Formal Methods for Control Engineering: A Validated Decision Procedure for Nichols Plot Analysis

A thesis to be submitted to the
UNIVERSITY OF ST ANDREWS
for the degree of
DOCTOR OF PHILOSOPHY

by

Ruth Hardy

School of Computer Science
University of St Andrews

February 2006

"Science is simply common sense at its best."

— Thomas Huxley

Abstract

Software and hardware implementations of control systems are commonly used to augment engineered products by enhancing performance features, such as handling qualities, mission critical features, such as energy efficiency, or safety critical features, such as stability. Classically, a control system is modelled by control engineers with mathematical formulae to represent the behaviour of the system and these formulae are analysed with respect to some design criteria. Classical control engineering techniques for this analysis are informal and numerical, typically involving the plotting and visual inspection of graphs. However, informal techniques provide no guarantee of the correctness of their results and numerical techniques only allow the analysis of the system at sample values rather than for all values.

This thesis presents work with automated formal mathematics applied in the field of control engineering. A method for formal symbolic analysis of control systems, based on the existing informal, numerical analysis technique, the Nichols plot, is introduced in terms of a widely applicable decision procedure. The procedure determines the positivity or negativity of a finitely inflective function, i.e. a function that has a finite number of points of inflection, based on its convexity. Systems for which this method is applicable are characterised, along with practical and technological limitations that limit the class of permissible systems.

The Nichols plot Requirements Verifier (NRV) implements this method to produce proofs that systems meet, or fail to meet, their Nichols plot requirements. NRV is implemented in the computer algebra system Maple, the formal theorem prover PVS and the quantifier elimination system QEPCAD. Maple is used for the manipulation and initial symbolic analysis of the mathematical formulae representing the control systems; PVS and QEPCAD

then provide a formal proof that Maple's results are correct and perform the final analysis of the system based on these results. The Maple procedures, extensive PVS libraries, and strategies that automate this process are introduced and the correctness of the underlying decision procedure for symbolic Nichols plot analysis is shown.

The technique of symbolic Nichols plot analysis is demonstrated using NRV on several moderate sized case studies and the results are compared with classical Nichols plot analysis. NRV allows not only the formal assurance of the correctness of results that is lacking in classical Nichols plot analysis but also the analysis of systems for all values of a parameter rather than for sample values only.

I, Ruth Hardy, hereby certify that this thesis, which is approximately 40000 words in length, has been written by me, that it is the record of work carried out by me, and that it has not been submitted in any previous application for a higher degree.

date 23.08.2006 *signature of candidate*.

I was admitted as a research student in September 2001 and as a candidate for the degree of Doctor of Philosophy in September 2001; the higher study for which this is a record was carried out in the University of St Andrews between 2001 and 2006.

date 23.08.2006 *signature of candidate*

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of Doctor of Philosophy in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

date 23 Aug 2006 *signature of supervisor*

In submitting this thesis to the University of St Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and abstract will be published, and that a copy of the work may be made and supplied to any *bona fide* library or research worker.

date 23.08.2006 *signature of candidate*

Acknowledgements

Thanks go to John Hall (DSTL), Rick Hyde (Mathworks) and Yoge Patel (QinetiQ) for sharing their insights into control engineering, and to Rob Arthan (Lemma 1), Richard Boulton, Tom Kelsey (University of St Andrews), Colin O'Halloran (QinetiQ) and Nicholas Tudor (QinetiQ) for many helpful discussions.

I am grateful to Hanne Gottliebsen (Queen Mary, University of London) for allowing me to use her transcendentals library and for the useful discussions about PVS, which allowed me to benefit from her experience.

Thanks go to Tim Storer and Amanda Martinson for allowing themselves to be used as sounding boards and for the many useful discussions that followed. Thanks also go to Tim for providing many useful latex packages that were used in the typesetting of this thesis.

I am grateful to my supervisor, Roy Dyckhoff, and second supervisor, Steve Linton, for their continuing help and guidance and to Ursula Martin for the help and guidance she provided, especially in the early stages of this work.

Thanks also go to my family for the support and encouragement they have given me throughout my studies and to David whose patience, support and love has been invaluable during my work toward this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis describes research into integrating automated formal and symbolic analysis into the classical analysis of control systems via Nichols plot analysis. An introduction to the relevant aspects of classical control system development and analysis is given along with a discussion of existing formal and symbolic techniques for control system analysis. A widely applicable procedure for deciding the positivity or negativity of finitely inflective functions is given, along with a description of an implementation of this procedure specifically applicable to formal and symbolic Nichols plot analysis of control systems.

Standard notation from the fields of control engineering, computer algebra and theorem proving are used where possible throughout this thesis. Where conflicts in notation arise one convention is chosen. This thesis adopts the convention in control engineering of referring to the mathematical formulae modelling the behaviour of a control system, rather than an implementation in software or hardware, as the control system.

In Section 1.1 the aims and motivation for the research are described. The main hypothesis is detailed in Section 1.2. Section 1.3 contains a summary of the main results and achievements of the work. An overview of the structure of this thesis is given in Section 1.4.

Several papers on the research presented in this thesis have been published. In Boulton et

al [23] an overview is given of the early stages of research (led by Ursula Martin) toward integrating formal methods into control system analysis. This represents the initial investigations into the use of formal methods in the analysis of control systems based on classical graphical analysis techniques. The paper presents an incomplete set of conditions for determining the positivity/negativity of a convex or concave function and demonstrates their use in the analysis of a very simple control system. In [58, 59, 60], I present the results of later stages of my research, including a complete set of conditions for determining the positivity/negativity of a convex or concave function (see Section 5.3), a decision procedure for determining the positivity/negativity of a finitely inflective function (see Section 5.7), the classification of formulae to which this decision procedure applies in terms of a logic $\mathcal{L}_1$ (see Section 5.4), and an implementation of this procedure designed specifically for the automated formal analysis of Nichols plot requirements (see Chapter 7). Related work in which I was peripherally involved is presented in [24] (see Section 3.3).

1.1 Aims and Motivation

The process for designing control systems is well established and well documented [22, 102, 108]. In traditional control engineering mathematical formulae modelling the behaviour of dynamical and control systems are developed, often as difference or differential equations. These formulae or *models* can be considered to be specifications for systems, which may later be implemented in software or hardware. In control engineering, the model rather than its implementations is referred to as the control system. Control systems are analysed with respect to some design criteria to ensure they display the correct behaviour. This analysis traditionally uses numerical techniques, often involving the visual inspection of a number of plots, or the performance of a number of simulations. Systems are analysed for a number of sample inputs and the assumption is made that the results of this analysis also hold for all the values between sample points. This assumption is not sound and may lead to incorrect conclusions being drawn about a system if it behaves unexpectedly between sample points. This problem is amplified when the precise values of parameters

are unknown (*uncertain parameters*); the system is only analysed for a limited number of values of the parameters (and combinations of values) and the assumption is made that the results of the analysis hold for *any* values of the parameters (and combination of values). Conclusions are drawn about the entire system without formal justification.

Although the control system requirements that are classically expressed graphically have precise underlying meanings that can be expressed formally, the analysis is performed informally, with the analyst relying on the visual representation rather than on a mathematical expression of the requirements as the basis for analysis.

The design and analysis of control and dynamical systems is the first stage in the development of a product. In the second stage software and/or hardware is developed from the models designed by the control engineer. The software (or hardware) is integrated to form the complete system and various tests are performed on it; for instance, rig testing and flight testing are performed on aeroplanes. The final stage of development is the certification of the system.

Since control system software and hardware are developed from the control system models and are designed to display equivalent behaviour, it is important to ensure that the models are correct. It is now widely accepted that the informal and numerical techniques for control and dynamical system analysis are not sufficient to ensure the correct behaviour of systems. Following the in–flight breakup of TWA Flight 800 in 1996, the National Transport Safety Board was forced to conclude that the design and certification process for aircraft fuel systems was flawed [100]. The fuel system of the aeroplane had been certified as "safe" by the Federal Aviation Administration (FAA), i.e. it had been certified that there was nothing that could ignite the fuel/air vapour in the tank. However, during the accident investigation, tests simulating the air pressure, altitude and fuel mass loading of TWA flight 800 showed that the temperature in the fuel tank could exceed the ignition temperature of the fuel/air vapour. The investigators found that the cause of the accident was poorly maintained hardware, which caused a spark that ignited the fuel/air vapour in the tank. Nevertheless, an underlying design flaw had been revealed, showing the certification to be unreliable. This

case highlights the problems inherent with simulation and numerical analysis; during the design process the system would have been tested for a range of different air pressures, altitudes, fuel mass loadings, etc, but the specific combination that was identified during the accident investigation was overlooked.

Owing to the inherent difficulty of identifying the exact cause of most catastrophic accidents and to the confidential nature of the aerospace and auto industries it is difficult to cite examples of flawed control system design causing accidents. It has, however, been acknowledged [108, p. 26] that several air–vehicle accidents, which were initially attributed to implementation faults, were, in fact, due to procedural failures or design errors, with the software performing exactly as specified.

The cost of implementing a change to a system increases substantially as the development moves from one stage to the next [108, p. 183]. For this reason, it is desirable to detect errors at as early a stage as possible.

There has not yet been a widespread integration of reliable formal and symbolic analysis techniques into control system development. In order for this to be achieved, the formal and symbolic analysis methods must be easy to use and must provide sufficient improvements over existing analysis techniques.

Symbolic techniques provide a clear benefit over numerical techniques as they allow analysis to be performed over a range of values rather than at specific sample points; however, they do not provide guarantees of correctness unless implemented in a suitably formal setting. Formal theorem proving techniques provide guarantees that results provided are correct; however, these techniques are largely unfamiliar to control engineers and are considered generally quite difficult to use, especially for those with little or no background in formal methods. This complexity of formal methods makes their integration into existing control system development infeasible without high levels of automation.

1.2 Main Hypothesis

In this thesis it is argued that

1. formal and symbolic methods can be integrated into classical informal and numerical analysis of linear, continuous–time, single–input single–output control systems in an unobtrusive manner,

2. using formal and symbolic techniques rather than the visual analysis of suites of plots increases the reliability of the results, both removing the possibility of erroneous results due to plotting errors and allowing systems to be analysed for all input values rather than just sample values,

3. formal and symbolic techniques can be used to analyse systems with uncertain parameters for all combinations of all permissible values of the parameters rather than just sample combinations,

4. the formal and symbolic analysis of Nichols plots in particular can be automated and integrated unobtrusively into classical control system development.

1.3 Results and Achievements

The underlying mathematical representation of a particular form of control system requirements — Nichols plot requirements — was examined. These requirements were reduced to their most basic form and a decision procedure was developed for use in the analysis of Nichols plot requirements. The procedure is widely applicable and can be used to decide the positivity or negativity of finitely inflective functions (as defined and explained in Chapter 5).

A logic $\mathcal{L}_1$ is developed to classify the formulae to which the procedure applies. The concept of a *minimal isolated* formula is introduced and a *quantifier isolation* algorithm to convert arbitrary formulae in $\mathcal{L}_1$ into this form is introduced.

The underlying theory of the procedure is built in the higher order theorem prover PVS as an extensive library. Proofs of the completeness and termination of the procedure and the quantifier isolation algorithm have also been developed in PVS.

The Nichols plot Requirements Verifier NRV uses the procedure as the basis for formal and symbolic Nichols plot analysis. NRV was developed in the Maple–PVS–QEPCAD system, which exploits the symbolic computation provided by the computer algebra system *Maple*, the formal techniques provided by the theorem prover PVS and the quantifier elimination routines provided by QEPCAD. Both *Maple* and PVS are industry standard tools and are highly programmable. PVS has a high level of built in automation and provides a powerful strategy language for developing further automation.

NRV is designed to be used by control engineers with little or no knowledge of formal methods. NRV provides a graphical user interface, similar in appearance to a java applet, which allows the user of the system to have no knowledge of the underlying decision procedure, formal methods or the *Maple* or PVS syntax.

NRV is highly automated and, in theory, can automatically produce a proof (if one exists) that a system meets its requirements or a proof (if one exists) that the system does not meet its requirements. In practice, several case studies have been performed and NRV has produced promising results rarely failing to find proofs. In cases in which a proof is not found automatically, NRV attempts to produce useful feedback to the analyst, indicating where the analysis failed, whether the failure was within a *Maple* computation or in a PVS proof, and highlighting areas that may require closer inspection.

It has been indicated by control engineers from companies such as DSTL, QinetiQ and The Mathworks that automated Nichols plot analysis would be useful if it could be applied to control systems of degree 5. NRV is highly successful in the analysis of control systems of this degree and has been successfully applied to control systems of up to degree 10.

1.4 Thesis Structure

In Chapter 2 a brief introduction to the relevant aspects of classical control system development is given. Various methods for the analysis of control systems in the complex and frequency domain are introduced and Nichols plots are discussed in detail. The commonly used tools for control system development are introduced and their strengths and weakness are discussed.

Chapter 3 discusses existing applications of formal and symbolic techniques to control system development and analysis. The benefits and particular usage of each method is highlighted and the areas of control system development that could benefit further from formal and symbolic techniques are highlighted.

The general topics of computer mathematics and theorem proving are introduced in Chapter 4. The strengths and weaknesses of computer algebra systems and theorem provers are introduced and shown to be complementary. The computer algebra system *Maple* and theorem prover PVS are introduced in some detail, highlighting some of the features that are important in the development of automated formal and symbolic formal methods for Nichols plot analysis.

In Chapter 5 a procedure for deciding the positivity or negativity of finitely inflective functions is described. The basic mathematical concepts upon which the procedure relies are introduced and the underlying set of conditions for determining the positivity or negativity of convex or concave functions is described. The formulae for which the procedure is applicable are classified in terms of a fragment of logic and an algorithm for converting arbitrary formulae into a form that can be analysed using the decision procedure. The procedure is described first for functions of one variable then extended to functions of two variables. The formalisation of the decision procedure in the higher order theorem prover PVS is described in Chapter 6 .

The method for applying the decision procedure to formal and symbolic Nichols plot analysis is described in Chapter 7. The requirements of any system in which this method is to

be automated are described and an implementation in the Maple–PVS–QEPCAD system, the Nichols plot Requirements Verifier (NRV), is introduced.

In Chapter 8 several moderate sized case studies using the NRV are presented. These case studies show different aspects of the NRV and illustrate the improvements this makes over classical Nichols plot analysis.

In Chapter 9 conclusions are drawn about the applicability of formal and symbolic methods to classical control system analysis. The success of the method for formal symbolic Nichols plot analysis and of the NRV are discussed. Directions for further development of formal and symbolic Nichols plot analysis are discussed along with potential improvements to the NRV. Other applications for the underlying decision procedure, both within and outwith the field of control engineering, are discussed.

An introduction to the analysis of control systems in the time domain is omitted from Chapter 2 as this thesis does not deal directly with this form of analysis; however, for completeness, a brief introduction to this is presented in Appendix A, along with an introduction to a commonly used form of controller, the PID (Proportional/Integral/Derivative) controller.

The PVS libraries for the formalisation of the decision procedure and it foundations are given in Appendix B and the libraries used in the formal analysis of control systems are given in Appendix C.

Chapter 2

Classical Control System Development

Many modern dynamical systems require a software or hardware control system to control their behaviour in order to meet certain design criteria. These criteria can be performance related, such as having desirable handling qualities, or safety or mission critical, such as being stable or energy efficient. These software or hardware control systems are implemented from the control systems as represented mathematically by control engineers.

This chapter presents some of the basic concepts and methods used by control engineers in the development of linear[1], continuous–time, single–input single–output control systems. A detailed treatment of these concepts can be found in any standard text book on control systems, such as [43, 102, 124]. This chapter focuses on the modelling and analysis of control systems in the complex and frequency domains and highlights the informal and numerical nature of classical analysis techniques.

In Section 2.1 various mathematical representations used in modelling continuous–time systems are presented. Section 2.2 introduces classical control engineering methods for analysing systems with respect to various design criteria, focusing particularly on the anal-

[1]For a system to be *linear* it is necessary that the *principle of superposition* and the property of *homogeneity* hold. The *principle of superposition* holds if, given that the response of the system to an input x_1 is y_1 and the response to x_2 is y_2, the response of the system to $x_1 + x_2$ is $y_1 + y_2$. The property of *homogeneity* holds if, given that the response of a system to an input x is y, the response of the system to βx is βy.

ysis of systems using Nichols plots. In Section 2.3 the properties of computer–aided control system design (CACSD) packages are described with reference to some of the most commonly used systems.

2.1 Modelling Dynamic Systems

Dynamical systems can be considered in several ways and at several different levels of abstraction. They are generally considered in terms of their response or behaviour when exposed to some input or force. In general, systems are modelled mathematically in one of three domains: (1) the time domain, (2) the complex domain (also known as the *s–domain*), (3) the frequency domain. In this section several different representations are introduced.

2.1.1 Differential Equation

Systems of ordinary or partial differential equations (ODEs or PDEs) are used to model the outputs of continuous-time systems in terms of the rates of change of their state variables over time. A general linear differential equation of order n for a system with input $u(t)$, output $y(t)$ and initial conditions $y(0) = y_0$, $y'(0) = y_1, \ldots, y^{n-1}(0) = y_{n-1}$, is shown in Equation 2.1.

$$a_n y^n(t) + a_{n-1} y^{n-1}(t) + \cdots + a_1 y'(t) + a_0 y(t) = u(t). \tag{2.1}$$

Example 2.1 *A simple spring/mass/damper system (see Figure 2.1), with the parameters m, k and b representing the mass of the body attached to the spring, the spring constant (the ratio of force applied to the spring and the corresponding displacement), and the coefficient by which friction is proportional to the velocity of the body, respectively. The input to the system $u(t)$ is the external force placed upon the system and the output $y(t)$ is the resulting displacement of the body at time t.*

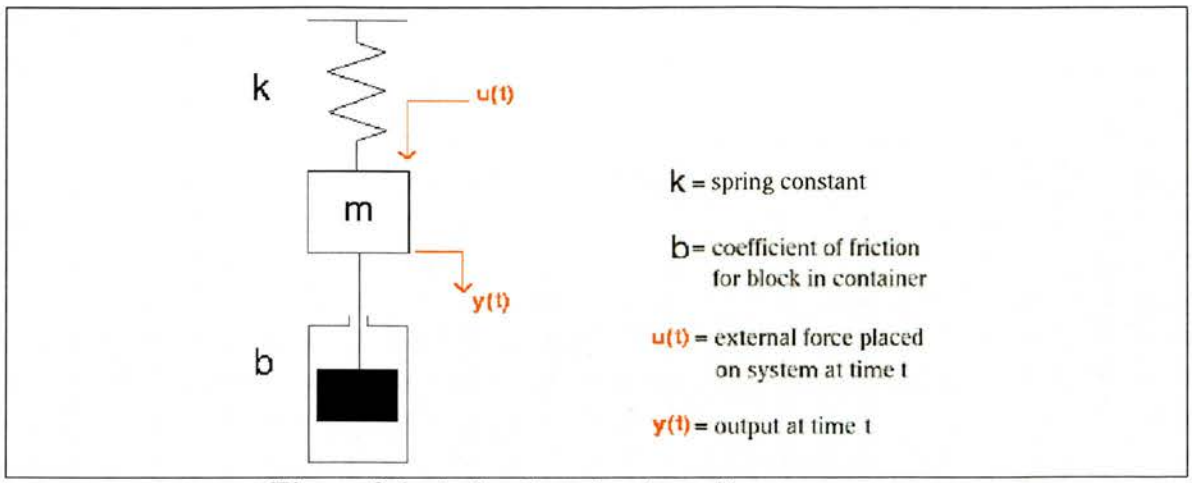


Figure 2.1: A simple spring/mass/damper system.

The differential equation for the spring/mass/damper system of Figure 2.1 is:

$$my''(t) + by'(t) + ky(t) = u(t) \tag{2.2}$$

with the initial conditions given by equations

$$y(0) = 0,\ y'(0) = 0.$$

2.1.2 Laplace Transform and Transfer Function

The Laplace transform provides a method for deriving a representation of a system in the complex or s domain from equations representing it in the time domain, with

$$\mathcal{L} : \mathbf{R}^{\mathbf{R}} \longrightarrow \mathbf{C}^{\mathbf{C}}.$$

Given a continuous–time linear system, the resulting equation is a rational equation in the complex variable s, which is comparatively easy to solve.

The Laplace transform of a function f of time is defined in terms of a *transformation integral* [2]

$$\mathcal{L}[f] = F = \lambda s.\ \int_0^\infty f(t)e^{-st}\,dt. \tag{2.3}$$

[2]The definition given here is of the *unilateral Laplace transform* rather than the *bilateral Laplace transform* as we are interested only in the former.

The Laplace transform for linear differential equations exists only if the transformation integral converges. For f to be transformable, it is sufficient that

$$\int_0^\infty |f(t)|e^{-\sigma t}\,dt < \infty \tag{2.4}$$

for some real positive σ. If there exist real constants M and α such that $|f(t)| < Me^{\alpha t}$ for all positive t, the integral will converge for $\sigma > \alpha$ [43, 41].

Using the transformation integral, the Laplace transform of the derivative of a signal over time is calculated as

$$\mathcal{L}[f^n] = \lambda s.\, s^n F(s) - s^{n-1}f(0) - s^{n-2}f'(0), \ldots, -f^{n-1}(0). \tag{2.5}$$

This is an important transformation and can be substituted directly into sets of differential equations to give the Laplace transform. For instance, given a general linear differential equation (see Equation 2.1) with all initial conditions set to zero (i.e, $f(0) = 0$, $f'(0) = 0, \ldots, f^{n-1}(0) = 0$), the Laplace transform is

$$a_n s^n Y(s) + a_{n-1}s^{n-1}Y(s) + \cdots + a_1 sY(s) + a_0 Y(s) = U(s). \tag{2.6}$$

The Laplace transform can be used to produce the complex valued *transfer function*, which represents the ratio of the output variable to the input variable in the s–domain. The transfer function $F(s)$ is simply calculated as the quotient of the Laplace transform of the output of the system $Y(s)$ and the Laplace transform of the input of the system $U(s)$ with all initial conditions assumed to be zero .

The transfer function of a general linear differential equation (Equation 2.1) is

$$F(s) = \frac{Y(s)}{U(s)} = \frac{1}{a_n s^n + a_{n-1}s^{n-1} + \cdots + a_1 s + a_0}. \tag{2.7}$$

Example 2.2 *The Laplace transform of the spring/mass/damper system of Example 2.1 is as follows:*

$$ms^2Y(s) + bsY(s) + kY(s) = U(s) \tag{2.8}$$

and thus the transfer function is:

$$F(s) = \frac{Y(s)}{U(s)} = \frac{1}{ms^2 + bs + k}. \tag{2.9}$$

2.1.3 Fourier Transform

The Fourier transform provides a method for deriving a representation of a system in the frequency domain from the set of differential equations representing it in the time domain.

$$\mathcal{F} : \mathbf{R}^{\mathbf{R}} \longrightarrow \mathbf{R}^{\mathbf{C}}.$$

The Fourier transformation of a function f of time is defined in terms of a transformation integral[3]

$$\mathcal{F}[f] = F = \lambda s. \int_{-\infty}^{\infty} f(t)e^{-j\omega t}\, dt \tag{2.10}$$

and exists for f if

$$\int_{-\infty}^{\infty} |f(t)|\, dt < \infty. \tag{2.11}$$

The Fourier transformation is clearly closely related to the Laplace transformation (Equations 2.3 and 2.4). The transformation integrals differ only in the complex variable.[4] Given this fact, the Fourier transform can be calculated directly from the Laplace transform simply by substituting $s = j\omega$. For instance, given a general linear differential equation (see Equation 2.1) with all initial conditions set to zero (i.e, $f(0) = 0,\ f'(0) = 0, \ldots, f^{n-1}(0) = 0$), the Fourier transform is

$$a_n(j\omega)^n Y(j\omega) + a_{n-1}(j\omega)^{n-1} Y(j\omega) + \cdots + a_1 j\omega Y(j\omega) + a_0 Y(j\omega) = U(j\omega). \tag{2.12}$$

The function gained from substituting $s = j\omega$ into the transfer function of a system is also referred to as a *transfer function*. The transfer function of a general linear differential equation (Equation 2.1) is

$$G(\omega) = \frac{Y(\omega)}{U(\omega)} = \frac{1}{a_n(j\omega)^n + a_{n-1}(j\omega)^{n-1} + \cdots + a_1 j\omega + a_0}. \tag{2.13}$$

[3] In control engineering j, rather than i, is used to represent the complex constant.

[4] The lower limit of the Fourier transformation integral may differ from the lower limit of the Laplace transformation integral. However, by defining $f(t)$ only on $t \geq 0$, as is often desired, the lower limit of the Fourier transformation integral can be considered to be 0. This is equivalent to the lower limit of the unilateral Laplace transformation integral.

Example 2.3 *The transfer function for the spring/mass/damper system of Example 2.1 is as follows:*

$$G(\omega) = \frac{Y(\omega)}{U(\omega)} = \frac{1}{-m\omega^2 + bj\omega + k}. \tag{2.14}$$

2.1.4 System Configuration

Systems can often be modelled hierarchically; that is, a system can be built from a number of subsystems, which in turn can be built from subsystems. This hierarchical modelling allows distinctions to be made between a controller and the dynamical system it controls (often referred to as the *plant*), and to be made between components within the controller and the plant. This hierarchical structure can be modelled graphically using *block diagrams* [43, pp. 62–66]. Block diagrams are graphical representations of the mathematical models of systems and are composed of a number of *blocks* representing a system component and directed edges between them. Each block can itself contain a block diagram allowing complex systems to be considered at different levels of abstraction.

There are several common system configurations, including *sequencing*, *closed–loop*, *closed–loop* with a component in the *feedback path* and *summation* (see Figure 2.2), by which most systems can be represented. *Closed–loop feedback* systems [43, p. 174] (also referred to as *closed–loop systems*) allow some measurement of the actual output of a system to be compared to the desired output and use the difference as a means of control. Often a component in the feedback path amplifies the difference, which is then used in the control process to ensure the difference is continually reduced. *Open–loop* systems operate without feedback, directly generating output in response to an input signal.

For each of the different system configurations there is a corresponding formula for determining the transfer function of the system as a whole from the transfer functions of the sub–systems [43, p. 64]. The simplest of these system configurations contains two systems represented by the transfer functions $H(s)$ and $G(s)$ in sequence. The transfer function for such a system configuration can be obtained by taking the product of the transfer functions

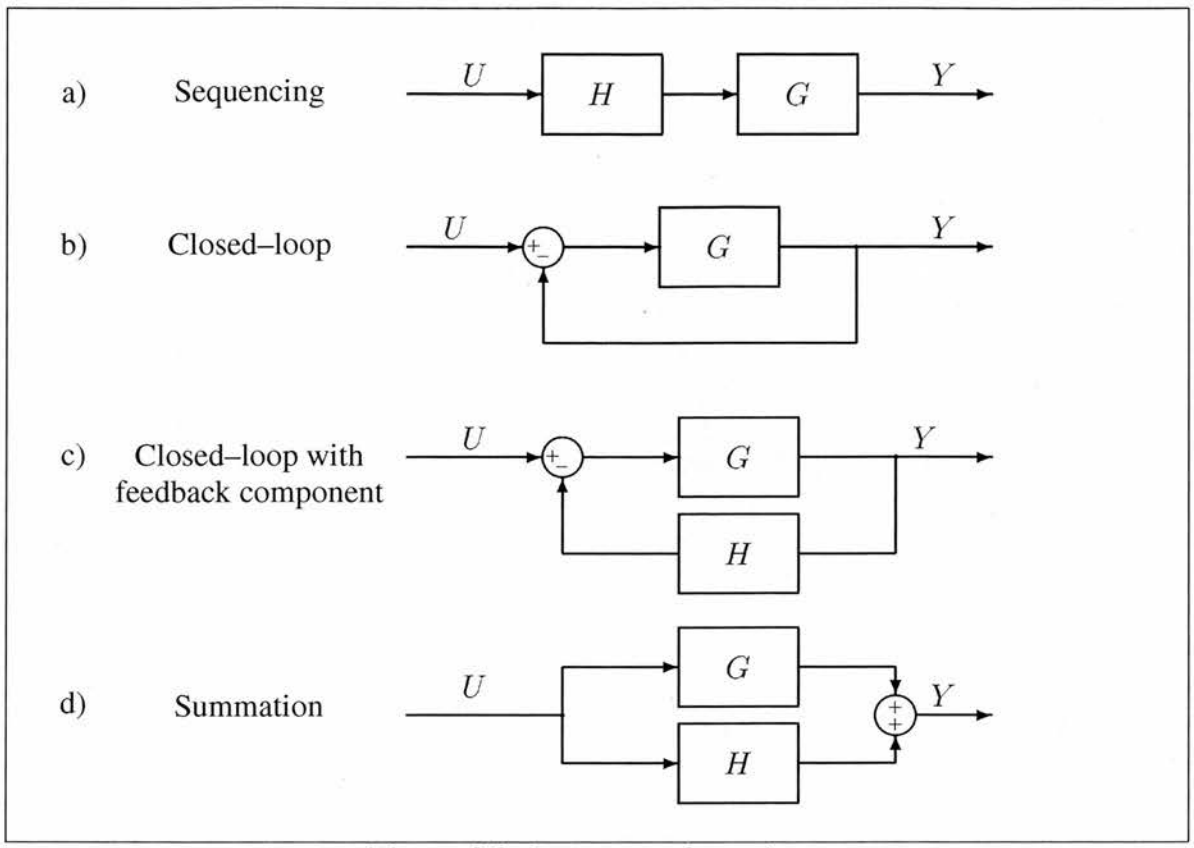


Figure 2.2: System configurations.

for the subsystems:

$$HG(s) = H(s)G(s). \tag{2.15}$$

The transfer function for a system $G(s)$ contained within a loop (the closed–loop transfer function) is obtained by evaluating and simplifying the following expression:

$$\frac{G(s)}{1 + G(s)}. \tag{2.16}$$

In this configuration $G(s)$ is referred to as the *loop transfer function* [43, p. 65].

A closed–loop system with a feedback component $H(s)$ has a transfer function obtained by evaluating and simplifying the following:

$$\frac{G(s)}{1 + G(s)H(s)}. \tag{2.17}$$

In this configuration $G(s)H(s)$ is referred to as the *loop transfer function* [43, p. 65].

The transfer function for a system consisting of the summing of the output of two systems $G(s)$ and $H(s)$ is obtained by evaluating and simplifying the following:

$$G(s) + H(s). \tag{2.18}$$

2.2 Classical System Analysis

Systems are analysed with respect to their specific design requirements. Analysis can be performed in the time domain, i.e, the response of the system to inputs over time, in the frequency domain, i.e, response of the system to inputs over frequency, or in the complex plane[5]. This section focuses on graphical methods for system analysis in the frequency domain and complex plane. A brief introduction to system analysis in the time–domain is given in Appendix A.

Many classical methods for system analysis focus on the property of stability. A system can be considered *stable* if for any bounded input the system has a bounded output (*bounded–input, bounded–output stability*). This definition of stability merely states whether a system is stable or not (sometimes referred to as *absolute stability*) but no inference is made about the degree to which the system is stable (*relative stability*). In general, stability is inversely proportional to performance and a balance must be found between the two. Modern fighter aircraft have a lower degree of relative stability than passenger aircraft as they require greater manoeuvrability. Absolute stability is a requirement for virtually all systems.

In the complex plane, the stability of a linear system can be determined by analysing the poles (the roots of the denominator) of the transfer function representing it. A system is stable if all the poles are in the left half of the complex plane (the poles have negative real parts), and it has a good transient response when the poles and zeros are away from the y–axis [43, p. 294].

[5]The complex plane is also referred to as the *s–plane* in control theory literature.

2.2.1 Graphical Analysis

There are three main graphical analysis techniques used in the analysis of systems in the frequency or complex plane: the Nyquist plot (complex plane), Bode diagrams (frequency domain) and Nichols plots (frequency domain).

Nyquist stability analysis [43, pp. 470–487] allows stability of a closed–loop system $F(s) = \frac{G(s)}{1+L(s)}$ to be determined from the analysis of its loop transfer function $L(s)$. The analysis relies upon Cauchy's theorem or *principle of the argument*. The Nyquist contour that encloses the right–hand s–plane is mapped into the $L(s)$–plane to give a *Nyquist plot* (Figure 2.3).

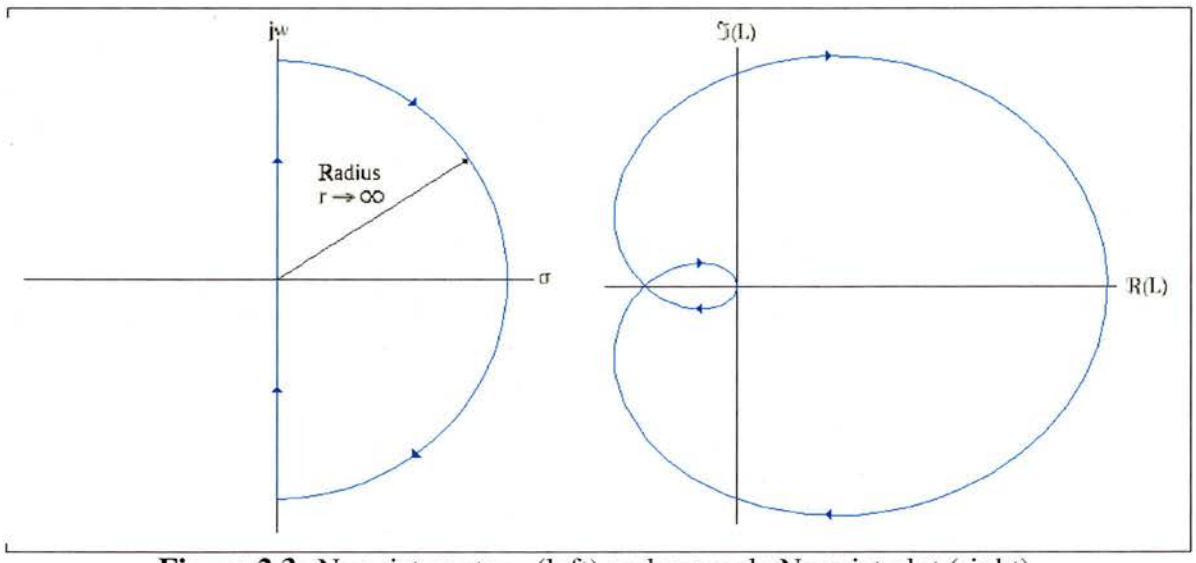


Figure 2.3: Nyquist contour (left) and example Nyquist plot (right).

The *Nyquist stability criterion* states that for a closed–loop system $F(s) = \frac{G(s)}{1+L(s)}$ to be stable, the Nyquist plot must make 1 counter–clockwise encirclement of the point $(-1, 0)$ for each pole of $L(s)$ lying in the right–hand plane and the plot must not encircle the point $(-1, 0)$ if no poles of $L(s)$ lie in the right–hand plane. The proximity of the plot to the stability point $(-1, 0)$ is a measure of the relative stability of the system. If the plot passes through the point $(-1, 0)$ then the system is marginally stable. This method of analysis requires that the number of poles of $L(s)$ with positive real parts be known and thus requires calculation using complex mathematics.

Bode diagrams and *Nichols plots* provide graphical representations of the response of a system in the frequency domain. A system is exposed to a sinusoidal signal as input and produces a sinusoidal output. The input sinusoid has constant amplitude but varying frequency. The system is analysed by comparing the output sinusoid to the input sinusoid.

The output sinusoid may have a different magnitude than the input and may also have undergone a displacement of waveform. The proportion by which the magnitude of the input sinusoid had increased is called the *gain*, which is often expressed in decibels. The displacement of waveform of the sinusoid is known as *phase–shift* and is measured in degrees.

The *Bode diagram* [50, p. 34] (e.g. see Figure 2.4) consists of two graphs: one plotting the phase–shift of the output sinusoid against logarithm of frequency and the other plotting the gain in decibels of the output sinusoid against the logarithm of frequency.

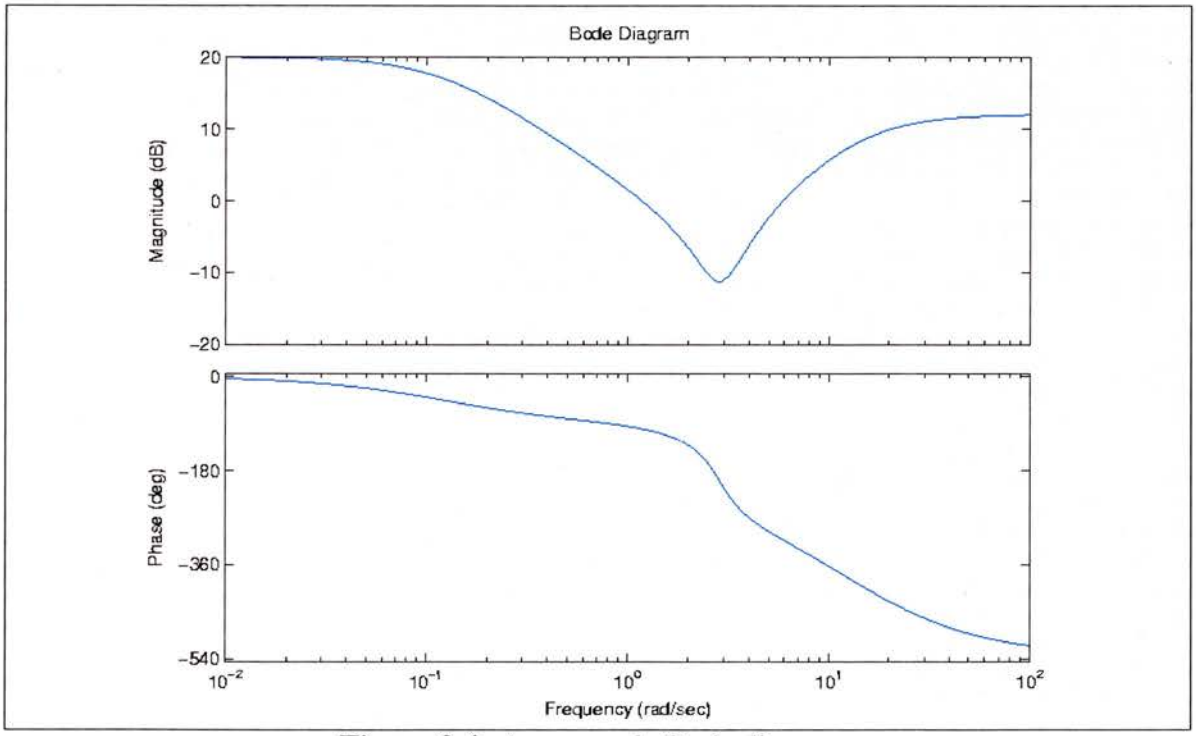


Figure 2.4: An example Bode diagram.

The *Nichols plot* [43, p.435] (e.g. see Figure 2.5) (also known as a *Nichols chart*) plots the gain (in decibels) against the phase-shift of the output sinusoid as the frequency varies. Nichols plots often show *exclusion regions* that are used in the analysis of systems.

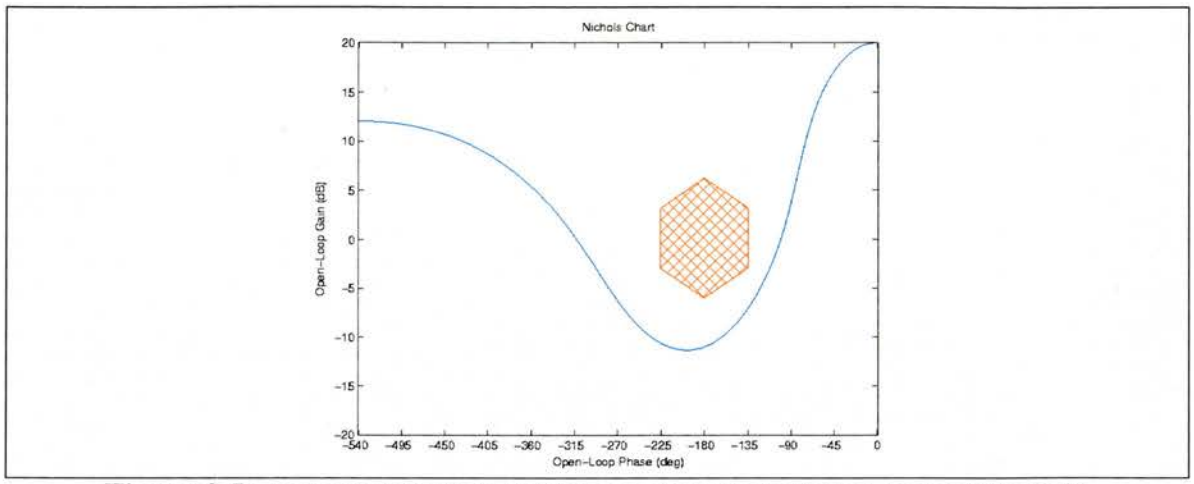


Figure 2.5: An example Nichols plot showing an *exclusion region* (shaded).

Nichols plots and Bode diagrams are very closely related: a Nichols plot can be constructed from a Bode diagram by reading the values of the gain and phase–shift for particular frequencies and plotting the gain against the corresponding phase–shift. Requirements that can be expressed in terms of Bode diagrams can be expressed in terms of Nichols plots, allowing equivalent analysis to be performed.

Nichols Plots

Nichols plots can be constructed by calculating the gain and phase–shift of a system F explicitly using Equations 2.19 and 2.20 and plotting the gain against the corresponding phase–shift

$$\text{gain} = 20\log_{10}(|F(j\omega)|) \tag{2.19}$$

$$\text{phase–shift} = \text{argument}(F(j\omega)) = \begin{cases} \arctan\left(\frac{\Im(F(j\omega))}{\Re(F(j\omega))}\right) + 180k & [\Re(F(j\omega)) \neq 0] \\ 90 + 180k & [\Re(F(j\omega)) = 0] \end{cases} \tag{2.20}$$

where $\Re$ denotes the real part of a complex number and k is some integer. When using arctan to calculate the value of phase–shift one must take into account the fact that the range of arctan is restricted to $(-90,\ 90)$ (or $(-\frac{\pi}{2},\ \frac{\pi}{2})$ in radians). If the shift in phase at ω is greater than 90° then $\arctan\left(\frac{\Im(F(j\omega))}{\Re(F(j\omega))}\right)$ must be adjusted by an appropriate multiple k of 90 to give the phase–shift.

The point $(0dB, -180°)$ on the Nichols plot corresponds to the stability point $(-1, 0)$ on the Nyquist plot [43, p. 487]. Just as the proximity of a Nyquist plot to the stability point is a measure of relative stability (see Section 2.2.1) so is the proximity of the Nichols plot. The proximity of the Nichols plot to the stability point is often measured in terms of the *gain margin* and the *phase margin*. The gain margin is the reciprocal of the gain at the frequency where phase–shift is $-180°$ and the phase margin is the difference between the phase–shift at the frequency where the gain is $0dB$ and $-180°$.

More complex analysis of the response of a system can also be determined by examining Nichols plots. The plot can be used to determine relative stability, but can also be used to determine other aspects of the response of the system. Nichols plot requirements are specified in terms of bounded regions on the plane that the plot must either be inside or be outside. The specific regions depend upon purpose and can be as restrictive as required. For example, the High Incidence Research Model (HIRM) [48, p. 49] specifies a region within which the handling qualities of an aeroplane are good, i.e.it has a good response (Figure 2.6).

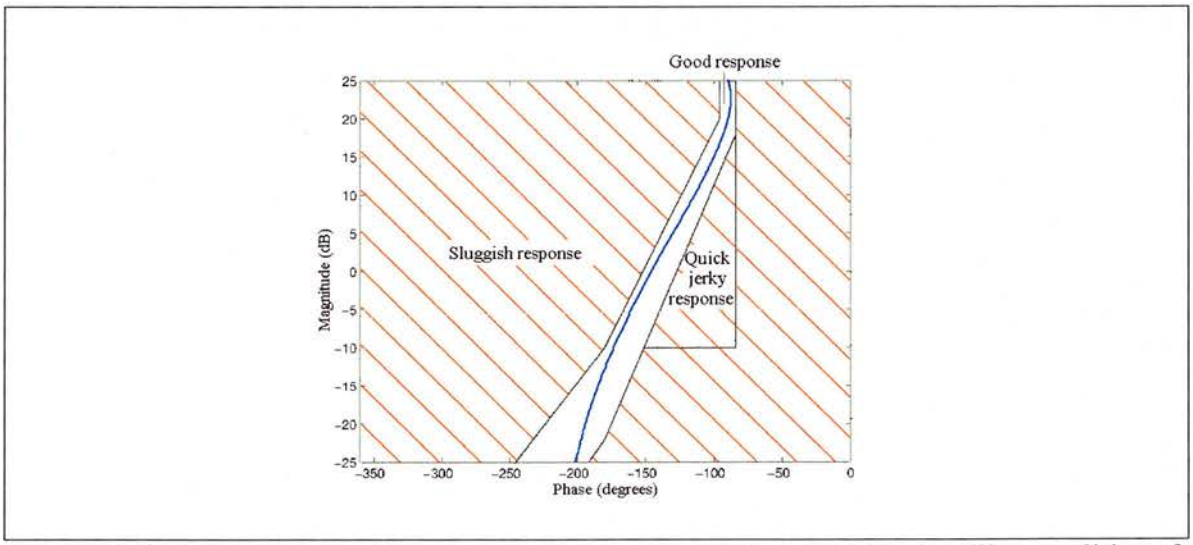


Figure 2.6: A Nichols plot showing a region (unshaded) in which the handling qualities of an aeroplane are good.

Common regions to be avoided are a hexagon about the point $(-180, 0)$, passing through the points $(-180, -6)$, $(-145, -3)$, $(-145, 3)$, $(-180, 6)$, $(-215, 3)$, $(-215, -3)$ and an extended hexagonal region [108, p. 243] (see Figure 2.7). In general, a good balance

between stability and performance is achieved when the plot is close to, but does not enter, these regions.

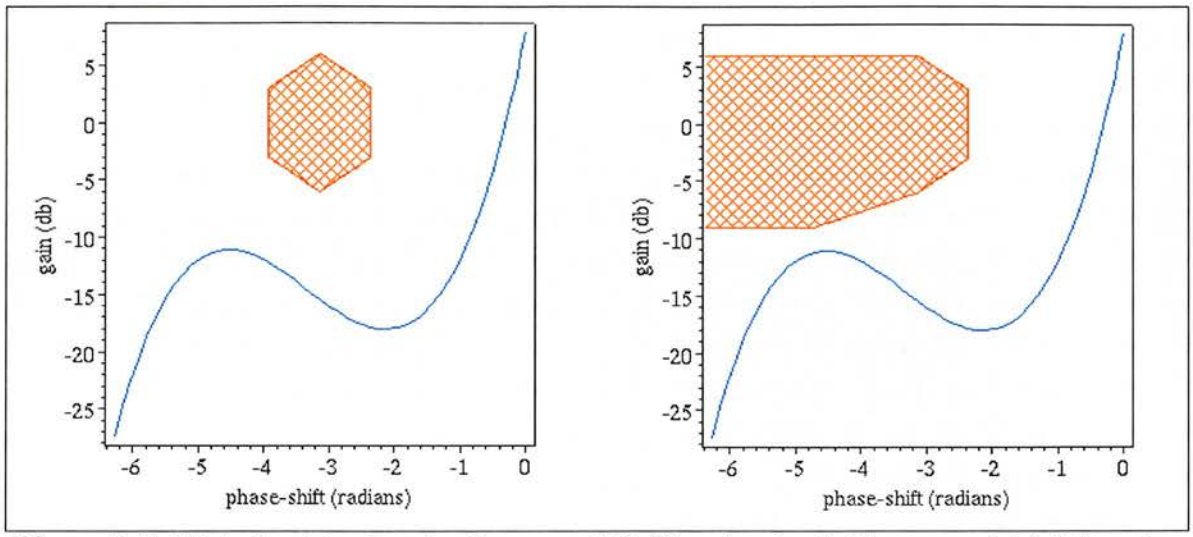


Figure 2.7: Nichols plots showing hexagonal (left) and extended hexagonal (right) exclusion regions.

2.3 Development Tools

There are many packages available for computer–aided control system design (CACSD). However, only a few CACSD systems are commonly used in industry, academia and education. The most popular are the freeware product Scilab [109] and the two commercial products the Mathworks' MATLAB [68] (MATrix LABoratory) and the National Instrument's suite of tools Matrix$_x$ [95]. These tools are so pervasive in control system development that they are used to introduce basic concepts of control system design in several introductory textbooks [43, 101, 114].

These systems are all at their core numerical computing systems providing libraries of generic mathematical functions and algorithms. Each of these systems provides efficient numerical computation, clear plotting graphics and allows the extension of the functionality of the systems by the inclusion of modules (MATLAB and Scilab refer to the modules as *toolboxes*, Matrix$_x$ refers to them as *Add–Ons*). Specialised libraries of functions and

algorithms for control system development and analysis are provided by means of modules for each of the systems. Dynamic systems can be represented in each of these systems in mathematical form and their response to various inputs can be simulated and analysed using classical techniques for control system analysis, including Nichols plots.

The systems are *matrix–based*, that is, data is represented in matrix form, and have an inherent ability to handle matrices (basic matrix manipulation, concatenation, transpose, inverse etc.,). This allows complex calculations on large data sets to be performed efficiently; however, they have no inherent ability to perform symbolic computation. This means that the response of systems to a continuous range of inputs can not be analysed directly, rather the system is analysed at sample values and then assumptions are made about the system's response over the range based on the results at the sample points. Systems containing parameters with a range of values must also be analysed at sample values rather than symbolically for all values of the parameter. MATLAB and Scilab provide links to the computer algebra (CA) system *Distributed Maple* to allow a number of symbolic operations to be performed from these systems using their language and syntax; however, these symbolic operations are not integrated into any control system analysis performed by these systems.

Although the input to each of the core computing systems is textual each can be linked with a graphical development environment (GDE), in which systems can be developed as hierarchical block diagrams. For example, the Simulink GDE [38] is closely coupled with MATLAB, the Matrix$_x$ suite includes the SystemBuild GDE [96] and Scilab can be extended with Scicos Toolbox [99] to provide a GDE. Systems modelled in the GDEs can be simulated and analysed from the core numerical computing systems. Tools exist to automatically generate code from systems modelled in the GDEs but no formal guarantees are provided that the code accurately implements the behaviour of the model.

Chapter 3

Formal and Symbolic Methods for Control System Development

Research into integrating formal or symbolic methods into the development of control systems generally falls into one of two categories: firstly, *design verification*, in which a design specification, usually in the form of a mathematical representation of the dynamic behaviour of a system, is analysed with respect to some design criteria and secondly, *implementation verification*, in which a software or hardware implementation is verified against a system specification.

Formal and symbolic methods for design verification can be further divided into those methods that are based on existing development techniques and those that introduce new methods for development, often based on techniques commonly used in computer science. New methods for development include new methods for modelling systems and their requirements as well as new methods for analysis. These methods are often unfamiliar to control engineers, which may hinder their widespread inclusion in control system development. Methods based on existing analysis techniques do not require any fundamental change in modelling or analysis but instead provide more assurance that the results of classical analysis are correct.

In this chapter an overview of formal methods for control system development is presented. In Sections 3.1, 3.2 and 3.3 a range of methods for formal and symbolic design verification are described. Section 3.4 details methods for use in the implementation verification phase.

Various methods for using hybrid systems to model and analyse control systems are described in Section 3.1. In Section 3.2 qualitative and semi–quantitative reasoning are introduced as methods for modelling and simulating the behaviour of dynamic and control systems. In Section 3.3 symbolic methods based on classical control system analysis are described. Methods for code generation and verification are described in Section 3.4.

3.1 Hybrid Systems and Model Checking

Control systems may consist of both continuous plants and digital controllers; these are examples of *hybrid control systems*. Hybrid control systems are networks of interacting continuous and discrete components. The structure of these systems is similar to that of (in)finite–state (concurrent) systems that arise in several areas of computer science.

There has been much research within the field of computer science into the development of formal languages and techniques for the specification and analysis of (in)finite–state (concurrent) systems. The use of these techniques has been well established in the development and analysis of software. The ISO standard formal description technique, *Full LOTOS* [21], whose semantics are basically those of state transition systems, is widely used.

The most common method for the formal analysis of (in)finite state systems is *Model checking* [33]. The aim of model checking is to perform an exhaustive search of all reachable states and associated behaviours using explicit or implicit enumeration. Explicit enumeration will often lead to an infinite state system, which is not amenable to an exhaustive search. Implicit enumeration will often lead to a significantly simpler state system. For this reason symbolic model checking, in which implicit enumeration is performed, is desirable. To allow symbolic model checking of *Full LOTOS* specifications, a theory of *symbolic*

transition system and a *modal logic* have been developed [89].

When model checking a system, if a property fails to hold for a given trace then this can be used as a counter example. If an exhaustive search is performed and no counter example is found then the property holds for the system. However, if the system is large it is unlikely that an exhaustive search can be performed, even using implicit enumeration, and in these cases failure to find a counter example does not guarantee the requirement holds. Model checking can be fully automated and is useful in pinpointing errors. It is suitable for determining whether a system fails to meet a requirement, though generally, can not provide guarantees that a requirement has been met.

Several approaches to modelling hybrid control systems for the application of analysis techniques, such as model checking, have been suggested [6, 88, 90, 98, 97]. These modelling techniques are generally very similar. The most commonly used are *hybrid automata* [6] and *hybrid I/O automata* [88]. A hybrid (I/O) automaton can be viewed as a transition system with (in)finitely many states and consists of a directed multigraph (V, E) with a set of *vertices* V a set of *edges* $E \subseteq V \times V$, a set of variables X, and two operators *prime* $'$ and *dot* $\dot{}$. The set X' represents the continuous change in the variables and $\dot{X}$ represents the change in variables after a discrete step. Hybrid I/O automata make a distinction between internal and external actions and variables.

There are several approaches to the specification of requirements for hybrid control systems. In [87] both the system and system requirements are modelled as hybrid I/O automata (HIOA). An *abstract protector* is used to ensure that a system meets its safety requirements. An abstract protector is parameterised with a physical plant PP, which is represented as a HIOA, subsets R, G and S of the states of PP, a *port index* j representing a communication channel between PP and the protector, and a sampling period d. R represents the set of states to which PP is restricted when considering the protector. G represents the set of states to which the protector should restrict PP. S represents the set of states for which the protector is said to guarantee G from R. A state $s' \in R$ is R–reachable from $s \in R$ if there is a trace from s to s' whose states are all in R. The sets S and G are subsets of R

and S is 'safe' for PP, R, G and j if there exists an input action on port j whose immediate execution guarantees that all subsequent R–reachable states with no input actions on port j are in G. Consider an abstract protector designed to ensure that a vehicle PP in an automated transit system does not go above a designated maximum speed. The protector and PP communicate on port j. R may be the set of states for which the vehicle has not collided. G would be the set of states for which the vehicle is below the designated speed limit. S would be the set of 'safe' states for PP, R, G and j. input action on port j The proof of correctness of an abstract protector leads to simple correctness proofs of a protector implementation for particular instantiations of the abstract model. The composition of independent and dependent protectors under certain conditions guarantee the conjunction of the safety properties guaranteed by the individual protectors. In [6] systems are represented as hybrid automata and their requirements are represented as formulae consisting of boolean combinations of inequalities over the variables of the system. To show that a system A meets the requirement represented by a formula ϕ it must be shown that ϕ is an invariant of A, that is, given a set R_f of 'bad' states as determined by ϕ it must be shown that R_f is unreachable from any reachable state.

Despite the fact that the problem of determining the reachability of a state is unsolvable [62] even for a highly restricted class of hybrid automata, it is shown [5, 6] that several techniques, such as fixed point computation and minimisation procedures for timed–automata, can be adapted to perform analysis of these systems, and many algorithms have been suggested to improve the efficiency of model checking to allow the analysis of large (finite) systems [32, 81, 129]. These analysis techniques still suffer from the problem of state space explosion, which occurs when the number of states becomes intractable and it is impractical to perform an exhaustive search. This occurs in large systems with many components or in systems in which variables can assume many different values. This is common in control systems and means that although these techniques are applicable to simple systems they are not useful in the analysis of large and complex systems that are found in practice.

Work in the field of hybrid system abstraction attempts to abstract away from large or possibly infinite state systems without loss of relevant behaviour, to reduce complexity and

allow the model checking of these systems. The aim is to produce a simplified system that is more accessible to analysis tools but is still sufficient to establish desirable properties. Various classes of hybrid systems that allow property preserving abstractions to discrete systems are presented in [8] and for those systems for which discrete abstractions are not possible the use of abstraction to 'sufficient' systems rather than equivalent systems is suggested. By over or under approximating the reachable sets of differential equations, systems belonging to undecidable classes can be analysed. Methods for the abstraction and verification of restricted classes of hybrid systems that are presented in [7, 117] also focus on constructing discrete finite state abstractions for hybrid systems, which can then be more easily model–checked for safety properties. These methods are only applicable to a restricted class of hybrid systems. In [128] a method for abstracting analogue infinite–state systems is presented.

Most control systems are not initially developed as hybrid (I/O) automata or state systems and must therefore be translated into this representation before model checking can be performed. This translation, if performed manually, can introduce errors. The Vapor (Verilog Abstraction for Processor verification) [10] tool was developed to eliminate these errors by automatically translating models into finite state system representations. This tool applies only to discrete models described using behavioural RTL (Register Transfer Level) Verilog. The system is translated into the CLU (Counter arithmetic with Lambda expressions and Uninterpreted functions) language used by the UCLID (Uninterpreted functions, Counter arithmetic and Lambda expressions for Infinite Domains) [28] tool for verification of (in)finite state systems.

3.2 Qualitative and Semi–Quantitative Reasoning

Many classical analysis techniques can only be applied to models of systems with exact numerical values for all parameters. For complex systems, exact quantitative information is often unknown; for instance, it may only be known that the relationship between two

quantities is monotonic or that a parameter lies within some range.

Qualitative reasoning [79] is a method for reasoning about systems where knowledge of them is incomplete. A *Qualitative description* of a system describes only the aspects of the system that make important qualitative differences and ignores those that do not. These descriptions do not rely on quantitative knowledge; they describe the system in terms of a continuous set of values split into qualitatively distinct regions at significant boundary points. These boundary points can either be *landmark values*, which have a precise numerical value that may be known or unknown, or be *fuzzy values*, which have no meaningful precise numerical value. For instance, an appropriate qualitative description of a water tank could consist of the two landmark values 'full' and 'empty' (representing the precise values of the maximum and minimum capacity of the tank), the three fuzzy values 'low', 'normal' and 'high', which have no meaningful precise values, and the six intervals ('empty','low'), ('low', 'normal'), ('normal', 'high'), ('high', 'full') and ('full', infinity). Other information about the tank, such as its dimensions and the material from which it is made, are ignored.

Any aspect of a system's behaviour that is modelled using ordinary differential equations (ODEs) can be represented in qualitative terms as *qualitative differential equations* (QDEs) [79] or using *confluences* [41]. QDEs allow the relationship between two variables to be specified as being in the class of monotonically increasing (or decreasing) functions over the extended reals without specifying the particular function. The set of values that the variables can take is described qualitatively in terms of landmark values from the extended reals and intervals between them. A QDE may be an abstraction of many ODEs. The relationship between the theory of differential equations and the qualitative representation is made explicit and precise in the QSIM (Qualitative SIMulator) framework for qualitative simulation [79]. Confluences are equations or qualitative expressions that evaluate to signs. Confluences can be constructed from equations by considering only the signs of their inputs and conclusions can be drawn about their qualitative properties.

Simulating the behaviour of a system represented by the QDEs Q given some initial state s involves the construction of a finite structure D of qualitative state descriptions that repre-

sents the possible behaviours consistent with Q and s. Each behaviour consists of a series of time points at which one or more variables change qualitative value and the intervals between them over which the qualitative behaviour is unchanging. The structure D can be used to infer which qualitative behaviours might occur in a given instance and can be used along with the qualitative description of the system to infer further constraints on the landmark values and other terms. The analysis of the structure D allows conclusions to be drawn about the behaviour of any 'reasonable' extended real–valued ODE abstracting to the QDE Q. However, qualitative simulations cannot infer which qualitative behaviour will be the one to actually occur in any given instance. This limits the usefulness of qualitative simulation.

Semi–Quantitative reasoning [79, pp. 203–236] combines qualitative and incomplete quantitative knowledge. *Semi–quantitative simulation* can be viewed as a constraint satisfaction problem that is used to refute potential qualitative behaviours of a system. Given the qualitative description of a system, a structure representing possible behaviours and the implied constraints on the landmark values and other terms, limited quantitative knowledge of the landmark values can be used to attempt to find inconsistencies between the allowable values and the implied constraints. Inconsistent behaviours can be excluded and consistent behaviours can be annotated with constraints on values. The quantitative knowledge provided can be in the form of bounding intervals, probability distribution functions and fuzzy sets.

Quantitative information inferred by semi–quantitative simulation can be very coarse due to weak quantitative inferences and can be viewed as a variant of the issue of step size in numerical simulation of ODEs, i.e. in semi–quantitative simulation the weaker the quantitative knowledge, the coarser the information inferred by semi–quantitative reasoning and the weaker the knowledge about the system, and in numerical simulation the larger the step size the less accurate the simulation. The inferences made in semi–quantitative simulation can been be strengthened somewhat using a variety of techniques, such as *step–size refinement* [19] and *dynamic envelopes* [74], though each has weaknesses. Step–size refinement is an iterative process in which new explicit states are interpolated between distinct qualita-

tive states. Constraints implied by these new states are propagated and more new states are interpolated if possible. This method is not guaranteed to terminate if constraints contain an infinite number of elements, for instance, real intervals. Dynamic envelopes are bounds on the derivatives of state variables built up from bounds on the state variables along with fundamental properties of addition, subtraction, multiplication, division and properties of monotonic functions. For instance, let $\underline{x}$ and $\overline{x}$ represent lower and upper bounds on any real number x, given a QDE $y' = c - f(y)$, where f is monotonic increasing and the constant c and state variable y are positive, the dynamic envelope for the QDE is $\underline{y'} = \underline{c} - f(\overline{y})$ and $\overline{y'} = \overline{c} - f(\underline{y})$. Integrating dynamic envelopes into semi–quantitative simulation potentially reduces step size and the range of variables.

Qualitative and semi–quantitative simulation can be used to predict the behaviour of a system making it ideally suited to the monitoring and diagnosis of existing systems. The existing system is observed and a qualitative model is built. This model is simulated and from this predictions can be made about the behaviour of the physical system [45, 115]. Qualitative and semi–quantitative reasoning can also be used in the development of control systems. In each region in which a system has a distinct qualitative behaviour, qualitative modelling and simulation can be used to synthesise a qualitative controller [80]. This qualitative controller can be used as a basis for the design of a control system.

Although there have been attempts to model large–scale, realistic mechanisms [73], qualitative and semi–quantitative techniques can lack the expressiveness, soundness, efficiency and tractability to apply to analysis of complex real world systems. There is also no unified framework that binds all the steps involved in qualitative and semi–quantitative reasoning; the representation of ODEs as QDEs, qualitative simulation and semi–quantitative simulation.

3.3 Formal and Symbolic Methods for Analysis

Classical control system analysis often involves either the plotting and visual analysis of graphs or the solving of real or complex equations or inequalities. The techniques classically used for solving these (in)equalities are numerical and often require the exact values of all parameters to be known. In many cases, symbolic methods can be used in place of the numerical techniques.

Gröbner bases [77, pp. 85 – 144] are a tool from constructive commutative algebra, which provide a means for solving systems of multivariate polynomial equations. Gröbner bases have been applied in many areas of classical control system analysis and development. In [49] Gröbner bases are applied to the analysis of linear systems and nonlinear systems that have polynomial nonlinearities. Gröbner bases are used in the analysis of linear systems in terms of their gain and phase margins. The analysis of nonlinear systems focuses on problems in *Lyapunov theory* [105] in particular those involving *local Lyapunov functions*, and on stability analysis using the *harmonic balancing method* [14]. These symbolic techniques are implemented in the CA system *Maple* 4.1.1 and provide improvements over the classically used numerical techniques; however, unless implemented in a formal setting little assurance can be given about the correctness of the results of the analysis.

In [70] Gröbner bases are also used in the analysis of linear systems in terms of their gain and phase margins; however, this analysis is augmented with quantifier elimination (see Section 4.3). These techniques are also applied to the analysis of linear system using the *Routh–Hurwitz conditions* [43, p. 295] for stability, the calculation of *equilibrium points* [14] of nonlinear systems, and the choice of parameters in controllers for systems with uncertain parameters. Quantifier elimination has been used in a similar manner in the stability analysis of difference schemes [86, 66], the stability analysis and design of linear control systems [1] and the design of nonlinear control systems [71].

A logic in the style of Hoare [64] for single–input single–output continuous–time control systems, which is used to infer properties of the gain and phase–shift of a system from

properties of its subsystems, is presented in [24]. Rules are formalised for computing the gain and phase–shift of large systems from their structure and the gains and phase–shifts of their subsystems. Based on these rules, a simple language *Cosy* for describing control systems in terms of their gain and phase–shift is defined. The language consists of two atomic and three compound constructs: **Unit** representing a system that makes no change in gain or phase–shift whose gain is 1 and phase–shift is 0; **Fcn**$< d, \theta >$ representing a system, with no subsystems, whose gain is d and phase–shift is θ; **Seq**$< G_1, G_2 >$ representing a system consisting of two subsystems G_1 and G_2 in sequence; **Loop**$< G_1, G_2 >$ representing a system consisting of a closed loop with G_1 in the feed forward path and G_2 in the feedback path; and **Sum**$< G_1, G_2 >$ representing a system consisting of the summation of two subsystems G_1 and G_2. The Hoare logic for the *Cosy* language operates on a triple consisting of a pre–condition P, a *Cosy* representation of a control system $C < d, \theta >$, and a post–condition Q. Both P and Q represent some condition on gain and phase–shift. The triple is interpreted to mean that, given that P holds, the system $C < d, \theta >$ will produce a gain and phase–shift for which Q holds. The Hoare rules and tactics provide a means either of determining constraints on the gain and phase–shift of a system from constraints on the gain and phase–shift of its subsystems or of determining conditions on parameters that ensure that the gain and phase–shift of the system meets their constraints. This method exploits the hierarchical nature of control systems and allows propositions about commonly used subsystems to be proved just once; however, this method requires symbolic reasoning about (nested) inverse trigonometric functions and may produce large and complex conditions which must be verified. The method is implemented [24] in the theorem prover HOL [52]; proof construction is semi–automatic, i.e. it may require user assistance.

Higher order logic is used in the analysis of DSP (digital signal processing) systems. The analysis focuses on formally verifying levels of error of fixed– and floating–point specifications against an idealised real number specification and then verifying an RTL (register transfer level) description against the fixed–point specification [4].

3.4 Code Generation and Verification

The ultimate aim of control system development is the production of a software or hardware implementation of a control system. In the first stage of the development process a control system design is produced by control engineers. This design usually takes the form of mathematical formulae representing the behaviour of the system. In the second stage of the development process, the design is treated either as a model from which a formal specification is developed or as a specification from which an implementation is developed. The commonly used tools for control system development (see Section 2.3) allow a control system design to be treated as a specification and provide means to automatically generate program code from it; however, no formal guarantees are provided that the code accurately implements the behaviour that the design models.

The "correct by construction" approach [15] aims to ensure that automatically generated code is correct by proving that the rules used to translate from a specification into an implementation are correct. If it can be shown that the translation process preserves the behavioural properties of the specification then it follows that the implementation generated by this process also displays the behaviour. A major benefit of this approach is that each rule for translation must be verified only once and no verification of an implementation generated using these rules is necessary. The approach has been successfully demonstrated in several case studies in which software for centralised controllers for moderate sized systems have been generated [120]; however, the source code produced is generally larger than that produced by manual implementation, which can be an issue for embedded systems, where space is seriously limited. If the source code is altered in any way all assurances of its correctness are lost and it must be verified using other methods. There is also a lack of support for common specification and programming languages.

Code that has been developed manually or has been automatically generated using potentially unsound methods must be verified against its specification to ensure that the implementation is correct. Formal verification of software against a specification is an inherently difficult task. This is due in part to the inclusion in most commonly used programming lan-

guages of certain features and constructs, such as recursion, pointers and goto statements, that are difficult to reason about. Owing to their complex nature, the use of these constructs is likely to introduce errors into an implementation. By omitting these problematic features and constructs the formal definitions of a language is simpler, making it easier to reason about. The Omnibus language [125] is an object–oriented language that is superficially similar to Java but does not contain certain aspects that complicate analysis, such as the default use of reference semantics for objects. The Omnibus language includes constructs for asserting properties about the behaviour and requirements of constructors, functions and operations. The Omnibus system includes interactive and automated verifiers along with a Java code generator. The system provides a range of techniques, such as, run-time assertion checking, extended static checking and full formal verification, to verify properties of an Omnibus implementation. SPARK Ada [17] is a subset of Ada that excludes features and constructs, such as exceptions, type aliasing and goto statements. SPARK Ada implementations must be annotated with non–Ada *formal comments* to allow their verification in the SPARK Examiner. This checks whether the implementation conforms to the rules of SPARK and checks consistency between the implementation and its annotations. The SPARK Examiner may generate conditions, which must be verified in some formal system to ensure the consistency of the implementation. The SPADE static analysis suite consists of the Automatic Simplifier, which allows the automatic verification of many of the trivial conditions, and the Proof Checker, in which the remaining conditions can be manually verified. SPARK Ada code can be verified against a formal specification in the Z language [112] using the Compliance Tool [83] component of the ProofPower [84] system. A 'compliance argument' is generated, which may consist of a large number of conditions that must be verified. Currently, the conditions must be verified manually; however, the conditions are largely trivial, for instance $\mathbf{[true \Rightarrow true]}$.

The ClawZ [12] tool uses a similar approach to the "correct by construction" method to produce a formal specification of a system in the Z language from a Simulink diagram (see Section 2.3). Auto–generated or manually developed implementation in any programming language can, in principle, be verified against the formal specification of the system. The

ClawZ and Compliance tools have been used in the verification of SPARK Ada code against a significant flight control system; this was shown to require significantly less effort and time than similar verifications in the past [11].

These methods and tools can dramatically reduce the effort, and therefore the time and cost, of producing verified software. However, these tools provide no means of assuring that the specification against which the implementation is verified is itself sound and there have been several cases in which a correct implementation of a flawed design has led to an accident [108, p. 26].

Chapter 4

Computer Mathematics and Automated Theorem Proving

The terms "computer mathematics" and "automated theorem proving" refer to the uses of computers to perform mathematical calculations and to produce formal proofs of theorems, respectively.

There is a wide range of computer mathematics systems and theorem provers all placing different emphasis on properties, such as efficiency, reliability, expressiveness and usability. In general, when performing mathematical calculations a balance must be found between the often mutually exclusive goals of efficiency and reliability. Computer mathematics systems tend to favour efficiency whereas theorem provers place the emphasis on reliability.

In this chapter the ideas of computer mathematics and theorem proving are introduced. The strengths and weaknesses of a number of computer algebra systems and theorem provers are discussed and these systems are evaluated in terms of their suitability for use in the formal and symbolic analysis of control systems. In Section 4.1 computer mathematics is introduced and the *Maple* [94], *Mathematica* [127] and *Axiom* [39] systems are evaluated. In Section 4.1.1 the *Maple* system is discussed in more detail. The concept of theorem proving is introduced in Section 4.2 and the theorem provers HOL [53] and PVS [103],

and the proof assistant Coq [26] are evaluated. In Section 4.2.1 the theorem prover PVS is described in more detail. The technique of quantifier elimination, which is an algorithmic technique for solving quantified formulae, is introduced in Section 4.3. Section 4.4 provides an overview of various systems designed to combine computer mathematics systems and theorem provers to take advantage of the strength of these systems and produce efficient and reliable mathematics systems.

4.1 Computer Mathematics Systems

Mathematical systems that are commonly used in the fields of science and engineering generally fall into one of two categories: numerical computing systems (see Section 2.3) and computer algebra (CA) systems. Numerical computing systems provide a means to perform efficient numerical calculations but do not generally have an inherent ability to perform symbolic calculations. CA systems provide a means to perform a wide variety of symbolic manipulation and numerical calculations; however, the methods for numerical calculations employed in these systems are generally not very efficient. The different strengths and weaknesses of these systems have led to numerical computing systems being favoured by the engineering community and CA systems being more popular among physicists.

CA systems, such as *Maple* [94], *Mathematica* [127] and *Axiom* [39], provide powerful tools for the solving and manipulation of formulae; but they do not guarantee the correctness of their results. In general it is deemed better for CA systems to produce an answer, in real–time, that is likely, but not guaranteed, to be correct than to produce no answer. The emphasis in these systems is generally on efficient computation rather than correctness or completeness of results.

Typically the mathematical formalisms and knowledge used by CA systems to perform calculations are stored in libraries, which contain *packages* or *modules* of procedures or functions. The libraries are generally separate from the kernel of the system, which performs

low–level operations, such as precision arithmetic, file input and output, and simple mathematics. This improves the efficiency of the systems by allowing the loading of libraries only when they are needed. The extent to which the implementation of the mathematical functions within the kernel and modules can be viewed and altered varies; for instance, *Mathematica* conceals large portions of its library and kernel, whereas *Maple* allows virtually all code in both its library and kernel to be viewed and allows code in most modules to be altered.

Strongly typed systems, such as *Axiom*, attempt to improve the reliability of their results by using type information to prevent functions being applied to inappropriate arguments. These systems are generally 'safer' than the more common weakly typed systems, that is they are less likely to produce an incorrect result. Weakly typed systems support datatypes but do not use them in a strict fashion. They generally do not type check the input to functions and procedures and may ignore type information and side conditions in calculations; however, they are often easier to use. Both *Mathematica* and *Maple* are weakly typed, performing very little type checking, but are considered easy to use and are consequently widely used in industry, academia and teaching.

A major factor contributing to the popularity of these systems is their programmability. They each provide their own programming languages. The *Maple* programming language is essentially procedural whereas the *Mathematica* language is essentially functional, implemented as a term–rewriting system with extensive pattern–matching capabilities.

The *Maple* and *Mathematica* systems are very similar and the choice of which to use often falls to personal preference. However, for application to the formal and symbolic analysis of control systems, *Maple* has a key benefit over *Mathematica*. Both *Maple* and *Mathematica* provide interfaces to external systems, which allow them to benefit from those systems' strengths; for instance they both provide links to MATLAB (see Section 2.3), which provides more efficient numerical calculation. However, *Maple* has been more widely integrated with theorem proving systems (see Section 4.4), which can be used to increase the reliability of results by providing formal proofs of their correctness.

4.1.1 Maple

Maple is a general purpose computer algebra system, which provides many powerful tools for manipulating and solving symbolic equations or inequalities. As with most computer algebra systems, the emphasis in *Maple* is placed upon the efficient calculation of a solution rather than on the correctness or completeness of the solution; however, recent versions of *Maple* implement more facilities that may be used to increase the reliability of its results.

Maple is provided with several different user interfaces. The most basic of these interfaces is the command line interface, which is most commonly used for batch processing or for solving large problems. The visualisation and plotting procedures provided by *Maple* can be accessed from the command line; however, the results of these procedures are generally very basic (see Figure 4.1). The *Maple* graphical user interface provides *worksheets* into

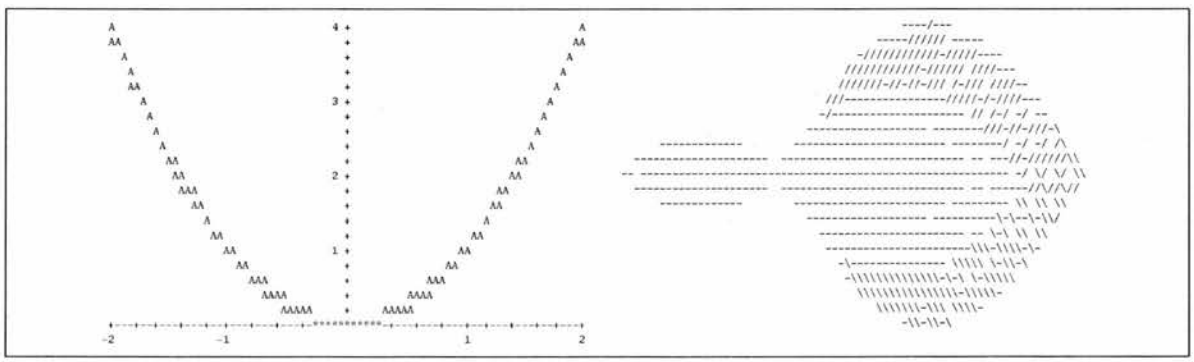


Figure 4.1: Illustration of 2–dimensional (left) and 3–dimensional (right) plotting quality of command line *Maple*.

which *Maple* commands are typed and results are displayed, and is generally used for interactive sessions and for visualisation and plotting (see Figure 4.2). *Maplets* are user defined graphical user interfaces. They are written using a Java–like programming language and produce applet–like interfaces, consisting of components such as windows, buttons, checkboxes, textboxes, etc. Interactions with components in a *Maplet* can be associated with actions in *Maple*; for example, the click of a button in a *Maplet* could be associated with the execution of a *Maple* procedure. This allows the end–user of the *Maplet* to use the computational power of *Maple* without knowledge of the underlying *Maple* syntax or language.

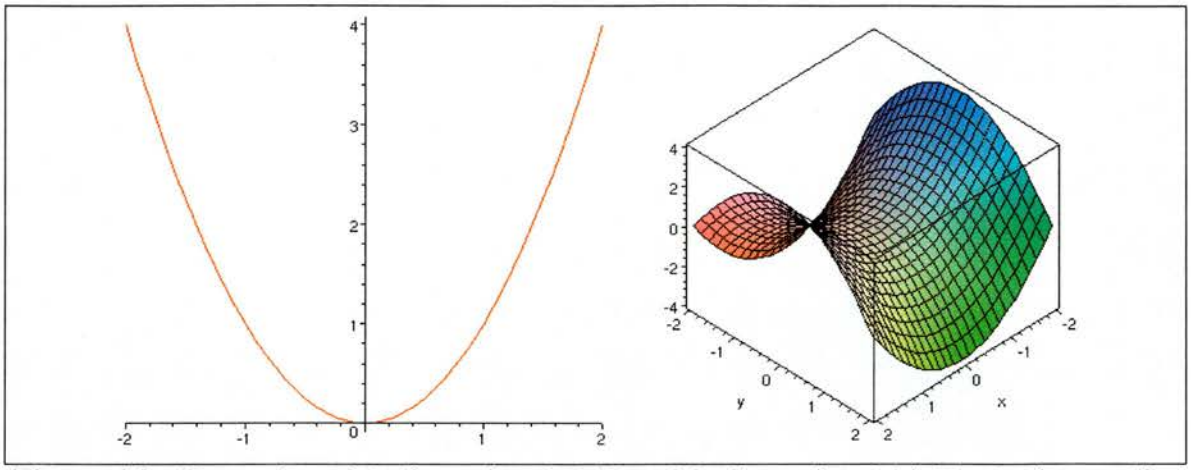


Figure 4.2: Illustration of 2–dimensional (left) and 3–dimensional (right) plotting quality of *Maple* GUI.

The *Maple* library can be extended with user defined functions, procedures and modules. These can be defined either within *Maple* in the *Maple* language or outwith the *Maple* environment in C, FORTRAN or Java.

The most fundamental procedures that *Maple* provides for solving equations and inequalities are the *solve* procedures. There are several variations of the *solve* procedure, each of which solves a specific problem; for example, *dsolve* solves differential equations and *fsolve* finds floating–point solutions to systems of equations. The *fsolve* procedure may fail to find all solutions to a problem and any results it does produce can often only be viewed as an approximation to a solution due to the inexact nature of floating–point arithmetic. The *solve* procedure attempts to provide a symbolic solution to inequalities or systems of equations. The *solve* procedure may return an implicit solution in terms of the *RootOf* function, which allows the representation of all the roots of an equation as a single variable. If *solve* returns no solutions this may mean either that no solutions exist or that the procedure has simply failed to find them. A more reliable method of finding the solutions to a univariate polynomial with integer coefficients is the *realroot* procedure, which determines isolating intervals for each real root. The procedure implements *Descartes' rule of signs* to determine the isolating intervals [37] and takes an optional argument, which allows the user to set the maximum width for the isolating intervals.

Maple provides a limited amount of automatic simplification of any input; however, more complex simplification rules can be applied using the *simplify* procedure. The procedure searches the input expression for function calls, square roots, radicals, and powers and invokes the appropriate simplification rules. The *simplify* procedure takes an optional second argument, which may be used to assert properties about the indeterminates in the expression; for instance, *assume=real* asserts that all indeterminates in the expression are real. These properties are used within the simplification procedure to further simplify the expression.

Maple allows assumptions to be made about the properties of any variable. The *assume* procedure sets properties of variables, associating given properties with given variables. Assumptions essentially provide information about the type or range of variables; for instance, *assume(x::real)* asserts that x is a real number, *assume(f,continuous)* asserts that f is continuous and *assume(z<y)* asserts both that z is less than y and that y is greater than z. Additional assumptions can be made about variables using the *additionally* procedure, which, unlike *assume*, preserves existing properties of the variable. Owing to the simple type checking provided by *Maple*, properties of a variable are not necessarily taken into account during various calculations; for instance, given the assumption *assume(x::posint)*, which asserts that x has the type *posint*, the *type* procedure, which determines whether an expression is of a given type, returns false for the arguments x and *posint*.

The *is* procedure determines whether an expression e satisfies the property p for all permissible values of e. This procedure uses the properties associated with variables within its calculation and will return true if all possible values of e satisfy p, false if any possible value of e does not satisfy p and FAIL if it cannot determine whether the property is true or false due to insufficient information or an inability to compute the logical derivation. Given the assumption *assume(x::posint)*, which asserts that x has the type *posint*, then *is(x::posint)* returns true.

4.2 Automated Theorem Proving

The idea of automated theorem proving is to use a computer to construct proofs of results that are expressed in a given logic from a basic set of 'facts' or *axioms*. Automatic theorem proving, that is theorem proving that requires no interaction with the user, is desirable. However, truely automatic theorem provers can be built only for very simple logics, which limits the types of theorems that can be expressed. Higher order logics allow quantification over individuals, predicates and functions and thus are more expressive than zero or first order logics, which have, at most, syntactic categories for individuals, functions, and predicates, but do not allow quantification over predicates and functions.

Within higher–order theorem proving there are various approaches to proof construction. The Coq proof assistant [26] is based on the calculus of inductive constructions, which is a higher–order typed λ–calculus. Coq is a *constructive proof* system, which does not allow the *law of excluded middle* and associated principles, such as the law of double negation and proof by contradiction, in proofs. Coq proves a proposition by representing it as a type and showing the type to be inhabited (by finding an inhabitant of the type). The theorem prover HOL [53] is an LCF–style theorem prover. It is based on typed λ–calculus and uses a core of primitive inference rules in typed classical higher–order logic to construct proofs. HOL theorems can only be created by applying sequences of these primitive rules, either manually or by invoking functions which apply a sequence of rules. When applied, these functions are broken down into their primitive inferences and these expansions may be very inefficient. Recent versions of HOL allow limited use of decision procedures that are not based on the primitive inference rules. The theorem prover PVS [103] is also based on a typed λ–calculus; however, PVS allows predicate and dependent subtyping. PVS provides primitive inference rules along with decision procedures for various theories, such as linear arithmetic, for the construction of proofs. Some may question the soundness of decision procedures; however, if used in a disciplined manner they can be quite safe and generally lead to simpler, more efficient proofs.

Coq, HOL and PVS are semiautomatic theorem provers, which means that they can perform

some automatic proving but may also require some user input to complete proofs. Each of these systems provides a means of writing *tacticals* or *strategies*, designed to capture common inference patterns. These proof strategies generally consist of sequences of rules to be applied and strategies for controlling their application.The ML language forms the strategy language of both Coq and HOL, whereas the PVS strategy language is based on the Lisp language.

Each of these systems offers varying degrees of automation, efficiency, expressibility and usability. There are various schools of thought on which of these systems is 'best'. Those who subscribe to constructivism rather than classicism prefer Coq and generally perceive PVS to be unreliable. For those who subscribe to classicism, their view on the use of decision procedures generally informs their preference between HOL and PVS.

PVS is chosen for use in the application to the formal and symbolic analysis of control systems.

4.2.1 PVS

PVS (Prototype Verification System) [104] is a specification and verification tool based on classical, higher order logic. It consists of a specification language and a theorem prover, which supports a high level of automation. Specifications in PVS are organised into *theories*. Conventionally, a theory appears in a file of the same name, which must end in the extension '.pvs'. Each theory has an associated proof file containing the proof commands that have been used to (partially) prove each lemma, corollary, postulate, etc, in the theory. Only the theories in the prelude file and the declarations preceding any point in the theory are visible at that point. To allow declarations in other theories to be accessed one may *import* theories. PVS keeps track of those theories that are currently visible through the import chain in a *context* file. During each proof PVS keeps track of the consequents to be proved (positive labels) and the antecedents currently available (negative labels). Lemmas that are visible at a given point can be used in a proof. Use of a lemma adds it to the

available antecedents.

The PVS language is strongly typed, that is every expression has an associated type, and supports predicate and dependent sub–typing. This subtyping provides PVS with an expressive language but also increases the complexity of the type system and of typechecking theories and expressions in PVS.

The type of an expression in PVS need not be unique. The type system is based on structural equivalence rather than name equivalence; two types are equal iff they have the same elements. There are strong links between predicates, sets and types in PVS; types are modelled as sets and sets are represented as predicates. Consider the following declarations:

```
p1: pred[real] = {x:real| 0<x}
p2: setof[real] = {x:real| 0<x}
p3: set[real] = {x:real| 0<x}
p4: TYPE = {x:real| 0<x}
p5: TYPE = {x:real| p1(x)}
```

The first two declarations `p1` and `p2` are semantically equivalent and are provided to emphasise the intention of their use; `pred` indicates a focus on properties whereas `setof` indicates a focus on elements. The third declaration `p3` can be used in place of the first two declarations but also has a number of the common operations on sets associated with it. The fourth and fifth declarations[1] describe the same type since every element of one of the types also belongs to the other; however, this is not immediately obvious to the PVS typechecker as named predicates, such as `p1`, are not expanded. PVS also allows types to be uninterpreted, for instance, one can declare a type `T` as some unspecified subset of $\mathbf{R}$:

```
T: TYPE FROM real
```

This is interpreted by PVS as shorthand for the following

```
T_pred: [real->bool]
T: TYPE = (T_pred)
```

[1]The fifth declaration can also be written `p5: TYPE = (p1)` in shorthand.

In order to cope with the complex sub–typing allowed in the PVS language the typechecker performs two passes of any theory being typechecked. In the first pass possible types for subexpressions are collected. In the second pass the typechecker recursively tries to determine a unique type for expressions based on the expected types. If the typechecker cannot do this then it generates a *type correctness condition* (TCC), which must be discharged. Given the above type declarations a type correctness condition would be generated in situations in which a variable of type `p4` is expected but a variable of type `p5` is found. The type information of a variable `x` of `pred4` is recognised by PVS as `real_pred(x) AND 0<x` whereas the type information of a variable `x` of `pred5` is recognised as `p1(x)`. This TCC can be discharged simply by expanding the definition of `p1(x)`. The typechecker often produces TCCs that are trivial to prove and may produce many redundant TCCs that essentially represent the same type obligation.

PVS allows *judgements* as means to provide the typechecker with extra type information about expressions, operations and types. When a judgement declaration is typechecked the judgement is added to the current *context* for use in typechecking expressions. Judgements inhibit the generation of TCCs during typechecking and are also used by the prover explicitly in the `typepred` command, which adds the type of subject to the sequent list, and implicitly in the `assert` command, where the type information is provided for simplification and rewriting. One can prevent the generation of TCCs where a variable of type `p5` is found in place of a variable of type `p4` declaring and proving the following judgement asserting that `p5` is a subtype of `p4`:

```
p5_is_p4: JUDGEMENT p5 SUBTYPE_OF p4
```

Theories may be parameterised; this allows abstraction from types and/or parameters used in the theory, allowing more general theories to be proven. For instance, the following theory named 'parameterised' is parameterised with the uninterpreted subtype of **R** `T` and an element of that type `a`, which are referenced in the (currently unprovable) lemma `not_one_element`.

```
parameterised [T: TYPE FROM real, a:T]: THEORY
```

```
  BEGIN
    not_one_element: LEMMA
      FORALL (x:T): EXISTS (y:T): x/=y
END parameterised
```

When accessing declarations in this theory from another one must provide values for T and a. PVS provides a facility to allow more detailed restrictions to be placed on a theory's parameters. One may *assume* that the parameters have given properties. For instance, one may wish to assert that a is not the smallest element in T:

```
parameterised [T: TYPE FROM real, a:T]: THEORY
  BEGIN
  ASSUMING
    a_not_least: ASSUMPTION EXISTS (x:T): x<a
  ENDASSUMING
    not_one_element: ASSUMPTION
      FORALL (x:T): EXISTS (y:T): x/=y
END parameterised
```

This assumption makes the previously unprovable lemma provable; however, when the declarations in this theory are accessed from outwith the theory the assumptions become TCCs which must be proven.

4.3 Quantifier Elimination

Quantifier Elimination (QE) is a general term meaning the removal of all quantifiers from a quantified formula to produce an equivalent formula in only the free variables of the original formula. An example of a quantifier elimination problem is that of determining whether a quadratic equation has any real roots i.e,

$$\exists x.\, \big(a \neq 0 \wedge ax^2 + bx + c = 0\big)$$

An equivalent quantifier free formula is

$$a \neq 0 \wedge b^2 - 4ac > 0$$

Many QE methods are applicable only to sentences in the language of real closed fields (RCF). Sentences in RCF are closed formulae built up from conjunctions, disjunctions and negations of equalities and inequalities between arbitrary polynomials in any number of variables with integer coefficients.

Tarski developed a QE method for the elementary theory of real closed fields [113] and showed that any QE method in this theory was a decision method. Many alternative QE methods for RCF have been suggested in attempts to improve efficiency [110, 34]. Various algorithms have been suggested for special types of problems involving trigonometric or transcendental functions [9, 123] but these are limited to very specific problems and often do not include support for inverse trigonometric or transcendental functions, such as arctan or the natural logarithm, which are important in control engineering and vital in the analysis of Nichols plot requirements.

An important algorithm in QE is *cylindrical algebraic decomposition* (CAD) [35], which significantly reduces the time complexity of QE. Several improvements to this method have been suggested which further improve efficiency and reduce complexity [36, 92].

In simplest terms, the CAD of a formula ϕ constructed from r-variate polynomials is a decomposition of r-dimensional real space $\mathbf{R}^r$ into a finite number of disjoint connected sets called *cells*, in each of which each distinct polynomial component of ϕ is sign–invariant. From this CAD any QE problem $\psi = (Q_{k+1}x_{k+1})...(Q_r x_r)\phi(x_1, ..., x_r)$, where $0 \leq k \leq r$, can be solved by examining sample points within each cell. The time complexity of this QE method is doubly–exponential in the number of variables, which is a significant improvement over previous methods and makes the practical application of QE to many real–world problems feasible.

Quantifier elimination and cylindrical algebraic decomposition have a wide range of applications in the fields of computer science, mathematics and control engineering. QE can be

used for the optimisation of polynomials, i.e, finding the maximum value of polynomials subject to polynomial inequality constraints, and CAD can be used to find the solutions of systems of polynomial equalities and inequalities. For this reason QE and CAD are ideally suited to stability analysis of control systems (as in Section 2.2.1) and have been successfully applied to Von Neumann stability analysis [86] and to common stability analysis problems [66]. QE has also been applied to the design of linear feedback control systems [70, Chapter 8] and to the design of nonlinear control systems [71] where design criteria can be reduced to problems involving polynomial constraints over polynomial inequalities.

4.4 Formalised Mathematics

The applications of CA and TP systems are limited by their respective weaknesses. CA systems (Section 4.1) are excellent at symbolic manipulation and often provide powerful methods for numerical calculations; however, they cannot guarantee correct results. For this reason their applicability to any formal, symbolic mathematical analysis is limited. Formal theorem provers (TPs) (Section 4.2) can guarantee correct results but are often difficult to use and inefficient for automatic symbolic manipulation and numerical calculations, which limits their practical application. Automated formal and symbolic analysis of control systems requires validated calculations, both symbolic and numerical. The Maple–PVS [2] system provides a link between the CA system *Maple* and the TP PVS. This system combines two industry–standard systems into a robust and highly programmable formalised mathematics system that is ideal for symbolic Nichols plot analysis.

There has been much interest in the development of systems that provide the power of a CA system and the rigour of a TP. Systems of this type fall into two main categories; computational support for TPs and formal support for CA systems. Systems such as Maple–HOL [61] and Maple–Isabelle [16] provide links between the TPs HOL and Isabelle and the CA system *Maple*, allowing the TPs to call upon the computational power of *Maple* under appropriate circumstance to increase efficiency of proof or proof search and the Omega proof

development system [75] supports the integration of computer algebra into mechanised reasoning systems at the proof planning stage. The majority of the workload falls on the TPs in these systems and only the limited operations of the CA system that can be validated can be used for any proof to remain sound. Redlog [42], Analytica [18] and Theorema [29] extend CA systems with support for formal theorem proving and the Maple–PVS [2] system provides a link between the CA system *Maple* and the TP PVS, allowing *Maple* to call upon the theorem proving power of PVS to increase the reliability of its results.

Chapter 5

A Decision Procedure for Positivity and Negativity of Finitely Inflective Functions

Many classical methods for control system analysis can be reduced to the problem of determining whether a given function is positive or negative in an interval, for example, Nichols plot analysis (see Section 2.2.1) and time–response analysis (see Appendix A.2). The functions involved in these forms of analysis often involve inverse trigonometric and transcendental functions. Many functions of this kind are *finitely inflective* in the sense that they are continuously differentiable with a finite number of points of inflection. Despite the power of quantifier elimination, the algorithms that have been suggested appear not to be applicable in sufficient degree to solve many problems involving this class of functions (Section 4.3).

The decision procedure presented here is applicable to this problem for functions $f : D \to \mathbf{R}$, where D is a closed convex subset of $\mathbf{R}$ and f is finitely inflective. The procedure is extended to be applicable to real functions of two variables.

The concepts of convexity and points of inflection are introduced in 5.1 and 5.2. Geometric

properties that can be inferred from convexity are introduced in 5.3. Some terminology is introduced in 5.4. The formulae for which the decision procedure is applicable are classified in Section 5.5 in terms of a fragment of a logic $\mathcal{L}_1$ for the reals and functions of a real variable. Section 5.6 introduces the concept of a (*minimal*) *isolated* formula along with an algorithm to convert arbitrary formulae in $\mathcal{L}_1$ into this form. Rather than apply to *prenex normal* formulae, as is often the case for decision procedures, the decision procedure described in this chapter applies to minimal isolated formulae. In Section 5.7 the procedure for deciding positivity/negativity of a finitely inflective function of one variable over an interval is described. In Section 5.8 the procedure is extended to functions of two variables.

5.1 Convexity

A set $D \subseteq \mathbf{R}^n$ is a *convex set* if every point that lies on the line segment between two points in D is also in D; it is *closed* if all limit points of D are in D. An interval is closed if it is one of the following: the empty interval $\emptyset$, $(-\infty, \infty)$, $(-\infty, b]$, $[a, \infty)$ or $[a, b]$, where a and b are real numbers. A set $D \subseteq \mathbf{R}$ is a (closed) convex set if and only if D is a (closed) interval.

Definitions for the convexity of functions appear in most introductory calculus text books [111, pp. 191–198], [3, pp. 264–268]. Although the definitions (and terminology[1]) vary slightly, fundamentally they represent the same geometric idea; that is that in some convex set $D \subseteq \mathbf{R}^n$ a function $f : D \to \mathbf{R}$ is convex if for all $\mathbf{x}$, $\mathbf{y} \in D$ the line segment between $(\mathbf{x}, f(\mathbf{x}))$ and $(\mathbf{y}, f(\mathbf{y}))$ lies on or above the graph of f. A function $f : D \to \mathbf{R}$, where D is a convex set in $\mathbf{R}^n$, is defined as *convex* (as illustrated in Figure 5.1) if the following inequality holds for all $\mathbf{x}$, $\mathbf{y} \in D$ and $0 \leq \theta \leq 1$:

$$f(\theta\mathbf{x} + (1-\theta)\mathbf{y}) \leq \theta f(\mathbf{x}) + (1-\theta) f(\mathbf{y}) \tag{5.1}$$

[1] [111] refers to functions as *convex* or *concave*, whereas, [3] uses *concave up* and *concave down*

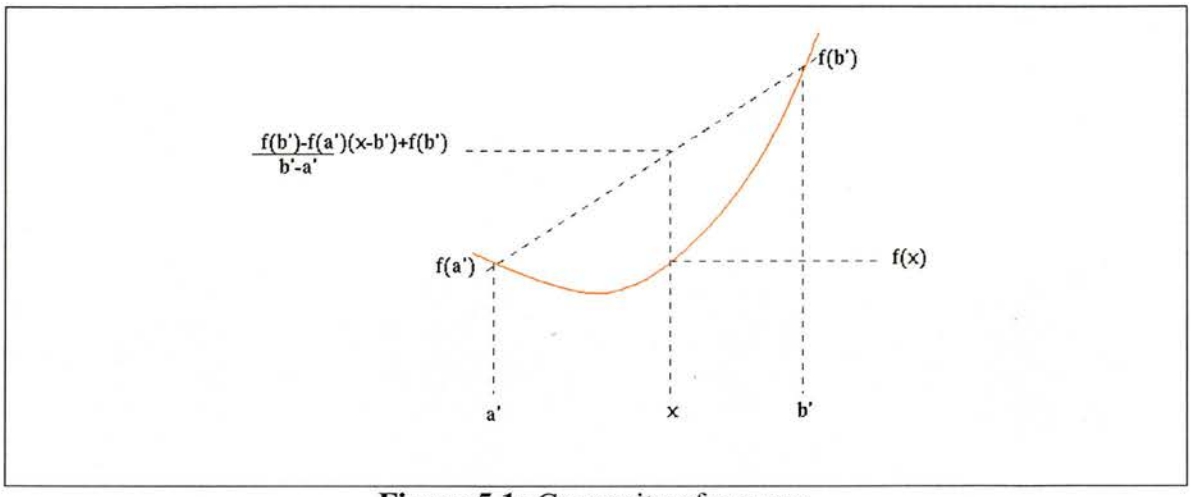


Figure 5.1: Convexity of a curve.

Reversing the inequality of Equation 5.1 gives the definition of a *concave function*. A function is *strictly convex* (or *strictly concave*) if the inequality is strict. A curve $\{(\mathbf{x}, y) : y = f(\mathbf{x})\}$ defined by a function $f : D \rightarrow \mathbf{R}$, where $D \subseteq \mathbf{R}^n$, is *convex/concave* in the regions in which the function is convex/concave. A function that is linear in all variables is both convex and concave but neither strictly convex nor strictly concave.

A differentiable function f in one variable is convex on the convex set $D \subseteq \mathbf{R}$ iff $f'(x)(y - x) \leq f(y) - f(x)$ and is concave on D iff the inequality is reversed. This is equivalent to the condition for convexity that f' is not decreasing on D and for concavity that f' is not increasing on D. A twice differentiable function f in one variable is convex on D iff $f'' \geq 0$. Proofs of these theorems appear in [13] and the theorems have also been formalised in PVS (see Chapter 6.1).

The sum of convex functions is a convex function [13] and equivalently the sum of concave functions is a concave function. From this it is clear that addition and subtraction of a function that is linear in all variables preserves convexity. Conclusions cannot be drawn about the convexity of the sum of a convex and a concave function.

For a function in several variables there may be regions over which the function is neither convex nor concave (consider $z = x^2 - y^2$). In these regions the function is convex in some directions and concave in others.

5.2 Points of Inflection

The concept of a point of inflection is rather vague and is generally not treated thoroughly in mathematical analysis texts. It can be described informally as a point at which the curvature of the graph of a function of one variable changes. This can be interpreted in a number of ways and is particularly ambiguous when dealing with linear functions; at any point the graph of a linear function is both convex and concave and can be considered both to change curvature and to retain the same curvature. To resolve this ambiguity we consider a point of inflection to be a point at which the graph of a function makes a 'smooth' transition between convexity and strict concavity (or between strict concavity and convexity). Given this informal description of a point of inflection it is clear that a function that is linear has no points of inflection even though it is both convex and concave on its domain. Trivially, the addition of a linear function to a given function leaves the points of inflection unchanged.

In general, for a differentiable function f in one variable the point c is a point of inflection if the tangent to the graph of f at $(c, f(c))$ crosses it at $(c, f(c))$ [111, p. 197]. For 'reasonable' twice differentiable functions the point c is a point of inflection iff $f''(c) = 0$ and, for all points a and b sufficiently close to c with $a < c < b$, $f''(a)$ has the opposite sign to $f''(b)$ [111, p. 197]. This would seem to be a suitably precise definition of a point of inflection except that the concept of a reasonable function is rather vague and is not specified in [111]. There are many seemingly appropriate definitions of reasonable functions, such as, those functions that are twice differentiable with a continuous second derivative; however, there are functions that meet these criteria but which do not behave in a reasonable manner. Consider a variation of the topologist's sine curve,

$$f = \begin{cases} x^5 \sin(\frac{1}{x}) & x \neq 0 \\ 0 & x = 0 \end{cases}$$

This curve is a sine wave whose wavelength and amplitude decrease as x approaches 0 (see Fig. 5.2). This function is continuously twice differentiable and has an infinite number of points of inflection in the interval $[-1, 1]$. At 0 both the first and second derivatives of this function are 0 and higher derivatives are undefined. The function has the property that one

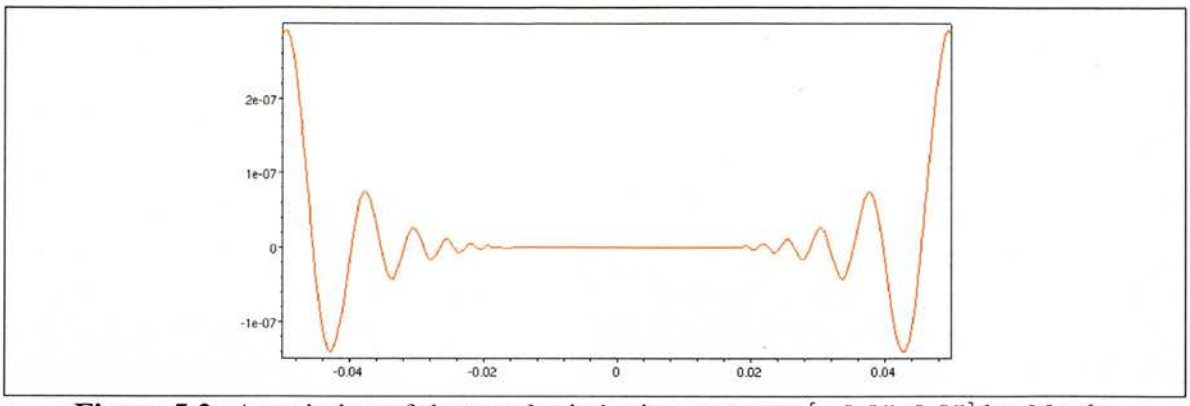


Figure 5.2: A variation of the topologist's sine curve on $[-0.05, 0.05]$ by *Maple*.

can not isolate any neighbourhood of 0 in which the sign of f'' is constant on each side of 0. This means that 0 neither lies in an interval of convexity or concavity nor is a point of inflection. One also cannot determine important properties of the function at 0, such as, whether it has a local maximum or minimum. This does not meet the intuitive concept of a reasonable function. In this context, for a function to be reasonable, each point in its range must be either lie in an interval of convexity or concavity, or be a point of inflection. This concept gives rise to the definition of a reasonable function and of a point of inflection:

Definition 5.1 *Reasonable function: a function of one variable that has a continuous second derivative and whose domain can be split into (in)finitely many intervals over each of which it is convex or concave*

Definition 5.2 *Point of inflection of a reasonable function f: a point c at which $f''(c) = 0$ and, for all points a and b sufficiently close to c with $a < c < b$, $f''(a)$ has the opposite sign to $f''(b)$*

This definition of a reasonable function includes functions such as x^2, sin and cos but excludes any variation of the topologist's sine curve over any interval containing 0. The trigonometric functions sin and cos over $\mathbf{R}$ are reasonable and have an infinite number of points of inflection.

5.3 Geometric Properties of Curves

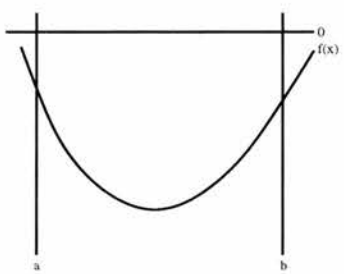
A number of geometric properties can be inferred from the definition of convexity. A differentiable convex function f is greater than or equal to any tangent to the graph of f. In any interval $[a,\ b]$ a convex function f is less than or equal to the chord joining $(a,\ f(a))$ and $(b,\ f(b))$.

Based on the convexity and these geometric properties of functions it is possible to determine the relative position of a convex (or concave) function f in one variable and the constant 0 function by examination of f at a number of carefully determined points.

The following details a set of conditions that allow the positivity or negativity of a convex function $f : D \to \mathbf{R}$, where D is the closed interval $[a,\ b]$, to be determined. The conditions for a concave function are simply reflections of the conditions for a convex function and can be worked out by looking at $-f$ in the appropriate intervals.

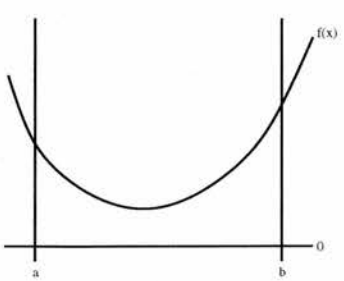
Suppose the function f is continuously differentiable and convex on the closed convex set $D = [a,\ b]$, then:

1. The function is negative on D iff the function is negative at the end points of D, i.e, $f(a) < 0$ and $f(b) < 0$

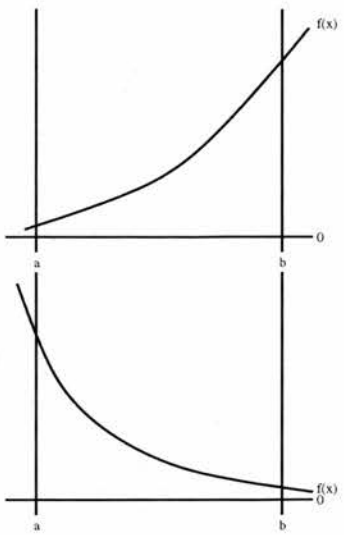


2. The function is positive on D iff one of the following two mutually exclusive conditions holds:

   (a) the gradient of the graph of f is equal to zero at some point within D and f is positive at that point, i.e, $\exists x \in D.\ f'(x) = 0 \wedge f(x) > 0$



(b) the gradient of the graph of f does not equal zero at any point within D and the function is positive at the end points of D, i.e, $\forall x \in D.\, f'(x) \neq 0$, $f(a) > 0$, and $f(b) > 0$



If none of the above conditions hold for a convex function then there is at least one point within the interval at which the function is equal to zero. These conditions have been formalised (see Section 6.3) in PVS along with proof of correctness and coverage of the cases.

These conditions can be used to determine the positivity or negativity of an arbitrary reasonable function f. The domain of f can be split into intervals $[a_i, b_i]$ over which f is either convex or concave. By applying the set of conditions detailed here to f in each interval $[a_i, b_i]$, one can determine whether f is positive or negative on its domain.

5.4 Classification

In this thesis, the notion of functions and their syntax corresponds to that of PVS [104]. As in the PVS language, functions are considered to be total maps from domain D to range B; that is each element in D is mapped to some element of B.

In this framework we define *linear* functions as those functions that can be represented by expressions that are linear in all variables, i.e, they are represented by expressions of the form $a_0 + \sum_{i=1}^{n} a_i x_i$ where a_i are real coefficients and x_i are real variables. *Polynomial* functions are those functions that can be represented by expressions consisting of the

sum of real variable(s) x_i raised to powers, multiplied by real coefficients a_i; for instance, polynomial functions in one variable are represented by expressions of the form $\sum_{i=0}^{n} a_i x^i$. *Rational* functions are those functions that can be represented by expressions made up of quotients of polynomials in the variables x_i with real coefficients a_i.

Rationally differentiable functions are defined to be those differentiable functions whose partial derivatives are rational functions, i.e, they are the expressions of the form $f(\mathbf{x})$ such that ∇f exists and is a vector of rational functions. *Continuously differentiable* functions are those differentiable functions whose partial derivatives are continuous, i.e, they are expressions of the form $f(\mathbf{x})$, such that ∇f exists and is a vector of continuous functions. *Continuously twice differentiable* functions are those twice differentiable functions whose partial derivatives are continuously differentiable. *Reasonable* functions (see Section 5.2) are those continuously twice differentiable functions of one variable whose domains consist of intervals over which the functions are convex or concave. *Finitely inflective* functions of one variable are those reasonable functions that have a finite number of points of inflection. A function of several variables is *finitely inflective in its i-th variable x_i* if fixing all but the i-th variable gives a finitely inflective function in the variable x_i. *Finitely inflective* functions of several variables are those functions of several variables that are finitely inflective in all variables.

5.5 Language $\mathcal{L}_1$

The metavariable symbol f is used to range over functions of a real variable, for example, $\lambda x.\ \sin^2(x)$, $\lambda y.\ \ln(y) - 3\cos(y)$, etc.

In the language $\mathcal{L}_1$ an *atomic formula in the variable x* and *with domain D* is an expression $f(x) \sim 0$, where D is a convex subset of $\mathbf{R}$, f is a total function[2] from D to $\mathbf{R}$, and $\sim$ is one of the relational operators $\leq$, $< =$, $>$, $\geq$, $\neq$; in interpreting the formula the variable

[2]As in the PVS language, predicate subtyping can be used to model a partial function f as a total function from the domain $D \subseteq \mathrm{dom}(f)$, upon which it is defined, to its range.

x will be restricted to range over D. Atomic formulae are essentially tuples of a domain and an inequality, for example, the tuples $(\sin^2(x) < 0, [0, \pi])$ and $(\ln(y) - 3\cos(y) \geq 0, [-1, 1])$ represent atomic formulae in the variables x and y and with the domains $[0, \pi]$, $[-1, 1]$ respectively.

It should be noted that the negation of an atomic formula in the variable x with the domain D is also such an atomic formula, since the negation of a relational operator is also a relational operator, for example, $\neq$ is the negation of $=$, $<$ is the negation of $\geq$, etc. The values *TRUE* and *FALSE* can be represented as atomic formulae using the expressions $\lambda x.\, 0 = 0$ and $\lambda x.\, 1 = 0$ with the domain $\mathbf{R}$. Thus, *literals* in the language $\mathcal{L}_1$ are simply atomic formulae.

Arbitrary formulae are built from atomic formulae using the propositional operators $\vee$ and $\wedge$ and the quantifiers $\exists$, $\forall$. Arbitrary formulae are associated with zero or more variables and domains but care must be taken when determining these. An arbitrary formula constructed from formulae in the same variable is also in this variable and its domain is the intersection of the domains of the subformulae, for example, $\sin^2(x) \geq 0 \wedge \ln(x) - 3\cos(x) < 0$ is a formula in the variable x with domain $D_1 \cap D_2$, where D_1 and D_2 are the domains of the subformulae. An arbitrary formula constructed from formulae in different variables is in these variables and its domain is the product of the domains of the subformulae, for example $\sin^2(x) \geq 0 \wedge \ln(y) - 3\cos(y) < 0$ is a formula in the two variables x and y with domain $D_1 \times D_2$, where D_1 and D_2 are the domains of the subformulae.

Quantifying over a variable x which appears in a formula $A(x)$ produces a formula in the variables excluding x and with domain no longer including the domain of x, i,e, given the formula $A(x_1, \ldots, x, \ldots, x_n)$ in variables including x, the range of x is D and the domain of $A(x)$ is $D_1 \times \cdots \times D \times \cdots \times D_n$, then the quantified formulae $\forall x \in D.\, A(x)$ and $\exists x \in D.\, A(x)$ are formulae in the variables $x_1, \ldots, x_n$, excluding x, with the domain $D_1 \times \cdots \times D_n$, excluding D. For example, applying universal quantification to the variable y in the formula $\sin^2(x) \geq 0 \wedge \ln(y) - 3\cos(y) < 0$, which has domain $D_1 \times D_2$, gives the formula $\forall y.\ \sin^2(x) \geq 0 \wedge \ln(y) - 3\cos(y) < 0$ in the variable x with the domain D_1.

If all occurrences of all variables in a formula A are quantified then A is a *closed formula* or *sentence* and, given the natural interpretation of formulae over reals, is either true or false.

5.6 Minimal Isolated Formulae

Many decision procedures are applicable to formulae in *prenex normal* form, that is formulae in which there are no quantifiers within the scope of any of the logical connectives. This normal form is not ideal for application of the decision procedure described in this chapter. A more appropriate normal form, *(Minimal) isolated form* is introduced.

A formula is in *isolated form* if there is no nesting of quantifiers within the formula, i.e, it is built up from conjunctions and disjunctions of formulae each of which contains either a single quantifier or no quantifier. For instance, given the atomic formulae $A(x)$, $B(x)$ and $C(y)$, where x does not appear free in C and y does not appear free in A or B, the following formula are in isolated form

$$A(x)$$
$$\exists x.\, A(x)$$
$$\exists y.\, A(x)$$
$$((\forall x.\, A(x)) \wedge (\forall x.\, B(x))) \vee \exists y.\, C(y)$$
$$(\forall x.\, A(x) \wedge B(x)) \vee \exists y.\, C(y).$$

However, the following are not in isolated form as they contain nested quantifiers

$$\exists x.\, \forall y.\, A(x)$$
$$\exists y.\, (\forall x.\, A(x) \wedge B(x)) \vee C(y).$$

A formula is in *minimal isolated form* if it can not be reduced using any of the rules given in Table 5.1 [76, p. 162] applied from left to right. A formula in isolated form need not be in minimal isolated form; for instance, the following formulae are in isolated but not minimal isolated form

$$\exists y.\, A(x)$$
$$(\forall x.\, A(x) \wedge B(x)) \vee \exists y.\, C(y).$$

A formula in minimal isolated form is also in isolated form; for instance, the following formulae are in both isolated and minimal isolated form

$$A(x)$$
$$\exists x.\, A(x)$$
$$((\forall x.\, A(x)) \wedge (\forall x.\, B(x))) \vee \exists y.\, C(y).$$

Any formula that can be written in isolated form can also be written in minimal isolated form by repeatedly applying the rules given in Table 5.1 from left to right to reduce the formula. Since there is no nesting of quantifiers in an isolated formula A, the scope B of each quantifier Q in A is a conjunction, a disjunction or an atomic formula. If B is an atomic formula then $Q.\, B$ is either in minimal isolated form or can be reduced to an atomic formula, which is by definition minimal isolated, by the application of rules (R1.a), (R1.b), (R2.a) or (R1.b). If B is a conjunction or disjunction then Q can be distributed across the disjunction or conjunction by application of rules (R3), (R4), (R5) or (R6). These rules can be applied repeatedly to reduce the scope of each quantifier until the scope cannot be reduced further. The formula is then by definition in minimal isolated formula. An algorithm for the conversion of a formula to a minimal isolated formula is given in Section 5.6.1 and a proof of the termination of the algorithm has been produced in PVS (Section 6.6)

5.6.1 Quantifier Isolation

Just as every formula of a first–order logic can be converted into prenex normal form, every formula of $\mathcal{L}_1$ can be converted into minimal isolated form. This is owing to the fact that each atomic formula of $\mathcal{L}_1$ contains at most a single variable. Formulae of languages that allow more than one variable in an atomic formula can not necessarily be converted into (minimal) isolated form; for instance, given the atomic formulae $A(x)$, $B(x)$ and $C(x, y)$ the following formula cannot be reduced to isolated form

$$(\forall x.\, A(x) \wedge B(x) \vee \exists y.\, C(x, y)).$$

(R1.a)	$\forall x.\, A \iff A$	x does not appear free in A and the range of x is non–empty
(R1.b)	$\forall x.\, A(x) \iff TRUE$	the range of x is empty
(R2.a)	$\exists x.\, A \iff A$	x does not appear free in A and the range of x is non–empty
(R2.b)	$\exists x.\, A(x) \iff FALSE$	the range of x is empty
(R3)	$\forall x.\, (A(x) \wedge B(x)) \iff \forall x.\, A(x) \wedge \forall x.\, B(x)$	
(R4)	$\exists x.\, (A(x) \vee B(x)) \iff \exists x.\, A(x) \vee \exists x.\, B(x)$	
(R5)	$\forall x.\, (A \vee B(x)) \iff A \vee \forall x.\, B(x)$	x does not appear free in A
(R6)	$\exists x.\, (A \wedge B(x)) \iff A \wedge \exists x.\, B(x)$	x does not appear free in A

Table 5.1: Rules for transfer of quantifiers

Quantifier isolation is a general term describing a process for reducing a formula into isolated form. Quantifier isolation can be performed using the standard rules governing the transfer of quantifiers over conjunction and disjunction (see Table 5.1) applied left to right and a modified notion of conjunctive and disjunctive normal form (CNF and DNF). A formula is in CNF (or DNF) if it is a conjunction of disjunctions (or a disjunction of conjunctions) of literals [55, p. 34] and every formula containing no quantifiers can be put into CNF or DNF. Modifying this notion for quantifier isolation, a formula is considered to be in modified CNF (or modified DNF) if it is a conjunction of disjunctions (or a disjunction of conjunctions) of literals and formulae in isolated form. Just as any formula containing no quantifiers can be converted into CNF or DNF, any formula containing only combinations of literals and isolated formulae can be converted into the modified CNF or modified DNF.

The following steps describe a method for quantifier isolation that reduces a formula A in $\mathcal{L}_1$ to minimal isolated form.

1. From A take the innermost quantified formula(e) $\psi = Qx.\, \phi$ that is not in minimal isolated form, where Q is a quantifier and ϕ is a formula that may or may not contain x. If the quantifier Q is $\exists$ then

(a) Rewrite ϕ in modified disjunctive normal form[3]

$$\phi' = \phi_1 \vee \cdots \vee \phi_n$$

where each ϕ_m is a conjunction of literals and minimal isolated formulae and may contain free occurrences of x and other variables.

(b) For each ϕ_m use the commutativity and associativity of conjunction[4] to collect all literals containing x free, giving [5]

$$\phi_m = \phi'_m[\wedge\phi''_m]$$
$$\psi' = Qx.\,(\phi'_0[\wedge\phi''_0]) \vee \cdots \vee (\phi'_n[\wedge\phi''_n])$$

where ϕ'_m and ϕ''_m are conjunctions of literals and minimal isolated formulae, ϕ'_m does not contain a free occurrence of x, and ϕ''_m must only contain free occurrences of only x.

(c) Apply (R2.a), (R2.b), (R4) and (R6) from left to right to move Q inward. This gives the minimal isolated formula

$$\psi'' = (\phi'_0[\wedge Qx.\,\phi''_0]) \vee \cdots \vee (\phi'_n[\wedge Qx.\,\phi''_n])$$

(d) Replace ψ with ψ'' in A to give A'. If A' is a minimal isolated formula then stop otherwise repeat the algorithm on A'.

2. If the quantifier Q is $\forall$ then

(a) Rewrite ϕ in modified conjunctive normal form[6]

$$\phi' = \phi_1 \wedge \cdots \wedge \phi_n$$

[3]Since ψ is the innermost quantified formula that is not in minimal isolated form, ϕ will contain only conjunctions or disjunctions of literals and minimal isolated formulae. Thus, ϕ contains only combinations of literals and isolated formulae and can be converted into modified DNF.

[4]Since ϕ' is in modified DNF, each ϕ_m will contain only conjunctions.

[5]The subformulae $[\wedge\phi''_n]$ do not appear if ϕ_n contains no free occurrences of x

[6]Since ψ is the innermost quantified formula that is not in minimal isolated form, ϕ will contain only conjunctions or disjunctions of literals and minimal isolated formulae. Thus, ϕ contains only combinations of literals and isolated formulae and can be converted into modified CNF.

where each ϕ_m is a disjunction of literals and minimal isolated formulae, and may contain free occurrences of x and other variables.

(b) For each ϕ_m use the commutativity and associativity of disjunction[7] to collect all literals containing x free, giving[8]

$$\phi_m = \phi'_m[\vee\phi''_m]$$
$$\psi' = Qx.\,(\phi'_0[\vee\phi''_0]) \wedge \cdots \wedge (\phi'_n[\vee\phi''_n])$$

where ϕ'_m and ϕ''_m are disjunctions of literals and minimal isolated formulae, ϕ'_m does not contain a free occurrence of x and ϕ''_m must only contain free occurrences of x.

(c) Apply (R1.a), (R1.b), (R3) and (R5) from left to right to move Q_i inward. This gives the minimal isolated formula

$$\psi'' = (\phi'_0[\vee Q_i x_i.\,\phi''_0]) \wedge \cdots \wedge (\phi'_n[\vee Q_i x_i.\,\phi''_n])$$

(d) Replace ψ with ψ'' in A to give A'. If A' is a minimal isolated formula then stop otherwise repeat the algorithm on A'.

Any quantified formula $Qx.\,\phi(x)$ within a minimally isolated formula ψ will have one of three forms: ϕ is a literal; Q is $\exists$ and ϕ is a conjunction of literals; Q is $\forall$ and ϕ is a disjunction of literals. Consider the formula

$$\psi = \forall x.\,A(x) \wedge B(x)$$

This formula is in isolated form but is not in minimal isolated form. Performing quantifier isolation as described above yields the following formula in minimal isolated form

$$\psi'' = (\forall x.\,A(x)) \wedge (\forall x.B(x))$$

Now consider the formula

$$\psi = \forall x.\,A(x) \vee B(x)$$

[7]Since ϕ' is in modified CNF, each ϕ_m will contain only disjunctions.

[8]The subformulae $[\vee\phi''_n]$ do not appear if ϕ_n contains no free occurrences of x

This formula is in isolated form and cannot be reduced using quantifier isolation so it is also in minimal isolated form.

Quantifier isolation relies on the fact that each literal in the language $\mathcal{L}_1$ contains only a single quantifiable variable. Consider the four formulae $A(x)$, $B(x)$, $C(x)$ and $D(x)$, and the prenex formula

$$\phi(x_1, x_2, x_3, x_4) = (A(x_4) \wedge A(x_1)) \vee ((B(x_2) \wedge C(x_3)) \vee D(x_4))$$
$$\psi = \forall x_1.\, \exists x_2.\, \forall x_3.\, \exists x_4.\, \phi(x_1, x_2, x_3, x_4)$$

This can be rewritten in minimal isolated form

$$\psi'' = ((\exists x_4.\, A(x_4)) \wedge \forall x_1.\, A(x_1)) \vee ((\exists x_2.\, B(x_2)) \wedge \forall x_3.\, C(x_3)) \vee \exists x_4.\, D(x_4)$$

Now consider the formulae $A(x)$, $B(x)$, $C(x)$ and $D(x, y)$, and the prenex formula

$$\phi(x_1, x_2, x_3, x_4) = (A(x_4) \wedge A(x_1)) \vee ((B(x_2) \wedge C(x_3)) \vee D(x_1, x_4))$$
$$\psi = \forall x_1.\, \exists x_2.\, \forall x_3.\, \exists x_4.\, \phi(x_1, x_2, x_3, x_4)$$

This cannot be written in isolated form as we cannot isolate $D(x_1, x_4)$ in such a way that it is within the scope of only one quantifier.

5.7 Decision Procedure for Functions of One Variable

The decision procedure described in this section was developed to take those minimal isolated formulae (as defined in Section 5.6) in which all functions are finitely inflective and the scope of any quantifier is a literal formula in the language $\mathcal{L}_1$, and output the truth of the formula[9].

The procedure was developed to be applied to the analysis of control systems and is applicable not only to sentences of real closed fields but also to any function whose derivative is a rational function, including the natural logarithm and arctan.

[9]The decision procedure cannot be applied to formulae in which the scope of any quantifier is a compound expression, e.g $\forall x.\, A(x) \vee B(x)$.

In order to use the conditions described here to decide a sentence ϕ of the language $\mathcal{L}_1$ the sentence must be converted into minimal isolated form using the algorithm described in Section 5.6.1. The domains of the functions in ϕ must be split into intervals over which the function is either convex or concave.

The decision procedure takes a sentence ϕ in minimal isolated form, in which the scope of any quantifier is a literal formula in the language $\mathcal{L}_1$ and all functions f_i in ϕ are finitely inflective, and performs the following steps:

1. Convert existential quantification in ϕ to universal quantification giving ϕ'. This is a syntactic conversion to simplify the algorithm: $\exists x.P(x)$ becomes $\neg\forall x.\neg P(x)$

2. Take each quantified formula A in ϕ' containing a single literal $f_i \sim_i 0$ then determine the intervals D_{ij} of convexity and concavity for f_i.

3. For each of the intervals D_{ij} within $\mathbf{dom} f_i$ apply the appropriate case from the set of conditions given in Section 5.3. If the correct conditions hold for all these intervals then the i-th formula has the value TRUE. If the condition fails to hold for any interval then the formula has the value FALSE.

4. Construct the truth value for the sentence ϕ' (and thus ϕ) by applying the propositional operators within it to the truth values of Step 3.

The procedure presented in this section is applicable to functions that are finitely inflective, that is reasonable functions that have a finite number of intervals in which the curve is convex or concave.

Section 6.7 gives the proof in PVS of the correctness and termination of the procedure.

5.8 Decision Procedure for Functions of Two Variables

Given a set $\mathbf{D} \subseteq \mathbf{R}^n$ and functions $a, b : \mathbf{D} \to \mathbf{R}$, the subset $[a, b] \times' \mathbf{D}$ of $\mathbf{R} \times \mathbf{R}^n$ is defined as

$$\{(x, \mathbf{y}) : \mathbf{y} \in \mathbf{D} \wedge a(\mathbf{y}) \leq x \leq b(\mathbf{y})\}.$$

For a (partial) function $f : \mathbf{R} \times \mathbf{R}^n \to \mathbf{R}$, with domain $\mathbf{D}' = [a, b] \times' \mathbf{R}^n$, the partial derivative $\frac{\partial f}{\partial x}\big|_{(x,\mathbf{y})}$ is denoted by $f_1(x, \mathbf{y})$.

By fixing all but the k–th variable of a continuously twice differentiable function f of $n + 1$ variables, the conditions presented in Section 5.3 for determining the positivity or negativity of a function in one variable may be applied to f. For simplicity, consider a continuously twice differentiable function $f : \mathbf{R} \times \mathbf{R} \to \mathbf{R}$ with domain $\mathbf{D}' = [a, b] \times' \mathbf{D}$, where $a, b : \mathbf{D} \to \mathbf{R}$ and $\mathbf{D} \subseteq \mathbf{R}$. If for all $y \in \mathbf{D}$, $\lambda x \in [a(y), b(y)].f(x, y)$ is convex then the conditions in Section 5.3 correspond to:

1. f is negative on $\mathbf{D}'$ iff for all $y \in \mathbf{D}$, f is negative at $(a(y), y)$ and at $(b(y), y)$.

2. f is positive iff for all $y \in \mathbf{D}$ one of the following two mutually exclusive conditions holds

   (a) the derivative of f with respect to its first argument (f_1) is equal to zero at some point within $[a(y), b(y)]$ and f is positive at that point, i.e,

   $$\exists x \in [a(y), b(y)].\, f_1(x, y) = 0 \wedge f(x, y) > 0.$$

   (b) the derivative of f with respect to its first argument (f_1) does not equal zero at any point within $[a(y), b(y)]$, and f is positive at $(a(y), y)$ and at $(b(y), y)$.

If the domain of f can be split into a finite number of regions over which it is convex or concave in one of its variables, then applying the above conditions in each of these regions will result in inequalities between the constant 0 function and functions g in only

one variable. If each g is finitely inflective then the decision procedure of Section 5.7 can be applied.

The conditions and decision procedure can be extended to higher dimensions by considering (partial) functions $f : \mathbf{R} \times \mathbf{R}^n \to \mathbf{R}$ with domain $\mathbf{D}' = [a, b] \times' \mathbf{D}$, where $a, b : \mathbf{D}' \to \mathbf{R}$ and $\mathbf{D} \subseteq \mathbf{R}^n$.

Chapter 6

Formalisation of the Decision Procedure

To demonstrate the correctness of the decision procedure introduced in Chapter 5, the procedure along with its mathematical foundations and the logic $\mathcal{L}_1$ were formalised in the formal theorem prover PVS (Section 4.2.1). Proofs of the correctness of the procedure and mathematical foundations were first sketched by hand; however, the formalisation of $\mathcal{L}_1$ was done directly in PVS. The formalisation in PVS provides greater assurance of correctness than hand proofs alone and many of the lemmas concerning the mathematical foundations are re–used for the automated analysis of Nichols plots. This chapter details this formalisation, highlighting some of the interesting issues arising from the use of PVS, and focuses mainly on the definitions of important concepts and the main results. The chapter includes fragments of the PVS code used to model and reason about the concepts introduced in the previous chapter along with high level descriptions of the code.

The PVS code is organised into separate theories. Many of the theories are parameterised with the uninterpreted subtype `T: TYPE FROM real`. As stated in Section 4.2.1 PVS interprets this essentially as shorthand for the the two definitions,

```
T_pred: [real ->bool]
T: TYPE = (T_pred)
```

the first of which is a predicate `T_pred` from $\mathbf{R}$ to boolean, and the second of which is a

type T that contains those elements for which T_pred is true.

As noted in Section 4.2.1 types and sets are closely related and in this chapter the two terms may be used interchangeably in reference to PVS types, sets and predicates. As in PVS notation, when referring to a predicate P as a type the predicate name is enclosed in parentheses (P).

The structure of this Chapter follows the structure of Chapter 5. Sections 6.1 and 6.2 details the modelling of convexity and of points of inflection in PVS. The representation of the geometric properties of curves in PVS is detailed in Section 6.3. The modelling of different classifications of functions is detailed in Section 6.4. The representation of the fragment of a logic $\mathcal{L}_1$ is detailed in Section 6.5. Section 6.6 details the representation of quantifier isolation in PVS. In Section 6.7 the representation of the decision procedure is described.

6.1 Convexity

Convexity is defined only for those functions whose domains are convex sets (Section 5.1). When modelling the concept of a convex set in PVS it is useful to keep in mind the particular usage to which it will be put. The ultimate aim is to reason about functions over subsets of their domains. The function domains are likely to be subsets of the reals.

Defining convexity for subsets of $\mathbf{R}$ is relatively straightforward in PVS. The new type convex_set is introduced. This type is the set of all subsets P of $\mathbf{R}$ for which all points lying between two points in P are also in P.

```
convex_set: TYPE = {P: set[real]|
  (FORALL (x,y: (P)), (z: real):
    x<=z AND z<=y IMPLIES P(z))}
```

This can be used to reason about the domains, and subsets of the domains, of functions; however, due to the complex nature of the PVS type system this definition will often cause

type correctness conditions (TCCs) to be generated. Consider a function f with domain T$\subset$ **R** and a predicate P from T. Since P can only be applied to elements of T, using the definition above to assert that P is convex leads to the generation of a TCC to ensure that any real variable to which P is applied is an element of T

```
FORALL (x,y: (P)), (z: real):
    x<=z AND z<=y IMPLIES T_pred(z)
```

If T is convex then this TCC is trivial to prove; however, it cannot be dismissed automatically.

To prevent the generation of this commonly occurring and often trivial TCC the definition of convex_set is altered to allow sets from arbitrary subsets of **R** to be declared as convex. The PVS theory in which this type is declared is parameterised with the uninterpreted type T, which is a subset of **R**. The altered type definition requires the implicit assumption that T is also a convex set. This can be modelled in two ways in PVS, either by using the assumption facility to assert that T is convex, or by building this assumption directly into the definition of a convex set. The first approach more accurately models the way mathematicians describe a convex subset i.e, 'given that T is convex and P$\subseteq$T then P is a convex set iff '...; however, the second is more convenient for use in PVS. The use of the assumption facility in the first approach means that each time the type is used outside the theory in which it is defined a TCC representing the assumption must be proved.

The second method for modelling a convex set essentially asserts that both P and T are convex in the domain of P avoiding the use of the assumption facility

```
convex_set: TYPE = {P: set[T]|
  (FORALL (x,y: (P), z: real): x<=z AND z<=y IMPLIES
    T_pred(z) AND P(z))}
```

Rather than generating TCCs this definition can be used to discharge some frequently occurring TCCs.

The concepts of nontrivial sets and nontrivial convex sets are modelled in a similar manner

as the new types `nontrivial_set` and `nontrivial_convex_set`.

The two types `convex_set` and `nontrivial_convex_set` have two important practical uses. Firstly, they allow the simplification of the definition of lemmas. Secondly, they allow the automatic discharge of some frequently occurring TCCs.

Two new theories are used to introduce several basic types representing subintervals. The first theory is parameterised with a subtype `T` of $\mathbf{R}$ and an element `a` of `T`. The four types `lt`, `le`, `ge` and `gt` are defined to represent the subsets of `T` containing all those elements of `T` that are less than, less than or equal, greater than or equal, or greater than the value `a`, respectively. The second theory is parameterised with a subtype `T` of $\mathbf{R}$ and two elements `a` and `b` of `T`. The four types `closed`, `open_l`, `open_r` and `open` are defined to represent the subsets of `T` containing all those elements of `T` that are in [a, b], (a, b], [a, b) or (a, b), respectively. These types are used to simplify the many definitions and lemmas used in the formalisation of the decision procedure. Defining these types in parametrised theories avoids problems associated with parameterised types (see Section 4.2.1). Rather than parameterising, for instance, the type `lt` with `T` and `a`

```
lt(T:pred[real],a:T): TYPE = {u:T|u<a}
```

the theory is parameterised allowing `lt` to be defined more elegantly as

```
lt: TYPE = {u:T|u<a}.
```

As well as allowing a more elegant definition of the types, parameterising the theory rather than the types also reduces the number of TCC generated when these types are used. For instance, given the definition `n: lt[real,2]`, the type of `n` is recognised as `n<2` rather than `lt(real,2)(n)`.

The concept of the convexity/concavity of a function is defined only on convex sets. The theories concerning the convexity of functions are parameterised with a type `T`$\subseteq \mathbf{R}$. In order to specify that `T` is convex the assumption facility is used.

```
connected_domainT: ASSUMPTION
  FORALL (x, y: T), (z: real):
```

```
x <= z AND z <= y IMPLIES T_pred(z)
```

The previously defined predicate `convex_set` can not be used in an assumption as it must appear at the beginning of the theory, before the main body in which definitions in other user defined theories may be accessed.

The concepts of convexity and concavity of functions could either be introduced as types of functions or as predicates over functions. Given the particular usage, predicates are the most natural choice, providing an easy means of reasoning about the convexity or concavity of arbitrary functions. The two predicates '`convex?`' and '`concave?`' are defined using the the basic geometric definition of convexity (Equation 5.1) and concavity.

```
concave?(f: [T->real]): bool =
  FORALL (x,y: T, l: closed[real,0,1]):
    f(l*x+(1-l)*y)>=l*f(x)+(1-l)*f(y)

convex?(f: [T->real]): bool =
  FORALL (x,y: T, l: closed[real,0,1]):
    f(l*x+(1-l)*y)<=l*f(x)+(1-l)*f(y)
```

The relationship between convexity and concavity is proven trivially in two lemmas.

```
concave_neg_convex: LEMMA FORALL (f: [T->real]):
  concave?(f) IFF convex?(LAMBDA (x: T): -f(x))

convex_neg_concave: LEMMA FORALL (f: [T->real]):
  convex?(f) IFF concave?(LAMBDA (x: T): -f(x))
```

The concepts of continuity and differentiability of functions are not defined within the PVS prelude library; however, they have been defined and made available in various user defined libraries. Dutertre developed a real analysis library [44], which was later updated and extended by Gottliebsen [54] in her transcendentals library. In both these libraries the definitions of continuity and differentiability are based on standard epsilon–delta definitions[1] using the predicates `continuous` and `derivable` respectively.

[1]Consider a non–trivial convex set T, a function $f : [T- > \mathbf{R}]$ and a point $x \in T$. f is continuous at x iff

$$\forall \epsilon \in \mathbf{R}^+ . \exists \delta \in \mathbf{R}^+ . \forall y \in T . |y - x| < \delta \implies |f(y) - f(x)| < \epsilon$$

Lemmas relating differentiability and continuity, as defined in the transcendentals library, to convexity, are given in their own theory. These lemmas are important in the modelling of the geometric properties of curves. The theory is parameterised with a type T. The same assumption about the convexity of T is made and the assumption that the set does not contain just a single element is also made. These assumptions are required as the differentiability of functions is only defined on domains for which these assumptions hold.

Two new types are introduced representing twice differentiable, and continuously twice differentiable functions.

```
fT: TYPE = {f:[T -> real] | derivable(f) AND
             derivable(deriv(f))}
fT3: TYPE = {f:[T -> real] | derivable(f) AND
              derivable(deriv(f)) AND
              continuous(deriv(deriv(f)))}
```

It is proven that if a function is (continuously) twice differentiable in T then it also has this property in any convex subset of T that does not contain only one element. The type not_one_convex_set_tcc is used to simplify the definition of the lemma and allow the assumptions that the domain of a differentiable function is a non singleton convex set to be discharged.

```
fT_subtype: LEMMA
  FORALL (P:not_one_convex_set_tcc[T]), (f:fT):
    derivable(LAMBDA (s:(P)): f(s)) AND
    derivable(deriv(LAMBDA (s:(P)): f(s)))

fT3_subtype: LEMMA
  FORALL (P:not_one_convex_set_tcc[T]), (f:fT3):
    derivable(LAMBDA (s:(P)): f(s)) AND
```

---

and is differentiable at x iff

$$\exists l \in \mathbf{R}.\forall \epsilon \in \mathbf{R}^{+}.\exists \delta \in \mathbf{R}^{+}.\forall y \in T.(y \neq x \wedge |y-x| < \delta) \implies \left| \frac{f(y)-f(x)}{y-x} - l \right| < \epsilon.$$

```
    derivable(deriv(LAMBDA (s:(P)): f(s))) AND
    continuous(deriv(deriv(LAMBDA (s:(P)): f(s))))
```

It is shown that on any nonempty nontrivial closed interval a differentiable function is convex (concave) iff the derivative is increasing (decreasing) on that interval and that a twice differentiable function is convex (concave) iff the second derivative is positive (negative) on that interval and that.

```
increasing_deriv: LEMMA
  FORALL (f:{f1:[T->real]|derivable(f1)}),
  FORALL (y:T,x:gt[T,y]):
    (convex?[closed[T,y,x]]
    (LAMBDA (s:closed[T,y,x]):f(s))
      IFF
    increasing
    (LAMBDA (s:closed[T,y,x]): deriv(f)(s)))

convex_aux4: LEMMA
  FORALL (f:fT,y:T,x:gt[T,y]):
    (convex?[closed[T,y,x]]
    (LAMBDA (s:closed[T,y,x]):f(s))
      IFF
    FORALL (s:closed[T,y,x]):
      deriv(deriv(f))(s)>=0)
```

6.2 Points of Inflection

It is impossible to model a change in convexity in PVS without reference to some neighbourhood around a point; however, as discussed in Section 5.2 this can lead to problems. Points of inflection are thus defined only for reasonable functions.

In order to model a reasonable function in PVS the concept of a reasonable domain is first introduced. Informally a reasonable domain for a function f is some group of sub–domains

in which f is either convex or concave. This could be modelled in PVS using one of several inbuilt PVS data structures — `set`, `list`, `sequence`, `finseq` — however, none of these structures is ideal.

Sets are unordered and do not provide an easy means of accessing specific elements. Lists in PVS are a recursive data type and although they are ordered and provide a means of accessing the n–th element they are generally cumbersome to use requiring the repeated expansion of definitions; for instance to access the n–th element of a list requires $n + 1$ expansions of the recursive function nth. Sequences in PVS are modelled using two different types `sequence` and `finseq` to represent infinite and finite sequences respectively. Sequences are ordered and provide a means of accessing the n–th element directly; however, infinite and finite sequences are unrelated in PVS, that is, `finseq` is not a subtype of `sequence`. An infinite sequence is simply a mapping from the natural numbers to elements of some type `T`; whereas, a finite sequence is a mapping from the natural numbers below some value `length` to elements of some type `T`.

Modelling reasonable domains for a function f using `sequence` would only allow domains which could be split into an infinite number of sub–domains for which f is either convex or concave, allowing, for instance, sin over $\mathbf{R}$ to have a reasonable domain but not x^2. On the other hand, modelling reasonable domains for a function f using `finseq` would only allow domains which could be split into a finite number of sub–domains for which f is either convex or concave, allowing, for instance, x^2 over $\mathbf{R}$ to have a reasonable domain but not sin.

To take advantage of the positive aspects of sequences but allow reasonable domains to be split into both finite and infinite sub–domains a new data type `mseq` is defined to bridge the gap between `sequence` and `finseq`. This type has a similar format to `finseq` in that it consists of a length and a mapping from the natural numbers indicated by the length to elements of some type `T`. If the length is -1 then the sequence is infinite and maps from the natural numbers to elements of `T` otherwise the sequence is a finite sequence and maps from the natural numbers less than the length to elements of `T`. A new type that allows this

mapping from length to the appropriate subset of the natural numbers is introduced.

```
mbelow [n:int]: THEORY
BEGIN

  mbelow: TYPE = {u:nat| IF n<0 THEN TRUE ELSE u<n ENDIF}

END mbelow
```

Defining this in a separate theory parametrised by an integer avoids problems associated with parameterised types allowing the type of, for instance, `n:mbelow[2]` to be recognised immediately as `n<2`.

The type `mseq` is defined in a separate theory parameterised with a type `T` as a record consisting of an integer greater than -2 that represents the length of the sequence and a function from the appropriate subset of the natural numbers to `T`. No functions are defined for elements of type `mseq`; however, judgements about equivalences between `mseq`, `finseq` and `sequence` are proved, allowing the functions defined for `finseq` and `sequence` to be applied to elements of `mseq` in appropriate circumstances.

```
mlength: TYPE = {u:int| -2<u}
m_sequence: TYPE =
  [# length: mlength, seq: [mbelow[length] -> T] #]
mseq: TYPE = m_sequence

finseq_mseq: JUDGEMENT finseq[T] SUBTYPE_OF mseq
mseq_finseq: JUDGEMENT {S:mseq| -1<S`length}
                SUBTYPE_OF finseq[T]
mseq_sequence: JUDGEMENT seq(S:{u:mseq| u`length<0})
                  HAS_TYPE sequence[T]
```

The predicate `reasonable_dom?` takes a twice differentiable function `f` and a sequence `S` of nontrivial convex sets. This predicate is true if for every sequential pair of nontrivial convex sets (S_n, S_{n+1}) in `S` the function `f` is convex in S_n (or S_{n+1}) and strict concave in S_{n+1} (or S_n). If `S` contains only one nontrivial convex set (S_0) then the predicate is true if `f`

is convex in S_0 or f is concave in S_0.

```
reasonable_dom?
(f:fT3, S:mseq[nontrivial_convex_set_tcc[T]]): bool =
  IF S`length/=0 AND S`length/=1 THEN
   (FORALL (n:mbelow[S`length-1]):
    (FORALL (u:(S`seq(n)),v:(S`seq(n+1))):
       sgn(deriv(deriv(f))(u)) /= sgn(deriv(deriv(f))(v))))
  ELSE
   (FORALL (n:mbelow[S`length]):
    (FORALL (u:(S`seq(n)),v:(S`seq(n))):
       sgn(deriv(deriv(f))(u)) = sgn(deriv(deriv(f))(v))))
  ENDIF
```

This strengthens the definition of a reasonable domain from a domain that can be spilt into intervals in which the function is convex or concave to a domain that can be spilt into intervals in which the function is convex or strictly concave. This removes any ambiguity introduced by linear functions and ultimately allows the domain of a function to be split into a unique sequence of sub–domains. To ensure that a unique sequence of sub–domain can be derived for a reasonable function two additional predicates are introduced:

```
complete?(S:mseq[pred[T]]): bool =
  FORALL (x:T): EXISTS (n:mbelow[S`length]): (S`seq(n))(x)

ordered?(S:mseq[pred[T]]): bool =
  FORALL (n:mbelow[S`length],m:mbelow[n]):
    FORALL (x:(S`seq(m)),y:(S`seq(n))): x<=y
```

The first predicate takes a sequence of predicates on T and returns true if every element of T is in at least one of the predicates in the sequence. The second predicate takes a sequence of predicates on T and returns true if every element of any predicate in the sequence is less than or equal to every element in any predicate later in the sequence than it.

A reasonable function is defined as a continuously twice differentiable function whose domain can be split into an ordered and complete sequence of nontrivial subsets of T

```
reasonable: TYPE =
  {f:fT3| EXISTS (S:mseq[nontrivial_convex_set_tcc[T]]):
   complete?(S) AND ordered?(S) AND reasonable_dom?(f,S)}
```

The set of points of inflection of a function are defined as the set of points z for which $f''(z) = 0$ and for which all points sufficiently close to z have different signs on either sides of z.

```
poi(f:reasonable): set[T] =
  {z:T| deriv(deriv(f))(z) = 0 AND
   EXISTS (x:lt[T,z], y:gt[T,z]):
     FORALL (m:open[T,x,z], n:open[T,z,y]):
       sgn(deriv(deriv(f))(m)) /=
       sgn(deriv(deriv(f))(n))}
```

6.3 Geometric Properties of Curves

The geometric properties of curves described in Section 5.3 involve reference to the derivative of functions. Differentiability is modelled in PVS as a predicate `derivable` on functions and is true if the function is differentiable on its domain. It is not immediately known in PVS that a differentiable function is also differentiable on any nontrivial subset of its domain. In the context of the decision procedure, which involves reasoning about functions on subsets of their domains, this could lead to a large number of type correctness conditions that are trivially true though not necessarily trivial to prove. For this reason the theory concerning geometric properties of curves is parameterised by a type `T`, which represents the domain of a function, and two values `a` and `b` from the type `T`, which represent the bounds on a closed convex set.

```
curve_bound[T:TYPE FROM real, a:T, b:{u:T|a<u}]: THEORY
```

Since differentiability is only defined on non–singleton convex sets the assumption must be made that `T` is such a set.

```
ASSUMING
  connected_domainT : ASSUMPTION
    FORALL (x, y: T), (z: real):
      x <= z AND z <= y IMPLIES T_pred(z)

  not_one_elementT: ASSUMPTION
    FORALL (x: T): EXISTS (y: T): x /= y

ENDASSUMING
```

Given these assumptions judgements are used to show that T has the type convex_set and not_one_convex_set, which suppresses the generation of certain TCCs and allows the automatic proof of others.

```
T_is_convex_set: JUDGEMENT
  T_pred HAS_TYPE convex_set[real]
```

The types AB and AB_open are defined as the closed and open intervals between a and b, respectively.

```
AB: TYPE = {u:real|a<=u AND u<=b}
AB_open: TYPE = {u:real| a<u AND u<b}
```

The closed interval is proven to be a convex set which does not contain only one element and the open interval is shown to be a subtype of the closed interval. Due to the definition of AB and AB_open it is not immediately obvious to PVS that these are in fact subtypes of T. Altering the definition of AB to {u:T|a<=u AND u<=b} would solve this but would cause the generation of many trivial TCCs during proofs. Using a judgement to show that AB is a subtype of T allows the PVS type checker to determine that AB is a subtype of T without causing the additional TCCs that would be generated with the altered definition.

Each of the conditions given in Section 5.3 is modelled and proven in a separate lemma, including the conditions for concave curves to be positive/negative. These lemmas are modelled in terms of functions lying above or below lines since addition or subtraction of linear functions does not change the convexity of a function. Six new predicates are defined

to represent the conditions for a convex or concave function to be less than, less than or equal, greater than, greater than or equal, equal, and not equal to a line in an interval. For example,

```
curve_gt_line
(f:{u:fT[T]|convex?(LAMBDA (x:AB): f(x)) OR
concave?(LAMBDA (x:AB): f(x))} ,m,c:real): bool =
   (convex?(LAMBDA (x:AB): f(x)) AND
    ((deriv(f,b)<m AND f(b)>m*b+c) OR
     (deriv(f,a)>m AND f(a)>m*a+c) OR
     (EXISTS (n:AB): deriv(f,n)=m AND f(n)>m*n+c)))
   OR
   (concave?(LAMBDA (x:AB): f(x)) AND
    (f(a)>m*a+c AND f(b)>m*b+c)).
```

These predicates are proven to hold iff the appropriate (in)equality holds. For example,

```
curve_gt_line_aux: LEMMA
  FORALL (f:{u:fT[T]|convex?(LAMBDA (x:AB): f(x)) OR
  concave?(LAMBDA (x:AB): f(x)f)},m,c:real):
   (FORALL (x:AB): f(x)>m*x+c) IFF
      curve_gt_line(f,m,c)
```

6.4 Classification

In tools such as MATLAB (Section 2.3), transfer functions are entered as a function applied to two lists of values. The first list represents the coefficients of the polynomial numerator of the transfer function, the second represents the coefficients of the denominator. A similar structure is adopted to model polynomials in PVS. A polynomial function is constructed from a finite sequence of real numbers. Given that a polynomial is a function of x, the i-th element of the sequence represents the coefficient of x^i. The polynomial is constructed by summing these values.

```
polynomial(S:finseq[real]): [real->real] =
  LAMBDA (x:real):
  sumto(S`length)
  (LAMBDA (n:below[S`length]): S`seq(n)*expt(x,n))
```

A function f is defined to be a polynomial if there exists a sequence of $\mathbf{R}$ from which f can be constructed using the `polynomial` function.

```
poly_type: TYPE =
  {f:[real->real]|
   EXISTS (S:finseq[real]): f=polynomial(S)}
```

The differentiability of any polynomial to any degree is proven along with value of those derivatives. Judgements are used to show that polynomials are differentiable functions. Proving this in a judgement rather than as a lemma allows the PVS type checker access to this information and can prevent the generation of certain TCCs.

This definition of a polynomial makes no assertion that the lead coefficient be nonzero. The recursive function `degree` is used to calculate the degree of a polynomial. The constant 0 function is considered to have degree 0.

```
degree(S:finseq[real]): RECURSIVE nat =
  IF S`length=0 THEN 0
  ELSIF S`seq(S`length-1)=0 THEN
    degree(subseq(S,0,S`length-2))
  ELSE
    S`length-1
  ENDIF
  MEASURE S`length
```

Linear, rational and rationally differentiable functions are all defined in terms of polynomials. Linear functions are defined as those polynomials whose degree is less than or equal to 1. Rational functions are defined as those functions that can be represented by a quotient of polynomials. This has the added requirement that the denominator of the quotient is always nonzero. Rationally differentiable functions are defined as those functions whose

derivative can be represented as a rational function.

Finitely inflective functions are defined in two ways. First, as a function whose domain can be split into a finite number of nontrivial intervals of convexity or strict concavity, and second, as a function with a finite number of points of inflection. Although the second definition models precisely the definition of a finitely inflective function as given in Section 5.4 the first is significantly easier to reason about and corresponds more closely to the notion of a reasonable function.

```
finitely_inflective: TYPE =
  {f:fT3| EXISTS (S:finseq[nontrivial_convex_set_tcc[T]]):
   complete?(S) AND ordered?(S) AND reasonable_dom?(f,S)}

finitely_inflective_alt: TYPE =
  {f:reasonable| is_finite(poi(f))}
```

In order to show the equivalence of the two definitions it is first shown that in any nonempty ordered sequence of nontrivial intervals all but the last interval in the sequence is guaranteed to have an upper bound. Similarly it is shown that all but the first is guaranteed to have a lower bound.

```
reasonable_dom_bounded_above: LEMMA
 FORALL (S:mseq[nontrivial_convex_set_tcc[T]]):
   ordered?(S) AND S`length/=0 IMPLIES
     FORALL (n:[below[S`length-1]]):
       EXISTS (x:T): upper_bound?(x,S`seq(n))

reasonable_dom_bounded_below: LEMMA
 FORALL (S:mseq[nontrivial_convex_set_tcc[T]]):
   ordered?(S) AND S`length/=0 IMPLIES
     FORALL (n:[below[S`length]]):
       n/=0 IMPLIES
         EXISTS (x:T): lower_bound?(x,S`seq(n))
```

The proofs of these lemmas rely on the fact that the sequence S is ordered and that every element S_n of S is a *nonempty* set. To show that S_n has a lower bound it must be shown that there exists a point that is less than or equal to all elements $x \in S_n$. An element y in S_{n-1} is chosen. Since S is ordered, $y \leq x$ and thus S_n is bounded below. A similar argument follows to show that S_n is bounded above.

For convenience the function `fi_dom` is declared to determine the sequence of domains into which the domain of a finitely inflective function can be split.

```
fi_dom(f:finitely_inflective):
 finseq[nontrivial_convex_set_tcc[T]] =
   choose({S:finseq[nontrivial_convex_set_tcc[T]]|
     complete?(S) AND ordered?(S) AND
       reasonable_dom?(f,S)})
```

6.5 Language $\mathcal{L}_1$

The formulae of $\mathcal{L}_1$ are modelled in PVS in two stages. First, they are modelled as syntactic constructs. This allows the manipulation of the structure of formulae, for instance, the conversion from disjunctive to conjunctive normal form or isolated form. Second, the semantics of the language are modelled. This allows assertions that the meaning of a formula does not change when a syntactic conversion is performed.

The formulae of $\mathcal{L}_1$ are built up recursively, for instance, a formula can be a conjunction of several formulae. PVS provides datatypes as a means of defining recursive types using a base case and constructors. Datatypes are simply a syntactic construction which hides the complexity involved in defining recursive types in PVS. Ideally, this construction would be used to model atomic formulae as the base type for the language, and conjunction, disjunction and quantification as constructors. However, it is not possible to do this and also specify the format of atomic formulae. Rather than using the datatype construction to hide the complexity, this must be modelled directly in PVS.

The uninterpreted type `frmla` is declared[2] and the base case and constructors are then defined axiomatically. The uninterpreted function `atomic` takes a predicate `P` over the reals, a name, a function from `(P)` to $\mathbf{R}$ and an element from an enumerated type `inq` that represents inequality signs and returns a `frmla`. An axiom is used to assert that such a function exists to allow the discharge of the equivalent TCC. The predicate `atomic?` takes a formula B and returns true if there exists a predicate `P`, a name `x`, a function `f` from `(P)` to $\mathbf{R}$ and an element `s` from `inq`, such that B =`atomic(P,x,f,s)`. The uninterpreted functions `dom`, `name`, `func` and `ineq` take a formula and return a predicate, name, function and element of `inq`, respectively. Axioms are used to assert that the predicate, name, function and element of `inq` returned when these functions are applied to an atomic formula are the expected values. For instance,

```
dom: [frmla -> pred[real]]

dom_atomic: AXIOM
  FORALL (P: pred[real], x: name),
  (f: [(P) -> real], s:inq):
    dom(atomic(P, x, f, s)) = P;
```

The values true and false are modelled as the atomic formulae `T` and `F`.

```
T: frmla = atomic({x:real| TRUE},
           char(0),
           LAMBDA (x:real): 0,
           eq)

F: frmla = atomic({x:real| TRUE},
           char(0),
           LAMBDA (x:real): 1,
           eq)
```

Conjunction and disjunctions are modelled as the uninterpreted functions `conj` and `disj` that take finite sequences of formulae, which have at least two elements, and return a for-

[2]Formula is a keyword in PVS therefore cannot be used as an identifier

mula. This construction allows the simplification of the syntactic conversions between the normal forms. The predicates `conj?` and `disj?` take a formula B and return true if there exists a finite sequence of formulae `S` such that B=`conj(S)` or B=`disj(S)` respectively. The uninterpreted function `args` takes a formula and returns a finite sequence of formulae. Axioms are again used to assert that the results of applying the function to a conjunction or disjunction are as expected. Axioms are also used to assert that atomic formulae, conjunctions and disjunctions are syntactically distinct, that is, a conjunction is not an atomic formula or a disjunction. For instance,

```
conj_not_atomic: AXIOM
  FORALL (A:frmla_seq2): NOT atomic?(conj(A))

conj_not_atomic_alt: LEMMA
  FORALL (A:(conj?)): NOT atomic?(A)
```

Universal and existential quantification are modelled as the uninterpreted functions `A` and `E` that take a name and a formula and return a formula. The predicates `A?` and `E?` take a formula B and return true if there exists a name `x` and a formula `C` such that B=`A(x,C)` or B=`E(x,C)` respectively. The uninterpreted functions `name` and `arg` take a formula and return a name and a formula, respectively. Axioms are again used to assert that the results of applying the function to a quantified formula are as expected. Axioms are also used to assert that atomic formulae, conjunctions, disjunctions and quantified formulae are syntactically distinct.

Many syntactic conversions of formulae can be modelled as recursive functions on formulae. However, in order to define recursive functions over formulae PVS requires a measure of the complexity of formulae that decreases on each recursive call of the function. This allows assertions about the termination of the recursive function to be made. Often the depth of a formula is used as a measure of its complexity; however, this measure is not suitable for the conversion of formulae into conjunctive normal, disjunctive normal or minimal isolated form as the depth of the formula may increase during this conversion. A more general measure of the complexity of a formula B can be considered to be some measure

of the complexity of the construction of B along with a measure of the complexity of the arguments of B; for instance if B is conjunction of the three formulae C, D and F then the complexity of B is a measure of complexity of the conjunction plus the complexity of C, D and F. The complexity of a formula is referred to as the *degree* of the formula.

The function deg_args is defined as a recursive function that takes a finite sequence of formulae and a function g from formulae to natural numbers representing some measure of the complexity of the formula. deg_args returns the sum of the complexity as defined by g of all the formulae in the sequence. The measure used to guarantee termination of this function is the length of the sequence as this decreases with each recursive call.

```
deg_args(f:finseq[frmla],g:[frmla->nat]):
RECURSIVE nat =
  IF f`length = 0 THEN 0
  ELSIF f`length = 1 THEN g(f`seq(0))
  ELSE g(f`seq(0)) + deg_args(f^(1,f`length-1),g)
  ENDIF
  MEASURE f`length
```

The uninterpreted function deg (for degree) is declared as a function that takes a formula and returns a natural number. Axioms are used to assert that the results of calling deg on an atomic formula is 0, of calling it on a conjunction or a disjunction is the length of the sequence of arguments plus the sum of the degree of the arguments, and of calling it on a quantified formula is 1 plus the degree of its argument. Mutual recursion is required in this definition; however, in PVS, mutually recursive functions cannot be defined directly. This is owing to the fact that in PVS a function can only refer to functions whose definitions precede it in the file or import chain. The definition of deg_args precedes the definition of deg, which means that deg can refer to deg_args in its definition but deg_args cannot refer to deg. By parameterising deg_args with a function g, deg can pass itself as an argument to deg_args. In this context, expanding the definition deg produces some expression involving deg_args and expanding deg_args produces some expression involving deg. This essentially produces mutual recursion between deg and deg_args.

```
deg:[frmla->nat]

deg_conj: AXIOM
  FORALL (A:frmla_seq2):
    deg(conj(A)) = A`length + deg_args(A,deg)
```

The type `frmla` is not restricted to those formulae that are constructed only from atomic formulae, conjunctions, disjunctions and quantified formulae, nor should it be. However, it is useful to be able to refer only to those (un)quantified formula, that are built up using the defined base case and constructors. The recursive predicate `frmla?` takes a formula and returns true if it is either an atomic formula or it is a conjunction or disjunction, whose arguments are of type `(frmla?)`.

```
frmla?(A:frmla): RECURSIVE bool =
  IF atomic?(A) THEN TRUE
  ELSIF conj?(A) OR disj?(A) THEN
    FORALL (n:below[args(A)`length]): frmla?(arg(A,n))
  ELSE FALSE
  ENDIF
  MEASURE deg(A)
```

The recursive predicate `qfrmla?` takes a formula and returns true if it is either an atomic formula, a conjunction or disjunction, whose arguments are of type `(qfrmla?)`, or a quantified formula whose argument is of type `(qfrmla?)`.

To determine the domain of arbitrary formulae the uninterpreted function `get_dom`, which takes a formula and returns a finite sequence of name and predicate pairings is declared. Axioms are used to define the behaviour of this function for specific types of formulae. Defining the function to determine the domain of an atomic or a quantified formula is relatively straight forward. The domain of an atomic formula `B` is simply its domain

```
get_dom(B) = dom(B).
```

The domain of a quantified formula `B` is the domain of the formula that is quantified with the domain of the quantified variable removed

```
get_dom(B) = remove(name(B),get_dom(arg(B))).
```

Defining a method to extract the domain of a conjunction or disjunction is a more complex matter. As stated in Section 5.5, the domain of such a formula is the intersection of the domains of the variables common to its subformulae and the product of the domains of the remaining variables. Since conjunctions and disjunctions are modelled essentially as functions that take two or more formulae and produce a formula, the domains of each of these subformulae must be found and composed in the appropriate manner. An auxiliary function that models the appropriate composition of two domains is defined. This is a recursive function that takes two sequences f and g of predicate/name pairings and returns a single sequence of such pairings. For each element of g, the function calls `contains` to determine whether the name appear in f. If it does, then the domain associated with that name is replaced in f with the intersection of the domain in f and the domain in g. If f does not contain the name then that element of g is appended to the sequence f. At this point there are no assertions made about the uniqueness of any name in either sequence.

```
dom_product(f,g:finseq[[pred[real],name]]):
RECURSIVE finseq[[pred[real],name]] =
  IF f`length=0 THEN g
  ELSIF g`length=0 THEN f
  ELSIF contains?(g`seq(0)`2,f) THEN
    dom_product((#length:=f`length,
      seq:=LAMBDA (n:below[f`length]):
      IF f`seq(n)`2=g`seq(0)`2 THEN
        ({x:real| f`seq(n)`1(x) AND g`seq(0)`1(x)},
         f`seq(n)`2)
      ELSE f`seq(n)
      ENDIF#),
      g^(1,g`length-1))
  ELSE
    dom_product((#length:=f`length+1,
      seq:=LAMBDA (n:below[f`length+1]):
      IF n=f`length THEN
```

```
          g`seq(0)
        ELSE f`seq(n)
        ENDIF#),
        g^(1,g`length-1))
    ENDIF
    MEASURE g`length
```

The definition of the domain of an arbitrary conjunction (or disjunction) can then be modelled using this function to combine the domains of each of its subformulae.

```
IF args(A)`length=2 THEN
  get_dom(A) =
    dom_product(get_dom(arg(A,0)),
       get_dom(arg(A,1)))
ELSE
  get_dom(A) =
    dom_product(get_dom(arg(A,0)),
      get_dom(conj(args(A)^(1,args(A)`length-1))))
ENDIF
```

The decision procedure of Section 5.7 is defined only on formulae that can be reduced to minimal isolated form, where the scope of each quantifier is a literal formula and the functions within the literal formulae are finitely inflective in the domain of the formulae. The domain of a formula must be bounded. Two new predicates are introduced, which together define formulae to which the procedure can be applied. The first defines a quantified formula whose scope is an atomic formula, whose `func` component is a finitely inflective function, and whose domain is bounded.

```
q_atomic?(B:frmla): bool =
  IF A?(B) THEN
    atomic?(arg(B)) AND
    (EXISTS (x,y:(dom(arg(B)))):
      x<y AND
      (FORALL (z:real):
         dom(arg(B))(z) IFF (x<=z AND z<=y))) AND
```

```
      derivable(func(arg(B))) AND
      derivable(deriv(func(arg(B)))) AND
      continuous(deriv(deriv(func(arg(B))))) AND
      convexity_props[(dom(arg(B)))].
         finitely_inflective?(func(arg(B)))
  ELSIF E?(B) THEN
    atomic?(arg(B)) AND
    (EXISTS (x,y:(dom(arg(B)))):
      x<y AND
      (FORALL (z:real):
         dom(arg(B))(z) IFF (x<=z AND z<=y))) AND
      derivable(func(arg(B))) AND
      derivable(deriv(func(arg(B)))) AND
      continuous(deriv(deriv(func(arg(B))))) AND
      convexity_props[(dom(arg(B)))].
         finitely_inflective?(func(arg(B)))
  ELSE
    FALSE
  ENDIF
```

The second predicate is recursive and defines formulae that are built from conjunctions and disjunctions of formulae for which q_atomic? holds.

```
dp_frmla?(B:frmla): RECURSIVE bool =
 IF qfrmla?(B) THEN
   IF atomic?(B) THEN
     FALSE
   ELSIF conj?(B) OR disj?(B) THEN
     FORALL (n:below[args(B)`length]):
        dp_frmla?(arg(B,n))
    ELSE
      q_atomic?(B)
    ENDIF
  ELSE
    FALSE
```

```
    ENDIF
    MEASURE deg(B)
```

6.6 Quantifier Isolation

To model quantifier isolation several key concepts are required. These concepts include the representation of formulae in various normal forms and conversion between these normal forms.

In Section 6.6.1 the representation in PVS of normal forms, including (minimal) isolated form, is detailed. Section 6.6.2 details the algorithm for conversion of formulae to normal forms, including conversion to minimal isolated form.

6.6.1 Normal Forms

Since conjunction and disjunction are modelled as functions which take a finite sequence of formulae rather than a pair of formulae, as is traditionally the case, conjunctions and disjunctions may be constructed as 'flat' as well as hierarchical structures, for instance, the following two formulae can be represented directly.

$$A \wedge B \wedge C \wedge D$$

$$((A \wedge B) \wedge C) \wedge D$$

No decision is forced about the associativity of the conjunctions in the first formulae but the associativity of the second can still be represented. The first formula is considered flat whereas the second is not. A flat formula is a formula that does not contain a conjunction of conjunctions or disjunctions of disjunctions. The concept of a flat formula is modelled as the recursive predicate `flat?`.

```
flat?(A:frmla): RECURSIVE bool =
  IF atomic?(A) THEN TRUE
```

```
ELSIF conj?(A) THEN
  FORALL (n:below[args(A)`length]):
    (NOT conj?(arg(A, n))) AND
    flat?(arg(A, n))
ELSIF disj?(A) THEN
 FORALL (n:below[args(A)`length]):
   (NOT disj?(arg(A, n))) AND
   flat?(arg(A, n))
ELSIF A?(A) OR E?(A) THEN
  flat?(arg(A))
ELSE FALSE
ENDIF
MEASURE deg(A)
```

The notions of conjunctive and disjunctive formulae are modelled as recursive predicates, which take a formula and return true if it is a conjunction (disjunction) and all of its arguments are either atomic formulae or conjunctive (disjunctive) formulae.

```
conjunctive_f?(B:frmla): RECURSIVE bool =
  IF conj?(B) THEN
    FORALL (n:below[args(B)`length]):
      (atomic?(arg(B,n)) OR conjunctive_f?(arg(B,n)))
  ELSE FALSE
  ENDIF
  MEASURE deg(B)
```

The concept of a *basic* formula is modelled using a predicate `basic?`, which takes a formula and returns true if it is either an atomic formula or a quantified formula that contains no other quantifiers within its scope. The modified notions of conjunctive and disjunctive formulae are modelled as recursive predicates, which take a formula and return true if it is a conjunction (disjunction) and all of its arguments are either basic formulae or conjunctive (disjunctive) formulae.

```
conjunctive?(B:frmla): RECURSIVE bool =
  IF conj?(B) THEN
```

```
    FORALL (n:below[args(B)`length]):
      (basic?(arg(B,n)) OR conjunctive?(arg(B,n)))
  ELSE FALSE
  ENDIF
  MEASURE deg(B)
```

Modified conjunctive and disjunctive normal forms are modelled similarly. Judgements are used to show that modified conjunctive (normal) and disjunctive (normal) formulae are quantified formulae (`qfrmla?`).

The concepts of an isolated and minimal isolated formula are modelled as recursive predicates. Rather than defining these predicates over elements of type (`qfrmla?`), which would generate a TCC each time the predicates were used, the predicates are defined over elements of type `frmla` but will return false if they are not also of type (`qfrmla?`).

```
isolated?(B:frmla): RECURSIVE bool =
  IF qfrmla?(B) THEN
    IF atomic?(B) THEN
      TRUE
    ELSIF conj?(B) OR disj?(B) THEN
      FORALL (n:below[args(B)`length]):
        isolated?(arg(B,n))
    ELSE
      frmla?(arg(B)) AND
      contains_only?(name(B), get_dom(arg(B)))
    ENDIF
  ELSE
    FALSE
  ENDIF
  MEASURE deg(B)

min_isolated?(B:frmla): RECURSIVE bool =
  IF qfrmla?(B) THEN
    IF atomic?(B) THEN
      TRUE
```

```
    ELSIF conj?(B) OR disj?(B) THEN
      FORALL (n:below[args(B)`length]):
        min_isolated?(arg(B,n))
    ELSIF A?(B) THEN
      disjunctive_f?(arg(B)) AND
      contains_only?(name(B), get_dom(arg(B)))
    ELSE
      conjunctive_f?(arg(B)) AND
      contains_only?(name(B), get_dom(arg(B)))
    ENDIF
  ELSE
    FALSE
  ENDIF
  MEASURE deg(B)
```

Given these definitions it is not immediately recognised that (minimal) isolated is a subtype of quantified formulae (`qfrmla?`). Judgements are used to show this.

6.6.2 Formula Manipulation

Two recursive functions are defined that take a name and a finite sequence of formulae and returns a finite subsequence containing only those formulae that contain the name in their domain or a finite subsequence containing only those formulae that do not contain the name.

The rules of Table 5.1 are modelled as functions that take a quantified formula and return a formula. For instance, the following models rules (1.a) and (1.b) from the table in a single function.

```
rule1_alt(B:(A?)): frmla =
  IF contains?(name(B),get_dom(arg(B))) THEN
    IF nonempty?(get_pred(name(B),get_dom(arg(B)))) THEN
      B
```

```
    ELSE
      T
    ENDIF
  ELSE arg(B)
  ENDIF
```

The recursive function `flatten` takes a `(qfrmla?)` and returns a `(flat?)` formula. It is proven that the degree of a flattened formula is less than or equal to the degree of the original formula and if the original formula is not a flat formula then the degree of the flattened formula is strictly less then the degree of the original.

The recursive functions `mcn_aux` and `mdn_aux` are defined as auxiliary function for converting formulae into conjunctive or disjunctive normal form (CNF or DNF). The function `mcn_aux` takes an isolated formula `(isolated?)` and returns either a `(basic?)`, `(disjunctive?)` or `(conj_norm?)` formula. The function `mdn_aux` returns a `(basic?)`, `(conjunctive?)` or `(disj_norm?)`). These functions are required as not all formulae can be converted into the required form without adding extra components; for instance, $B \vee C$ can only be converted into CNF by taking its conjunction with true, i.e. $\text{true} \wedge (B \vee C)$. These extra components should only be added after it has been determined that the formula cannot be put into CNF without them.

The conversion of a conjunction B into CNF is relatively simple to define in PVS. Converting the subformulae of B into CNF converts B into such a form. The conversion of disjunction B into CNF is not such a simple matter. If its subformulae contain any conjunctions then, given that B is flat, at least one of its arguments must be a conjunction. This can be distributed across the disjunction converting it into a conjunction B'. This can then be converted into CNF by converting the subformulae of B'. If the formula is not flat then, although it contains a conjunction within its subformulae, none of the arguments of B are necessarily a conjunction; for instance, $C \vee (D \vee (F \wedge G))$. The formula B must be flattened and then converted into CNF.

```
mcn_aux(B:(isolated?)):
 RECURSIVE {B:(isolated?)|(basic?(B) OR
```

```
                disjunctive?(B) OR conj_norm?(B))} =
  IF basic?(B) OR conj_norm?(B) OR disjunctive?(B) THEN
    B
  ELSIF conj?(B) THEN
      conj((#length:=args(B)`length,
      seq:=LAMBDA (n:below[args(B)`length]):
      mcn_aux(arg(B,n))#))
  ELSE
    IF flat?(B) THEN
    LET
      m=choose({n:below[args(B)`length]|
      conj?(arg(B,n))}),
      ml=args(arg(B,m))`length
    IN
    conj((#length:=ml,
    seq:=LAMBDA (n:below[ml]):
     mcn_aux(
        disj((#length:=args(B)`length,
        seq:=LAMBDA (p:below[args(B)`length]):
        IF p=args(B)`length-1 THEN arg(arg(B,m),n)
        ELSE
          remove_nth(args(B),m)`seq(p)
        ENDIF#)))#))
     ELSE mcn_aux(flatten(B))
     ENDIF
  ENDIF
  MEASURE deg(B)
```

The functions `make_conj_norm` and `make_disj_norm` are defined to convert formulae into conjunctive or disjunctive normal form and make use of the auxiliary functions above, adding extra components where necessary to complete the conversion. Judgements are used to show that the result of applying `make_conj_norm` (`make_disj_norm`) to an isolated formula is a formula in conjunctive (disjunctive) normal form.

```
make_conj_norm(B:(isolated?)): (isolated?) =
```

```
  IF basic?(B) THEN
    conj((#length:=2,
    seq:= LAMBDA (n:below[2]):
    IF n=1 THEN T(name(B))
    ELSE B ENDIF #))
  ELSIF disjunctive?(B) THEN
    conj((#length:=2,
    seq:= LAMBDA (n:below[2]):
    IF n=1 THEN T
    ELSE B ENDIF #))
  ELSE mcn_aux(B)
  ENDIF
```

To perform quantifier isolation, the transfer of quantifiers over conjunctions and disjunctions is required. Two functions are defined that apply the appropriate rules for this transfer.

```
move_A_in(B:{u:(A?)| basic?(arg(u)) OR
disjunctive?(arg(u)) OR conj_norm?(arg(u))}):
(min_isolated?) =
  IF basic?(arg(B)) THEN
    rule1_alt(B)
  ELSIF conj?(arg(B)) THEN
    rule3(B)
  ELSE
    rule5(B)
  ENDIF

move_E_in(B:{u:(A?)| basic?(arg(u)) OR
conjunctive?(arg(u)) OR disj_norm?(arg(u))}):
(min_isolated?) =
  IF basic?(arg(B)) THEN
    rule2_alt(B)
  ELSIF disj?(arg(B)) THEN
    rule4(B)
  ELSE
```

```
    rule6(B)
  ENDIF
```

The recursive function qi (quantifier isolation) takes a (qfrmla?) and returns a formula in minimal isolated form. This function is more complex than the previous recursive functions as it may require nested applications, i.e, qi(A(name(B),qi(arg(B)))). The simple measure of the degree of a formula that has previously been used when defining recursive functions over formulae is no longer sufficient to guarantee termination. A new measure deg_qi is introduced. This measure is very similar to deg only the new degree of minimal isolated formulae is 0.

```
qi(B:(qfrmla?)): RECURSIVE (min_isolated?) =
  IF min_isolated?(B) THEN B
  ELSIF A?(B) THEN
    IF min_isolated?(arg(B)) THEN
      move_A_in(A(name(B),flatten(mcn_aux(arg(B)))))
    ELSE
      qi(A(name(B),qi(arg(B))))
    ENDIF
  ELSIF E?(B) THEN
    IF min_isolated?(arg(B)) THEN
      move_E_in(E(name(B),flatten(mdn_aux(arg(B)))))
    ELSE
      qi(E(name(B),qi(arg(B))))
    ENDIF
  ELSIF conj?(B) THEN
    conj((#length:=args(B)`length,
    seq:= LAMBDA (n:below[args(B)`length]):
    qi(arg(B,n))#))
  ELSE
    disj((#length:=args(B)`length,
    seq:= LAMBDA (n:below[args(B)`length]):
    qi(arg(B,n))#))
  ENDIF
```

```
MEASURE deg_qi(B)
```

6.7 Decision Procedure for Functions of One Variable

Sections 6.5 to 6.6.2 have dealt purely with the modelling of syntactic aspects of formulae of $\mathcal{L}_1$. Since the decision procedure is concerned with semantic properties of formulae these must be modelled before the decision procedure.

Formulae are interpreted with respect to a variable. When interpreting a formula, a symbol table is constructed that associates a name and type predicate with a variable. This table is modelled using the finite sequence type that is inbuilt in PVS. A symbol table is defined for a specific formula and contains a single entry for each name in the domain of the formula. The type predicate associated with the name is a subset of (or is equal to) the domain associated with it within the formula.

```
symb_table(A:frmla): TYPE =
  {u:finseq[[P:pred[real],name,(P)]]|
   u`length>=get_dom(A)`length AND
   FORALL (n:below[get_dom(A)`length]):
   (EXISTS (m:below[u`length]):
     get_dom(A)`seq(n)`2 = u`seq(m)`2)
   AND
   (FORALL (m:below[u`length]):
     get_dom(A)`seq(n)`2 = u`seq(m)`2
       IMPLIES
         FORALL (x:(u`seq(m)`1)):
           (get_dom(A)`seq(n)`1)(x))}
```

When interpreting a formula, those variables that are free (not quantified) are interpreted as implicitly universally quantified. This is represented in the function `interpret`, which takes a formula. The sequence `x` is universally quantified and contains the same number of variables as there are free variables in `A`. The symbol table is initialised using these vari-

ables, ensuring that each variable is associated with the appropriate name and type predicate. An auxiliary function is called to interpret the formula with respect to this symbol table.

```
interpret(A:frmla): bool =
  FORALL (x:vars(get_dom(A))):
    interpret_aux(A,
     (#length:=x`length,
      seq:= LAMBDA (n:below[x`length]):
       (get_dom(A)`seq(n)`1,
          get_dom(A)`seq(n)`2, x`seq(n))#))
```

The auxiliary function interprets formulae as would be expected; for instance a formula of type `(conj?)` is interpreted as the conjunction of the interpretation of its subformulae. The symbol table remains the same for all arguments of the conjunction. The symbol table is only updated when a quantifier is encountered. Since quantifying a variable removes it from the domain of the resulting formula, this process must essentially be reversed when interpreting it. If the formula is of type `(E?)` then the formula is interpreted as the existential quantification of the variable y over the interpretation of the subformula. The variable is added along with the appropriate name and type predicate to the symbol table. If the table already contains an entry for the given name then the entry is overwritten. This causes a name to be connected with the most recent binding.

```
interpret_aux(B:frmla,x:symb_table(B)):
  RECURSIVE bool =
  IF qfrmla?(B) THEN
    IF atomic?(B) THEN
      interpret_atomic(B,get_var(name(B),x))
    ELSIF conj?(B) THEN
       IF args(B)`length=2 THEN
         interpret_aux(arg(B,0),x) AND
         interpret_aux(arg(B,1),x)
       ELSE
         interpret_aux(arg(B,0),x) AND
```

```
        interpret_aux(
          conj(args(B)^(1,args(B)`length-1)),x)
      ENDIF
    ELSIF disj?(B) THEN
      IF args(B)`length=2 THEN
        interpret_aux(arg(B,0),x) OR
        interpret_aux(arg(B,1),x)
      ELSE
        interpret_aux(arg(B,0),x) OR
        interpret_aux(
          disj(args(B)^(1,args(B)`length-1)),x)
      ENDIF
    ELSIF E?(B) THEN
      EXISTS (y:(get_pred(name(B),get_dom(arg(B))))):
        interpret_aux(arg(B), x o
        (#length:=1, seq:= LAMBDA (n:below[1]):
         (get_pred(name(B),get_dom(arg(B))),
          name(B),y)#))
    ELSE
      FORALL (y:(get_pred(name(B),get_dom(arg(B))))):
        interpret_aux(arg(B),
          insert(get_pred(name(B),get_dom(arg(B))),
            name(B),y,x))
      ENDIF
    ELSE
      FALSE
    ENDIF
  MEASURE deg(B)
```

The interpretation of an atomic formula is based on its component parts. It is interpreted as an (in)equality between a function and 0 for a given variable x.

```
interpret_atomic(A:(atomic?),x:(dom(A))): bool =
  IF ineq(A) = lt THEN
    func(A)(x) < 0
```

```
  ELSIF ineq(A) = le THEN
    func(A)(x) <= 0
  ELSIF ineq(A) = eq THEN
    func(A)(x) = 0
  ELSIF ineq(A) = ge THEN
    func(A)(x) >= 0
  ELSIF ineq(A) = gt THEN
    func(A)(x) > 0
  ELSE
    func(A)(x) /= 0
  ENDIF
```

Given this interpretation of formulae the decision procedure, described in Section 5.7, is modelled in four separate predicates.

The theory in which the first two predicates are defined is parameterised with two real variables a and b, which represent the maximum and minimum elements of a closed set. The function f is defined as finitely inflective over this interval. The predicate dp_single_aux recursively applies a predicate to determine the positivity/negativity (as appropriate) of f in some of the intervals in [a, b] in which f is either convex or concave.

```
dp_single_aux(f:finitely_inflective[closed[real,a,b]],
  m:real, c:real, n:below[fi_dom(f)`length],s:inq):
  RECURSIVE bool =
  LET lb = glb(fi_dom(f)`seq(n)),
      ub = lub(fi_dom(f)`seq(n)) IN
  IF n = 0 THEN
    IF s=lt THEN
      curve_lt_line[closed[real,a,b],lb,ub](f,m,c)
    ELSIF s=le THEN
      ⋮
    ELSE
      curve_neq[closed[real,a,b],lb,ub](f,m,c)
    ENDIF
```

```
    ELSE
      (IF s=lt THEN
        curve_lt_line[closed[real,a,b],lb,ub](f,m,c)
       ELSIF s=le THEN
         ⋮
       ELSE
         curve_neq[closed[real,a,b],lb,ub]
           (f,m,c)
       ENDIF)
      AND
       dp_single_aux(f,m,c,n-1,s)
  ENDIF
MEASURE n;
```

The function f is finitely inflective. By definition (Section 6.4), this means that the domain of f can be split into a unique sequence fi_dom(f) of disjoint interval in each of which the function is either convex or concave. The predicate dp_single_aux takes the natural number n as an argument and determines whether f is above/below/etc the line m*x+c in all intervals in fi_dom(f) whose indices are less than or equal to n.

The predicate dp_single simply calls dp_single_aux the with n equal to the maximum index of elements in fi_dom(f) to ensure that the function is examined over all of [a, b].

```
dp_single(f:finitely_inflective[closed[real,a,b]],
  m:real,c:real,s:inq):
  bool =
    dp_single_aux(f,m,c,fi_dom(f)`length-1,s)
```

The predicate dp_q_atomic takes an element of (q_atomic?) and extracts the quantifier, and the components from the atomic formula. If the formula is existentially quantified it is converted into an equivalent universally quantified formula. The predicate dp_single is applied to the components of the atomic formula.

```
dp_q_atomic(B:(q_atomic?)): bool =
```

```
  IF A?(B) THEN
    dp_single[lub(dom(arg(B))),glb(dom(arg(B)))]
     (func(arg(B)),0,0,ineq(arg(B)))
  ELSE
    NOT dp_single[lub(dom(arg(B))),glb(dom(arg(B)))]
     (func(arg(B)),0,0,neg(ineq(arg(B))))
  ENDIF
```

The predicate dp represents the complete decision procedure of Section 5.7. It takes an element of (dp_frmla?) and if the formula is an element of (q_atomic?) the predicate dp_q_atomic is applied. If the formula is a conjunction then dp is applied recursively to the components of the formula and the conjunction of the results is the taken. If the formula is a disjunction then dp is applied recursively to the components of the formula and the disjunction of the results is taken.

```
dp(B:(dp_frmla?)): RECURSIVE bool =
  IF conj?(B) THEN
    IF args(B)`length=2 THEN
      dp(arg(B,0)) AND dp(arg(B,1))
    ELSE
      dp(arg(B,0)) AND
      dp(conj(args(B)^(1,args(B)`length-1)))
    ENDIF
  ELSIF disj?(B) THEN
    IF args(B)`length=2 THEN
      dp(arg(B,0)) OR dp(arg(B,1))
    ELSE
      dp(arg(B,0)) OR
      dp(disj(args(B)^(1,args(B)`length-1)))
    ENDIF
  ELSE
   dp_q_atomic(B)
 ENDIF
 MEASURE deg(B)
```

Theorem 6.1 *The decision procedure terminates.*

PROOF: The termination of the procedure relies on two facts: (1) any formula B to which the procedure applies is in minimal isolated form and is thus composed of a finite or countably infinite number of (quantified) atomic formulae B_i, (2) every function f_i in B is finitely inflective, which means that the domain of f_i can be split into a finite number of intervals over which f_i is convex or concave. The procedure determines the truth of each B_i in each sub–domain of f_i in which f_i is convex or concave. It then composes the results. Since there is a finite or countably infinite number of formulae B_i and each f_i is finitely inflective the procedure will eventually determine the truth of every subformula B_i in every sub–domain of f_i. ■

The proof of termination of the decision procedure in PVS requires the proof of termination of both `dp_single_aux` and `dp`. As with all recursive functions, PVS requires measures, which decrease with each successive call of `dp_single_aux` and `dp`, to guarantee termination. The measure of `dp_single_aux` is a natural number `n`, which is explicitly decreased on each recursive call. This leads to a trivial termination condition, that is, `n-1<n`. The measure of `dp` is the degree of the formula `B`. The termination conditions for this function are more complex. Several TCCs representing the termination argument are generated, one for each recursive call of `dp`. Each TCC ensures that the measure decreases for a single recursive call. For instance, the termination TCCs generated for the case when `B` is a conjunction with more than two arguments[3] are:

```
dp_TCC9: OBLIGATION
  FORALL (B: (dp_frmla?)):
    conj?(B) AND NOT args(B)`length = 2 IMPLIES
      deg(arg(B, 0)) < deg(B);

dp_TCC13: OBLIGATION
  FORALL (B: (dp_frmla?)),
```

[3]The case when the input is a conjunction with more than two arguments corresponds to the sixth and seventh lines of `dp`, that is, `dp(arg(B,0)) AND dp(conj(args(B)^(1,args(B)`length-1)))`.

```
(v: [{z: (dp_frmla?) | deg(z) < deg(B)} -> bool]):
  conj?(B) AND NOT args(B)`length = 2 AND v(arg(B, 0))
    IMPLIES
      deg(
        conj(^[frmla](args(B), (1, args(B)`length - 1))))
      < deg(B);
```

The proof of these TCCs, and thus of the termination of the decision procedure, rely on the fact that the degree of any subformula C of a conjunctive, disjunctive or quantified formula B has a lower degree than B itself. Showing that the degree of C is less than the degree of B is relatively simple; involving the expansion of the definition of `deg`. Further TCC are generated to ensure that these subformulae have the type `(dp_frmla?)`. These TCCs are also relatively simple to prove and involve the expansion of the definition of `(dp_frmla?)`. In total, 30 TCCs are generated for this recursive function, 8 of which are concerned with termination.

Theorem 6.2 *For all minimal isolated formulae B in which the scope of any quantifier is a literal formula in the language $\mathcal{L}_1$ and in which all functions are finitely inflective, the procedure correctly determines the truth of B.*

PROOF: The truth of any conjunction $B \wedge C$ can be determined by determining the truth of B and C independently and then taking the conjunction of the results. The truth of any disjunction $B \vee C$ can be determined by determining the truth of B and C independently and then taking the disjunction of the results. The truth of any inequality $f \sim 0$ can be determined by determining the truth of the inequality in sub–domains D_i of f, where $D_0 \cup ... \cup D_n = \text{domain}(f)$, and taking the conjunction of the results. The set of conditions given in Section 5.3 are correct. The decision procedure uses the above facts to decomposes an input formula B, determine the truth of the sub–formulae and then correctly build truth B. ■

In PVS, the decision procedure is shown to be correct by proving that the result gained by applying the decision procedure to a formula `B` is equivalent to the interpretation of `B`.

```
dp_atomic_lem: LEMMA
  FORALL (B:(q_frmla?)):
    interpret(B) IFF dp_q_atomic(B)

dp_lem: LEMMA
  FORALL (B:(dp_frmla?)):
    interpret(B) IFF dp(B)
```

The proof of the first lemma requires induction to be performed over the number of intervals n over which the function `func(B)` is convex or concave. The base case ($n = 0$) is proven by contradiction. Since `func(B)` is a finitely inflective function, if there are no intervals in `dom(B)` over which it is convex or concave then `dom(B)` must be empty; however, by definition, the domain of any `q_frmla?` is non–empty. In the inductive step, the fact that `dom(B)` can be represented as a sequence S of intervals over which `func(B)` is convex or concave is exploited. `dom(B)` is split into two intervals I_1 and I_2, where I_1 is equal to the first $n - 1$ intervals in S, and I_2 is equal to the final interval in S. It must then be shown that `interpret(B) IFF dp_q_atomic(B)` in both intervals. In I_1 this is done using the inductive step. In I_2, a case split is performed on the type of the inequality `inq(B)` (for instance `inq(B)=lt` or `inq(B)=le`). In each of the cases, the lemma representing the appropriate condition (see Section 6.3) is applied. The proof of the second lemma requires induction over the natural number n, which is greater than or equal to the degree of `B`[4]. The base case is proved by contradiction. The degree of a formula is 0 iff it is an atomic formula; however, by definition, a `dp_frmla?` cannot be atomic. If `B` is a `q_atomic?` formula then the previous lemma is applied. If `B` is a conjunction or disjunction then use the induction step on the components of the conjunction or disjunction. During each of these proofs many TCCs are generated to ensure that intervals are nontrivial convex sets and that formulae are of type `(dp_frmla?)`.

[4]Inducting over the degree of the formula would not be useful in producing a proof of the lemma. The degree of the component parts of a conjunction or disjunction B is not one less than the degree of B, thus the induction step could not be used in the proof.

Chapter 7

Automated Formal and Symbolic Nichols Plot Analysis

The current process for the Nichols plot analysis of control and dynamical systems involves the plotting and visual analysis of suites of Nichols plots. Commonly used control engineering tools (Section 2.3), such as the Mathwork's MATLAB and Simulink [38], produce Nichols plots (without showing exclusion or desired regions) for a system given only its transfer function, thus allowing the engineer to avoid dealing with the underlying complex mathematics involved in the plotting process. When automating the process of formal symbolic Nichols plot analysis it is desirable that this complex mathematics remains hidden from the engineer.

In this chapter a tool, NRV (Nichols plot Requirements Verifier), for the automation of formal and symbolic Nichols plot analysis is described. NRV was developed to allow the replacement of the informal visual analysis with formal symbolic analysis of Nichols plots. The tool is designed to fit into the development process and require no extra work to be performed by the control engineer. The tool requires the user to provide the transfer function representing the system and the Nichols plot requirements in terms of the boundary lines for the exclusion region. The tool applies the decision procedure of Section 5.7 and produces a formal proof that the system meets its requirements, a counter example to show

that the system does not meet its requirements, or, if it can do neither of these, highlights the particular areas for which the process failed for closer examination by the control engineer.

In Section 7.1 some of the issues associated with the automation of formal symbolic Nichols plot analysis are discussed in terms of the required functionality of any tool in which the process is to be automated. The Maple–PVS–QEPCAD system, which meets these requirements, is introduced. An overview of NRV is given in Section 7.2 and a more detail description of the division of labour between *Maple* and PVS is given in 7.3. It is important to note that *Maple* uses radians rather than degrees in trigonometric calculations and in the preceding chapters the phase–shift of a control system will be measured in radians.

7.1 Requirements for Formal and Symbolic Nichols Plot Analysis

In order to use the decision procedure of Section 5.7 in formal symbolic Nichols plot analysis one must be able to reliably calculate the points of inflection of a reasonable function, the convexity of the corresponding curve and the sign of the curve at given points. This requires not only powerful symbolic manipulation whose results are assured but also numerical calculations that are correct to within certain guaranteed tolerances.

As discussed in Chapter 4, computer algebra (CA) systems provide a powerful method for symbolic manipulation and analysis of mathematical formulae and are ideal for performing the transformations and calculations required for the application of decision procedure to Nichols plot analysis. However, they can not always guarantee correct results, for instance, they may ignore assumptions and side conditions or produce floating point errors during numerical calculation. Formal theorem provers provide powerful methods for formal analysis but lack the ability to perform symbolic manipulation or numerical calculations efficiently. A combination of these two types of system can provide the desirable properties of both.

The Maple–PVS tool [2] (Section 4.4) provides a link between the CA system *Maple* [94]

and the theorem prover PVS [103], allowing PVS processes to be spawned by a *Maple* process. This system allows the calculations performed by *Maple* to be formally verified by PVS, providing efficient and reliable mathematics. The onus is on *Maple* to formulate the lemmas to be proved and pass them to PVS along with the proof steps to be taken, usually in the form of high level PVS strategies.

During formal symbolic Nichols plot analysis PVS is required to prove that rationally differentiable functions are convex or concave in intervals, that is to prove lemmas of the form $\forall x \in D.\, f(x) \sim 0$, where D is some closed convex set, f is some rational function and $\sim$ is one of $\geq$ or $<$. One efficient method for proving such lemmas is quantifier elimination (Section 4.3).

The QEPCAD tool [65] provides automation of quantifier elimination using the highly efficient method of (partial) cylindrical algebraic decomposition (Section 4.3). As input QEPCAD takes a prenex normal formula ϕ constructed from conjunctions and disjunctions of inequalities between r-variate polynomials and returns an equivalent quantifier free formula in only the free variables of ϕ. If there are no free variables in ϕ then the only possible output is either *true* or *false*.

The QEPCAD–PVS [116] tool provides a link between QEPCAD and PVS, which allows quantifier elimination routines to be accessed by PVS via foreign function calls. Rather than two separate processes running, as is the case with the Maple–PVS system, the QEPCAD–PVS system only requires a single PVS process, which loads a shared object file that provides two PVS strategies. The first of these strategies converts formulae into prenex normal form. The second applies quantifier elimination to a formula. PVS considers the results of calling these strategies to be reliable. This system allows PVS to use powerful and efficient quantifier elimination within its proofs.

The Maple–PVS system has been extended to allow the automatic loading of the QEPCAD–PVS shared object file into the PVS child process (see Figure 7.1). This, in effect, allows *Maple* to invoke QEPCAD routines as part of PVS proofs.

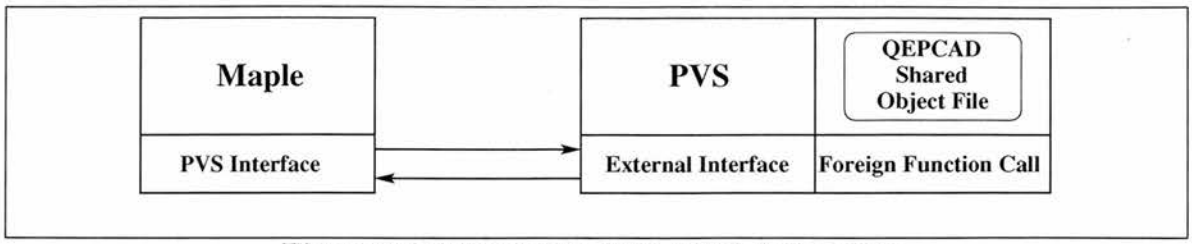


Figure 7.1: The Maple–PVS–QEPCAD system.

7.2 Overview of the Nichols Plot Requirements Verifier

The Nichols plot Requirements Verifier (NRV) is implemented in the Maple–QEPCAD–PVS system. *Maple* is used to perform the bulk of the calculations as well as providing the user interface. PVS is used to verify the results of *Maple*'s calculations, using, when necessary, QEPCAD routines.

The minimum amount of data required to perform Nichols plot analysis is a representation of the system to be analysed, usually in the form of a transfer function, and some representation of the exclusion/desired region. The decision procedure of Section 5.7 can not be applied directly to this input, thus it is necessary to perform some pre–processing to correctly formulate the problem. This pre–processing requires both symbolic manipulation of the input and numerical calculation and is a task ideally suited to *Maple*.

Maple provides the front end of NRV via *Maple Maplets*. These provide a Java applet–like graphical user interface into which the input is entered and any results or error messages are displayed. A simple type check mechanism ensures that the input is of the correct type and format. *Maple* processes the input to form the appropriate sentences for use in the decision procedure and invokes PVS, which in turn may invoke QEPCAD, to perform the required verification. If PVS fails to provide proofs then attempts to find counter examples are made and the process continues. *Maple* records appropriate messages depending on the results of the PVS calls. Once the process is complete *Maple* displays the results of the decision procedure, the messages recorded as a result of PVS calls, and a plot showing the bounding lines for the specified region along with the plot of the curve representing the system. This allows the analyst to view the Nichols plot as traditionally used tools allow

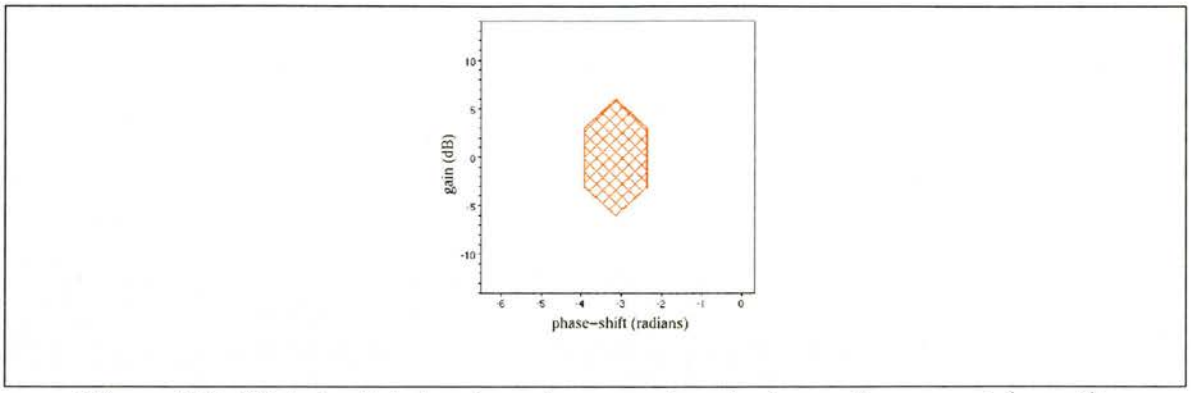


Figure 7.2: Nichols plot showing a hexagonal exclusion region around $(-\pi, 0)$.

but also provides either assurances that the system meets or does not meet its requirements, or suggestions of areas that require closer examination.

The input to NRV is, where possible, similar in format to input required by traditional tools. A transfer function is entered as two lists of integers or variables; the first represents the coefficients of the polynomial numerator of a transfer function, the second represents the coefficients of the polynomial denominator. Each list is ordered such that the n-th element in it represents the coefficient of s^{l-n} where l is the length of the list, for example $[a, b, c]$ represents $as^2 + bs + c$. The permissible range for any variable coefficient of the transfer function is entered in standard *Maple* format as *name = range*, for example $c = 0..10$. The exclusion region is entered as a list B of pairs each of which has as components an interval $[a_i, b_i]$, and a list L_i of pairs of lines l_{ij} and inequality signs $\sim_{ij}$; for example,

$$\begin{aligned} B = \; & [(-\tfrac{5}{4}\pi .. -\pi, [(y + \tfrac{12}{\pi}x + 18, <), (y - \tfrac{12}{\pi}x - 18, >)]), \\ & (-\pi .. -\tfrac{3}{4}\pi, [(y + \tfrac{12}{\pi}x + 6, <), (y - \tfrac{12}{\pi}x - 6, >)]), \\ & (-3..3, [(x + \tfrac{3}{4}\pi, >), (x + \tfrac{5}{4}\pi, <)])] \end{aligned}$$

Each element of L_i represents a constraint on the range of the parametric function; in the above example $(y + \frac{12}{\pi}x + 18, <)$ represents the constraint $y < -\frac{12}{\pi}x - 18$, and $(x + \frac{3}{4}\pi, >)$ represents the constraint $x > -\frac{3}{4}\pi$. Each tuple in B represents the disjunction of the constraints in L_i in the domain $[a_i, b_i]$. The list B represents the conjunction of the constraints represented by the tuples. The above example represents the constraint that the plot for a system does not enter a hexagonal region about the point $(-\pi, 0)$ (see Figure 7.2).

Given input of the form described above the steps performed by NRV can be separated into

a number of logical tasks.

1. Calculate the transfer function from the lists of coefficients along with the formulae for the gain and phase–shift of the system. Calculate the first and second derivatives of the parametric equation representing the system with respect to x.

2. Safely converting the intervals $[a_i,\ b_i]$ from being in terms of x to being in terms of ω, i.e, determine all ω_k, ω_j such that $[a_i,\ b_i] \subseteq [X(\omega_k),\ X(\omega_j)]$.

3. Calculate the points of inflection of the curve representing the system including any points at which it becomes vertical in the intervals and split the intervals at these points.

4. Based on the convexity of the curve in each of the intervals, formulate the appropriate problem to be solved, i.e, of the form $\lambda\omega \in [a,\ b].\ f_{ij}(\omega) \sim_{ij} 0$, and apply the correct case from the conditions of Section 5.3.

5. Determine whether the system meets its Nichols plot requirements by building up the conjunctions or disjunctions, as appropriate, of the truth of the subformulae.

6. If it can not be shown that the system meets its Nichols plot requirements then attempt to find a counter example.

7. Produce a plot of the lines and the parametric equation representing the system.

Each of these logical tasks corresponds to a group of *Maple* procedures designed specifically for the task, which can be further separated into those procedures that perform initial calculations, those that perform appropriate adjustments where necessary to ensure the results are 'safe', and those that confirm, by invoking PVS, that *Maple*'s calculations are correct.

The procedure for Nichols plot requirements verification depends on the type of lines that bound the exclusion region. If the lines bounding the exclusion region are not vertical then in order to show that the system meets its requirements it must be shown that in

each interval of interest the gain of the system lies above or below the bounding lines as appropriate. If any of the lines bounding the exclusion region are vertical then it must also be shown that the phase–shift of the system lies above or below these lines as appropriate.

7.3 Interactions Between *Maple*, PVS and QEPCAD

The top level *Maple* procedure in the Nichols plot requirements verifier accepts a constant δ by which adjustments are to be made, a transfer function and a representation of an exclusion region as a list B of pairs each of which has as components an interval $[a_i,\ b_i]$, and a list L_i of pairs of lines l_{ij} and inequality signs $\sim_{ij}$. The gain, phase–shift and derivatives are calculated and for each element in the list B a *Maple* procedure is called to determine whether the gain or phase–shift lies above or below the bounding lines as appropriate.

The following describes the steps taken by the prototype tool when considering a system $F(s)$ and an exclusion/desired region, described in terms of a number of intervals $[a_i,\ b_i]$, in which there are a number of disjoint regions (e.g, of the form $l_{ij}(x) < y < l_{ik}(x)$) described in terms of the lines $l_{in}(x)$ bounding them.

1. The user supplied input is type checked and if it is not of the correct format an error message is produced and the prototype tool halts.

2. *Maple* calculates the equations for gain $y = Y(\omega)$ and phase–shift $x = X(\omega)$ of the system $F(j\omega)$.

3. *Maple* functions are used to calculate, rewrite and simplify the first and second derivatives (expressed as functions of ω) of y (with respect to x) given the set of parametric equations $y = Y(\omega)$ and $x = X(\omega)$.

4. To analyse the Nichols plot requirements of a system the intervals of interest must first be calculated in terms of ω rather than x[1]. Ideally, one wishes to find all solutions

[1]The intervals may be in terms of y rather than x. For simplicity the procedure described here refers only to intervals in terms of x but it applies equally to intervals in terms of y

ω_{ai} and ω_{bi} such that $[X(\omega_{ai}), X(\omega_{bi})] = [a_i, b_i]$; however, since these solutions are likely to be calculated using floating point arithmetic it is not likely that the intervals will correspond directly. In practice, the best solution is to find 'safe' conversions, that is, the values ω_{ai} and ω_{bi} such that $[X(\omega_{ai}), X(\omega_{bi})] \supseteq [a_i, b_i]$. To calculate these intervals *Maple* uses inbuilt functions to solve equalities and evaluate floating point numbers to find all solutions ω_{ik} to $a_i = X(\omega)$ or $b_i = X(\omega)$. All non–real and non–positive solutions are discarded and the remaining results are sorted into ascending order. *Maple* calculates small intervals $[\omega_{ik} - \delta,\ \omega_{i(k+1)} + \delta]$ around each of its solutions to compensate for floating point error. The intervals in ω that correspond to the intervals in x are then calculated. This is done by determining whether $X(0)$ lies within or outwith $[a_i, b_i]$. If $X(0)$ is not in $[a_i, b_i]$ then the intervals $[\omega_{2ik} - \delta,\ \omega_{i(2k+1)} + \delta]$ correspond to $[a_i, b_i]$ (see Figure 7.3), otherwise the intervals

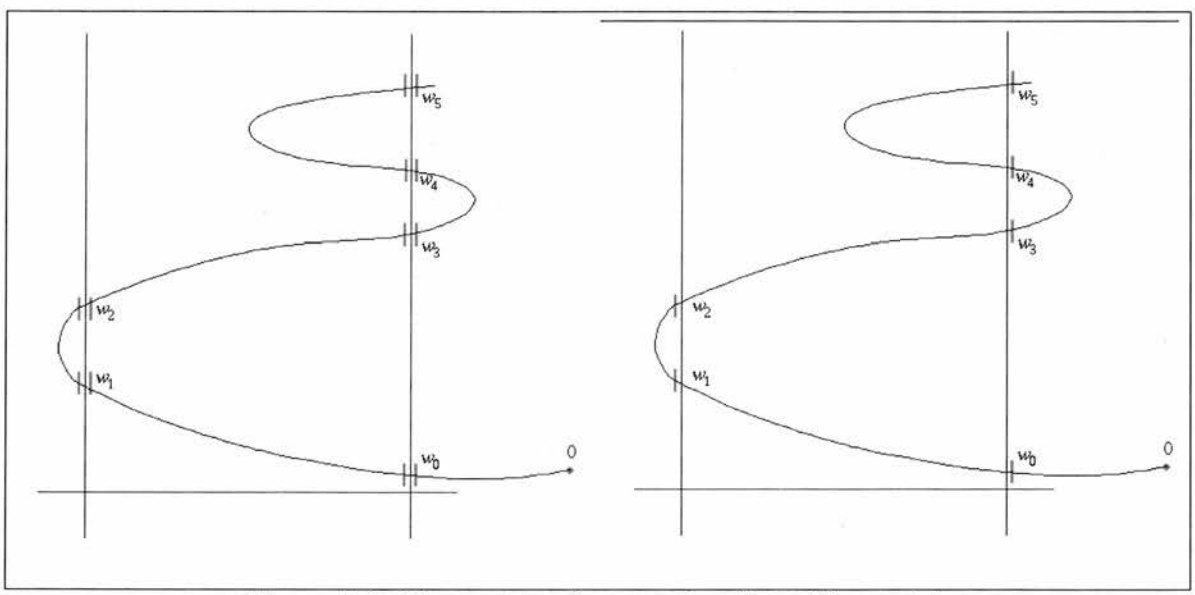


Figure 7.3: Illustration of conversion from X to ω (a).

$[0, \omega_{i0} + \delta]$ and $[\omega_{2ik+1} - \delta,\ \omega_{i(2k+2)} + \delta]$ (see Figure 7.4). To ensure that these intervals are 'safe', i.e. that $[X(\omega_{ai}), X(\omega_{bi})] \supseteq [a_i, b_i]$, NRV must show that for all ω outside the intervals $X(\omega)$ is outside $[a, b]$. This problem can be formulated as a conjunction of inequalities and can be solved by essentially reapplying NRV to it. The problem does not involve parametric equations and thus does not require step 4. Owing to this, there will not be any uncontrolled recursion. If NRV fails to provide any of the

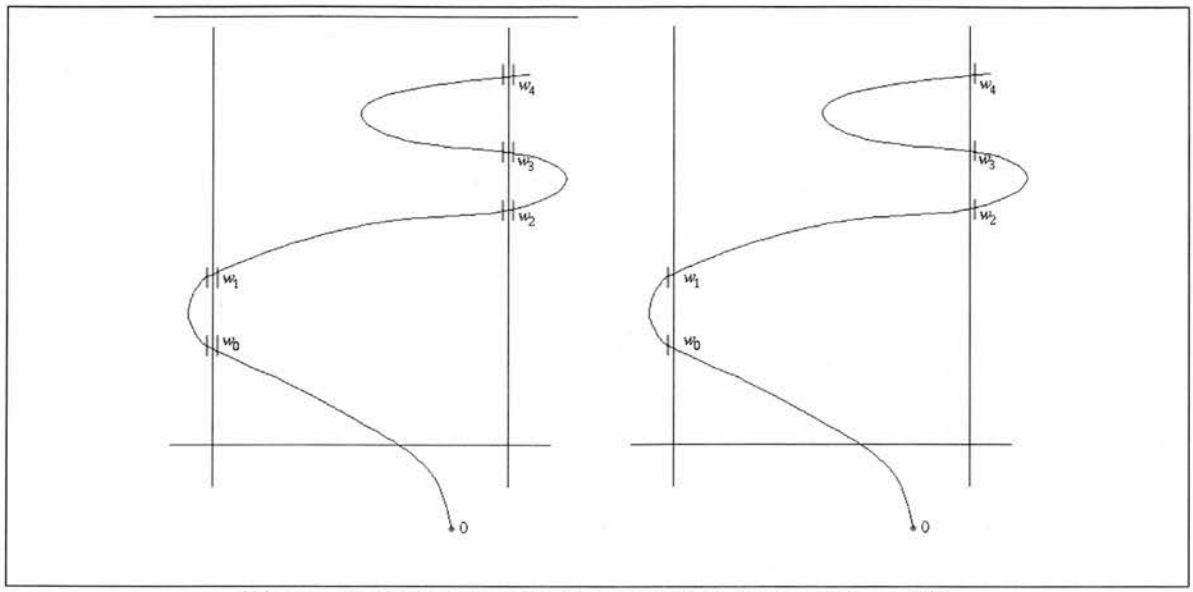


Figure 7.4: Illustration of conversion from X to ω (b).

required proofs, an appropriate error message is recorded and the process continues.

5. *Maple* calculates the points of inflection of the function represented by the set of parametric equations $x = X(\omega), y = Y(\omega)$, including any points at which it becomes vertical, in the intervals $[\omega_{ik} - \delta, \omega_{i(k+1)} + \delta]$. This is achieved using numerical methods to find points at which the second derivative of the function with respect to x is zero and as a consequence is subject to errors due to inexact arithmetic. In an attempt to avoid this problem *Maple* calculates small intervals $[p_{ikm} - \delta, p_{ikm} + \delta]$ in which these points should lie (referred to as *intervals of inflection*). PVS is called to confirm not only that these intervals contain true points of inflection rather than points of zero curvature between two regions both strictly convex or concave but also that for each l_{ij} the derivative of the parametric equations is not equal to the derivative of l_{ij} in the interval unless it is exactly at the point of inflection. This is a relatively difficult problem for PVS to solve; however, since the derivative and second derivative of the parametric equations are rational the problem is ideal for quantifier elimination. PVS uses the QEPCAD–PVS link to invoke the QEPCAD strategies to verify *Maple*'s results. If NRV fails to provide any of the required proofs an appropriate error message is recorded and the process continues.

6. It must be ensured that in those intervals between the intervals of inflection the

Nichols plot for the system is either convex or concave. *Maple* examines the sign of second derivative of the curve at a point within each interval. Based on the results of this examination PVS is used to determine whether the curve is convex or concave in each interval. Again, this is a relatively difficult problem for PVS to solve; however, since the derivative and second derivative of the curve are rational QEPCAD strategies can be invoked. If NRV fails to provide any of the required proofs an appropriate error message is recorded and the process continues. The intervals $[\omega_{ik} - \delta,\ \omega_{i(k+1)} + \delta]$ are split into $[\omega_{ik} - \delta,\ p_{ikm} - \delta]$ $[p_{ikm} - \delta,\ p_{ikm} + \delta]$ $[p_{ikm} + \delta,\ \omega_{i(k+1)} + \delta]$. over which the curve is either convex or concave, or is an interval of inflection.

7. *Maple* formulates the lemmas to be solved by PVS in the form $\lambda\omega \in [a_n,\ b_n].\ Y(\omega) - l_{ij}(X(\omega))(\omega)\ \sim_{ij}\ 0$, where $\sim_{i,j}$ is the inequality sign indicating whether the curve should lie above ($>$) or below ($<$) the line and $[a_n,\ b_n]$ are the intervals calculated in step 6. For intervals in which the curve is convex or concave, PVS is called by *Maple* to prove these lemmas; determining whether the desired case from the set of conditions holds. Owing to the nature of intervals of inflection, the maximum and minimum of $Y(\omega) - l_{ij}(X(\omega))$ must lie on the bounds of the interval. If it has already been proven that the curve lies on the correct side of the line in the intervals adjacent to an interval of inflection then it can be inferred that the curve lies on the correct side of the line in that interval. For this reason it is only necessary to explicitly determine whether the curve lies above or below the line in an interval of inflection if it has not been shown that the curve lies on the correct side of the line in the intervals adjacent to it. Whether the curve lies out with/within any given region in any interval is determined by the conjunction of the truth of the appropriate lemmas. *Maple* maintains a list of potential counter examples and if it can not be shown that the Nichols plot lies outwith the exclusion region in any given interval NRV uses *Maple* to select a likely counter example from its list and PVS to prove that the counter example holds. If NRV fails to provide any of the required proofs an appropriate error message is recorded and the process continues.

8. The truth of whether the system meets its Nichols plot requirements in each interval is built up from the disjunction of the truth values for the curve remaining out with-/within each of the regions within the interval. The truth of whether the system meets its Nichols plot requirements is built up from the conjunction of the truth values in each interval.

9. NRV produces an appropriate message indicating whether the Nichols requirements have been met along with details of any failure in proofs. A Nichols plot for the system, showing the exclusion region is displayed.

The NRV process is fully automated. Once a user has entered the transfer function of a control system and a set of requirements NRV requires no further intervention to produce its result. *Maple* procedures and high level PVS strategies are defined to automate the calculations and proofs required by NRV. During the NRV process, PVS uses custom built libraries (see Appendix C) containing lemmas concerning various functions important in control system analysis, such as arctan, natural logarithm and logarithm to the base ten. These libraries contain definitions of the natural logarithm and arctan as Taylor series, which allow bounds on the values of these functions for any given input to be defined and numerical calculations using these functions to be validated. In practice the Taylor series for the natural logarithm converges so slowly that it is of little use. For this reason, the natural logarithm and logarithm to the base ten are not reasoned about directly; rather, they are eliminated from inequalities using exponentiation. This allows the substantially more efficient Taylor series expansion of the exponential, as defined in the transcendentals library [54], to be used.

The PVS libraries used by NRV are designed to allow efficient proofs to be carried out in PVS. In many case, two alternative definitions are given for a function. The first definition represents the standard 'text book' definition, while the second is an equivalent definition that can be expanded more efficiently in PVS. For instance, exponentiation is defined in the prelude library in the standard manner,

```
expt(r:real, n:nat): RECURSIVE real =
```

```
    IF n = 0 THEN 1
    ELSE r * expt(r, n-1)
    ENDIF
  MEASURE n;
```

but is also defined as follows:

```
expt(x:real,n:nat): RECURSIVE real =
    IF n>=10 THEN
      x * (x * (x * (x * (x * (x * (x * (x * (x * (x *
      expt(x,n-10)))))))))
    ELSIF n>5 THEN
      x * (x * (x * (x * (x * expt(x,n-5)))))
    ELSIF n=5 THEN
      x * (x * (x * (x * x)))
    ELSIF n = 1 THEN
      x
    ELSIF n = 0 THEN
      1
    ELSE
      x * expt(x, n-1)
    ENDIF
 MEASURE n.
```

The second definition is equivalent to the first but reduces the number of times the function must be expanded in proofs.

The lemmas that are used directly in the NRV process are designed to eliminate the need to deal with the infinite Taylor series definitions of `arctan` and `ln`. For instance, to show that `arctan(p)` is greater than `c`, the following lemma is used.

```
arctan_gt_line: LEMMA
    FORALL (p,nm:real,d:nzreal,c,cb:real,n:nat):
     p>=nm/d AND cb>=c
        IMPLIES
         (arctan_lb(n)(nm,d)>cb
```

```
IMPLIES
  arctan(p)>c);
```

The PVS strategies used in NRV apply the appropriate lemma, which discharges the original proof goal and leaves a number of significantly simpler subgoals to be proved. The strategies then repeatedly split the subgoals, simplify and expand function definitions until the proof is complete.

Chapter 8

Case Studies

In this chapter two case studies are presented to demonstrate the practical application of the theory presented in Chapter 5 using the NRV system presented in Chapter 7. The case studies involve the analysis of moderately sized systems and each demonstrates a different utility of NRV. In each case study the system is analysed with respect to a certain hexagonal exclusion region about the point $(-\pi,\ 0dB)$, which is commonly used to represent stability (Section 2.2.1).

Section 8.1 formalises the hexagonal exclusion region representing the Nichols plot requirements in each of the case studies. In Section 8.2 a classic example from control engineering, the inverted pendulum [106] is analysed. The system is analysed twice: firstly with parameters that ensure the system meets its requirements; and secondly with a parameter that causes the system to fail to meet its requirements. In Section 8.3 a disk drive reader system [43, p. 444] is analysed. This system has an 'uncertain' parameter, whose value is known to lie within some range. Based on the results of these case studies, conclusions about the successes and failures of NRV are drawn in Section 8.4.

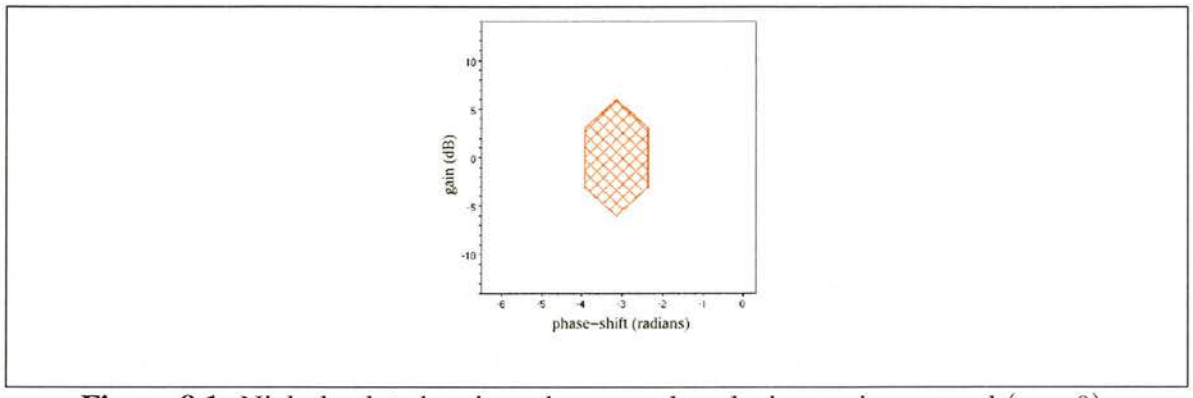


Figure 8.1: Nichols plot showing a hexagonal exclusion region around $(-\pi, 0)$.

8.1 Nichols Plot Requirements

In general, a system is considered stable (Section 2.2.1) if its Nichols plot does not enter a certain hexagonal region about the point $(-\pi,\ 0)$ as shown in Figure 8.1. This requirement can be expressed in terms of the lines bounding the region in particular intervals. The Nichols plot for the system must lie below the line between the points $(-\frac{5}{4}\pi,\ -3)$ and $(-\pi,\ -6)$ or above the line between the points $(-\frac{5}{4}\pi,\ 3)$ and $(-\pi,\ 6)$. It must lie below the line between the points $(-\pi,\ -6)$ and $(-\frac{3}{4}\pi,\ -3)$ or above the line between the points $(-\pi,\ 6)$ and $(-\frac{3}{4}\pi,\ 3)$. It must lie to the left of the line between the points $(-\frac{5}{4}\pi,\ -3)$ and $(-\frac{5}{4}\pi,\ 3)$ or to the right of the line between the points $(-\frac{3}{4}\pi,\ -3)$ and $(-\frac{3}{4}\pi,\ 3)$.

The exclusion region representing the Nichols plot requirements of the system is formulated as follows for input into NRV

$$\begin{aligned} B = \ & [(-\tfrac{5}{4}\pi..-\pi,\ [(y+\tfrac{12}{\pi}x+18,<),\ (y-\tfrac{12}{\pi}x-18,>)]),\\ & (-\pi..-\tfrac{3}{4}\pi,\ [(y+\tfrac{12}{\pi}x+6,<),\ (y-\tfrac{12}{\pi}x-6,>)]),\\ & (-3..3,\ [(x+\tfrac{3}{4}\pi,>),\ (x+\tfrac{5}{4}\pi,<)])] \end{aligned}$$

8.2 Inverted Pendulum

This section focuses on the modelling and analysis of an inverted pendulum system [106]. In Section 8.2.1 a model for the system is described. Sections 8.2.2 and 8.2.3 describe the

analysis of the system using the NRV tool described in Chapter 7. In Section 8.2.2 the system is shown to meet its requirements, whereas in Section 8.2.3 the system fails to meet its requirements.

8.2.1 Modelling an Inverted Pendulum

The inverted pendulum is a classic example from control engineering. An inverted pendulum is balanced on a cart (see Figure 8.2 and Table 8.1); when a force F is applied to the cart the pendulum and the cart move. There are two outputs of interest: the displacement

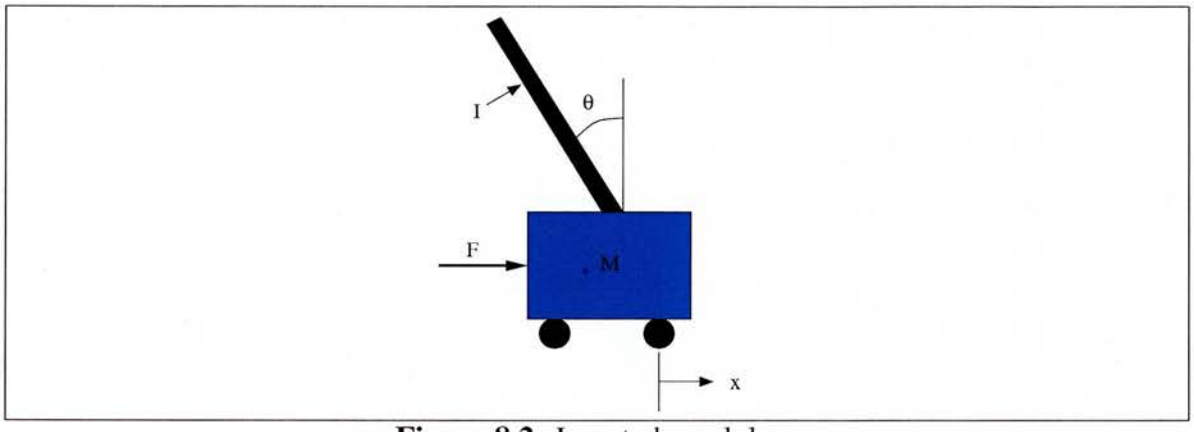


Figure 8.2: Inverted pendulum

of the cart x and the angle of the pendulum θ. When concerned only with the angle of the pendulum, the behaviour of the system can be represented using the following transfer function (see Table 8.1)

$$G_1 = \frac{mls}{(MI + Mml^2 + mI)s^3 + (bI + bml^2)s^2 - (Mmgl + m^2gl)s - bmgl}$$

The system can be modelled as the sequential combination (see Figure 8.3) of a controller G_c and the system G_1. The controller G_c used in this example is a PID (Proportional/Integral/Derivative) controller (see Appendix A.4), which is a commonly used form of controller

$$G_c = \frac{K_d s^2 + K_p s + K_i}{s}$$

designed to make the system produce the desired response.

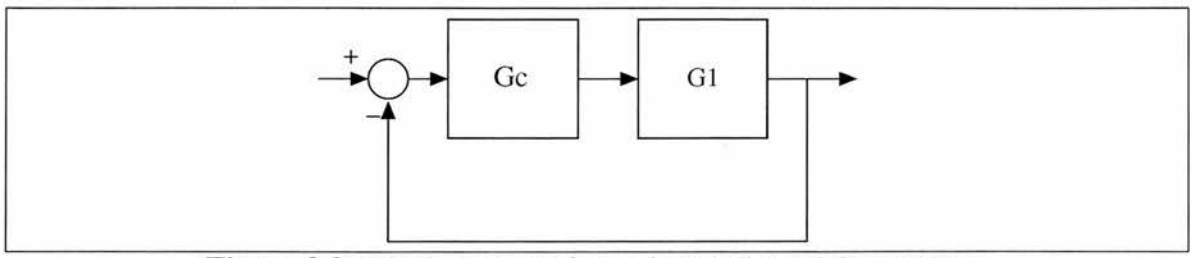


Figure 8.3: Block diagram for an inverted pendulum system

Table 8.1 shows the values for the parameters of the system chosen for this example. The value of the mass of the pendulum m is left undecided.

Mass of cart	M	0.5 kg
Friction of the cart	b	0.1 N/m/sec
Length to the pendulum's centre of mass	l	0.3 m
Inertia of the pendulum	I	0.006 kgm^2
Force of gravity	g	9.8msec^2
Proportional coefficient	K_p	3.5
Integral coefficient	K_i	-1
Derivative coefficient	K_d	-1

Table 8.1: Values for parameters in an inverted pendulum system.

8.2.2 Analysis of an Inverted Pendulum that Meets its Requirements

In order to analyse the inverted pendulum system modelled in Section 8.2.1 with regards to the Nichols plot requirements of Section 8.1, one must provide NRV with the transfer function of the system along with the correct formalisation of the exclusion region as defined in Section 7.2.

Assuming that the mass of the pendulum is 0.2kg, the open loop transfer function for the inverted pendulum system is

$$G = G_c G_1 = \frac{-25(2s^2 - 7s + 2)}{11s^3 + 2s^2 - 343s - 49}.$$

The Nichols plot for the system, showing the exclusion region is shown in Figure 8.4. Note the difficulty of ensuring that the Nichols plot requirements are met by visual inspection.

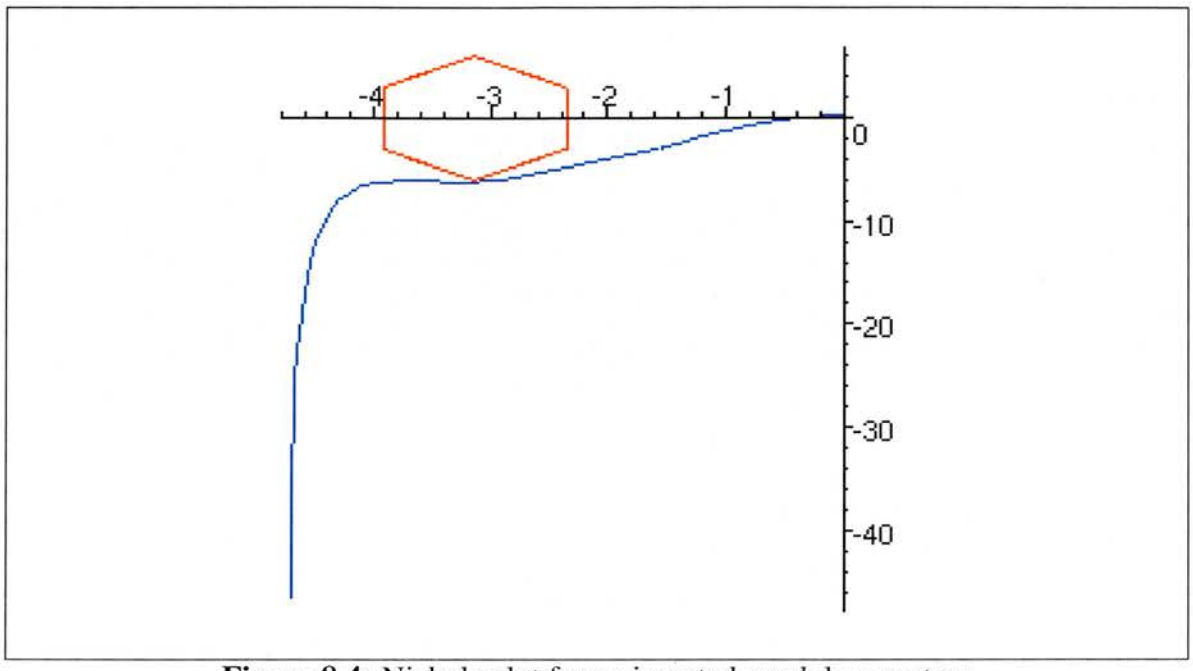


Figure 8.4: Nichols plot for an inverted pendulum system

NRV considers the relationship between the curve and the bounds of the exclusion region in each interval.

1. NRV calculates that the interval $[-\frac{5}{4}\pi, -\pi]$, in terms of x, corresponds to the interval $[\frac{157}{128}, \frac{129}{32}]$, in terms of ω and uses PVS to show that

$$\forall\omega.\frac{157}{128} \geq \omega \vee \omega \geq \frac{129}{32} \implies -\frac{5}{4}\pi \geq \text{argument}(G) \vee \text{argument}(G) \geq -\pi.$$

   Within this interval there is one point of inflection, which lies in the interval $[\frac{569}{256}, \frac{1139}{512}]$. The curve is convex for $\omega \in [\frac{157}{128}, \frac{569}{256}]$ and concave for $\omega \in [\frac{1139}{512}, \frac{129}{32}]$. NRV uses the conditions given in Section 5.3 to determine points at which the curve should be examined to ensure that it lies below the line $-\frac{12}{\pi}x - 18$ or above the line $\frac{12}{\pi}x + 18$. PVS proves that the curve lies below $-\frac{12}{\pi}x - 18$ at $\frac{157}{128}$, $\frac{569}{256}$ and $\frac{1139}{512}$, and thus that it lies outwith the exclusion region for $x \in [-\frac{5}{4}\pi, -\pi]$.

2. NRV calculates that the interval $[-\pi, -\frac{3}{4}\pi]$, in terms of x, corresponds to the interval

$[\frac{57}{128}, \frac{629}{512}]$, in terms of ω and uses PVS to show that

$$\forall\omega.\frac{57}{128} \geq \omega \vee \omega \geq \frac{629}{512} \implies -\pi \geq \text{argument}(G) \vee \text{argument}(G) \geq -\frac{3}{4}\pi.$$

Within this interval there are no points of inflection. The curve is convex for $\omega \in [\frac{57}{128}, \frac{629}{512}]$ NRV uses the conditions given in Section 5.3 to determine points at which the curve should be examined to ensure that it lies below the line $\frac{12}{\pi}x + 6$ or above the line $-\frac{12}{\pi}x - 6$. PVS proves that the curve lies below $\frac{12}{\pi}x + 6$ at $\frac{57}{128}$ and $\frac{629}{512}$, and thus that it lies outwith the exclusion region for $x \in [-\pi, -\frac{3}{4}\pi]$.

Given the nature of the exclusion region, the final condition:

$$-3 \leq y \wedge y \leq 3 \implies -\frac{3}{4}\pi < x \vee x < -\frac{5}{4}\pi,$$

could be excluded, as it is implied by the first two conditions. However, for completeness this condition is also proved. In order to prove this condition is met, rather than considering the curve represented by the set of parametric equations $x = \text{argument}(G)$, $y = \text{gain}(G)$, one must consider the curve represented by the set of parametric equations $x = \text{gain}(G)$, $y = \text{argument}(G)$.

3. NRV calculates that the interval $[-3, 3]$, in terms of y, corresponds to the interval $[0, \frac{101}{512}]$, in terms of ω and uses PVS to show that

$$\forall\omega.\omega \geq \frac{101}{512} \implies -3 \geq \text{gain}(G) \vee \text{gain}(G) \geq 3.$$

Within this interval there are no points of inflection. The curve is convex for $\omega \in [0, \frac{101}{512}]$ NRV uses the conditions given in Section 5.3 to determine points at which the curve should be examined to ensure that it lies below $-\frac{5}{4}\pi$ or above $-\frac{3}{4}\pi$. PVS proves that the curve lies above $-\frac{3}{4}\pi$ at $\frac{101}{512}$ and thus that it lies outwith the exclusion region for $y \in [-3, 3]$.

8.2.3 Analysis of an Inverted Pendulum that Fails to Meet its Requirements

In order to analyse the inverted pendulum system modelled in Section 8.2.1 with regards to the Nichols plot requirements of Section 8.1, one must provide NRV with the transfer function of the system along with the correct formalisation of the exclusion region as defined in Section 7.2.

Given that the mass of the pendulum in the inverted pendulum system has the value 0.17, the open loop transfer function for the system is

$$G = G_c G_1 \frac{-4250(2s^2 - 7s + 2)}{1945s^3 + 355s^2 - 55811s - 8330}$$

The Nichols plot for the system, showing the exclusion region is shown in Figure 8.5. Note the difficulty of ensuring that the Nichols plot requirements are met by visual inspection.

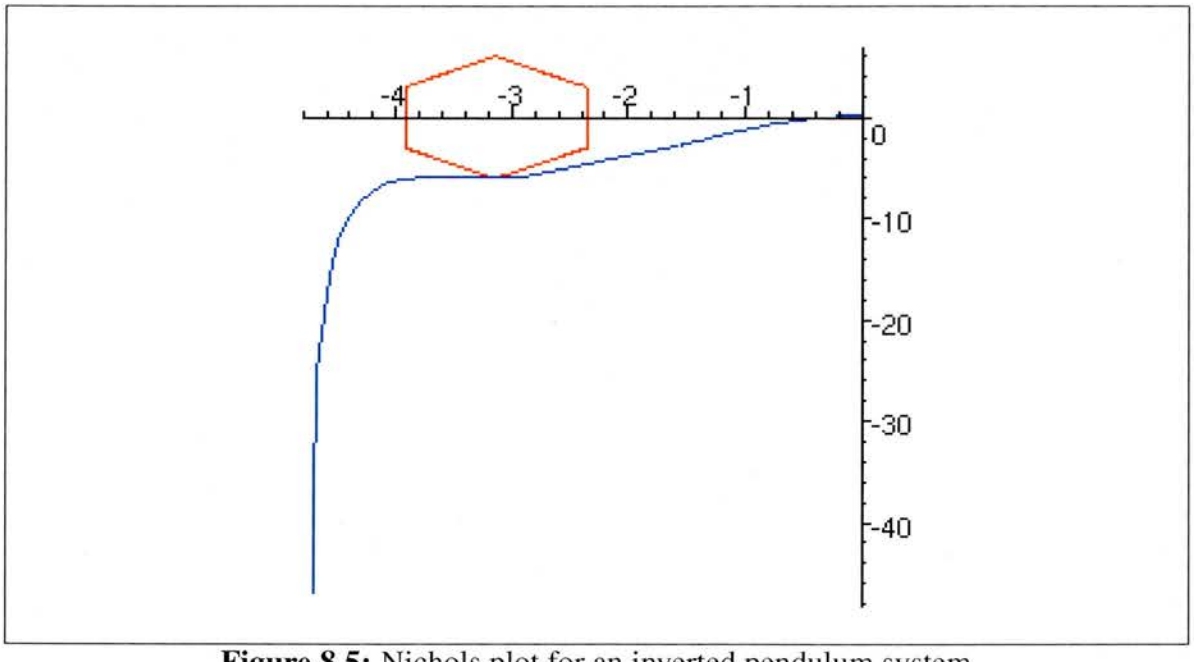


Figure 8.5: Nichols plot for an inverted pendulum system

NRV considers the relationship between the curve and the bounds of the exclusion region in each interval.

1. NRV calculates that the interval $[-\frac{5}{4}\pi, -\pi]$, in terms of x, corresponds to the interval $[\frac{79}{64}, \frac{517}{128}]$, in terms of ω and uses PVS to show that

$$\forall\omega.\frac{79}{64} \geq \omega \vee \omega \geq \frac{517}{128} \implies -\frac{5}{4}\pi \geq \text{argument}(G) \vee \text{argument}(G) \geq -\pi.$$

Within this interval there is one point of inflection, which lies in the interval $[\frac{1059}{512}, \frac{265}{128}]$. The curve is convex for $\omega \in [\frac{79}{64}, \frac{1059}{512}]$ and concave for $\omega \in [\frac{265}{128}, \frac{517}{128}]$. NRV uses the conditions given in Section 5.3 to determine points at which the curve should be examined to ensure that it lies below the line $-\frac{12}{\pi}x - 18$ or above the line $\frac{12}{\pi}x + 18$. PVS proves that the curve lies below $-\frac{12}{\pi}x - 18$ at $\frac{79}{64}$ and $\frac{1059}{512}$ but cannot show that it lies below it at $\frac{265}{128}$. NRV attempts to show that the curve lies above $\frac{12}{\pi}x + 18$ at $\frac{79}{64}$, $\frac{1059}{512}$ and $\frac{265}{128}$ but fails. NRV attempts to find a counter example to show that the Nichols plot does not meets its requirements in this interval. Since a proof could not be found that the curve either lay above the upper or below the lower bounding lines at $\frac{265}{128}$, this is used. PVS proves that at this point the curve lies within the exclusion region and thus the Nichols plot fails to meets its requirements for $x \in [-\frac{5}{4}\pi, -\pi]$.

2. NRV calculates that the interval $[-\pi, -\frac{3}{4}\pi]$, in terms of x, corresponds to the interval $[\frac{231}{512}, \frac{633}{512}]$, in terms of ω and uses PVS to show that

$$\forall\omega.\frac{231}{512} \geq \omega \vee \omega \geq \frac{633}{512} \implies -\pi \geq \text{argument}(G) \vee \text{argument}(G) \geq -\frac{3}{4}\pi.$$

Within this interval there are no points of inflection. The curve is convex for $\omega \in [\frac{231}{512}, \frac{633}{512}]$ NRV uses the conditions given in Section 5.3 to determine points at which the curve should be examined to ensure that it lies below the line $\frac{12}{\pi}x + 6$ or above the line $-\frac{12}{\pi}x - 6$. NRV attempts to prove that the curve lies below $\frac{12}{\pi}x + 6$ at $\frac{57}{128}$ and $\frac{629}{512}$, however, no proof can be found. NRV then attempts to prove that the curve lies above $-\frac{12}{\pi}x - 6$ at $\frac{57}{128}$, however, no proof can be found. NRV attempts to find a counter example to show that the Nichols plot does not meets its requirements in this interval. Since a proof could not be found that the curve either lay above the upper or below the lower bounding lines at $\frac{57}{128}$, this is used. PVS proves that at this point the curve lies within the exclusion region and thus the Nichols plot fails to meets its requirements for $x \in [-\pi, -\frac{3}{4}\pi]$.

Given the nature of the exclusion region, the final condition:

$$-3 \leq y \wedge y \leq 3 \implies -\frac{3}{4}\pi < x \vee x < -\frac{5}{4}\pi,$$

could be excluded, as it is implied by the first two conditions. However, for completeness this condition is also examined. In order to prove this condition is met, rather than considering the curve represented by the set of parametric equations $x = \text{argument}(G)$, $y = \text{gain}(G)$, one must consider the curve represented by the set of parametric equations $x = \text{gain}(G)$, $y = \text{argument}(G)$.

1. NRV calculates that the interval $[-3, 3]$, in terms of y, corresponds to the interval $[0, \frac{55}{256}]$, in terms of ω and uses PVS to show that

   $$\forall \omega . \omega \geq \frac{55}{256} \implies -3 \geq \text{gain}(G) \vee \text{gain}(G) \geq 3.$$

   Within this interval there is one point of inflection, which lies in the interval $[\frac{103}{512}, \frac{13}{64}]$. The curve is convex for $\omega \in [0, \frac{103}{512}]$ and concave for $\omega \in [\frac{13}{64}, \frac{55}{256}]$ NRV uses the conditions given in Section 5.3 to determine points at which the curve should be examined to ensure that it lies below $-\frac{5}{4}\pi$ or above $-\frac{3}{4}\pi$. PVS proves that the curve lies above $-\frac{3}{4}\pi$ at $\frac{103}{512}$, $\frac{13}{64}$ and $\frac{55}{256}$, and thus that it lies outwith the exclusion region for $y \in [-3, 3]$.

8.3 Disk Drive Read System

This section focuses on the modelling and analysis of a magnetic disk drive reader system [43, p. 444]. In Section 8.3.1 a model for the system is described. Section 8.3.2 describes the analysis of the system using the NRV system described in Chapter 7.

8.3.1 Modelling a Disk Drive Reader

Modern computers use magnetic disks to store data. Data on a disk is accessed using a disk drive reader, which positions a reader head over a track on the disk in order to read the data.

A disk drive reader consists of a controller (or *amplifier*), a motor, an arm and a read head. A metal spring (or *flexure*) is used to hold the read head slightly above the disk.

The system can be modelled as the sequential combination of a controller G_c, a motor coil G_1, an arm G_2, and a flexure and head G_3 [43, p. 444] (see 8.6).

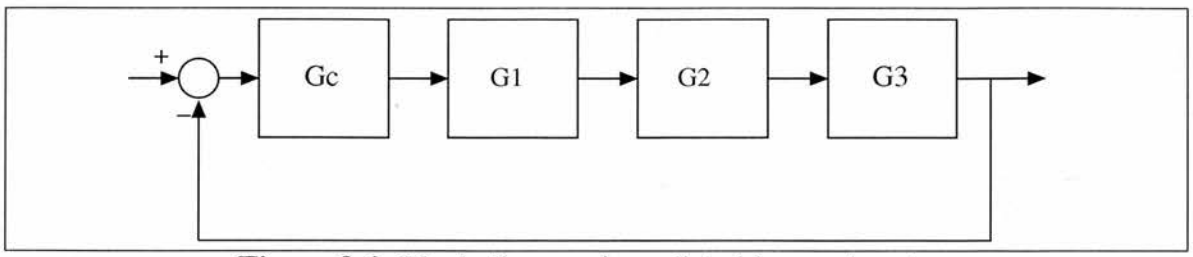


Figure 8.6: Block diagram for a disk drive read system

The motor coil and arm are modelled as a two mass spring/mass/damper system G_0 with a rigid spring.

$$G_0 = G_1 G_2$$

$$G_1 = \frac{K_m}{Ls+R}$$

$$G_2 = \frac{1}{s((M_1+M_2)s+b_1)}$$

The flexure and head is modelled as a single mass spring/mass/damper system (see Example 2.2).

$$G_3 = \frac{1}{M_0 s^2 + bs + k}$$

The controller is a simple proportional controller (Appendix A.4) designed to make the system produce the desired response more quickly.

$$G_c = K_p$$

Table 8.2 shows the values for these parameters based on the typical values for disk drive systems [43, p. 156]. The value of the motor constant K_m is left undecided.

Field inductance	L	1 mH
Field resistance	R	1 Ω
Motor mass	M_1	0.02 kg
Arm mass	M_2	0.005 kg
Friction at motor	b_1	0.5 kg/m/s
Head mass	M_0	$0.3\,10^{-6}$kg
Friction at head	b	$0.3\,10^{-2}$kg/m/s
Force of spring	k	10
Proportional coefficient	K_p	700

Table 8.2: Values for parameters in a disk drive system.

8.3.2 Analysis of a Disk Drive Reader that meets its Requirements

In order to analyse the disk drive reader system modelled in Section 8.3.1 with regards to the Nichols plot requirements of Section 8.1, one must provide NRV with the transfer function of the system along with the correct formalisation of the exclusion region as defined in Section 7.2.

Given that the motor constant is known to lie within the interval $[120, 130]$ and is represented by the constant K_m, the open loop transfer function for the disk drive reader system is

$$G = GcG_0G_3\frac{280,000,000,000K_m}{(s+1,000)s(s+20)(3s^2+30,000s+100,000,000)}$$

The Nichols plot for the system, showing the exclusion region is shown in Figure 8.7.

NRV considers the relationship between the curve and the bounds of the exclusion region in each interval.

1. NRV calculates that the interval $[-\frac{5}{4}\pi, -\pi]$, in terms of x, corresponds to the interval $[\frac{15839}{128}, \frac{354991}{512}]$, in terms of ω and uses PVS to show that

$$\forall\omega.\frac{15839}{128} \geq \omega \vee \omega \geq \frac{354991}{512} \implies -\frac{5}{4}\pi \geq \text{argument}(G) \vee \text{argument}(G) \geq -\pi.$$

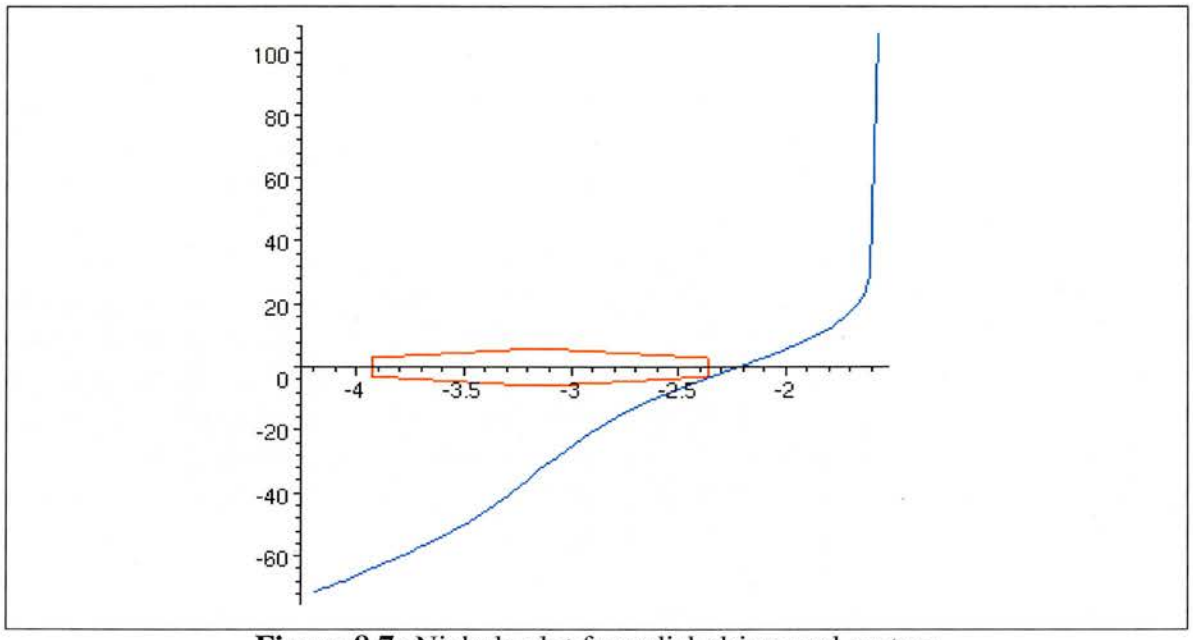


Figure 8.7: Nichols plot for a disk drive read system

To determine whether the Nichols plot meets its requirements in this interval it must be shown that $\text{gain}(G) < -\frac{12}{\pi}\text{argument}(G) - 18$ or $\text{gain}(G) > \frac{12}{\pi}\text{argument}(G) + 18$. First the function $\text{gain}(G) + \frac{12}{\pi}\text{argument}(G) + 18$ is considered. By differentiating this function with respect to K_m it is determined that it is convex with respect to K_m for all K_m and ω. NRV uses the conditions given in Section 5.8 to determine that the function should be examined at $K_m = 120$ and $K_m = 130$ to ensure that it is negative for all K_m and ω. The problem has been reduced to two one–dimensional problems in the variable ω only. NRV determines that the function is convex for $K_m = 120$ for all $\omega \in [\frac{15839}{128}, \frac{354991}{512}]$. PVS proves that the function is negative at $w = \frac{15839}{128}$ and $w = \frac{354991}{512}$. NRV determines that the function is convex for $K_m = 130$ for all $\omega \in [\frac{15839}{128}, \frac{354991}{512}]$. PVS proves that the function is negative at $w = \frac{15839}{128}$ and $w = \frac{354991}{512}$. Thus, using the conditions of Section 5.3 it is concluded that the Nichols plot lies outwith the exclusion region for $x \in [-\frac{5}{4}\pi, -\pi]$.

2. NRV calculates that the interval $[-\pi - \frac{3}{4}\pi]$, in terms of x, corresponds to the interval $[\frac{9745}{512}, \frac{63357}{512}]$, in terms of ω and uses PVS to show that

$$\forall\omega.\frac{9745}{512} \geq \omega \vee \omega \geq \frac{63357}{512} \implies -\pi \geq \text{argument}(G) \vee \text{argument}(G) \geq -\frac{3}{4}\pi.$$

To determine whether the Nichols plot meets its requirements in this interval it must

be shown that $\text{gain}(G) < \frac{12}{\pi}\text{argument}(G) + 6$ or $\text{gain}(G) > -\frac{12}{\pi}\text{argument}(G) - 6$. First the function $\text{gain}(G) - \frac{12}{\pi}\text{argument}(G) - 6$ is considered. By differentiating this function with respect to K_m it is determined that it is convex with respect to K_m for all K_m and ω. NRV uses the conditions given in Section 5.8 to determine that the function should be examined at $K_m = 120$ and $K_m = 130$ to ensure that it is negative for all K_m and ω. The problem has been reduced to two one–dimensional problems in the variable ω only. NRV determines that the function is convex for $K_m = 120$ for all $\omega \in [\frac{9745}{512}, \frac{63357}{512}]$. PVS proves that the function is negative at $w = \frac{9745}{512}$ and $w = \frac{63357}{512}$. NRV determines that the function is convex for $K_m = 130$ for all $\omega \in [\frac{9745}{512}, \frac{63357}{512}]$. PVS proves that the function is negative at $w = \frac{9745}{512}$ and $w = \frac{63357}{512}$. Thus, using the conditions of Section 5.3 it is concluded that the Nichols plot lies outwith the exclusion region for $x \in [-\pi, -\frac{3}{4}\pi]$.

Given the nature of the exclusion region, the final condition:

$$-3 \leq y \wedge y \leq 3 \implies -\frac{3}{4}\pi < x \vee x < -\frac{5}{4}\pi,$$

could be excluded, as it is implied by the first two conditions. However, for completeness this condition is also proved.

1. NRV calculates that the interval $[-3, 3]$, in terms of y, corresponds to the interval $[\frac{1347}{128}, \frac{9601}{512}]$, in terms of ω and uses PVS to show that

$$\forall\omega.\frac{1347}{128} \geq \omega \vee \omega \geq \frac{9601}{512} \implies -3 \geq \text{gain}(G) \vee \text{gain}(G) \geq 3.$$

   To determine whether the Nichols plot meets its requirements in this interval it must be shown that $\text{argument}(G) > -\frac{3}{4}\pi$ or $\text{argument}(G) < -\frac{5}{4}\pi$. First the function $\text{argument}(G) + \frac{3}{4}\pi$ is considered. By differentiating this function with respect to K_m it is determined that it linear with respect to K_m for all K_m and ω; that is it is both convex and concave. By default this is treated as a convex. NRV uses the conditions given in Section 5.8 to determine that the function should be examined at $K_m = 120$ and $K_m = 130$ to ensure that it is positive for all K_m and ω. The problem

has been reduced to two one–dimensional problems in the variable ω only. NRV determines that the function is convex for $K_m = 120$ for all $\omega \in [\frac{1347}{128}, \frac{9601}{512}]$. PVS proves that the function is positive at $w = \frac{1347}{128}$. NRV determines that the function is convex for $K_m = 130$ for all $\omega \in [\frac{1347}{128}, \frac{9601}{512}]$. PVS proves that the function is positive at $w = \frac{1347}{128}$. Thus, using the conditions of Section 5.3 it is concluded that the Nichols plot lies outwith the exclusion region for $y \in [-3, 3]$.

8.4 Case Study Conclusions

In this chapter, two moderately sized case studies are presented. Both case studies are based on example systems that appear regularly within control engineering texts.

In Section 8.2, an inverted pendulum system is analysed with respect to stability. The stability criteria are specified in terms of three intervals in which the Nichols plot of the system must not enter a given bounded region on the graph. In Section 8.2.2 NRV is used to analyse this system with respect to these criteria and provides guarantees that it meets its requirements. In Section 8.2.3 a parameter of the inverted pendulum system is altered slightly and the system is re–analysed with respect to the same criteria. NRV attempts to show that the system meets its requirements and when this fails NRV demonstrates that the requirements are not met by providing a counter example. In both cases, the Nichols plot for the system lies close to the exclusion region and it is difficult to be certain whether or not the plot enters it by simple visual analysis.

In Section 8.3, a disk drive reader system is analysed with respect to stability. This system has an 'uncertain' parameter, whose value is known to lie within an interval. This type of problem is difficult to analyse using classical Nichols plot techniques as it is a three dimensional rather than two dimensional problem. The classical solution is generally to plot a suite of Nichols plots showing the system response for various values of the parameter. If the system meets its requirements in all of these plots the assumption is made that the system meets its requirements for all permissible values of the parameter. In this case study

NRV provides symbolic analysis of the system for all permissible values of the parameter, providing a formal proof that the system meets its requirements.

Chapter 9

Conclusions and Further Work

The main results and achievements of the work presented in this thesis are summarised in Section 9.1 and Section 9.2 presents suggestions for further work.

9.1 Conclusions

In this thesis it has been argued that formal and symbolic methods can be integrated into classical informal and numerical analysis of linear, continuous–time, single–input single–output control systems. It has been argued that this integration can be done in an unobtrusive manner and that it is of benefit as it increases the assurance that control systems meet their requirements. This has been demonstrated by the development of a decision procedure and a system NRV for the automated formal and symbolic analysis of Nichols plot requirements.

The underlying mathematical representation of Nichols plot requirements have been examined and reduced to their most basic form. A decision procedure has been developed for use in the analysis of Nichols plot requirements. The procedure is widely applicable and can be used to decide the positivity or negativity of finitely inflective functions (as defined and explained in Chapter 5).

A logic $\mathcal{L}_1$ has been developed to classify the formulae to which the procedure applies. The concept of *minimal isolated* formulae has been developed and a *quantifier isolation* algorithm to convert arbitrary formulae in $\mathcal{L}_1$ into this form has been introduced.

The underlying theory of the procedure has been formalised in the higher order theorem prover PVS as an extensive library. Both the geometric properties of functions, upon which the procedure relies, and the logic $\mathcal{L}_1$ have been formalised. Proofs of the completeness and termination of the procedure and the quantifier isolation algorithm have also been developed in PVS.

A system for the automated and symbolic verification of Nichols plot requirements NRV, which uses the procedure as the basis of its analysis, has been developed. NRV was developed in the Maple–PVS–QEPCAD system, which exploits the symbolic computation provided by the computer algebra system Maple, the formal techniques provided by the theorem prover PVS and the quantifier elimination routines provided by QEPCAD. NRV is highly automated and provides a graphical user interface, similar in appearance to a java applet, which allows the user of the system to have no knowledge of the underlying decision procedure, formal methods or the *Maple* or PVS syntax.

Two case studies have been presented, in which NRV either produces proofs that a system meets its requirements, or produces a counter example to demonstrate that the system fails to meets its requirements.

NRV may fail to provide a proof or counter example automatically. In these cases, NRV attempts to produce useful feedback to the analyst, indicating where the analysis failed and whether the failure was within a Maple computation or in a PVS proof. NRV attempts to highlight areas that may require closer inspection. In practice, several case studies have been performed and NRV has produced promising results rarely failing to find proofs. The case studies presented in Chapter 8 are representative of the case studies performed.

It has been indicated by control engineers from companies such as DSTL, QinetiQ and The Mathworks that automated Nichols plot analysis would be useful if it could be applied to

control systems of degree 5. Several case studies have been performed that show that NRV is highly successful in the analysis of control systems of this degree, a representative case of which has been presented in this thesis.

9.2 Further Work

This chapter suggests several directions for further work based on the work presented in this thesis. These directions fall into one of two broad categories. In Section 9.2.1 potential improvements to the NRV tool described in Chapter 7 are suggested. Section 9.2.2 presents potential applications of the decision procedure of Chapter 5 both within and outwith the field of control engineering.

9.2.1 Nichols Plot Requirements Verification

The NRV tool for the automated formal and symbolic analysis of Nichols plot requirements for control or dynamical systems is described in Chapter 5. The tool is implemented in the Maple–PVS–QEPCAD system and is designed to take advantage of the strengths of each of the individual systems. The tool provides improvements over traditional Nichols plot requirement analysis; however, there are several improvements that could be made to NRV.

NRV uses Maple to provide efficient symbolic calculation and PVS to provide assurances of correctness. NRV is more efficient than an implementation of the decision procedure in PVS alone; however, owing to its reliance on PVS for certain key calculations, it can still take several hours to produce proofs that even a moderately sized system meets its requirements. This is acceptable if NRV is used only a small number of times, particularly during the final stages of analysis but may not be acceptable during the earlier stages of experimentation. As evidenced in Section 3.3 the use of symbolic methods in the analysis of systems is desirable; even without any formal assurance of correctness symbolic analysis still provides improvements over classical numerical analysis. Allowing NRV to have two

modes – *formal mode*, which uses Maple–PVS–QEPCAD to provide formal assurances of correctness, and *symbolic mode*, which uses only Maple to provide symbolic analysis – would allow NRV to be used at any stage of system development. Symbolic mode could be used in the earliest stages of system development to provide higher levels of assurance than numerical techniques, with little or no extra cost. The more time consuming formal mode of analysis could be used in the later stages of system development to provide formal assurance of correctness. The alteration to NRVto allow two separate modes is relatively minor and could be implemented using a flag in the *Maple* code that indicates whether the PVS should be used or not, along with some simple *Maple* procedures that perform numeric calculations in place of the PVS proofs.

Currently the system is fairly monolithic, providing no means of 'pausing' or halting the calculation. NRV will analyse a system for all given requirements; if it is found that the system fails to meet one of its requirements NRV will continue to analyse the system for any remaining requirements. It may be desirable to allow the user to instruct NRV to halt as soon as it determines that a system fails to meet its requirements. This could be implemented using a flag in the *Maple* code and would require fairly minor alterations to the structure of the *Maple* procedures.

NRV accepts an input δ that represents the degree of accuracy to which calculations should be performed. This value is propagated throughout the system. NRV does not allow different degrees of accuracy to be specified for different calculations. A lower degree of accuracy generally allows efficiency to be increased and may be sufficient for certain calculations but not others. If the degree of accuracy is too low for any particular calculation, conclusions can not be drawn about a control system. Allowing the user to specify different degrees of accuracy for different calculations may allow the efficiency of the analysis to be increased without increasing the risk of failure. This alteration could be done in various different ways, with varying degrees of ease. In the simplest method, the user would be prompted to provide several degrees of accuracy, one for each distinct type of calculation. For instance, δ_1 would specify the degree of accuracy for calculating of points of inflection, δ_2 would specify the degree of accuracy for converting between coordinate systems, and

δ_3 would specify the degree of accuracy for calculating bounds on the natural logarithm and arctan. These values would be propagated throughout the system and used where appropriate. In a more complex method, the user could specify several degrees of accuracy for different types of calculation. The different degrees would be used when analysing the system in different intervals.

If NRV fails to find a proof that a system meets its requirements it attempts to produce a counter example. Currently, if a counter example is found NRV concludes that the system fails to meet its requirements and presents the user with this counter example. By incorporating simple control system design strategies into NRV, counter examples could be presented along with suggested alterations that would allow the control system to meet its requirements. If NRV fails to find either a proof or a counter example, an indication is given as to where this failure took place, in Maple or in PVS, and the user is presented with the values of the gain, phase–shift and frequency at the point of failure. This is very basic information designed to indicate to the user where the failure took place. By examining the values of various internal variables NRV could perform an amount of self–diagnosis and produce suggestions to correct the failure. These suggestions would likely involve altering the degree of accuracy in various calculations. The structure of NRV allows the easy addition of *Maple* procedures that could be used for either of these tasks.

NRV attempts to show that the Nichols plot of a system meets given requirements. These requirements are represented as a list. NRV analyses the system for each element in the list. As indicated in Chapter 8, there may be cases in which the conditions represented in an element of the list are implied by conditions represented in the other elements. NRV could use simple inference rules to simplify conditions, reducing the amount of analysis required thus increasing efficiency. *Maple* has a limited ability to determine whether logical formulae are true or false. By using the *is* procedure to determine whether a (pair of) condition(s) implies another, *Maple* may be able to simplify the conditions. Investigation should be performed into how effective this strategy for simplification is.

9.2.2 Applications of the Decision Procedure

The decision procedure presented in Chapter 5 can be used to determine the positivity or negativity of finitely inflective functions in intervals. The procedure is general enough to be widely applicable both within and outwith the field of control engineering. The suggestions for further work presented in this section are speculative and would require more extensive research than those in the previous section.

It has been shown that this procedure can be used to provide formal and symbolic analysis of control systems with respect to Nichols plot requirements. These requirements are expressed in terms of regions on a graph, which the plot must not enter, and are classically analysed visually. Many classical control system analysis techniques are graphical, with requirements that can be expressed in a similar manner to Nichols plot requirements. It is likely that the decision procedure could be applied directly to the analysis of Bode diagrams, since Bode diagrams and Nichols plots are essentially different methods of viewing the same gain/phase–shift data. There is potential for further investigation into the application of the procedure to other graphical analysis techniques, such as, Nyquist (see Section 2.2.1) and time–response plots (see Appendix A.3).

In this thesis the focus has been placed on the analysis of continuous–time, single–input, single–output, linear, time–invariant control systems. Further investigations could be performed into the applicability of the procedure to the analysis of multi–input, multi-output, discrete and nonlinear systems. Initially, it must be determined whether the analysis techniques and requirements for these types of systems can be expressed appropriately for the decision procedure. The NRVtool could then be extended to allow the analysis of these systems.

Investigations could be performed into the use of the procedure in fields other than control engineering. For instance, in 3–dimensional graphical imaging, the technique of *ray tracing*, in which it must be determined whether a ray intersects a curve, is used. Efficiency of computation is often extremely important in ray tracing. Currently, the *Lipschitz method* is

most commonly used for ray tracing. Investigations could be performed into the efficiency and accuracy of the Lipschitz method versus the decision procedure.

Appendix A

Time–Domain Analysis

This appendix gives a brief introduction to some of the basic concepts and methods used by control engineers in the development of control systems in the time domain. This appendix complements Chapter 2, highlighting the fact that analysis in the time domain is also numerical and often graphical, and is included for completeness. The commonly used PID controller is detailed in Section A.4

A.1 State-space Representation

Many systems of linear differential equations can be written in state-space form. State-space representation uses vector-matrix notation to represent the system of first-order differential equations that model a system, and comprise a *state vector* $\mathbf{x}(t)$, which represents the state variables of a system at a given time t, and an output equation $y(t)$.

Given the general linear differential equation shown in 2.1 and introducing additional variables $x_1(t) = y(t)$, $x_2(t) = y'(t), \ldots, x_n(t) = y^{n-1}(t)$, then:

$$x_1'(t) = x_2(t),\ x_2'(t) = x_3(t), \ldots,\ x_{n-1}'(t) = x_n(t),$$

$$x_n'(t) = y^n(t) = -a_{n-1}x_n(t) \cdots - a_0x_1(t) + q_0u(t) \tag{A.1}$$

and the state-space representation is as follows:

$$\mathbf{x}'(t) = \mathbf{A}\mathbf{x}(t) + u(t)\mathbf{B}$$

$$y(t) = \mathbf{c}.\mathbf{x}(t) \tag{A.2}$$

where the state vector $\mathbf{x}(t)$ and its derivative with respect to time are:

$$\mathbf{x}(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \\ \dots \\ x_n(t) \end{bmatrix}, \quad \mathbf{x}'(t) = \begin{bmatrix} x_1'(t) \\ x_2'(t) \\ \dots \\ x_n'(t) \end{bmatrix}$$

the system matrix $\mathbf{A}$ is an nxn matrix given by

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 0 & 1 \\ -a_0 & -a_1 & -a_2 & \dots & -a_{n-1} \end{bmatrix}$$

and the control vector $\mathbf{B}$ is an n-vector of the form

$$b = \begin{bmatrix} 0 \\ 0 \\ \dots \\ 0 \\ q_0 \end{bmatrix}$$

The output equation is of the form:

$$y(t) = \mathbf{c}.\mathbf{x}(t)$$

where

$$\mathbf{c} = [c_1 \quad c_2 \quad \dots \quad c_n]$$

Example A.1 *The state-space representation of the spring/mass/damper system of Example 2.1 is as follows:*

$$\text{Let } x_1(t) = y(t),\ x_2(t) = y'(t)$$

$$\text{Then } x_1'(t) = x_2(t),\ x_2'(t) = -\frac{k}{m}x_1(t) - \frac{b}{m}x_2(t) + \frac{u}{m},$$

$$\mathbf{x}'(t) = \mathbf{A}\mathbf{x}(t) + u(t)\mathbf{B} = \begin{bmatrix} 0 & 1 \\ -\frac{k}{m} & -\frac{b}{m} \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} + u(t) \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix} \tag{A.3}$$

The complexity of the equations in this representation does not increase with the numbers of inputs, outputs and state variables. This means systems with many inputs and outputs, which may interrelate in a complex way, can be represented in a relatively simple manner.

A.2 Time–Response Analysis

To analyse the stability of a system expressed in state-space form, the Liapunov stability criteria can be used. Given a linear time–invariant continuous–times system $\mathbf{x}'(t) = \mathbf{A}\mathbf{x}(t)$, where $\mathbf{x}(t)$ is a state–vector (n–vector) and $\mathbf{A}$ is an $n \times n$ matrix, the system is stable if for any real and symmetric matrix Q there exists a real, symmetric, positive definite[1] matrix P such that $\overline{A}^T P + PA = -Q$.

To analyse the real–time response of a system one can examine the shape of the curve produced by exposing the system to various inputs. There are three types of input for time-response analysis, as shown in Figure A.1.

There are three types of response that a system could have to these inputs and a system may have a different response depending on the chosen input. It would be desirable for the system to have the same response to all classes of inputs, since one usually does not know exactly what inputs the system will encounter in practice. The three responses that a system could have are shown in Figure A.2.

[1] A matrix P is *positive definite* if $\overline{\mathbf{x}}^T P\mathbf{x} > 0$ for all $\mathbf{x} \in \mathbf{C}^n$

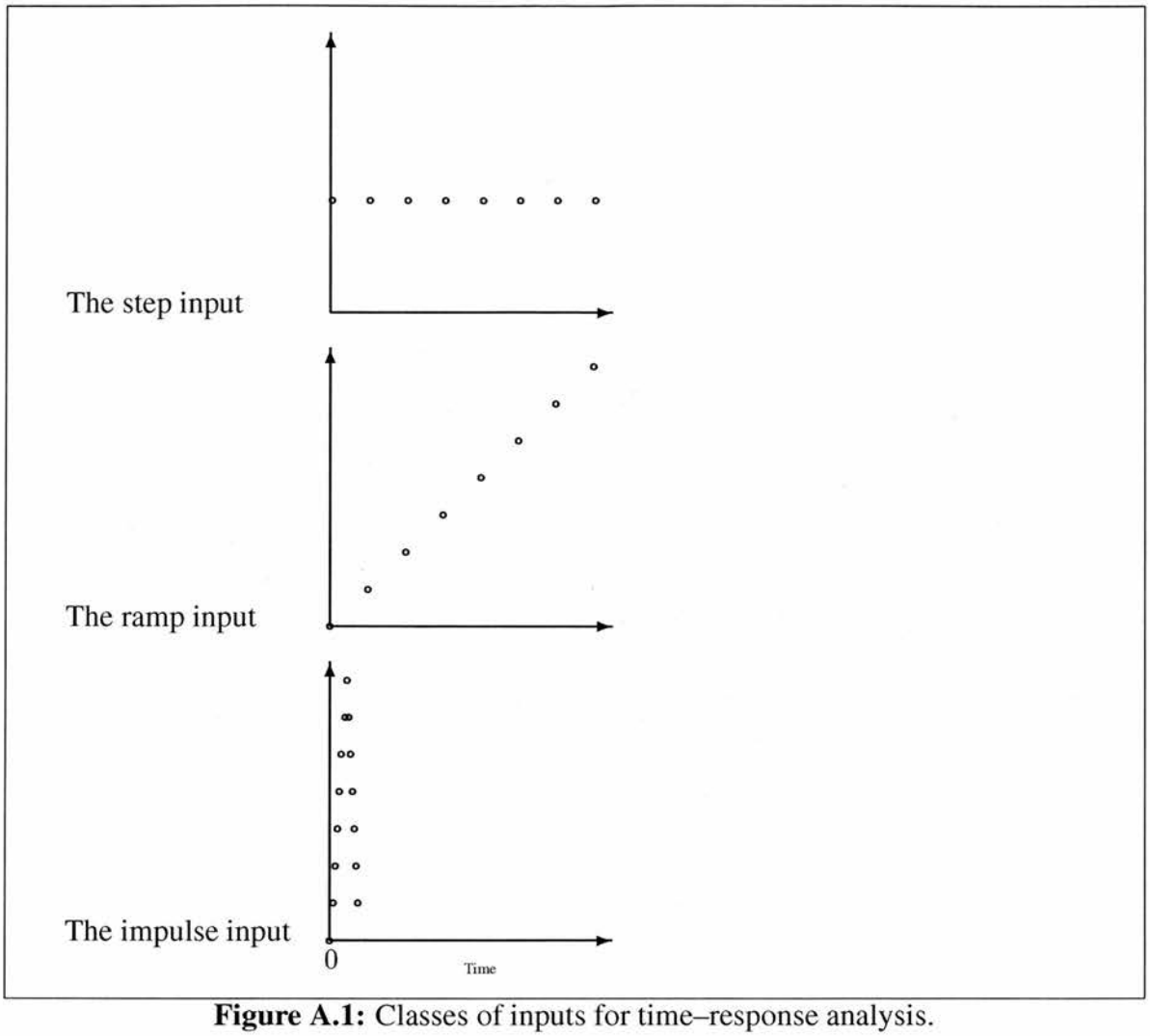


Figure A.1: Classes of inputs for time–response analysis.

This analysis of a system can be performed by examining the poles of the Laplace transform of a system. For example, suppose a closed loop system has the Laplace transform

$$\frac{C(s)}{R(s)} = \frac{f(\omega_n)}{s^2 + 2\zeta\omega_n s + \omega_n^2}$$

where ω_n is the natural frequency of the system. The poles of this system can be found by finding the roots of $s^2 + 2\zeta\omega_n s + \omega_n^2$. Given any quadratic equation there are three categories that the roots could fall into — two complex roots; two real, equal roots; two real, non-equal roots — and each of these categories corresponds to a different type of response that the system could have (see Table A.1). Using the table one could easily see what the type of response a particular system of this form would have by examining the value of ζ.

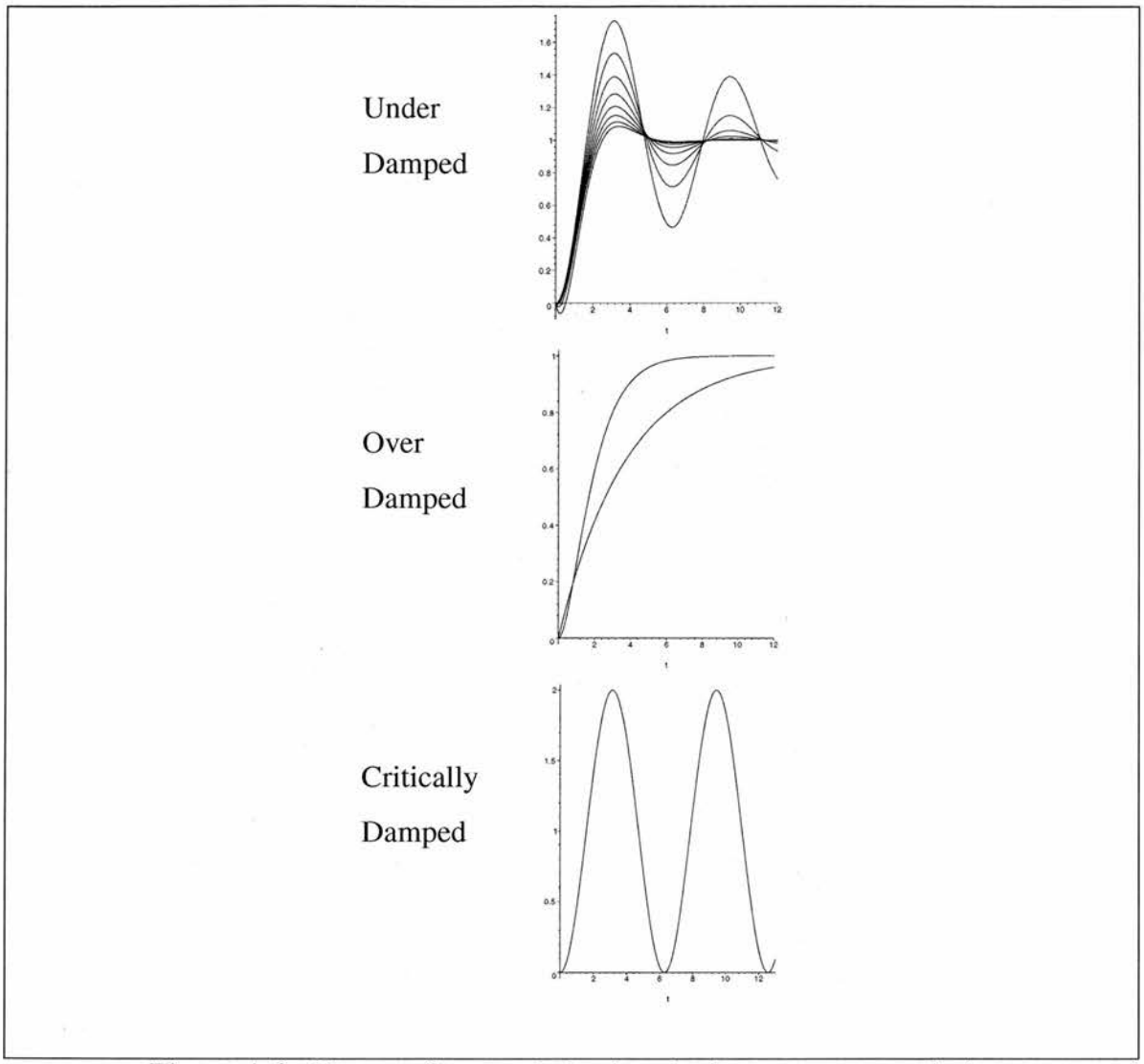


Figure A.2: Classes of system behaviour in time–response analysis.

A.3 Time-Response Requirements

When designing a system one usually has a more detailed set of requirements for its real–time response; for example, one may have a desired time in which the system should become steady. These requirements can be expressed in the form of a response specification. It is usual for response specifications to contain the following information:

1. Desired delay time t_d — the time required for the response to first reach half the desired final value.

Roots	Response	Parameter	Inverse Laplace Transform of Quadratic Transfer Function
Complex	under damped	$0 < \zeta < 1$	$1 - \frac{\zeta e^{-\zeta\omega_n t}}{\sqrt{1-\zeta^2}} \sin(\omega_n t) + e^{-\zeta\omega_n t}\cos(\omega_n t)$
Real and Equal	critically damped	$\zeta = 1$	$1 - e^{-\omega_n t}(1 + \omega_n t)$
Real and Non–equal	over damped	$\zeta > 1$	$1 - e^{-(\zeta - \sqrt{\zeta^2-1})\omega_n t}$

Table A.1: Response analysis for systems with quadratic transfer functions.

2. Desired rise time t_r — the time required for the response to rise from either 10% to 90%, 5% to 95% or 0% to 100% of its desired final value, depending on the system.

3. Desired peak time t_p — the time required for the response to reach the first peak of the overshoot.

4. Maximum percent M_p — the percentage of the final value that the maximum peak can overshoot the desired final output by.

5. Desired settling time t_s — the time required for the response to reach and stay within a specified range of the final value (usually 2% or 5%).

If all of these values are specified then the shape of the response curve is virtually determined (see Figure A.3).

A.4 Basic Control Actions: The PID controller

Often when modelling a system one finds that the system as is cannot meet the desired requirements, for instance it is a general requirement for aeroplanes that they are stable, however, many modern fighter jets are inherently unstable. In these cases a controller must be added to modify the system response. These controllers are combined with the original

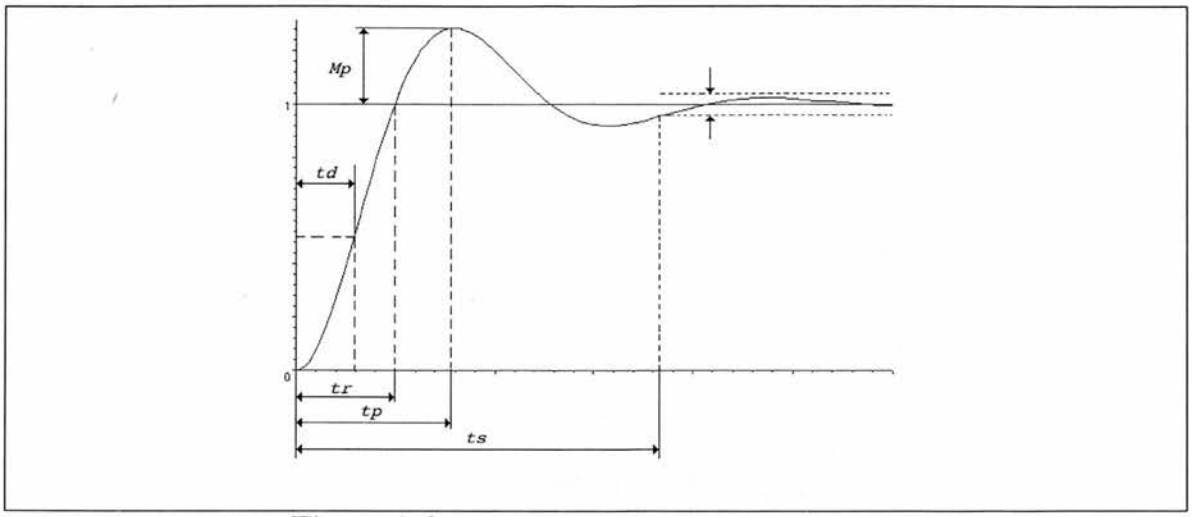


Figure A.3: An example time–response curve.

system using one of the configurations show in the Section 2.1.4. Controllers can take many different forms. Three commonly used control actions are the proportional gain, the integral gain, and the derivative gain controllers (see Table A.2).

Proportional	$E(s)$	K_p	$U(s)$	$u(t) = K_p e(t)$
Integral	$E(s)$	$\frac{K_i}{s}$	$U(s)$	$\frac{du(t)}{dt} = K_i e(t)$
Derivative	$E(s)$	$K_d s$	$U(s)$	$u(t) = K_d \frac{de(t)}{dt}$

Table A.2: Control actions.

The three different control actions have different effects on a system (see Table A.3). The control action that should be used depends on which requirement(s) the system is failing to meet. One may add a combination of controllers if necessary. For example, if a system is

under damped (i.e has a long rise time) one may add a proportional controller to decrease the rise time. From Table A.3, one can see that along with decreasing the rise time a proportional controller also increases the overshoot. One may find that the proportional controller increases the overshoot to an unacceptable level. In this case one could add a derivative controller, which decreases overshoot but makes very little difference to the rise time. By adjusting the values of K_p and K_d one can increase or decrease the effect that the controllers have on the system.

	Rise time t_r	Overshoot M_p	Settling time t_s
K_p	$\searrow$	$\nearrow$	$\approx$
K_i	$\searrow$	$\nearrow$	$\nearrow$
K_d	$\approx$	$\searrow$	$\searrow$

Table A.3: PID effects on response.

Appendix B

PVS Libraries Formalising the Decision Procedure

This appendix contains the PVS library formalising the mathematical foundations used as a basis for the decision procedure detailed in Section 5.7, along with the logic $\mathcal{L}_1$ and the decision procedure itself. This formalisation is presented in some detail in Chapter 6.

In the following theory, polynomial, linear, rational and rationally differentiable functions are defined.

```
polynomial [T: TYPE from real]: THEORY

 BEGIN

%-----------------------------------------------------------
% Since this theory deals with the differentiability of
% polynomials and differentiability is only defined on
% non-singleton convex sets, assumptions must be made
% that T is such a set.
%-----------------------------------------------------------
 ASSUMING

   connected_domain : ASSUMPTION
```

```
    FORALL (x, y : T), (z : real) :
      x <= z AND z <= y IMPLIES T_pred(z)

  not_one_element : ASSUMPTION
    FORALL (x : T) : EXISTS (y : T) : x /= y

 ENDASSUMING

 IMPORTING transcendentals@diffl_pow,
           nth_derivatives@nth_derivatives


%-----------------------------------------------------------
% The following are auxiliary definitions concerning
% subsequences, sums and factorials that are required
% in the definition of polynomials.
%-----------------------------------------------------------
 subseq(f:finseq[real],m:nat,n:int): finseq[real] =
   IF m > n OR m >= f`length THEN
     empty_seq
   ELSE
     LET len = min(n - m + 1, f`length - m) IN
       (# length := len,
        seq := (LAMBDA (x: below[len]): f`seq(x + m)) #)
   ENDIF

 nonempty_list : TYPE = {l:list[real] |length(l)>0}

 facn(n:nat,m:{x:nat|x<=n}): RECURSIVE posnat =
   IF n = m THEN 1 ELSE n * facn(n - 1, m) ENDIF
   MEASURE n

 sumto(m:nat)(f:[below[m]->real]): RECURSIVE real =
   IF m = 0 THEN 0
   ELSE sumto(m - 1)(LAMBDA (n:below[m-1]): f(n)) + f(m-1)
   ENDIF
   MEASURE m
```

```
%-----------------------------------------------------------
% Relate the definition of sumto to Hanne's definition of
% sum; sum is defined for functions over the entire
% naturals whereas sumto is defined for functions over
% a subset of the naturals.
%-----------------------------------------------------------
 sumto_sum: LEMMA FORALL (m:nat, f:[below[m]->real]):
   sumto(m)(f) =
   sum(0,m)(LAMBDA (n:nat): IF n<m THEN f(n) ELSE 0 ENDIF)

%-----------------------------------------------------------
% The function polynomial takes a sequence of reals and
% returns a polynomial functions whose coefficients are
% the elements of the sequence.
%-----------------------------------------------------------
 polynomial(S:finseq[real]): [T->real] =
   LAMBDA (x:T):
   sumto(S`length)
   (LAMBDA (n:below[S`length]): S`seq(n)*expt(x,n))

%-----------------------------------------------------------
% This function finds index of the last nonzero element in
% a sequence of reals. This is used to find the degree of
% a polynomial.
%-----------------------------------------------------------
 degree(S:finseq[real]): RECURSIVE nat =
   IF S`length=0 THEN 0
   ELSIF S`seq(S`length-1) = 0 THEN
     degree(subseq(S,0,S`length-2))
   ELSE S`length-1
   ENDIF
   MEASURE S`length

%-----------------------------------------------------------
% The function type of polynomials is defined as a
```

```
% function that can be constructed for a sequence of reals
% using the polynomial predicate. Judgements are used to
% relate the varoius definitions.
%-----------------------------------------------------------
 poly_type: TYPE =
  {f:[T->real]| EXISTS (S:finseq[real]): f=polynomial(S)}

 nzreal_poly_type: TYPE =
  {p:poly_type| FORALL (x:T): p(x)/=0}

 nzreal_poly_type_nzreal: JUDGEMENT
   nzreal_poly_type SUBTYPE_OF [T->nzreal]

 polynomial_has_poly_type: JUDGEMENT
   polynomial(S:finseq[real]) HAS_TYPE poly_type


%-----------------------------------------------------------
% The following definitions and lemmas concern the
% differentiability of polynomials.
%-----------------------------------------------------------
 poly_diff1: LEMMA
   FORALL (S:finseq[real],x:T):
     diff1(polynomial(S),x,
        IF S`length <=1 THEN 0
        ELSE
          sumto(S`length-1)
          (LAMBDA (n:below[S`length-1]):
          (n+1)*S`seq(n+1)*expt(x,n))
        ENDIF)

 poly_derivable: LEMMA
   FORALL (S:finseq[real]):
     derivable(polynomial(S))

 poly_deriv_fun: JUDGEMENT
   polynomial(S: finseq[real]) HAS_TYPE deriv_fun[T]
```

```
poly_type_deriv_fun: JUDGEMENT
  poly_type SUBTYPE_OF deriv_fun[T]

poly_deriv: LEMMA
  FORALL (S:finseq[real],x:T):
    deriv(polynomial(S),x)=
      IF S`length <= 1 THEN 0
      ELSE
        sumto(S`length-1)
        (LAMBDA (n:below[S`length-1]):
          (n+1)*S`seq(n+1)*expt(x,n))
      ENDIF

poly_deriv2: LEMMA
  FORALL (S:finseq[real]):
    deriv(polynomial(S))=LAMBDA (x:T):
      IF S`length <= 1 THEN 0
      ELSE
        sumto(S`length-1)
        (LAMBDA (n:below[S`length-1]):
          (n+1)*S`seq(n+1)*expt(x,n))
      ENDIF

deriv_poly(S:finseq[real]): finseq[real] =
  IF S`length <= 1 THEN
    (#length:=1, seq:= LAMBDA (n:below[1]): 0#)
  ELSE
    (#length:= S`length-1,
      seq:= LAMBDA (n:below[S`length-1]):
          (n+1)*S`seq(n+1) #)
  ENDIF

deriv_length: LEMMA
  FORALL (S:finseq[real]):
    deriv_poly(S)`length =
      IF S`length<=1 THEN 1
```

```
      ELSE S`length-1 ENDIF

deriv_nth: LEMMA
  FORALL (S:finseq[real],n:{m:nat|m<S`length-1}):
    deriv_poly(S)`seq(n)=(n+1)*S`seq(n+1)

deriv_poly_deriv: LEMMA
  FORALL (S:finseq[real],x:T):
    deriv(polynomial(S),x)=
      polynomial(deriv_poly(S))(x)

deriv_poly_deriv2: LEMMA
  FORALL (S:finseq[real]):
    deriv(polynomial(S))=
      polynomial(deriv_poly(S))

deriv_poly_type: JUDGEMENT
  deriv(p:poly_type) HAS_TYPE poly_type

cubic_deriv: LEMMA
  FORALL (a,b,c,d:real,w:T):
    deriv(polynomial(list2finseq[real]((:d,c,b,a:))))(w)=
      3*a*expt(w,2)+2*b*w+c

poly_derivable_n_times_aux: LEMMA
  FORALL (S:finseq[real]):
    derivable_n_times(polynomial(S),S`length)

poly_derivable_n_times: LEMMA
  FORALL (S:finseq[real]):
    FORALL (n:nat): derivable_n_times(polynomial(S),n)

nderiv_poly_type: JUDGEMENT
  nderiv(x:nat,p:poly_type) HAS_TYPE poly_type

poly_nderiv: LEMMA
  FORALL (S:finseq[real],m:nat,x:T):
```

```
    nderiv(m,polynomial(S))(x)=
      IF m<S`length THEN sumto(S`length-m)
        (LAMBDA (n:below[S`length-m]):
            facn(n+m,n)*S`seq(n+m)*expt(x,n))
      ELSE 0 ENDIF

poly_nderiv2: LEMMA
  FORALL (S:finseq[real],m:nat):
    nderiv(m,polynomial(S))=
      LAMBDA (x:T):
        IF m<S`length THEN sumto(S`length-m)
          (LAMBDA (n:below[S`length-m]):
            facn(n+m,n)*S`seq(n+m)*expt(x,n))
        ELSE 0 ENDIF

nderiv_poly(m:nat,S:finseq[real]): finseq[real] =
  IF m<S`length THEN
    (#length:= S`length-m,
      seq:= LAMBDA (n:below[S`length-m]):
      facn(n+m,n)*S`seq(n+m) #)
  ELSE
    (#length:=1, seq:= LAMBDA (n:below[1]): 0#)
  ENDIF

nderiv_length: LEMMA
  FORALL (S:finseq[real],m:nat):
    nderiv_poly(m,S)`length =
      IF S`length<=m THEN 1
      ELSE S`length-m ENDIF

nderiv_poly_nderiv: LEMMA
  FORALL (S:finseq[real],n:nat,x:T):
    nderiv(n,polynomial(S))(x)=
    polynomial(nderiv_poly(n,S))(x)

nderiv_poly_nderiv2: LEMMA
  FORALL (S:finseq[real],n:nat):
```

```
    nderiv(n,polynomial(S))=
    polynomial(nderiv_poly(n,S))

%-----------------------------------------------------------
% The following are definitions of linear, rational and
% rationally differentiable functions. These are defined
% in terms of polynomials.
%-----------------------------------------------------------
 linear : TYPE = {f:[T->real]|
   EXISTS (S: finseq[real]):
     degree(S)<=1 AND f=polynomial(S)}

 rational : TYPE = {f:[T->real]|
   EXISTS (p: poly_type, p2: nzreal_poly_type):
     f=p/p2}

 rational_differentiable : TYPE = {f:[T->real]|
   derivable(f) AND
     EXISTS (p: poly_type, p2: nzreal_poly_type):
     deriv(f)=p/p2}

END polynomial
```

In the following theory, convex and non–trivial sets are defined.

```
convex_set_aux [T: TYPE FROM real]: THEORY

BEGIN

%-----------------------------------------------------------
% A convex subset P of T has the implicit condition that
% T also be a convex set. This is so that we can show
% that any real z to which P is applied is of type T,
% preventing the generation of unsolveable TCCs.
%-----------------------------------------------------------
 convex_set: TYPE = {P:set[T]|
  (FORALL (x, y:(P), z:real): x <= z AND z <= y
```

```
    IMPLIES T_pred(z) AND P(z))}

 not_singleton_set: TYPE = {P:set[T]|
  (FORALL (x: (P)): EXISTS (y: (P)) : x /= y)}

 not_one_convex_set: TYPE = {P:set[T]|
  (FORALL (x, y:(P), z:real): x <= z AND z <= y
    IMPLIES T_pred(z) AND P(z))
  AND
  (FORALL (x: (P)): EXISTS (y: (P)) : x /= y)}

 nontrivial_set: TYPE = {P:set[T]|
  (EXISTS (x: T): P(x)) AND
  (FORALL (x: (P)): EXISTS (y: (P)) : x /= y)}

 nontrivial_convex_set: TYPE = {P:set[T]|
  (FORALL (x, y:(P), z:real): x <= z AND z <= y
    IMPLIES T_pred(z) AND P(z))
  AND
  (FORALL (x: (P)): EXISTS (y: (P)) : x /= y)
  AND
  (EXISTS (x: T): P(x))}

%-----------------------------------------------------------
% Relate the previous definitions of sets using
% judgements. This can prevent the generation of TCC when
% a subtype is used where its supertype is expected.
%-----------------------------------------------------------
 nontrivial_convex_set1: JUDGEMENT
   nontrivial_convex_set SUBTYPE_OF nontrivial_set
 nontrivial_convex_set2: JUDGEMENT
   nontrivial_convex_set SUBTYPE_OF convex_set

END convex_set_aux
```

In the following theory, convexity and concavity of functions of one variable are defined.

Various auxiliary lemmas that are useful in reasoning about convexity and concavity are proven.

```
convexity[T: TYPE FROM real]: THEORY

  BEGIN

%-----------------------------------------------------------
% Since convexity is only defined on functions whose
% domains are convex sets, assumptions must be made
% that T is such a set.
%-----------------------------------------------------------
  ASSUMING

    connected_domainT : ASSUMPTION
        FORALL (x, y : T), (z : real) :
            x <= z AND z <= y IMPLIES T_pred(z)

  ENDASSUMING

  IMPORTING convex_set, NRV_lib@types1, NRV_lib@types2

 T_is_convex_set: JUDGEMENT
   T_pred HAS_TYPE convex_set[real]

 line_equiv: LEMMA FORALL (f:[T->real],x,a:T,b:gt[T,a]):
  (f(b) - f(a)) / (b - a) * (x - b) + f(b)
       =
  (f(b) - f(a)) / (b - a) * (x - a) + f(a)

%-----------------------------------------------------------
% Define convexity and concavity and relate the
% definitions.
%-----------------------------------------------------------
 concave?(f:[T->real]): bool =
   FORALL (x,y:T, l:closed[real,0,1]):
     f(l*x+(1-l)*y)>=l*f(x)+(1-l)*f(y)
```

```
convex?(f:[T->real]): bool =
  FORALL (x,y:T, l:closed[real,0,1]):
    f(l*x+(1-l)*y)<=l*f(x)+(1-l)*f(y)

concave_neg_convex: LEMMA FORALL (f:[T->real]):
  concave?(f) IFF convex?(LAMBDA (x:T):-f(x))

convex_neg_concave: LEMMA FORALL (f:[T->real]):
  convex?(f) IFF concave?(LAMBDA (x:T):-f(x))

%-----------------------------------------------------------
% The following lemmas are fairly trivial but are useful
% in the proof of later lemmas.
%-----------------------------------------------------------
concave_aux: LEMMA
  FORALL(f:[T->real],a:T,b:gt[T,a],x:open_l[T,a,b]):
   concave?(f) IMPLIES
     (f(x)-f(a))/(x-a)>=(f(b)-f(a))/(b-a)

concave_aux2: LEMMA
  FORALL(f:[T->real]):
   concave?(f) IFF
    FORALL (x:T,x1:lt[T,x],x2:ge[T,x]):
       (f(x)-f(x1))/(x-x1)>=(f(x2)-f(x1))/(x2-x1)

concave_aux3: LEMMA
  FORALL(f:[T->real]):
   concave?(f) IFF
    FORALL (x:T,x1:le[T,x],x2:gt[T,x]):
       (f(x2)-f(x))/(x2-x)<=(f(x2)-f(x1))/(x2-x1)

convex_aux: LEMMA
  FORALL(f:[T->real],a:T,b:gt[T,a],x:open_l[T,a,b]):
   convex?(f) IMPLIES
     (f(x)-f(a))/(x-a)<=(f(b)-f(a))/(b-a)
```

```
convex_aux2: LEMMA
  FORALL(f:[T->real]):
   convex?(f) IFF
    FORALL (x:T,x1:lt[T,x],x2:ge[T,x]):
       (f(x)-f(x1))/(x-x1)<=(f(x2)-f(x1))/(x2-x1)

convex_aux3: LEMMA
  FORALL(f:[T->real]):
   convex?(f) IFF
     FORALL (x:T,x1:le[T,x],x2:gt[T,x]):
       (f(x2)-f(x))/(x2-x)>=(f(x2)-f(x1))/(x2-x1)

END convexity
```

In the following theory, further properties of convex and concave functions are given. The notion of a reasonable function is formalised and two alternative definitions of a finitely inflective function are given and shown to be equivalent.

```
convexity_props[T: TYPE FROM real]: THEORY

 BEGIN

%-----------------------------------------------------------
% Since this theory deals with the differentiability of
% functions and differentiability is only defined on
% non-singleton convex sets, assumptions must be made
% that T is such a set.
%-----------------------------------------------------------
 ASSUMING

   connected_domainT : ASSUMPTION
     FORALL (x, y : T), (z : real) :
       x <= z AND z <= y IMPLIES T_pred(z)

   not_one_elementT : ASSUMPTION
     FORALL (x : T) : EXISTS (y : T) : x /= y
```

```
  ENDASSUMING

  IMPORTING
     convexity, convex_set, inv_func@deriv_eq,
     inv_func@deriv_help, mseq,
     transcendentals@continuous_functions_props_general

add(S:mseq[pred[T]], n:mbelow[S`length], P:pred[T]):
mseq[pred[T]] =
  (# length:=S`length,
     seq:= LAMBDA (m:mbelow[S`length]):
     IF m=n THEN {x:T| P(x) OR S`seq(m)(x)}
     ELSE S`seq(m) ENDIF#)

fT: TYPE = {f:[T -> real] |
  derivable(f) AND derivable(deriv(f))}
fT3: TYPE = {f:[T -> real] | derivable(f) AND
  derivable(deriv(f)) AND continuous(deriv(deriv(f)))}

 T_is_not_singleton:
   JUDGEMENT T_pred HAS_TYPE not_singleton_set[real]
 T_is_convex: JUDGEMENT T_pred HAS_TYPE convex_set[real]
 T_is_not_one_convex: JUDGEMENT T_pred HAS_TYPE
                                  not_one_convex_set[real]
 fT_subtype: LEMMA FORALL (P:not_one_convex_set[T]):
    FORALL (f:fT):
      derivable(LAMBDA (s:(P)): f(s)) AND
      derivable(deriv(LAMBDA (s:(P)): f(s)))

 fT3_subtype: LEMMA FORALL (P:not_one_convex_set[T]):
     FORALL (f:fT3):
       derivable(LAMBDA (s:(P)): f(s)) AND
       derivable(deriv(LAMBDA (s:(P)): f(s))) AND
       continuous(deriv(deriv(LAMBDA (s:(P)): f(s))))

connected_subset: LEMMA FORALL (n:T,m:{u:T|u>n}):
  FORALL (x, y: closed[T,n,m]), (z : real) :
```

```
      x <= z AND z <= y
        IMPLIES T_pred(z) and n<=z and z<=y

 not_one_subset : LEMMA FORALL (n:T,m:{u:T|u>n}):
   FORALL (x: closed[T,n,m]) :
     EXISTS (y: closed[T,n,m]) : x /= y

%-----------------------------------------------------------
% Lemmas relating differentiability and convexity/concavity
%-----------------------------------------------------------
 decreasing_deriv_aux: LEMMA
   FORALL (f:[T->real],y:T,x:gt[T,y]):
     concave?[closed[T,y,x]](LAMBDA (s:closed[T,y,x]):f(s))
     AND derivable(f,y) IMPLIES
        deriv(f,y)>=(f(x)-f(y))/(x-y)

 decreasing_deriv_aux2: LEMMA
   FORALL (f:[T->real],y:T,x:lt[T,y]):
     concave?[closed[T,x,y]](LAMBDA (s:closed[T,x,y]):f(s))
     AND derivable(f,y) IMPLIES
       deriv(f,y)<=(f(x)-f(y))/(x-y)

 decreasing_deriv: LEMMA
   FORALL (f:{f1:[T->real]|derivable(f1)},y:T,x:gt[T,y]):
   (concave?[closed[T,y,x]](LAMBDA (s:closed[T,y,x]):f(s))
   IFF
   decreasing(LAMBDA (s:closed[T,y,x]): deriv(f)(s)))

 concave_aux4: LEMMA FORALL (f:fT,y:T,x:gt[T,y]):
   (concave?[closed[T,y,x]](LAMBDA (s:closed[T,y,x]):f(s))
   IFF
   Forall (s:closed[T,y,x]): deriv(deriv(f))(s)<=0)

 increasing_deriv: LEMMA
   FORALL (f:{f1:[T->real]|derivable(f1)},y:T,x:gt[T,y]):
   (convex?[closed[T,y,x]](LAMBDA (s:closed[T,y,x]):f(s))
   IFF
```

```
    increasing(LAMBDA (s:closed[T,y,x]): deriv(f)(s)))

  convex_aux4: LEMMA FORALL (f:fT,y:T,x:gt[T,y]):
    (convex?[closed[T,y,x]](LAMBDA (s:closed[T,y,x]):f(s))
    IFF
    Forall (s:closed[T,y,x]): deriv(deriv(f))(s)>=0)

  deriv2zero(f: fT): set[T] = {u:T| deriv(deriv(f))(u) = 0}

%-----------------------------------------------------------
% Definitions for reasonable functions.
%-----------------------------------------------------------
  complete?(S:mseq[pred[T]]): bool =
    FORALL (x:T): EXISTS (n:mbelow[S`length]): (S`seq(n))(x)

  ordered?(S:mseq[pred[T]]): bool =
    FORALL (n:mbelow[S`length],m:mbelow[n]):
     FORALL (x:(S`seq(m)),y:(S`seq(n))): x<=y

  reasonable_dom?
  (f:fT3, S:mseq[nontrivial_convex_set_tcc[T]]): bool =
    IF S`length/=0 AND S`length/=1 THEN
     (FORALL (n:mbelow[S`length-1]):
      (FORALL (u:(S`seq(n)),v:(S`seq(n+1))):
         sgn(deriv(deriv(f))(u)) /= sgn(deriv(deriv(f))(v))))
    ELSE
     (FORALL (n:mbelow[S`length]):
      (FORALL (u:(S`seq(n)),v:(S`seq(n))):
         sgn(deriv(deriv(f))(u)) = sgn(deriv(deriv(f))(v))))
    ENDIF

%-----------------------------------------------------------
% Auxiliary lemmas, which are used in the proof of
% equivalence between two definitions of finitely
% inflective.
%-----------------------------------------------------------
  reasonable_dom_ordered: LEMMA
```

```
  FORALL (S:mseq[nontrivial_convex_set_tcc[T]]):
    ordered?(S) AND complete?(S) IMPLIES
      FORALL (n,m:[mbelow[S`length]]):
        FORALL (x:(S`seq(n)),y:(S`seq(m)),z:real):
          x<=z AND z<=y IMPLIES T_pred(z) AND
            EXISTS (k:nat):
            min(n,m)<=k AND k<=max(n,m) AND (S`seq(k))(z)

reasonable_dom_bounded_above: LEMMA
  FORALL (S:mseq[nontrivial_convex_set_tcc[T]]):
    ordered?(S) AND S`length/=0 IMPLIES
      FORALL (n:[mbelow[S`length-1]]):
        EXISTS (x:T): upper_bound?(x,S`seq(n))

reasonable_dom_bounded_above2: LEMMA
  FORALL (S:mseq[nontrivial_convex_set_tcc[T]]):
    ordered?(S) AND S`length/=0 IMPLIES
      FORALL (n:[mbelow[S`length-1]]):
        nonempty?[real]
         (extend[real, T, bool, FALSE](S`seq(n)))
        AND
        bounded_real_defs.bounded_above?
         (extend[real, T, bool, FALSE](S`seq(n)))

reasonable_dom_bounded_below: LEMMA
  FORALL (S:mseq[nontrivial_convex_set_tcc[T]]):
    ordered?(S) AND S`length/=0 IMPLIES
      FORALL (n:[mbelow[S`length]]):
        n/=0 IMPLIES
          EXISTS (x:T): lower_bound?(x,S`seq(n))

reasonable_dom_bounded_below2: LEMMA
  FORALL (S:mseq[nontrivial_convex_set_tcc[T]]):
    ordered?(S) AND S`length/=0 IMPLIES
      FORALL (n:[mbelow[S`length]]):
        n/=0 IMPLIES
        nonempty?[real]
```

```
        (extend[real, T, bool, FALSE](S`seq(n)))
      AND
      bounded_real_defs.bounded_below?
        (extend[real, T, bool, FALSE](S`seq(n)))

reasonable_dom_bounded: LEMMA
  FORALL (S:mseq[nontrivial_convex_set_tcc[T]]):
    ordered?(S) AND S`length/=0 IMPLIES
      FORALL (n:mbelow[S`length-1]): n/=0
        IMPLIES bounded?(S`seq(n))

glb_T_pred: LEMMA
  FORALL (S:mseq[nontrivial_convex_set_tcc[T]]):
    ordered?(S) AND S`length/=0 IMPLIES
      FORALL (n:mbelow[S`length-1]):
        T_pred(glb(S`seq(n+1)))

glb_typepred: LEMMA
  FORALL (S:mseq[nontrivial_convex_set_tcc[T]]):
    ordered?(S) AND complete?(S) AND S`length/=0 IMPLIES
      FORALL (n:mbelow[S`length-1]):
        (S`seq(n))(glb(S`seq(n+1))) OR
        (S`seq(n+1))(glb(S`seq(n+1)))

lub_T_pred: LEMMA
  FORALL (S:mseq[nontrivial_convex_set_tcc[T]]):
    ordered?(S) AND S`length/=0 IMPLIES
      FORALL (n:mbelow[S`length-1]):
        T_pred(lub(S`seq(n)))

lub_typepred: LEMMA
  FORALL (S:mseq[nontrivial_convex_set_tcc[T]]):
    ordered?(S) AND complete?(S) AND S`length/=0 IMPLIES
      FORALL (n:mbelow[S`length-1]):
        (S`seq(n))(lub(S`seq(n))) OR
        (S`seq(n+1))(lub(S`seq(n)))
```

```
reasonable_dom_bounds: LEMMA
  FORALL (S:mseq[nontrivial_convex_set_tcc[T]]):
    ordered?(S) AND complete?(S) AND S`length/=0 IMPLIES
      FORALL (n:mbelow[S`length-1]):
        glb(S`seq(n+1))=lub(S`seq(n))

reasonable_dom_lub_ordered: LEMMA
  FORALL (S:mseq[nontrivial_convex_set_tcc[T]]):
    ordered?(S) AND complete?(S) AND S`length/=0 IMPLIES
      FORALL (n:[mbelow[S`length-1]]):
        FORALL (x:(S`seq(n)),z:real):
          x<z AND z<lub(S`seq(n)) IMPLIES
            T_pred(z) AND (S`seq(n))(z)

reasonable_dom_glb_ordered: LEMMA
  FORALL (S:mseq[nontrivial_convex_set_tcc[T]]):
    ordered?(S) AND complete?(S) AND S`length/=0 IMPLIES
      FORALL (n:[mbelow[S`length-1]]):
        FORALL (y:(S`seq(n+1)),z:real):
          glb(S`seq(n+1))<z AND z<y IMPLIES
            T_pred(z) AND (S`seq(n+1))(z)

reasonable_dom_bounds_ordered: LEMMA
  FORALL (S:mseq[nontrivial_convex_set_tcc[T]]):
    ordered?(S) AND complete?(S) AND S`length/=0 IMPLIES
      FORALL (n:[mbelow[S`length-1]]):
        FORALL (m:mbelow[S`length],x:(S`seq(m))):
          (x<lub(S`seq(n)) IMPLIES m<=n)
          AND
          (lub(S`seq(n))<x IMPLIES n<m)

choose_ordered: LEMMA
  FORALL (S:mseq[nontrivial_convex_set_tcc[T]]):
    ordered?(S) AND S`length/=0 IMPLIES
    (FORALL (n:[mbelow[S`length-1]],x:(S`seq(n))):
        EXISTS (y:(S`seq(n+1))): x<y)
```

```
choose_ordered2: LEMMA
  FORALL (S:mseq[nontrivial_convex_set_tcc[T]]):
    ordered?(S) IMPLIES
    (FORALL (n:[mbelow[S`length]],x:(S`seq(n)),
     m:{u:mbelow[S`length]|n+1<u}, z:(S`seq(m))):
        EXISTS (y:(S`seq(n+1))): x<y AND y<z)

reasonable_dom_n: LEMMA
  FORALL (f:fT3, S:mseq[nontrivial_convex_set_tcc[T]]):
    reasonable_dom?(f,S) IMPLIES
      FORALL (n:mbelow[S`length]):
        FORALL (u,v:(S`seq(n))):
          sgn(deriv(deriv(f))(u))=sgn(deriv(deriv(f))(v))

reasonable: TYPE =
  {f:fT3| EXISTS (S:mseq[nontrivial_convex_set_tcc[T]]):
    complete?(S) AND ordered?(S) AND reasonable_dom?(f,S)}

r_dom(f:reasonable):
mseq[nontrivial_convex_set_tcc[T]] =
  choose({S:mseq[nontrivial_convex_set_tcc[T]]|
    complete?(S) AND ordered?(S) AND reasonable_dom?(f,S)})

r_dom_typepred: LEMMA FORALL (f:reasonable):
  complete?(r_dom(f)) AND
  ordered?(r_dom(f)) AND
  reasonable_dom?(f,r_dom(f))

r_bounds_0: LEMMA
  FORALL (f:reasonable):
    r_dom(f)`length>0 IMPLIES
      FORALL (n:mbelow[r_dom(f)`length-1]):
        deriv(deriv(f))(glb(r_dom(f)`seq(n+1)))=0
        AND
        deriv(deriv(f))(lub(r_dom(f)`seq(n)))=0

finitely_inflective: TYPE =
```

```
  {f:fT3| EXISTS (S:finseq[nontrivial_convex_set_tcc[T]]):
   complete?(S) AND ordered?(S) AND reasonable_dom?(f,S)}

finitely_inflective?(f:[T->real]): bool =
  EXISTS (S:finseq[nontrivial_convex_set_tcc[T]]):
   complete?(S) AND ordered?(S) AND reasonable_dom?(f,S)

fi_dom(f:finitely_inflective):
finseq[nontrivial_convex_set_tcc[T]] =
  choose({S:finseq[nontrivial_convex_set_tcc[T]]|
    complete?(S) AND ordered?(S) AND reasonable_dom?(f,S)})

fi_dom_typepred: LEMMA FORALL (f:finitely_inflective):
  complete?(fi_dom(f)) AND
  ordered?(fi_dom(f)) AND
  reasonable_dom?(f,fi_dom(f))

fi_dom_length_is_r_dom_aux1: LEMMA
 FORALL (f:finitely_inflective):
   1<fi_dom(f)`length AND
   (1<r_dom(f)`length OR r_dom(f)`length<0)
   IMPLIES
   lub(fi_dom(f)`seq(0)) = lub(r_dom(f)`seq(0))

fi_dom_length_is_r_dom_aux0: LEMMA
 FORALL (f:finitely_inflective,n:nat):
   n/=0 AND n<fi_dom(f)`length AND
   (r_dom(f)`length>=0 IMPLIES n<r_dom(f)`length)
   IMPLIES
   lub(fi_dom(f)`seq(n-1)) = lub(r_dom(f)`seq(n-1))

fi_dom_is_r_dom_aux: LEMMA
 FORALL (f:finitely_inflective),
 (n:mbelow[r_dom(f)`length],x:T):
   n<fi_dom(f)`length IMPLIES
   (fi_dom(f)`seq(n)(x) IFF r_dom(f)`seq(n)(x))
```

```
fi_dom_is_r_dom: LEMMA
 FORALL (f:finitely_inflective):
   fi_dom(f)=r_dom(f)

fi_bounds_0: LEMMA
  FORALL (f:finitely_inflective):
    fi_dom(f)`length>0 IMPLIES
      FORALL (n:below[fi_dom(f)`length-1]):
        deriv(deriv(f))(glb(fi_dom(f)`seq(n+1)))=0
        AND
        deriv(deriv(f))(lub(fi_dom(f)`seq(n)))=0

poi(f:reasonable): set[T] =
 {z:T| deriv(deriv(f))(z) = 0 AND
  EXISTS (x:lt[T,z],y:gt[T,z]):
   FORALL (m:open[T,x,z],n:open[T,z,y]):
     sgn(deriv(deriv(f))(m)) /= sgn(deriv(deriv(f))(n))}

finitely_inflective_alt: TYPE =
  {f:reasonable| is_finite(poi(f))}

%-----------------------------------------------------------
% More auxiliary lemmas, which are used in the proof of
% equivalence between the two definitions of finitely
% inflective functions.
%-----------------------------------------------------------
no_poi_dom: LEMMA
  FORALL (f:finitely_inflective):
    fi_dom(f)`length<=1 IFF
      (FORALL (x:T):
        deriv(deriv(f))(x)>=0)
      OR
      (FORALL (x:T):
        deriv(deriv(f))(x)<0)

empty_poi: LEMMA
  FORALL (f:finitely_inflective):
```

```
    empty?(poi(f)) IFF
      (FORALL (x:T):
        deriv(deriv(f))(x)>=0)
      OR
      (FORALL (x:T):
        deriv(deriv(f))(x)<0)

%-----------------------------------------------------------
% Lemmas showing the equivalence of the two definitions
% of finitely inflective functions.
%-----------------------------------------------------------
 fi_poi: LEMMA
   FORALL (f:finitely_inflective):
     IF fi_dom(f)`length=0 THEN poi(f) = emptyset
     ELSIF fi_dom(f)`length=1 THEN poi(f) = emptyset
     ELSE poi(f) =
          {x:T| EXISTS (n:below[fi_dom(f)`length-1]):
            x=lub(fi_dom(f)`seq(n))}
     ENDIF

 fi_poi_is_finite: LEMMA
   FORALL (f:finitely_inflective):
     is_finite(poi(f))

 fi_is_fi_alt: JUDGEMENT finitely_inflective
                 SUBTYPE_OF finitely_inflective_alt
 fi_alt_is_fi: JUDGEMENT finitely_inflective_alt
                 SUBTYPE_OF finitely_inflective

 END convexity_props
```

In the following theory, geometric properties of curves are given of functions of one variable.

```
curve_bound[T:TYPE FROM real, a:T, b:{b1:T|a<b1}]: THEORY

 BEGIN
```

```
%-----------------------------------------------------------
% Since this theory deals with the differentiability of
% functions and differentiability is only defined on
% non-singleton convex sets, assumptions must be made
% that T is such a set.
%-----------------------------------------------------------
 ASSUMING

   connected_domainT : ASSUMPTION
     FORALL (x, y : T), (z : real) :
       x <= z AND z <= y IMPLIES T_pred(z)

   not_one_elementT : ASSUMPTION
     FORALL (x : T) : EXISTS (y : T) : x /= y

 ENDASSUMING

 IMPORTING convexity_props , inv_func@deriv_eq ,
   inv_func@deriv_help

%-----------------------------------------------------------
% Definition of various types used to simplify the
% definition of lemmas. Judgements are used to relate the
% types.
%-----------------------------------------------------------
 T_is_convex_set: JUDGEMENT
   T_pred HAS_TYPE convex_set_tcc[real]
 T_is_nontrivial_set: JUDGEMENT
   T_pred HAS_TYPE nontrivial_set[real]

 AB: TYPE = {u:real|a<=u AND u<=b}

 AB_pred: nontrivial_convex_set_tcc[T] =
   {u:real|a<=u AND u<=b}

 AB_pred_is_AB: JUDGEMENT (AB_pred) SUBTYPE_OF AB
```

```
 AB_is_AB_pred: JUDGEMENT AB SUBTYPE_OF (AB_pred)


 connected_domainAB : LEMMA
   FORALL (x, y : AB), (z : real) :
     x <= z AND z <= y IMPLIES a<=z AND z<=b

  not_one_elementAB : LEMMA
    FORALL (x : AB) : EXISTS (y : AB) : x /= y

 AB_open: TYPE = {u:real| a<u AND u<b}

 AB_subtype_T : JUDGEMENT AB SUBTYPE_OF T
 AB_open_subtype_AB : JUDGEMENT AB_open SUBTYPE_OF AB
 AB_is_nontrivial_convex_set: JUDGEMENT
   AB_pred HAS_TYPE nontrivial_convex_set_tcc[real]


%-----------------------------------------------------------
% Lemmas relating lines.
%-----------------------------------------------------------
 line_intersection: LEMMA FORALL (m,c,n,d:real):
  m/=n IMPLIES EXISTS (x:real): m*x+c=n*x+d

 line_intersection2: LEMMA FORALL (m,c,n,d:real):
  m/=n IMPLIES
    FORALL (x:real):
      m*x+c=n*x+d
      IFF
      x=(d-c)/(m-n)

 line_below_line: LEMMA FORALL (m,c,n,d:real):
  IF m=n THEN
    (IF c<d THEN
        (FORALL (x:real): m*x+c<n*x+d)
     ELSIF c>d THEN
        (FORALL (x:real): m*x+c>n*x+d)
```

```
      ELSE
         (FORALL (x:real): m*x+c=n*x+d)
      ENDIF)
  ELSIF m<n THEN (EXISTS (x1:real): m*x1+c=n*x1+d
                  AND (FORALL (x:lt[real,x1]): m*x+c>n*x+d)
                  AND (FORALL (x:gt[real,x1]): m*x+c<n*x+d))
  ELSE  (EXISTS (x1:real): m*x1+c=n*x1+d
                  AND (FORALL (x:lt[real,x1]): m*x+c<n*x+d)
                  AND (FORALL (x:gt[real,x1]): m*x+c>n*x+d))
  ENDIF

 intersect_exists: LEMMA
   FORALL (f:[T->real],a,b:T,m,c:real):
    (f(a)<=m*a+c AND f(b)>m*b+c AND continuous(f))
       IMPLIES
       (IF (a<b) THEN
          EXISTS (n:closed[T,a,b]): f(n)=m*n+c
        ELSE
          EXISTS (n:closed[T,b,a]): f(n)=m*n+c
        ENDIF)

%-----------------------------------------------------------
% Lemmas relating convex/concave curves to their tangents.
%-----------------------------------------------------------
 concave_above_line: LEMMA
   FORALL (f:[T->real]):
     concave?(LAMBDA (x:AB): f(x)) IMPLIES
       (FORALL (x:AB): f(x)>=(f(b)-f(a))/(b-a)*(x-b)+f(b))

 convex_below_line: LEMMA
   FORALL (f:[T->real]):
    convex?(LAMBDA (x:AB): f(x)) IMPLIES
      (FORALL (x:AB): f(x)<=(f(b)-f(a))/(b-a)*(x-b)+f(b))

 concave_below_tang: LEMMA
   FORALL (f:[T->real],y:AB):
     derivable(f,y) AND concave?(LAMBDA (x:AB): f(x))
```

```
      IMPLIES
        (FORALL (x:AB): f(x)<=deriv(f,y)*(x-y)+f(y))

 concave_below_tang2: LEMMA
   FORALL (f:deriv_fun[T]):
     concave?(LAMBDA (x:AB): f(x)) IMPLIES
       (FORALL (x,y:AB): f(x)<=deriv(f,y)*(x-y)+f(y))

 convex_above_tang: LEMMA
   FORALL (f:[T->real],y:AB):
     derivable(f,y) AND convex?(LAMBDA (x:AB): f(x))
       IMPLIES
         (FORALL (x:AB): f(x)>=deriv(f,y)*(x-y)+f(y))

 convex_above_tang2: LEMMA
   FORALL (f:[T->real]):
     derivable(f) AND convex?(LAMBDA (x:AB): f(x))
       IMPLIES
         (FORALL (x,y:AB): f(x)>=deriv(f,y)*(x-y)+f(y))

%-----------------------------------------------------------
% Lemmas about the relationship of a concave function to
% a line. These lemmas represent the conditions used
% within the decision procedure.
%-----------------------------------------------------------
 concave_gradient: LEMMA
   FORALL (f:[T->real],m:real):
     (concave?(LAMBDA (x:AB): f(x)) AND derivable(f) AND
      continuous(deriv(f)) AND
      deriv(f,a)>=m AND deriv(f,b)<=m) IMPLIES
        (EXISTS (n:AB): deriv(f,n)=m)

 concave_gradient2: LEMMA
   FORALL
     (f:{f1:deriv_fun[T]| concave?(LAMBDA (x:AB): f1(x))
      AND continuous(deriv(f1))},m:real):
        (deriv(f,a)>=m AND deriv(f,b)<=m) IFF
```

```
        (EXISTS (n:AB): deriv(f,n)=m)

concave_curve_above_line: LEMMA
  FORALL (f:[T->real],m,c:real):
    concave?(LAMBDA (x:AB): f(x)) IMPLIES
      IF (f(a)>m*a+c AND f(b)>m*b+c)
         THEN (FORALL (x:AB): f(x)>m*x+c)
      ELSE
         (EXISTS (x:AB): f(x)<=m*x+c)
      ENDIF

concave_curve_above_line2: LEMMA
  FORALL (f:[T->real],m,c:real):
    concave?(LAMBDA (x:AB): f(x)) IMPLIES
   ((f(a)>m*a+c AND f(b)>m*b+c) IFF
    (FORALL (x:AB): f(x)>m*x+c))

concave_curve_below_line: LEMMA
  FORALL (f:[T->real],m,c:real):
    concave?(LAMBDA (x:AB): f(x)) AND derivable(f) AND
    continuous(deriv(f)) IMPLIES
      IF (deriv(f,a)<m AND f(a)<m*a+c)
        THEN (FORALL (x:AB): f(x)<m*x+c)
      ELSIF (deriv(f,b)>m AND f(b)<m*b+c)
        THEN (FORALL (x:AB): f(x)<m*x+c)
      ELSIF (EXISTS (n:AB): deriv(f,n)=m AND f(n)<m*n+c)
        THEN (FORALL (x:AB): f(x)<m*x+c)
      ELSE (EXISTS (x:AB): f(x)>=m*x+c)
      ENDIF

concave_curve_below_line2: LEMMA
  FORALL (f:[T->real],m,c:real):
    concave?(LAMBDA (x:AB): f(x)) AND derivable(f) AND
    continuous(deriv(f)) IMPLIES
      (((deriv(f,a)<m AND f(a)<m*a+c) OR
       (deriv(f,b)>m AND f(b)<m*b+c) OR
       (EXISTS (n:AB): deriv(f,n)=m AND f(n)<m*n+c))
```

```
        IFF
        (FORALL (x:AB): f(x)<m*x+c))

 concave_curve_below_line3: LEMMA
   FORALL (f:deriv_fun[T],m,c:real):
   deriv(f)(b)/=deriv(f)(a) IMPLIES
   LET p = (f(a)-deriv(f)(a)*a-f(b)+deriv(f)(b)*b)/
     (deriv(f)(b)-deriv(f)(a))
   IN
     concave?(LAMBDA (x:AB): f(x)) AND
     continuous(deriv(f)) AND
     (EXISTS (n:AB): deriv(f,n)=m) AND
     f(a)+deriv(f)(a)*(p-a)<m*p+c
       IMPLIES
         (FORALL (x:AB): f(x)<m*x+c)

 concave_intersect_line: LEMMA
   FORALL (f:[T->real],m,c:real):
     concave?(LAMBDA (x:AB): f(x)) AND derivable(f) AND
     continuous(deriv(f)) IMPLIES
       IF (f(a)>m*a+c AND f(b)>m*b+c)
         THEN (FORALL (x:AB): f(x)>m*x+c)
       ELSIF (deriv(f,a)<m AND f(a)<m*a+c)
         THEN (FORALL (x:AB): f(x)<m*x+c)
       ELSIF (deriv(f,b)>m AND f(b)<m*b+c)
         THEN (FORALL (x:AB): f(x)<m*x+c)
       ELSIF (EXISTS (x:AB): deriv(f,x)=m AND f(x)<m*x+c)
         THEN (FORALL (x:AB): f(x)<m*x+c)
       ELSE
        (EXISTS (x:AB): f(x)=m*x+c)
       ENDIF

%-----------------------------------------------------------
% Lemmas about the relationship of a convex function to
% a line. These lemmas represent the conditions used
% within the decision procedure.
%-----------------------------------------------------------
```

```
convex_gradient: LEMMA
  FORALL (f:[T->real],m:real):
    (convex?(LAMBDA (x:AB): f(x)) AND derivable(f) AND
     continuous(deriv(f)) AND
     deriv(f,a)<=m AND deriv(f,b)>=m) IMPLIES
       (EXISTS (n:AB): deriv(f,n)=m)

convex_gradient2: LEMMA
  FORALL (f:{f1:[T->real]| convex?(LAMBDA (x:AB): f1(x)) AND
  derivable(f1) AND continuous(deriv(f1))},m:real):
    (deriv(f,a)<=m AND deriv(f,b)>=m) IFF
    (EXISTS (n:AB): deriv(f,n)=m)

convex_curve_above_line: LEMMA
  FORALL (f:[T->real],m,c:real):
    convex?(LAMBDA (x:AB): f(x)) AND derivable(f) AND
    continuous(deriv(f)) IMPLIES
      IF (deriv(f,b)<m AND f(b)>m*b+c)
        THEN (FORALL (x:AB): f(x)>m*x+c)
      ELSIF (deriv(f,a)>m AND f(a)>m*a+c)
        THEN (FORALL (x:AB): f(x)>m*x+c)
      ELSIF (EXISTS (n:AB): deriv(f,n)=m AND f(n)>m*n+c)
        THEN (FORALL (x:AB): f(x)>m*x+c)
      ELSE (EXISTS (x:AB): f(x)<=m*x+c)
      ENDIF

convex_curve_above_line2: LEMMA
  FORALL (f:[T->real],m,c:real):
    convex?(LAMBDA (x:AB): f(x)) AND derivable(f) AND
    continuous(deriv(f)) IMPLIES
      (((deriv(f,b)<m AND f(b)>m*b+c) OR
        (deriv(f,a)>m AND f(a)>m*a+c) OR
        (EXISTS (n:AB): deriv(f,n)=m AND f(n)>m*n+c))
        IFF
        (FORALL (x:AB): f(x)>m*x+c))

convex_curve_above_line3: LEMMA
```

```
  FORALL (f:deriv_fun[T],m,c:real):
  LET p = (f(a)-deriv(f)(a)*a-f(b)+deriv(f)(b)*b)/
    (deriv(f)(b)-deriv(f)(a))
  IN
    convex?(LAMBDA (x:AB): f(x)) AND continuous(deriv(f))
    AND (EXISTS (n:AB): deriv(f,n)=m) AND
    f(a)+deriv(f)(a)*(p-a)>m*p+c
      IMPLIES
        (FORALL (x:AB): f(x)>m*x+c)

convex_curve_below_line: LEMMA
  FORALL (f:[T->real],m,c:real):
    convex?(LAMBDA (x:AB): f(x)) IMPLIES
      IF (f(a)<m*a+c AND f(b)<m*b+c)
        THEN (FORALL (x:AB): f(x)<m*x+c)
      ELSE
       (EXISTS (x:AB): f(x)>=m*x+c)
      ENDIF

convex_curve_below_line2: LEMMA
  FORALL (f:[T->real],m,c:real):
    convex?(LAMBDA (x:AB): f(x)) IMPLIES
      ((f(a)<m*a+c AND f(b)<m*b+c) IFF
      (FORALL (x:AB): f(x)<m*x+c))

convex_intersect_line: LEMMA
  FORALL (f:[T->real],m,c:real):
    convex?(LAMBDA (x:AB): f(x)) AND derivable(f) AND
    continuous(deriv(f)) IMPLIES
      IF (f(a)<m*a+c AND f(b)<m*b+c)
        THEN (FORALL (x:AB): f(x)<m*x+c)
      ELSIF (deriv(f,a)>m AND f(a)>m*a+c)
        THEN (FORALL (x:AB): f(x)>m*x+c)
      ELSIF (deriv(f,b)<m AND f(b)>m*b+c)
        THEN (FORALL (x:AB): f(x)>m*x+c)
      ELSIF (EXISTS (x:AB): deriv(f,x)=m AND f(x)>m*x+c)
        THEN (FORALL (x:AB): f(x)>m*x+c)
```

```
        ELSE
         (EXISTS (x:AB): f(x)=m*x+c)
        ENDIF

%-----------------------------------------------------------
% Predicates that are used in the decision procedure.
%-----------------------------------------------------------
 curve_gt_line
  (f:{u:fT[T]|convex?(LAMBDA (x:AB): u(x)) OR
   concave?(LAMBDA (x:AB): u(x))} ,m,c:real): bool =
     (convex?(LAMBDA (x:AB): f(x)) AND
        ((deriv(f,b)<m AND f(b)>m*b+c) OR
         (deriv(f,a)>m AND f(a)>m*a+c) OR
         (EXISTS (n:AB): deriv(f,n)=m AND f(n)>m*n+c)))
     OR
     (concave?(LAMBDA (x:AB): f(x)) AND
        (f(a)>m*a+c AND f(b)>m*b+c))

 curve_lt_line
  (f:{u:fT[T]|convex?(LAMBDA (x:AB): u(x)) OR
   concave?(LAMBDA (x:AB): u(x))},m,c:real): bool =
     (convex?(LAMBDA (x:AB): f(x)) AND
       (f(a)<m*a+c AND f(b)<m*b+c))
     OR
     (concave?(LAMBDA (x:AB): f(x)) AND
       ((deriv(f,a)<m AND f(a)<m*a+c) OR
        (deriv(f,b)>m AND f(b)<m*b+c) OR
        (EXISTS (n:AB): deriv(f,n)=m AND f(n)<m*n+c)))

 curve_ge_line
  (f:{u:fT[T]|convex?(LAMBDA (x:AB): u(x)) OR
   concave?(LAMBDA (x:AB): u(x))} ,m,c:real): bool =
     (convex?(LAMBDA (x:AB): f(x)) AND
        ((deriv(f,b)<m AND f(b)>=m*b+c) OR
         (deriv(f,a)>m AND f(a)>=m*a+c) OR
         (EXISTS (n:AB): deriv(f,n)=m AND f(n)>=m*n+c)))
     OR
```

```
      (concave?(LAMBDA (x:AB): f(x)) AND
         (f(a)>=m*a+c AND f(b)>=m*b+c))

 curve_le_line
  (f:{u:fT[T]|convex?(LAMBDA (x:AB): u(x)) OR
   concave?(LAMBDA (x:AB): u(x))},m,c:real): bool =
     (convex?(LAMBDA (x:AB): f(x)) AND
       (f(a)<=m*a+c AND f(b)<=m*b+c))
     OR
    (concave?(LAMBDA (x:AB): f(x)) AND
      ((deriv(f,a)<m AND f(a)<=m*a+c) OR
       (deriv(f,b)>m AND f(b)<=m*b+c) OR
       (EXISTS (n:AB): deriv(f,n)=m AND f(n)<=m*n+c)))

 curve_neq
   (f:{u:fT[T]|convex?(LAMBDA (x:AB): u(x)) OR
    concave?(LAMBDA (x:AB): u(x))},m,c:real): bool =
      curve_gt_line(f,m,c) OR curve_lt_line(f,m,c);

 curve_eq(f:fT[T],m,c:real): bool =
   (FORALL (x:T): deriv(f,x) = 0) AND
   f(a)=m*a+c;

%-----------------------------------------------------------
% Lemma asserting that if the predicates defined above
% are true for a given curve and line then the curve lies
% on the expected side of the line.
%-----------------------------------------------------------
 curve_gt_line_aux: LEMMA
   FORALL (f:{u:fT[T]|convex?(LAMBDA (x:AB): u(x)) OR
    concave?(LAMBDA (x:AB): u(x))} ,m,c:real):
      (FORALL (x:AB): f(x)>m*x+c) IFF curve_gt_line(f,m,c)

 curve_lt_line_aux: LEMMA
   FORALL (f:{u:fT[T]|convex?(LAMBDA (x:AB): u(x)) OR
    concave?(LAMBDA (x:AB): u(x))} ,m,c:real):
      (FORALL (x:AB): f(x)<m*x+c) IFF curve_lt_line(f,m,c)
```

```
curve_ge_line_aux: LEMMA
  FORALL (f:{u:fT[T]|convex?(LAMBDA (x:AB): u(x)) OR
   concave?(LAMBDA (x:AB): u(x))} ,m,c:real):
     (FORALL (x:AB): f(x)>=m*x+c) IFF curve_ge_line(f,m,c)

curve_le_line_aux: LEMMA
  FORALL (f:{u:fT[T]|convex?(LAMBDA (x:AB): u(x)) OR
   concave?(LAMBDA (x:AB): u(x))} ,m,c:real):
     (FORALL (x:AB): f(x)<=m*x+c) IFF curve_le_line(f,m,c)

curve_eq_aux: LEMMA
  FORALL (f:{u:fT[T]|convex?(LAMBDA (x:AB): u(x)) OR
   concave?(LAMBDA (x:AB): u(x))} ,m,c:real):
     (FORALL (x:AB): f(x)=m*x+c) IFF curve_eq(f,m,c)

curve_neq_aux: LEMMA
  FORALL (f:{u:fT[T]|convex?(LAMBDA (x:AB): u(x)) OR
   concave?(LAMBDA (x:AB): u(x))} ,m,c:real):
     (FORALL (x:AB): f(x)/=m*x+c) IFF curve_neq(f,m,c)

END curve_bound
```

In the following theory, geometric properties of curves are given of functions of two variables.

```
curve_bound_2D[T:TYPE FROM real, T2:TYPE FROM real]: THEORY

 BEGIN

%-----------------------------------------------------------
% Since this theory deals with the differentiability of
% functions and differentiability is only defined on
% non-singleton convex sets, assumptions must be made
% that T is such a set.
%-----------------------------------------------------------
 ASSUMING
```

```
  connected_domainT : ASSUMPTION
    FORALL (x, y : T), (z : real) :
      x <= z AND z <= y IMPLIES T_pred(z)

  not_one_elementT : ASSUMPTION
    FORALL (x : T) : EXISTS (y : T) : x /= y

  connected_domainT2 : ASSUMPTION
    FORALL (x, y : T2), (z : real) :
      x <= z AND z <= y IMPLIES T2_pred(z)

  not_one_elementT2 : ASSUMPTION
    FORALL (x : T2) : EXISTS (y : T2) : x /= y

ENDASSUMING

IMPORTING curve_bound, NRV_lib@types2

AB: TYPE from real
AB_open: TYPE from real
AB2: TYPE from real
a,b: real

%-----------------------------------------------------------
% Lemmas about the relationship of a convex function of
% two variables to a line. These lemmas make explicit
% which points should be examined to determine whether
% the function is greater than the line.
%-----------------------------------------------------------
  concave_above_line_2D1: LEMMA
   FORALL (f:[[T,T2]->real],m,n,c:real)
   (a:T, b:gt[T,a])
   (a2:[T->T2],
   b2:{u:[T->T2]| FORALL (x:closed[T,a,b]): a2(x)<=u(x)}):
    (FORALL (x:closed[T,a,b]):
      concave?(LAMBDA (y:closed[T2,a2(x),b2(x)]): f(x,y)))
```

```
      AND
      concave?(LAMBDA (z:closed[T,a,b]): f(z,a2(z))-n*a2(z))
      AND
      concave?(LAMBDA (z:closed[T,a,b]): f(z,b2(z))-n*b2(z))
      AND
      f(a,a2(a))>m*a+n*a2(a)+c AND f(a,b2(a))>m*a+n*b2(a)+c
      AND
      f(b,a2(b))>m*b+n*a2(b)+c AND f(b,b2(b))>m*b+n*b2(b)+c
      IMPLIES
       (FORALL (z:closed[T,a,b], y:closed[T2,a2(z),b2(z)]):
         f(z,y) > m*z+n*y+c)

concave_above_line_2D2: LEMMA
  FORALL (f:[[T,T2]->real],m,c:real)
  (a:T, b:gt[T,a])
  (a2:[T->T2],
  b2:{u:[T->T2]| FORALL (x:closed[T,a,b]): a2(x)<=u(x)}):
   (FORALL (x:closed[T,a,b]):
      concave?(LAMBDA (y:closed[T2,a2(x),b2(x)]): f(x,y)))
      AND
      concave?(LAMBDA (z:closed[T,a,b]): f(z,a2(z))) AND
      convex?(LAMBDA (z:closed[T,a,b]): f(z,b2(z))) AND
      derivable(LAMBDA (z:T): f(z,b2(z))) AND
      continuous(deriv(LAMBDA (z:T): f(z,b2(z)))) AND
      f(a,a2(a))>m*a+c AND f(b,a2(b))>m*b+c AND
      ((deriv(LAMBDA (y:T): f(y,b2(y)),b)<m AND
        f(b,b2(b))>m*b+c)
      OR
       (deriv(LAMBDA (y:T): f(y,b2(y)),a)>m AND
        f(a,b2(a))>m*a+c)
      OR
       (EXISTS (n:closed[T,a,b]):
          deriv(LAMBDA (y:T): f(y,b2(y)),n)=m AND
          f(n,b2(n))>m*n+c))
      IMPLIES
       (FORALL (z:closed[T,a,b], y:closed[T2,a2(z),b2(z)]):
         f(z,y) > m*z+c)
```

```
concave_above_line_2D3: LEMMA
  FORALL (f:[[T,T2]->real],m,c:real)
  (a:T, b:gt[T,a])
  (a2:[T->T2],
  b2:{u:[T->T2]| FORALL (x:closed[T,a,b]): a2(x)<=u(x)}):
   (FORALL (x:closed[T,a,b]):
     concave?(LAMBDA (y:closed[T2,a2(x),b2(x)]): f(x,y)))
     AND
     concave?(LAMBDA (z:closed[T,a,b]): f(z,b2(z))) AND
     convex?(LAMBDA (z:closed[T,a,b]): f(z,a2(z))) AND
     derivable(LAMBDA (z:T): f(z,a2(z))) AND
     continuous(deriv(LAMBDA (z:T): f(z,a2(z)))) AND
     f(a,b2(a))>m*a+c AND f(b,b2(b))>m*b+c AND
     ((deriv(LAMBDA (y:T): f(y,a2(y)),b)<m AND
       f(b,a2(b))>m*b+c)
     OR
      (deriv(LAMBDA (y:T): f(y,a2(y)),a)>m AND
       f(a,a2(a))>m*a+c)
     OR
      (EXISTS (n:closed[T,a,b]):
        deriv(LAMBDA (y:T): f(y,a2(y)),n)=m AND
        f(n,a2(n))>m*n+c))
     IMPLIES
      (FORALL (z:closed[T,a,b], y:closed[T2,a2(z),b2(z)]):
        f(z,y) > m*z+c)

concave_above_line_2D4: LEMMA
  FORALL (f:[[T,T2]->real],m,c:real)
  (a:T, b:gt[T,a])
  (a2:[T->T2],
  b2:{u:[T->T2]| FORALL (x:closed[T,a,b]): a2(x)<=u(x)}):
   (FORALL (x:closed[T,a,b]):
     concave?(LAMBDA (y:closed[T2,a2(x),b2(x)]): f(x,y)))
     AND
     convex?(LAMBDA (z:closed[T,a,b]): f(z,a2(z))) AND
     derivable(LAMBDA (z:T): f(z,a2(z))) AND
```

```
      continuous(deriv(LAMBDA (z:T): f(z,a2(z)))) AND
      convex?(LAMBDA (z:closed[T,a,b]): f(z,b2(z))) AND
      derivable(LAMBDA (z:T): f(z,b2(z))) AND
      continuous(deriv(LAMBDA (z:T): f(z,b2(z)))) AND
      (((deriv(LAMBDA (y:T): f(y,a2(y)),b)<m AND
         f(b,a2(b))>m*b+c)
        OR
        (deriv(LAMBDA (y:T): f(y,a2(y)),a)>m AND
         f(a,a2(a))>m*a+c)
        OR
        (EXISTS (n:closed[T,a,b]):
          deriv(LAMBDA (y:T): f(y,a2(y)),n)=m AND
          f(n,a2(n))>m*n+c))
       AND
       ((deriv(LAMBDA (y:T): f(y,b2(y)),b)<m AND
         f(b,b2(b))>m*b+c)
        OR
        (deriv(LAMBDA (y:T): f(y,b2(y)),a)>m AND
         f(a,b2(a))>m*a+c)
        OR
        (EXISTS (n:closed[T,a,b]):
          deriv(LAMBDA (y:T): f(y,b2(y)),n)=m AND
          f(n,b2(n))>m*n+c)))
     IMPLIES
      (FORALL (z:closed[T,a,b],y:closed[T2,a2(z),b2(z)]):
        f(z,y) > m*z+c)

concave_below_line_2D2: LEMMA
  FORALL (f:[[T,T2]->real],m,c:real)
  (a:T, b:gt[T,a])
  (a2:T2, b2:gt[T2,a2]):
    (FORALL (x:closed[T,a,b]):
     concave?(LAMBDA (y:closed[T2,a2,b2]): f(x,y)) AND
     derivable(LAMBDA (y: T2): f(x, y)) AND
     continuous(deriv(LAMBDA (y: T2): f(x, y))) AND
     deriv(LAMBDA (y:T2): f(x,y))(a2)<0) AND
    concave?(LAMBDA (z:closed[T,a,b]): f(z,a2)) AND
```

```
    derivable(LAMBDA (z:T): f(z,a2)) AND
    continuous(deriv(LAMBDA (z:T): f(z,a2)))
    IMPLIES
     ((FORALL (y:closed[T2,a2,b2],z:closed[T,a,b]):
        f(z,y) < m*z+c)
      IFF
     ((deriv(LAMBDA (y:T): f(y,a2),b)>m AND f(b,a2)<m*b+c)
      OR
      (deriv(LAMBDA (y:T): f(y,a2),a)<m AND f(a,a2)<m*a+c)
      OR
      (EXISTS (n:closed[T,a,b]):
        deriv(LAMBDA (y:T): f(y,a2),n)=m AND
        f(n,a2)<m*n+c)))

concave_below_line_2D3: LEMMA
  FORALL (f:[[T,T2]->real],m,c:real)
  (a:T, b:gt[T,a])
  (a2:T2, b2:gt[T2,a2]):
    (FORALL (x:closed[T,a,b]):
     concave?(LAMBDA (y:closed[T2,a2,b2]): f(x,y)) AND
     derivable(LAMBDA (y: T2): f(x, y)) AND
     continuous(deriv(LAMBDA (y: T2): f(x, y))) AND
     deriv(LAMBDA (y:T2): f(x,y))(b2)>0) AND
    concave?(LAMBDA (z:closed[T,a,b]): f(z,b2)) AND
    derivable(LAMBDA (z:T): f(z,b2)) AND
    continuous(deriv(LAMBDA (z:T): f(z,b2)))
    IMPLIES
     ((FORALL (y:closed[T2,a2,b2],z:closed[T,a,b]):
        f(z,y) < m*z+c)
      IFF
     ((deriv(LAMBDA (y:T): f(y,b2),b)>m AND f(b,b2)<m*b+c)
      OR
      (deriv(LAMBDA (y:T): f(y,b2),a)<m AND f(a,b2)<m*a+c)
      OR
      (EXISTS (n:closed[T,a,b]):
        deriv(LAMBDA (y:T): f(y,b2),n)=m AND
        f(n,b2)<m*n+c)))
```

```
concave_below_line_2D4: LEMMA
  FORALL (f:[[T,T2]->real],m,c:real)
  (a:T, b:gt[T,a])
  (a2:T2, b2:gt[T2,a2]):
    (FORALL (x:closed[T,a,b]):
     concave?(LAMBDA (y:closed[T2,a2,b2]): f(x,y)) AND
     derivable(LAMBDA (y: T2): f(x, y)) AND
     continuous(deriv(LAMBDA (y: T2): f(x, y))) AND
     EXISTS (p:closed[T2,a2,b2]):
      deriv(LAMBDA (y:T2): f(x,y))(p)=0 AND
       concave?(LAMBDA (z:closed[T,a,b]): f(z,p)) AND
       derivable(LAMBDA (z:T): f(z,p)) AND
       continuous(deriv(LAMBDA (z:T): f(z,p))))
    IMPLIES
     ((FORALL (y:closed[T2,a2,b2],z:closed[T,a,b]):
        f(z,y) < m*z+c)
      IFF
     (FORALL (x:closed[T,a,b]): EXISTS (p:closed[T2,a2,b2]):
      deriv(LAMBDA (y:T2): f(x,y))(p)=0 AND
      ((deriv(LAMBDA (y:T): f(y,p),b)>m AND f(b,p)<m*b+c)
      OR
      (deriv(LAMBDA (y:T): f(y,p),a)<m AND f(a,p)<m*a+c)
      OR
      (EXISTS (n:closed[T,a,b]):
        deriv(LAMBDA (y:T): f(y,p),n)=m AND
        f(n,p)<m*n+c))))

END curve_bound_2D
```

In the following theory, geometric properties of curves are given of functions of several variables.

```
%-----------------------------------------------------------
% Since we cannot express functions from an arbitrary
% number of real variables to the reals, we must use
% functions from a sequence of reals to the reals. To
```

```
% restrict the domain of the function we use a predicate
% over lists -- the desired usage is that this predicate
% restricts the ranges of each variable (possibly as a
% function of some of the other variables).
%-----------------------------------------------------------
curve_bound_multi
 [N:posnat, T: pred[finseq[real]], T1: TYPE from real,
  a:[(T)->T1], b:{u:[(T)->T1]| FORALL (x:(T)): a(x)<=u(x)}]:
 THEORY

 BEGIN

%-----------------------------------------------------------
% Since this theory deals with the differentiability of
% functions and differentiability is only defined on
% non-singleton convex sets, assumptions must be made
% that T is such a set.
%-----------------------------------------------------------
 ASSUMING

   connected_domain_x: ASSUMPTION
     FORALL (x: T1, y: T1),(z:real):
       x<=z AND z<=y IMPLIES T1_pred(z)

   not_one_element_x: ASSUMPTION
     FORALL (x: T1):
      (EXISTS (y: T1): x/=y)

 ENDASSUMING

 IMPORTING curve_bound

 a_le_b: JUDGEMENT a(p:(T)) HAS_TYPE le[T1,b(p)]
 b_ge_a: JUDGEMENT b(p:(T)) HAS_TYPE ge[T1,a(p)]

 finseq(n:nat): TYPE = {u:finseq[real]| u`length=n}
```

```
 TN: TYPE = {u:(T)| u`length=N}
 AB(p:TN): TYPE = closed[real,a(p),b(p)]

 connected_AB: JUDGEMENT AB(p:TN) SUBTYPE_OF T1
 not_one_element_AB: LEMMA
   FORALL (p:TN): a(p)/=b(p) IMPLIES
     FORALL (x:AB(p)): EXISTS (y:AB(p)): x/=y

%-----------------------------------------------------------
% These lemmas demonstate that, using the geometric
% properties of a function of several variables,
% inequalities can be reduced to problems involving
% functions of fewer variables.
%-----------------------------------------------------------
 convex_curve_below_line_multi: LEMMA
   FORALL (f:[[T1,TN]->real], m,c:real):
    (FORALL (p:TN):
      convex?[AB(p)](LAMBDA (x:AB(p)): f(x,p)))
      IMPLIES
        ((FORALL (p:TN,x:AB(p)):
            f(x,p)< m*x+c)
        IFF
        (FORALL (p:TN):
          f(a(p),p)<m*a(p)+c AND
          f(b(p),p)<m*b(p)+c))


  convex_curve_above_line_multi: LEMMA
    FORALL (f:[[T1,TN]->real],mn:finseq(N), m,c:real):
     (FORALL (p:TN):
       derivable(LAMBDA (x:T1): f(x,p)) AND
       continuous(deriv(LAMBDA (x:T1): f(x,p))) AND
       convex?[AB(p)](LAMBDA (x:AB(p)): f(x,p)))
      IMPLIES
        ((FORALL (p:TN,x:AB(p)): m*x+c < f(x,p))
         IFF
         (FORALL (p:TN):
```

```
        (EXISTS (x:AB(p)):
          deriv(LAMBDA (y:T1): f(y,p),x)=m
          AND f(x,p)>m*x+c)
       OR
        ((NOT EXISTS (x:AB(p)):
          deriv(LAMBDA (y:T1): f(y,p),x)=m) AND
         f(a(p),p)>m*a(p)+c AND
         f(b(p),p)>m*b(p)+c)))


concave_curve_above_line_multi: LEMMA
 FORALL (f:[[T1,TN]->real], mn:finseq(N), m,c:real):
  (FORALL (p:TN):
    concave?[AB(p)](LAMBDA (x:AB(p)): f(x,p)))
   IMPLIES
    ((FORALL (p:TN,x:AB(p)): f(x,p)> m*x+c)
     IFF
      (FORALL (p:TN):
        f(a(p),p)>m*a(p)+c AND
        f(b(p),p)>m*b(p)+c))


concave_curve_below_line_multi: LEMMA
  FORALL (f:[[T1,TN]->real],mn:finseq(N), m,c:real):
   (FORALL (p:TN):
     derivable(LAMBDA (x:T1): f(x,p)) AND
     continuous(deriv(LAMBDA (x:T1): f(x,p))) AND
     concave?[AB(p)](LAMBDA (x:AB(p)): f(x,p)))
    IMPLIES
     ((FORALL (p:TN,x:AB(p)): m*x+c > f(x,p))
      IFF
      (FORALL (p:TN):
        (EXISTS (x:AB(p)):
          deriv(LAMBDA (y:T1): f(y,p),x)=m
          AND f(x,p)<m*x+c)
       OR
        ((NOT EXISTS (x:AB(p)):
```

```
        deriv(LAMBDA (y:T1): f(y,p),x)=m) AND
      f(a(p),p)<m*a(p)+c AND
      f(b(p),p)<m*b(p)+c)))
```

END curve_bound_multi

In the following theory, the geometric properties of the previous theories are used to show where a finitely inflective function in one variable lies in relation to a line.

decision_proc_single [a:real, b:{u:real|a<u}]: **THEORY**

BEGIN

IMPORTING NRV_lib@types2, curve_bound, language@language

```
%-----------------------------------------------------------
% This function takes a function f, two reals m and c, and
% and element ~ of inq. This represents the inequality
% f ~ m*x+c. The inq ~ must be interpreted and the
% appropriate predicate is applied to determine whether
% the inequality holds. The function takes a natural
% number and uses this as a measure to repreatedly apply
% the appropriate predicate in intervals.
%-----------------------------------------------------------
 dp_single_aux(f:finitely_inflective[closed[real,a,b]],
   m:real, c:real, n:below[fi_dom(f)`length],s:inq):
   RECURSIVE bool =
   LET lb = glb(fi_dom(f)`seq(n)),
       ub = lub(fi_dom(f)`seq(n)) IN
   IF n = 0 THEN
     IF s=lt THEN
       curve_lt_line
         [closed[real,a,b],lb,ub]
         (f,m,c)
     ELSIF s=le THEN
       curve_le_line
         [closed[real,a,b],lb,ub]
```

```
      (f,m,c)
  ELSIF s=eq THEN
    curve_eq
      [closed[real,a,b],lb,ub]
      (f,m,c)
  ELSIF s=ge THEN
    curve_ge_line
      [closed[real,a,b],lb,ub]
      (f,m,c)
  ELSIF s=gt THEN
    curve_gt_line
      [closed[real,a,b],lb,ub]
      (f,m,c)
  ELSE
    curve_neq
      [closed[real,a,b],lb,ub]
      (f,m,c)
  ENDIF
ELSE
  (IF s=lt THEN
    curve_lt_line
       [closed[real,a,b],lb,ub]
       (f,m,c)
   ELSIF s=le THEN
     curve_le_line
       [closed[real,a,b],lb,ub]
       (f,m,c)
   ELSIF s=eq THEN
     curve_eq
       [closed[real,a,b],lb,ub]
       (f,m,c)
   ELSIF s=ge THEN
     curve_ge_line
       [closed[real,a,b],lb,ub]
       (f,m,c)
   ELSIF s=gt THEN
     curve_gt_line
```

```
          [closed[real,a,b],lb,ub]
          (f,m,c)
      ELSE
        curve_neq
          [closed[real,a,b],lb,ub]
          (f,m,c)
      ENDIF)
     AND
      dp_single_aux(f,m,c,n-1,s)
  ENDIF
 MEASURE n;

%-----------------------------------------------------------
% This function applies the above function in all
% intervals in which a function is convex or concave.
%-----------------------------------------------------------
 dp_single(f:finitely_inflective[closed[real,a,b]],
   m:real,c:real,s:inq):
   bool =
     dp_single_aux(f,m,c,fi_dom(f)`length-1,s)

END decision_proc_single
```

In the following theory, atomic formulae are formalised.

```
atomic_frmla: THEORY

 BEGIN

 IMPORTING language

 frmla: TYPE

%-----------------------------------------------------------
% Define atomic formulae
%-----------------------------------------------------------
 atomic_TCC: AXIOM
```

```
    EXISTS (x: [[P:pred[real],
    name, [(P)->real], inq]->frmla]): TRUE;
      atomic:[[P:pred[real], name, [(P)->real], inq] -> frmla]

  atomic?(A:frmla): bool =
   (EXISTS (P:pred[real], v:name, f:[(P)->real], s:inq):
     A=atomic(P,v,f,s))

  atomic_atomic?: JUDGEMENT
    atomic(P:pred[real], x:name, f:[(P)->real], s:inq)
      HAS_TYPE (atomic?)

  atomic_is_atomic?: LEMMA
    FORALL (P:pred[real], x:name, f:[(P)->real], s:inq):
      atomic?(atomic(P,x,f,s))

%-----------------------------------------------------------
% Axioms used in the definition of atomic formulae.
%-----------------------------------------------------------
  dom: [frmla -> pred[real]]
  name: [frmla -> name]
  func: [A:frmla -> [(dom(A))-> real]]
  ineq: [A:frmla -> inq]

  atomic_extensionality: AXIOM
    FORALL (A,B: (atomic?)):
      dom(A) = dom(B) AND
      name(A) = name(B) AND
      func(A) = func(B) AND
      ineq(A) = ineq(B)
        IMPLIES A = B;

  atomic_eta: AXIOM
    FORALL (A: (atomic?)):
      atomic(dom(A), name(A), func(A), ineq(A)) = A;

  dom_atomic: AXIOM
```

```
    FORALL (P:pred[real], x:name, f:[(P) -> real], s:inq):
      dom(atomic(P, x, f, s)) = P;

  name_atomic: AXIOM
    FORALL (P:pred[real], x:name, f:[(P) -> real], s:inq):
      name(atomic(P, x, f, s)) = x;

  func_atomic: AXIOM
    FORALL (P:pred[real], x:name, f:[(P) -> real], s:inq):
      func(atomic(P, x, f, s)) = f;

  ineq_atomic: AXIOM
    FORALL (P:pred[real], x:name, f:[(P) -> real], s:inq):
      ineq(atomic(P, x, f, s)) = s;

%-----------------------------------------------------------
% Define atomic formula representing true and false.
%-----------------------------------------------------------
  T: frmla = atomic({x:real| TRUE},
              char(0),
              LAMBDA (x:real): 0,
              eq)

  F: frmla = atomic({x:real| TRUE},
              char(0),
              LAMBDA (x:real): 1,
              eq)

  T(n:name): frmla = atomic({x:real| TRUE},
              n,
              LAMBDA (x:real): 0,
              eq)

  F(n:name): frmla = atomic({x:real| TRUE},
              n,
              LAMBDA (x:real): 1,
              eq)
```

```
END atomic_frmla
```

In the following theory, conjunctive and disjunctive formulae are formalised.

```
frmla: THEORY

BEGIN

  IMPORTING atomic_frmla

  frmla_seq: TYPE = {u:finseq[frmla]| 0<u`length}
  frmla_seq2: TYPE = {u:finseq[frmla]| 1<u`length}

  frmla_seq: JUDGEMENT frmla_seq2 SUBTYPE_OF frmla_seq

%-----------------------------------------------------------
% Define conjunction and disjunction as functions that
% take a sequence, which contains at least two formulae
% and constructs a formula.
%-----------------------------------------------------------
  conj:[frmla_seq2 -> frmla]
  disj:[frmla_seq2 -> frmla]

  conj?(A:frmla): bool =
    (EXISTS (B:frmla_seq2): A=conj(B))
  disj?(A:frmla): bool =
    (EXISTS (B:frmla_seq2): A=disj(B))

  conj_conj?: JUDGEMENT
    conj(A:frmla_seq2) HAS_TYPE (conj?)
  conj_conj_or_disj?: JUDGEMENT
    (conj?) SUBTYPE_OF {A:frmla| conj?(A) OR disj?(A)}
  disj_disj?: JUDGEMENT
    disj(A:frmla_seq2) HAS_TYPE (disj?)
  disj_conj_or_disj?: JUDGEMENT
    (disj?) SUBTYPE_OF {A:frmla| conj?(A) OR disj?(A)}
```

```
  args:[frmla -> finseq[frmla]]

%-----------------------------------------------------------
% Axioms used in the definition of conjunctions and
% disjunctions.
%-----------------------------------------------------------
 args_atomic: AXIOM
   FORALL (A:(atomic?)):
     args(A) = empty_seq[frmla]

 conj_extensionality: AXIOM
   FORALL (A,B: (conj?)):
     args(A) = args(B)
       IMPLIES A = B;

 args_conj: AXIOM
   FORALL (A:frmla_seq2):
     args(conj(A)) = A;

 length_conj_args: JUDGEMENT args(A:(conj?))
   HAS_TYPE frmla_seq2

 conj_eta: AXIOM
   FORALL (A: (conj?)):
     conj(args(A)) = A;

 disj_extensionality: AXIOM
   FORALL (A,B: (disj?)):
     args(A) = args(B)
       IMPLIES A = B;

 args_disj: AXIOM
   FORALL (A:frmla_seq2):
     args(disj(A)) = A;

 length_disj_args: JUDGEMENT args(A:(disj?))
```

```
    HAS_TYPE frmla_seq2

  disj_eta: AXIOM
    FORALL (A: (disj?)):
      disj(args(A)) = A;

  cd_args: JUDGEMENT
    args(A:{u:frmla| conj?(u) OR disj?(u)})
      HAS_TYPE frmla_seq2


%-----------------------------------------------------------
% Preserve important syntactic differences between atomic,
% disj, and conj. This does not affect semantic
% equivalence e.g A n T is semantically equivalent to A
% but syntactically different
%-----------------------------------------------------------
  conj_not_atomic: AXIOM
    FORALL (A:frmla_seq2): NOT atomic?(conj(A))

  conj_not_atomic_alt: LEMMA
    FORALL (A:(conj?)): NOT atomic?(A)

  atomic_not_conj: COROLLARY
    FORALL (A:(atomic?)): NOT conj?(A)

  disj_not_atomic: AXIOM
    FORALL (A:frmla_seq2): NOT atomic?(disj(A))

  disj_not_atomic_alt: COROLLARY
    FORALL (A:(disj?)): NOT atomic?(A)

  atomic_not_disj: LEMMA
    FORALL (A:(atomic?)): NOT disj?(A)

  conj_not_disj: AXIOM
    FORALL (A:frmla_seq2): NOT disj?(conj(A))
```

```
 conj_not_disj_alt: COROLLARY
   FORALL (A:(conj?)): NOT disj?(A)

 disj_not_conj: COROLLARY
   FORALL (A:frmla_seq2): NOT conj?(disj(A))

 disj_not_conj_alt: COROLLARY
   FORALL (A:(disj?)): NOT conj?(A)

 arg(A:frmla,n:below[args(A)`length]): frmla =
   args(A)`seq(n)

%-----------------------------------------------------------
% Define measure of degree for formulae.
%-----------------------------------------------------------
 deg:[frmla->nat]

 deg_args(f:finseq[frmla],g:[frmla->nat]): RECURSIVE nat =
   IF f`length = 0 THEN 0
   ELSE g(f`seq(0)) + deg_args(f^(1,f`length-1),g)
   ENDIF
   MEASURE f`length

 deg_args_alt(f:finseq[frmla],g:[frmla->nat]):
RECURSIVE nat =
   IF f`length = 0 THEN 0
   ELSIF f`length = 1 THEN g(f`seq(0))
   ELSE
     g(f`seq(0)) +
     deg_args_alt(f^(1,f`length-1),g)
   ENDIF
   MEASURE f`length

 deg_args_alt_equiv: LEMMA
   FORALL (f:finseq[frmla]):
     deg_args(f,deg)=deg_args_alt(f,deg)
```

```
deg_atomic: AXIOM
  FORALL (A:(atomic?)):
    deg(A) = 0

deg_conj: AXIOM
  FORALL (A:frmla_seq2):
    deg(conj(A)) = A`length + deg_args(A,deg)

deg_disj: AXIOM
  FORALL (A:frmla_seq2):
    deg(disj(A)) = A`length + deg_args(A,deg)

deg_cd: AXIOM
  FORALL (A:frmla):
    conj?(A) OR disj?(A) IMPLIES
    deg(A) = args(A)`length + deg_args(args(A),deg)

deg_args_o: LEMMA
  FORALL (A,B:finseq[frmla]):
    deg_args(o[frmla](A, B),deg) =
    deg_args(A,deg) + deg_args(B,deg)

deg_arg: LEMMA
  FORALL (A:frmla,n:below[args(A)`length]):
    conj?(A) OR disj?(A) IMPLIES
      deg(arg(A,n)) < deg(A)

deg_args_cd_equiv: LEMMA
  FORALL (A:frmla_seq2):
    deg(conj(A)) = deg(disj(A))

deg_args_cd: LEMMA
  FORALL (A:frmla):
    conj?(A) OR disj?(A) IMPLIES
      FORALL (m,n:below[args(A)`length]):
        1<n-m IMPLIES
```

```
          deg(conj(args(A)^(m,n-1))) < deg(A)
          AND
          deg(disj(args(A)^(m,n-1))) < deg(A)

deg_args_conj: LEMMA
  FORALL (A:frmla):
    conj?(A) IMPLIES
      FORALL (m,n:below[args(A)`length]):
        1<n-m IMPLIES
          deg(conj(args(A)^(m,n-1))) < deg(A)

deg_args_disj: LEMMA
  FORALL (A:frmla):
    disj?(A) IMPLIES
      FORALL (n,m:below[args(A)`length]):
        1<n-m IMPLIES
          deg(disj(args(A)^(m,n-1))) < deg(A)

deg_args_le: LEMMA
  FORALL (A,B:finseq[frmla]):
    0<A`length AND A`length = B`length AND
    (FORALL (n:below[A`length]):
    deg(A`seq(n)) <= deg(B`seq(n)))
      IMPLIES
        deg_args(A,deg) <= deg_args(B,deg)

deg_args_lt: LEMMA
  FORALL (A,B:finseq[frmla]):
    0<A`length AND A`length = B`length AND
    (FORALL (n:below[A`length]):
    deg(A`seq(n)) <= deg(B`seq(n))) AND
    (EXISTS (n:below[A`length]):
    deg(A`seq(n)) < deg(B`seq(n)))
      IMPLIES
        deg_args(A,deg) < deg_args(B,deg)

%-----------------------------------------------------------
```

```
% Define the predicate frmla? that represents those
% formulae that are made up from atomic, conjunctive or
% disjunctive formulae.
%-----------------------------------------------------------
 frmla?(A:frmla): RECURSIVE bool =
   IF atomic?(A) THEN TRUE
   ELSIF conj?(A) OR disj?(A) THEN
     FORALL (n:below[args(A)`length]): frmla?(arg(A,n))
   ELSE FALSE
   ENDIF
   MEASURE deg(A)

 frmla?_seq: TYPE = {u:finseq[(frmla?)]| 0<u`length}
 frmla?_seq2: TYPE = {u:finseq[(frmla?)]| 1<u`length}

 conj_frmla?: JUDGEMENT
   conj(A:frmla?_seq2) HAS_TYPE (frmla?)
 disj_frmla?: JUDGEMENT
   disj(A:frmla?_seq2) HAS_TYPE (frmla?)
 atomic_frmla?: JUDGEMENT
   (atomic?) SUBTYPE_OF (frmla?)
 frmla_frmla?_seq: JUDGEMENT
   args(A:{u:(frmla?)| conj?(u) OR disj?(u)})
   HAS_TYPE frmla?_seq2


%-----------------------------------------------------------
% Function that gets the domain of a formula. The
% function is interpreted by means of axioms
%-----------------------------------------------------------
 get_dom:[frmla -> finseq[[pred[real], name]]]

 get_dom_frmla?: AXIOM
   FORALL (A:frmla):
     IF atomic?(A) THEN
       get_dom(A) = (# length:= 1,
       seq:= LAMBDA (n:below[1]): (dom(A), name(A))#)
```

```
    ELSIF conj?(A) THEN
      IF args(A)`length=2 THEN
        get_dom(A) =
        dom_product(get_dom(arg(A,0)),
        get_dom(arg(A,1)))
      ELSE
        get_dom(A) =
        dom_product(get_dom(arg(A,0)),
        get_dom(conj(args(A)^(1,args(A)`length-1))))
      ENDIF
    ELSIF disj?(A) THEN
      IF args(A)`length=2 THEN
        get_dom(A) =
        dom_product(get_dom(arg(A,0)),
        get_dom(arg(A,1)))
      ELSE
        get_dom(A) =
        dom_product(get_dom(arg(A,0)),
        get_dom(disj(args(A)^(1,args(A)`length-1))))
      ENDIF
    ELSE TRUE
    ENDIF

deg_0: LEMMA
  FORALL (A:(frmla?)):
    deg(A) = 0 IFF atomic?(A)

deg_1: LEMMA
  FORALL (A:(frmla?)):
    deg(A) = 2 IFF
    ((conj?(A) OR disj?(A)) AND
      args(A)`length = 2 AND
      atomic?(args(A)`seq(0)) AND
      atomic?(args(A)`seq(1)))

%-----------------------------------------------------------
% Define formulae that consist only of conjunctions or
```

```
% (disjunctions) of atomic formulae.
%-----------------------------------------------------------
 conjunctive_f?(B:frmla): RECURSIVE bool =
   IF conj?(B) THEN
     FORALL (n:below[args(B)`length]):
       atomic?(arg(B,n)) OR conjunctive_f?(arg(B,n))
   ELSE FALSE
   ENDIF
   MEASURE deg(B)

 conjunctive_frmla?: JUDGEMENT
   (conjunctive_f?) SUBTYPE_OF (frmla?)

 disjunctive_f?(B:frmla): RECURSIVE bool =
   IF disj?(B) THEN
     FORALL (n:below[args(B)`length]):
       atomic?(arg(B,n)) OR disjunctive_f?(arg(B,n))
   ELSE FALSE
   ENDIF
   MEASURE deg(B)

 disjunctive_frmla?: JUDGEMENT
   (disjunctive_f?) SUBTYPE_OF (frmla?)

END frmla
```

In the following theory, quantified formulae are formalised.

```
quantification: THEORY

BEGIN

 IMPORTING frmla

%-----------------------------------------------------------
% Define universal and existential quantification as
% functions that take a name and a formula and constructs
```

```
% a formula.
%-----------------------------------------------------------
 A:[[name, frmla] -> frmla]
 E:[[name, frmla] -> frmla]

 A_n?(x:name, B:frmla): bool =
   (EXISTS (C:frmla): B=A(x,C))
 E_n?(x:name, B:frmla): bool =
   (EXISTS (C:frmla): B=E(x,C))

 A?(B:frmla): bool =
   (EXISTS (x:name, C:frmla): B=A(x,C))
 E?(B:frmla): bool =
   (EXISTS (x:name, C:frmla): B=E(x,C))

 A_A?: JUDGEMENT A(x:name, B:frmla) HAS_TYPE (A?)
 E_E?: JUDGEMENT E(x:name, B:frmla) HAS_TYPE (E?)

 arg: [frmla -> frmla]

%-----------------------------------------------------------
% Axioms used in the definition of quantification.
%-----------------------------------------------------------
 A_extensionality: AXIOM
   FORALL (B,C: (A?)):
     name(B) = name(C) AND
     arg(B) = arg(C)
      IMPLIES B = C;

 A_eta: AXIOM
   FORALL (B: (A?)):
     A(name(B),arg(B)) = B;

 name_A: AXIOM
   FORALL (x:name, B:frmla):
     name(A(x,B)) = x;
```

```
arg_A: AXIOM
  FORALL (x:name, B:frmla):
    arg(A(x,B)) = B;

E_extensionality: AXIOM
  FORALL (B,C: (E?)):
    name(B) = name(C) AND
    arg(B) = arg(C)
     IMPLIES B = C;

E_eta: AXIOM
  FORALL (B: (E?)):
    E(name(B),arg(B)) = B;

name_E: AXIOM
  FORALL (x:name, B:frmla):
    name(E(x,B)) = x;

arg_E: AXIOM
  FORALL (x:name, B:frmla):
    arg(E(x,B)) = B;


%-----------------------------------------------------------
% Preserve important syntactic differences between atomic,
% disj, conj, A and E. This does not affect semantic
% equivalence.
%-----------------------------------------------------------
A_not_atomic: AXIOM
  FORALL (x:name, B:frmla): NOT atomic?(A(x,B))

A_not_atomic_alt: LEMMA
  FORALL (B:(A?)): NOT atomic?(B)

atomic_not_A: COROLLARY
  FORALL (B:(atomic?)): NOT A?(B)
```

```
A_not_conj: AXIOM
  FORALL (x:name, B:frmla): NOT conj?(A(x,B))

A_not_conj_alt: LEMMA
  FORALL (B:(A?)): NOT conj?(B)

conj_not_A: COROLLARY
  FORALL (B:(conj?)): NOT A?(B)

A_not_disj: AXIOM
  FORALL (x:name, B:frmla): NOT disj?(A(x,B))

A_not_disj_alt: LEMMA
  FORALL (B:(A?)): NOT disj?(B)

disj_not_A: COROLLARY
  FORALL (B:(disj?)): NOT A?(B)

A_not_frmla?: LEMMA
  FORALL (x:name, B:frmla): NOT frmla?(A(x,B))

A_not_frmla?_alt: COROLLARY
  FORALL (B:(A?)): NOT frmla?(B)

frmla?_not_A: COROLLARY
  FORALL (B:(frmla?)): NOT A?(B)

E_not_atomic: AXIOM
  FORALL (x:name,B:frmla): NOT atomic?(E(x,B))

E_not_atomic_alt: COROLLARY
  FORALL (B:(E?)): NOT atomic?(B)

atomic_not_E: LEMMA
  FORALL (B:(atomic?)): NOT E?(B)

E_not_conj: AXIOM
```

```
  FORALL (x:name,B:frmla): NOT conj?(E(x,B))

E_not_conj_alt: COROLLARY
  FORALL (B:(E?)): NOT conj?(B)

conj_not_E: LEMMA
  FORALL (B:(conj?)): NOT E?(B)

E_not_disj: AXIOM
  FORALL (x:name,B:frmla): NOT disj?(E(x,B))

E_not_disj_alt: COROLLARY
  FORALL (B:(E?)): NOT disj?(B)

disj_not_E: LEMMA
  FORALL (B:(disj?)): NOT E?(B)

E_not_frmla?: LEMMA
  FORALL (x:name,B:frmla): NOT frmla?(E(x,B))

E_not_frmla?_alt: COROLLARY
  FORALL (B:(E?)): NOT frmla?(B)

frmla?_not_E: COROLLARY
  FORALL (B:(frmla?)): NOT E?(B)

A_not_E: AXIOM
  FORALL (x:name,B:frmla): NOT E?(A(x,B))

A_not_E_alt: COROLLARY
  FORALL (B:(A?)): NOT E?(B)

E_not_A: COROLLARY
  FORALL (x:name,B:frmla): NOT A?(E(x,B))

E_not_A_alt: COROLLARY
  FORALL (B:(E?)): NOT A?(B)
```

```
%-----------------------------------------------------------
% Define measure of degree for quantified formulae.
%-----------------------------------------------------------
 deg_A: AXIOM
   FORALL (x:name,B:frmla):
     deg(A(x,B)) = 1+deg(B)

 deg_E: AXIOM
   FORALL (x:name,B:frmla):
     deg(E(x,B)) = 1+deg(B)

 deg_arg_AE: LEMMA
   FORALL (B:frmla):
     A?(B) OR E?(B) IMPLIES deg(arg(B)) < deg(B)

%-----------------------------------------------------------
% Define the predicate qfrmla? that represents those
% formulae that are made up from atomic, conjunctive,
% disjunctive, universal and quantified formulae.
%-----------------------------------------------------------
 qfrmla?(B:frmla): RECURSIVE bool =
   IF atomic?(B) THEN TRUE
   ELSIF conj?(B) OR disj?(B) THEN
     FORALL (n:below[args(B)`length]): qfrmla?(arg(B,n))
   ELSIF A?(B) OR E?(B)
     THEN qfrmla?(arg(B))
   ELSE FALSE
   ENDIF
   MEASURE deg(B)

 frmla_qfrmla: JUDGEMENT (frmla?) SUBTYPE_OF (qfrmla?)

%-----------------------------------------------------------
% Axioms defining the domain of a quantified formula.
%-----------------------------------------------------------
 get_dom_qfrmla: AXIOM
```

```
  FORALL (B:frmla):
    A?(B) OR E?(B) IMPLIES
      get_dom(B) = remove(name(B), get_dom(arg(B)))

get_dom_frmla: AXIOM
  FORALL (B:frmla):
    IF atomic?(B) THEN
      get_dom(B) = (# length:= 1,
      seq:= LAMBDA (n:below[1]): (dom(B), name(B))#)
    ELSIF conj?(B) THEN
      IF args(B)`length=2 THEN
        get_dom(B) =
        dom_product(get_dom(arg(B,0)),
        get_dom(arg(B,1)))
      ELSE
        get_dom(B) =
        dom_product(get_dom(arg(B,0)),
        get_dom(conj(args(B)^(1,args(B)`length-1))))
      ENDIF
    ELSIF disj?(B) THEN
      IF args(B)`length=2 THEN
        get_dom(B) =
        dom_product(get_dom(arg(B,0)),
        get_dom(arg(B,1)))
      ELSE
        get_dom(B) =
        dom_product(get_dom(arg(B,0)),
        get_dom(disj(args(B)^(1,args(B)`length-1))))
      ENDIF
    ELSIF (A?(B) OR E?(B)) THEN
      get_dom(B) = remove(name(B), get_dom(arg(B)))
    ELSE TRUE
    ENDIF

END quantification
```

In the following theory, the concept of a *flat* formula is formalised.

```
frmla_props: THEORY

 BEGIN

 IMPORTING quantification, seq_props

%-----------------------------------------------------------
% Definitions concerning the extraction of variables and
% domains from formulae.
%-----------------------------------------------------------
 vars(A:finseq[[pred[real],name]]): TYPE =
   {u:finseq[real]|u`length=A`length AND
    FORALL (n:below[u`length]): (A`seq(n)`1)(u`seq(n))}

 dom_vars(A:frmla): TYPE =
   {u:finseq[[P:pred[real],name,(P)]]|
    u`length>=get_dom(A)`length AND
    FORALL (n:below[get_dom(A)`length]):
    EXISTS (m:below[u`length]):
      get_dom(A)`seq(n)`1 = u`seq(m)`1 AND
      get_dom(A)`seq(n)`2 = u`seq(m)`2}

 get_var(x:name,f:finseq[[P:pred[real],name,(P)]]):
  RECURSIVE real =
   IF f`length=0 THEN choose({x:real| TRUE})
   ELSIF (f`seq(0))`2=x THEN (f`seq(0))`3
   ELSE get_var(x, f^(1,f`length-1))
   ENDIF
   MEASURE f`length

%-----------------------------------------------------------
% Definitions and lemmas concerning the concept of a
% flat formula. This is important in the termination of
% various syntactic conversions, such as conversion to
% conjunctive normal form.
%-----------------------------------------------------------
 flat?(A:frmla): RECURSIVE bool =
```

```
  IF atomic?(A) THEN TRUE
  ELSIF conj?(A) THEN
    FORALL (n:below[args(A)`length]):
      (NOT conj?(arg(A, n))) AND
      flat?(arg(A, n))
  ELSIF disj?(A) THEN
   FORALL (n:below[args(A)`length]):
     (NOT disj?(arg(A, n))) AND
     flat?(arg(A, n))
  ELSIF A?(A) OR E?(A) THEN
    flat?(arg(A))
  ELSE FALSE
  ENDIF
  MEASURE deg(A)

flatten_deg: LEMMA
  FORALL (B:(qfrmla?),n:below[args(B)`length],
  P,Q:pred[frmla], f:[frmla_seq2->frmla]):
    (conj? = P OR disj? = P) AND P(B) AND
    (conj? = Q OR disj? = Q) AND Q(arg(B, n)) AND
    (f=conj OR f=disj)
     IMPLIES
       deg(f(o[frmla]
         (args(arg(B, n)),
          remove_nth[frmla](args(B), n))))
        < deg(B)

flatten_qfrmla: LEMMA
  FORALL (B:(qfrmla?),n:below[args(B)`length],
  P,Q:pred[frmla], f:[frmla_seq2->frmla]):
    (conj? = P OR disj? = P) AND P(B) AND
    (conj? = Q OR disj? = Q) AND Q(arg(B, n)) AND
    (f=conj OR f=disj)
     IMPLIES
       qfrmla?(f(o[frmla]
         (args(arg(B, n)),
          remove_nth[frmla](args(B), n))))
```

```
flatten(B:(qfrmla?)): RECURSIVE (qfrmla?) =
  IF flat?(B) THEN B
  ELSIF conj?(B) THEN
    IF EXISTS (n:below[args(B)`length]): conj?(arg(B, n))
    THEN
      LET n =
       choose({n:below[args(B)`length]| conj?(arg(B, n))})
      IN
        flatten(conj(args(arg(B, n))
        o remove_nth(args(B),n)))
    ELSE
      conj((#length:=args(B)`length,
       seq:= LAMBDA (n:below[args(B)`length]):
       flatten(arg(B, n))#))
    ENDIF
  ELSIF disj?(B) THEN
    IF EXISTS (n:below[args(B)`length]): disj?(arg(B, n))
    THEN
      LET n =
       choose({n:below[args(B)`length]| disj?(arg(B, n))})
      IN
        flatten(disj(args(arg(B, n))
        o remove_nth(args(B),n)))
    ELSE
      disj((#length:=args(B)`length,
      seq:= LAMBDA (n:below[args(B)`length]):
        flatten(arg(B,n))#))
    ENDIF
  ELSIF A?(B) THEN
    A(name(B),flatten(arg(B)))
  ELSE
    E(name(B),flatten(arg(B)))
  ENDIF
  MEASURE deg(B)
```

```
%-----------------------------------------------------------
% Judgements to show that flattening a formula does not
% change the topmost conective.
%-----------------------------------------------------------
 flatten_atomic: JUDGEMENT
   flatten(B:(atomic?)) HAS_TYPE (atomic?)
 flatten_conj: JUDGEMENT
   flatten(B:{u:(qfrmla?)|conj?(u)}) HAS_TYPE (conj?)
 flatten_disj: JUDGEMENT
   flatten(B:{u:(qfrmla?)|disj?(u)}) HAS_TYPE (disj?)
 flatten_A: JUDGEMENT
   flatten(B:{u:(qfrmla?)|A?(u)}) HAS_TYPE (A?)
 flatten_E: JUDGEMENT
   flatten(B:{u:(qfrmla?)|E?(u)}) HAS_TYPE (E?)
 conj_flatten: LEMMA
   FORALL (B:(qfrmla?)):
     conj?(flatten(B)) IMPLIES conj?(B)
 disj_flatten: LEMMA
   FORALL (B:(qfrmla?)):
     disj?(flatten(B)) IMPLIES disj?(B)

 flatten_flat: JUDGEMENT
   flatten(B:(qfrmla?)) HAS_TYPE (flat?)

 frmla_remove_nth_conj: LEMMA
   FORALL (B:(frmla?),n:below[args(B)`length]):
   conj?(B) AND conj?(arg(B,n))
     IMPLIES
       frmla?(conj(args(arg(B, n)) o
         remove_nth(args(B), n)))

 frmla_remove_nth_disj: LEMMA
   FORALL (B:(frmla?),n:below[args(B)`length]):
   disj?(B) AND disj?(arg(B,n))
     IMPLIES
     frmla?(disj(args(arg(B, n)) o
       remove_nth(args(B), n)))
```

```
 flatten_frmla: JUDGEMENT flatten(B:(frmla?))
   HAS_TYPE (frmla?)

%-----------------------------------------------------------
% Lemma concerning the degree and domain of flattened
% formulae.
%-----------------------------------------------------------
 flatten_deg2: LEMMA
   FORALL (B:(qfrmla?),n:below[args(B)`length],
   m:below[args(arg(B,n))`length],
   P,Q:pred[frmla], f:[frmla_seq2->frmla]):
     (conj? = P OR disj? = P) AND P(B) AND
     (conj? = Q OR disj? = Q) AND Q(arg(B, n)) AND
     (f=conj OR f=disj)
      IMPLIES
        deg(f((# length := args(B)`length,
          seq := LAMBDA (p: below[args(B)`length]):
            IF p = args(B)`length - 1
            THEN arg(arg(B, n),m)
            ELSE remove_nth(args(B), n)`seq(p)
            ENDIF #)))
         < deg(B)


 flatten_get_dom: LEMMA
   FORALL (B:(frmla?)):
     get_dom(B) = get_dom(flatten(B))

 deg_flatten_le: LEMMA
   FORALL (B:(qfrmla?)):
     deg(flatten(B)) <= deg(B)

 deg_flatten: LEMMA
   FORALL (B:(qfrmla?)):
     NOT flat?(B) IMPLIES
     deg(flatten(B)) < deg(B)
```

```
END frmla_props
```

In the following theory, various normal forms are formalised along with functions to convert from arbitrary forms in to these normal forms.

```
normal_forms: THEORY

 BEGIN

 IMPORTING frmla_props, seq_props


%-----------------------------------------------------------
% Rules for 'collecting' all subformulae which contain
% (or do not contain) a given name.
%-----------------------------------------------------------
  collect_x(x:name,B:finseq[frmla]):
    RECURSIVE [finseq[frmla]] =
     IF B`length = 0 THEN empty_seq[frmla]
     ELSIF contains?(x,get_dom(B`seq(0))) THEN
       (#length:=1, seq:=LAMBDA (n:below[1]): B`seq(0)#) o
       collect_x(x, B^(1, B`length-1))
     ELSE
       collect_x(x,B^(1, B`length-1))
     ENDIF
    MEASURE length(B)

  collect_non_x(x:name,B:finseq[frmla]):
    RECURSIVE [finseq[frmla]] =
     IF B`length = 0 THEN empty_seq[frmla]
     ELSIF contains?(x,get_dom(B`seq(0))) THEN
       collect_non_x(x,B^(1, B`length-1))
     ELSE
       (#length:=1, seq:=LAMBDA (n:below[1]): B`seq(0)#) o
       collect_non_x(x,B^(1, B`length-1))
     ENDIF
```

```
    MEASURE length(B)

  collecting: LEMMA
    FORALL (x:name,B:finseq[frmla]):
      collect_x(x,B)`length+collect_non_x(x,B)`length=
      B`length


%-----------------------------------------------------------
% Formalisation of rules for transfering quantifiers
% over disjunctions and conjunctions.
%-----------------------------------------------------------
 rule1_alt(B:(A?)): frmla =
   IF contains?(name(B),get_dom(arg(B))) THEN
     IF nonempty?(get_pred(name(B),get_dom(arg(B)))) THEN
      B
     ELSE
      T
     ENDIF
   ELSE arg(B)
   ENDIF

 rule1(B:(A?)): frmla =
   IF contains?(name(B),get_dom(arg(B))) AND
      nonempty?(get_pred(name(B),get_dom(arg(B)))) THEN
      B
   ELSE arg(B)
   ENDIF

 rule2_alt(B:(E?)): frmla =
   IF contains?(name(B),get_dom(arg(B))) THEN
     IF nonempty?(get_pred(name(B),get_dom(arg(B)))) THEN
      B
     ELSE
      F
     ENDIF
   ELSE arg(B)
```

```
  ENDIF

rule2(B:(E?)): frmla =
  IF contains?(name(B),get_dom(arg(B))) THEN B
  ELSE arg(B)
  ENDIF

rule3(B:{u:(A?)| conj?(arg(u))}): frmla =
  conj((#length:=args(arg(B))`length,
  seq:= LAMBDA (n:below[args(arg(B))`length]):
    A(name(B),arg(arg(B),n)) #))

rule4(B:{u:(E?)| disj?(arg(u))}): frmla =
  disj((#length:=args(arg(B))`length,
  seq:= LAMBDA (n:below[args(arg(B))`length]):
    E(name(B),arg(arg(B),n)) #))

rule5(B:{u:(A?)| disj?(arg(u))}): frmla =
  LET m = collect_non_x(name(B),args(arg(B)))`length IN
  IF m = args(arg(B))`length THEN
    disj(collect_non_x(name(B),args(arg(B))))
  ELSIF m=args(arg(B))`length-1 THEN
    disj((#length:= m+1,
    seq:= LAMBDA (n:below[m+1]):
    IF n=m THEN
      A(name(B),collect_x(name(B),args(arg(B)))`seq(0))
    ELSE
      collect_non_x(name(B),args(arg(B)))`seq(n)
    ENDIF#))
  ELSIF m = 0 THEN
    A(name(B),disj(collect_x(name(B),args(arg(B)))))
  ELSE
  disj((#length:= m+1,
    seq:= LAMBDA (n:below[m+1]):
    IF n=m THEN
      A(name(B),disj(collect_x(name(B),args(arg(B)))))
    ELSE
```

```
        collect_non_x(name(B),args(arg(B)))`seq(n)
      ENDIF#))
    ENDIF

 rule6(B:{u:(E?)| conj?(arg(u))}): frmla =
    LET m = collect_non_x(name(B),args(arg(B)))`length IN
    IF m = args(arg(B))`length THEN
      conj(collect_non_x(name(B),args(arg(B))))
    ELSIF m=args(arg(B))`length-1 THEN
      conj((#length:= m+1,
      seq:= LAMBDA (n:below[m+1]):
      IF n=m THEN
        E(name(B),collect_x(name(B),args(arg(B)))`seq(0))
      ELSE
        collect_non_x(name(B),args(arg(B)))`seq(n)
      ENDIF#))
    ELSIF m = 0 THEN
      E(name(B),conj(collect_x(name(B),args(arg(B)))))
    ELSE
    conj((#length:= m+1,
      seq:= LAMBDA (n:below[m+1]):
      IF n=m THEN
        E(name(B),conj(collect_x(name(B),args(arg(B)))))
      ELSE
        collect_non_x(name(B),args(arg(B)))`seq(n)
      ENDIF#))
    ENDIF

%-----------------------------------------------------------
% Define the concept of isolated and minimal isolated
% formulae, along with judgements concerning the
% preservation of types.
%-----------------------------------------------------------
 isolated?(B:frmla): RECURSIVE bool =
    IF qfrmla?(B) THEN
      IF atomic?(B) THEN
        TRUE
```

```
    ELSIF conj?(B) OR disj?(B) THEN
      FORALL (n:below[args(B)`length]):
        isolated?(arg(B,n))
    ELSE
      frmla?(arg(B)) AND
      contains_only?(name(B), get_dom(arg(B)))
    ENDIF
  ELSE
    FALSE
  ENDIF
  MEASURE deg(B)

isolated_qfrmla: JUDGEMENT
  (isolated?) SUBTYPE_OF (qfrmla?)
atomic_isolated: JUDGEMENT
  (atomic?) SUBTYPE_OF (isolated?)
conj_isolated: JUDGEMENT
  conj(B:{u:finseq[(isolated?)]|1<u`length})
    HAS_TYPE (isolated?)
disj_isolated: JUDGEMENT
  disj(B:{u:finseq[(isolated?)]|1<u`length})
    HAS_TYPE (isolated?)

isolated_remove_nth_conj: LEMMA
  FORALL (B:(isolated?),n:below[args(B)`length]):
  conj?(B) AND conj?(arg(B,n))
    IMPLIES
      isolated?(conj(args(arg(B, n)) o
        remove_nth(args(B), n)))

isolated_remove_nth_disj: LEMMA
  FORALL (B:(isolated?),n:below[args(B)`length]):
  disj?(B) AND disj?(arg(B,n))
    IMPLIES
    isolated?(disj(args(arg(B, n)) o
      remove_nth(args(B), n)))
```

```
 flatten_isolated: JUDGEMENT
   flatten(B:(isolated?)) HAS_TYPE (isolated?)

 min_isolated?(B:frmla): RECURSIVE bool =
   IF qfrmla?(B) THEN
     IF atomic?(B) THEN
       TRUE
     ELSIF conj?(B) OR disj?(B) THEN
       FORALL (n:below[args(B)`length]):
         min_isolated?(arg(B,n))
     ELSIF A?(B) THEN
       disjunctive_f?(arg(B)) AND
       contains_only?(name(B), get_dom(arg(B)))
     ELSE
       conjunctive_f?(arg(B)) AND
       contains_only?(name(B), get_dom(arg(B)))
     ENDIF
   ELSE
     FALSE
   ENDIF
   MEASURE deg(B)

  min_isolated_isolated: JUDGEMENT
    (min_isolated?) SUBTYPE_OF (isolated?)

%-----------------------------------------------------------
% Define basic formulae and the modified concept of
% conjunctive and disjunctive formulae.
%-----------------------------------------------------------
 basic?(B:frmla): RECURSIVE bool =
   IF atomic?(B) THEN TRUE
   ELSIF A?(B) OR E?(B) THEN
     frmla?(arg(B)) AND
     contains_only?(name(B), get_dom(arg(B)))
   ELSE FALSE
   ENDIF
   MEASURE deg(B)
```

```
basic_qfrmla: JUDGEMENT
 (basic?) SUBTYPE_OF (qfrmla?)
basic_isolated: JUDGEMENT
 (basic?) SUBTYPE_OF (isolated?)

disj_not_basic: LEMMA
  FORALL (B:(disj?)): NOT basic?(B)

conj_not_basic: LEMMA
  FORALL (B:(conj?)): NOT basic?(B)

conjunctive?(B:frmla): RECURSIVE bool =
  IF conj?(B) THEN
    FORALL (n:below[args(B)`length]):
      (basic?(arg(B,n)) OR conjunctive?(arg(B,n)))
  ELSE FALSE
  ENDIF
  MEASURE deg(B)

conjunctive_qfrmla?: JUDGEMENT
  (conjunctive?) SUBTYPE_OF (qfrmla?)
conjunctive_isolated: JUDGEMENT
  (conjunctive?) SUBTYPE_OF (isolated?)

disjunctive?(B:frmla): RECURSIVE bool =
  IF disj?(B) THEN
    FORALL (n:below[args(B)`length]):
      (basic?(arg(B,n)) OR disjunctive?(arg(B,n)))
  ELSE FALSE
  ENDIF
  MEASURE deg(B)

disjunctive_qfrmla?: JUDGEMENT
  (disjunctive?) SUBTYPE_OF (qfrmla?)
disjunctive_isolated: JUDGEMENT
  (disjunctive?) SUBTYPE_OF (isolated?)
```

```
 disj_isolated_alt: JUDGEMENT
   {u:(isolated?)| NOT basic?(u) AND
    NOT conj?(u) AND NOT disjunctive?(u)}
   SUBTYPE_OF (disj?)

%-----------------------------------------------------------
% Define conjunctive (disjunctive) normal form and
% recursive functions to convert arbitrary formulae into
% these forms.
%-----------------------------------------------------------
 conj_norm?(B:frmla): RECURSIVE bool =
   IF conj?(B) THEN
     FORALL (n:below[args(B)`length]):
       (basic?(arg(B,n)) OR
       disjunctive?(arg(B,n)) OR
       conj_norm?(arg(B,n)))
   ELSE
     FALSE
   ENDIF
   MEASURE deg(B)

 conj_norm_isolated: JUDGEMENT
   (conj_norm?) SUBTYPE_OF (isolated?)
 conjunctive_conj_norm: JUDGEMENT
   (conjunctive?) SUBTYPE_OF (conj_norm?)

 disj_norm?(B:frmla): RECURSIVE bool =
   IF disj?(B) THEN
     FORALL (n:below[args(B)`length]):
       (basic?(arg(B,n)) OR
       conjunctive?(arg(B,n)) OR
       disj_norm?(arg(B,n)))
   ELSE
     FALSE
   ENDIF
   MEASURE deg(B)
```

```
disj_norm_isolated: JUDGEMENT
  (disj_norm?) SUBTYPE_OF (isolated?)
disjunctive_disj_norm: JUDGEMENT
  (disjunctive?) SUBTYPE_OF (disj_norm?)

mcn_aux(B:(isolated?)):
RECURSIVE {B:(isolated?)|(basic?(B) OR
                disjunctive?(B) OR conj_norm?(B))} =
  IF basic?(B) OR conj_norm?(B) OR disjunctive?(B) THEN
    B
  ELSIF conj?(B) THEN
      conj((#length:=args(B)`length,
      seq:=LAMBDA (n:below[args(B)`length]):
      mcn_aux(arg(B,n))#))
  ELSE
    IF flat?(B) THEN
    LET m=
     choose({n:below[args(B)`length]|conj?(arg(B,n))}),
        ml=args(arg(B,m))`length
    IN
    conj((#length:=ml,
    seq:=LAMBDA (n:below[ml]):
     mcn_aux(
        disj((#length:=args(B)`length,
        seq:=LAMBDA (p:below[args(B)`length]):
        IF p=args(B)`length-1 THEN arg(arg(B,m),n)
        ELSE
          remove_nth(args(B),m)`seq(p)
        ENDIF#)))#))
     ELSE mcn_aux(flatten(B))
     ENDIF
  ENDIF
  MEASURE deg(B)

mcn_aux_typepred_aux: LEMMA
  FORALL (B:{u:(isolated?)| flat?(u)}):
   (basic?(B) IMPLIES
```

```
      basic?(mcn_aux(B))) AND
    (disjunctive?(B) IMPLIES
      disjunctive?(mcn_aux(B))) AND
    (NOT (basic?(B) OR disjunctive?(B)) IMPLIES
      conj?(mcn_aux(B)))

mcn_aux_typepred: LEMMA
  FORALL (B:{u:(isolated?)|flat?(u)}):
  (basic?(B) IFF
   basic?(mcn_aux(B))) AND
  (disjunctive?(B) IFF
   disjunctive?(mcn_aux(B))) AND
  (NOT (basic?(B) OR disjunctive?(B)) IFF
   conj_norm?(mcn_aux(B)))

make_conj_norm(B:{u:(isolated?)|flat?(u)}):
{u:(isolated?)|flat?(u)} =
  IF basic?(B) THEN
    conj((#length:=2,
    seq:= LAMBDA (n:below[2]):
    IF n=1 THEN T(name(B))
    ELSE B ENDIF #))
  ELSIF disjunctive?(B) THEN
    conj((#length:=args(B)`length+1,
    seq:= LAMBDA (n:below[args(B)`length+1]):
    IF n=args(B)`length THEN T
    ELSE B ENDIF #))
  ELSE mcn_aux(B)
  ENDIF

make_conj_norm: JUDGEMENT
  make_conj_norm(B:{u:(isolated?)|flat?(u)})
    HAS_TYPE (conj_norm?)


mdn_aux(B:(isolated?)):
RECURSIVE {B:(isolated?)|basic?(B) OR
```

```
                conjunctive?(B) OR disj_norm?(B)} =
  IF basic?(B) OR disj_norm?(B) OR conjunctive?(B) THEN
    B
  ELSIF disj?(B) THEN
    disj((#length:=args(B)`length,
    seq:=LAMBDA (n:below[args(B)`length]):
    mdn_aux(arg(B,n))#))
  ELSE
    IF flat?(B) THEN
    LET m=
      choose({n:below[args(B)`length]|disj?(arg(B,n))}),
        ml=args(arg(B,m))`length
    IN
    disj((#length:=ml,
    seq:=LAMBDA (n:below[ml]):
     mdn_aux(
       conj((#length:=args(B)`length,
       seq:=LAMBDA (p:below[args(B)`length]):
       IF p=args(B)`length-1 THEN arg(arg(B,m),n)
       ELSE
         remove_nth(args(B),m)`seq(p)
       ENDIF#)))#))
    ELSE mdn_aux(flatten(B))
    ENDIF
  ENDIF
  MEASURE deg(B)


mdn_aux_typepred_aux: LEMMA
 FORALL (B:{u:(isolated?)|flat?(u)}):
  (basic?(B) IMPLIES
     basic?(mdn_aux(B))) AND
  (conjunctive?(B) IMPLIES
     conjunctive?(mdn_aux(B))) AND
  (NOT (basic?(B) OR conjunctive?(B)) IMPLIES
     disj?(mdn_aux(B)))
```

```
mdn_aux_typepred: LEMMA
  FORALL (B:{u:(isolated?)|flat?(u)}):
  (basic?(B) IFF
   basic?(mdn_aux(B))) AND
  (conjunctive?(B) IFF
   conjunctive?(mdn_aux(B))) AND
  (NOT (basic?(B) OR conjunctive?(B)) IFF
   disj_norm?(mdn_aux(B)))

make_disj_norm(B:{u:(isolated?)|flat?(u)}):
{u:(isolated?)|flat?(u)} =
  IF basic?(B) THEN
    disj((#length:=2,
    seq:= LAMBDA (n:below[2]):
    IF n=1 THEN T(name(B))
    ELSE B ENDIF #))
  ELSIF conjunctive?(B) THEN
    disj((#length:=args(B)`length+1,
    seq:= LAMBDA (n:below[args(B)`length+1]):
    IF n=args(B)`length THEN T
    ELSE B ENDIF #))
  ELSE mdn_aux(B)
  ENDIF

make_disj_norm: JUDGEMENT
  make_disj_norm(B:{u:(isolated?)|flat?(u)})
    HAS_TYPE (disj_norm?)

%-----------------------------------------------------------
% Functions that apply the rules for transfering
% quantifiers. These functions are key components of the
% quantifier isolation algorithm.
%-----------------------------------------------------------
move_A_in(B:{u:(A?)| basic?(arg(u)) OR
disjunctive?(arg(u)) OR conj_norm?(arg(u))}):
(min_isolated?) =
  IF basic?(arg(B)) THEN
```

```
    rule1_alt(B)
  ELSIF conj?(arg(B)) THEN
    rule3(B)
  ELSE
    rule5(B)
  ENDIF

move_E_in(B:{u:(E?)| basic?(arg(u)) OR
conjunctive?(arg(u)) OR disj_norm?(arg(u))}):
(min_isolated?) =
  IF basic?(arg(B)) THEN
    rule2_alt(B)
  ELSIF disj?(arg(B)) THEN
    rule4(B)
  ELSE
    rule6(B)
  ENDIF
```

END normal_forms

In the following theory, (minimal) isolated formulae are formalised along with an algorithm for quantifier isolation.

quant_isolation: **THEORY**

BEGIN

IMPORTING normal_forms

```
%-----------------------------------------------------------
% Define new measure for the degree of formulae. This
% measure is required as it can not be shown the the
% quantifier isolation algorithm (defined as a recursive
% function) terminates.
%-----------------------------------------------------------
deg_qi:[frmla->nat]
```

```
deg_qi_min_isolated: AXIOM
  FORALL (A:(min_isolated?)):
    deg_qi(A) = 0

deg_qi_conj: AXIOM
  FORALL (A:frmla_seq2):
    deg_qi(conj(A)) = A`length + deg_args(A,deg_qi)

deg_qi_disj: AXIOM
  FORALL (A:frmla_seq2):
    deg_qi(disj(A)) = A`length + deg_args(A,deg_qi)

deg_qi_cd: AXIOM
  FORALL (A:frmla):
    conj?(A) OR disj?(A) IMPLIES
    deg_qi(A) = args(A)`length + deg_args(args(A),deg_qi)

deg_qi_A: AXIOM
  FORALL (A:frmla):
    A?(A) IMPLIES
    deg_qi(A) = 1 + deg_qi(arg(A))

deg_qi_E: AXIOM
  FORALL (A:frmla):
    E?(A) IMPLIES
    deg_qi(A) = 1 + deg_qi(arg(A))

deg_qi_ae: AXIOM
  FORALL (A:frmla):
    A?(A) OR E?(A) IMPLIES
    deg_qi(A) = 1 + deg_qi(arg(A))

deg_qi_min: LEMMA
  FORALL (B:(qfrmla?)):
    (conj?(B) OR disj?(B)) AND
    min_isolated?(B)
      IMPLIES
```

```
        deg_args(args(B),deg_qi) = 0

 deg_qi_arg: LEMMA
   FORALL (B:(qfrmla?)):
     (conj?(B) OR disj?(B)) AND
     NOT min_isolated?(B)
       IMPLIES
         FORALL (n:below[args(B)`length]):
         deg_qi(arg(B,n)) < deg_qi(B)

 deg_qi_arg_ae: LEMMA
   FORALL (B:(qfrmla?)):
     (A?(B) OR E?(B)) AND
     NOT min_isolated?(B)
       IMPLIES
         deg_qi(arg(B)) < deg_qi(B)

%-----------------------------------------------------------
% The algorithm for quantifier isolation is defined as a
% recursive function.
%-----------------------------------------------------------
 qi(B:(qfrmla?)): RECURSIVE (min_isolated?) =
   IF min_isolated?(B) THEN B
   ELSIF A?(B) THEN
     IF min_isolated?(arg(B)) THEN
       move_A_in(A(name(B),flatten(mcn_aux(arg(B)))))
     ELSE
       qi(A(name(B),qi(arg(B))))
     ENDIF
   ELSIF E?(B) THEN
     IF min_isolated?(arg(B)) THEN
       move_E_in(E(name(B),flatten(mdn_aux(arg(B)))))
     ELSE
       qi(E(name(B),qi(arg(B))))
     ENDIF
   ELSIF conj?(B) THEN
     conj((#length:=args(B)`length,
```

```
      seq:= LAMBDA (n:below[args(B)`length]):
      qi(arg(B,n))#))
    ELSE
      disj((#length:=args(B)`length,
      seq:= LAMBDA (n:below[args(B)`length]):
      qi(arg(B,n))#))
    ENDIF
    MEASURE deg_qi(B)

 qi_minimal: JUDGEMENT qi(B:(qfrmla?)) HAS_TYPE (min_isolated?)

END quant_isolation
```

In the following theory, the semantics of the logic $\mathcal{L}_1$ are formalised.

```
frmla_semantics: THEORY

 BEGIN

 IMPORTING frmla_props, seq_props

%-----------------------------------------------------------
% Define the type symb_table to represent a symbol table
% for use in the interpretation of a formula. The table
% is modelled as a sequence of predicate, name, variable
% tuples.
%-----------------------------------------------------------
 symb_table(A:frmla): TYPE =
   {u:finseq[[P:pred[real],name,(P)]]|
    u`length>=get_dom(A)`length AND
    FORALL (n:below[get_dom(A)`length]):
    (EXISTS (m:below[u`length]):
      get_dom(A)`seq(n)`2 = u`seq(m)`2)
    AND
    (FORALL (m:below[u`length]):
      get_dom(A)`seq(n)`2 = u`seq(m)`2
        IMPLIES
```

```
          FORALL (x:(u`seq(m)`1)):
           (get_dom(A)`seq(n)`1)(x))}

%-----------------------------------------------------------
% Define function for inserting and selecting elements of
% a symbol table.
%-----------------------------------------------------------
 insert(P:pred[real],n:name,x:(P),
 S:finseq[[Q:pred[real],name,(Q)]]):
   finseq[[P:pred[real],name,(P)]] =
   (#length := S`length,
     seq := LAMBDA (m:below[S`length]):
       IF S`seq(m)`2 = n THEN (P,n,x) ELSE S`seq(m) ENDIF#)

 get_n(x:name,f:finseq[[P:pred[real],name,(P)]]): int =
   IF (EXISTS (m:below[f`length]): x=(f`seq(m))`2) THEN
     choose({m:below[f`length]| x=f`seq(m)`2})
   ELSE
     -1
   ENDIF

 get_var(x:name,f:finseq[[P:pred[real],name,(P)]]): real =
   IF (EXISTS (m:below[f`length]): x=(f`seq(m))`2) THEN
     f`seq(get_n(x,f))`3
   ELSE
     choose({x:real| TRUE})
   ENDIF

%-----------------------------------------------------------
% Define functions for interpreting formulae.
%-----------------------------------------------------------
 interpret_atomic(A:(atomic?),x:(dom(A))): bool =
   IF ineq(A) = lt THEN
     func(A)(x) < 0
   ELSIF ineq(A) = le THEN
     func(A)(x) <= 0
   ELSIF ineq(A) = eq THEN
```

```
    func(A)(x) = 0
  ELSIF ineq(A) = ge THEN
    func(A)(x) >= 0
  ELSIF ineq(A) = gt THEN
    func(A)(x) > 0
  ELSE
    func(A)(x) /= 0
  ENDIF

interpret_aux(B:frmla,x:symb_table(B)):
RECURSIVE bool =
  IF qfrmla?(B) THEN
    IF atomic?(B) THEN
      interpret_atomic(B,get_var(name(B),x))
    ELSIF conj?(B) THEN
       IF args(B)`length=2 THEN
         interpret_aux(arg(B,0),x) AND
         interpret_aux(arg(B,1),x)
       ELSE
         interpret_aux(arg(B,0),x) AND
         interpret_aux(
           conj(args(B)^(1,args(B)`length-1)),x)
       ENDIF
     ELSIF disj?(B) THEN
       IF args(B)`length=2 THEN
         interpret_aux(arg(B,0),x) OR
         interpret_aux(arg(B,1),x)
       ELSE
         interpret_aux(arg(B,0),x) OR
         interpret_aux(
           disj(args(B)^(1,args(B)`length-1)),x)
       ENDIF
     ELSIF E?(B) THEN
       EXISTS (y:(get_pred(name(B),get_dom(arg(B))))):
         interpret_aux(arg(B), x o
         (#length:=1, seq:= LAMBDA (n:below[1]):
          (get_pred(name(B),get_dom(arg(B))),
```

```
            name(B),y)#))
      ELSE
        FORALL (y:(get_pred(name(B),get_dom(arg(B))))):
          interpret_aux(arg(B),
            insert(get_pred(name(B),get_dom(arg(B))),
              name(B),y,x))
      ENDIF
    ELSE
      FALSE
    ENDIF
 MEASURE deg(B)

 interpret(A:frmla): bool =
   FORALL (x:vars(get_dom(A))):
     interpret_aux(A,
      (#length:=x`length,
       seq:= LAMBDA (n:below[x`length]):
        (get_dom(A)`seq(n)`1,
         get_dom(A)`seq(n)`2,
         x`seq(n))#))

%-------------------------------------------------------------
% Lemmas showing that atomic formulae representing true
% and false are interpreted in the correct way.
%-------------------------------------------------------------
 T_true: LEMMA
  (FORALL (x:name):
     interpret(T(x)) = TRUE)

 F_false: LEMMA
  (FORALL (x:name):
     interpret(F(x)) = FALSE)

%-------------------------------------------------------------
% Declaring interpret as a CONVERSION means that at any
% point at which PVS expects a boolean but finds a
% formula it will apply the interpret function.
```

```
%-----------------------------------------------------------
  CONVERSION interpret

END frmla_semantics
```

In the following theory, the decision procedure is formalised.

```
decision_proc: THEORY

BEGIN

 IMPORTING normal_forms, frmla_semantics,
           curve_bound@decision_proc_single,
           curve_bound@convexity_props,
           NRV_lib@types2

 inflective_equiv: LEMMA
  FORALL (T:pred[real],a,b:(T),f:fT3[(T)]):
   (FORALL (x:real): T(x) IFF (a<=x AND x<=b)) IMPLIES
   ((EXISTS (S: finseq[nontrivial_convex_set_tcc[(T)]]):
        complete?[(T)](S) AND
         ordered?[(T)](S) AND
          reasonable_dom?[(T)](f, S)) IFF
    (EXISTS (S:
     finseq[nontrivial_convex_set_tcc[closed[real,a,b]]]):
        complete?[closed[real,a,b]](S) AND
         ordered?[closed[real,a,b]](S) AND
          reasonable_dom?[closed[real,a,b]](f, S)))

%-----------------------------------------------------------
% Classify the formulae to which the procedure applies.
%-----------------------------------------------------------
 q_atomic?(B:frmla): bool =
   IF A?(B) THEN
     atomic?(arg(B)) AND
     (EXISTS (x,y:(dom(arg(B)))):
       x<y AND
```

```
      (FORALL (z:real):
         dom(arg(B))(z) IFF (x<=z AND z<=y))) AND
      derivable(func(arg(B))) AND
      derivable(deriv(func(arg(B)))) AND
      continuous(deriv(deriv(func(arg(B))))) AND
      convexity_props[(dom(arg(B)))].
        finitely_inflective?(func(arg(B)))
  ELSIF E?(B) THEN
    atomic?(arg(B)) AND
    (EXISTS (x,y:(dom(arg(B)))):
      x<y AND
      (FORALL (z:real):
         dom(arg(B))(z) IFF (x<=z AND z<=y))) AND
      derivable(func(arg(B))) AND
      derivable(deriv(func(arg(B)))) AND
      continuous(deriv(deriv(func(arg(B))))) AND
      convexity_props[(dom(arg(B)))].
        finitely_inflective?(func(arg(B)))
  ELSE
    FALSE
  ENDIF

bound(B:(q_atomic?)): [(dom(arg(B))),(dom(arg(B)))] =
  choose({x,y:(dom(arg(B)))|
    FORALL (z:real):
      dom(arg(B))(z) IFF (x<=z AND z<=y)})

bound_ordered: LEMMA FORALL (B:(q_atomic?)):
  bound(B)`1<bound(B)`2

bound_dom: LEMMA FORALL (B:(q_atomic?)):
  FORALL (x:real): dom(arg(B))(x) IFF
    (bound(B)`1 <= x AND x <= bound(B)`2)

func_dom: LEMMA
  FORALL (B:(q_atomic?),f:[(dom(arg(B)))->real]):
    (LAMBDA
```

```
       (x: closed[real, bound(B)`1, bound(B)`2]): f(x))
      = f

 dp_frmla?(B:frmla): RECURSIVE bool =
  IF qfrmla?(B) THEN
    IF atomic?(B) THEN
      FALSE
    ELSIF conj?(B) OR disj?(B) THEN
      FORALL (n:below[args(B)`length]):
         dp_frmla?(arg(B,n))
     ELSE
       q_atomic?(B)
     ENDIF
   ELSE
     FALSE
   ENDIF
   MEASURE deg(B)

 get_dom_dp_frmla?: LEMMA
   FORALL (B:(dp_frmla?)):
     get_dom(B)=empty_seq

%-----------------------------------------------------------
% Define the decision procedure as (recursive) functions
% and show that the procedure produces the same boolean
% result as the interpretation of the formula.
%-----------------------------------------------------------
 dp_q_atomic(B:(q_atomic?)): bool =
   IF A?(B) THEN
     dp_single[bound(B)`1,bound(B)`2]
      (func(arg(B)),0,0,ineq(arg(B)))
   ELSE
     NOT dp_single[bound(B)`1,bound(B)`2]
      (func(arg(B)),0,0,neg(ineq(arg(B))))
   ENDIF

 dp_atomic_lem: LEMMA
```

```
    FORALL (B:(q_atomic?)):
      interpret(B) IFF dp_q_atomic(B)

  dp(B:(dp_frmla?)): RECURSIVE bool =
    IF conj?(B) THEN
      IF args(B)`length=2 THEN
        dp(arg(B,0)) AND
        dp(arg(B,1))
      ELSE
        dp(arg(B,0)) AND
        dp(conj(args(B)^(1,args(B)`length-1)))
      ENDIF
    ELSIF disj?(B) THEN
      IF args(B)`length=2 THEN
        dp(arg(B,0)) OR
        dp(arg(B,1))
      ELSE
        dp(arg(B,0)) OR
        dp(disj(args(B)^(1,args(B)`length-1)))
      ENDIF
   ELSE
     dp_q_atomic(B)
   ENDIF
   MEASURE deg(B)

  dp_lem: LEMMA
    FORALL (B:(dp_frmla?)):
      interpret(B) IFF dp(B)

END decision_proc
```

Appendix C

PVS Libraries used by NRV

This appendix contains main PVS theories used by NRV to produce proofs that a system meets its Nichols plot requirements.

In the following, theory arctan is defined using its Taylor series.

```
arctan: THEORY

 BEGIN

 IMPORTING derivable_inv, deriv_help, types1, types2,
           transcendentals@inv_trig

 x, y, l: var real
 k, n: var nat
 i: var int

%-----------------------------------------------------------
% The Taylor series is only defined over (-1, 1]. These
% lemmas assert that this interval does not contain only
% a single element and that it is connected. These appear
% frequently as TCC.
%-----------------------------------------------------------
 not_one_element_mod1: LEMMA
```

```
    FORALL (x: open_l[real,-1,1]):
      EXISTS (y: open_l[real,-1,1]): x /= y

  connected_domain_mod1: LEMMA
    FORALL (x, y: open_l[real,-1,1]), (z: real):
      x <= z AND z <= y IMPLIES -1 < z AND z <= 1

%-----------------------------------------------------------
% Tan is only defined over (k*pi-pi/2, k*pi+pi/2). These
% lemmas assert that this interval does not contain only
% a single element and that it is connected. These appear
% frequently as TCC.
%-----------------------------------------------------------
  not_one_element: LEMMA FORALL (k:int):
    FORALL (x: {u: cos_nz_type |
      k * pi - pi / 2 < u AND u < k * pi + pi / 2}):
      EXISTS (y: {u: cos_nz_type |
        k * pi - pi / 2 < u AND u < k * pi + pi / 2}):
          x /= y

  connected_domain: LEMMA FORALL (k:int):
    FORALL (x, y: {u: cos_nz_type |
      k * pi - pi / 2 < u AND u < k * pi + pi / 2}),
      (z: real):
        x <= z AND z <= y IMPLIES
         (FORALL (k2:int):
            NOT z = (2 * (k2 * pi) + pi) / 2) AND
              k * pi - pi / 2 < z AND
                z < pi / 2 + k * pi

%-----------------------------------------------------------
% This function defines a component of the Taylor series.
%-----------------------------------------------------------
  arctan_ser(n): real = ((-1)^n)/(2*n+1)

%-----------------------------------------------------------
% These lemmas are auxiliary lemmas required to show that
```

```
% the Taylor series converges.
%-----------------------------------------------------------
 arctan_ser_2n_1_decreasing: LEMMA
   decreasing(LAMBDA (r: nat):
     sum(0, r)(LAMBDA n:
       ((-1) ^ (1 + 2 * n)) / (3 + 4 * n) +
        (-1) ^ (2 + 2 * n) / (5 + 4 * n)))

 arctan_ser_2n_1_bounded: LEMMA
   bounded_below?[nat](LAMBDA (r: nat):
     sum(0, r)(LAMBDA n:
       ((-1) ^ (1 + 2 * n)) / (3 + 4 * n) +
        (-1) ^ (2 + 2 * n) / (5 + 4 * n)))

 arctan_ser_2n_increasing: LEMMA
   increasing((LAMBDA (r: nat):
     sum(0, r)(LAMBDA n:
       ((-1) ^ (2 * n)) / (1 + 4 * n) +
       ((-1) ^ (1 + 2 * n)) / (3 + 4 * n))))

 arctan_ser_2n_bounded: LEMMA
   bounded_above?[nat](LAMBDA (r: nat):
     sum(0, r)(LAMBDA n:
       ((-1) ^ (2 * n)) / (1 + 4 * n) +
       ((-1) ^ (1 + 2 * n)) / (3 + 4 * n)))

 arctan_at_1_conv: LEMMA
   EXISTS l:
     sums(LAMBDA n: arctan_ser(n) * (1 ^ (2*n+1)), l)

 arctan_converges: LEMMA
   FORALL (y: {z:real|(-1)<z AND z<=1}):
     EXISTS l:
       sums(LAMBDA n: arctan_ser(n) * (y ^ (2*n+1)), l)

%-----------------------------------------------------------
% This lemma shows that the Taylor series for arctan
```

```
% converges on (-1,1].
%-----------------------------------------------------------
 arctan_alt_converges: LEMMA
   FORALL (y: {z:real|(-1)<z AND z<=1}):
    summable(LAMBDA n:
      (if even?(n) then 0 else ((-1)^((n-1)/2))/n ENDIF)*y^n)

%-----------------------------------------------------------
% This function defines arctan over (-1,1].
%-----------------------------------------------------------
  arctan_mod1(y:{y1:real| -1<y1 AND y1<=1}):
    real = suminf(LAMBDA n: arctan_ser(n)*(y^(2*n+1)))

%-----------------------------------------------------------
% These functions provide an alternative definition of the
% Taylor series for arctan.
%-----------------------------------------------------------
 arctan_alt_aux: LEMMA
   FORALL (n: nat): NOT even?(n) IMPLIES
     integer_pred((n-1) / 2)

 arctan_mod1_alt: LEMMA
   FORALL (y:{y1:real| -1<y1 AND y1<=1}):
     arctan_mod1(y)=suminf(LAMBDA (n:nat):
       (if even?(n) then 0 else ((-1)^((n-1)/2))/n ENDIF)*y^n)

%-----------------------------------------------------------
% These lemmas concern the differentiablity of arctan.
% They use the alternative definintion of arctan as this
% power series is easier to reason about for this purpose.
%-----------------------------------------------------------
 atn_alt_diffs: LEMMA
   FORALL (y:{y1:real| -1<y1 AND y1<=1}):
     (LAMBDA (n:nat): diffs(
       LAMBDA (m:nat): if even?(m) then 0
         else ((-1)^((m-1)/2))/m ENDIF)(n)*y^n) =
     (LAMBDA (n:nat):
```

```
      IF even?(1 + n) THEN 0 ELSE (-1 * y^2) ^ (n/2) ENDIF)

 atn_alt_diffs_sums: LEMMA
   FORALL (x:{y1:real| -1<y1 AND y1<1}):
    sums(LAMBDA (n:nat):
      diffs(LAMBDA (m:nat): if even?(m) then 0
        else ((-1)^((m-1)/2))/m ENDIF)(n)*x^n, 1/(1+x^2))

 arctan_mod1_derivable1: LEMMA
   FORALL (x:{y1:real| -1<y1 AND y1<1}):
     derivable(arctan_mod1,x)

 arctan_mod1_derivable2: LEMMA
   derivable(arctan_mod1,1)

 arctan_mod1_derivable3: LEMMA
   derivable(arctan_mod1)
%-----------------------------------------------------------
% This lemma shows the derivative of arctan.
%-----------------------------------------------------------
 arctan_mod1_deriv1: LEMMA
   FORALL (y:{y1:real| -1<y1 AND y1<=1}):
     deriv(arctan_mod1,y)=1/(1+y^2)

%-----------------------------------------------------------
% This lemma shows that arctan is increasing on (-1,1].
%-----------------------------------------------------------
 arctan_mod1_increasing: LEMMA
   FORALL (x,y:{y1:real| -1<y1 AND y1<=1}):
     x<y IMPLIES arctan_mod1(x)<arctan_mod1(y)

%-----------------------------------------------------------
% This function defines arctan for all reals.
%-----------------------------------------------------------
 arctan(x:real): real =
   IF -1 < x AND x <= 1 THEN arctan_mod1(x)
   ELSIF x=-1 THEN -arctan_mod1(-x)
```

```
    ELSIF 1<x THEN pi/2-arctan_mod1(1/x)
    ELSE -pi/2-arctan_mod1(1/x) ENDIF

%-----------------------------------------------------------
% This function shows that arctan is positive/negative
% for positive/negative inputs. This is important when
% defining the upper and lower bounds of arctan for any
% given input.
%-----------------------------------------------------------
 arctan_sign: LEMMA
    FORALL (x:real):
      IF x<0 then arctan(x)<0
      ELSIF x>0 THEN arctan(x)>0
      ELSE arctan(x)=0 ENDIF

%-----------------------------------------------------------
% These lemmas show the relationship between arctan(x)
% and arctan(1/x).
%-----------------------------------------------------------
 arctan_periodic_pos: LEMMA
    FORALL (x:posreal): arctan(x) = pi/2-arctan(1/x)

 arctan_periodic_neg: LEMMA
    FORALL (x:negreal): arctan(x) = -pi/2-arctan(1/x)

%-----------------------------------------------------------
% This shows that the derivative of arctan at x is equal
% to the derivative at -x.
%-----------------------------------------------------------
 arctan_deriv_inv: LEMMA
    FORALL (x:nzreal):
      deriv(arctan,x)= deriv(arctan,-x)

END arctan
```

In the following theory, bounds on arctan are defined.

```
arctan_bounds: THEORY

  BEGIN

  IMPORTING arctan

  x, l: var real
  k, n: var nat
  mod1: TYPE = {y1:real| -1<y1 AND y1<=1}
  y: var mod1

 mod1_0: JUDGEMENT 0 HAS_TYPE mod1

%-----------------------------------------------------------
% The following formula is declared as a bound on arctan
% on (-1,1]. Lemmas are used to show under what
% conditions this is an upper or a lower bound.
%-----------------------------------------------------------
 arctan_mod1_bound
   (y:{y1:real| -1<y1 AND y1<=1})(n:nat) : real =
     sum(0,n)(LAMBDA (n:nat): arctan_ser(n)*y^(2*n+1))

 arctan_mod1_0: LEMMA arctan_mod1(0)=0

 arctan_mod1_bound_0: LEMMA FORALL (n:nat):
   arctan_mod1_bound(0)(n)=arctan_mod1(0)

 arctan_mod1_bound_is_lb: LEMMA
   FORALL (y:{z:nzreal | -1<z AND z<=1}, n:nat):
    ((odd?(n) and y>0) or (even?(n) and y<0))
      IMPLIES
        arctan_mod1_bound(y)(n) < arctan_mod1(y)

 arctan_mod1_bound_is_ub: LEMMA
   FORALL (y:{z:nzreal | -1<z AND z<=1}, n:nat):
    ((even?(n) and y>0) or (odd?(n) and y<0))
      IMPLIES
```

```
        arctan_mod1_bound(y)(n) > arctan_mod1(y)


%-----------------------------------------------------------
% The following formula is declared as a bound on arctan
% on the reals. A Lemma is used to show under what
% conditions this is an upper or a lower bound.
%-----------------------------------------------------------
 arctan_bound(x:real)(n:nat): real =
   IF -1 < x AND x <= 1 THEN arctan_mod1_bound(x)(n)
   ELSIF x=-1 THEN -arctan_mod1_bound(-x)(n)
   ELSIF 1<x THEN pi/2-arctan_mod1_bound(1/x)(n)
   ELSE -pi/2-arctan_mod1_bound(1/x)(n) ENDIF

 arctan_bound_is_bound: LEMMA
   FORALL (x:real,n:nat):
     IF x=0 THEN
       arctan(x)=0
     ELSIF (odd?(n) AND 1>=x AND x>0) or
           (even?(n) AND -1<=x AND x<0) THEN
       arctan_bound(x)(n) < arctan(x)
       AND
       arctan(x) < arctan_bound(x)(n+1)
     ELSIF (odd?(n) AND -1>x) or (even?(n) AND x>1) THEN
       arctan_bound(x)(n) < arctan(x)
       AND
       arctan(x) < arctan_bound(x)(n+1)
     ELSE
       arctan_bound(x)(n+1) < arctan(x)
       AND
       arctan(x) < arctan_bound(x)(n)
     ENDIF

END arctan_bounds
```

The following library was developed by Hanne Gottliebsen [54]. It has been extended to

include definitions of bounds on the exponential function.

```
transc : THEORY

 BEGIN

 IMPORTING powser, exponent_props,
           continuous_functions_props_general,
           chain_rule, more_infseries

  n, m, i, j, r: VAR nat

  a, b, c, d, l, x, y, x1, x2: VAR real

  z: VAR nzreal

%-----------------------------------------------------------
% The three functions we define by series are exp, sin, cos
%-----------------------------------------------------------
exp_ser(n): real = 1 / fac(n)

sin_ser(n): real =
  IF even?(n) THEN 0
  ELSE ((-1) ^ ((n - 1) / 2)) / fac(n) ENDIF

cos_ser(n): real =
  IF even?(n) THEN ((-1) ^ (n / 2)) / fac(n) ELSE 0 ENDIF

%-----------------------------------------------------------
% Show the series for exp converges, using the ratio test
%-----------------------------------------------------------
exp_converges: LEMMA
  FORALL x:
    EXISTS l: sums(LAMBDA n: exp_ser(n) * (x ^ n), l)

exp_converges_abs: LEMMA
  FORALL x:
```

```
    EXISTS l: sums(LAMBDA n: abs(exp_ser(n) * (x ^ n)), l)

exp(x): real = suminf(LAMBDA n: exp_ser(n) * (x ^ n))

%-------------------------------------------------------
% Show by the comparison test that sin and cos converge
%-------------------------------------------------------
sin_converges: LEMMA
  FORALL x:
    EXISTS l: sums(LAMBDA n: sin_ser(n) * (x ^ n), l)

sin(x): real = suminf(LAMBDA n: sin_ser(n) * (x ^ n))

cos_converges: LEMMA
  FORALL x:
    EXISTS l: sums(LAMBDA n: cos_ser(n) * (x ^ n), l)

cos(x): real = suminf(LAMBDA n: cos_ser(n) * (x ^ n))

%-------------------------------------------------------
% Show what the formal derivatives of these series are
%-------------------------------------------------------
exp_diff: LEMMA FORALL n: diffs(exp_ser)(n) = exp_ser(n)

sin_diff: LEMMA FORALL n: diffs(sin_ser)(n) = cos_ser(n)

cos_diff: LEMMA FORALL n: diffs(cos_ser)(n) = -sin_ser(n)

%-------------------------------------------------------------
% Now at last we can get the derivatives of exp, sin and cos
%-------------------------------------------------------------
diff_exp: LEMMA FORALL x: diffl(exp, x, exp(x))

exp_continuous: LEMMA
  FORALL x: continuous_functions[real].continuous(exp, x)

exp_convergent: LEMMA FORALL x: convergent(exp, x)
```

```
exp_limit: LEMMA FORALL x: lim(exp, x) = exp(x)

exp_convergence: LEMMA FORALL x: convergence(exp, x, exp(x))

diff_sin: LEMMA FORALL x: diffl(sin, x, cos(x))

sin_continuous: LEMMA
  FORALL x: continuous_functions[real].continuous(sin, x)

sin_convergent: LEMMA FORALL x: convergent(sin, x)

sin_limit: LEMMA FORALL x: lim(sin, x) = sin(x)

sin_convergence: LEMMA FORALL x: convergence(sin, x, sin(x))

diff_cos: LEMMA FORALL x: diffl(cos, x, -sin(x))

cos_continuous: LEMMA
  FORALL x: continuous_functions[real].continuous(cos, x)

cos_convergent: LEMMA FORALL x: convergent(cos, x)

cos_limit: LEMMA FORALL x: lim(cos, x) = cos(x)

cos_convergence: LEMMA FORALL x: convergence(cos, x, cos(x))

%----------------------------------------
% Properties of the exponential function
%----------------------------------------
exp_0_lemma: LEMMA sums(LAMBDA n: exp_ser(n) * (0 ^ n), 1)

exp_0: LEMMA exp(0) = 1

exp_le_x_lemma: LEMMA
  FORALL x, a:
    (0 <= x AND sums(LAMBDA n: exp_ser(n) * (x ^ n), a))
```

```
      IMPLIES
        (1 + x) <= a

exp_le_x: LEMMA FORALL x: 0 <= x IMPLIES (1 + x) <= exp(x)

exp_le_1_lemma: LEMMA
  FORALL x, a:
    (0 < x AND sums(LAMBDA n: exp_ser(n) * (x ^ n), a))
      IMPLIES 1 < a

exp_le_1: LEMMA FORALL x: 0 < x IMPLIES 1 < exp(x)

binomial_formula: LEMMA
  FORALL x, y, n:
    y /= 0 IMPLIES
      exp_ser(n) * ((x + y) ^ n) =
      exp_ser(n) *
        sum(0, n)
          (LAMBDA m: bin_q(n, m) * (x ^ m) * (y ^ (n - m)))

exp_add_mul_lemma1: LEMMA
  FORALL x, a, b:
    diff1(LAMBDA y: exp(a + b + y) * exp(-y), x, 0)

exp_add_mul_lemma2: LEMMA
  FORALL x, a, b:
    (LAMBDA y: exp(a + b + y) * exp(-y))(x) = exp(a + b)

exp_add_mul: LEMMA
  FORALL x, y: exp(x + y) * exp(-x) = exp(y)

exp_neg_mul: LEMMA FORALL x: exp(x) * exp(-x) = 1

exp_neg_mul2: LEMMA FORALL x: exp(-x) * exp(x) = 1

exp_add: LEMMA FORALL x, y: exp(x + y) = exp(x) * exp(y)
```

```
  exp_pos_le: LEMMA FORALL x: 0 <= exp(x)

  exp_nz: LEMMA FORALL x: exp(x) /= 0

  exp_neg: LEMMA FORALL x: exp(-x) = 1 / exp(x)

  exp_pos_lt: LEMMA FORALL x: 0 < exp(x)

  exp_pos: JUDGEMENT exp(x:real) HAS_TYPE posreal

  exp_n: LEMMA FORALL x, n: exp(n * x) = exp(x) ^ n

  exp_sub: LEMMA FORALL x, y: exp(x) / exp(y) = exp(x - y)

  exp_mono_imp: LEMMA FORALL x, y: x < y IMPLIES exp(x) < exp(y)

  exp_mono_lt: LEMMA FORALL x, y: exp(x) < exp(y) IFF x < y

  exp_mono_le: LEMMA FORALL x, y: exp(x) <= exp(y) IFF x <= y

  exp_inj: LEMMA FORALL x, y: exp(x) = exp(y) IFF x = y

  exp_total_lemma: LEMMA
    FORALL y:
      1 <= y IMPLIES
        (EXISTS x: 0 <= x AND x <= y - 1 AND exp(x) = y)

  exp_total: LEMMA
    FORALL y: 0 < y IMPLIES (EXISTS x: exp(x) = y)

%-----------------------------------------------------
% The following definitions and lemmas were added by
% R Hardy and define bounds on exp.
%-----------------------------------------------------

  exp_neg_lb(x:negreal)(n:nat): nnreal =
    max(0,sum(0,n)(LAMBDA n: exp_ser(2*n) * (x ^ (2*n))
```

```
                    + exp_ser(2*n+1) * (x ^ (2*n+1))))

exp_neg_ub(x:negreal)(n:nat): real =
  1 + sum(0,n)(LAMBDA n: exp_ser(2*n+1) * (x ^ (2*n+1))
                       + exp_ser(2*n+2) * (x ^ (2*n+2)))

exp_neg_lb_aux: LEMMA
  FORALL (x:negreal,n:nat):
    (-x<1+2*n IFF
     0<exp_ser(2*n) * (x ^ (2*n)) +
        exp_ser(2*n+1) * (x ^ (2*n+1)))

exp_neg_lb_aux2: LEMMA
  FORALL (x:negreal,n:nat):
    (-x=1+2*n IFF
     0=exp_ser(2*n) * (x ^ (2*n)) +
        exp_ser(2*n+1) * (x ^ (2*n+1)))

exp_neg_ub_aux: LEMMA
  FORALL (x:negreal,n:nat):
    (-x<2+2*n IFF
     0>exp_ser(2*n+1) * (x ^ (2*n+1)) +
        exp_ser(2*n+2) * (x ^ (2*n+2)))

exp_neg_ub_aux2: LEMMA
  FORALL (x:negreal,n:nat):
    (-x=2+2*n IFF
     0=exp_ser(2*n+1) * (x ^ (2*n+1)) +
        exp_ser(2*n+2) * (x ^ (2*n+2)))

exp_neg_bound_is_bound: LEMMA
  FORALL (x:negreal,n:nat):
    (-x<2+2*n IMPLIES exp(x)<exp_neg_ub(x)(n))
    AND
    (-x<1+2*n IMPLIES exp_neg_lb(x)(n)<exp(x))

exp_pos_bound_is_bound: LEMMA
```

```
    FORALL (x:posreal,n:nat):
      (x<1+2*n AND exp_neg_lb(-x)(n)/=0 IMPLIES
         exp(x)<1/exp_neg_lb(-x)(n))
      AND
      exp_neg_lb(x)(n)<exp(x)

  exp_bound_is_bound: LEMMA
    FORALL (x:real,n:nat):
      IF x<0 AND -x<1+2*n THEN
        exp(x)<exp_neg_ub(x)(n)
        AND
        exp_neg_lb(x)(n)<exp(x)
      ELSIF x>0 AND x<1+2*n AND exp_neg_lb(-x)(n)/=0 THEN
        exp(x)<1/exp_neg_lb(-x)(n)
        AND
        exp_neg_lb(x)(n)<exp(x)
      ELSIF x>0 THEN
        exp_neg_lb(x)(n)<exp(x)
      ELSIF x=0 THEN
        exp(x)=1
      ELSE
        true
      ENDIF

  END transc
```

In the following theory, alternative definitions are given for several recursive functions previously defined in PVS. These alternative definitions allow more efficient proofs and are used to increase the efficiency of NRV. The new definitions are shown to have equivalent meaning to the original, mathematically cleaner functions.

```
NRV_exp: THEORY

BEGIN

  IMPORTING types1, types2, arctan_bounds,
    transcendentals@pi_prop
```

```
%-----------------------------------------------------------
% Useful judgements about the type (-1,1]
%-----------------------------------------------------------
 nzreal_mod1: JUDGEMENT
   /(y:closed[real,1,1], x:gt[nzreal,1]) HAS_TYPE
   open[nzreal,-1,1]

 nzreal_mod1_2: JUDGEMENT
   /(y:closed[real,1,1], x:lt[nzreal,-1]) HAS_TYPE
   open[nzreal,-1,1]

%-----------------------------------------------------------
% An alternative definition of sumc that can be expanded
% significantly faster and thus increase the efficiency
% of proofs. A lemma shows that this and the original
% definition have the same meaning.
%-----------------------------------------------------------
 sumc(n:nat, m:nat, f:[nat->real]): RECURSIVE real =
   IF m-n >= 5 THEN
     sumc(n, m-5, f) + f(m-4) + f(m-3) +
     f(m-2) + f(m-1) + f(m)
   ELSIF m < n THEN
     0
   ELSIF m = n THEN
     f(m)
   ELSE
    sumc(n, m-1, f) + f(m)
   ENDIF
    MEASURE m

 sumc_equiv: LEMMA
   FORALL(n:nat, m:nat, f:[nat->real]):
    sumc(n,m,f) = series.sumc(n,m,f)

%-----------------------------------------------------------
% An alternative definition of expt that can
```

```
% be used to increase the efficiency of proof.
% A lemma shows that this and the original
% definition have the same meaning.
%-----------------------------------------------------------
 expt(x:real,n:nat): RECURSIVE real =
   IF n>=10 THEN
     x * (x * (x * (x * (x * (x * (x * (x * (x * (x *
     expt(x,n-10)))))))))
   ELSIF n>5 THEN
     x * (x * (x * (x * (x * expt(x,n-5)))))
   ELSIF n=5 THEN
     x * (x * (x * (x * x)))
   ELSIF n = 1 THEN
     x
   ELSIF n = 0 THEN
     1
   ELSE
     x * expt(x, n-1)
   ENDIF
 MEASURE n

 expt_equiv: LEMMA
   FORALL (x:real,n:nat):
     expt(x,n) = exponentiation.expt(x,n)

 expt_pos: JUDGEMENT
   expt(x:nzreal,n:{u:nat|even?(u)}) HAS_TYPE posreal

 expt_pos2: JUDGEMENT
   expt(x:posreal,n:nat) HAS_TYPE posreal

 expt_neg: JUDGEMENT
   expt(x:negreal,n:{u:nat|odd?(u)}) HAS_TYPE negreal

%-----------------------------------------------------------
% An alternative definition of factorial that
% can be used to increase the efficiency of proof.
```

```
% A lemma shows that this and the original
% definition have the same meaning.
%-----------------------------------------------------------
 fac(n:nat): RECURSIVE posnat =
   IF n>=5 THEN
     n * ((n-1) * ((n-2) * ((n-3) * ((n-4) * fac(n-5)))))
   ELSIF n = 1 THEN
     1
   ELSIF n = 0 THEN
     1
   ELSE
     n*fac(n-1)
   ENDIF
 MEASURE n

 fac_equiv: LEMMA
   FORALL (n:nat):
     fac(n) = aux.fac(n)

%-----------------------------------------------------------
% Alternative definitions of the bounds on exp are given.
% These take two values, which are intended to represent
% the numerator and denominator of the input to exp. If
% these values are integers then these definitions are
% significantly more efficient (and clearer) to expand -
% PVS will not automatically simplify 2/3*2/3 this can
% lead to incredibly long and difficult to interpret
% proof statements; however, PVS will simplify (2*2)/(3*3)
% to 4/6.
% sgn and abs are used for the same reason - PVS will not
% automatically simplify, for instance -2*-3, it will
% however simplify -(2*3) to -6.
% Lemmas show that these and the original
% definitions have the same meanings.
%-----------------------------------------------------------
 exp_neg_lb(x:real,y:{u:nzreal| x/u<0})(n:nat): nnreal =
   max(0,sumc(0,n, LAMBDA (n:nat):
```

```
    (expt(abs(x)*abs(x), n)*((2*n+1)*y + x))/
    (expt(abs(y)*abs(x), n)*y*fac(2*n+1))))

exp_neg_ub(x:real,y:{u:nzreal| x/u<0})(n:nat): real =
  1 + sumc(0,n, LAMBDA (n:nat):
    (x*expt(abs(x)*abs(x), n)*((2*n+2)*y + x))/
    (y*expt(abs(y)*abs(y), n)*y*fac(2*n+2)))

exp_neg_lb_equiv: LEMMA
  FORALL (x:real,y:nzreal):
   (LAMBDA (n:nat):
    (expt(abs(x)*abs(x), n)*((2*n+1)*y + x))/
    (expt(abs(y)*abs(y), n)*y*fac(2*n+1)))
    =
   (LAMBDA (n:nat): exp_ser(2*n) * ((x/y) ^ (2*n)) +
                    exp_ser(2*n+1) * ((x/y) ^ (2*n+1)))

exp_neg_ub_equiv: LEMMA
  FORALL (x:real,y:nzreal):
   (LAMBDA (n:nat):
    (x*expt(abs(x)*abs(x), n)*((2*n+2)*y + x))/
    (y*expt(abs(y)*abs(y), n)*y*fac(2*n+2)))
    =
   (LAMBDA (n:nat): exp_ser(2*n+1) * ((x/y) ^ (2*n+1)) +
                    exp_ser(2*n+2) * ((x/y) ^ (2*n+2)))

exp_lb(n:nat)(x:real,y:nzreal): nnreal =
  IF x = 0 THEN 1
  ELSE
    max(0,sumc(0,n, LAMBDA (n:nat):
      (expt(abs(x)*abs(x), n)*((2*n+1)*y + x))/
      (expt(abs(y)*abs(y), n)*y*fac(2*n+1))))
  ENDIF

exp_ub(n:nat)(x:real,y:{u:nzreal| x/u>0 implies
 exp_neg_lb(-x,u)(n)/=0}): real =
  IF x = 0 THEN 1
```

```
  ELSIF x/y<0 THEN
    1 + sumc(0,n, LAMBDA (n:nat):
    (x*expt(abs(x)*abs(x), n)*((2*n+2)*y + x))/
    (y*expt(abs(y)*abs(y), n)*y*fac(2*n+2)))
  ELSE
    1/exp_neg_lb(-x,y)(n)
  ENDIF

exp_lb_aux: LEMMA
  FORALL (x:real,y:nzreal,n:nat):
    IF x/y<0 THEN
      -x/y<1+2*n implies exp_lb(n)(x,y)<exp(x/y)
    ELSIF x/y>0 THEN
      exp_lb(n)(x,y)<exp(x/y)
    ELSE
      exp(x/y)=1
    ENDIF

exp_ub_aux: LEMMA
  FORALL (x:real,y:nzreal,n:nat):
    IF x/y<0 THEN
      -x/y<1+2*n implies exp(x/y)<exp_ub(n)(x,y)
    ELSIF x/y>0 THEN
      x/y<1+2*n AND exp_neg_lb(-x,y)(n)/=0 implies
        exp(x/y)<exp_ub(n)(x,y)
    ELSE
      exp(x/y)=1
    ENDIF

exp_bound_is_bound: LEMMA
  FORALL (x:real,y:nzreal,n:nat):
    IF x/y<0 THEN
      -x/y<1+2*n implies
        (exp(x/y)<exp_ub(n)(x,y)
         AND
         exp_lb(n)(x,y)<exp(x/y))
    ELSIF x/y>0 THEN
```

```
        (x/y<1+2*n AND exp_neg_lb(-x,y)(n)/=0 implies
           exp(x/y)<exp_ub(n)(x,y))
        AND
        exp_lb(n)(x,y)<exp(x/y)
      ELSE
        exp(x/y)=1
      ENDIF

%-----------------------------------------------------------
% Since Nichols plot requirements involve reasoning about
% decibels (log10(f) - or ln(f)/ln(10)), upper and lower
% bounds on ln(10) are defined. These bounds are defined
% as macros so that there is no need to expand them within
% proofs.
%-----------------------------------------------------------
 ln_10_lb: MACRO posreal = 2302585/1000000

 ln_10_ub: MACRO posreal = 23025851/10000000

 ln_exp: LEMMA
   FORALL (x:real): ln(exp(x)) = x

 exp_ln: LEMMA
   FORALL (x:posreal): exp(ln(x)) = x

 ln_bounds: LEMMA
  ln_10_lb<ln(10) AND ln(10)<ln_10_ub AND ln(2)+ln(5)=ln(10)

 ln_10_pos: JUDGEMENT ln(u:gt[real,1]) HAS_TYPE posreal

 ln_monotonic: LEMMA
   FORALL (x, y:posreal): ln(x) < ln(y) IFF x < y

 Pi: posreal = pi

 Pi_lb: MACRO posreal = 314159/100000
```

```
 Pi_ub: MACRO posreal = 31416/10000

 Pi_bounds: LEMMA Pi_lb < Pi AND Pi < Pi_ub

 arctan_mod1_bound2
  (x:real,y:{u:nzreal| -1<=x/u AND x/u<=1})(n:nat): real =
  IF x/y= -1 THEN
    -sumc(0,n, LAMBDA (m:nat):
    (IF even?(m) THEN 1 ELSE -1 ENDIF)*expt(1,2*m+1)/(2*m+1))
  ELSE
    sumc(0,n, LAMBDA (m:nat):
    (IF even?(m) THEN 1 ELSE -1 ENDIF)*
    sgn(x)*expt(abs(x),2*m+1)/
       (sgn(y)*(2*m+1)*expt(abs(y),2*m+1)))
   ENDIF

%-----------------------------------------------------------
% Alternative definitions of the bounds on arctan. These take
% two values, which are intended to represent the
% numerator and denominator of the input to exp. If these
% values are integers then these definitions are
% significantly more efficient (and clearer) to expand.
% Lemmas show that these and the original
% definitions have the same meanings.
%-----------------------------------------------------------
 arctan_mod1_bound
 (x:nzreal,y:{u:nzreal| -1<=x/u AND x/u<=1})(n:nat): real =
  IF (x/y=-1) THEN
    -sumc(0,(IF even?(n) THEN n ELSE n-1 ENDIF),
      LAMBDA (m:nat): 1/(4*m+1) - 1/(4*m+3))
    - (IF even?(n) THEN 1/(4*n+3) ELSE 0 ENDIF)
  ELSE
    sumc(0, (IF even?(n) THEN n ELSE n-1 ENDIF),
      LAMBDA (m:nat):
     (((4*m+3)*y*y-(4*m+1)*x*x)*x*
        expt(abs(x)*abs(x)*abs(x)*abs(x),m))/
       ((4*m+3)*y*y*(4*m+1)*y*
```

```
          expt(abs(y)*abs(y)*abs(y)*abs(y),m)))
    + (IF even?(n) THEN
        x*abs(x)*abs(x)*expt(abs(x)*abs(x)*abs(x)*abs(x),n)/
        ((4*n+3)*y*abs(y)*abs(y)*
          expt(abs(y)*abs(y)*abs(y)*abs(y),n))
       ELSE 0 ENDIF)
 ENDIF

arctan_lb(n:nat)(x:real,y:nzreal): real =
   IF x=0 THEN
     0
   ELSIF abs(x/y)=1 THEN
     sgn(x/y)*Pi/4
   ELSIF -1<=x/y AND x/y<0 THEN
     arctan_mod1_bound(x,y)(2*n)
   ELSIF 1>=x/y AND x/y>0 THEN
     max(0,arctan_mod1_bound(x,y)(2*n+1))
   ELSIF -1>x/y THEN
     -Pi_ub/2-arctan_mod1_bound(y,x)(2*n+1)
   ELSE
     max(0,Pi_lb/2-arctan_mod1_bound(y,x)(2*n))
   ENDIF

arctan_ub(n:nat)(x:real,y:nzreal): real =
   IF x=0 THEN
     0
   ELSIF abs(x/y)=1 THEN
     sgn(x/y)*Pi/4
   ELSIF -1<=x/y AND x/y<0 THEN
     min(0,arctan_mod1_bound(x,y)(2*n+1))
   ELSIF 1>=x/y AND x/y>0 THEN
     arctan_mod1_bound(x,y)(2*n)
   ELSIF -1>x/y THEN
     min(0,-Pi_lb/2-arctan_mod1_bound(y,x)(2*n))
   ELSE
     Pi_ub/2-arctan_mod1_bound(y,x)(2*n+1)
   ENDIF
```

```
arctan_mod1_bound_equiv3: LEMMA
  FORALL (x:nzreal,y:{u:nzreal| -1<=x/u AND x/u<=1},n:nat):
    arctan_mod1_bound(x,y)(n) =
(if x/y= -1 then
  -sumc(0,2*floor(n/2),
   LAMBDA (m:nat): 1/(4*m+1) - 1/(4*m+3))
  - (if (2*floor(n/2)=n) then 1/(4*n+3) else 0 endif)
else
  sumc(0,2*floor(n/2),
   LAMBDA (m:nat): 1/(4*m+1) * (x/y)^(4*m+1)
          -1/(4*m+3) * (x/y)^(4*m+3))
  + (if (2*floor(n/2)=n) then 1/(4*n+3)*(x/y)^(4*n+3)
     else 0 endif)
endif)

arctan_mod1_bound_equiv: LEMMA
  FORALL (x:nzreal,y:{u:nzreal| -1<x/u AND x/u<=1},n:nat):
     arctan_mod1_bound(x,y)(n) =
     arctan_bounds.arctan_mod1_bound(x/y)
       (IF even?(n) THEN 2*n ELSE 2*n-1 ENDIF)

arctan_mod1_bound_equiv_m1: LEMMA
  FORALL (x:real,y:nzreal,n:nat):
    x/y=-1 IMPLIES
     arctan_mod1_bound(x,y)(n) =
     -arctan_bounds.arctan_mod1_bound(1)
       (IF even?(n) THEN 2*n ELSE 2*n-1 ENDIF)

arctan_bound_equiv: LEMMA
  FORALL (x:nzreal,y:{u:nzreal| abs(x/u)/=1},n:nat):
    arctan_lb(n)(x,y)< arctan(x/y)
    AND
    arctan(x/y)< arctan_ub(n)(x,y)

arctan_bound_equiv2: LEMMA
  FORALL (x:real,y:nzreal,n:nat):
```

```
    arctan_lb(n)(x,y)<= arctan(x/y)
    AND
    arctan(x/y)<= arctan_ub(n)(x,y)

END NRV_exp
```

In the following theory, various *rewrite rules* are defined. These rules are automatically applied when the *(assert)* proof command is called. These rewrite rules simplify proofs by reducing the need to expand certain common functions such as abs and sgn (absolute value and sign of a real variable, respectively).

```
NRV_rewrites: THEORY

BEGIN

 IMPORTING NRV_exp

 neg_neg: LEMMA FORALL (x:real): --x = x

 AUTO_REWRITE+ abs
 a: real

 abs1 : FORMULA a>=0 IMPLIES abs(a) = a
 abs2 : FORMULA a<0 IMPLIES abs(a) = -a

 AUTO_REWRITE+ sgn
 b: real

 sgn1 : FORMULA b>=0 IMPLIES sgn(b) = 1
 sgn2 : FORMULA b<0 IMPLIES sgn(b) = -1

 AUTO_REWRITE+ neg_neg
 c: real

 neg1 : FORMULA c>=0 IMPLIES --c = c

 AUTO_REWRITE+ Pi_bounds
```

```
 d: real

 Pi1 : FORMULA d>=0 IMPLIES d/Pi>=0

END NRV_rewrites
```

In the following theory, definitions are given of various functions used in NRV in the construction of lemmas along with several lemmas that are used within proofs by NRV.

```
NRV: THEORY

BEGIN

  IMPORTING NRV_rewrites, Field@extra_tegies

%-----------------------------------------------------------
% Define various functions that simplify lemma definitions
% in NRV.
%-----------------------------------------------------------
 poi_interval(n:[real->real],d:[real->real],z,x,y:real):
 bool =
   ((n(x)>=0 AND d(x)>=0) OR (n(x)<=0 AND d(x)<=0)) AND
   (n(y)>0 AND d(y)<0);

 poi_interval2(n:[real->real],d:[real->real],z,x,y:real):
 bool =
  ((n(x)>=0 AND d(x)>=0) OR (n(x)<=0 AND d(x)<=0)) AND
   (n(y)<0 AND d(y)>0);

 convexity(n:[real->real],d:[real->real],x:real): bool =
   (n(x)>=0 AND d(x)>=0) OR (n(x)<=0 AND d(x)<=0)

 concavity(n:[real->real],d:[real->real],x:real): bool =
   (n(x)>=0 AND d(x)<=0) OR (n(x)<=0 AND d(x)>=0);

 decr(n,d:real): bool =
   (n>=0 AND d<=0) OR (n<=0 AND d>=0);
```

```
%-----------------------------------------------------------
% Lemmas used within proofs in NRV.
%-----------------------------------------------------------
 sqrt_pos: JUDGEMENT sqrt(x:posreal) HAS_TYPE posreal

 line_lb(X:real,Xlb:le[real,X],
 Xub:{u:ge[real,X]|sgn(u)=sgn(Xlb)}, c,m:real,
 mlb:le[real,m],mub:{u:ge[real,m]|sgn(u)=sgn(mlb)}): real =
   if m>=0 and Xlb>=0 then
     ln_10_lb/20 * mlb * Xlb +
     min(ln_10_lb/20*c,(ln_10_ub)/20*c)
   elsif m>=0 and Xlb<0 then
     ln_10_ub/20 * mub * Xlb +
     min(ln_10_lb/20*c,(ln_10_ub)/20*c)
   elsif m<0 and Xlb>=0 then
     ln_10_ub/20 * mlb * Xub +
     min(ln_10_lb/20*c,(ln_10_ub)/20*c)
   else
     ln_10_lb/20 * mub * Xub +
     min(ln_10_lb/20*c,(ln_10_ub)/20*c)
   endif;


 ln_lt_line: LEMMA
   FORALL (X,Xlb,Xub,c,m,mlb,mub:real):
    mlb<=m AND m<=mub AND sgn(mlb)=sgn(mub) AND
    Xlb<=X AND X<=Xub AND sgn(Xlb)=sgn(Xub)
      IMPLIES
       line_lb(X,Xlb,Xub,c,m,mlb,mub)
        <= ln(10)/20*(m*X+c)

 ln_lt_line2: LEMMA
   FORALL (p:posreal,q,m,c,X:real,n:nat):
    (q<0 IMPLIES -2*q<2*n+1) AND
    q<=ln(10)/20*(m*X+c)
      IMPLIES
```

```
      (p < exp_lb(n)(2*q,1)
      IMPLIES
        20*ln(sqrt(p))/ln(10)<m*X+c)

ln_lt_line3: LEMMA
  FORALL (p:posreal,q,m,c,X:real,
   mlb:le[real,m], mub:{u:ge[real,m]|sgn(u)=sgn(mlb)},
   Xlb:le[real,X],Xub:{u:ge[real,X]|sgn(u)=sgn(Xlb)},n:nat):
    (q<0 IMPLIES -2*q<2*n+1) AND
    q<=line_lb(X,Xlb,Xub,c,m,mlb,mub)
      IMPLIES
       (p < exp_lb(n)(2*q,1)
       IMPLIES
         20*ln(sqrt(p))/ln(10)<m*X+c)

ln_lt_line4: LEMMA
  FORALL (p:posreal,nm:real,d:nzreal,m,c,X:real,
   mlb:le[real,m], mub:{u:ge[real,m]|sgn(u)=sgn(mlb)},
   Xlb:le[real,X],Xub:{u:ge[real,X]|sgn(u)=sgn(Xlb)},n:nat):
    (nm/d<0 IMPLIES -nm/d<2*n+1) AND
    nm/(2*d)<=line_lb(X,Xlb,Xub,c,m,mlb,mub)
      IMPLIES
       (p < exp_lb(n)(nm,d)
       IMPLIES
         20*ln(sqrt(p))/ln(10)<m*X+c)

ln_le_line4: LEMMA
  FORALL (p:posreal,nm:real,d:nzreal,m,c,X:real,
   mlb:le[real,m], mub:{u:ge[real,m]|sgn(u)=sgn(mlb)},
   Xlb:le[real,X],Xub:{u:ge[real,X]|sgn(u)=sgn(Xlb)},n:nat):
    (nm/d<0 IMPLIES -nm/d<2*n+1) AND
    nm/(2*d)<=line_lb(X,Xlb,Xub,c,m,mlb,mub)
      IMPLIES
       (p <= exp_lb(n)(nm,d)
       IMPLIES
         20*ln(sqrt(p))/ln(10)<=m*X+c)
```

```
line_ub(X:real,Xlb:le[real,X],
Xub:{u:ge[real,X]|sgn(u)=sgn(Xlb)}, c,m:real,
mlb:le[real,m],mub:{u:ge[real,m]|sgn(u)=sgn(mlb)}): real =
  if m>=0 and Xlb>=0 then
    ln_10_ub/20 * mub * Xub +
    max(ln_10_lb/20*c,(ln_10_ub)/20*c)
  elsif m>=0 and Xlb<0 then
    ln_10_lb/20 * mlb * Xub +
    max(ln_10_lb/20*c,(ln_10_ub)/20*c)
  elsif m<0 and Xlb>=0 then
    ln_10_lb/20 * mub * Xlb +
    max(ln_10_lb/20*c,(ln_10_ub)/20*c)
  else
    ln_10_ub/20 * mlb * Xlb +
    max(ln_10_lb/20*c,(ln_10_ub)/20*c)
  endif;

ln_gt_line: LEMMA
  FORALL (X,Xlb,Xub,c,m,mlb,mub:real):
   mlb<=m AND m<=mub AND sgn(mlb)=sgn(mub) AND
   Xlb<=X AND X<=Xub AND sgn(Xlb)=sgn(Xub)
     IMPLIES
      line_ub(X,Xlb,Xub,c,m,mlb,mub)
       >= ln(10)/20*(m*X+c)

ln_gt_line2: LEMMA
  FORALL (p:posreal,q,m,c,X:real,n:nat):
 (q>0 IMPLIES  exp_neg_lb(-2*q,1)(n)/=0) AND
   abs(2*q)<2*n+1 AND
   q>=ln(10)/20*(m*X+c)
     IMPLIES
      (p > exp_ub(n)(2*q,1)
      IMPLIES
        20*ln(sqrt(p))/ln(10)>m*X+c);
```

```
ln_gt_line3: LEMMA
  FORALL (p:posreal,q,m,c,X:real,
   mlb:le[real,m], mub:{u:ge[real,m]|sgn(u)=sgn(mlb)},
   Xlb:le[real,X],Xub:{u:ge[real,X]|sgn(u)=sgn(Xlb)},n:nat):
  (q>0 IMPLIES  exp_neg_lb(-2*q,1)(n)/=0) AND
    abs(2*q)<2*n+1 AND
    q>=line_ub(X,Xlb,Xub,c,m,mlb,mub)
      IMPLIES
       (p > exp_ub(n)(2*q,1)
       IMPLIES
         20*ln(sqrt(p))/ln(10)>m*X+c);

ln_gt_line4: LEMMA
  FORALL (p:posreal,nm:real,d:nzreal,m,c,X:real,
   mlb:le[real,m], mub:{u:ge[real,m]|sgn(u)=sgn(mlb)},
   Xlb:le[real,X],Xub:{u:ge[real,X]|sgn(u)=sgn(Xlb)},n:nat):
  (nm/d>0 IMPLIES  exp_neg_lb(-nm,d)(n)/=0) AND
    abs(nm/d)<2*n+1 AND
    nm/(2*d)>=line_ub(X,Xlb,Xub,c,m,mlb,mub)
      IMPLIES
       (p > exp_ub(n)(nm,d)
       IMPLIES
         20*ln(sqrt(p))/ln(10)>m*X+c);

ln_ge_line4: LEMMA
  FORALL (p:posreal,nm:real,d:nzreal,m,c,X:real,
   mlb:le[real,m], mub:{u:ge[real,m]|sgn(u)=sgn(mlb)},
   Xlb:le[real,X],Xub:{u:ge[real,X]|sgn(u)=sgn(Xlb)},n:nat):
  (nm/d>0 IMPLIES  exp_neg_lb(-nm,d)(n)/=0) AND
    abs(nm/d)<2*n+1 AND
    nm/(2*d)>=line_ub(X,Xlb,Xub,c,m,mlb,mub)
      IMPLIES
       (p >= exp_ub(n)(nm,d)
       IMPLIES
         20*ln(sqrt(p))/ln(10)>=m*X+c);

arctan_lt_line: LEMMA
```

```
  FORALL (p,nm:real,d:nzreal,c,cb:real,n:nat):
   p<=nm/d AND cb<=c
     IMPLIES
      (arctan_ub(n)(nm,d)<cb
      IMPLIES
        arctan(p)<c)

arctan_le_line: LEMMA
  FORALL (p,nm:real,d:nzreal,c,cb:real,n:nat):
   p<=nm/d AND cb<=c
     IMPLIES
      (arctan_ub(n)(nm,d)<=cb
      IMPLIES
        arctan(p)<=c)

arctan_gt_line: LEMMA
  FORALL (p,nm:real,d:nzreal,c,cb:real,n:nat):
   p>=nm/d AND cb>=c
     IMPLIES
      (arctan_lb(n)(nm,d)>cb
      IMPLIES
        arctan(p)>c);

arctan_ge_line: LEMMA
  FORALL (p,nm:real,d:nzreal,c,cb:real,n:nat):
   p>=nm/d AND cb>=c
     IMPLIES
      (arctan_lb(n)(nm,d)>=cb
      IMPLIES
        arctan(p)>=c);

cross_mult_le: lemma
  FORALL (px,py:posreal,q,r:real):
    q/px <= r/py IFF q*py <= r*px

cross_mult_ge: lemma
  FORALL (px,py:posreal,q,r:real):
```

```
    q/px >= r/py IFF q*py >= r*px

%-----------------------------------------------------------
% Maple repesents square roots using ^(1/2) so the
% definition of ^ is extended to refelect this.
%-----------------------------------------------------------
  ^(x:real,n:{u:real| u=1/2 or (integer_pred(u) and u>=0)}):
  real =
  IF n=1/2 then
    sqrt(x)
  ELSE
    expt(x,n)
  ENDIF

END NRV
```

Bibliography

[1] C Abdallah, P Dorato, R Liska, S Steinberg, and W Yang. Application of quantifier elimination theory to control theory. Technical Report EECE95–007, Dept. of Electrical and Computer Engineering, University of New Mexico, 1995.

[2] A Adams, M Dunstan, H Gottliebsen, T Kelsey, U Martin, and S Owre. Computer algebra meets automated theorem proving: Integrating Maple and PVS. In Boulton and Jackson [25], pages 27–42.

[3] R A Adams. *Calculus: a complete course*. Addison–Wesley, third edition, 1995.

[4] B Akbarpour and S Tahar. A methodology for the formal verification of FFT algorithms in HOL. In Hu and Martin [67], pages 37–51.

[5] R Alur, C Courcoubetis, N Halbwachs, T A Henzinger, P H Ho, X Nicollin, A Olivero, J Sifakis, and S Yovine. The algorithmic analysis of hybrid systems. *Theor. Comput. Sci.*, 138(1):3–34, 1995.

[6] R Alur, C Courcoubetis, T A Henzinger, and P H Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In Grossman et al. [56], pages 209–229.

[7] R Alur, T Dang, and F Ivancic. Counter–example guided predicate abstraction of hybrid systems. In Garavel and Hatcliff [51], pages 208–223.

[8] R Alur, T Henzinger, G Lafferriere, and G J Pappas. Discrete abstractions of hybrid systems. In *Proceedings of the IEEE*, volume 88, pages 971–984, 2000.

[9] H Anai and V Weispfenning. Deciding linear–trigonometric problems. In Traverso [119], pages 14–22.

[10] Z S Andraus and K A Sakallah. Automatic abstraction and verification of Verilog models. In Malik et al. [91], pages 218–223.

[11] R Arthan, P Caseley, C O'Halloran, and A Smith. ClawZ: Control laws in Z. In McDermid et al. [93], pages 169–176.

[12] R D Arthan. *ClawZ*. Lemma 1, 2003. Available at `http://www.lemma-one.com/clawz_docs/clawz_docs.html`.

[13] E Artin. *The Gamma Function*. Holt, Rinehart and Winston, Inc, 1964.

[14] D Atherton. *Nonlinear Control Engineering*. Van Nostrand, 1975.

[15] R Backhouse. *Program construction and verification*. Prentice–Hall, 1986.

[16] C Ballarin, K Homann, and J Calmet. Theorems and algorithms: An interface between Isabelle and Maple. In Levelt [85], pages 150–157.

[17] J Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison–Wesley, 2003.

[18] A Bauer, E Clark, and X Zhao. Analytica — an experiment in combining theorem proving and symbolic computation. *Journal of Automated Reasoning*, 21(3):295–325, 1998.

[19] D Berleant and B Kuipers. Qualitative and quantitative simulation: Bridging the gap. *Artificial Intelligence*, 95(2):215–255, 1997.

[20] E A Boiten, J Derrick, and G Smith, editors. *Integrated Formal Methods, 4th International Conference*, volume 2999 of *Lecture Notes in Computer Science*. Springer, 2004.

[21] T Bolognesi and E Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[22] O H Bosgra, H Kwakernaak, and G Meinsma. *Design Methods for Control Systems: Notes for a course of the Dutch Institute of Systems and Control, Winter term 2001–2002*. Department of Systems, Signals and Control, University of Twente, 2001.

[23] R Boulton, H Gottliebsen, R Hardy, T Kelsey, and U Martin. Design verification for control engineering. In Boiten et al. [20], pages 21–35.

[24] R Boulton, R Hardy, and U Martin. A Hoare logic for single–input single–output continuous–time control systems. In Pnueli and Maler [107], pages 113–125.

[25] R J Boulton and P B Jackson, editors. *Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2001)*, volume 2152 of *Lecture Notes in Computer Science*. Springer–Verlag, 2001.

[26] W Brauer, G Rozenberg, and A Salomaa, editors. *Interactive theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*. Springer, 1998.

[27] E Brinksma and K Guldstrand Larsen, editors. *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*. Springer–Verlag, 2002.

[28] R E Bryant, S K Lahiri, and S A Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In Brinksma and Guldstrand Larsen [27], pages 78–92.

[29] B Buchberger, T Jebelean, F Kriftner, M Marin, E Tomuta, and D Vasaru. A survey of the Theorema project. In Kuechlin [78], pages 384–391.

[30] M Butler, C Jones, A Romanovsky, and E Troubitsyna, editors. *Proceedings of the Workshop on Rigorous Engineering of Fault–Tolerant Systems*, 2005.

[31] J Carette and W M Farmer, editors. *Proceedings of the 12th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning (Calculemus 2005)*. ENTCS, 2005.

[32] E M Clarke, A Biere, R Raimi, and Y Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[33] E M Clarke Jr, O Grumberg, and D A Peled. *Model Checking*. MIT Press, 2001.

[34] P J Cohen. Decision procedures for real and p–adic fields. *Commun. Pure Appl. Math.*, 22(2):131–151, 1969.

[35] G E Collins. Quantifier elimination for the elementary theory of real closed fields by cylindrical algebraic decomposition. *Automata Theory and Formal Languages*, 33:134–138, 1975.

[36] G E Collins and H Hong. Partial cylindrical algebraic decomposition for quantifier elimination. *Journal of Symbolic Computation*, 12(3):299–328, 1991.

[37] G E Collins and R Loos. Polynomial real root isolation by differentiation. In Jenks [69], pages 15–25.

[38] J B Dabney and T L Harman. *Mastering Simulink*. Prentice Hall, 2004.

[39] T Daly. *Axiom: The 30 Year Horizon*, 2003. Available at `http://page.axiom-developer.org/zope/Plone/refs/books/axiom-book2.pdf/`.

[40] J W de Bakker, C Huizing, W P de Roever, and G Rozenberg, editors. *Real–Time: Theory in Practice, REX Workshop*, volume 600 of *Lecture Notes in Computer Science*. Springer, 1992.

[41] J de Kleer and J S Brown. A qualitative physics based on confluences. *Artificial Intelligence*, 24(1–3):7–83, 1984.

[42] A Dolzmann and T Sturm. Redlog: Computer algebra meets computer logic. *ACM SIGSAM Bulletin*, 31(2):2–9, 1997.

[43] R C Dorf and R H Bishop. *Modern Control Systems*. Prentice–Hall, ninth edition, 2001.

[44] B Dutertre. Elements of mathematical analysis in PVS. In von Wright et al. [121], pages 141–156.

[45] D L Dvorak. Monitoring and diagnosis of continuous dynamic systems using semi-quantitative simulation. (Doctoral dissertation, Department of Computer Sciences) AI 92-170, University of Texas at Austin, Artificial Intelligence Laboratory, 1992.

[46] EPSRC. *Proceedings of PREP 2004*, 2004.

[47] R Fikes and W Lehnert, editors. *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI–93)*. The AAAI Press/The MIT Press, 1993.

[48] Action Group FM(AG08). Robust flight control design challenge problem formulation and manual: the High Incidence Research Model (HIRM). Technical Report TP–088–4, version 3, Group for Aeronautical Research and Technology in Europe (GARTEUR), 1997.

[49] K Forsman. *Constructive commutative algebra in nonlinear control theory*. PhD thesis, Linköping University, 1991.

[50] G F Franklin, J D Powell, and M Workman. *Digital Control of Dynamic Systems*. Addison Wesley Longman, third edition, 1998.

[51] H Garavel and J Hatcliff, editors. *Tools and Algorithms for the Construction and Analysis of Systems TACAS 2003*, volume 2619 of *Lecture Notes in Computer Science*. Springer, 2003.

[52] M Gordon, A Cohn, T Melham, A Pitts, K Slind, and M Norrish. *The HOL System description*. Cambridge HOL group. Available at `http://hol.sourceforge.net/documentation.html`.

[53] M J C Gordon and T F Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[54] H Gottliebsen. *Automated Theorem Proving for Mathematics: Real Analysis in PVS*. PhD thesis, University of St Andrews, 2001.

[55] W K Grassmann and J Tremblay. *Logic and Discrete Mathematics: A Computer Science Perspective*. Prentice Hall, 1996.

[56] R L Grossman, A Nerode, A P Ravn, and H Rischel, editors. *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*. Springer, 1993.

[57] M H Hamza, editor. *Proceedings of Modelling, Identification and Control MIC–2004*. ACTA Press, 2004.

[58] R Hardy. Formal methods for control engineering. In *Proceedings of PREP 2004* [46], pages 137–138.

[59] R Hardy. Interactions between PVS and Maple in symbolic analysis of control systems. In Carette and Farmer [31].

[60] R Hardy. Symbolic analysis of control systems. Technical Report RR–05–06, School of Computer Science, Queen Mary University of London, 2005. Proceedings of Workshop on Verification and Theorem Proving for Continuous Systems (NetCA Workshop 2005).

[61] J Harrison and L Théry. Reasoning about the reals: The marriage of HOL and Maple. In Voronkov [122], pages 351–359.

[62] T Henzinger, P Kopke, A Puri, and P Varaiya. What's decidable about hybrid automata? In Leighton and Borodin [82], pages 373–382.

[63] T A Henzinger and S Sastry, editors. *Hybrid Systems: Computation and Control HSCC 98*, volume 1386 of *LNCS*. Springer, april 1998.

[64] C A R Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, oct 1969.

[65] H Hong. QEPCAD. Available at `http://www.cs.usna.edu/~qepcad/B/QEPCAD.html`.

[66] H Hong, R Liska, and S Steinberg. Testing stability by quantifier elimination. *Journal of Symbolic Computation*, 24(2):161–188, 1997.

[67] A J Hu and A K Martin, editors. *Formal Methods in Computer-Aided Design FM-CAD 2004*, volume 3312 of *Lecture Notes in Computer Science*. Springer, 2004.

[68] B R Hunt, R L Lipsman, and J M Rosenberg. *A Guide to MATLAB*. Cambridge University Press, 2001.

[69] R D Jenks, editor. *SYMSAC '76: Proceedings of the third ACM symposium on Symbolic and algebraic computation*. ACM Press, 1976.

[70] M Jirstrand. *Algebraic Methods for Modeling and Design in Control*. PhD thesis, Linköping University, 1996.

[71] M Jirstrand. Nonlinear control system design by quantifier elimination. *Journal of Symbolic Computation*, 24(2):137–152, 1997.

[72] D Kapur, editor. *11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*. Springer, 1992.

[73] H Kay. A qualitative model of the space shuttle reaction control system. Technical Report AI92-188, University of Texas at Austin, Artificial Intelligence Laboratory, 1992.

[74] H Kay and B Kuipers. Numerical behavior envelopes for qualitative models. In Fikes and Lehnert [47], pages 606–613.

[75] M Kerber, M Kohlhase, and V Sorge. Integrating computer algebra into proof planning. *Journal of Automated Reasoning*, 21(3):327–355, 1998.

[76] S C Kleene. *Introduction to Metamathematics*. Wolters–Noordhoff Publishing and North–Holland Publishing Company, 1971.

[77] M Kreuzer and L Robbiano. *Computational Commutative Algebra 1*. Springer, 2000.

[78] W Kuechlin, editor. *ISSAC '97: Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation*. ACM Press, 1997.

[79] B Kuipers. *Qualitative Reasoning: Modeling and Simulation with Incomplete Knowledge*. MIT Press, 1994.

[80] B Kuipers and S Ramamoorthy. Qualitative modeling and heterogeneous control of global system behavior. In Tomlin and Greenstreet [118], pages 294–307.

[81] T Latvala, A Biere, K Heljanko, and T A Junttila. Simple bounded LTL model checking. In Hu and Martin [67], pages 186–200.

[82] F T Leighton and A Borodin, editors. *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing*. ACM Press, 1995.

[83] Lemma 1. *ProofPower Compliance Tool – User Guide*, 2000. Available at `http://www.lemma-one.com/ProofPower/doc/doc.html`.

[84] Lemma 1. *ProofPower Description*, 2000. Available at `http://www.lemma-one.com/ProofPower/doc/doc.html`.

[85] A H M Levelt, editor. *ISSAC '95: Proceedings of the 1995 International Symposium on Symbolic and Algebraic Computation*. ACM Press, 1995.

[86] R Liska and S Steinberg. Applying quantifier elimination to stability analysis of difference schemes. *Comput. Journal*, 36(5):497–503, 1993.

[87] C Livadas and N A Lynch. Formal Verification of Safety–Critical Hybrid Systems. In Henzinger and Sastry [63], pages 253–272.

[88] N A Lynch, R Segala, and F W Vaandrager. Hybrid I/O automata. *Inf. Comput.*, 185(1):105–157, 2003.

[89] S Maharaj. A PVS theory of Symbolic Transition Systems. In R J Boulton and P B Jackson, editors, *Supplemental Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2001)*, pages pp. 255–266. University of Edinburgh, Devision of Infomatics, 2001. Research Report EDI-INF-RR-0046.

[90] O Maler, Z Manna, and A Pnueli. From timed to hybrid systems. In de Bakker et al. [40], pages 447–484.

[91] S Malik, L Fix, and A B Kahng, editors. *DAC '04: Proceedings of the 41st annual conference on Design automation*. ACM Press, 2004.

[92] S McCallum. An improved projection operation for cylindrical algebraic decomposition of three-dimensional space. *Journal of Symbolic Computation*, 5(1/2):141–161, 1988.

[93] J A McDermid, M G Hinchey, and S Liu, editors. *Proceedings of the 3rd IEEE International Conference on Formal Engineering Methods (ICFEM 2000)*. IEEE Computer Society Press, 2000.

[94] M B Monagan, K O Geddes, K M Heal, G Labahn, S M Vorkoetter, J McCarron, and P DeMarco. *Maple 7 Programming Guide*. Waterloo Maple Inc, 2001.

[95] National Instruments. *Matrix$_x$ — Getting Started Guide*.

[96] National Instruments. *SystemBuild user Guide*.

[97] X Nicollin, A Olivero, J Sifakis, and S Yovine. An approach to the description and analysis of hybrid systems. In Grossman et al. [56], pages 149–178.

[98] X Nicollin, J Sifakis, and S Yovine. From ATP to timed graphs and hybrid systems. In de Bakker et al. [40], pages 549–572.

[99] R Nikoukhah. Scicos: A dynamic systems modeler and simulator. In Hamza [57].

[100] In–flight breakup over the Atlantic Ocean of Trans World Airlines Flight 800 Boeing 747-131, n93119, near East Moriches, New York july 17 1996. Aircraft Accident Report AAR–00/03, National Transport Safety Board, 2000.

[101] K Ogata. *Discrete–Time Control Systems*. Prentice–Hall, second edition, 1995.

[102] K Ogata. *Modern control engineering*. Prentice–Hall, third edition, 1997.

[103] S Owre, J M Rushby, and N Shankar. PVS: A prototype verification system. In Kapur [72], pages 748–752. Available at `http://www.csl.sri.com/papers/cade92-pvs/`.

[104] S Owre, N Shankar, J M Rushby, and D W J Stringer-Calvert. *PVS Language Reference*. SRI International. Available at `http://pvs.csl.sri.com/documentation.shtml`.

[105] P C Parks. A M Lyapunov's stability theory — 100 years on. *IMA Journal of Mathematical Control and Information*, 9(4):275–303, 1992.

[106] A B Pippard. The inverted pendulum. *European Journal of Physics*, 8:203–206, 1987.

[107] A Pnueli and O Maler, editors. *Proceedings of the 6th International Workshop on Hybrid Systems: Computation and Control (HSCC 2003)*, volume 2623 of *Lecture Notes in Computer Science*. Springer, 2003.

[108] R W Pratt, editor. *Flight control systems: Practical issues in design and implementation*, volume 57 of *IEE Control Engineering Series*. The Institution of Electrical Engineers, 2000. Copublished by The American Institute of Aeronautics and Astronautics.

[109] Scilab Group, INRIA, Unité de recherche de Rocquencourt, Projet Meta2, Domaine de Voluceau, Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex (France). *Introduction to Scilab*.

[110] A Seidenberg. A new decision method for elementary algebra. *Annals of Math*, 60:365–374, 1954.

[111] M Spivak. *Calculus*. Addison–Wesley, 1973.

[112] J M Spivey. *The Z notation: A reference manual*. Prentice–Hall, second edition, 1992.

[113] A Tarski. *A Decision method for elementary algebra and geometry*. University of California Press, 1951.

[114] A Tewari. *Modern Control Design with MATLAB and Simulink*. John Wiley & Sons, Inc, 2002.

[115] D Throop. *Model–based diagnosis of complex, continuous mechanisms*. PhD thesis, Department of Computer Sciences, University of Texas at Austin, 1991.

[116] A Tiwari. PVS–QEPCAD. Available at `http://www.csl.sri.com/users/tiwari/qepcad.html`.

[117] A Tiwari and G Khanna. Series of abstractions for hybrid automata. In Tomlin and Greenstreet [118], pages 465–478.

[118] C J Tomlin and M R Greenstreet, editors. *Hybrid Systems: Computation and Control HSCC*, volume 2289 of *LNCS*. Springer, March 2002.

[119] C Traverso, editor. *ISSAC '00: Proceedings of the 2000 International Symposium on Symbolic and Algebraic Computation*. ACM Press, 2000.

[120] E Tronci. Automatic synthesis of control software for an industrial automation control system. In Wing et al. [126], pages 247–250.

[121] J von Wright, J Grundy, and J Harrison, editors. *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2001)*, volume 1125 of *Lecture Notes in Computer Science*. Springer–Verlag, 1996.

[122] A Voronkov, editor. *Logic programming and automated reasoning: proceedings of the 4th international conference, LPAR '93*, volume 698 of *Lecture Notes in Computer Science*. Springer–Verlag, 1993.

[123] V Weispfenning. Deciding linear–exponential problems. *SIGSAM Bullettin*, 34(1):30–31, 2000.

[124] J Wilkie, M Johnson, and R Katebi. *Control Engineering: An Introductory Course*. Palgrave Publishers Ltd, October 2001.

[125] T Wilson, S Maharaj, and R G Clark. Omnibus: A clean language and supporting tool for integrating different assertion–based verification techniques. In Butler et al. [30], pages 43–52.

[126] J M Wing, J Woodcock, and J Davies, editors. *FM '99: Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems – Volume II*. Springer-Verlag, 1999.

[127] S Wolfram. *The Mathematica Book*, 2005. Available at `http://documents.wolfram.com/v5/TheMathematicaBook/`.

[128] M H Zaki, A Habibi, S Tahar, and G Bois. On the formal analysis of analogue systems using interval abstraction. Technical Report RR–05–06, School of Computer Science, Queen Mary University of London, 2005. Proceedings of Workshop on Verification and Theorem Proving for Continuous Systems (NetCA Workshop 2005).

[129] E Zarpas. Simple yet efficient improvements of SAT based bounded model checking. In Hu and Martin [67], pages 174–185.