permutation constraint generated by this technique to reach the time limit of CVC3, and only 15 of these were still proved non-terminating by some other permutation of the heuristic (these were once again omitted from the average time calculation). Of the remaining 28 for which the heuristic subsequently failed to draw a conclusion, 2 were proved non-terminating by the RS falsification technique, while 26 remained undecided by any of the techniques.

Note how a sample of two-constraint three-variable loops has not been considered: although a single-constraint loop with three variables requires the checking of simpler conditions than a single-constraint loop of five variables, the introduction of a second guard constraint raises the maximum number of sign permutations from  $3^4 = 81$  to  $3^8 = 6561$ , far more than the  $3^6 = 729$  upper bound for a single-constraint loop with five variables. The application of the PWC falsification heuristic to a wealth of more complicated randomised loops unfortunately falls outside of the scope of this implementation, as is alluded to in the previous paragraph. This is not to say the algorithm is not applicable to loops of larger dimensions — see Appendix B.3. A randomly generated loop, however, lacks the structure of common industrial loops, and a random sample likely includes loops which maximise the computational power required to

apply the heuristic; the computational stress of a loop is directly linked to the ability of the CVC3 satisfiability library to solve the generated conditions, and more complex loops more often cause the CVC3 satisfiability procedure to reach its time limit. It is also important to note that conditions are not simplified before being passed to CVC3; such a simplification would likely reduce the computational effort required.

Excluding the PWC falsification from the test suite, along with its numerous conditions, allows the higher-degree results presented in Figure 5.3 to be obtained. The only significant information contained within these results, however, is the number of loops decided by the PWC verification and RS falsification techniques — conclusions regarding their completeness cannot be drawn, due to the large number of undecided loops (87 and 83, of which 85 and 68 respectively were marked ‘likely’ terminating). The omission of the PWC heuristic seems to leave many loops undecided, as is to be expected, since neither of the falsification algorithms detected all of the non-terminating loops in Figure 5.2. In fact, the number of undecided loops exceeds the number of non-terminating loops in both cases.

To conclude this section, a note concerning the inherent structure of loops found in program code as opposed to those constructed from randomly generated matrices: common properties such as equal matrix entries or simple guard constraints, in the experience of the authors, often lead to more manageable decompositions, particularly engendering polynomials with simple rational, or even zero-valued, coefficients. Exponential sums containing simpler coefficients in turn produce constraints which are easier to solve; or, on the other hand, which may contain non-terminating behaviour which is particularly difficult to detect. One noticeable feature of the randomly generated matrices considered here is their density — a zero-valued entry seldom appears, since 0 is given the same probability as any other integer; this is dissimilar to industrial loops. In an attempt to generate loops which more naturally resemble those which appear in program code, an additional set of tests were run in which matrix entries were given a fifty percent chance of being zero; the results were similar to those of Figures 5.1 and 5.3, with a slight increase in the number of decided loops compared to Figure 5.3. Another pertinent property of these matrices is the large variety present in their entries — seldom are an affine loop’s coefficients arbitrary entries from the range  $-100$  to  $100$ ; to address this, the single-constraint tests were duplicated with entries of absolute value no greater than 20, and subsequently 5, once again yielding no significant differences.



**Figure 5.3:** The application of the algorithms, excluding the PWC falsification heuristic, to larger affine loops.

From the experimental results, it appears that the algorithms are almost perfectly capable of deciding the termination of practical (relatively uncomplicated) affine loops, however, due to the combination of complex conditions for large loops, a further-optimisable implementation, and the inability of CVC3 to handle the induced over-complicated conditions, the results decrease in strength as more complex loops are considered.

## 5.2 Conditional non-termination

Recall that the initial reason for developing a Jordan decomposition-based falsification technique was due to the periodic limitation of the recurrent set technique. In addition, it was



supposed that the decomposition of the loop would shed light on the loop's mechanics, perhaps allowing one to gain a greater understanding of the conditions under which the loop is non-terminating. The periodic weighted coefficient heuristic was developed to capture as many non-terminating witnesses as possible. Although its performance in this regard cannot be measured as automatically as falsification soundness was in the previous section, several examples of its practical applicability have been grouped here. All of the example loops presented here consist of sequential (as opposed to simultaneous) affine transformations.

$$\begin{array}{l} \text{while } (x > 0) \{ \\ \quad x := x + y \\ \} \end{array}$$

**Figure 5.4:** A two-variable non-terminating loop [13].

Firstly, consider Figure 5.4's loop: a technique detailed in the literature [13] yields the termination condition  $(x \leq 0 \vee y < 0)$ , which is in fact the weakest precondition for termination of this non-terminating loop. If one had not inspected the loop, however, the completeness of this termination condition would not be clear. Note therefore that the application of the PWC heuristic to the loop returns the (weakest) non-termination precondition  $(x > 0 \wedge y \geq 0)$  — the converse of the synthesised termination condition. In this ideal situation, the termination properties of the loop in Figure 5.4 have been completely decided.

$$\begin{array}{l} \text{while } (x > 0) \{ \\ \quad x := x + y \\ \quad y := y + z \\ \} \end{array}$$

**Figure 5.5:** A three-variable non-terminating loop [13].

Yet another example from the literature, the synthesised termination condition for the loop

in Figure 5.5 is given [13] as

$$\begin{aligned} x < 0 \vee z < 0 \vee (z \leq 0 \wedge y < 0) \vee x + y \leq 0 \\ \vee x + 2y + z \leq 0 \\ \vee x + 3y + 3z \leq 0, \end{aligned}$$

and captures basic termination conditions along with those related to the first three iterations of the loop. The PWC heuristic returns (before simplification) the non-terminating condition

$$x > 0 \wedge y > 0 \wedge \frac{1}{2}z > 0 \wedge x + y > 0,$$

which is easily reduced to

$$x > 0 \wedge y > 0 \wedge z > 0,$$

a logical non-termination condition for the loop. Note that the path generated by the initial values  $(x, y, z) = (10, -2, 0)$  is not covered by either of the conditions, but would fall under the synthesised termination condition had it been applied to higher iterations.

```
while (x > 0) {
  x := x + 2y + 1
  y := 0
}
```

**Figure 5.6:** A three-variable non-terminating loop.

Figure 5.6 presents a loop which is non-terminating if, and only if,  $(x > 0 \wedge x + 2y + 1 > 0)$ . The PWC heuristic returns the non-termination condition  $(x > 0 \wedge x + 2y > 0)$ , failing to detect the non-terminating possibility of  $(x + 2y = 0)$ . For interest's sake, if  $\mathbf{v}$  denotes the variables  $x, y$ , the exponential sum generated by this loop is (where  $0^k$  denotes the Kronecker Delta function):

$$A(\mathbf{v}, k) = (x + 2y + k)1^k - (2y)0^k.$$

Furthermore, if the update transformation  $y = 0$  is removed loop from the loop under consideration (and replaced with  $y = y$ ), then the loop is non-terminating if, and only if,  $(x > 0 \wedge y \geq 0)$

— this is the condition returned by the PWC heuristic, simplified from

$$x > 0 \wedge 2y + 1 \geq 0.$$

```

while (x - y > 0) {
    x := x + z
    y := y - w
}

```

**Figure 5.7:** A possibly non-terminating loop.

The loop in Figure 5.7 depends on the balance between the variables  $x$  and  $y$ . If the negative variable grows faster, the loop shall terminate ( $z$  and  $w$  do not alter their value). The PWC heuristic returns the weakest precondition

$$x - y > 0 \wedge z + w \geq 0.$$

The previous few loops should provide a sufficient view of the capabilities of the PWC heuristic; for further examples the reader may proceed to Appendix B.3.

### 5.3 Summary

The reader has been provided with an application of this text’s techniques to samples of random affine loops, revealing the ability of the techniques to decide termination on the vast majority of smaller loops.

This chapter should also explain why it is the opinion of the authors that falsification and verification techniques are nothing but complimentary when combined. By practical standards, the termination of affine loops seems to be a manageable problem; in addition, reasonable non-termination preconditions are often within reach. As a final note on conditional termination, consider that termination conditions — a topic which has received only slightly more attention than non-termination conditions [13] — could in fact be extracted from the verification technique described in Section 3.3.8, which is based on sign permutations. The conditions generated by the permutations which clearly induce termination promise to provide a good idea of the loop’s terminating behaviour.

Not only this, but the non-termination witness approximations returned by the presented heuristics may do much to augment the known approaches to the approximation of termination conditions; the knowledge of subsets of the terminating and non-terminating generators for a loop allow for a more focussed investigation of the undecided initial values — in the best case these may even be of a finite number. Such values can be inserted into the exponential sum descriptions of the given loop, potentially granting insight into their termination which was overlooked by a technique such as the positive weighted coefficient heuristic.

## Chapter 6

# Conclusion

The topic of non-termination has been discussed, particularly concerning simple affine program loops; two approaches have been investigated: that of recurrent sets, and another based on Jordan decomposition and exponential sums.

The recurrent set technique is not a novel one, however its specific restriction to affine loops has not yet been studied in detail. The investigation described in this text made use of the concept of periodic monotonicity to obtain completeness for certain affine loops, and yielded the decidability of single-variable loop termination. The decision procedure for single-variable affine loops is particularly simple. In addition, a concrete constraint template, as well as a practically useful upper bound on the number of periods which should be checked for recurrence were discovered. This approach is useful for small affine loops, but one would likely desire a more thorough technique when attempting to prove non-termination.

It might interest the reader to recall the obstacles encountered in the subsequent development of the positive weighted coefficient heuristic — obstacles which arose from the Jordan decomposition of the general affine loops. In the diagonalisable case, exponential sums were obtained whose coefficients are linear in the loop variables; in the general case, these coefficients contained binomial coefficients in the iteration variable  $k$ , and were thus transformed into polynomials in  $k$ . Since the exponential sum remains explicitly unsolvable, the first compromise was encountered in the search for an approximate solution: attempting to give weight to the coefficients of positive terms so as to induce non-termination; the mere comparison of such coefficients required a further abstraction in the form of a lower bound for the exponential sum

which involves only positive eigenvalues and polynomials with real-valued coefficients. A further requirement was the constraining of polynomial signs, leading towards the third and final incomplete technique: the application of the upper bound theorem to quickly obtain constraints which imply a polynomial's (non-)positivity.

In the shadow of such obstacles, it is perhaps remarkable to note the success with which the heuristic was applied to random, as well as notable, affine loops. Indeed, as one would predict from Chapter 3, the implementation of the decomposition heuristics is not an elementary programming task, nonetheless it is a rewarding one, as reflected in the results of Chapter 5. In a practical sense, this implementation must be performed with a focus on optimisation; the authors' implementation could not comfortably scale to large samples of loops of higher complexity (those with large numbers of variables and constraints), and the algorithm's ability is closely related to that of the constraint solver used.

An unexpectedly practical algorithm was that of Lemma 8 — the verification algorithms derived from sign permutations. This approach is computationally efficient in practice, never taking more than a few seconds, and managed to verify a large number of loops in Chapter 5, regardless of their size. Recall that this approach was also able to verify the termination of loops for which no linear ranking functions exist, and is not restricted to periodic behaviour.

Since the Jordan decomposition of an affine loop yields a perfect representation of the loop's mechanics in the form of exponential sums (Lemma 4), it is the authors' opinion that termination techniques based on this decomposition, particularly those which are concerned with obtaining reasonable preconditions, shall in turn be more complete than those techniques which are restricted to considering certain periods of the loop, but equivalently more expensive computationally. Clearly, however, the decomposition technique presented in this text is only applicable to loops of an affine nature.

## 6.1 Further work

### 6.1.1 Termination verification and conditional termination

The most appealing area of possible research arises from the unexpected aptness of the termination verification technique suggested in Section 3.3.8 — Chapter 5 displays the practical ability of this method, and it remains scalable. Since it targets specific characterisations of

loop variables which induce termination, the termination preconditions it yields shall likely be particularly useful.

### 6.1.2 Complex loop forms

The approaches of this text, and specifically those based on Jordan decomposition, may be generalised in a weak manner to consider nested loops and conditional loop bodies, however the concept of extracting rigorous conditions to describe non-termination rests on the requirement that the body of a loop remain constant. Hence, if one were to generalise these techniques, one would need to do so for specific execution paths within a loop's body — for example, one might search for infinite paths which appear when a certain path of a nested *if* block is continually selected. On the other hand, if a simple affine loop is contained within the body of another loop, the non-termination of the nested loop induces the same property to its parent.

If the guard condition of an affine loop is expanded to include the *or* constraint operator, the techniques could be adapted to consider each constraint argument of the operator individually, as the infinite satisfaction of only one such argument leads to non-termination.

Another consideration which may have occurred to the reader is that the deductions of Section 3.3 are not necessarily dependent on the fact that the affine loop's matrix consists of integer entries — the technique could be generalised to matrices with rational entries, thereby allowing for affine transformations with rational coefficients.

### 6.1.3 Test case generation

It is an elementary conclusion, when one encounters symbolic test case generation, that non-terminating loops can cause incongruities within the resulting variable constraints. The preconditions generated by the approaches of this text shall allow such generators to more accurately describe the behaviour of test case executions in the presence of affine loops.

There remains much ground to cover with regards to program termination; the contents of this text are merely a combination of verification desires [14, 18, 17] and affine loop properties [28]. These discussions have hopefully imparted to the reader the opinion that the termination problem, often viewed as a dead end, is in fact an exciting avenue of research.

# Appendix A

## Supporting Concepts

### A.1 Simple program representation

A simple program  $P = (\mathcal{V}, \mathcal{L}, \ell_0, \mathcal{T})$  consists of a set of variables  $\mathcal{V}$  which may assume values from a set  $\mathcal{X}$ , locations  $\mathcal{L}$ , an initial location  $\ell_0 \in \mathcal{L}$ , and a set of transitions  $\mathcal{T}$ . A program state  $\sigma$  is an assignment of values in  $\mathcal{X}$  to the variables in  $\mathcal{V}$ , and the set of program states is denoted  $\mathcal{S}$ ; a transition  $\tau$  is a triple  $(\ell, \rho, \ell')$ : consisting of pre- and post-locations  $\ell$  and  $\ell'$ , and a relation  $\rho$  over  $\mathcal{V}$  and  $\mathcal{V}'$ . A location-state pair,  $(\ell, \sigma)$ , is called a configuration, and a sequence of configurations beginning at the initial location (as well as an initial state, if a set of initial states has been defined) is termed a computation. The set of program configurations (any combination of location and state) is referred to as the state space, and the reachable state space of the system is the set of all configurations which appear in any computation — intuitively: a reachable configuration is one which can be reached from the initial location; reachable states and locations are similarly defined. The system's reachable state space is more valuable than its valid state space when addressing program verification, and can be notably smaller.

A program conforming to the structure defined above can be represented visually as a control-flow graph, in which vertices and edges depict locations and transitions, respectively [12]. Such simple programs suffice for the objectives of this text, and are commonly employed in the literature, where more extensive descriptions can also be found [20].



## A.2 Complex arithmetic

The complex conjugate operator satisfies a few distributive properties: for  $z_1 = a + bi$ ,  $z_2 = c + di \in \mathbb{C}$  and  $s \in \mathbb{R}$ ,

- $\overline{z_1 z_2} = \overline{(ac - bd) + (bc + ad)i} = (ac - bd) - (bc + ad)i = \overline{z_1} \times \overline{z_2}$ ,
- $\overline{z_1 + z_2} = \overline{(a + c) + (b + d)i} = (a + c) - (b + d)i = \overline{z_1} + \overline{z_2}$ , and
- $\overline{s z_1} = s \overline{z_1}$ .

Given a real matrix  $A$ , and some real eigenvalue  $\lambda$  of  $A$ , a real eigenvector corresponding to  $\lambda$  exists. Take any eigenvector  $\mathbf{v} \in \mathbb{C}$  corresponding to  $\lambda$ , and note that

$$\begin{aligned} A\mathbf{v} = \lambda\mathbf{v} &\Leftrightarrow \overline{A\mathbf{v}} = \overline{\lambda\mathbf{v}} \\ &\Leftrightarrow A\overline{\mathbf{v}} = \lambda\overline{\mathbf{v}} \\ &\Leftrightarrow A(\mathbf{v} + \overline{\mathbf{v}}) = \lambda(\mathbf{v} + \overline{\mathbf{v}}). \end{aligned}$$

Thus  $(\mathbf{v} + \overline{\mathbf{v}}) \in \mathbb{R}$  is an eigenvector corresponding to  $\lambda$ .

## A.3 Complex eigenvalues and Lemma 4

Assume that  $L = (G, T)$  is a general affine loop, retaining the notation of Lemma 4, and that  $T$  has a complex eigenvalue  $\lambda_a$ ; then  $\lambda_b = \overline{\lambda_a}$  is also an eigenvalue of  $T$ , and both eigenvalues have equal algebraic and geometric multiplicities. Furthermore, the eigenvectors of  $\lambda_a$  and  $\lambda_b$  can be chosen to form conjugate pairs, so that the eigenvalues engender similarly formed sets of Jordan blocks. Consider two Jordan blocks of dimension  $(\psi_a + 1)$ , located at indices  $u_a$  and  $u_b$ , generated by conjugate eigenvalues of  $\lambda_a$  and  $\lambda_b$  respectively; then the columns at indices  $(u_a + d)$  and  $(u_b + d)$  of  $P$  are pointwise conjugate, for all  $d = 0, \dots, \psi_a$ .

To show that the rows at indices  $(u_a + d)$  and  $(u_b + d)$  of  $P^{-1}$  are also conjugates, for  $d = 0, \dots, \psi_a$ , note that row  $i$  of  $P^{-1}$  is defined by the fact that its matrix product with column  $i$  of  $P$  is 1, and 0 with any other column. Assume then that columns  $a$  and  $b$  of  $P$  are conjugate; that is,

$$p_{lb} = \overline{p_{la}} \text{ for } l = 1, \dots, n + 1,$$

as the matrix  $T$  has dimension  $(n+1) \times (n+1)$ . It shall be shown that, given row  $a$  of  $P^{-1}$ , its conjugate is a valid choice for row  $b$  of  $P^{-1}$ . Then, since inverses are unique, rows  $a$  and  $b$  of  $P^{-1}$  must in fact be pointwise conjugate. Firstly,

$$\sum_{l=1}^{n+1} q_{al}p_{lc} = \begin{cases} 1 & \text{if } c = a, \\ 0 & \text{otherwise.} \end{cases}$$

Thus rows  $a$  and  $b$  of  $P^{-1}$  are conjugate if

$$\sum_{l=1}^{n+1} \overline{q_{al}}p_{lc} = \begin{cases} 1 & \text{if } c = b, \\ 0 & \text{otherwise.} \end{cases}$$

Well,

$$\sum_{l=1}^{n+1} \overline{q_{al}}p_{lb} = \sum_{l=1}^{n+1} \overline{q_{al}p_{la}} = \sum_{l=1}^{n+1} q_{al}p_{la} = 1.$$

Now, if  $c \neq b$ , and column  $c$  contains only real values,

$$\sum_{l=1}^{n+1} \overline{q_{al}}p_{lc} = \sum_{l=1}^{n+1} q_{al}p_{lc} = 0;$$

if column  $c$  is complex, then its conjugate column must also be present in  $P$  — say column  $d$  ( $d \neq a$  since  $c \neq b$ ). Then

$$\sum_{l=1}^{n+1} \overline{q_{al}}p_{lc} = \sum_{l=1}^{n+1} \overline{q_{al}p_{ld}} = \sum_{l=1}^{n+1} q_{al}p_{ld} = 0,$$

and it follows that rows  $a$  and  $b$  of  $P^{-1}$  are pointwise conjugate.

Using the knowledge that the Jordan blocks constructed from conjugate eigenvectors yield sets of columns in  $P$  and rows in  $P^{-1}$  which are conjugate, it shall be shown that pairs of conjugate exponential functions, with coefficients as in Lemma 4, yield real values.

Let  $\lambda_a, \lambda_b$  remain conjugate eigenvalues of  $T$ ; then, in the  $i$ th exponential function  $A_i(\mathbf{v}, k)$  of  $L$ , the coefficient  $C_{ib}$  is concerned with the Jordan block at  $u_b$  in  $P$ . Since the Jordan block at  $u_a$  contains conjugate columns,  $\psi_a = \psi_b$ , and

$$\begin{aligned} C_{ib} &= \sum_{e=0}^{\psi_b} \sum_{l=1}^{n+1} \sum_{m=1}^{n+1} \sum_{d=0}^{\psi_b} \sum_{s=0}^{\psi_b-d} g_{im} I_{se} \lambda_b^{-s} p_{m(u_b+d)} q_{(u_b+d+s)l} v_l k^e \\ &= \sum_{e=0}^{\psi_a} \sum_{l=1}^{n+1} \sum_{m=1}^{n+1} \sum_{d=0}^{\psi_a} \sum_{s=0}^{\psi_a-d} g_{im} I_{se} \overline{\lambda_a^{-s}} (\overline{p_{m(u_a+d)} q_{(u_a+d+s)l}}) v_l k^e \\ &= \overline{C_{ia}} \end{aligned}$$

It then follows that the sum of the terms generated by two conjugate Jordan blocks is real:

$$C_{ia}\lambda_a^k + C_{ib}\lambda_b^k = C_{ia}\lambda_a^k + \overline{C_{ia}\lambda_a^k} = 2 \operatorname{Re} \left( C_{ia}\lambda_a^k \right).$$

Furthermore, if  $u_\alpha$  and  $u_\beta$  are the indices of another pair of Jordan blocks for  $\lambda_a$  and  $\lambda_b$ , then

$$\begin{aligned} C_{ia}\lambda_a^k + C_{ib}\lambda_b^k + C_{i\alpha}\lambda_a^k + C_{i\beta}\lambda_b^k &= (C_{ia} + C_{i\alpha})\lambda_a^k + (C_{ib} + C_{i\beta})\lambda_b^k \\ &= (C_{ia} + C_{i\alpha})\lambda_a^k + (\overline{C_{ia} + C_{i\alpha}})\overline{\lambda_a^k} \\ &= 2 \operatorname{Re} \left( (C_{ia} + C_{i\alpha})\lambda_a^k \right) \\ &= \operatorname{Re} \left( 2(C_{ia} + C_{i\alpha})\lambda_a^k \right), \end{aligned}$$

so that all of the terms corresponding to a given complex eigenvalue and its conjugate may be grouped into a single, real-valued term.

The complex-based terms within an exponential sum of the form described in Lemma 4 thus together contribute a real value. Considering the remaining sum of real-based terms, it must also produce a real value; this in turn implies that its dominant term (or grouped pair of terms over odd- and even-powered sums, if the greatest eigenvalue is present in both positive and negative form) must have a real coefficient for large values of  $k$ . Because this coefficient is a polynomial, every one of its coefficients must in turn be real (otherwise the highest-degree non-real coefficient would eventually cause the polynomial to assume a complex value); thus the leading term has a real coefficient, and the same logic can be applied to the remaining sum to show that every term corresponding to a real eigenvalue has a real coefficient. If a combined term  $C_1|\lambda_1|^k + C_2(-|\lambda_1|)^k$  was present, note that  $\operatorname{Im}(C_1 + C_2) = 0 = \operatorname{Im}(C_1 - C_2) \Rightarrow \operatorname{Im}(C_1) = 0 = \operatorname{Im}(C_2)$ .

## Appendix B

# Implementation Notes

### B.1 Algorithms

Algorithm 2 constructs a tree representation of a loop’s guard condition directly from compiled bytecode, in which leaf nodes are constraints, while all other nodes are constraint operators, describing the grouping and nesting of the condition’s constraints. The success and failure indices of the condition being examined are represented by *succInd* and *failInd*; and during the iterative inspection of each constraint in the condition, the starting index and target of a constraint *c* are depicted by *c.startInd* and *c.targInd* respectively; for the sake of sensibility, *c*’s successor constraint is denoted *c.next*.

---

**Algorithm 2** Construct a tree representation of a guard condition from generated bytecode.

---

```

1: procedure PARSECONDITION(succInd, failInd, firstConst, lastConst)
2:   for all constraints c from firstConst to lastConst do
3:     if c.targInd = failInd then
4:       if c = firstConst then                                ▷ this must be an and condition.
5:         operator  $\leftarrow$   $\wedge$ 
6:         subconditions.add(c.not)
7:       else if (operator =  $\wedge$ )  $\vee$  (c = lastConst) then    ▷ a singular subcondition.
8:         subconditions.add(c.not)
9:       else                                                    ▷ the final subcondition is a nested or.
10:        subconditions.add(parseCondition(succInd, failInd, c, lastConst))
11:      end if

```

---

---

**Algorithm 2** Condition-building algorithm (continued).
 

---

```

12:   else if  $c.targInd = succInd$  then
13:       if  $c = firstConst$  then                                     ▷ this must be an or condition.
14:            $operator \leftarrow \vee$ 
15:            $subconditions.add(c)$ 
16:       else if  $(operator = \vee) \vee (c = lastConst)$  then
17:            $subconditions.add(c)$ 
18:       else                                                         ▷ the final subcondition is a nested and.
19:            $subconditions.add(parseCondition(succInd, failInd, c, lastConst))$ 
20:       end if
21:   else                                                         ▷ a compound subcondition.
22:        $tempTarg = c.targInd$                                        ▷ to locate the start of the next subcondition.
23:       for all constraints  $d$  from  $c.next$  to  $lastConst$  do
24:           if  $d.targInd \notin \{succInd, failInd\} \wedge d.targInd > tempTarg$  then
25:                $tempTarg = d.targInd$ 
26:           end if
27:           if  $d.next.startInd = tempTarg$  then ▷ the subcondition has been located.
28:               break for all loop
29:           end if
30:       end for                                                 ▷  $d$  is the subcondition's last constraint.
31:       if  $d.targInd = succInd$  then                               ▷ the (parent) condition is an or.
32:            $operator \leftarrow \vee$ 
33:            $subconditions.add(parseCondition(succInd, constraint\ at\ tempTarg, c, d))$ 
34:       else if  $d.targInd = failInd$  then                         ▷ the (parent) condition is an and.
35:            $operator \leftarrow \wedge$ 
36:            $subconditions.add(parseCondition(constraint\ at\ tempTarg, failInd, c, d))$ 
37:       end if
38:   end if
39: end for
40:   return subconditions
41: end procedure

```

---

## B.2 Class structure

What follows is a skeletal overview of the JPF extension, detailing the classes associated with each facet of the implementation. These classes cannot be ordered specifically, but an attempt has been made to do so in relation to the hierarchical structure within which they rely on each other. For interests sake, the entire implementation consisted of roughly 9500 lines of code.

LoopListener	The listener communicates with the JPF virtual machine to detect and gather data on <i>while</i> loops within Java bytecode. It makes use of the auxiliary class <i>LoopInstructionFactory</i> .
Loop	Generated by the listener, provides an abstract description of a bytecode loop, including the loop's <i>ConstraintTree</i> .
ConstraintTree	Portrays the guard condition of a while loop as a tree in which constraints are leaves, and other nodes denote constraint operators.
AffineLoop	An extension of the abstract <i>Loop</i> class which contains the data which defines the loop, as well as methods to interact with the loop's matrices as well as iterations and symbolic expressions thereof.
MatrixDecomposer	An abstract class describing the matrix operations which should be implemented by the two matrix decomposition classes <i>MatrixMathematica</i> and <i>MatrixJblas</i> .
ApcomplexMatrix	Represents complex matrices of arbitrary precision, since such matrices are generated during the decomposition process of an affine loop's transformation matrix.
LinearCombination	A linear combination of integer variables; coefficients are real-valued. Provides methods for constraint generation.
PolynomialCoefficient	Polynomial-formed coefficients which appear in the exponential sums of a Jordan decomposed affine loop. Relies on <i>LinearCombination</i> .

ApfloatOperations	Houses methods relating to Apfloat objects which are required when considering affine loops.
AffineLoopTermination	Contains implementations of the approaches described within this text.
AffineLoopEvaluator	Tests sets of AffineLoops (constructed from randomly generated matrices) for termination, to evaluate the recurrent set and heuristic solutions.
BinaryExpression	At the time of writing, SPF provided classes for compound expressions built strictly of integer or real expressions, but not both. This class provides support for combinations of real coefficients and integer variables.

### B.3 Example loops

This section catalogues select examples from the text, along with further examples and results which the reader may find of interest. Where significant, the execution time of the algorithm is given.

<pre>while (x &gt; 0) {     x := -x + 10 }</pre>	<p>Figure 3.3, non-terminating.</p> <p>Both the RS and PWC heuristics produce the weakest non-termination precondition</p> $x > 0 \wedge x < 10.$
<pre>while (x &gt; 0) {     x := 5x + y + z     y := 4y + 3z     z := -3y + 4z }</pre>	<p>Figure 3.18, non-terminating.</p> <p>Even though the exponential sum of this loop lacks a dominant term, both the RS and PWC heuristic detect non-termination.</p>

<pre> while (<math>x &gt; 0</math>) {   <math>x := x + y + 2</math>   <math>y := -x</math> } </pre>	<p>Figure 3.19, non-terminating.</p> <p>The variable <math>x</math> assumes the value 2 from the second iteration onwards; the PWC heuristic returns the weakest non-termination precondition</p> $(x > 0 \wedge x + y + 2 > 0),$ <p>whereas the RS heuristic is limited to periodic conditions, and for the period <math>k = 2</math> returns the non-termination precondition</p> $(x > 0 \wedge x + y + 2 > 0 \wedge 2 \geq x \wedge 2 \geq x + y + 2).$
<pre> while (<math>x \geq 0</math>) {   <math>x := -2x + 10</math> } </pre>	<p>Figure 3.25, from [24], terminating.</p> <p>Although no linear ranking function for this loop exists, the PWC termination technique manages to verify termination.</p>
<pre> while (<math>x &gt; 0 \wedge y &gt; 0</math>) {   <math>x := -8x + 6y</math>   <math>y := 2y</math> } </pre>	<p>Non-terminating.</p> <p>The PWC heuristic returns the non-termination precondition</p> $(y > 0 \wedge x = \frac{3}{5}y).$
<pre> while (<math>x &gt; 0</math>) {   <math>x := 9x + 2z + 8a + 6b</math>   <math>y' := -8y</math>   <math>z' := -2b - 8</math>   <math>a := -3y + 2z - 9a</math>   <math>b := -10b</math>   <math>y := y'; z := z'</math> } </pre>	<p>Non-terminating.</p> <p>The RS technique fails to falsify this five-variable loop within a 30 second time bound; the PWC heuristic returns the non-terminating witness <math>(x, y, z, a, b) = (1, 1, 99, -25, 0)</math> almost immediately.</p>



<pre> while (x &gt; 0 ∧ y &gt; 0) {   x := x + y   y := y - 1 } </pre>	<p>Terminating.</p> <p>The Jordan matrix of the loop is</p> $J = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$ <p>leading to the rather elegant exponential sums</p> $A_1(\mathbf{v}, k) = x + (y + \frac{1}{2})k - \frac{1}{2}k^2$ <p>and</p> $A_2(\mathbf{v}, k) = y - k,$ <p>from which termination is clear, and detected by both the RS and PWC techniques.</p>
<pre> while (x &gt; 0 ∧ y + 2 &gt; 0) {   x := -7x - 10y } </pre>	<p>Terminating.</p> <p>This loop is verified by the PWC corollary, but not by the RS corollary.</p>
<pre> while (x - 5 &gt; 0) {   x' := -4x - y - 19z + 13   y' := 5x - 14y + 13z + 20   z := -13x + 19y - 16   x := x'; y := y' } </pre>	<p>Terminating.</p> <p>This loop is verified by the PWC corollary in an unusually large 22 seconds; the RS corollary fails to prove termination.</p>
<pre> while (x - 20 &gt; 0) {   x := -8x + y - 9   y := 20y - 9 } </pre>	<p>Terminating.</p> <p>This loop is verified by the RS corollary (in 7ms), but not by the PWC corollary, as it engenders a positive leading eigenvalue.</p>

<pre>while (<math>x - 12 &gt; 0</math>) {     <math>x' := -4x + 3y - 15</math>     <math>y := 13x + 16y - 8</math>     <math>x := x'</math> }</pre>	<p>Non-terminating.</p> <p>This loop is proved non-terminating by the RS falsification algorithm — <math>(x, y) = (13, 27)</math> — but not by the PWC falsification technique.</p>
<pre>while (<math>x + 11 &gt; 0</math>) {     <math>x' := -18x - y + 16</math>     <math>y := 10x + 20y - 9</math>     <math>x := x'</math> }</pre>	<p>Non-terminating.</p> <p>The PWC heuristic produces the non-terminating witness <math>(x, y) = (1, -5)</math>, however the RS algorithm fails to prove non-termination.</p>

# Bibliography

- [1] *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
- [2] Bowen Alpern and Fred Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, September 1987.
- [3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.
- [4] Emil Artin and Arthur Milgram. *Galois Theory: Lectures Delivered at the University of Notre Dame*. Courier Dover Publications, 1997.
- [5] Clark Barrett and Cesare Tinelli. CVC3. In *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, 2007.
- [6] Aaron Bradley, Zohar Manna, and Henny Sipma. Termination analysis of integer linear loops. In *Concurrency Theory*, volume 3653 of *Lecture Notes in Computer Science*, pages 488–502. Springer-Verlag, 2005.
- [7] Mikio Braun. Jblas. <http://jblas.org>, January 2011. Version 1.2.0.
- [8] Mark Braverman. Termination of integer linear programs. In *Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 372–385. Springer-Verlag, 2006.
- [9] NASA Ames Research Center. Java pathfinder. <http://babelfish.arc.nasa.gov/trac/jpf>, November 2010. Version 6.

- [10] Alonzo Church. A note on the entscheidungsproblem. *Journal of Symbolic Logic*, 1(1):40–41, March 1936.
- [11] Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. Linear invariant generation using non-linear constraint solving. In *Computer Aided Verification*, volume 2742 of *Lecture Notes in Computer Science*, pages 420–432. Springer-Verlag, 2003.
- [12] Michael Colón and Henny Sipma. Practical methods for proving program termination. [1], pages 227–240.
- [13] Byron Cook, Sumit Gulwani, Tal Lev-Ami, Andrey Rybalchenko, and Mooly Sagiv. Proving conditional termination. In *Computer Aided Verification*, volume 5123 of *Lecture Notes in Computer Science*, pages 328–340. Springer-Verlag, 2008.
- [14] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Proving program termination. *Communications of the ACM*, 54(5):88–98, May 2011.
- [15] Oracle Corporation. Java. <http://www.java.com/en>, December 2011. Version SE 6.
- [16] Edsger Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
- [17] Patrice Godefroid. The soundness of bugs is what matters (position statement). In *Proceedings of BUGS'2005 (PLDI'2005 Workshop on the Evaluation of Software Defect Detection Tools)*, Chicago, June 2005, 2005.
- [18] Ashutosh Gupta, Thomas Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Rung-Gang Xu. Proving non-termination. In *Principles of Programming Languages*, pages 147–158. ACM New York, 2008.
- [19] Gerard Holzmann. Software analysis and model checking. [1], pages 1–16.
- [20] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):1–54, October 2009.
- [21] Sarfraz Khurshid, Corina Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568. Springer-Verlag, 2003.

- [22] James Kuzmanovich and Andrey Pavlichenkov. Finite groups of matrices whose entries are integers. *The American Mathematical Monthly*, 109(2):173–186, February 2002.
- [23] Andreas Podelski and Andrey Rybalchenko. Software model checking of liveness properties via transition invariants. Technical report, Max-Planck-Institut für Informatik, December 2003.
- [24] Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 239–251. Springer-Verlag, 2004.
- [25] Corina Păsăreanu and Neha Rungta. Symbolic pathfinder: symbolic execution of java bytecode. In *Automated Software Engineering*, pages 179–180. ACM New York, 2010.
- [26] Daniel Rockmore and Ki-Seng Tan. A note on the order of finite subgroups of  $gl(n, \mathbb{Z})$ . *Archiv der Mathematik*, 64(4):283–288, April 1995.
- [27] Yuri Shestopaloff. *Sums of Exponential Functions and Their New Fundamental Properties, With Applications to Natural Phenomena*. AKVY PRESS, 2008.
- [28] Ashish Tiwari. Termination of linear programs. In *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 387–390. Springer-Verlag, 2004.
- [29] Mikko Tammila. Apfloat for java. [http://www.apfloat.org/apfloat\\_java](http://www.apfloat.org/apfloat_java), November 2011. Version 1.6.3.
- [30] Alan Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of The London Mathematical Society*, Series 2, 42(1):230–265, 1937.
- [31] Alan Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69. University Mathematical Laboratory, 1950.
- [32] Helga Velroyen and Philipp Rümmer. Non-termination checking for imperative programs. In *Tests and Proofs*, volume 4966 of *Lecture Notes in Computer Science*, pages 154–170. Springer-Verlag, 2008.
- [33] Wolfram Research, Inc. Mathematica. <http://www.wolfram.com/mathematica>, 2010. Version 8.