MIT/LCS/TR-525

AUTOMATIC ANALYSIS OF SYSTEMS AT STEADY-STATE: HANDLING ITERATIVE DYNAMIC SYSTEMS AND PARAMETER UNCERTAINTY

Alexander Sen Yeh

December 1991

Automatic Analysis of Systems at Steady-State: Handling Iterative Dynamic Systems and Parameter Uncertainty

by

Alexander Sen Yeh

December 21, 1991

This report is a slightly modified version of a dissertation submitted to the Department of Electrical Engineering and Computer Science on August 17, 1990 in partial fulfillment of the requirements for the degree of Doctor of Philosophy. The research was supported by the National Institute of Health through grant R01-LM04493 from the National Library of Medicine and grant R01-HL33041 from the National Heart, Lung, and Blood Institute.

Keywords: artificial intelligence, steady-state systems, dynamic systems, iterative systems, Monte Carlo, probability bound, uncertainty propagation, uncertainty representation

Abstract

A way to analyze a system at steady-state is to construct and then analyze (use) a static model which describes that steady-state. This thesis deals with two problems in carrying out this task. The first concerns constructing a static model of an iterative dynamic sub-system: a sub-system that at steady-state is steadily iterating some set of parameter value changes. The second problem is being uncertain not just about parameter values, but also their distribution of values, when analyzing a static model.

To deal with the iterative dynamic system problem, an implemented computer program called AIS is described. AIS takes in a description of the parameter changes over time during an iteration of the iterative sub-system and produces a summary description of how that sub-system behaves over many repetitions. At present, the summary consists of the extreme values of some parameters, the symbolic average rates of change in parameter values and information on how those rates would be different if various constants and functions had been different. Parts of this summary can then be used to represent the iterative sub-system in a static model of the overall system. One way to view AIS is that it takes in a description of an iterative dynamic sub-system that is easy for users to give, and produces a description that is easy to analyze and incorporate into steady-state models. Another way to view AIS is that it analyzes iterative dynamic systems at steady-state.

AIS deals only with behavior where each repetition changes parameters by the same amounts. This limitation lets AIS perform the needed computations and still lets it handle an important subset of dynamic systems. Unlike some other approaches, AIS does not require that a repeating behavior be described in terms of a set of differential equations. Three examples of running the current version of AIS are given: two concern the human heart, the third a steam engine.

A standard approach to the problem of parameter uncertainty when using a model is to use a joint input probability density on the parameter values to estimate the likelihood of some behavior, such as a state variable being inside a numeric range. However, this input density's parameters and shape are often also not exactly known. To deal with this added uncertainty, this thesis describes two methods that use instead an upper or lower bound on the joint input density to bound the likelihood of a behavior. The first method produces analytic bounds, but is limited to using lower density bounds. It finds rough bounds at first, and then refines them as more iterations of the method are allowed. The second method is a hit-or-miss version of sample-mean Monte Carlo. Unlike the first method, the second method can also handle upper density bounds, which are more useful than lower density bounds, but the generated probability bounds are only approximate. However, standard deviations on the bounds are given and become small as the sample size increases.

Besides these two methods, various moment schemes can also use density bounds to estimate bounds on the likelihood of some behavior, but such schemes' estimates can be quite bad. However, moment schemes can be useful for estimating how average parameter values and variations in values affect each other, and this thesis describes a moment scheme for estimating such effects.

Acknowledgments

For years I have been in a steady-state of steadily accumulating more semesters as a graduate student by iteratively traveling between my apartment and my office. I have finally broken that steady-state, and I have many people to thank for taking part in making this possible. Below is but a partial list.

My thesis supervisor Peter Szolovits supplied many years of financial support and insightful comments, even when on the other side of the country.

William Long also supplied many years of financial assistance. In addition, he served as my "expert" on the cardiovascular system and supplied the cardiovascular model which I played with so much while hunting for thesis ideas. Outside of the cardiovascular domain, he prodded me to clarify my examples and supplied some of the utilities that AIS is dependent on.

Alvin Drake kept urging me on during my early years working on this thesis, and he and Peter Kempthorne helped me with some of the probability and statistics.

During the last several years, I have been fortunate enough to have had the following office-mates: Elisha Sacks conceived of, programmed and maintained the Bounder system used in this thesis. Dennis Fogg asked questions that made me think hard about various topics. Ira Haimowitz had an enthusiastic outlook that would rub off on me.

Nearby my office was Jon Doyle, who along with Peter Szolovits helped in formalizing AIS' abilities. Mike Wellman was also nearby, and one of my discussions with him lead to the idea for SAB.

The past and present various members of "Club MEDG" (including most of the people mentioned above), a group of people who are always willing to help one another. Various members have aided me in numerous ways, including helping me to clarify my ideas and presentations and to get the computer systems to do my bidding. They have also been a fun group to be with. Graduate school would not have been the same without them.

Outside of MEDG, Peter Huber and several other members of the MIT mathematics and statistics communities provided useful advice, as have the conference and workshop reviewers of my various submissions and the conference and workshop participants with whom I have had discussions.

Eric Sollee and the other fencers at MIT kept injecting tempo changes in my day-to-day life. My practice with them came in handy at times during presentations and heated discussions.

The friends I had "accumulated" before I started my recent steady-state of being in the MEDG group were a reminder that I did not have to stay in that steady-state forever.

Saida Memon was always around, made life pleasant and kept encouraging me onward. She likes to get things done quickly, but she definitely displayed a lot of patience in waiting for me to finish.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Building and using models are common ways of analyzing many continuous systems, including the human body, spacecraft, and ecological and social systems. The models may be used to predict what will happen, indicate what should have happened if nothing went wrong, or show how and why some result occurs the way it does. These models may either be dynamic or static in nature. Dynamic models describe how various system quantities (parameters) change over time in response to an input. These models are often used to simulate a system. Static models can either describe a system's condition at a point in time or describe aspects of a system response that do not change with time. These models may either describe a static system or a dynamic one. This thesis deals with static models of either static systems or of time invariant aspects of dynamic systems that have reached a steady-state (equilibrium). A static model for a dynamic system (when such a model can be made) is simpler than a dynamic model of that same system. Even so, both building and using static models can be complicated.

This thesis deals with two problems. One concerns iterative dynamic systems. This problem occurs when building a static model to describe some of a steady-state dynamic system's time invariant properties: Sometimes, one finds that at steady-state, parts of the system form an iterative dynamic sub-system that cannot be really viewed as lying still at a single state (set of parameter values). Instead, a better way to view the situation is that these parts constantly repeat a set of actions that change the parameter values at a constant rate. For example, consider the human circulatory (cardiovascular) system. A person at rest for some time is considered to be at a steady-state. At this steady-state, the heart is (one would hope) still pumping: Parameters like the instantaneous blood pressure and amount of blood in the heart have values that are changing periodically, and the amount of blood that has exited the heart is steadily increasing.

The other problem concerns parameter uncertainty and occurs when one wants to use the model. Often, one is uncertain about what values to assign to the model parameters, and in fact is not quite even certain of what the distribution of possible values is.

1.1 The Iterative Dynamic System Problem

For the first problem, it is still possible to build a static model of a system with iterating parts if only system properties that are time invariant at steady-state need to be described: one can either ignore the parameters (quantities) that change at steady-state or substitute for the changing parameters some variation that *is* constant, which may be their average or extreme values, or their rate of change. As mentioned in the cardiovascular system example above, many parameters change in value over time at steady-state. However, variations of those parameters do stay constant over time at steady state. Examples include the average and extreme values of blood pressure and amount of blood in the heart, and the average rate at which blood exits the heart (cardiac output). Since for many purposes, one can describe the cardiovascular system state with these types of "parameters" that stay constant at steady-state, a static model of these properties is useful.

After choosing the parameters of the static model to be built, there still remains the task of finding and describing time-invariant relationships between these parameters at steady-state. Unfortunately, it may not be easy to do this directly. For example, when describing a steadily beating ventricle (part of a heart), it is hard to give directly the simultaneous equations that describe the relationships between such parameters as the input and output blood pressure, the beat rate, and the rate of blood entering and leaving the ventricle.

An easier method of describing what happens with system parts that repeat a sequence of actions (parameter value changes or transformations over time) may be to give the sequence and the relationships that hold in each part of the sequence. With the ventricle example, let P and V be the ventricle's pressure and volume respectively, Bi be the amount of blood that has moved into the ventricle, Bo be the amount that has moved out, Pi be the pressure of the entering blood, Po be the pressure of the exiting blood, $Vd[Pi]$ (a function of Pi) be the amount of blood in the ventricle when it is relaxed, and $Vs[Po, HR]$ be the amount of blood in the ventricle when it is squeezing as hard as it can. HR is the rate at which the ventricle beats. Then part of such a description may be as follows: The sequence is that

1. The ventricle squeezes the blood in it without releasing any: V, Bi and Bo stay the same. P changes to a value of Po.

2. The squeezing continues with blood exiting out the ventricle's output: P and Bi stay the same. V changes to a value of $Vs[Po, HR]$. Bo increases by the opposite of the change in V's value.

3. The ventricle relaxes: V, Bi and Bo stay the same. P changes to a value of Pi.

4. The relaxation continues with blood entering via the ventricle's input: P and Bo stay the same. V changes to a value of $Vd[Pi]$. Bi increases by the change in V's value.

While such a description is easier to give, it alone does not complete the static model. To complete the model, one needs to derive from that description the time-invariant parameter relationships at steady-state. In the case of the ventricle, two of the relationships are (dX/dt is the *average* rate of change in parameter X):

$$d(Bi)/dt = d(Bo)/dt = HR \cdot (\,Vd[Pi] - Vs[Po, HR]),$$

where the symbols are as described above. AIS (short for Analyzer of Iterated Sequences), a program to automatically perform such derivations, is described in the first part of this thesis. AIS also performs sensitivity analysis on how such relationships would be different if various constants had different values. This analysis can be useful both in getting a feel for how the relationships behave and also in giving something to compare against experiments done on the system of interest.

1.2 The Parameter Uncertainty Problem

Problems exist even after a static model is built. One such problem is that when trying to use the model to determine if the parameters satisfy some criteria[1], one or more of the input parameters may not have a precise value, or that precise value may be unknown. One way to handle such imprecision is to specify a set of numeric bounds on each input parameter value and then propagate those bounds to the output values by using algorithms like the ones found in [36, 38]. Unfortunately, the bounds get broader with each successive level of propagation through the parameter relationships, so the bounds on the output parameters tend to be too broad to indicate anything useful.

Another way to handle imprecision is to give the joint probability distribution for the input parameter values, and then use this distribution and some method like Monte Carlo [16, 22] to estimate the probability of satisfying the criteria of interest. There are at least two possible complications to this approach. The first is that specifying the input value joint distribution usually requires estimates for the joint distribution's parameters[2], such as the means and standard deviations. These estimates may be based on only statistics from a few samples, or even intuition or heuristics, and so may be subject to error or be slightly inconsistent with each other. For example, in the control run in [14], a parameter called *PWP* had a sample mean of 23 and a sample variance of about 37. Like in many other medical studies, these numbers were the averages of ten patients, so assuming that the sample variance is roughly correct, the sample mean itself has a standard deviation of $\sqrt{37/10} \approx 2$. This implies that the mean value's estimate has a good chance of being off by 10 to 20%. Assuming an approximately Gaussian distribution for the sample mean, its 95% confidence interval is between 19 and 27.

A second complication is that even when one gets the input value distribution's parameters, the distribution's form may be hard to approximate. As an example, data from [14] is used to estimate the means, variances and correlations of variables

[1]An example of some criteria is that neither the blood pressure nor the heart rate exceed certain thresholds.

[2]These parameters are not to be mistaken with the previously mentioned parameters of the model.

in that paper. Sets of those variables were treated as either having a jointly Gaussian or lognormal probability distribution[3] and considered to be inputs to a model of the human cardiovascular system (described in Appendix B). When Monte Carlo simulations of the system were performed[4], many of the samples (from a third to over half) were rejected for violating a numeric limit on some model variable. Rejecting these samples produced results with generally lower means and standard deviations than the corresponding data. Some of the worst offenders were differences between two variables that became negative. Among other things, this led to samples that implied that blood was flowing in the reverse direction. Trying to remodel the system with the offending differences as lognormal variables[5] would help when it could be done. But often, the correlation estimates indicated that no jointly lognormal distribution was possible, and the correlations were too high to ignore.

A way out of these two complications of specifying a set of distribution parameters and a distribution form is to give a bound on the distribution rather than specifying the exact distribution. SAB (short for Split And Bound) and HMC (short for Hit-or-miss sample-mean Monte Carlo), two methods for analyzing models with such bounds, are described in the second part of this thesis. Among other things, using a distribution bound lets one relax the estimates of only slightly inconsistent distribution parameters from point estimates to some range of values.

An alternative to specifying a bound on the probability density is to use a sample of possible probability densities. However, sampling is not as complete an examination as bounding. Some class of important behaviors may lie between all the samples and not be observed.

An issue related to that of finding/bounding the chances of some criteria being fulfilled is to find how average parameter values and variations in values affect each other. This is useful when one can change some averages or variations (say by taking more measurements to reduce variability) and one wants to find if taking the effort to produce these changes can help the chances of satisfying the criteria, etc. Unfortunately, the methods discussed in this section (Monte Carlo, SAB, HMC) do not find the relationships between parameter averages and variations. Using various moment equation schemes can find these relationships, but these schemes have problems of their own. Moment schemes are discussed in a section at the end of this chapter.

1.3 AIS: Handling Repetitive Actions

AIS is an implemented program that deals with the problem described in Section 1.1: building models of iterative dynamic systems at steady-state. When given a continuous state-description of a system and a sequence of actions or transformations on that state, AIS symbolically finds some of the extreme and time-averaged effects of continually iterating that sequence. The specific effects found at present include

[3]Examinations of the few data points involved did not indicate that this was unreasonable.

[4]To perform a simulation, repeatedly take samples from the distribution of input values and propagate each sample through the model.

[5]Lognormal variables are always positive and have a unimodal distribution [3].

1) the extreme values of parameters that vary periodically with each iteration, 2) the symbolic average rate of change in parameters, and 3) an assessment of how those rates of change would be different with different values for various constants and functions (sensitivity analysis). The sequences handled by AIS are ones which have the following "constancy" (invariant property over time): the sequence always repeats the same actions in the same order and each occurrence of a particular action always changes the parameters by the same amounts. Examples of such iterated sequences of actions include the ones taken by a heart in going through a beat cycle at steady-state and the actions taken by a steam engine in making one rotation of its drive shaft at steady-state. Effects to be found include the extreme pressures in an engine, the average rate at which blood enters the heart, and how increasing that entering blood's pressure affects that rate.

As eluded to in Section 1.1, one motivation for finding such effects may be to find what stresses a device needs to tolerate, such as the maximum pressure an engine or heart is subject to. A second motivation is that many periodic sub-systems iterate at such a fast rate that the other parts of a system respond only to the behavior of such a sub-system β averaged over many iterations. Then a steady-state model for the entire system would only require a description of β's averaged behavior; β can be modeled as a constant iteration of the same sequence of parameter value changes. Examples of such sub-system and system combinations include 1) the heart and the human circulatory system, and 2) an engine and a car.

An alternative way to describe a repetitive system is to characterize all the forces in the system and the parameters that they affect. Then, analyze the description with a method that combines qualitative simulation with cycle detection [6]. For complicated systems (such as the heart), these methods predict many possible sequences of actions besides the actual sequence. If the actual sequence can be isolated, one can use *aggregation* [50] to find which parameters change as the sequence repeats and *comparative analysis* [51, 52, 53] to find the effects of perturbing model constants.

Another alternative for describing a repetitive system is via a single set of differential equations that is always applicable. A system to analyze such sets of equations is described in [38, 39]. However, coming up with such a description for a complicated system like a ventricle or a steam engine is quite difficult (especially for the current implementation in [39], which is limited to 2nd order differential equations). In contrast, the input for both the qualitative simulation approaches and AIS can have many sets of simple equations along with the conditions to determine when a particular set is applicable.

Examples of trying to model the ventricles using differential equations are given in [30] and [24]. In [30], instead of using a single set of equations that is always applicable, the authors use one set of auxiliary variables and equations for modeling a ventricle's contraction and another set for relaxation. In [24], a ventricle's contraction and relaxation are modeled by elastance/capacitance versus time graphs rather than differential equations. With enough additional auxiliary variables, functions like step functions[6], and additional equations, one could probably model ventricular con-

[6]For a step function $u(t)$, $u(t) = 1$ when $t \geq 0$ and $u(t) = 0$ otherwise. This function is often

traction and relaxation using a single set of differential equations. But the resulting model will be large, and hard to derive, to comprehend and to reason about.

1.4 SAB and HMC: Handling Uncertainty in the Uncertainty

SAB and HMC are two methods that deal with the problem described in Section 1.2 on uncertainty in the distributions of parameter values. The two methods predict the likely steady-state behaviors of a continuous nonlinear system in which the input values can vary. The methods use a parameterized steady-state equation model and upper or lower bounds on the joint input probability density to bound the likelihood that one or more state variables stay inside or outside a given set of numeric ranges. SAB stands for ***split and bound***, the method's basic steps. SAB utilizes lower density bounds only and produces analytical likelihood bounds. HMC is a form of Monte Carlo. It can utilize both lower and upper density bounds and produces estimates of likelihood bounds with standard deviations on those estimates.

Other prediction-making methods have one or more of the following problems: not finding the likelihood of behaviors or only finding the likelihood of the variable values falling in certain ranges; not estimating the result's error and not being able to improve on an initial result's accuracy when given more computation time; not being able to handle density bounds, or handling them too slowly; needing to explicitly find and describe every region of input values that satisfies the criteria.

Compared to these other techniques, SAB and HMC can produce estimates of their errors, improve their answers as more samples or iterations are allowed, and deal with distributions of continuous variable values. SAB produces analytical answers but can only handle lower density bounds. HMC is a modification of the Monte Carlo techniques that integrate the density bound. The modification makes it possible to determine the interval to be integrated over.

1.5 Moments: Finding Relationships Between Averages and Variances

Equations between statistical moments (averages, variances, etc. of parameters) give relationships on how these moments affect each other. For example, if one has the equation

$$E[Y] = \exp(E[X] + V[X]/2)$$

for the variables Y and X, where $E[\alpha]$ is α's average (expected) value and $V[\alpha]$ is α's variance, then one knows that increasing either $E[X]$ or $V[X]$ will increase $E[Y]$, etc. Such equations are useful to have when one wants to alter some parameter's moment but can only do so indirectly via other parameter moments (typically moments of inputs).

used to combine expressions that are valid under different conditions.

Except for special cases, one cannot find exact equations between statistical moments. One method to find the approximate moments is to use truncated Taylor series expansions [16, Ch. 7]. It is exact only when finding the moments of linear combinations. For example, when $Z = X + Y$, then the corresponding moment equations derived by this method, $E[Z] = E[X] + E[Y]$ and $V[Z] = V[X] + V[Y] + 2 \cdot C[X, Y]$, are exact (C[X,Y] is the covariance between X and Y).

Another method to find the approximate moments is described in a later chapter of this thesis and is called GLO. Like the truncated Taylor series method, it is exact when the relationships between the original parameters are a linear combination. GLO is also exact when finding the moments of products or exponentiations of parameters that are jointly lognormally distributed. So if the parameters X_{ik}'s are jointly lognormally distributed, then GLO can find the exact moments of expressions of the form

$$\sum_i b_i \cdot \prod_k (X_{ik})^{(a_{ik})},$$

where the b_i's and a_{ik}'s are constants. As previously mentioned, lognormal variables are always positive and have a unimodal distribution [3]. Parameters which look close to being normally distributed but only have positive values will resemble being lognormally distributed.

Unfortunately these approximate methods sometimes give bad approximations. For example, when data from [14] is used as input to the model of the human cardiovascular system described in Appendix B, the resulting moment values often came out quite different from the ones generated by the approximation methods. Neither method was convincingly "better". Sometimes one method would generate a moment that was close to the data's, and sometimes the other method would.

Another complication is that if there are n original parameter equations, there will be n^2 moment equations.[7] If some of these equations have to be solved as simultaneous nonlinear equations (such as when testing for therapies in the model in Appendix B), this increase from n to n^2 equations can spell the difference between a problem that the equation solver can solve and one that the solver cannot. In fact this was true when I tried to use the model in Appendix B with a Newton-Raphson equation solver [31, Ch. 9.6].

There are two possible solutions to these complications. One solution is to use Monte Carlo or some other more accurate method to calculate the moments and then use a moment approximation equation when it "agrees" with the more accurate method. If one just compares the final results, it will be a fast, but not a very thorough check. On the other hand, if one compares all the intermediate results generated by each subset of operands/inputs, it will be a slow (for m operands, there are 2^m subsets of operands), but thorough check. Intermediate levels of checking are possible.

Another possible solution is to use a version of Monte Carlo to find the derivative between anything and a moment of the input parameters [45, Ch. 5.6]. This version requires that the analytic form of the input parameter joint probability density be known and is restricted to looking at how moments of input parameters that are

[7]There n^2 pairs of equations to find covariances for. This was pointed out to me by an anonymous workshop paper referee.

explicitly mentioned in that distribution affect other things. So one cannot look at the relationship between the moments of two intermediate parameters. For example, let x and y be the inputs and let them be jointly Gaussian. Then x and y's joint density is an expression involving the moments $E[x]$, $E[y]$, $V[x]$, $V[y]$, and $C[x,y]$. As a result, one can only examine how changing these five moments will affect things. One cannot examine how changing $V[z]$, where $z = x \cdot y^2$, will affect things.

1.6 Thesis Outline

The next two chapters concern AIS. First, AIS is described in detail, and then, examples of using AIS are given. The examples illustrate AIS's mechanics, compare AIS's results with results derived either by hand or from empirical experiments, and illustrate a use of AIS results in building a steady-state model of a system which has a quickly iterating sub-system. The examples also show how AIS's results degrade as the input description becomes more vague. In addition, the normal ventricle example shows how being consistent with experimental results is no guarantee of model accuracy. These two chapters show that AIS can analyze a small but useful subset of dynamic systems by exploiting time invariant properties present in that subset.

These two chapters also show that AIS is an example of three general ideas in artificial intelligence. The first is that one can often solve a small subset of a general problem that is very hard to solve (and maybe unsolvable). With AIS, the general problem is predicting the behavior of dynamic systems, and the small subset that is solved is the set of iterative systems that have certain constancy properties. The second general idea illustrated is the power of having a good representation in facilitating problem solving [55, Ch. 2]. Trying to describe a steadily iterating system directly in terms of the simultaneous equations that give the steady-state relationships of its parameters is hard. AIS lets one describe such a system in terms of how the parameters change during one iteration of the system. This latter type of description is much easier to give. In addition, as the chapter that describes AIS in detail shows, once this latter type of description is given, the steady-state relationships are quite easy to derive. The third general idea also concerns good representations. The idea is that what constitutes a good representation often depends on the aspect of a problem being tackled. Hence, one may need multiple representations to solve a given problem.[8] For a system like the human cardiovascular system at a steady-state, using a set of simultaneous equations to describe its parameter relationships is useful way to look at it when trying to determine its behavior and how altering parameter values will change that behavior. As has been previously mentioned, giving the simultaneous equations describing steadily iterating sub-systems is hard, and an easier way to describe such sub-systems is to describe how their parameters change during an iteration. So when the problem is to take in a description of a system like the cardiovascular system at steady-state and then determine its behavior, the former part is facilitated by using a representation that includes how parameters change during an iteration, while the

[8]This need has been mentioned in the form of needing multiple ontologies to predict the behavior of certain systems [13].

latter part is facilitated by using simultaneous equations to represent the parameter relationships. AIS is a way of obtaining the form of representation that is easier to use/analyze for determining behavior from the form that is easier for users to give when describing steadily iterating sub-systems.

After the chapters on AIS are four chapters on SAB and HMC. The lead chapter gives a simple example of using SAB and HMC and also describes some alternatives to SAB and HMC in more detail. This is followed by a chapter on SAB and one on HMC. The chapter on SAB describes how it works and also discusses some of its limitations. These limitations led to the work producing the HMC algorithm. The chapter on HMC describes how it works and also gives some large examples of using HMC.

A chapter on the moment approximation method called GLO follows the SAB and HMC chapters.

Unfortunately, SAB, HMC and GLO illustrate a "converse" of the first general idea illustrated by AIS: often, generalizations of solvable subsets of problems may be real hard to solve or as yet unsolvable. With SAB and HMC, the solvable subset is the class of problems of predicting how likely certain system behaviors are when the input parameter values have a known joint probability distribution that is of a standard type, such as a multi-variate Gaussian. With this subset, one can use Monte Carlo simulation. But say the class of problems is generalized to include ones where the input parameter value probability distribution is either not exactly known or for which no fast pseudo-random generator exists, and so one needs to use SAB or HMC. Then a result can take a very long time to find and the result found may be an ambiguous one. With GLO, the solvable subset is the class of problems of determining the relationships between the means, variances and covariances of an expression like a linear combination (and certain forms of multiplication and exponentiation) with the corresponding moments of the expression's components. Exact moment relationships exist for this class. But for generalizations of this class, GLO and the other moment equation generation schemes can produce moment relationships that are quite bad approximations. Another way of viewing what the SAB, HMC and GLO results show is that there are at present many uncertain parameter value problems for which no good method exists to solve them.

The last chapter gives some conclusions, including a discussion on possible future directions. The first appendix gives some derivations for SAB. The second describes the cardiovascular model used in parts of this thesis. The third gives some derivations for the GLO method. The last two appendices respectively describe how to use the current implementation of AIS and give a listing of the code for that implementation.

Chapter 2

Description of AIS

This chapter describes AIS, a program that analyzes an iterative dynamic system by taking in a description of that system iterating its sequence of parameter value changes (transformations), and then finding some of the effects of iterating that sequence. In order, the sections describe the input for AIS, some of AIS's preliminary processing, and AIS's output. Appendix D describes how to use the current implementation of AIS. Appendix E gives a listing of the code for that implementation.

2.1 Input

An input description consists of three parts: the parameters which describe the system state, static conditions on those parameters, and the sequence of transformations (actions) that gets iterated. The description only has to try to describe what happens in a sequence of actions, not necessarily how or why that sequence occurs or repeats.

Parameters are divided by the model-builder into four types. The first three types are classified by how a parameter behaves as the sequence of actions is iterated:

1. *Constant parameters* do not change in value at all during the iterations.
2. *Periodic parameters* change in value, but the sequence of values repeats exactly with each new action sequence iteration.
3. *Accumulating parameters* monotonically increase or decrease in value with each action sequence iteration.

In general, parameters are represented by symbols. The constant parameter type also includes numbers and arbitrary functions of expressions of constant parameters such as $f[x+3, g[5]]$, where x is a constant. The fourth parameter "type" has only one parameter: the rate at which the sequence of actions is iterated. At present, the rate must be expressed as a constant parameter that is a symbol or number.

The second part of the input is a set of static conditions between constant parameters. These conditions are inequalities between numbers and expressions made up of constant parameters. The expressions can have algebraic and the more common transcendental functions. Also permissible are (partial) derivatives of constant parameters which are arbitrary functions.[1] The inequalities can be either definitions that are always true or conditions that are required for the given sequence of actions to iterate. An example of a definition is to say that some volume is ≥ 0. An example of a necessary condition is to say that for a normal sequence of actions in the heart, the input pressure is less than the output pressure.[2]

A note on conditions with expressions that involve (partial) derivatives of constant parameters which are arbitrary functions: currently, such a condition only makes a statement of the derivative with respect to the argument(s) mentioned. For example, mentioning that $0 < d^2f(x)/dx^2$ says nothing about $d^2f(y)/dy^2$ because x and y are syntactically different. It is conceptually straightforward to modify AIS in the future to be able to represent a derivative's properties more abstractly (like representing the properties of the mth derivative of a function with respect to its nth argument independently of the particular symbol(s) being used for the nth argument). However, the ability to describe derivative properties that *are* dependent on syntactically different arguments is a useful one to keep. For example, if x can take on different values from y, then $d^2f(x)/dx^2$ may indeed have different properties from $d^2f(y)/dy^2$.

The last part of the input gives the sequence of actions (transformations) that is iterated. The sequence is partitioned into *phases* so that each part of a sequence is put into exactly one phase and each part where different actions are occurring is put in a separate phase. What is desired is that all the important and possibly extreme parameter values appear at the end of some phase. The specific requirements are that the phases must be chosen so that 1) every part of a sequence (including all the parts with parameter value changes) is put in exactly one phase, and 2) during each phase, every parameter is either monotonically non-decreasing or non-increasing in value.

Beyond these two requirements, a model-builder is free to divide a sequence into as few or many phases as desired. As an example, look at Figure 2.1, where the values of the parameters A and B versus time (for one iteration) are given. A model-builder may put each of the five marked intervals into a separate phase. An alternative is to have two phases, with intervals 1 and 2 in one phase and intervals 3 through 5 in the other. Other groupings of the intervals are also possible, as is dividing an interval over more than one phase. One constraint on the grouping is that if two intervals of an iteration belong to one phase, then so do all the intervals in between those two (intervals 1 and 5 count as being adjacent). Another constraint is that intervals 2 and 3 have to be in different phases because parameter B is increasing in interval 2 and decreasing in 3. For a similar reason, intervals 1 and 5 have to be in different phases.

A model-builder might violate these requirements if the violation's consequences are judged to be negligible. For example, a modeler may deem some pressure to be constant during some time period and therefore may put that period into one phase, when in fact the pressure at first rises and then falls a little.

For each phase, the input description needs to supply an expression for every

[1] The derivative of a constant parameter here makes sense and may need to be described because: 1) the function itself is not constant, only the arguments are; and 2) one may need to describe how an argument's value being different would affect the function's "output".

[2] Otherwise, all the heart valves will open, letting blood flow freely through the heart.

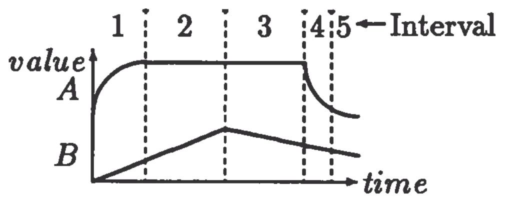


Figure 2.1: Example of Possible Intervals for Phases

parameter that changes in value during that phase. For a periodic parameter, the corresponding expression gives that parameter's value at the end of the phase.[3] For an accumulating parameter, the expression gives the change in that parameter's value each time that phase occurs. An expression may have algebraic and the more common transcendental functions. The expression's arguments can consist of constant parameters, periodic parameters' values at the beginning or end of that phase, and/or accumulating parameters' change in values[4] each time that phase occurs.

The limitations on describing parameter changes are to assure that each occurrence of a phase alters the parameters by the same constant amount. Without some restrictions on how phases alter parameters, it will be hard to impossible for AIS to determine the effects of steadily iterating the sequence of actions. There are at least two interesting alternatives to having constant alterations. The first is a generalization of constant alterations. In the current version of AIS, a particular parameter changes by the same constant amount each time a particular phase occurs. In the generalization, what needs to stay constant will be not the amount of change, but rather the *change* in the amount of change (or an even higher order of change). The second is having the alterations form a converging series [49, Ch. 18]. Neither of these alternatives has been needed so far to model a "steadily running" device.

It is sometimes difficult to provide expressions for the periodic parameter values at the end of a phase. For example, one might not be able to explicitly give the pressure at any point in a water pipe circuit. Unfortunately, if one provides only changes to the periodic parameter values, finding their actual values during the sequence would be impossible or hard, involving symbolically solving simultaneous (nonlinear) equations. With only changes in their value solved for, periodic parameters would be just like accumulating parameters that have a zero net change on each sequence iteration.

Each phase also has a list of the conditions that either are true by definition or need to be true for the phase to occur as stated. The conditions are inequalities between expressions and numbers. Note that the definitions of phase expressions and conditions are slightly different from the definitions given earlier for static conditions between constant parameters.

AIS makes the "closed world" assumption that all changes are mentioned. So if

[3]Due to the requirements on choosing phases, a periodic parameter's value at a phase's beginning and the preceding phase's end is the same. And because the sequence iterates, the last phase in the sequence is also considered to "precede" the first phase.

[4]Only the change in value can be referred to because it stays the same from one iteration of the sequence to the next. The actual value changes with each iteration of the sequence.

some phase's description does not mention a new value for a parameter, that parameter is assumed not to change in value during that phase.

Here is an example of an input description for a phase. Let X_b stand for parameter X's value at the beginning of a phase, X_e for the value at the end, and X_c for X's change in value when the phase occurs. Furthermore, let a be an accumulating parameter, q and r be periodic parameters, and c be a constant parameter. The sample phase description is:

$$(5 \leq q_e), \quad q_e = (c + a_c), \quad a_c = (q_b \cdot r)$$

Whenever this phase occurs: r's value is constant, q is ≥ 5 at the phase's end, a changes by the product of q's value at the phase's beginning and r's value during the phase, and q ends with c's value plus the change in a's value.

2.2 Preliminary Processing

Before producing any of the desired output, AIS needs to solve the equations given in the phase description and to check for obvious inconsistencies between the equations and given conditions.

To solve the equations, AIS computes for each phase: the change in value for each accumulating parameter, and the beginning and end values for each periodic parameter. These values and changes are expressed in terms of constant parameters. The beginning value of each periodic parameter is taken from that parameter's value at the end of the previous phase. The solver currently handles only simple substitutions of the solved for the unsolved. Complicated equations like quadratics are left unsolved.

As an example of equation solving, suppose the equations

$$V_e = Y, \quad A_c = (V_e - V_b), \quad W_c = (P \cdot A_c)$$

are given, where Y is a constant and P is a periodic parameter that does not change during the phase. Let AIS find $V_b = Z$ and $P = Pi$ by looking at the values of V_e and P_e in the previous phase (Z and Pi are constants). Then AIS derives $V_e = Y$, $P = Pi, \quad A_c = Y - Z, \quad W_c = Pi \cdot (Y - Z)$.

To check for obvious inconsistencies, AIS enters the solved equations, the assumption that the rate of sequence repetition is positive, and the conditions given in the input (with periodic and accumulation parameter values substituted by the appropriate expression of constants) into the Bounder system [36, 38]. This system checks for consistency by deriving an upper and lower numeric bound for every constant parameter. An inconsistency is declared if some parameter's lower bound is greater than its upper bound. Bounder derives the bounds with the *bounds propagation* and *substitution* methods. The former method reasons over numeric bounds. The latter method will also perform substitutions of symbolic expressions for symbols. For example, if $c > d+5$, then the latter can find a lower bound on $(c-d)$ of $c-(c-5) = 5$. In addition to these methods, the Bounder system uses an algebraic simplifier. Bounder is also used to perform the bounding and inequality testing needed in the steps described below to produce the output.

2.3 Output

After performing the above equation solving and inconsistency checking, AIS can infer the following about continually repeating the input sequence: 1) the extreme values of a periodic parameter, 2) the average rate of change in an accumulating parameter, including numeric bounds on that rate and the relative contribution of each phase to that rate, and 3) how that rate would differ if a constant symbol or function had a different value (sensitivity analysis).

To try to derive the minimum and maximum values of a periodic parameter p is fairly easy. The requirements for the phase description input assures that the extreme periodic parameter values can be found at the end of some phase. So AIS just needs to look for p's value at the end of every phase (p_e) and find the possible minimums and maximums from among those values.

To derive the average rate of change in an accumulating parameter a, AIS locates the change in that parameter's value (a_c) during each phase of a sequence, adds all those changes together, and then multiplies the sum by the rate of cycle repetition. Next, AIS finds numeric bounds on this rate. Then AIS tries to determine which phases helped to increase or decrease this rate by observing which phases have a_c values that are bounded above and/or below by zero. As an example of deriving a rate of change, let A be an accumulating parameter and R be the rate of sequence repetition. Furthermore, let two phases in this sequence alter A's value. One phase has $A_c = C$ and the other has $A_c = K$, where C and K are constant parameters. Then the average rate of change in A is $dA/dt = R \cdot (C + K)$.

After deriving an average rate for a, AIS can observe how that rate would be different if any one constant symbol or function were different. For each symbol, AIS takes the first two (symbolic) derivatives of the rate with respect to that symbol, obtains numeric bounds on those derivatives, and tries to determine which phases helped to increase or decrease each derivative. Each constant symbol is considered to be independent of all other symbols. AIS performs the phase determination task by looking at the derivatives (with respect to the symbol) of each phase's contribution to the rate (the phase's a_c value multiplied by the sequence repetition rate) and observing which are bounded above and/or below by zero. Those phases with a derivative of a_c that is > 0 made a positive contribution to the derivative, etc.

At present, AIS also tries to plot a "qualitative" graph of the rate versus each constant symbol. The first derivative described above provides slope information and the second provides convexity information. AIS makes the assumption that the curve for the rate versus each constant is smooth (differentiable). If the second derivative can be both more or less than zero, AIS gives up. Otherwise, depending on how the second derivative is bounded by zero and on how the first derivative's bounds relate to zero, AIS determines which of the following shapes the curve may possibly have:

⟍, –, ⟋, ◟, ◡, ◞, ◜, ◠ and/or ◝.

For example, if the first derivative is < 0 and the second is $= 0$ (such as when the rate is $-3x$ and the symbol is x), then the curve shape is ⟍. However, if the second is instead > 0 (such as when the rate is $\exp[-x]$) then the shape is ◟. If the first

1st derivative	2nd derivative	Possible curve shapes
$=0$	$0<$	impossible curve
≤ 0	$0<$	╰
$0\leq$	$0<$	╯
any value	$0<$	╰ , ∪ , ╯
<0	$=0$	╲
$=0$	$=0$	—
$0<$	$=0$	╱
≤ 0	$=0$	╲ , —
$0\leq$	$=0$	— , ╱
any value	$=0$	╲ , — , ╱
$=0$	<0	impossible curve
≤ 0	<0	╮
$0\leq$	<0	╭
any value	<0	╭ , ∩ , ╮
<0	$0\leq$	╰ , ╲
$=0$	$0\leq$	—
$0<$	$0\leq$	╱ , ╯
≤ 0	$0\leq$	╰ , ╲ , —
$0\leq$	$0\leq$	— , ╱ , ╯
any value	$0\leq$	╰ , ╲ , — , ╱ , ╯ , ∪
<0	≤ 0	╮ , ╲
$=0$	≤ 0	—
$0<$	≤ 0	╱ , ╭
≤ 0	≤ 0	╮ , ╲ , —
$0\leq$	≤ 0	— , ╱ , ╭
any value	≤ 0	╮ , ╲ , — , ╱ , ╭ , ∩

Table 2.1: Restrictions on the possible curve shapes from derivative information

derivative has no bounds, but the second is < 0, then the possible shapes are ╭ , ∩ or ╮. Table 2.1 shows the restrictions on the possible curve shapes given knowledge on how the first and second derivatives are bounded with respect to zero. If the second derivative can be both more than and less than zero, no inferences can be made about the curve shape.

In the future, the QS system [37, 39] will probably be used to perform the plotting. The advantage of QS is that it can detect complications like discontinuities and sketch curves with such complications. However, before QS can be used, it needs to be extended to handle functions for which derivative and smoothness information exists, but where the exact analytic form is unknown. Such functions are often used in system descriptions.

Besides deriving the effects of symbols having different values on a rate, AIS also derives the effects of functions having different values. One cannot take a derivative with respect to a function. But if one wants to observe how rates would be different if function f were larger in value, one can substitute $f(x) + e(x)$ for every occurrence of $f(x)$ in the rate (making the side assumption that $\forall x : [e(x) > 0]$), symbolically subtract the original rate from this altered rate, and bound the difference. If the difference is > 0, then if f were larger, the rate would be also, and so on.

Chapter 3

AIS Examples

Three examples of using AIS are presented in this chapter. The input to run them in the implementation is given in Appendix E.6.

The first concerns a normal ventricle (part of the heart). It is the most detailed in illustrating the mechanics of AIS and in comparing AIS's results with results either derived by others by hand or determined empirically from experiments. The example shows how some of these previous results are inaccurate even though they are consistent with experimental results.

The second example is on a ventricle with a disease called mitral stenosis. The model in this example is larger and more ambiguous ("qualitative") than in the first example. The example shows that AIS can handle fairly ambiguous models, but that the results will reflect that ambiguity.

The third example changes domains and is on a steam engine. This example is like the second in that it is larger than the first. But unlike the second one, the steam engine model is a lot more precise on the forms of the functions involved, and AIS's output reflects this. This example also shows how some of AIS's results can be incorporated into a steady-state model.

While examining these examples, one may notice that AIS has many known shortcomings that are termed "conceptually straightforward to fix in the future." The reason they haven't been fixed yet is that things that are "conceptually straightforward" may not be straightforward and easy to program, and the shortcomings cited are examples of such things.

3.1 Normal Ventricle

This section describes the current version of AIS running on a model of the beating of the part the human heart called the left ventricle.[1] The example given in this section is the most detailed in illustrating AIS's mechanics and in comparing AIS's results with results either derived by hand or from empirical experiments.

[1]The description is based on various texts and articles [35, 41] [7, Ch. 13: Mechanisms of Cardiac Contraction and Relaxation] and makes many assumptions. One assumption is that blood is an incompressible fluid without inertia.

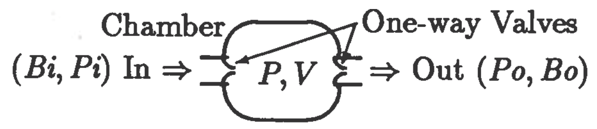


Figure 3.1: Left Ventricle

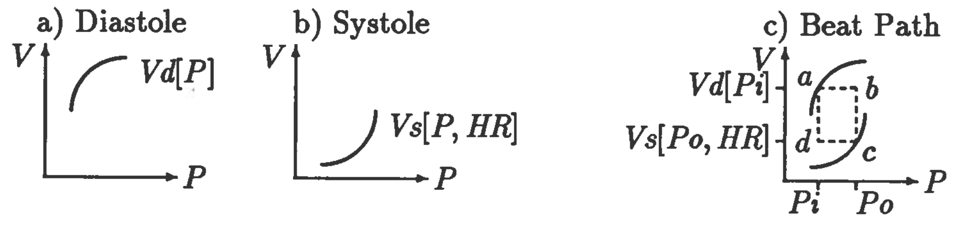


Figure 3.2: Curves for a Normal Left Ventricle

The ventricle (shown in Figure 3.1) is a chamber with two one-way valves: one valve lets in blood from the lungs at a pressure of Pi, and the other valve lets out blood going to the rest of the body at a pressure of Po. The chamber consists of muscle which can either relax or contract. When relaxed (*diastole*), the ventricle's volume (V) versus pressure (P) curve ($Vd[P]$) is roughly as shown in Figure 3.2a (the P and V axes are interchanged from their usual positions). When contracted (*systole*), the V versus P curve ($Vs[P, HR]$) is roughly as shown in Figure 3.2b. The symbol HR appears because with Vs, V decreases as the rate at which the ventricle contracts and relaxes increases. This rate is known as the heart rate (HR). Figure 3.2c shows with a dashed line the V versus P path that ventricle takes as it contracts and relaxes once (a beat sequence): 1) The ventricle contracts, but no blood moves. So, V stays the same while P increases to Po. Move from a to b in the diagram. 2) The ventricle continues contracting, but now, blood is ejected out the output valve. P stays the same while V decreases to $Vs[Po, HR]$. Move from b to c. 3) The ventricle now starts to relax and the blood movement stops. V becomes constant as P decreases to Pi. Go from c to d. 4) The ventricle continues relaxation, but now blood enters from the input valve. P stays the same while V increases to $Vd[Pi]$. Go from d back to a.

The input to AIS has the following: The symbol HR gives the rate at which the ventricle beat sequence repeats. The constants are Pi, Po, $Vd[Pi]$ and $Vs[Po, HR]$.[2] The periodic parameters are P and V. The accumulating parameters are the amount of work done by the blood in moving through the ventricle (W), and the amount of blood that has gone into the ventricle (Bi) and out of the ventricle (Bo). The static conditions on the constants are:

$$Pi < Po, \quad Vd[Pi] > Vs[Po, HR], \quad 0 \leq Vd[Pi], \quad 0 \leq Vs[Po, HR],$$

[2] Pi and Po are assumed to be constant during the ventricle beats. These assumptions then force $Vd[Pi]$ and $Vs[Po, HR]$ to be also constant during the beats.

$$0 < d(Vd[Pi])/d(Pi), \quad 0 > d^2(Vd[Pi])/d(Pi)^2, \quad 0 < \partial(Vs[Po,HR])/\partial(Po),$$
$$0 < \partial^2(Vs[Po,HR])/\partial(Po)^2, \quad 0 > \partial(Vs[Po,HR])/\partial(HR).$$

The first two conditions ($Pi < Po$ and $Vd[Pi] > Vs[Po,HR]$) set up the proper operating conditions for a ventricle to pump blood. The rest of the conditions describe the shape of $Vd[Pi]$ and $Vs[Po,HR]$.

There are four phases in the sequence. Each phase has a name, condition(s), and equation(s) for value changes. In order, the phases are (as before, π_b and π_e stand for the periodic parameter π's value at the beginning and end of the phase respectively, and α_c stands for the accumulating parameter α's change in value during the phase):

1. Isovolumetric Contraction: $0 \leq V, \quad P_e = Po$.
   The ventricle is contracting and both valves are shut. Assumptions: The valves do not leak or move (the latter would let blood move with the moving valve), so V stays constant. Also, the ventricle is strong enough (and V is high enough) so that a pressure of Po is reached.

2. Ejection: $0 \leq V_b$, $0 \leq V_e$, $V_e = Vs[Po,HR]$, $W_c = -P \cdot Bo_c$, $Bo_c = V_b - V_e$.
   The ventricle is contracting, but the output valve is open, letting blood out. Assumptions: Despite the ventricle pumping blood out, the area by the ventricle output stays at a constant pressure of Po. P is constant at Po: blood pumps out of the contracting ventricle fast enough so that P does not rise above Po; if P every drops just below Po, the output valve immediately closes, ceasing blood flow, and the contracting ventricle will repressurize with the remaining blood in the chamber so that P is immediately at Po again.

3. Isovolumetric Relaxation: $0 \leq V, \quad P_e = Pi$.
   The ventricle is relaxing and both valves are shut. Assumptions: The valves do not leak or move (the latter would let blood move with the moving valve), so V will stay constant. Also, the ventricle is elastic enough (and V is low enough) so that a pressure of Pi is reached.

4. Filling: $0 \leq V_b, \quad 0 \leq V_e, \quad V_e = Vd[Pi], \quad W_c = P \cdot Bi_c, \quad Bi_c = V_e - V_b$.
   The ventricle is relaxing, but the input valve is open, letting blood in. Assumptions: Despite the ventricle taking blood in, the area by the ventricle input stays at a constant pressure of Pi. P is constant at Pi: blood enters the relaxing ventricle fast enough so that P does not drop below Pi; if P every rises just above Pi, the input valve immediately closes, ceasing blood flow, and the relaxing ventricle will depressurize with the blood in the chamber so that P is immediately at Pi again.

AIS takes in this input (parameters and expressions) and solves the equations in the following manner: First, AIS scans all the phases to find what aspects of the periodic and accumulating parameters need to be solved and what other periodic and accumulating parameter values need to be found in order to solve these "aspects" (dependencies of the aspects). For example, in the *ejection* phase, P, V, W, and Bo are all the periodic and accumulating parameters of interest. For the accumulating parameters W and Bo, the aspect of interest is W_c and Bo_c, their change in value during the phase. From the phase's equation for W_c, solving for W_c's value requires that one have the phase's P and Bo_c values. Similarly, Bo_c requires the values of V_b and V_e. For a periodic parameter that keeps a constant value during the phase, such as P (no equation for P_e is given), the aspect of interest is that constant value, which will just be labeled with the parameter name itself (P). "Solving" such aspects in this case requires that one look up the parameter's value at the end of the preceding phase (and as mentioned before, the last phase is considered to precede the first phase). For a periodic parameter that changes in value during the phase, such as V, the aspects of interest are its values at the beginning (V_b) and end (V_e) of the phase. Its value at the beginning of the phase is gotten by looking up the parameter's value at the end of the preceding phase. Its value at the end of the phase is gotten by solving the appropriate equation in the phase. In this example, the phase's V_e equation has no unsolved parameter values ($Vs[Po, HR]$ is a constant and is considered already "solved"). As a result of this scan, the *ejection* phase has the following aspects to be solved and their dependencies:

$$W_c \leftarrow (P, Bo_c), \quad Bo_c \leftarrow (V_b, V_e), \quad P\triangleleft, \quad V_e \leftarrow (), \quad V_b\triangleleft,$$

where $\gamma\triangleleft$ stands for γ needing to be solved by looking at γ's parameter value at the end of the previous phase, and $\alpha \leftarrow (\beta_1, \beta_2, \ldots)$ stands for α needing to be solved and the solution depends on the values for β_1, β_2, etc. The aspects to be solved (and their dependencies) for all the phases are as follows:

1. Isovolumetric Contraction: $V\triangleleft, \quad P_b\triangleleft, \quad P_e \leftarrow ()$.

2. Ejection: $P\triangleleft, \quad V_b\triangleleft, \quad V_e \leftarrow (), \quad W_c \leftarrow (P, Bo_c), \quad Bo_c \leftarrow (V_b, V_e)$.

3. Isovolumetric Relaxation: $V\triangleleft, \quad P_b\triangleleft, \quad P_e \leftarrow ()$.

4. Filling: $P\triangleleft, \quad V_b\triangleleft, \quad V_e \leftarrow (), \quad W_c \leftarrow (P, Bi_c), \quad Bi_c \leftarrow (V_b, V_e)$.

Now that AIS knows what it needs to look for, it repeatedly scans the above list of aspects and dependencies. Whenever AIS finds an aspect whose dependencies have all been solved, it solves that aspect's value and takes the aspect off the list of items to be solved. In this example, on the first scan of the above list, AIS first notices that P_e in the first phase is not dependent on anything, so it can be solved by using the appropriate phase equation to get $P_e = Po$. Then, looking at the second phase, AIS notices the same conditions for V_e and so finds that $V_e = Vs[Po, HR]$. AIS also notices that this phase's P has the same value as P's value at the end of the first phase (P_e, which has been solved), so $P = Po$ is also derived. In the third phase, all aspects are solvable: P_e depends on nothing (use the equation $P_e = Pi$), V is the same as V_e of the previous phase (so $V = Vs[Po, HR]$), and P_b is the same as last phase's P end value (so $P_b = Po$). At this point, the last phase is mostly solvable: V_e is dependent on nothing ($V_e = Vd[Pi]$), P and V_b have the same value as the corresponding parameters at the end of the previous phase ($P = Pi$ and $V_b = Vs[Po, HR]$), and AIS can solve for Bi_c by substituting in the just solved for V_b and V_e values into the phase equation $Bi_c = V_e - V_b$ to get $Bi_c = Vd[Pi] - Vs[Po, HR]$. At the completion of this first scan, the following are solved:

1. Isovolumetric Contraction: $P_e = Po$.
2. Ejection: $V_e = Vs[Po, HR]$, $P = Po$.
3. Isovolumetric Relaxation: $V = Vs[Po, HR]$, $P_b = Po$, $P_e = Pi$.
4. Filling: $V_b = Vs[Po, HR]$, $V_e = Vd[Pi]$, $P = Pi$, $Bi_c = Vd[Pi] - Vs[Po, HR]$.

Additional scans of the above list of aspects results in AIS solving the rest of the aspects to get:

1. Isovolumetric Contraction: $V = Vd[Pi]$, $P_b = Pi$, $P_e = Po$.
2. Ejection: $V_b = Vd[Pi]$, $V_e = Vs[Po, HR]$, $P = Po$,
   $W_c = -Po \cdot (Vd[Pi] - Vs[Po, HR])$, $Bo_c = Vd[Pi] - Vs[Po, HR]$.
3. Isovolumetric Relaxation: $V = Vs[Po, HR]$, $P_b = Po$, $P_e = Pi$.
4. Filling: $V_b = Vs[Po, HR]$, $V_e = Vd[Pi]$, $P = Pi$,
   $W_c = Pi \cdot (Vd[Pi] - Vs[Po, HR])$, $Bi_c = Vd[Pi] - Vs[Po, HR]$.

After this "aspect" solving, AIS substitutes in the solutions to the phase conditions. For example, in the *ejection* phase, the condition $0 \leq V_b$ becomes $0 \leq Vd[Pi]$. After these substitutions, the solved phase equations and conditions are:

1. Isovolumetric Contraction: $0 \leq Vd[Pi]$, $V = Vd[Pi]$, $P_b = Pi$, $P_e = Po$.
2. Ejection: $0 \leq Vd[Pi]$, $0 \leq Vs[Po, HR]$, $V_b = Vd[Pi]$, $V_e = Vs[Po, HR]$,
   $P = Po$, $W_c = -Po \cdot (Vd[Pi] - Vs[Po, HR])$, $Bo_c = Vd[Pi] - Vs[Po, HR]$.
3. Isovolumetric Relaxation: $0 \leq Vs[Po, HR]$, $V = Vs[Po, HR]$, $P_b = Po$, $P_e = Pi$.
4. Filling: $0 \leq Vs[Po, HR]$, $0 \leq Vd[Pi]$, $V_b = Vs[Po, HR]$, $V_e = Vd[Pi]$,
   $P = Pi$, $W_c = Pi \cdot (Vd[Pi] - Vs[Po, HR])$, $Bi_c = Vd[Pi] - Vs[Po, HR]$.

Now AIS checks all the equations and conditions for inconsistencies. None are found.

After this solving and consistency checking, AIS discovers that while the beat sequence is iterating, the periodic parameter P ranges from a lower value of Pi to an upper value of Po. V ranges from $Vs[Po, HR]$ to $Vd[Pi]$.

AIS also discovers the following average rates of change for the accumulating parameters and bounds on those rates:

$$\begin{aligned} dW/dt &= HR \cdot ((Pi \cdot (Vd[Pi] - Vs[Po, HR])) + (-Po \cdot (Vd[Pi] - Vs[Po, HR]))) \\ \frac{d(Bi)}{dt} &= \frac{d(Bo)}{dt} = HR \cdot (Vd[Pi] - Vs[Po, HR]) > 0 \qquad (3.1) \end{aligned}$$

The accumulating parameter rates were derived by summing all the changes in an accumulating parameter's value that occur in a sequence and then multiplying the sum by the rate of sequence iteration. For example, the accumulating parameter

W changes in value during the *ejection* ($W_c = -Po \cdot (Vd[Pi] - Vs[Po, HR])$) and *filling* ($W_c = Pi \cdot (Vd[Pi] - Vs[Po, HR])$) phases. Sum these two changes together and multiply by HR, the rate of iteration, to get the above equation for dW/dt.

AIS does not always find the tightest bounds on the rates. For example, one can show that $dW/dt < 0$ by noting that

$$dW/dt = HR \cdot (Vd[Pi] - Vs[Po, HR]) \cdot (Pi - Po),$$

which is a product of two positive values with a negative value, but the bounding mechanism cannot pick this up.

After finding and bounding the rates, AIS looks at the contributions of the phases to these rates. In this example, AIS discovers that the *ejection* phase is the only phase to affect $d(Bo)/dt$, making it as positive as it is. Similarly, the *filling* phase is the only phase to affect $d(Bi)/dt$. AIS can deduce that the *ejection* and *filling* phases are the ones that affect dW/dt, but cannot deduce how they affect dW/dt because AIS doesn't know whether Pi and Po are bounded above or below by 0. Declaring that Pi and Po are positive (their usual sign) would let AIS deduce that *filling* increases dW/dt while *ejection* decreases it.

After finding the rates, AIS derives and bounds the first two derivatives of those rates with respect to each constant symbol, and tries to give the shape of the curve of each rate versus each constant. For $d(Bi)/dt$, its first derivative with respect to HR is > 0, but no bounds are found for the second derivative. No curve shape is deduced. With respect to the constant Pi, the first derivative is > 0 but the second is < 0. Assuming smoothness, AIS deduces a ╭ shape for $d(Bi)/dt$ versus Pi. This shape is consistent with the Frank-Starling mechanism [35, p. 212]. With respect to Po, both derivatives are < 0, so the curve has a ╮ shape. These results also apply to $d(Bo)/dt$. As a check on the ventricle model, these rate shape results are compared to experimental results. The results for Pi and Po agree [40] in that the corresponding AIS and experiment curves have the same general shapes (signs of the first and second derivatives are the same). For HR, the AIS and experimental results are incomparable because the latter came from intact systems where changing HR can change Pi and Po.

For the rate dW/dt, the only bound AIS can derive is that this rate's second derivative with respect to either Pi or Po is > 0. So for dW/dt versus either Pi and Po, the possible curve shapes are ╰, ◡ or ╯.

As for the Vd and Vs functions, AIS deduces that if Vd were larger, both the $d(Bi)/dt$ and $d(Bo)/dt$ rates would be also. But if Vs were larger, these rates would be smaller. These results agree with the description in [41].

When modeling a circulatory system that has been averaged over many heart beats and is in a steady-state, such as done in [15, 41] and Appendix B, most of the system's mechanics can be modeled by using direct current electrical circuit analogies, such as [pressure drop] = [resistance]·[flow] and [pressure] = [amount]/[compliance]. Too complicated to be modeled this way is the part of the mechanics that relates the Pi, Po, HR, Vs, and Vd for each ventricle to the average rate at which blood flows through that ventricle ($d(Bi)/dt = d(Bo)/dt$). Current modeling efforts either directly use empirically derived relationships (like [40]) or derive the needed equations by hand from an AIS-input-like description (done in [41]). AIS can perform the latter derivations automatically: equation (3.1) found by AIS for $d(Bi)/dt$ provides the desired relationship for the left ventricle. The right ventricle is similar. Actually, to use equation (3.1) numerically, one must be more specific about the Vs and Vd curves, such as specifying that $Vd[x] = \log x$.

Other than needing more specific curve shapes, the AIS $d(Bi)/dt$ equation is similar to the equations derived by others. The differences are caused by modeling with slightly different sets of assumptions and beliefs on what relationships exist and are important.

Sagawa [40] experimentally measured the effects of different Pi and Po values on the flow of blood ($d(Bi)/dt = d(Bo)/dt$) through the left ventricles of dogs. The results were numerically fitted to a relationship (curve) of the form (translated to the notation used in this thesis):

$$d(Bi)/dt = K_1 \cdot (Pi - Pi_0) \cdot (1 - \exp[-(1 - Po/Po_{max})/(K_2 \cdot (Pi - Pi_0))]),$$

where K_1, Pi_0, Po_{max} and K_2 are constants. This result agrees with AIS's result in that both have a positive first derivative for rate $d(Bi)/dt$ with respect to Pi and a negative first and second derivative for that rate with respect to Po. The major difference between this result and AIS's result is that this result does not consider the effects of HR at all. This omission is not surprising given that HR's effects were never tested in the experiments. Another difference is that with Sagawa, the minimum Pi and maximum Po needed to keep $d(Bi)/dt$ above zero are given by the simple thresholds Pi_0 and Po_{max} respectively. With the AIS result, the minimum Pi is a more complex function of Po, and similarly with the maximum Po and Pi. A possible reason for this difference is that Sagawa determined the effects of Pi and Po on $d(Bi)/dt$ separately in the experiments and then combined the resulting equations. A third difference is that the effects of Pi have been linearized somewhat to simplify the relationship: the actual data in the reference indicates that at large values of Pi, $d(Bi)/dt$ starts to increase sub-linearly with respect to Pi, which agrees with AIS's result rather than the equation fitted in the reference.

Sato and associates [41] have built a simultaneous equation model of the cardiovascular system at steady-state. The model was built to show the effects of heart failure (the heart muscle gets weaker or less elastic) and to help find the optimum drug dosages for heart failure therapies. Among the equations are the ones that give $d(Bi)/dt$ for each ventricle (as before, the $d(Bo)/dt$ equations are equivalent). These equations have the form (translated to the notation used in this thesis):

$$d(Bi)/dt = K \cdot \ln(Pi - Pi_0) - H \cdot Po + M,$$

where K, Pi_0, H and M are constants. As mentioned above, these $d(Bi)/dt$ equations were derived by essentially carrying out what AIS does by hand. The shape of the $d(Bi)/dt$ versus Pi curve from these equations is , which is the same shape as the one given by the AIS results. A difference between these equations and the ones found by AIS is due to Sato *et al.* having more specific forms for the Vd and Vs functions:

$$\begin{aligned} Vd[Pi] &= (K \cdot \ln(Pi - Pi_0) + Md)/HR \\ Vs[Po, HR] &= (H \cdot Po - Ms)/HR, \end{aligned}$$

where $M = Md + Ms$, so their equations have those more specific forms in place of the Vd and Vs functions. Also, their Vs function has been linearized with respect to Po, so the resulting $d(Bi)/dt$ versus Po curve has a ∖ shape instead of the ╮ shape found by AIS. Another difference is that their versions of the Vd and Vs functions are proportional to $1/HR$, so their $d(Bi)/dt$ is independent of HR instead of increasing with increases in HR. More will be said about this latter difference later on.

Another simultaneous equation model of the cardiovascular system at steady-state was built earlier by Greenway [15]. This model was built to show the effects of a multitude of drugs on the cardiovascular system. Greenway uses some of the same relationships given to AIS as input. But a relationship[3] comparable to equation (3.1) is never explicitly derived. Instead, Po is solved out by the addition of the parameters for the arterial capacitance and the body's resistance to blood flow. However, by noting that $d(Bi)/dt = HR \cdot SV$, where SV is the stroke volume, one can rearrange Greenway's equations into one comparable to equation (3.1) to derive (translated to the notation used in this thesis):

$$d(Bi)/dt = HR \cdot ((Pi + K \cdot F_A) \cdot C_{DV} - Po/E_{max} - V_D),$$

where K, F_A, E_{max} and V_D are constants, and C_{DV} is a "constant" that decreases as Pi gets very large. This equation matches the ones produced by AIS and Sato *et al.* in that all three predict a ╭ shape for the $d(Bi)/dt$ versus Pi curve (C_{DV} decreases in size as Pi increases). And like with Sato and associates, this equation has more specific forms in the place of the Vd and Vs functions in AIS's result. In this equation:

$$Vd[Pi] = (Pi + K \cdot F_A) \cdot C_{DV} \quad \text{and} \quad Vs[Po, HR] = Po/E_{max} + V_D.$$

Also like Sato *et al.* and unlike AIS's result, the Vs function in this equation has been linearized with respect to Po, so the resulting $d(Bi)/dt$ versus Po curve also has a ∖ shape. On the other hand, this equation predicts that $d(Bi)/dt$ will increase as HR increases, which is what the AIS result predicts but not Sato *et al.*'s result. A difference between this equation and the ones given by both Sato *et al.* and AIS is that this one has some terms to account for the affects of the atria (the $K \cdot F_A$ term) while the other two do not (they were given models that assume that the atrial effects are either negligible or can be folded into the expressions for the ventricles). Also, unlike the AIS result, Vs in this equation is independent of HR.

This comparison of AIS's results to previous work on steady-state ventricle models shows that the former is fairly similar to the latter. Furthermore, the existing differences are due to different assumptions being made about the ventricles, not to deficiencies in AIS itself. Two major differences between AIS's results and the existing models are that the latter have more specific relationships for the volume versus pressure curves than the former and that these more specific curves are also more linearized. In addition, in Sagawa's and Sato *et al.*'s ventricular models, the blood flow rate ($d(Bi)/dt = d(Bo)/dt$) is independent of the heart rate (HR), which is often quite inaccurate, especially during exercise or other times of increased venous return [7, p. 414] [35, p. 222]. Also, even when this independence is true (when a person

[3] $d(Bi)/dt$ as a function of HR, Pi, Po, Vd and Vs.

is at rest), [35, p. 222, 294] attributes the constancy of $d(Bi)/dt$ as HR increases to a decline in Pi. So the independence arises from interactions between parts of the cardiovascular system (the interactions that cause Pi to decline as HR increases), not from the ventricle itself, as is implied by the two models.

3.2 Ventricle with Mitral Stenosis

This next example is of a model of a left ventricle in which the mitral (input) valve cannot open wide enough to let blood flow freely through that valve [7, Ch. 33: Valvular Heart Disease]. The model in this section is larger and more ambiguous ("qualitative") than the normal ventricle model in the previous section. AIS can handle this fairly ambiguous model, but the results will reflect the ambiguities.

The defective mitral valve in this ventricle causes a pressure drop across the valve during the *filling* (4) phase: the pressure inside the ventricle (P) is lower than the pressure of Pi at the input. Also, after the *filling* phase, there will be less blood in the ventricle than if the mitral valve were normal.

Like in the previous example of the normal ventricle, the amount of blood in the ventricle at the end of *filling* is dependent on Pi. In addition, in this example, this amount of blood is also dependent on the amount of time spent in *filling*. The longer the ventricle spends in *filling*, the more time it has to let more blood in to raise P to closer to Pi. Two other factors that influence the amount of blood at the end of *filling* are the amount of blood at the start of *filling* ($Vs[Po, HR]$) and the amount of blood the ventricle can hold ($Vd[Pi]$) at the given input pressure Pi. All else being equal, an increase in any of these four factors increases the amount of blood in the ventricle at the end of *filling*. A fifth factor affecting this amount of blood is the time needed for a contracted ventricle to fully relax. The faster it can relax, the more fully the ventricle can fill.

Added to these partial dependencies of the amount of blood in the ventricle at the end of *filling* on various constants is the overall effect that this amount will increase as Pi increases. This overall effect is the net of the direct effect and indirect effects via effects on filling time and the maximum volume. Reasons for this include the fact that a relaxing ventricle will let blood enter the ventricle earlier as Pi increases and the fact that a higher Pi value will force more blood through the defective input valve. The latter reason is what the partial effect of Pi mentioned in the preceding paragraph is about. Another way to look at this is that a higher Pi should lead to a higher pressure in the ventricle (P) at the end of *filling*. P is an increasing function of volume, so a higher P at the end of *filling* means that the volume of blood at this time is also higher.

The amount of time that can be spent in the *filling* phase is limited by $1/HR$, the amount of time that is available for all the phases of a heart beat cycle. Also, as HR increases, ventricular muscle contracts and relaxes faster [23].

The input to AIS is similar to the input given in the normal ventricle example. The constant parameters are as before plus the following additions and changes: $Tr[HR]$ is the amount of time needed to fully relax the ventricle. $Tc[HR]$ is amount of time needed to fully contract the ventricle.[4] Note that when HR is high enough, the ventricle may not fully contract or relax. $Tf[Pi, Po, HR, Tc[HR], Tr[HR]]$ is the function that gives the amount of time spent in the filling phase (abbreviation is $Tf[\ldots]$). $Vd2[Vs[Po, HR], Vd[Pi], Tf[\ldots], Tr[HR], Pi]$ (abbreviation is $Vd2[\ldots]$) is now the function that gives the *filling* phase's V_e. $Vd^{-1}[Vd2[\ldots]]$ now gives the same phase's P_e. The last new constant is $Vd^{-1}[Vs[Po, HR]]$, which gives *filling*'s P_e if no blood were to enter the ventricle during *filling* (when the only blood in the ventricle after the *filling* phase was already there as the ventricle started to relax).

The periodic parameters do not change, and the only change to the accumulating parameters is that W is eliminated.[5] The static conditions on the constants are as before plus the following additions: the relations

$$0 \leq Tr[HR], \quad 0 > d(Tr[HR])/d(HR), \quad 0 \leq Tc[HR], \quad 0 > d(Tc[HR])/d(HR)$$

describe the relaxation and contraction time functions; $0 \leq Tf[\ldots] \leq \frac{1}{HR}$ is the condition that describes the *filling* time function;

$$\begin{gathered} 0 < \partial(Vd2[\ldots])/\partial(Vs[Po, HR]), \quad 0 < \partial(Vd2[\ldots])/\partial(Vd[Pi]), \\ 0 < \partial(Vd2[\ldots])/\partial(Tf[\ldots]), \quad 0 > \partial(Vd2[\ldots])/\partial(Tr[HR]), \\ 0 < \partial(Vd2[\ldots])/\partial(Pi), \quad 0 < d(Vd2[\ldots])/d(Pi) \end{gathered}$$

give the conditions of the function for the amount of blood at the end of the *filling* phase (note that the last derivative describes the overall effect of a different Pi value on $Vd2$); the shape of the inverse of the Vd function as applied to two different arguments is described by

$$\begin{gathered} Vd^{-1}[Vs[Po, HR]] < Pi, \quad 0 < d(Vd^{-1}[Vs[Po, HR]])/d(Vs[Po, HR]), \\ 0 < d^2(Vd^{-1}[Vs[Po, HR]])/d(Vs[Po, HR])^2, \\ 0 < d(Vd^{-1}[Vd2[\ldots]])/d(Vd2[\ldots]), \quad 0 < d^2(Vd^{-1}[Vd2[\ldots]])/d(Vd2[\ldots])^2. \end{gathered}$$

The shape of Vd itself was given in Figure 3.2a.

From this static conditions description, one can observe that two improvements for AIS would be for it to be able to handle function descriptions independent of the arguments and to be able to link descriptions of functions and their inverses. In this example, Vd^{-1}'s derivatives had to be described even after Vd's derivatives were given. In fact, Vd^{-1}'s derivatives had to be described twice, once for each set of arguments. Both of these abilities are conceptually straightforward to add in the future.

Like the previous example, there are four phases in the sequence, and except for the W_c equations being taken out, the first three phases are as before. The new equations for the fourth (*filling*) phase are as follows:

$$\begin{gathered} 0 \leq V_b, \quad 0 \leq V_e, \quad Bi_c = V_e - V_b, \quad V_b < V_e, \quad V_e < Vd[Pi], \\ Vd^{-1}[Vs[Po, HR]] < P_e, \quad P_e < Pi, \quad V_e = Vd2[\ldots], \quad P_e = Vd^{-1}[Vd2[\ldots]]. \end{gathered}$$

[4]The Tc function should not be confused with T_c, which represents the change in value of an accumulating parameter T.

[5]With mitral stenosis, the ventricle no longer is assumed to fill at constant pressure, so the equations for work given in the normal ventricle example are no longer valid.

The changes in the V_b and V_e expressions from the previous example's *filling* phase indicate that V still increases during *filling*, but not as much as it did with a normal ventricle ($V_e = Vd[Pi]$). The P_e expressions reflect the effects of the new V values on the P values.

Two caveats should be mentioned about the model being given here. The first is that P is nonmonotonic during the *filling* phase: P starts at Pi, drops, and then rises back towards Pi. Fortunately, the expressions in the model for accumulating parameters do not depend on P being monotonic during *filling*, so one can get away with the nonmonotonicity for the average rate determinations. The second caveat is that no relationship is given between how strongly a ventricle can contract (Vs) and how fast it can contract (Tc). A similar shortcoming is true for the corresponding relaxation functions Vd and Tr. So any conclusions AIS reaches concerning these functions (none are reached) should be treated cautiously. The relationships were not modeled because I am not sure of what the relationships are.

Given this input, AIS deduces that while the beat sequence is iterating, the periodic parameter P ranges from a lower value of $Vd^{-1}[Vd2[\ldots]]$ to an upper value of Po. V ranges from $Vs[Po, HR]$ to $Vd2[\ldots]$. AIS's deduction for P's minimum value is wrong. This is due to (as mentioned above) P not being monotonic during the *filling* phase. The actual minimum P value is below the one given by AIS, but it is unknown by how much.

AIS also derives the following rates and bounds on those rates: $d(Bi)/dt = d(Bo)/dt = (Vd2[\ldots] - Vs[Po, HR])) \cdot HR > 0$.

Out of all of the first two derivatives of these rates with respect to each constant symbol, AIS can only bound the first derivative with respect to Pi to > 0. The second derivative with respect to Pi is unboundable because no information is given on the second derivatives of the $Vd2$ function. With respect to Po, the derivatives are unbounded because Po being larger has an unknown effect on $Vd2[\ldots]$: a larger Po would increase $Vd2[\ldots]$ via an increase in $Vs[Po, HR]$, but would also have an unknown effect on $Vd2[\ldots]$ via an unknown effect on $Tf[\ldots]$. HR being larger would also have an unknown effect on $Vd2[\ldots]$ because HR has an unknown effect on $Tf[\ldots]$. In fact, even if increasing HR would have the direct effect of decreasing $Tf[\ldots]$ by shortening the amount of time allotted to a beat cycle, HR being larger would still have an ambiguous overall effect on $Tf[\ldots]$: HR being larger would now directly decrease $Tf[\ldots]$, but still have an unknown indirect effect on it via changing (decreasing) $Tc[HR]$ and $Tr[HR]$.

AIS can only deduce the effects of a difference in one function, $Vd2$: if $Vd2$ were larger, both $d(Bi)/dt$ and $d(Bo)/dt$ would also be larger. The current implementation of AIS cannot deal with the Vs, Vd, Tr, Tc and Tf functions because it cannot handle functions that are arguments of other functions. This shortcoming is conceptually straightforward to repair in the future.

The unambiguous results produced by AIS make sense. These results are incomparable to the qualitative descriptions given in such sources as [7, Ch. 33: Valvular Heart Disease]: the results deal with a ventricle in isolation, while the sources deal with the ventricle in an intact circulatory system.

The problem with the AIS results is that most of them are ambiguous. Less ambiguous and more quantitative results can be achieved by applying a fluid mechanics

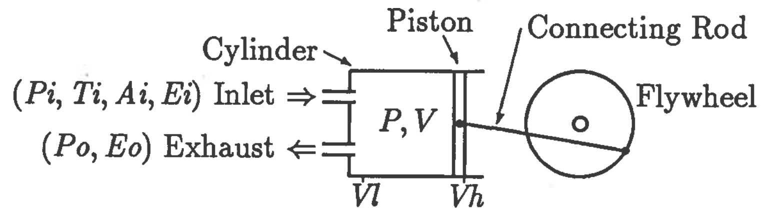


Figure 3.3: Steam Engine

formula that states that the fluid flow through a valve is proportional to both the valve area and the square root of the pressure difference across the valve [7, Ch. 9: Cardiac Catherization]. However, this formula assumes that the fluid flow is steady when it occurs and that the valve area is constant while the valve is open. Both assumptions are only approximately true with the mitral valve. Hence this attempt to model mitral stenosis without using the fluid mechanics formula. I will leave it to others to try modeling with intermediate numbers of assumptions.

3.3 Steam Engine

AIS has been applied to a second example of an iterative system, a simple steam engine (simplified version of the ones in [8]). The engine model is fairly precise concerning the forms of the functions involved, and AIS's output reflects this. This section ends with a demonstration of how to incorporate some of AIS's results into a steady-state model.

The engine in this example (shown in Figure 3.3) has one cylinder and a piston that slides back and forth along the inside of that cylinder. The piston also covers the main opening in the cylinder. The sequence of actions is that the piston slides further out in the cylinder and then back in. As the piston slides out, the volume contained by the cylinder and piston combination (V) increases, moving from a low value of Vl to a high of Vh. Steam (at a pressure of Pi and a temperature of Ti) is let into the cylinder from $V = Vl$ to $V = Vex$. From $V = Vex$ to $V = Vh$, no steam is let in or out (steam in the cylinder expands adiabatically [17]). At $V = Vh$, the inertia of a rotating flywheel (connected to the piston via a connecting rod) pushes the piston back into the cylinder. As the piston slides back in, V decreases from a value of Vh back to Vl. From $V = Vh$ to $V = Vcp$, steam is let out of the cylinder via an exhaust port (at a pressure of Po). From $V = Vcp$ to $V = Vl$, no steam is let in or out (steam in the cylinder is compressed adiabatically). At $V = Vl$, the sequence repeats. The model makes many assumptions, including one that steam behaves almost like an ideal gas.[6]

The model is in terms of the following parameters: The symbol RPM (for revolu-

[6]Steam is assumed to behave like an ideal gas except that in addition to translational motion, the molecules may store energy in the form of rotational or vibrational motion using an equipartition of energy. As a result, k (a constant to be described later) may be greater than 3/2.

tions per minute) gives the rate of sequence repetition. The constants are Pi, Ti, Po, Vl, Vex, Vcp, Vh, R and k. R is the constant in the ideal gas law $P \cdot V = n \cdot R \cdot T$, and $k \cdot R$ is the molar specific heat of steam at constant volume [17]. The periodic parameters are V and the pressure inside the cylinder (P). The accumulating parameters are the amount of work done in driving the piston (W), the energy of all the steam entering the cylinder (Ei) and leaving the cylinder (Eo), and the amount of steam that has entered the cylinder (Ai).

Static conditions on the constants are:

$$0 < Po < Pi, \quad 0 < Vl < Vex < Vh, \quad Vl < Vcp < Vh, \quad 0 < Ti, \quad 0 < R, \quad \tfrac{3}{2} \le k$$

All but the last three conditions (ones for Ti, R and k) are to set up the proper operating conditions for a steam engine. All the conditions that place numeric lower bounds (mostly zeroes) on the constants are due to the way that nature is modeled. For example, with ideal gases, where the law $P \cdot V = n \cdot R \cdot T$ holds, P, V, n, R and T all have to be non-negative, and in fact, as long as some gas is present, even a zero value is not reachable.

The sequence has six phases, which are in order:

1. Open steam inlet: $0 \le V$, $0 \le P_b \le P_e$, $0 < Ti$, $0 \le Ei_c$, $0 \le Ai_c$, $P_e = Pi$, $Ei_c = k \cdot (P_e - P_b) \cdot V$, $Ai_c = (P_e - P_b) \cdot V/(R \cdot Ti)$.
   The inlet port opens and the exhaust port stays closed. Assumptions: The inlet can supply steam at a pressure of Pi and a temperature of Ti even though steam is getting sucked into the cylinder. The cylinder pressurizes to $P = Pi$ fast enough so that the piston does not have the chance to move before the pressurization occurs (V stays constant).

2. Admit steam: $0 \le P$, $0 \le V_b \le V_e$, $0 < Ti$, $0 \le W_c$, $0 \le Ei_c$, $0 \le Ai_c$, $V_e = Vex$, $W_c = P \cdot (V_e - V_b)$, $Ei_c = (1 + k) \cdot W_c$, $Ai_c = W_c/(R \cdot Ti)$.
   The exhaust port stays closed, the inlet port stays open and the piston is sliding out of the cylinder (pushed out by the steam). Assumptions: The inlet can supply steam at a pressure of Pi and a temperature of Ti even though steam is getting sucked into the cylinder from the piston sliding out and increasing V. As a result, P stays constant at Pi. The environment around the engine (what the open end of the cylinder is exposed to) is a gas or liquid that is at a pressure less than Pi.

3. Adiabatically expand steam: $0 \le P_b$, $0 \le P_e$, $0 \le V_b$, $0 \le V_e$, $V_e = Vh$, $P_e = P_b \cdot (V_b/V_e)^{(1+1/k)}$, $W_c = k \cdot P_b \cdot V_b \cdot (1 - \sqrt[k]{V_b/V_e})$.
   Both valves are closed. Assumption: the cylinder's steam (being at a higher pressure than the environment) expands according to the adiabatic expansion laws for gases.

4. Open exhaust: $0 \le V$, $P_b \ge P_e \ge 0$, $0 \le Eo_c$, $P_e = Po$, $Eo_c = k \cdot (P_b - P_e) \cdot V$.
   The exhaust port opens and the inlet port stays closed. Assumptions: The exhaust can maintain a pressure of Po even though steam is getting sucked into the exhaust. The cylinder depressurizes to $P = Po$ fast enough so that the

piston does not have the chance to move before the depressurization occurs (V stays constant).

5. Exhaust steam: $0 \leq P$, $V_b \geq V_e \geq 0$, $0 \geq W_c$, $0 \leq Eo_c$, $V_e = Vcp$, $W_c = P \cdot (V_e - V_b)$, $Eo_c = (1+k) \cdot P \cdot (V_b - V_e)$.
The exhaust port stays open, the inlet port stays closed and the piston is sliding into the cylinder (pushed in by the flywheel inertia). Assumptions: The exhaust can maintain a pressure of Po even though steam is getting pushed into the exhaust from the piston sliding in and decreasing V. As a result, P stays constant at Po.

6. Adiabatically compress steam: $0 \leq P_b$, $0 \leq P_e$, $0 \leq V_b$, $0 \leq V_e$, $V_e = Vl$, $P_e = P_b \cdot (V_b/V_e)^{(1+1/k)}$, $W_c = k \cdot P_b \cdot V_b \cdot (1 - \sqrt[k]{V_b/V_e})$.
Both valves are closed. Assumption: the cylinder's steam is compressed according to the adiabatic expansion laws for gases by the flywheel's inertia pushing the piston in.

In addition, each phase has the conditions $3/2 < k$, and $0 < R$, and has the additional assumptions that no steam heat is lost through the cylinder or piston walls, that the engine parts are frictionless and that the valves and piston/cylinder "seam" are not leaky. In the last two phases, if the environment's pressure is greater than Po, then the flywheel is not needed to push the piston in: the environment will push the piston from the open end of the cylinder. If the flywheel is needed, last two phases assume that the flywheel has enough inertia to push the piston appropriately.

This phase description comes from using some simplifying assumptions to link the verbal description in [8] with equations from the chapters on the kinetic theory of gases and thermodynamics in [17]. The *open steam inlet* and *open exhaust* phase descriptions assume that a large opening valve connects the cylinder to a much (infinitely) larger body of gas (the steam source or exhaust) and so P immediately changes to the pressure of that much larger body. This pressure change at constant volume in turn alters the amount of steam (and energy contained in the steam) in the cylinder as given by the equations. The *admit steam* and *exhaust steam* phase descriptions assume that the piston is moving (so the volume of the cylinder/piston combination is increasing or decreasing in size) with the steam at a constant pressure. So the equations are the ones for a gas changing volume at a constant pressure. The descriptions for the two phases with adiabatically expanding/compressing steam use the equations for a gas that is adiabatically expanding/compressing (the basic equation is that $P \cdot V^{(1+1/k)}$ is a constant): the only energy loss or gain for the cylinder's steam comes from pushing or being pushed by the cylinder. Steam does not lose or gain energy from contact with the cylinder or piston walls, and no steam is let in to or out of the cylinder.

After solving these phase equations, AIS discovers that while the sequence is iterating, the periodic parameter V ranges from a lower value of Vl to an upper value of Vh. P ranges from Po to Pi. In addition, AIS finds that $Pi \cdot (Vex/Vh)^{(1+1/k)}$ (P's value at the end of adiabatic expansion) may be as low as Po and $Po \cdot (Vcp/Vl)^{(1+1/k)}$ (P's value at the end of adiabatic compression) may be as high as Pi.

AIS also deduces the following average rates of change for the accumulating parameters:

$$
\begin{aligned}
d(Ai)/dt &= (Vl\cdot(Pi - Po\cdot(Vcp/Vl)^{(1+1/k)})/(R\cdot Ti) \\
&\quad +Pi\cdot(Vex - Vl)/(R\cdot Ti))\cdot RPM \\
d(Ei)/dt &= (k\cdot Vl\cdot(Pi - Po\cdot(Vcp/Vl)^{(1+1/k)}) \\
&\quad +Pi\cdot(Vex - Vl)\cdot(1+k))\cdot RPM \\
d(Eo)/dt &= (k\cdot Vh\cdot(Pi\cdot(Vex/Vh)^{(1+1/k)} - Po) \\
&\quad +Po\cdot(1+k)\cdot(Vh - Vcp))\cdot RPM \\
dW/dt &= (Pi\cdot(Vex - Vl) + k\cdot Pi\cdot Vex\cdot(1-\sqrt[k]{Vex/Vh}) \\
&\quad +Po\cdot(Vcp - Vh) + k\cdot Po\cdot Vcp\cdot(1-\sqrt[k]{Vcp/Vl}))\cdot RPM
\end{aligned}
$$

These rates all have expressions of the form $\alpha\cdot RPM$, where α is the change in the accumulating parameter's value per sequence iteration.

For the rate $d(Ai)/dt$, α consists of two parts: $Vl\cdot(Pi-Po\cdot(Vcp/Vl)^{(1+1/k)})/(R\cdot Ti)$ is the contribution from the *open steam inlet* phase, when steam enters the cylinder (at a constant volume of Vl) to increase P from $Po\cdot(Vcp/Vl)^{(1+1/k)}$ (P at the end of last iteration's *adiabatically compress steam* phase) to the inlet pressure of Pi. $Pi\cdot(Vex - Vl)/(R\cdot Ti)$ is the contribution from the *admit steam* phase, when steam enters the cylinder at a constant pressure of Pi to push the piston so that V increases from Vl to Vex.

These contributions of steam are the energy input into the engine. So for $d(Ei)/dt$, α also has two parts: $k\cdot Vl\cdot(Pi-Po\cdot(Vcp/Vl)^{(1+1/k)})$, the energy of the steam entering during the *open steam inlet* phase, and $Pi\cdot(Vex - Vl)\cdot(1+k)$, the energy of letting in steam during the *admit steam* phase.

$d(Eo)/dt$ is another rate with a two part α: $k\cdot Vh\cdot(Pi\cdot(Vex/Vh)^{(1+1/k)} - Po)$ is the loss in energy during the *open exhaust* phase, when steam escapes from the cylinder (at a constant volume of Vh) to decrease P from $Pi\cdot(Vex/Vh)^{(1+1/k)}$ (P at the end of the preceding *adiabatically expand steam* phase) to the exhaust pressure of Po. $Po\cdot(1+k)\cdot(Vh - Vcp)$ is the loss in energy during the *exhaust steam* phase from pushing steam out the cylinder (V decreases from Vh to Vcp) at a constant pressure of Po.

Each of the four phases where the piston moves (V changes) affects dW/dt, so its α has four parts: $Pi\cdot(Vex - Vl)$ and $Po\cdot(Vcp - Vh)$ are the contributions from steam pushing (*admit steam* phase) and being pushed (*exhaust steam* phase) by the piston at constant pressure respectively. $k\cdot Pi\cdot Vex\cdot(1-\sqrt[k]{Vex/Vh})$ and $k\cdot Po\cdot Vcp\cdot(1-\sqrt[k]{Vcp/Vl})$ are the contributions from steam pushing (*adiabatically expand steam* phase) and being pushed (*adiabatically compress steam* phase) by the piston adiabatically respectively.

AIS can determine which phases affect these rates, but cannot always determine how these phases affect the rates. In addition, the fact that the $d(Ai)/dt$, $d(Ei)/dt$ and $d(Eo)/dt$ rates are all positive also eluded AIS's inference abilities. A reason for this shortcoming is that the bounding algorithms do not always find the tightest bounds on a given expression. An example of this shortcoming is that in phase 1 (open steam inlet), AIS finds that the condition $0 \leq Ai_c$ becomes

$$0 \leq (Vl \cdot (Pi - Po \cdot (Vcp/Vl)^{(1+1/k)})/(R \cdot Ti).$$

However, the bounding mechanism never stores this condition in this form, so it cannot bound Ai_c's value of

$$(Vl \cdot (Pi - Po \cdot (Vcp/Vl)^{(1+1/k)})/(R \cdot Ti)$$

in this phase to be ≥ 0, even though this condition was just asserted. Because of this, AIS cannot tell that phase 1 did not decrease Ai, nor can AIS bound the rate $d(Ai)/dt$ to be > 0. Another example is that the bounding mechanism cannot conclude that $(a-b)/(b-a) = -1$ even when told that $a > b$ (to eliminate divide by 0). It is conceptually straightforward to fix these cited specific examples of shortcomings, but in general, one has to find a specific type of shortcoming before one knows what to add to eliminate it. A theoretical limit on what bounding mechanisms can accomplish is given in [34], which proves the following: Let $A(x)$ be an arbitrary expression that can be composed of addition, subtraction, multiplication, and function composition of the following primitives: $\log 2$, π, $\exp(x)$, $\sin(x)$, x and rational numbers. Then determining the truth of $\exists x$:[x is real and $A(x) < 0$] is undecidable. Another limit of a theoretical nature is that the problem of determining the satisfiability of a set of arithmetic constraints is NP-hard. One can show this property by mapping in linear time the problem of determining the satisfiability of a boolean expression in conjunctive normal form (CNF) into a problem of the former type.[7] The problem for boolean expression satisfiability is NP-complete [2, p. 383].

In addition, this general shortcoming affects AIS's ability to bound the derivatives of the rates with respect to various constants. Among the inferences that AIS missed were ones to indicate that the $d(Ai)/dt$, $d(Ei)/dt$ and $d(Eo)/dt$ rates all would be larger if either RPM or k were larger, that the $d(Ai)/dt$ versus k curve has a ⌒ shape and the $d(Ai)/dt$ versus Ti curve has a ╰ shape, and that dW/dt would be larger if either Vex or Pi were larger, but would be smaller if either Vcp or Po were larger.

However AIS could still deduce many of the relationships, especially ones for the Ai, Ei and Eo rates. Among the ones found by AIS: With respect to the constant parameter Pi, all three of these rates have first derivatives of > 0 and second derivatives of 0. This means that the plot of each of these rates versus Pi has an upward slope with no curvature (shape is ∕). Bounds on the first two derivatives of these rates with respect to some other constants are given in Table 3.1. Also given in that

[7]The mapping is as follows: Let the boolean expression in CNF have the form $\prod_i \alpha_i$, where each α_i has the form $\sum_k \beta_{ik}$, each β_{ik} is either x_{ik} or $\neg x_{ik}$, each x_{ik} is a boolean variable, the summations are disjunctions, and the products are conjunctions. To convert this boolean expression into a set of arithmetic constraints in linear time, let 0 and 1 represent *false* and *true* respectively. Then, letting the summations now represent additions, converting each $\neg x_{ik}$ into $(1 - x_{ik})$, and replacing $\prod_i \alpha_i$ with $\alpha_i > 0$ for each value of i, $\prod_i \alpha_i$ in the original notation (boolean) is satisfiable *iff* $\forall i : [\alpha_i > 0]$ is true in the new notation (arithmetic). In the worst case, each two symbols in the original notation produce five symbols in the new notation (The worst case is $\neg x_{ik}$, which becomes $(1 - x_{ik})$ in the new notation.), so the problem reformulation is in linear time.

	Constant Parameter								
	Po			*Vex*			*Vcp*		
Rate	1d	2d	sh	1d	2d	sh	1d	2d	sh
$d(Ai)/dt$	< 0	$= 0$	╲	$0 <$	$= 0$	╱	< 0	< 0	╮
$d(Ei)/dt$	< 0	$= 0$	╲	$0 <$	$= 0$	╱	< 0	< 0	╮
$d(Eo)/dt$	none	$= 0$	╲, ─ or ╱	$0 <$	$0 <$	╯	< 0	$= 0$	╲

Table 3.1: Steam engine: some derivatives and curve shapes

table are the possible shapes of the "rate versus constant" curves, assuming that the curves are smooth. The "1d" and "2d" columns give bounds on the first and second derivatives respectively, and the "sh" columns give the possible curve shapes. For example, both the first and second derivatives of $d(Eo)/dt$ with respect to *Vex* are > 0, so the shape of the $d(Eo)/dt$ versus *Vex* curve is an upward slope with an upward curve (╯). The results make intuitive sense. For example, if *Pi* were larger, more steam at higher pressure would have entered the cylinder on each sequence repetition and there would also be more steam to exhaust on each repetition. So the *Ai*, *Ei* and *Eo* rates should all be larger as a result. Similarly, as *Vex* increases, steam will be let into the cylinder longer on each cycle, so on each repetition, more steam will enter and there will be more steam to exhaust.

Compared to the other rates, AIS can make relatively few deductions about dW/dt. Some of this is probably due to dW/dt's expression being larger than the others, and some of this is probably due to less being deducible about the dW/dt expression's general properties. For example, depending on the exact values of various constants, dW/dt can either be positive or negative and can either increase or decrease as *RPM* increases.

3.3.1 Using the Engine Equations in a Train Model

One of the motivations of having AIS is to find the steady-state relationships of some quickly iterating sub-system so that one can build a steady-state model of the slower overall system. In this example, the steam engine that AIS just analyzed is placed into a locomotive pulling a large train full of sightseers. The question of interest is to find the steady-state speed of the train on level ground.[8] It is assumed that the train's mass is large enough to smooth out the force variations of the steam engine as it goes through an iteration.

The engine is connected so that one revolution of the locomotive drive wheels corresponds to one iteration of the steam engine. Assuming that the drive wheels do not slip or slide on the tracks, the train's speed $S = RPM \cdot D$, where D is the diameter of the drive wheels. There are three forces on the train:

1. The steam engine, which produces power (dW/dt) at the rate found by AIS earlier in this example.

[8]Level ground is chosen so that no gravitational potential energy effects are present.

2. Air resistance to the train's movement. The force is assumed to be proportional to the train's speed with a positive proportionality constant of B.

3. Friction from the train wheels moving over the tracks. The force is assumed to be constant at a positive value of F.[9]

One can derive the train's steady state speed by looking at the train's rate of change in kinetic energy (KE), which is equal to the net rate of change in energy due to the three forces:[10]

$$d(KE)/dt = dW/dt - (B \cdot S) \cdot S - F \cdot S. \tag{3.2}$$

AIS gives the dW/dt result, which can be put in the form of $RPM \cdot WPI$, where

$$\begin{aligned} WPI &= Pi \cdot (Vex - Vl) + k \cdot Pi \cdot Vex \cdot (1 - \sqrt[k]{Vex/Vh}) \\ &\quad + Po \cdot (Vcp - Vh) + k \cdot Po \cdot Vcp \cdot (1 - \sqrt[k]{Vcp/Vl}) \end{aligned}$$

is the work done per steam engine iteration. At a steady state, the train's speed is constant, so its kinetic energy is constant ($d(KE)/dt = 0$). Combining this with the two equations just given and the relationship $S = RPM \cdot D$ yields

$$\begin{aligned} 0 &= RPM \cdot WPI - B \cdot S \cdot RPM \cdot D - F \cdot RPM \cdot D \\ S &= (WPI/D - F)/B. \end{aligned}$$

This last relationship uses AIS's rate results for W to find the train's overall steady state speed as a function of various steam engine and overall train parameters.

In fact, in the case of this train model, one can also find the train's change in velocity over time when one or more of these parameters is altered. To do so, note that $KE = M \cdot S^2/2$, where M is the train's mass. Then assuming a constant M, $d(KE)/dt = M \cdot S \cdot (dS/dt)$. Combine this with equation 3.2 and the relationships $dW/dt = RPM \cdot WPI$ and $S = RPM \cdot D$ to get

$$dS/dt = (WPI/D - F - B \cdot S)/M.$$

This is a first order linear differential equation, which can be solved by using the technique given in [49, Ch. 20.5]. Given S_0, the train's speed at $t = 0$ (just before the parameter change(s)), the train's speed over time after a parameter(s) change is:

$$S = \frac{WPI/D - F}{B}[1 - \exp(-B \cdot t/M)] + S_0 \cdot \exp(-B \cdot t/M).$$

At $t = 0$, $S = S_0$. As t increases, the altered parameter(s) take effect ($\exp(-B \cdot t/M)$ decreases from one to zero) and the train smoothly changes from this initial speed to the new steady-state speed of $(WPI/D - F)/B$.

[9]One model of friction over a smooth surface is that it is proportional to the mass. The mass in this example is assumed to stay the same: the loss of fuel and water to the steam engine over time is assumed to have a negligible effect on the train mass.

[10]The change in energy due to air resistance or friction equals the force multiplied by the train's speed.

The derivations in this subsection give an example of how one can use AIS's results for an iterative sub-system (steam engine) to help construct a steady-state model of an overall system (train moving at a steady speed). By handling the iterative sub-system, AIS makes the modeling of the overall system easier. With this particular example, a model of how the system responds to a step change in some parameter also emerges. Such models for changes will probably be more likely to be derivable for simpler systems.

Chapter 4

Using Probability Bounds: a Simple Example and Alternatives

The thesis now shifts from the problem of analyzing an iterative dynamic system to construct a model to the problem of handling parameter uncertainty when using a model. These next three chapters describe SAB and HMC, two methods that use a model to bound the probability of some steady-state system behavior when given a bound on the joint probability density of the input parameters. This particular chapter has two parts. The first gives a simple example of using SAB and HMC on some bounds of probability densities. The second part describes some current alternatives to using probability density bounds and either SAB or HMC. Following this is a chapter on SAB and then one on HMC.

4.1 Simple Example Using *PVR*

A simple example of using SAB and HMC involves finding a patient's pulmonary vascular resistance (*PVR*) given the constraint

$$PVR = (PAP - LAP)/CO \tag{4.1}$$

and information on the patient's pulmonary arterial pressure (*PAP*), left atrial pressure (*LAP*) and cardiac output (*CO*). *PVR* is of interest because a high value indicates that the heart's right ventricle has to work very hard to keep the blood moving through the lungs [27, p. 234]. Ideally to look at *PVR*, one should have a curve that gives the "cost" of having any particular *PVR* value and use this curve in conjunction with indications of how likely each *PVR* value is to see if *PVR* should be monitored. Alternatively, one could have several ranges of *PVR* values, each with a cost of being in that range. For now, what I could find in the medical literature are partitions of parameters into two ranges with a threshold in between. One threshold condition for *PVR* is $PVR \leq 1.62$ (in *mmHg/(l/min)*). Critically ill surgical patients with values above this are less likely to survive [44, p.54-59].[1] *PAP*, *LAP*, and *CO* have patient

[1]Assume that patients each have a body surface area of $1.74m^2$, the average for humans. The *PVR* threshold has three significant figures (and not two or one) in this example because the threshold

			Correlation Coef.		
NAME	MEAN	STD DEV	*PAP*	*LAP*	*CO*
PAP	23.94	3.38	1.0	.861	.096
LAP	15.29	3.08	.861	1.0	-0.044
CO	6.49	1.20	.096	-0.044	1.0

Table 4.1: *PVR* Example

and time dependent values, and are not easy to measure accurately. Table 4.1 gives statistics for the patient of interest, a heart attack victim. The question is, given information on *PAP*, *LAP*, and *CO* for the patient involved, is *PVR* at all likely to be above the threshold? If so, one ought to monitor *PVR*.

The numbers are close enough so that the answer is not obvious from looking at Table 4.1: For example, substituting the mean values into Equation 4.1 results in $PVR < 1.62$, but increasing *PAP*'s value in the substitution by 3.38 (one standard deviation) while maintaining *LAP* and *CO*'s values would result in $PVR > 1.62$. However, the latter is not that likely to happen because *LAP* tends to increase when *PAP* does (high positive correlation).

So, one has to look at the joint density of *PAP*, *LAP*, and *CO*. Like most statistics, the ones in Table 4.1 are subject to sampling error, and in addition, the density shape is not exactly known. To get around this difficulty, one can hypothesize plausible bounds on the joint density and bound the probabilities of satisfying the criteria given each density bound. Ideally, the set of density bounds used will cover all the possible variations.

In this example, four particular density bounds are considered. They show the kinds of bounds the methods can handle. One-dimensional (marginal) views of these are in Figure 4.1, where the areas under the density bounds are marked by vertical lines. The one on the right is an upper bound, the middle two are lower bounds, and the one on the left can be either. The three left-most input bounds are given to SAB, and the right-most two are given to HMC. As will be described later, the two right-most bounds cover all Gaussian densities where *CO*'s mean is somewhere within a bounded interval and all the other parameters are as given in Table 4.1. The details how of these results are produced are given later.

The first "bound" is a regular joint Gaussian density[2] with the parameters listed in Table 4.1 and a one-dimensional view of it is shown in the left diagram of Figure 4.1. A 1000-sample Monte Carlo simulation with this bound (a normal probability density) indicates that $PVR > 1.62$ about 20% of the time. One can use SAB to place analytic bounds on this figure. SAB is an iterative routine. It produces loose bounds initially and then tightens those bounds as it iterates (see the end of Section 5.2 for details). In this case, SAB was stopped when it had bounded the figure to be between 4% and 57%. This is consistent with the Monte Carlo simulation and with patient data,

given in the reference has three significant figures (but in different units). I am not sure whether all three figures are really significant. I will need to ask surgeons about that.

[2]Being a density (has an area of 1), it is both a lower and an upper density bound.

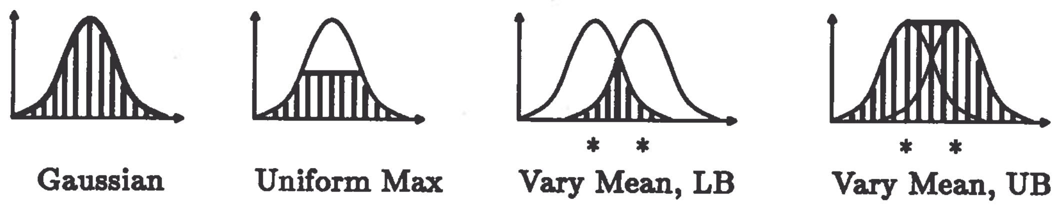


Figure 4.1: Four Density Bounds

where 4 of 17 (23.5%) data points had $PVR > 1.62$.[3]

The second density bound is a

1. joint Gaussian density with the parameters listed in Table 4.1
2. in which the maximum density value is limited to that of a jointly uniform density with the same means and standard deviations.

In other words, the density bound looks like a Gaussian far from the variables' means, but has the low flat top of a uniform density near the means. A one-dimensional view of it is shown in the middle-left diagram of Figure 4.1. Integrating the bound indicates that it includes $\sim$ 70% of the probability mass. SAB was run with this bound and stopped when it analytically bounded $\Pr(PVR > 1.62)$ to be between 4% and 79%. SAB found the 79% figure by finding a lower bound on $\Pr(PVR \leq 1.62)$ and then subtracting it from 100%. These results are again consistent with the patient data.

The third density bound is the lower bound of a Gaussian density where CO's mean is allowed to be anywhere between 6.20 to 6.78.[4] This constraint might have been determined by using information in some confidence interval for CO's mean. The middle-right diagram of Figure 4.1 shows a one-dimensional view of this bound: CO's mean can lie anywhere between the two *'s. The lower density bound is the intersection of the areas under all the densities possible due to allowable variations in CO's mean. Because Gaussian densities are unimodal, the lower bound is the intersection of the areas under the two Gaussian density curves[5] shown. Integrating the bound indicates that it includes $\sim$ 65% of the probability mass. SAB was run with this bound and stopped when it analytically bounded $\Pr(PVR > 1.62)$ to be between 1% and 76%. HMC was also run with this bound. Like SAB, HMC is an iterative algorithm. Unlike SAB, HMC does not produce tighter bounds as it runs. Instead, HMC produces bounds that are not so accurate at first, but become more accurate (smaller standard deviation) as it iterates. When HMC was stopped in this case, it had an estimate of 12% for a lower bound and 44% for an upper bound on $\Pr(PVR > 1.62)$, and with an estimated standard deviation of $< 2\%$ on both figures. Both the SAB and HMC results are also consistent with the patient data.

The fourth density bound is the upper bound version of the third density bound. Instead of taking the intersection of the areas under the Gaussian density curves

[3]Here, the data could have been used by itself to answer the question of whether $PVR > 1.62$ is at all likely. SAB and HMC are meant to be used when such data are not available.

[4]The variances, covariances, and other means could also be allowed to vary.

[5]They are the ones with the extreme CO mean values.

with *CO* means in [6.20,6.78], take the union. When HMC was stopped in this case, it had an estimate of 33% (standard deviation of 1%) for the upper bound on $\Pr(PVR > 1.62)$. As with all upper bounds, all of the probability mass is within the bound.

The main purpose of this small example is to show the type of input SAB and HMC take in, and the type of output they produce. The example does not show the limits of their capabilities, nor is the example one of where they really make a difference. The probability bounds found in this small example (especially the ones found by SAB) are fairly loose. Despite this, some of the bounds do give one useful information on the problem: the 4% and 12% lower bounds on $\Pr(PVR > 1.62)$ indicate that the event $PVR > 1.62$ is not extremely rare. Also, all the runs were stopped early. If one allowed the runs with SAB to continue on, SAB would have produced tighter bounds.

4.2 Alternatives

Current alternatives to using probability density bounds and both SAB and HMC fit into one of four categories. The first category of methods finds all the possible system behaviors (sometimes including impossible ones), but does not tell the likelihood of the behaviors. Such methods include systems either performing qualitative reasoning [6, 54], or providing numeric bounds [36].

Category two methods estimate the distributions of possible outcomes without giving some measure of each estimate's error and will not improve the accuracy of those estimates when given more computation time. One of these methods is to use one of the moment approximation schemes mentioned in Section 1.5 to estimate moments of parameters of interest and then use these estimated moments to specify a density (from a family of densities) for the parameters of interest. [16] combines truncated Taylor series expansions of the model equations to find various moments with the Johnson or Pearson family of distributions. A possibility using lognormal and Gaussian densities only is described in the chapter on GLO. As mentioned in Section 1.5, these moment schemes

1. can be quite inaccurate, and
2. can produce too many equations for a simultaneous equation solver to handle when the equations have to be solved simultaneously.

Björke [5] and Pearl [29, Sec. 7.2] describe similar methods that are more limited in that they assume independent inputs and linearized equations. Another similar method is given by Shachter and his associates [43, 42]. This method assumes linearized equations and Gaussian densities, but does *not* assume independence between variables. The method uses variable transformations to make the random variables more Gaussian and the variables' relationships more linear. How well this last method works depends on a user's ability to find variable transformations that convert the random variables into a set that satisfies the assumptions.

A third category is the set of evidential reasoners [20, 48], which includes most of the current work done on uncertainty in AI. These reasoners can only handle a variable value in terms of the possibility of it belonging to one or more regions in a preset discretization of the possible variable values. For example, blood pressure (*BP*) may be only thought of in terms of being low, normal, or high. This limitation is a problem because what is considered normal, desirable, etc. can change with each use of a model. For example, when trying to lower a patient's *BP*, an acceptable pressure depends on the patient's former normal blood pressure and the patient's ability to withstand therapy side-effects.

Monte Carlo techniques [16, 19, 22], which fall into two general classes, constitute the fourth category. The first class simulates a system by generating samples according to some probability distribution. Most methods in this class cannot handle density bounds. The acceptance/rejection method can handle density bounds, but it is too slow due to the large number of potential samples it rejects. The second class of Monte Carlo techniques integrates the density or density bound involved. These integration techniques include hit-or-miss and sample-mean Monte Carlo. Unfortunately, determining the interval(s) to be integrated over (the region(s) satisfying the criteria) is very hard. The section on HMC gives more details on this problem. Also, as with all Monte Carlo techniques, every answer is inexact and has a standard deviation associated with it.

A variation on the first class of Monte Carlo techniques is described in [21]. In this variation, one does not need to fully specify the joint distribution of the input variables. One just needs to supply the marginal distributions (probability distribution of each variable by itself) and correlation matrix of the input variables. The method then generates samples according to those marginal distributions and rearranges the samples so that their correlations will be "similar" to the given correlation matrix. One can then use these samples in a Monte-Carlo simulation. However, beyond the marginals and the correlations, the nature of the joint distribution of the samples is left to the whim of the method.

Chapter 5

SAB

This chapter describes SAB, one of the two methods presented which use bounds on a probability density to bound the probability of satisfying some criteria. The chapter leads off with a general description of how SAB works. The next section goes back to the example in Section 4.1 to illustrate how SAB worked on that simple example. Following this are sections on the details and limitations of SAB, respectively.

5.1 Overview

SAB successively narrows the probability bound of achieving or failing some criteria by iteratively *splitting* regions comprising a partition of the possible input values *and* then *bounding* both the possible behaviors within the smaller regions and the probabilities of being in those smaller regions (using the input probability density bound). SAB marks the regions whose possibilities always satisfy or fail the criteria.

Figure 5.1 shows two examples of splitting. In the one marked *Behavior*, the criterion is $a \cdot b < 3$, and the original region is $a, b \in [0, 2]$. In this region $a \cdot b \in [0, 4]$, so it sometimes passes and sometimes fails the criterion. Split[1] the region along $a = 1$ into the two sub-regions X and Y. In X, $a \in [0, 1]$, so $a \cdot b \in [0, 2]$. Because the criterion is always satisfied, mark X. In contrast, $a \in [1, 2]$ in Y, which means $a \cdot b \in [0, 4]$, so Y is not marked.

In the example marked *Probability*, $\alpha = (c \in [0, 1])$ is the original region and $f(c)$ is a lower bound on probability density at c. SAB finds the lower bound on the probability of being in α, $\Pr(\alpha) > 0.5$. This bound is the sum of areas q and r, found by multiplying 1, α's length, by 0.5, the lowest value of $f(c)$ in α.[2] Split the region at $c = 0.5$ into the two sub-regions Z and W. By a method similar to the one above, SAB finds a lower bound on $\Pr(Z)$ of 0.5 (sum areas r and s), and a lower bound on $\Pr(W)$ of 0.25 (area q). Sum the lower bounds of $\Pr(Z)$ and $\Pr(W)$ to get a new lower bound of 0.75 on $\Pr(\alpha)$.

As hinted by these two examples, *as long as* the bounding method used tends to reduce the range of possibilities as a region of input values gets smaller, this continued

[1]At present, SAB splits a region by bisecting it.

[2]Better methods of bounding probabilities are described later.

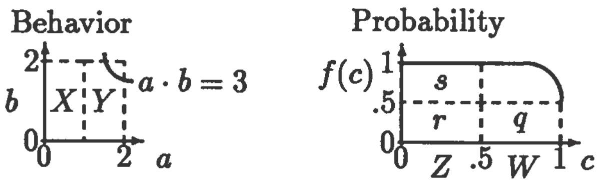


Figure 5.1: Examples of Splitting

splitting will mark more and more of the interval of all possible input values. And *as long as* the bounding method tends to reduce the gap between a density bound's upper and lower bound[3] in a region as the region gets smaller, the bound on the probability of being in a marked region will improve.

To find a lower bound on Pr(satisfy criteria) sum the lower probability bounds of all the regions marked as satisfying the criteria. Similarly, one can find a lower bound on Pr(fail criteria). One minus the latter is an upper bound on Pr(satisfy criteria).

5.2 *PVR* Example Revisited

This section re-examines the *PVR* example in Section 4.1 when using the Gaussian density as a "bound" (first density bound) and SAB as the bounding method. To bound Pr($PVR > 1.62$), SAB looked at the interval of all possible inputs[4] (given to SAB as one region):

$$PAP \in [1.0, 88.0], LAP \in [1.0, 88.0], CO \in [1.0, 100].$$

A lower bound on *PVR*, written $lb(PVR)$, is

$$\max(0, [lb(PAP) - ub(LAP)]/ub(CO)) = 0,$$

and an upper bound ($ub(PVR)$) is

$$[ub(PAP) - lb(LAP)]/lb(CO) = 87.0.$$

PVR can be either greater or less than 1.62, so SAB split the region in two along the *CO* dimension:

$$\begin{array}{ll} subspace1: & PAP \in [1.0, 88.0], LAP \in [1.0, 88.0], CO \in [1.0, 50.5] \\ subspace2: & PAP \in [1.0, 88.0], LAP \in [1.0, 88.0], CO \in [50.5, 100.0] \end{array}$$

[3]Yes, we are bounding a bound here.

[4]Note that the given range of possible values is wider than necessary. For example, one could tighten them to $CO \leq 30$ and $LAP \leq 45$. These wider bounds will not affect the correctness of the results. They will slow SAB down, but probably not by much: the inputs should have little (if things are slightly inconsistent) or no chance of being in the extra area included by the wider bounds, and SAB concentrates first on the regions of the possible input values that have the highest estimated probability of occurring. So when in doubt about the bounds, err on the side of including something that cannot occur.

SAB then checked and split as appropriate. Regions like

$$PAP \in [20.75, 25.47], LAP \in [15.95, 17.32], CO \in [6.41, 7.19], (PVR \in [0.756, 1.484])$$

where PVR is either always $>$, or ≤ 1.62, were marked. SAB found lower bounds on the probabilities of being in these marked regions (the one above has a probability ≥ 0.002).

As SAB recursively splits and checks regions, it tightens the probability bound for satisfying the criteria. When the bound is tight enough, or SAB runs out of time or another resource, it can be stopped. In this example, when SAB was stopped, it gave a lower bound of 0.042 on the probability of being in a passing region (one where $PVR > 1.62$), and 0.438 for a failing region ($PVR \leq 1.62$). As mentioned in Section 4.1, the probability bounds found in this small example are fairly loose. Despite this, the bounds do give one useful information on the problem: the 4% lower bound found for $\Pr(PVR > 1.62)$ indicates that the event $PVR > 1.62$ is not extremely rare. If a tighter bound was desired, one could have restarted SAB with the then current set of regions. Since this joint density bound includes all of the probability mass, SAB can, barring round-off error in the floating point math, get the bound to be arbitrarily tight if given enough computing time. In general, if a joint density bound includes $n \times 100\%$ of the probability mass, SAB can, barring round-off error, get the bound to have a gap of $1.0 - n$ between the upper and lower figure. So if a lower density bound includes 70% of the probability mass, the tightest bound SAB could give on the chances of passing some criteria would have a gap of 0.3 between the lower and upper figures (such as a lower bound of 0.6 and an upper bound of 0.9).

5.3 Details

Various details of SAB method are presented here. The first subsection gives the main loop that SAB iterates. The rest of the subsections describe various parts of this loop. The next subsection describes how SAB ranks regions of the input parameter value space and estimates the probability of being in them. These two activities tell SAB which region to examine next. Following this is a subsection on how SAB bounds the probability of being in a region. The last two subsection are on how SAB splits regions and finds numeric bounds on expressions, respectively.

5.3.1 Main Loop

Perform the following cycle until told to stop:

1. Select the region α with the highest *rank* (see below). SAB can start with either one universal region (as in the example), or any number of predefined regions.

2. What type of region is it?

(a) *Marked* for being known to always satisfy or fail the given criteria. An example is when a region's *PVR* range is 0.0 to 1.2 and the criterion is $PVR \leq 1.62$. Here, split the region into two, and using the given density bound, estimate and bound the greatest lower probability bound of being in each of the two sub-regions. Mark them for the same reason as the original region.

(b) *Unsure.* The region can still either pass or fail the given criteria. An example is when a region's *PVR* range is 0.0 to 2.0 and the criterion is $PVR \leq 1.62$.

   i. If the possibilities of the region (*PVR*'s range in the *PVR* example) have not been bounded yet, bound them (in the *PVR* example, use the given formulas for an upper and lower bound on *PVR*). If the region should be marked, do so and bound the probability of being in it.

   ii. If the possibilities have been bounded, split the region in two. Bound both sub-regions' possibilities, and estimate the greatest lower probability bound of being in each sub-region. If a sub-region should be marked, do so and bound that sub-region's probability.

The probability estimations made are just used to suggest the next best step for SAB by helping to rank the sub-regions. They are *not* used as part of any probability bound.

The only overlap allowed between regions is shared borders. No overlap is permitted if the probability density bound has impulse(s).[5]

5.3.2 Ranking Regions & Estimating Region Probabilities

A region's ***rank*** estimates to what extent splitting it will increase the known lower bound on the probability of either satisfying or failing the criteria. An "unsure" (unmarked) region's rank is the estimated greatest lower probability bound (using the given density bound) of being in that region. Estimate as follows:

1. Observe how many input and parameter sample points (out of a thousand picked using a "density" which resembles the given joint density bound) fall within the region. If > 10 samples (1%) fall inside, the fraction falling inside is the estimate.

2. If ≤ 10 samples fall inside, estimate with a formula that quickly, but approximately integrates the density bound in the region. The *PVR* example uses formula $C_n : 3\text{-}3$ in [47, page 230]: Suppose one wants to integrate the n-dimensional curve $f(x_1, \ldots, x_n)$ inside the rectangular region where

[5] An impulse occurs when part of the bound becomes infinitely high and leads to a non-zero probability of the variables taking on a particular set of values. An example of such a set for the variables PAP and LAP is $(PAP = 45) \wedge (LAP = 30)$.

$\forall i : [l_i \leq x_i \leq h_i]$. This formula makes the following approximation:

$$\int_{l_n}^{h_n} \cdots \int_{l_1}^{h_1} f(x_1, \ldots, x_n) \cdot dx_1 \cdot \ldots \cdot dx_n \approx$$
$$[\prod_{i=1}^{n}(h_i - l_i)] \cdot [\frac{3-n}{3} \cdot f(m_1, \ldots, m_n) +$$
$$\frac{1}{6}\sum_{i=1}^{n}[f(m_1, \ldots, m_{i-1}, l_i, m_{i+1}, \ldots, m_n) + f(m_1, \ldots, m_{i-1}, h_i, m_{i+1}, \ldots, m_n)]],$$

where $m_i = (l_i + h_i)/2$.

These two parts compensate for each other's weaknesses:

1. The first part is bad for low probabilities because any region α will have large gaps between the sample points within it. So many sub-regions of α will have no sample points even though they may have high values for the lower probability density bound.

2. The second part is bad for high probabilities because the regions involved are either large or probably contain a complicatedly shaped part of the density bound.[6] The integration formulas only work well when a region's section of the density bound is easily approximated by a simple polynomial.

A marked region's rank is the gap between the estimated greatest lower probability bound of being in the region and the known lower bound on that probability. This works better than the gap between the upper and lower bounds on the greatest lower probability bound because SAB often finds very loose upper bounds, while the estimates are usually accurate.

5.3.3 Bounding Region Probabilities

The basic way SAB finds a lower bound on the probability of being in a region is to multiply the region's volume[7] by its minimum probability density lower bound value (found by the bounding mechanism described in Section 5.3.5). I derived the *PVR* example's first density bound expression (a Gaussian density) by taking the density parameters (Table 4.1) and substituting them into the general form for a Gaussian density. After some simplification, I got (numbers rounded-off):

$$0.01033\exp(-0.01323(13.70P^2-26.09P{\cdot}L-10.36P{\cdot}C+16.42L^2+10.77L{\cdot}C+28.08C^2))$$

where $P = (PAP - 23.94)$, $L = (LAP - 15.29)$, and $C = (CO - 6.487)$.

[6]Most of the common probability densities only have complicated shapes where the density values are high. I am assuming that this complication will be reflected in the corresponding part of the bound.

[7]For a region α, let its variables x_i ($i = 1 \ldots n$) range between l_i and h_i. Then α's volume is $\prod_{i=1}^{n}(h_i - l_i)$. SAB only deals with n-dimensional rectangular regions.

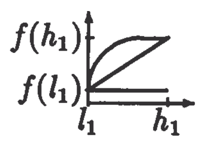


Figure 5.2: One Dimensional Convex Density and Lower Bound

To help tighten this bound, SAB tries to use any monotonicity and/or convexity present in the region's part of the density bound in the following manner (derivations in Appendix A):

Let $f(x_1, \ldots, x_n)$ be the probability density and within a region α let x_i range between l_i and h_i. The probability of being in α is

$$F = \int_{l_n}^{h_n} \cdots \int_{l_1}^{h_1} f(x_1, \ldots, x_n) \cdot dx_1 \cdot \ldots \cdot dx_n.$$

If $\partial f / \partial x_1$ is always > 0 in α, then

$$F \geq [\prod_{i=1}^{n}(h_i - l_i)] \cdot [(\min_{*} f(l_1, x_2, \ldots, x_n)) + (\min \frac{\partial f}{\partial x_1}(x_1, \ldots, x_n)) \cdot (\frac{h_1 - l_1}{2})],$$

where the minimization of f is over the x_2 through x_n values within α ($\min_*$ means that x_1 is NOT part of the minimization) and the minimization of $\partial f / \partial x_1$ is over the x_1 through x_n values within α. This bound is tighter than the basic lower bound:

$$[\prod_{i=1}^{n}(h_i - l_i)] \cdot [\min f(x_1, \ldots, x_n)].$$

Similar expressions can be derived for the other variables and for when $\partial f / \partial x_i < 0$.

If $\partial^2 f / \partial x_1^2$ is always ≤ 0 in α (convex down), then

$$F \geq [\prod_{i=1}^{n}(h_i - l_i)] \cdot [(\min_{*} f(l_1, x_2, \ldots, x_n)) + (\min_{*} f(h_1, x_2, \ldots, x_n))]/2,$$

where the minimizations of f are over the x_2 through x_n values within α. This bound is also tighter than the basic one. See Figure 5.2 for the one dimensional case: the $\cap$ curve is the density, the area under the diagonal line is F's new lower bound, and the area under the horizontal line is the original bound. Similar expressions can be derived for the other variables.

Several methods exist to integrate a region's probability density bound, including Monte Carlo [19] and quadrature (numeric integration) methods [47]. These cannot truly bound the integration error because they only take numeric samples at particular points.

5.3.4 Splitting Regions

SAB may split a selected region α in either step 2a or step 2(b)ii. In either, SAB picks a variable in α to split along and then bisects α. Select the variable as follows: in step 2(b)ii, find the one with the largest difference between its upper and lower bound within the region, normalized by its standard deviation. In step 2a, find the one with the largest apparent variation in the density's slope with respect to it.

5.3.5 Numeric Bounding in SAB

Many of SAB's parts need expressions to be bounded. For expressions with algebraic, logarithmic, and exponential functions (the type in the models to be used), perfect bounding algorithms have not been built. The type of algorithm used here will find bounds that indicate what is truly unachievable,[8] but those bounds may not be the tightest possible. For example, the algorithm may find that $x < 7$ when in fact it can be shown that $x < 3$. I have implemented an augmented version of bounds propagation [36], which does the following interval arithmetic [33]:

- Bound an operation's result using bounds on the operands. For example: $ub(a+b) \leq ub(a) + ub(b)$.
- Bound an operand using bounds on an operation's result and the other operands. For example: $ub(a) \leq ub(a+b) - lb(b)$.

The "bounder" examines expressions and updates bounds with these operations. It *iterates* over the expressions until every one that might produce a change has been examined at least once and all the recent bound changes are below a certain threshold.

5.4 Limitations

SAB has two limitations. First, as the number of inputs increases, the complexity of the density bound often increases to beyond SAB's capacity. In fact, SAB is not fast for small problems either. The examples with SAB in Section 4.1 each took 30 to 60 minutes (1 to 10 thousand iterations) to run on a Symbolics 3640 (or 3650) Lisp machine.[9]

SAB's second limitation is that it can make little use of upper density bounds. This is because SAB derives its bounds by summing the bounds of the sub-regions. On a lower input density bound, SAB sums up lower probability bounds, and on an upper density bound, SAB sums up upper probability bounds. If a few of the upper sub-region probability bounds being summed are loose, say close to one, then the entire sum will be close to one or more (since all the probability bounds are at least zero), which will make for a loose bound. With lower bounds on the other hand, a few loose lower sub-region probability bounds, say close to zero, will not affect the sum very much. For example, let there be 5 sub-regions, each with a probability of 0.1 of containing the measured input values. Then the chances of event α, that the measured values are in one of the sub-regions, is $5(0.1) = 0.5$. If the upper bound on the chances of each of four of these sub-regions is 0.1, and is 0.9 in the fifth sub-region, then an upper bound on $\Pr(\alpha)$ is $4(0.1) + 0.9 \geq 1$, a loose bound. On the other hand, if the lower bound on the chances of each of four of these sub-regions is 0.1, and is

[8]In practice, the accuracy of this may be limited by round-off error.

[9]Part of the reason for the slowness was that the small examples were run with all the bounding machinery needed for the larger problems. No attempt was made to optimize. LISP was used because of its storage management, which was needed to easily handle the newly split regions.

0.0 in the fifth one, then a lower bound on $\Pr(\alpha)$ is $4(0.1) = 0.4$, which is still close to the actual chance of 0.5.

Unfortunately for SAB, if one gives an input density bound where a mean or some other parameter can vary over a wide range, the upper bound becomes much more important than the lower one. This is because as some parameter varies over a wider range, the intersection of the densities with all the different allowable parameter values gets small, so the lower bound covers little (is loose). For example, in the *"vary mean, LB"* bound in Figure 4.1, if the mean is allowed to vary more, the two extreme Gaussian curves depicted will move farther away from each other, and the area in common to them (the lower bound) will shrink. When a lower bound covers little, most of the probability mass will not be covered by the bound and can go anywhere. At this point, one will not be able to infer much on the chances of any input values occurring. An example of a lower bound covering little is one which only has 0.05 (5%) of the probability mass under its curve. If one uses this bound to bound the chances of some event α, one will get a lower bound on $\Pr(\alpha)$ of x and a lower bound on $\Pr(\neg\alpha)$ of $0.05 - x$, where x is a number between 0.0 and 0.05. An upper bound on $\Pr(\alpha)$ is one minus a lower bound on $\Pr(\neg\alpha)$, so the upper bound is $1 - (0.05 - x) = 0.95 + x$. Then the lower bound on $\Pr(\alpha)$ is at most 5% while the upper bound is at least 95%, which does not tell one much about $\Pr(\alpha)$.

While a loose lower density bound is useless, a loose upper density bound can still yield useful probability upper bounds. This is because there may be large regions of possible input values which are known to be very unlikely to occur.

Chapter 6

HMC

These problems with SAB led to trying to apply Monte Carlo (MC) techniques in conjunction with density upper bounds. As mentioned in Section 4.2, the common MC techniques are inadequate. This section describes HMC, a hit-or-miss version of sample-mean Monte Carlo which applies to the problems of interest. The first section describes the original sample-mean MC and why it is inadequate for the problems of interest. The next section describes how to convert sample-mean MC into HMC. The last two sections give examples of applying HMC to a larger problem.

6.1 Sample-Mean Monte Carlo

Sample-mean Monte Carlo (called *crude Monte Carlo* in [19]) estimates the area under a curve c within some region (interval) as follows:

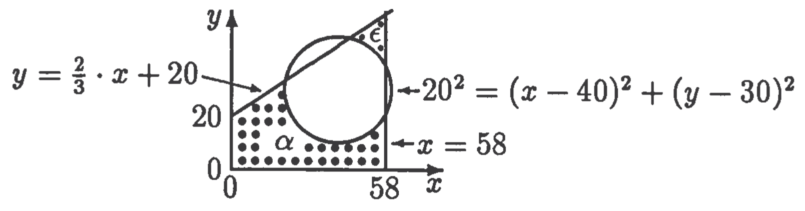
1. Randomly sample the curve in the region of interest using a uniform distribution: first take a random sample of sets of input values in that region, and then for each set of values, $\underline{x}_i$, find $c(\underline{x}_i)$, the value of the curve at $\underline{x}_i$.

2. Estimate the average value of the curve in the region by finding the average value of the $c(\underline{x}_i)$'s: $A_c = \sum_i c(\underline{x}_i)/n$, where n is the size of the sample. The sample standard deviation of the $c(\underline{x}_i)$'s is

$$s = \sqrt{\sum_i (c(\underline{x}_i) - A_c)^2/(n-1)}.$$

   The standard deviation of A_c is estimated by $s_A = s/\sqrt{n}$.

3. Multiply A_c by the size or volume S of the region or interval: $A_c \times S$. This result estimates the area under the curve within the region of interest. The standard error of this estimate is $s_A \times S$. Since S is constant and $s_A \propto 1/\sqrt{n}$, the estimate becomes more accurate as the size of the sample increases. However, since the proportionality is to $1/\sqrt{n}$ and not to $1/n$, each successive addition to the sample has less effect on the accuracy.



$y = \frac{2}{3} \cdot x + 20$
$20^2 = (x - 40)^2 + (y - 30)^2$
$x = 58$
ϵ
α
y
x
20
0
0
58

Figure 6.1: Regions Delineated by the Criteria

As previously mentioned, sample-mean MC is an MC integration technique. Such techniques have the disadvantage that one has to find all the regions of possible input values that satisfy the criteria, and then integrate the curve in those regions. As an illustration of these difficulties, consider an example that has two parameters x and y and the following criteria:

$$0 \leq x \leq 58, \qquad 0 \leq y, \qquad y \leq 2 \cdot x/3 + 20, \qquad 20^2 \leq (x - 40)^2 + (y - 30)^2.$$

To use sample-mean MC, one has to first find the region(s) that satisfy the criteria. In this case, there are two such regions. One region borders the x and y axes and is marked with α and black dots in Figure 6.1. The other region is a "triangular" area marked with ϵ and black dots in Figure 6.1. I do not know of ways of always automatically finding such regions given non-linear criteria.

If there were such ways, non-linear programming (constrained optimization) methods would be able to at least decide when a feasible solution (a solution that satisfies the criteria) exists and when one does not. At present, these methods cannot. For example, consider the optimization of expressions containing algebraic (+, −, ·, /, and exponentiating a variable to a constant power) functions under constraints also containing algebraic functions. A program implementing the algorithm given in [4, Ch. 10] came out tied for first in two comparisons of existing programs for optimizing such problems [4, Ch. 11 & 13]. Part of the algorithm (called "phase 1") is to find a point that satisfies the constraints. As the reference mentions, this part does not always succeed even when such a point exists. So this algorithm cannot always decide if a set of algebraic constraints is satisfiable. The reference did not give any pattern for the failures.

Also, if there were ways of always automatically finding the regions satisfying the criteria, non-linear equation solvers would always work (give a solution when one exists or indicate that none exist): these solvers only search for one point that satisfies the criteria specified by the equations. However, as mentioned in references on the subject ([31, Ch. 9] and [11, Ch. 2] for example), a solver of a system of non-linear equations that always works has not been developed and is quite unlikely to ever be developed.

A theoretical limit on what types of regions can always be automatically found is given in [34]. As mentioned in the steam engine example for AIS (Section 3.3), this reference shows that determining the truth of $\exists x$:[x is real and $A(x) < 0$] is undecidable, when $A(x)$ is an arbitrary expression that can be composed of addition, subtraction, multiplication, and function composition of the following primitives: $\log 2$, π, $\exp(x)$,

$\sin(x)$, x and rational numbers. Section 3.3 also provides another limit of a theoretical nature by demonstrating that deciding if a set of arithmetic constraints is satisfiable is NP-hard.

Even if the satisfying regions can be found, integrating the curve in each of those regions using the sample-mean Monte Carlo method is difficult. The method requires that one knows the size of each region (S in the method description above). The method also needs to sample points in each region according to some known probability distribution (often the uniform distribution). The odd shapes of these regions make it difficult to meet either of these needs (see α and ϵ in Figure 6.1). In fact, finding the size of an odd-shaped region (integrating the volume inside the region) is one of the standard uses for Monte-Carlo techniques. This would create two Monte-Carlo problems where there used to be one. Not much work has been done for sampling from odd-shaped regions. One method is a type of acceptance/rejection Monte Carlo: For each region β,

1. Enclose β inside an n-dimensional rectangle. Make the rectangle as small as possible.
2. Sample uniformly from the rectangle. Reject the sampled points that fall outside of β. Accept the rest. The accepted points will give a uniform sampling of β.

One way to meet both the knowledge about size and sampling requirements is to try to cover the regions exactly with rectangles. Performing sample-mean Monte Carlo in a rectangular region with a uniform distribution is fairly easy. However, covering the satisfying regions exactly with rectangles is impossible.

As a last comment, these problems using sample-mean Monte Carlo all become more difficult as the number dimensions in the input increases.

6.2 From Sample-Mean MC to HMC

Although it is hard to find all the regions of input values that satisfy the criteria, one can easily tell if a given set of values do so. In the example above with Figure 6.1, if one were given the input values of $x = 54$ and $y = 50$, one could substitute these values into the criteria of that example to find that this set of values meets the criteria.

Another observation is the following. Suppose one defines a new curve $c^*(\underline{x})$:

$$c^*(\underline{x}) = \begin{cases} c(\underline{x}) & \text{if } \underline{x} \text{ satisfies the criteria} \\ 0 & \text{otherwise} \end{cases}$$

where $c(\underline{x})$ is the original curve and $\underline{x}$ can take on all possible input values. Then the area under c^* within the region of all possible input values is equivalent to the area under c within the regions of values that satisfy the criteria.

One can use these two ideas to modify sample-mean MC to obtain HMC by substituting all occurrences of $c(\underline{x}_i)$s with $c^*(\underline{x}_i)$s in the three-step algorithm described in Section 6.1. To obtain $c^*(\underline{x}_i)$ for each sample member, $\underline{x}_i$, see if $\underline{x}_i$ satisfies the criteria. If it does, find $c(\underline{x}_i)$; otherwise use 0. When the $\underline{x}_i$s come from the interval

x	0	1	2	3	4	5	6	7	8	9	A_c	s_A
$c(x)$	0.1	0.12	0.14	0.16	0.18	0.20	0.18	0.16	0.14	0.12	.15	.01
$c^*(x)$	0.0	0.0	0.0	0.16	0.0	0.0	0.18	0.16	0.14	0.12	.076	.026
$q(x)$	0.0	11.3	16.7	18.0	17.3	16.7	18.0	23.3	34.7	54.0		

Table 6.1: HMC Example Sample

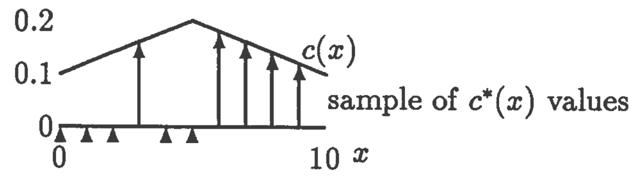


Figure 6.2: HMC Example

of all possible input values, the result $A_c \times S$ in the last step estimates the area under c which satisfies the criteria.[1] When c is a density upper bound, the result estimates an upper bound on the probability of satisfying the criteria. Similarly, when c is a density lower bound, the result estimates a lower probability bound.

An example of using this routine is as follows: let x range from 0 to 10 (so $S = 10$), let the criterion be $q(x) = x^3/3 - 4x^2 + 15x \geq 18$, and let the density upper bound, $c(x)$, be $0.02(10 - |x - 5|)$. The modified density bound is

$$c^*(x) = \begin{cases} c(x) & \text{if } q(x) \geq 18 \\ 0 & \text{otherwise} \end{cases}$$

Let the sample-mean routine pick a sample of 10 independent random values of x.[2] Then the resulting sample is as shown in Table 6.1 and Figure 6.2.

An estimate of the area under the upper density bound is 1.5, which is derived by multiplying the size S, by 0.15, the average of the sampled c values. The estimated standard deviation on this estimate is 0.1, which is derived by multiplying S, by 0.01, the sample standard deviation for the average of c. The actual area equals the estimate of 1.5.

To get an estimate of the upper bound on $\Pr(q(x) \geq 18)$ and a guess of the standard deviation on this estimate, multiply the corresponding summary figures for c^* by S. The result is $\Pr(q(x) \geq 18) \leq 0.76$ with a standard error of about 0.26. The actual upper bound is 0.56, which is less than one standard error from the estimate.

6.3 A Larger Example

For a larger example, I ran HMC on a 58 parameter cardiovascular model (Appendix B) that includes Equation 4.1. Using data on mitral stenosis patients (see

[1]HMC, the resulting algorithm, is related to conditional Monte Carlo [19, Ch. 6] [18, p. 12]. HMC may be a type of the rejection technique described on page 12 of Halton [18], but the reference's description is unclear.

[2]To simplify the example, the "independent random" values "happen" to be the integers between 0 and 9.

Variable	*BP*	*CO*	*HR*	*LAP*	*LVEDP*	*PAP*
Lower Bound on mean	80.43	5.15	75.655	19.945	7.19	26.91
Upper Bound on mean	94.91	6.25	90.145	26.055	10.81	37.09
Standard Deviation	14.48	1.104	14.49	6.110	3.621	10.18

Table 6.2: Input for Larger Example

Appendix B.4), an upper density bound for the six input variables was constructed. The bound has the shape of a joint Gaussian density in which the means were allowed to take on any possible value between the sample mean minus half a sample standard deviation and the sample mean plus half a sample standard deviation (see Table 6.2).[3] For example, in this bound, *PAP*'s standard deviation is 10.18 and its mean is allowed to be anywhere between $32 - 10.18/2 = 26.91$ and 37.09. The bound is formed by using bounds propagation (interval arithmetic) [36, 33] to find the maximum possible value for the density at each sampled point. An example of bounds propagation is that an upper bound on $a + b$ is $ub(a) + ub(b)$.

This density bound was used with HMC to bound the chances of both $PVR \leq 0.3$ and $24.5 \leq SVR$ occurring[4] in the population of mitral stenosis patients represented by the reference for Appendix B.4. The variable SVR is the systemic vascular resistance (general body resistance to blood flow). HMC found that the upper bound on the probability of satisfying this criteria is less than 0.125 (with a sample standard deviation of 0.020 on the figure). These results are consistent with the population studied: none of the ten patients in the reference satisfied this PVR and SVR criterion.

Unfortunately, getting these results took a sample of over 1,800,000 points (over 130 minutes on a Symbolics 3600 Lisp machine). Two major problems are that a compiled version of the model equations and constraints takes about a hundredth of a second to evaluate for each sample member, and that only 40 (!!!) of these 1.8 million$^+$ points were both consistent with the model and satisfied the PVR and SVR criteria.[5] Evidently, the region(s) of parameter values that are both consistent with the model and satisfy the PVR and SVR criteria are quite small.

The following was done to get this example to run this quickly:

1. A first cut at the space of possible values from which to sample the *BP*, *CO*, *HR*, *LAP*, *LVEDP*, and *PAP* parameters is the space formed by the lower and upper bounds on their possible values (Appendix B.2). To save time, the sampled points are limited to ones that are within plus or minus four sample standard deviations from the possible mean values. Parameter values beyond this range

[3] Given that there are 10 patients or data points (9 degrees of freedom), half a sample standard deviation of the parameter is over three halves a standard deviation of the parameter's sample mean. Assuming either a Gaussian or t distribution for each sample mean, this range will include more than the 80% confidence interval of that sample mean.

[4] See Section 4.1 for a comment about simple thresholds.

[5] It takes about half a second to find an upper bound on the density at a given point in the sample, but this does not add too much time because HMC only needs the upper density bound of the 40 points that are both consistent and satisfy the criteria.

were judged to have negligible density values and so will contribute little to the probability of the criteria being satisfied.[6] As a result, CO was sampled for values between 1.0 and 12.0 (instead of 30.0), BP was sampled between 50.0 and 170.0 (instead of 200.0), HR between 25.0 and 160.0 (instead of 250.0), $LVEDP$ between 2.0 and 30.0 (instead of 40.0), and PAP between 10.0 and 90.0 (instead of 100.0). These additional limitations meant that one needed to sample from less than 1/8 of the space possible input values.

2. About half of the sampled input points can be declared inconsistent just by comparing the $LVEDP$, LAP, and PAP values given by the random number generator to the constraint $LVEDP \leq LAP \leq PAP$. These input points can be rejected without needing to take the one hundredth of a second to evaluate the model equations and examining the rest of the constraints. In the future, one should be able to handle such constraints by sampling so that only points that meet that constraint are generated. However, one must be sure either that the sampling is uniform within the region satisfying the constraint, or that any non-uniformity is compensated for when averaging the sample.

Also in the future, one could further diminish the space of possible values that needs to be sampled by taking advantage of the fact that satisfying the criteria requires that the input parameters take on certain values. In this example, one can use the present limits on the input parameters (Appendix B.2) and the numeric bounding method described in Section 5.3.5 to find that in the sample space where criteria $PVR \leq 0.3$ and $24.5 \leq SVR$ are met, $CO \in [1.2, 8.2]$, $LAP \in [7.5, 60]$, and $PAP \in [10.0, 62.5]$. These bounds would further diminish the space of possible input values that needs to be sampled from by another factor of 2.5.

Things could be worse. In this example, the model equations could be explicitly solved, so they were solved and compiled automatically using a simple symbolic equation solver. If the solver were to be unable to explicitly solve the equations, a numeric technique to solve simultaneous equations (like Newton-Raphson [31, Ch. 9.6]) would be needed. On the type of problem given in this example, when treating all the model equations as needing to be solved simultaneously, solving those equations with my version of Newton-Raphson (in which the derivative formulas are symbolically derived and then compiled) takes about 45 times as long to run as just evaluating the explicitly solved and compiled equations. So it is best to explicitly solve as many equations as possible. Unfortunately, using the model in this example to test the effects of some proposed therapy will involve some simultaneous equation solving.

On the more optimistic side, each member of the sample examined by HMC can be generated and tested independently of the others, so parallel processing can be used to speed up the computations considerably.

[6]For a 10 data point (9 degree of freedom) t distribution, the probability of a variable being outside the mean plus or minus four standard deviations is less than 0.005. For a Gaussian density, the probability is less than 10^{-4}.

6.4 More Testing with the Larger Example

Overall, the amount of computation needed in this last example does not make HMC look too promising for the size of the problem being considered. To further test HMC's promise, this example was rerun while varying three factors: the criteria's severity, the tightness of the upper density bound and a third factor that will be described later. To vary the criteria's severity, the *SVR* minimum threshold was set at different values. To vary the density bound tightness, some input variable means for the density bound were allowed to vary as described in Table 6.2, while other means were restricted to taking just the sample mean value (as given in Table B.2). As the number of means allowed to vary in value increases, the bound gets looser.

6.4.1 Description of Trials and Data Collected

The data from these runs (trials) are shown in Tables 6.3 through 6.7. Each table shows the result of using a particular set of criteria with various density bounds. For each set of criteria, all the density bounds are run with the same sample of sets of input values. The density bounds are labeled from 0 to 6, with the label giving the number of input variable means that are allowed to vary. With label i, the first i input variable means (using alphabetic order of names) are allowed to vary.[7] For example, with density bound 2, the first two input variable means (*BP* and *CO*) are allowed to take on any value within the bounds given in Table 6.2, while the last four input variable means (*HR* through *PAP*) are restricted to being point values equal to the sample means given in Table B.2.

Each table is arranged in eight columns. The leftmost column is labeled "Size". The other columns are labeled with the type of density bound being considered. Each row above the double line has a set of two figures for each column. For the "Size" column, the number on the top in each row gives the size of the sample for that row[8]. The number in parentheses on the bottom in each row gives the number of sets of values in the sample that have a non-zero value for the modified curve c^* (the number of sets of values that satisfy the criteria). The other columns also have two figures in each row. In each, the number[9] on top gives an estimate on an upper bound on the probability of satisfying the criteria for the density bound of concern for the column. The number in parentheses (and maybe in combination with square and/or curly brackets) on the bottom estimates the standard deviation of the previous number. As an example, in Table 6.3, when the sample size was about 2,100,000 points, 44 of them had non-zero values, and density bound 2 yielded an estimate of 0.01729 on an upper bound of satisfying the criteria of "$PVR \leq 0.3$ and $24.5 \leq SVR$" with an estimated standard deviation of 0.00624 on that estimate of the upper bound. The

[7]Actually, for the 0 case, where no means are allowed to vary, one does not need the slow bounding mechanism to generate the density bound. But for the purposes of testing HMC, the mechanism was used.

[8]The letter "K" means that the figure is in thousands. The letter "M" means that the figure is in millions.

[9]The notation $X.XXeQ$ stands for the value $X.XX \cdot 10^Q$.

three rows below the double line are explained later.

Each table shows one run with the criteria indicated. Data were recorded every time the sample size doubled and also when the run was stopped. The tables show every recording that had at least one non-zero valued point in the sample. Recordings that have only zero valued points give an estimate of 0 for the upper bound with an estimated standard deviation of 0, an obviously inaccurate result.

Given this data, a question to ask is how large did the sample need to have been to produce a "result". The answer of course depends on the result desired. For this experiment, desired result is one of the following statements: "HMC is fairly confident that the upper bound on the probability of satisfying the criteria is at most X" or the same statement with the phrase "at most" replaced with "close to", or the same statement with the phrase "at most" replaced with "more than", where X some threshold. The first two statements tell one that HMC is confident that the probability of satisfying the criteria is at most about X. The last statement tells one that with the given input density bound and criteria, HMC cannot say much about the probability of satisfying the criteria. The value of the threshold X is the third factor that is varied in the experiment.

To be more specific, the requirements used in this experiment to declare that a run at sample size n is large enough to conclude a result is the following: At $X = 0.1$, there are two sets of conditions that must be both satisfied. The first set is independent of the threshold, and is:

1. There are at least several (3) non-zero values in the sample.
2. At sample size n, the estimate of the probability bound is within two standard deviations[10] of the estimate at size $n/2$.
3. There are more non-zero values at sample size n then at size $n/2$.

In all runs, the vast majority of the sample points are zero (did not satisfy the criteria or were inconsistent with the model). The first requirement in this set just insures that a few non-zero points are in the sample. As mentioned before, if all the points in the sample have a zero value, HMC will produce an estimate of 0 for the upper bound on the probability of satisfying the criteria with an estimated standard deviation of 0 on that bound, an obviously inaccurate result. The last two requirements in this set are to try to prevent stopping when only a few non-zero points exist in the sample and all of these non-zero points are orders of magnitude lower than the average for non-zero points. If HMC were to stop at this time, not only would the estimate of the probability upper bound be likely to be orders of magnitude too low, but so would the estimate of the standard deviation of that upper bound estimate.[11] Such a low standard deviation estimate would not warn users that the bound estimate may be

[10]Using the standard deviation figure from sample size $n/2$.

[11]The sample standard deviation on the upper bound estimate is determined by the standard deviation between the points chosen for the sample. If all of these points are orders of magnitude lower than the average, then the sample standard deviation will also be orders of magnitude lower than the actual standard deviation.

very inaccurate. This result would be similar to if HMC just found zero points in the sample so far. Examples of this bound estimate and standard deviation both being orders of magnitude too low appear in columns 3 and 4 of the 1024 through 8196 rows in Table 6.6.

The second set of conditions at $X = 0.1$ that must be satisfied in order to declare that a run at sample size n is large enough for a result is that one of the following conditions is true at sample size n:

1. The upper bounds estimate plus two sample standard deviations is at most equal to the threshold of 0.1. If this occurs, one can be fairly confident that the upper bound on the probability of satisfying the criteria is at most 0.1.

2. The upper bounds estimate plus two sample standard deviations is ≤ 0.13, and the upper bounds estimate minus two sample standard deviations is ≥ 0.07. If this occurs, one can be fairly confident that an upper bound on the probability of satisfying the criteria is about 0.1.

3. The upper bounds estimate minus two sample standard deviations is greater than the threshold of 0.1. If this occurs, one can be fairly confident that with this input density bound, HMC cannot answer the question "Is the probability of satisfying the criteria ≤ 0.1?" one way or another.

The use of a two sample standard deviation buffer between the estimate and the threshold of interest is to increase the confidence that the actual bound has the same relation to a threshold as the estimate. The Chebyshev inequality [10, p. 227] indicates that independent of the estimate's probability distribution, the chances of the upper bound estimate being more than two standard deviations away (in either direction) from the actual upper bound (given the input density bound used) is at most 0.25. So assuming that the sample standard deviation is close to the actual standard deviation, the above condition set produces statements with a 75% level of confidence.

This level of confidence increases if one assumes that the estimate's distribution is approximately Gaussian: then the chances of the bound estimate being more than two standard deviations away in a particular direction from the actual bound is less than 0.023 [10, p. 689]. Then, again assuming that the sample standard deviation is close to the actual standard deviation, the above condition set produces statements with a greater than 95% level of confidence. The rationale for the Gaussian assumption is that in this experiment, the estimates are the average of a sample with thousands to millions of independent, identically distributed points. Since an average is a sum, the estimates in this experiment are large sums. From the Central Limit Theorem [10, Ch. 5], large sums of independent, identically distributed random numbers tend to approach having a Gaussian distribution.

For the threshold $X = 0.2$, I used the sets of conditions above with 0.2 substituting for the old threshold of 0.1, 0.25 substituting for 0.13, and 0.15 substituting for 0.07.

The result of applying these requirements (to make a conclusion) on the recorded data are shown in Tables 6.3 through 6.7. The entries with the bold face upper

Size	Density Bound (Number of Varying Means)						
	0	1	2	3	4	5	6
262K (2)	9.69e-13 (9.6e-13)	6.35e-11 (6.3e-11)	7.49e-10 (7.3e-10)	1.155e-6 (9.76e-7)	3.224e-3 (3.09e-3)	.03861 (.0279)	.04673 (.033)
524K (14)	7.2e-4 (4.9e-4)	7.724e-3 (4.14e-3)	.04818 (.0225)	.07512 (.029)	0.1021 (.0334)	0.1478 (.0412)	0.1539 (.0422)
1.0M (28)	**5.143e-4** [2.74e-4}	**4.718e-3** [2.13e-3}	**.03438** [.0125}	**.070** (.0197}	**.09671** (.023}	**0.1338** (.0277}	0.1475 (.0292)
2.1M (44)	2.579e-4 (1.37e-4)	2.365e-3 (1.06e-3)	.01729 (6.24e-3)	**.04392** [.0111)	**.06006** [.0129)	.08175 (.0153)	**0.1158** (.0182}
3.4M (72)	1.689e-4 (8.57e-5)	1.604e-3 (6.68e-4)	.01203 (3.94e-3)	.05016 (9.33e-3)	.07269 (.0113)	.08992 (.0127)	0.1192 (.0146)
[0.1	Below 0.1 Threshold					?	
0.2}	Below 0.2 Threshold						
X	Relationship to Threshold X						

Table 6.3: Criteria: $PVR \leq 0.3$ and $24.5 \leq SVR$

bound estimates are the first entries (entries with the smallest sample size) in their columns to meet the requirements for the $X = 0.1$ and/or $X = 0.2$ thresholds. If the sample standard deviation of the entry has a "[" in place of a left parenthesis, then the $X = 0.1$ requirements were first met at the entry. If the standard deviation has "}" in place of a right parenthesis, then the $X = 0.2$ requirements were first met at the entry. In each table, the two rows (plus header row) below the double line indicate the conclusions made by applying these requirements. The row with X at "[0.1" gives the conclusions of using the $X = 0.1$ threshold.[12] The row with X at "0.2}" gives the conclusions of using the $X = 0.2$ threshold. The conclusion of "?" means that the run did not have data meeting the requirements given for drawing a conclusion.

For an example of applying these requirements, look at Table 6.7. For the density bound where 2 input parameter means can vary in value, the $X = 0.1$ requirements to make a conclusion are first met at the sample size of sixty six thousand points. The conclusion made is that the upper bound on the probability of meeting the criteria ($PVR \leq 0.3$ and $12.5 \leq SVR$) that HMC can find is over 0.1. The run for this density bound and criteria never progresses to a sample size where the $X = 0.2$ requirements to draw a conclusion are met.

6.4.2 Analysis of Trials

What do the above trials (runs) tell about HMC? One thing is that since HMC is a probabilistic algorithm, strange looking results will appear with enough runs. For example, look at Table 6.7 in the column where 1 input parameter mean can vary in

[12]The symbol "[" in the label is to remind readers that this symbol is used to mark when the data first meets the $X = 0.1$ requirements.

Size	Density Bound (Number of Varying Means)						
	0	1	2	3	4	5	6
8192 (1)	2.28e-10 (2.3e-10)	2.161e-8 (2.16e-8)	4.137e-7 (4.14e-7)	4.103e-3 (4.1e-3)	0.7477 (0.748)	0.7477 (0.748)	0.7477 (0.748)
16K (1)	1.14e-10 (1.1e-10)	1.081e-8 (1.08e-8)	2.069e-7 (2.07e-7)	2.052e-3 (2.05e-3)	0.3739 (0.374)	0.3739 (0.374)	0.3739 (0.374)
33K (4)	2.796e-5 (2.08e-5)	5.706e-3 (5.07e-3)	0.1219 (0.11)	0.3749 (0.264)	0.5608 (0.324)	0.5608 (0.324)	0.5667 (0.324)
66K (9)	1.891e-4 (1.25e-4)	**6.986e-3** [3.99e-3}	.08731 (.0584)	0.3798 (0.187)	**0.6807** [0.249)	**0.7477** [0.264}	**0.7507** [0.264}
131K (16)	**1.018e-4** [6.32e-5}	3.624e-3 (2.0e-3)	**.04424** (.0292}	0.2203 (.0983)	0.4194 (0.136)	0.5774 (0.163)	0.6704 (0.175)
262K (35)	3.045e-4 (2.13e-4)	3.272e-3 (1.37e-3)	**.03577** [.0167)	**0.2621** [.0758)	**0.4714** (0.102}	0.6661 (0.124)	0.7558 (0.132)
524K (73)	2.245e-4 (1.12e-4)	3.469e-3 (1.11e-3)	.04527 (.016)	0.25 (.0523)	0.4139 (.0682)	0.5951 (.0825)	0.7477 (.093)
1.0M (153)	7.769e-4 (5.43e-4)	.01524 (7.57e-3)	.05449 (.0145)	0.2459 (.0367)	0.3899 (.0469)	0.5542 (.0562)	0.7722 (.0666)
2.1M (306)	1.076e-3 (3.54e-4)	.01666 (5.09e-3)	.07493 (.013)	**0.2802** (.0278}	0.4449 (.0354)	0.5829 (.0409)	0.7647 (.047)
4.2M (590)	9.618e-4 (2.18e-4)	.01744 (3.55e-3)	.07621 (9.38e-3)	0.2828 (.0198)	0.4471 (.0251)	0.5713 (.0286)	0.7454 (.0328)
8.4M (1180)	1.019e-3 (1.66e-4)	.01471 (2.14e-3)	.0687 (6.28e-3)	0.2728 (.0137)	0.4453 (.0177)	0.5848 (.0205)	0.7515 (.0233)
17M (2369)	8.476e-4 (1.02e-4)	.01255 (1.33e-3)	.06142 (4.14e-3)	0.2726 (9.66e-3)	0.4503 (.0126)	0.5894 (.0145)	0.7505 (.0165)
24M (3216)	7.433e-4 (7.82e-5)	.0119 (1.1e-3)	.05804 (3.4e-3)	0.2568 (7.91e-3)	0.4265 (.0104)	0.5668 (.012)	0.7249 (.0137)

[0.1	Below 0.1 Threshold	Above 0.1 Threshold
0.2}	Below 0.2 Threshold	Above 0.2 Threshold
X	Relationship to Threshold X	

Table 6.4: Criteria: $PVR \leq 0.3$ and $22.5 \leq SVR$

Size	Density Bound (Number of Varying Means)						
	0	1	2	3	4	5	6
4096	1.51e-13	2.58e-11	3.2e-10	1.518e-6	1.492e-4	.03262	1.495
(1)	(1.5e-13)	(2.6e-11)	(3.2e-10)	(1.52e-6)	(1.49e-4)	(.0326)	(1.5)
8192	1.15e-10	7.208e-9	1.219e-7	1.289e-3	1.495	1.512	2.243
(3)	(8.3e-11)	(6.45e-9)	(1.02e-7)	(1.19e-3)	(1.06)	(1.06)	(1.29)
16K	2.622e-3	.03667	0.3169	0.3749	1.442	1.504	**1.869**
(5)	(2.62e-3)	(.0367)	(0.317)	(0.374)	(0.722)	(0.748)	[0.836)
33K	**1.346e-3**	**.01946**	0.1817	0.3747	**1.069**	**1.5**	**1.874**
(11)	[1.31e-3}	[.0184}	(0.16)	(0.264)	[0.432}	[0.529}	(0.591}
66K	6.792e-4	9.82e-3	.09146	0.2793	1.001	1.647	2.062
(28)	(6.56e-4)	(9.18e-3)	(.0801)	(0.146)	(0.289)	(0.388)	(0.438)
131K	3.825e-4	8.555e-3	**.08694**	0.3202	0.7547	1.225	1.639
(45)	(3.29e-4)	(5.51e-3)	(.0538}	(0.115)	(0.179)	(0.236)	(0.276)
262K	2.415e-3	.03092	0.1078	**0.4389**	0.8006	1.174	1.499
(78)	(1.56e-3)	(.0186)	(.0449)	[.0951}	(0.133)	(0.164)	(0.187)
524K	1.814e-3	.03007	0.1008	0.4578	0.7894	1.166	1.518
(153)	(8.88e-4)	(.015)	(.0297)	(.0691)	(.0935)	(0.116)	(0.133)
1.0M	2.43e-3	.0338	0.1111	0.4576	0.7472	1.094	1.439
(288)	(9.96e-4)	(.0108)	(.0218)	(.0498)	(.0647)	(.0791)	(.0912)
2.1M	2.076e-3	.02864	0.106	0.451	0.755	1.047	1.356
(542)	(6.56e-4)	(6.81e-3)	(.0152)	(.0352)	(.046)	(.0547)	(.0626)

[0.1	Below 0.1 Threshold	?	Above 0.1 Threshold
0.2}	Below 0.2 Threshold		Above 0.2 Threshold
X	Relationship to Threshold X		

Table 6.5: Criteria: $PVR \leq 0.3$ and $20.5 \leq SVR$

value. At a sample size of 2048, the bound estimate of .07109 is low enough to be more than two sample standard deviations (2×0.0591) below the $X = 0.2$ threshold. As the sample size increases eight-fold to about sixteen thousand, the bound estimate increases to above the 0.2 threshold (but by less than one sample standard deviation). But another four-fold increase to a sample size of about sixty six thousand brings the bound estimate back down below the threshold (but also within one sample standard deviation). The sample at a size of 2048 satisfied the requirements to draw the conclusion that the particular density bound gave a less than 0.2 chance of satisfying the criteria. But as the sample size increased, one wonders if that conclusion was drawn prematurely and in fact if the correct conclusion was drawn. Probabilistic algorithms always have a chance of making the wrong conclusions.

Besides this general warning, these trials also give an indication of when HMC tends to take less time to draw a conclusion and when it tends to be able to draw more useful conclusions (A conclusion that a high upper bound exists on a value is not very useful.). The effects of varying the criteria's strictness, the input upper density bound's tightness and the probability threshold are as follows:

Criteria Strictness

In the trials, the criteria become more stringent as the minimum acceptable value for *SVR* increases. As the criteria get quite stringent (Table 6.3), few of the points sampled both meet the criteria and are consistent with the model. These points are the only ones given a non-zero value by HMC, so one will probably need a large sample before one has enough non-zero valued points to draw a conclusion. When only a few non-zero points exist in the sample, there is a good chance that all of these points will be an order of magnitude or more less than the average non-zero point. As previously mentioned, the requirements to stop (and draw a conclusion) are designed to avoid stopping when this particular event occurs and instead go on to collect a larger sample. This design helps prevent the bound estimate and sample standard deviation from being grossly underestimated. The benefit of having stringent criteria is that one tends to be able to draw the conclusion that the upper bound on the probability of the criteria being satisfied is low. So as the criteria become more stringent, one takes a longer time to make a conclusion, but is more likely to be able to conclude that certain events are not likely to occur.

On the other hand, as the criteria become quite loose (Table 6.7), a much larger percentage of the sampled points both meet the criteria and are consistent with the model. But at the same time, the upper bound that HMC can find on the probability of the criteria being satisfied will tend to increase. So as the criteria loosen, one takes less time to reach a conclusion, but is more likely to conclude that HMC is not telling very much.

Criteria in between the extremes tend to have a corresponding interpolation of the properties above. An exception is when the upper probability bound found by HMC is near the threshold. In this situation, HMC needs a large sample to decide which side of the threshold the probability bound falls on (or that the bound is indeed close to the threshold). An example of this exception occurs with $X = 0.1$ threshold and

Size	Density Bound (Number of Varying Means)						
	0	1	2	3	4	5	6
1024 (1)	2.62e-11 (2.6e-11)	1.175e-9 (1.17e-9)	8.609e-9 (8.61e-9)	6.38e-6 (6.38e-6)	3.115e-4 (3.11e-4)	.0658 (.0658)	5.982 (5.98)
2048 (1)	1.31e-11 (1.3e-11)	5.87e-10 (5.9e-10)	4.304e-9 (4.3e-9)	3.19e-6 (3.19e-6)	1.557e-4 (1.56e-4)	.0329 (.0329)	2.991 (2.99)
4096 (1)	6.54e-12 (6.5e-12)	2.94e-10 (2.9e-10)	2.152e-9 (2.15e-9)	1.595e-6 (1.59e-6)	7.787e-5 (7.79e-5)	.01645 (.0164)	1.495 (1.5)
8192 (7)	3.06e-10 (3.0e-10)	1.019e-8 (1.0e-8)	1.04e-7 (1.03e-7)	2.244e-4 (2.23e-4)	0.1463 (0.146)	0.8196 (0.749)	2.991 (1.5)
16K (13)	1.49e-8 (1.4e-8)	9.367e-6 (9.06e-6)	3.763e-4 (3.67e-4)	0.4692 (0.384)	1.195 (0.652)	**1.732** [0.774}	**3.366** [1.12}
33K (25)	4.37e-3 (4.22e-3)	.07587 (.0602)	0.3811 (0.264)	**0.9933** [0.42)	**1.436** [0.503}	2.228 (0.63)	3.366 (0.793)
66K (47)	**.01106** [5.18e-3}	**.07826** (.0368}	0.3927 (0.173)	**1.171** (0.325}	1.673 (0.388)	2.22 (0.445)	3.274 (0.553)
131K (97)	9.491e-3 (3.8e-3)	.06347 (.0239)	0.3025 (0.108)	1.065 (0.213)	1.98 (0.298)	2.564 (0.342)	3.428 (0.399)
262K (198)	8.118e-3 (2.84e-3)	**.05449** [.0205)	0.2049 (.0606)	1.033 (0.149)	1.977 (0.21)	2.669 (0.246)	3.609 (0.289)
524K (396)	6.315e-3 (1.99e-3)	.04927 (.0163)	0.1796 (.0397)	0.9545 (0.101)	1.898 (0.145)	2.728 (0.176)	3.734 (0.208)
1.0M (789)	6.494e-3 (1.38e-3)	.06362 (.0129)	**0.2444** [.0329)	1.046 (.0754)	1.893 (0.103)	2.682 (0.123)	3.662 (0.145)
2.1M (1574)	9.136e-3 (1.42e-3)	.07399 (.0101)	**0.2683** (.0241}	1.126 (.0553)	1.977 (.0745)	2.736 (.0883)	3.703 (0.103)
2.7M (1962)	8.848e-3 (1.34e-3)	.0701 (8.72e-3)	0.254 (.0207)	1.075 (.0475)	1.887 (.064)	2.632 (.0761)	3.566 (.0892)

[0.1	Below 0.1 Threshold	Above 0.1 Threshold
0.2}	Below 0.2 Threshold	Above 0.2 Threshold
X	Relationship to Threshold X	

Table 6.6: Criteria: $PVR \leq 0.3$ and $16.5 \leq SVR$

Size	Density Bound (Number of Varying Means)						
	0	1	2	3	4	5	6
256 (1)	2.231e-3 (2.23e-3)	0.1053 (0.105)	1.103 (1.1)	23.93 (23.9)	23.93 (23.9)	23.93 (23.9)	23.93 (23.9)
512 (1)	1.115e-3 (1.12e-3)	.05264 (.0526)	0.5513 (0.551)	11.96 (12.)	11.96 (12.)	11.96 (12.)	11.96 (12.)
1024 (4)	.01762 (.0171)	0.1415 (0.118)	0.7672 (0.563)	17.94 (10.4)	17.94 (10.4)	17.94 (10.4)	17.94 (10.4)
2048 (6)	**8.826e-3** [8.54e-3}	**.07109** (.0591}	0.3875 (0.281)	11.15 (5.54)	**14.95** [6.68}	**14.95** [6.68}	**14.95** [6.68}
4096 (9)	.01207 (8.32e-3)	0.1343 (.0972)	0.906 (0.693)	**7.881** [3.25}	10.47 (3.95)	10.47 (3.95)	10.47 (3.95)
8192 (15)	6.034e-3 (4.16e-3)	.06714 (.0486)	0.453 (0.347)	3.945 (1.63)	6.003 (2.11)	6.729 (2.24)	7.511 (2.36)
16K (26)	.01886 (.0155)	0.2502 (0.203)	0.9834 (0.556)	3.015 (1.01)	4.442 (1.27)	5.288 (1.4)	6.751 (1.59)
33K (40)	.02515 (.0167)	0.1865 (0.114)	0.7043 (0.317)	2.267 (0.629)	3.342 (0.782)	3.954 (0.857)	5.279 (0.989)
66K (104)	.01588 (8.52e-3)	0.1783 (.0868)	**0.5809** [0.208)	2.539 (0.47)	3.798 (0.584)	5.002 (0.681)	7.358 (0.823)
[0.1	Below 0.1	?	Above 0.1 Threshold				
0.2}	Below 0.2 Threshold		?	Above 0.2 Threshold			
X	Relationship to Threshold X						

Table 6.7: Criteria: $PVR \leq 0.3$ and $12.5 \leq SVR$

the density bound where 2 means can vary in value. As the minimum acceptable SVR value drops from 24.5 to 12.5, the sample size needed to make a decision drops from about one million to sixty six thousand. But in-between, at a minimum acceptable SVR value of 20.5 (Table 6.5), when the probability bound estimate is about 0.1, even a sample size of over two million is not large enough to make a conclusion as to whether the probability upper bound is over, under or quite close to 0.1.

When the criteria are quite stringent, the following two possible techniques exist to speed up the decision-making process:

1. As mentioned above, the fraction of the sample that has non-zero values (call the fraction fnz) is quite small. If the fraction is small enough, a simple estimate of the upper bound on the probability of satisfying the criteria is $S \cdot c_{max} \cdot fnz$, where S is the size of the input value space and c_{max} is the maximum value that the input density bound can have.

2. One may try running HMC with a loosened version of the criteria. One will be able to make conclusions faster with the loosened version. If the loosened version is unlikely to occur, then the original criteria will also be unlikely to occur.

Input Upper Density Bound Tightness

The second factor varied is the tightness of the input upper density bound. In the trials, the density bound gets tighter as more input parameter means are fixed at the sample means given in Table B.2 and not allowed to vary within the bounds given in Table 6.2. In Tables 6.3 through 6.7, this occurs as the numbers at the tops of the columns decrease.

When the input density bound is very loose, HMC will find a high upper bound on the probability of satisfying the criteria, so the conclusion that can be drawn is that HMC is not of much use in this situation. To compensate somewhat, this conclusion can be drawn quickly because probability upper bound estimates tend to be considerably more than most thresholds that one would consider.

At the other end, when a input density bound is tight, HMC will find a quite low upper bound on the probability of satisfying the criteria. This lets a user draw the conclusion that the criteria are not likely to be met. Also, because the upper bound is often much less than the acceptable threshold, this conclusion can often be quickly quickly drawn. The main shortcoming in this situation is that finding a tight input density bound that models the situation is not easy. Also, if the input bound is too tight, the fraction of the sample that has non-zero values may get quite small. As mentioned above, this would increase the sample size needed to draw a conclusion.

In between these extremes, the upper probability bound found by HMC decreases as the input density bound is more tight. The amount of time needed to draw a conclusion increases as the density bound tightness or looseness causes HMC to find an upper probability bound that is closer to the chosen threshold. As mentioned before, when the probability bound is close to the threshold, HMC needs a large sample to decide which side of the threshold the probability bound falls on (or that the bound is indeed close to the threshold).

All these effects of varying the density bound tightness occur in Table 6.5.

Threshold on Acceptable Upper Probability Bound

The third (and last) factor varied is the threshold X on what is an acceptably low upper bound on the probability of satisfying the criteria. X is set at a value such that when the probability upper bound is below X, a user is willing to say that the criteria are not likely to be satisfied. In the trials, two values of X are used: 0.1 and 0.2.

When X is lower, one is more likely to reject a probability upper bound for being too high and just conclude that HMC is not useful for the problem. Conversely, one is less likely to accept a probability upper bound as being low enough to indicate that the criteria are unlikely to be satisfied. However, with a lower X, one is more sure that the actual probability of satisfying the criteria is indeed low when one accepts the upper bound as being low enough. One will reach a conclusion faster as the threshold X is farther away from HMC's estimates of the probability upper bound.

All the opposite things occur when X is higher.

Speed-Up Suggestions

Some possible changes to speed up HMC have already been suggested at the end of Section 6.3.

Another possible change deals with the problem of needing a large sample when the upper bound on the probability is near the threshold X for what is an acceptably low upper bound on the probability. This change involves having two thresholds (X_l and X_h) in place of the original threshold X, where $X_l < X < X_h$.

1. When HMC's estimate on the probability upper bound is clearly above X_l (≥ 2 sample standard deviations above), stop and conclude that the upper bound that HMC is finding will be too high to be useful.

2. When HMC's estimate on the probability upper bound is clearly below X_h, stop and conclude that the probability of satisfying the criteria is low (below X_h).

The farther apart X_l and X_h are, the more likely that when the upper probability bound estimate is close to one threshold, the estimate will be far (in terms of sample standard deviations) from the other threshold (the one that is likely to cause HMC to stop). This change to two thresholds will make HMC stop faster, but the conclusions drawn are not quite the same as when only one threshold exists.

Chapter 7

GLO. A Moment Approximation Method.

7.1 Introduction

This chapter describes a method named GLO for analyzing continuous systems which can be described by a static model. Given a set of equations modeling a system's steady-state conditions, GLO will use input parameter moments (means, variances, etc.) to approximate the output and intermediate parameter moments and distributions; it will also find and graphically display the relative contributions of each operation's operands to that operation's result by graphing the result's and operands' moments. One can use GLO to quickly approximate a model's reactions to a given set of inputs, to identify the major influences on these reactions, and to show where one might simplify the model.

GLO works by manipulating moments (means, variances, etc.) of various parameters. Compared to other current techniques that also manipulate moments (means, variances, etc.), GLO makes fewer approximations and does not require hard-to-get higher order joint moments, like $E[(x - E[x]) \cdot (y - E[y])^2]$.

GLO assumes that all variables are approximately Gaussian (normal) and/or LOg-normal [3], which gives GLO its name. Products and exponentiations often result in approximately lognormal densities; the same is true for linear combinations and Gaussian densities. Both types of densities are bell-shaped with a peak near the mean, so lognormals and Gaussians can resemble each other as well as many commonly found distributions of values. Lognormals and Gaussians can be characterized by giving just their means ("center" of the possible values), variances ("spread" of the possible values), and covariances (how one variable's value relates to another variable's value).[1] A simple set of equations relate the mean, variance, and covariances of a linear combination with the corresponding information about the combination's operands; the same is true for products and exponentiations of lognormal operands.

[1]The covariances can be obtained from the corresponding variances and correlation coefficients. Like covariances, correlations measure relationships between pairs of variables. However, correlations are limited to values between -1 and 1, while covariances have no numeric limits. So correlations may be easier for experts to estimate.

These two sets of equations show how the operands combine to produce the result of a linear combination or product.

The next section of this chapter describes some current moment manipulation methods. It is followed by section that gives an example of using GLO and gives a discussion on graphing the effects of operands on an operation's result. Afterwards is a section is on GLO's mathematical basis and assumptions, and then a section on a test of those assumptions. The last section is a short summary.

7.2 Current Moment Manipulation Methods

Björke [5] and Pearl [29, Sec. 7.2] give methods that, like GLO, use formulas to find moments such as means and standard deviations. The methods are more limited in that they assume independent inputs and linearized equations. In addition, Pearl assumes Gaussian densities. This assumption will be quite inappropriate for densities that turn out to be skew (asymmetric). It would have given one positive variable a 15% chance of being negative in the empirical accuracy test of GLO to be described in a later section. However, Björke has a way of using numeric bounds to determine the output density shape that might make an interesting addition to GLO.

Shachter and his associates [43, 42] describe a similar method that assumes linearized equations and Gaussian densities, but does *not* assume independence between variables. The method uses variable transformations to make the random variables more Gaussian and the variables' relationships more linear. These transformations increase the accuracy of the results, but may obscure the relationships among the original variables.

Another similar method [16, Ch. 6,7] is more general than Björke's and the others. However, this particular method needs hard-to-get and unintuitive higher order joint moments, like $E[(x-E[x])\cdot(y-E[y])^2]$, in order to accurately determine the density shape. GLO only needs correlations. One could always assume a Gaussian density, but as mentioned in the description on Pearl's method, this assumption will be quite inappropriate for densities that turn out to be skew. A partial use would be to use the formulas in [16] to find just the operations' means and variances. However, the formulas for products and exponentiations are approximations, while the ones in GLO are exact when given lognormal operands. The approximations are truncated Taylor series expansions [49]. They ignore the higher order derivative terms and become less accurate as these terms become more significant, such as when $|x|$ approaches 0 for $f(x) = x^{-1}$, where $|d^k f(x)/dx^k| = (k!|x|^{-k-1})$.[2] These approximations resemble GLO more as the normalized (by the means) variance and covariance magnitudes decrease. It should be noted however, that as mentioned in Section 1.5, empirical tests comparing this method and GLO have had mixed results.

[2]Mentioned to me by Alvin Drake.

Name	Mean	Std Dev	Correlations:	co	bp	lap
co	5.69	1.104		1.0	-.1161	-.4100
bp	87.67	14.48		-.1161	1.0	.0653
lap	23.0	6.110		-.4100	.0653	1.0

Table 7.1: Input Statistics for the Blood Volume Problem

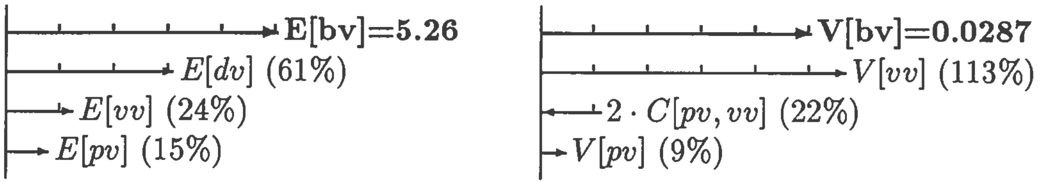


Figure 7.1: Density Parameters for the Sum $bv \leftarrow vv + pv + dv$

7.3 Example & Graphing Discussion

Suppose that one has a simple model to find bv (blood volume) and svr (systemic vascular resistance) from the inputs lap (left atrial pressure), bp (blood pressure), co (cardiac output), and dv (dead volume). Dead volume is constant at 3.2, and the other inputs have a joint Gaussian density with the parameters shown in Table 7.1. The model equations are $vv \leftarrow (svr \cdot 0.025 + 0.9) \cdot co/5.7$, $pv \leftarrow [(lap - 1.316)/59.8]^{1/4.125}$, $bv \leftarrow vv + pv + dv$, and $svr \leftarrow bp \cdot co^{-1}$.

Given this input, GLO finds that bv is approximately Gaussian with a mean of 5.26 liters and a standard deviation (*s.d.*, equals $\sqrt{\text{variance}}$) of 0.166. GLO also finds that svr is approximately lognormal with a mean of 16.0 *mmHg/(l/min)* and an *s.d.* of 4.35. The svr result implies that $\sim$ 7% of the subjects have a less than desirable svr (the minimum desirable value is 10.4).

Besides finding the output values, it is also useful to show the relative strengths of the contributions of the different operands to an operation's result for the model equations' sums and products (the operations with more than one operand). Such a display can show the most significant influences in a model, the insignificant influences to eliminate if one simplifies the model, and the amount needed to alter an operand to significantly affect a result. For sums, the display consists of two graphs. One graph compares the components of the sum's expected (mean or average) value (E) with the sum's E. Another graph does likewise with the sum's variance (V). For products, the description consists of similar graphs for the product's mean (E) and normalized variance (V_n, equals V/E^2). Figures 7.1 & 7.2 give examples of graphing

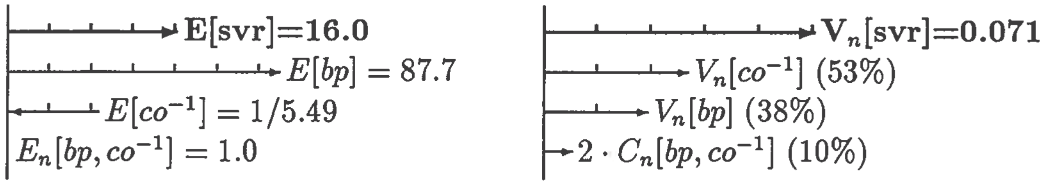


Figure 7.2: Density Parameters for the Product $svr \leftarrow bp \cdot co^{-1}$

a sum and product of random variables, respectively.

The graphs are based on formulas in the *Basis* section of the chapter. Every factor or result is displayed with an arrow whose length is proportional to that factor/result's significance. The results are printed in boldface. The more significant factors are displayed above the less significant ones. Also, right pointing arrows mark factors that increased the result and vice-versa for left pointing ones. The combination of several factors corresponds to the vector addition of those factors' arrows.

Except in the graphs for the expected values of products ($E[svr]$ in the example), the arrow lengths are proportional to the corresponding values' magnitudes and each tick marks another 20% of the result. Each result is viewed as a sum of every operand's corresponding parameter. In the cases involving V or V_n, the sum also includes twice the covariance ($2 \cdot C$) of each pair of statistically dependent operands[3] to compensate for that pair's interaction. For example, in Figure 7.1, $V[bv]$ is a sum of $V[vv]$ (113% of the total), $V[pv]$ (9% of the total), and $2 \cdot C[pv, vv]$ (decreased $V[bv]$ by 22% of the total).

This description of graphing sums also applies to the normalized variance of a product of random variables ($V_n[svr]$ in the example). So in Figure 7.2, $V_n[svr] = V_n[co^{-1}] + V_n[bp] + 2 \cdot C_n[bp, co^{-1}]$, even though $svr = bp \cdot co^{-1}$.

For graphs of the expected values of products ($E[svr]$ in the example), the result is viewed as a product, and the arrow lengths are proportional to the *log* of the corresponding values' magnitudes.[4] The product includes each operand's mean and a term for each pair of statistically dependent operands X and Y. Each term compensates for its pair's interaction and is the mean of the product of the pair normalized by the product of their means ($E_n[X \cdot Y] = E[X \cdot Y]/(E[X] \cdot E[Y])$).[5] For example, in Figure 7.2, svr's mean is a product of bp's mean (which increased the result by a factor of 87.7), co^{-1}'s mean (decreased the result by a factor of 5.49), and the normalized mean of $bp \cdot co^{-1}$ ("increased" $E[svr]$ by a factor of 1).

An alternative graph format exists which explicitly displays the vector addition of the factors' arrows, but otherwise follows the conventions described for the original graphs. This alternative both places more emphasis on how the operands combined to produce a result, and is also possibly more confusing. The convention of displaying more significant factors above less significant ones means that the upper factors give an approximate estimate, while the lower factors adjust that estimate. For example, Figure 7.3 shows the alternative graphs for $E[svr]$ and $V[bv]$. In the $V[bv]$ graph, $V[vv]$ gives the approximate $V[bv]$ value. Adding $2 \cdot C[pv, vv]$ adjusts that approximate value. Adding $V[pv]$ further adjusts the already adjusted value.

Based on these graphs, one can see from Figures 7.1 and 7.3 that bv is basically $dv + vv$, with pv acting as a correction factor. One can also see from Figures 7.2 and 7.3 that the dependence between bp and co^{-1} has little effect on svr over bp and co^{-1}'s effects independently.

[3]If a V_n graph, use twice the covariance normalized by the pairs' means ($2 \cdot C_n[X,Y]$, equals $2 \cdot C[X,Y]/(E[X] \cdot E[Y])$).

[4]The *log* is used so that the vector addition of the arrows corresponds to the multiplication of the corresponding factors. Each tick marks a change of a factor of 2.

[5]The compensation terms are derived from the corresponding covariances and means.

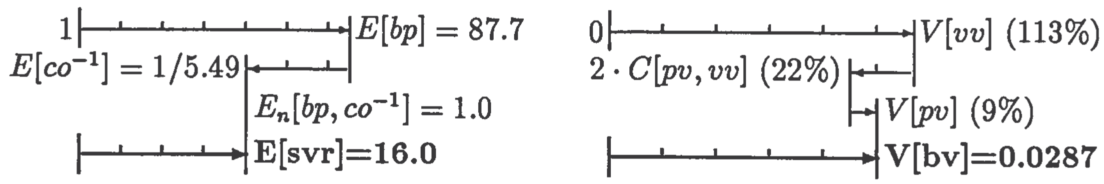


Figure 7.3: Alternative Graphs for $E[svr]$ and $V[bv]$

In all the graphs, each factor to compensate for a pair interaction varies monotonically from the factor's identity element as the corresponding pair's correlation varies from zero.

7.4 Basis

The basis for GLO are rules that determine the distribution type, mean, variance and covariances of a linear combination, product, or exponentiation when given the corresponding information about the operation's operands. This section gives those rules. The derivations are in Appendix C.

For a linear combination of random variables, $X = \sum_i a_i \cdot X_i$, where the a_i's are constants and the X_i's are random variables, the following formulas hold for all densities (mostly from [10, Ch. 4]): $E[X] = \sum_i a_i \cdot E[X_i]$,

$$V[X] = \sum_i a_i^2 \cdot V[X_i] + 2\sum_i \sum_{i<k} a_i \cdot a_k \cdot C[X_i, X_k] \quad \text{and} \quad C[X, Z] = \sum_i a_i \cdot C[X_i, Z],$$

where C is the covariance between 2 variables.

In addition, when the X_i's are approximately Gaussian, then so is X:

1. For $X_k = \sum_i a_{ik} \cdot X_i$, the X_k's are jointly Gaussian when the X_i's are jointly Gaussian [9, Ch. 14].

2. By the Central Limit Theorem, a linear combination of independent variables tends to approach being Gaussian [10, Ch. 5]. So a linear combination of independent and nearly jointly Gaussian variables will be more nearly Gaussian.

3. The above also holds, though more weakly, for nearly independent random variables because random variables can be broken-up into independent and dependent components. For nearly independent random variables, the independent components are the major ones.

Products and exponentiations, such as $Y = \prod_i Y_i^{a_i}$, where the a_i's are constants and the Y_i's are random variables, are dealt with by observing that if $X = \log Y$ and $X_i = \log Y_i$, then $Y = \prod_i Y_i^{a_i}$ corresponds to $X = \sum_i a_i \cdot X_i$. Also, jointly lognormal Y_i's correspond to jointly Gaussian X_i's [3]. Combining these observations with the equations [3, Ch. 2] (they also hold for Y_i and X_i):

$$E[Y] = \exp(E[X] + V[X]/2) \quad \text{and} \quad V[Y] = E^2[Y] \cdot (e^{V[X]} - 1),$$

and the above description of $X = \sum_i a_i \cdot X_i$ gives the following results: Y is usually approximately lognormal when the Y_i's are also. When the Y_i's are jointly lognormal, $Q = \prod Y_i$ may be expressed as $E[Q] = (\prod_i E[Y_i]) \cdot (\prod_i \prod_{i<k} \frac{E[Y_i \cdot Y_k]}{E[Y_i] \cdot E[Y_k]})$, where $E[Y_i \cdot Y_k] = C[Y_i, Y_k] + E[Y_i] \cdot E[Y_k]$, plus

$$\begin{aligned} \frac{V[Q]}{E^2[Q]} + 1 &= [\prod_i (\frac{V[Y_i]}{E^2[Y_i]} + 1)] \cdot [\prod_i \prod_{i<k} (\frac{C[Y_i, Y_k]}{E[Y_i] \cdot E[Y_k]} + 1)^2] \\ \frac{C[Q, Z]}{E[Q] \cdot E[Z]} + 1 &= \prod_i (\frac{C[Y_i, Z]}{E[Y_i] \cdot E[Z]} + 1) \end{aligned}$$

For exponentiations of the form Y^a, the formulas are

$$E[Y^a] = (E[Y])^a \cdot (\frac{V[Y]}{E^2[Y]} + 1)^{[a(a-1)/2]} \quad \text{and} \quad \frac{V[Y^a]}{E^2[Y^a]} + 1 = (\frac{V[Y]}{E^2[Y]} + 1)^{a^2}$$

$$\text{and} \quad \frac{C[Z, Y^a]}{E[Z] \cdot E[Y^a]} + 1 = (\frac{C[Z, Y]}{E[Z] \cdot E[Y]} + 1)^a$$

For independent Y_i's, the $E[Q]$ formula holds for all densities. Simplifications of the complicated variance and covariance formulas for Q are

$$\frac{V[Q]}{E^2[Q]} \sim \sum_i \frac{V[Y_i]}{E^2[Y_i]} + 2 \cdot \sum_i \sum_{i<k} \frac{C[Y_i, Y_k]}{E[Y_i] \cdot E[Y_k]} \quad \text{and} \quad \frac{C[Q, Z]}{E[Q] \cdot E[Z]} \sim \sum_i \frac{C[Y_i, Z]}{E[Y_i] \cdot E[Z]}$$

respectively. They hold true if all the result and summation term magnitudes are $\ll 1$, which will usually be the case in the models to be used. These simplifications were used to help graph $V_n[svr]$ in Figure 7.2. The exact variance and covariance formulas were used as checks. These exact formulas can also be graphed, but the meaning of the graphs are less comprehensible.

These formulas were derived with the help of [3], which also shows that like a Gaussian density, a lognormal density is more or less bell-shaped with a peak near the mean and median. Unlike Gaussian densities, lognormal ones are skewed (asymmetric), with the mean larger than the median, which occurs in turn after the density peak. Also, the density is defined only for positive random variable values, so use a lognormal density to represent the magnitude of a negative random variable.

The product and exponentiation rules need approximately lognormal operands; the same is true for the linear combination rules and Gaussian operands. When finding a sum of products or vice-versa, GLO will need some lognormal and Gaussian densities with the same mean and *s.d.* to resemble each other. To observe the degree of resemblance, examine the ratio $\frac{s.d.}{|\text{mean}|}$. At zero, the two densities are the same. As the ratio increases, the two densities will differ more in the following ways (first two ways derived from [3, Ch. 2]):

1. The lognormal will be more skewed (Gaussians are symmetric).
   $\gamma_1 = E[Y - E[Y]]^3 / V^{(3/2)}[Y]$ is the coefficient of skewness. For Gaussian densities, $\gamma_1 = 0$, but for lognormal densities,

$$\gamma_1 = |\frac{s.d.}{\text{mean}}|^3 + 3 \cdot |\frac{s.d.}{\text{mean}}|.$$

2. The difference between the Gaussian and lognormal medians will increase. The ratio of the Gaussian median to the corresponding lognormal median is

$$\sqrt{(\frac{s.d.}{\text{mean}})^2 + 1}.$$

3. For positive means, a variable with a Gaussian density is more likely to have a negative value (variables with lognormal densities only have positive values).[6] With any density that is non-zero everywhere (like Gaussians), the chance that a variable x takes on a value less than X is $\Pr(x < X) = f(X)$, a monotonically-increasing function of X. If x is Gaussian with zero mean and unit standard deviation ($N(0,1)$), one can look-up $\Pr(x < X)$ in any of the many books which have a table of the cumulative distribution function for this Gaussian density. If a variable x_1 is $N(\text{mean}, s.d.)$ instead, and one wants to find $\Pr(x_1 < X_1)$, one can convert x_1 into x with $x = (x_1 - \text{mean})/(s.d.)$ and X_1 into X with $X = (X_1 - \text{mean})/(s.d.)$. Then $\Pr(x < X) = \Pr(x_1 < X_1)$.

   Now one wants to find $\Pr(x_1 < 0)$, so $X_1 = 0$. This means that $X = -\frac{\text{mean}}{s.d.}$. For a positive mean, $|\text{mean}| = \text{mean}$, so as $\frac{s.d.}{|\text{mean}|}$ increases, so does $\frac{s.d.}{\text{mean}}$. This decreases $\frac{\text{mean}}{s.d.}$, which increases $X = -\frac{\text{mean}}{s.d.}$. This increases $f(X) = \Pr(x_1 < 0)$. So as $\frac{s.d.}{|\text{mean}|}$ increases, so does $\Pr(x_1 < 0)$ for a Gaussian x_1.

7.5 Empirical Accuracy Tests

GLO relies on several approximations. Empirical tests of these assumptions had mixed results. As already mentioned in Section 1.5, some tests came out not so well. One test that did come out well is the following: GLO was run on an earlier version of the cardiovascular model described in Appendix B. GLO used summary data (means, *s.d.*'s, correlations) on 6 parameters from 10 patients (summary data given in Appendix B.4) to estimate the distributions of 13 of their other parameters. The example section shows a subset of this run. These results were compared to the values derived by finding the model's reactions to each patient's set of 6 parameters and then summarizing those reactions over the 10 patients. The 13 parameters' mean estimates were off by $\leq 3\%$ of an *s.d.* All but one estimate were off by $\leq 2\%$ of the actual mean (one was off by 7%). The *s.d.* estimates were off by $\leq 11\%$ of the actual values. When the estimated 10th and 90th percentiles were examined to test the density shapes, it was seen that all but one of the estimates were $\leq 21\%$ of one *s.d.* from where they should be: for the 10%-tile estimate, between the lowest and second lowest data points, and for the 90%-tile, between the ninth and tenth (highest) data points. One percentile estimate was 41% off.

[6] For a variable y with a negative mean, one can give y a lognormal shape by letting $-y$ have a lognormal density.

7.6 Summary

GLO uses Gaussian and lognormal densities to try to improve on current methods to help find and explain the likely steady-state reactions of complicated continuous systems. For explanations, GLO finds and compares the significant influences on the moments of sums and products. GLO has a theoretical advantage over the more standard moment manipulation method which uses truncated Taylor series in that GLO finds the exact moments for products and exponentiations of lognormal variables while the truncated Taylor series method only approximates the moments of products and exponentiations of any variables. However, as mentioned in Section 1.5, neither method seems decisively "better" in empirical tests, and in fact, unfortunately, either can produce large errors at times. Still, when the approximations hold, these methods provide useful relationships between moments.

Chapter 8

Conclusions

This thesis has dealt with two problems that one can encounter in building and using a static model of some system that is in a steady state. The first problem is constructing a static model of an iterative dynamic sub-system which enters a steady-state not by maintaining a single set of parameter values over time, but by steadily iterating a sequence of actions to change parameters at a steady pace. The second problem is using static models when one is uncertain about the distribution of parameter values.

To deal with the first problem, a program called AIS has been implemented and tested. It takes in a description of a sequence of actions (parameter value transformations) and tries to find information associated with the symbolic average rate of change for various parameters. The description is in a form that is easy to give and compute from. The normal ventricle, mitral stenosis and steam engine examples presented all show some of AIS's abilities, as well as some of the present limits of those abilities.

Compared to some other work on automatically analyzing dynamic systems, AIS is limited in that it only analyzes systems which have the invariant property of steadily repeating a fixed sequence of parameter value changes. AIS takes advantage of this invariant to make the necessary computations, and despite this limitation, AIS can still analyze some non-trivial problems. In exchange for having this limitation, AIS does not get lost trying to find the iterated sequence, nor is AIS limited to descriptions in the form of a single set of differential equations. The work on AIS furthers the ability to automatically analyze dynamic systems, a goal of much work in artificial intelligence.

To answer the second problem, two methods have been introduced which use a bound on the joint density of the inputs to bound the likelihood of some possible behavior. The first is called SAB. It analytically bounds the likelihood, narrowing the bound as more iterations are permitted. However, SAB can only use lower-bound information on the joint density. This information is less useful than upper-bound information. The second method is a hit-or-miss version of sample-mean Monte Carlo technique called HMC. It only finds approximate likelihood bounds – but the accuracy increases with the number of samples taken – and it gives a measure of the error magnitude. Also, unlike SAB, HMC can handle both upper and lower density bounds, and can handle larger problems than SAB can.

Despite uncertainty in input density shapes and parameter values, bounds on input densities have not really been utilized to bound the chances of events. SAB and HMC are two bounding methods which have the features of being able to handle events beyond the ones in a pre-enumerated list, estimating the amount of error in an answer, and giving better answers as more iterations or samples are allowed. Unfortunately, SAB and HMC can also be quite slow and/or return ambiguous results.

Finding how parameter averages and variances are related to each other is an aspect of the second problem not handled by SAB, HMC or the standard Monte Carlo techniques. To deal with this aspect, a scheme called GLO to propose relationships between parameter moments (means, variances, etc.) was developed. GLO is exact in more situations than the more standard truncated Taylor Series scheme of estimating moment relationships. However, in practice, neither scheme is clearly better. The two seem to fail, often at different times, making them in some sense complements of each other. They both can fail quite drastically, so one must be quite cautious when trying to use either to predict moments, as opposed to proposing relationships which are then independently checked.

The rest of this chapter is in four parts. The first two present possible future directions for the work. The last two parts give some observations and final comments.

8.1 Future Directions for AIS

The normal ventricle and steam engine examples show that having methods that find tighter bounds on mathematical expressions would help AIS make more conclusions. However if a model has ambiguities to begin with, better bounding of mathematical expressions will not help clear up those ambiguities. For instance, in the mitral stenosis example, the model has cases in which a different value for some constant would have both positive and negative influences on other parameters without saying which influences are stronger.

Other limitations shown in the mitral stenosis example are the inability to infer characteristics about a function from knowledge about its inverse, as well as an inability to express knowledge about a function independent of the specific arguments given to that function. Also, in the example, when a function α is used in an argument to another function β, AIS cannot examine how α being different will affect the rates of accumulation. All of these limitations are correctable with some difficulty.

A possible improvement not suggested by the examples but mentioned in Section 2.3 would be to use QS [37, 39] in place of the present routines to determine curve shapes. This would reduce the need for assumptions about mathematical expressions being smooth. Another possible improvement would be to let AIS use order of magnitude reasoning [32] in its symbolic math manipulations.

At present, AIS handles behaviors where the change in parameter values in a phase is invariant over time: each repetition of a phase changes the parameters by the same constant amounts. A way to describe these behaviors is constant "velocity": the change in an accumulating parameter (like distance) is constant. A way to extend AIS to handle other types of repetitive behavior is to look at the properties that

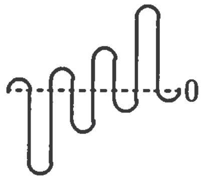


Figure 8.1: Strange "Periodic" Curve

are invariant over time in those other types of behavior while keeping in mind the limitations imposed by the present abilities (or lack of) in automatically symbolically solving possibly nonlinear and simultaneous equations.

Two examples of other repetitive behavior and their invariants have been briefly mentioned in Section 2.1. That section discusses enabling AIS to handle behaviors where either it is the *change* in the amount changed (or higher order of change) that remains constant (instead of the amount changed) or the changes form a converging series. The former addition would let AIS handle constant "acceleration" situations (the change in the change is constant). An example of such a situation is when a pendulum increases its motion's amplitude by a constant amount with each back and forth swing. The latter addition would let AIS handle situations where an iterating system is moving from one equilibrium (steady-state) to another (after a perturbation, the system *converges* to the new equilibrium). Handling these latter situations may also need a detailed look at how an iterating system initially responds to a perturbation. To get these details, one might combine AIS with a more detailed qualitative simulation [6] of the processes involved. AIS can help to disambiguate what happens next when the qualitative simulation is unsure as to what possible action occurs next, and the simulation can show the possible actions just after a perturbation appears (which temporarily violates AIS' assumptions).

Suppose that AIS is generalized so that it can handle behaviors where each repetition of a phase may not change the parameters by the same constant amounts. Then the definition of a periodic parameter needs revising. For example, in a pendulum with a decaying swing, the position of the pendulum bob (mass) is not periodic in the sense that the position does not repeat the exact same sequence of values with each back and forth swing. This the sense used in the present design of AIS. However, the position is periodic in the sense that it swings back and forth about the zero position. So a possible revised definition for AIS is that a parameter is periodic when there exists some constant value α that the parameter reaches during every iteration. This definition will let some pretty strange curves be called periodic. For example, a "sinusoidal" curve where each positive peak is twice the amplitude of the previous positive peak and each negative peak is half the amplitude of the previous negative peak would be considered periodic under this definition: each iteration passes through the value zero (see Figure 8.1). Another possible revised definition is that a parameter is periodic when there is some simple transform (function of time) that converts the function of the parameter's value over time into a cyclic function (like sin). This cyclic function should be periodic in the original sense used for AIS: on each iteration, the transformed parameter repeats the same exact sequence of values. With this revised definition, a decaying oscillation of the form $\exp(-t) \cdot \cos(t)$ will be considered periodic because when it is transformed by being multiplied by $\exp(t)$, it becomes the cyclic function $\cos(t)$. Unfortunately, the reverse is also true: $\exp(t)$ will also be considered periodic because it will be transformed into $\cos(t)$ by being multiplied by the "simple" function $\exp(-t) \cdot \cos(t)$. A good definition of being periodic remains elusive and may in fact be dependent on the purpose for having the label "periodic".

A generalization of AIS would also need a revised definition for accumulating parameters to accommodate parameters whose average value is changing by potentially different amounts on each iteration. An example of such a parameter is one which has the above curve, where the value averaged over an iteration increases with each iteration and the amount of the increase changes from one iteration to the next.

Often, modeling involves deciding whether or not to make certain simplifying assumptions that make it possible/easier to draw certain conclusions when the assumptions are true. To enable AIS to decide on which assumptions to make, one might add to AIS some of the work done in [1, 28, 12].

At present, one cannot input to AIS such a situation where the phase equations are conditional on certain parameter values. Enabling AIS to handle such situations would mean that AIS will need to detect and deal with when different phase equations are active on different iterations of an action sequence.

8.2 Future Directions for SAB and HMC

This section has two parts. The first presents some possible improvements for SAB and the second presents some possible improvements for HMC. Unfortunately, none of the proposed improvements for SAB would enable it to handle an upper bound on an input parameter value density.

Possible future work on SAB itself includes testing how large a problem it can handle and expanding it to more quickly bound a variable's mean, variance, median, 90% confidence interval, etc. One can also explore splitting a region at the selected variable's median value (or some approximation) within the region. This can handle infinite intervals (bisection cannot), which permits an initial region where each variable is within the all-inclusive range of $[-\infty, \infty]$.

There are also two other possible alternatives for splitting regions in step 2(b)ii of Section 5.3.1 (splitting *unsure* regions in SAB's main loop). A description of these two possibilities and why I do not plan to try them is as follows: In the first alternative, instead of bisecting a region α along some variable x,

1. use the SAB bounding mechanism to find the minimum (l) and maximum (h) possible values of x in α that can satisfy the criteria.

2. If no values of x in α are excluded (l and h are the extreme x values in α), use another method to split α.

3. Otherwise, split α into three pieces along x: in first piece, $x \leq l$, in the second piece, $l \leq x \leq h$, and in the third piece, $h \leq x$. By the way the split is done,

the first and third pieces definitely fail the criteria. The second (middle) piece will remain *unsure*.

This alternative quickly marks off the regions that fail the criteria. It does not help with dividing regions to produce better bounds on the probability of being in some region. Unfortunately, as mentioned in the section on SAB's problems, finding regions' probability bounds is SAB's weak point, so this alternative will not help SAB much.

The second alternative is to permit non-rectangular regions. This lets SAB split a region along an edge formed by a part of the criteria, which will speed-up the splitting of *unsure* regions into ones that either always pass or always fail the criteria. One needs to be careful about the resulting regions' shapes, or finding their volume (needed to bound the probability of being in the region) will be difficult. For the same reason as mentioned for the first alternative, this alternative will not help much, so I do not plan to try it.

Sometimes, one might use SAB to test many sets of criteria on a particular model and input. A way to eliminate much of the computation needed in this situation is to reuse the regions made in response to one set of criteria to examine the other sets of criteria.

Possible future work on HMC includes employing it to run the cardiovascular model mentioned in Section 6.3 in order to examine the effects of heart disease therapies. To speed up HMC, one might try variance reduction techniques for MC algorithms (like stratified sampling), the method suggested in Section 6.3 on using the criteria to cut down on the space of possible input values that needs to be sampled from, and the method suggested in at the end of Section 6.4 on using two thresholds. Probably the method with the most speed-up potential is the brute-force approach of examining sample points in parallel. Parallel examination is easy to do because each of the points can be examined independently from the other points.

On matters other than algorithms, work needs to be done on finding the types of density bounds that are the most common, easiest to specify, and most useful. Candidates for easy-to-specify bounds are common densities with bounded parameters. An example is a Gaussian density with a mean between 0 and 1. One can generate such bounds by using information from parameter confidence intervals. Besides being easy-to-specify, a density bound needs to be tight enough for HMC to find a useful result: one that gives a low upper bound on the probability of some event occurring.

8.3 Observations

8.3.1 AIS

Work on AIS is an example of how the term "steady-state" is view dependent. A system is at a steady-state when its parameters are not changing with time. However, if (as is usual) one is only concerned with some of the parameters, then for one's purposes, regardless of how the parameters are changing or not changing in value, the system is at a steady-state when all the parameters of concern are maintaining constant values. With AIS, an iterating system can be viewed as being at a steady-state because for any non-constant system parameter, one is only interested in examining variants or functions of that parameter that stay constant: For a periodic parameter, the variants are the extreme values. For an accumulating parameter, the variant is the averaged rate of change in value over an iteration. The last variant is a bit strange in that what is claimed as constant is the particular function of time that is followed by an accumulating parameter averaged over an iteration. In this case, the function has the form $a \cdot t + b$, where a and b are constants, and t is time. Still, it does not seem too bizarre to say some system is at a steady-state because the rate(s) of change is constant (a is constant). However, what if a parameter constantly follows a particular function like $t \cdot \sin(t)$, for which there are no obvious characteristics (like some derivative) that stay constant? Would one be comfortable describing such a system as being at a steady-state? Probably not.

Still, even if such systems are not described as being in a steady-state, finding the function of time that a parameter "constantly" follows is useful. That is what differential equation solvers do. In fact, one way to look at AIS is that it is a simple method to solve a certain class of differential equations. Future work includes expanding the class of equations that AIS can handle, and finding other classes of equations that can be handled with other simple techniques.

AIS is also an example of a method to perform abstraction/aggregation. In this case, the abstraction/aggregation is over time, as opposed to space, parts, organ systems, etc.

8.3.2 Predicting with Varying Inputs

Much of this thesis deals with the task of predicting about how likely certain events are to occur in a system as complicated as the human cardiovascular system when the parameters can vary as much as in the human population (as opposed to the fluctuations of an individual over time). For this task, methods to estimate the relationships between parameter moments (means, variances, etc.) are sometimes useful at giving these relationships. However, the estimates can be quite inaccurate and there are no real guidelines for predicting when this will happen. So these methods to manipulate moment relationships are not so useful for actually making the predictions.

Monte Carlo simulation sometimes seems to work well for this task. It will work well when one can make a good characterization of the input parameter probability distribution. Unfortunately, finding a good characterization can be tricky when the "standard" multivariate probability densities (like jointly Gaussian or lognormal) don't seem to work. One way to use not so "standard" densities is given by Iman and Conover in [21]. Unfortunately, as mentioned in Section 4.2, with this method, one is not sure of how the input density is characterized beyond its marginals and correlation structure. It would be helpful if one could make some general statement about this method like the type made for the maximum entropy method: "It finds the density which is consistent with the given information and which maximizes the independence between the density's variables."

Having users place a bound on the input density is the alternative developed in this thesis for when Monte Carlo simulation does not work well. One of the methods in this thesis, SAB, produces analytic bounds on the probability of satisfying some criteria, but cannot deal with models having the size of the cardiovascular model used in this thesis. HMC, the other method in this thesis, is a type of Monte Carlo. Given a model of this size, HMC can sometimes make useful statements about the probability of certain events occurring being low. However, HMC may take a long time to conclude such statements, and if the given input density bound is too loose, may not be able to make very many, if any, useful conclusions. The latter is not too surprising.

So when trying to predict the likelihood of certain events in a system with the complexity and parameter variability of the human cardiovascular system, try using Monte Carlo simulation first. If this does not work well, try deriving an upper bound on the input parameter probability density and using HMC. Iman and Conover's work ([21]) has possibilities, but the work needs to be analyzed more.

8.3.3 All The Systems

This thesis present a number of different methods. Most of them: AIS, SAB and HMC, can be viewed as examples of constraint manipulation methods. That is, they all deal with information in not which everything needs to be known, but what is known is thought to be consistent. The methods try to conclude as much as possible with what is known, and the more that is known, the more they can conclude. Actually, many methods that do not have to handle conflicting information (beyond detecting it and stopping) can be viewed as types of constraint manipulation methods.

These presented methods also show how both symbolic *and* numeric techniques are needed to reason about continuous systems. In addition, these methods are examples of the two types of methods needed to reason about a continuous system: methods to help construct a model of that system and methods to observe what that model has to say about a particular situation.

Many potential model users are mainly interested/knowledgeable in some specific domain and not so much in modeling formalisms. To make model construction more accessible to such users, one needs methods that automatically construct/transform a piece of a model from an easier-to-give description of the part being modeled. AIS is an example of such a method: it builds a part of a steady-state model by transforming a description of an iterative process. As can be seen from this example, such methods need to perform symbol manipulation to create a model.

By itself, a model does not indicate how the modeled system might behave in a particular situation or set of situations. To predict how the system behaves, one needs methods to evaluate the model at the set of parameters values (or range of values or distribution of values or some combination of these) that describes the situation. This evaluation involves manipulating numbers. It also will probably involve some symbol manipulation in order to get the model's expressions in the form needed for evaluation. Many of the so called numeric methods are methods to perform such evaluations. Often, the symbol manipulation needed for these methods have been hidden by having a user perform the needed manipulations (such as solving equations and taking derivatives) before the problem is handed over to a computer. SAB, HMC, and GLO are examples of these evaluation methods, as are Monte Carlo and interval arithmetic methods.

8.4 Final Comments

The methods presented here help one reason about continuous systems, but much work remains to be done: all the presented methods are limited in the types of problems they can handle, and they sometimes can take a long time to derive an answer. More specifically, AIS combines a knowledge representation scheme for describing iterative dynamic systems with knowledge about certain behavioral invariants over time to analyze a small but useful subset of dynamic systems. HMC lets one analyze steady-state systems when only a bound on the probability density of the input parameter values is known, but HMC may take a long time to find a result, and the result may be quite ambiguous.

Appendix A

Some Region Probability Bounds Derivations

This appendix shows derivations for some of the expressions that bound the probability of being in a region α. Let $f(x_1, \ldots, x_n)$ be the probability density, and within α let x_i range between l_i and h_i. Then the probability of being in α is

$$F = \int_{l_n}^{h_n} \cdots \int_{l_1}^{h_1} f(x_1, \ldots, x_n) dx_1 \ldots dx_n.$$

A.1 Basic Bound

This section derives the following lower bound on F:

$$[\prod_{i=1}^{n}(h_i - l_i)][\min f(x_1, \ldots, x_n)],$$

which is the 'volume' of the region multiplied by the lowest density value within it. The minimization of f is over the x_1 through x_n values within α.

$$\begin{aligned} F &\geq \int_{l_n}^{h_n} \cdots \int_{l_1}^{h_1} [\min f(x_1, \ldots, x_n)] dx_1 \ldots dx_n \\ &\geq [\min f(x_1, \ldots, x_n)] \int_{l_n}^{h_n} dx_n \cdots \int_{l_1}^{h_1} dx_1 \\ &\geq [\min f(x_1, \ldots, x_n)] \prod_{i=1}^{n}(h_i - l_i) \end{aligned}$$

A.2 Bound Using Monotonicity

This section shows that if $\partial f / \partial x_1$ is always > 0 in α, then

$$F \geq [\prod_{i=1}^{n}(h_i - l_i)][(\min_{*} f(l_1, x_2, \ldots, x_n)) + (\min \frac{\partial f}{\partial x_1}(x_1, \ldots, x_n))(\frac{h_1 - l_1}{2})],$$

where the minimization of f is over the x_2 through x_n values within α ($\min_*$ means that x_1 is NOT part of the minimization), and the minimization of $\partial f/\partial x_1$ is over the x_1 through x_n values within α.

$$\begin{aligned}
F &= \int_{l_n}^{h_n} \cdots \int_{l_1}^{h_1} f(x_1, \ldots, x_n) dx_1 \ldots dx_n \\
&= \int_{l_n}^{h_n} \cdots \int_{l_1}^{h_1} [f(l_1, x_2, \ldots, x_n) + \int_{l_1}^{x_1} \frac{df}{dx_1}(x_1, \ldots, x_n) dx_1] dx_1 \ldots dx_n \\
&= \int_{l_n}^{h_n} \cdots \int_{l_1}^{h_1} [f(l_1, x_2, \ldots, x_n) + \int_{l_1}^{x_1} [\sum \frac{\partial f}{\partial x_i}(x_1, \ldots, x_n) \frac{dx_i}{dx_1}] dx_1] dx_1 \ldots dx_n
\end{aligned}$$

Since the x_i's are integrated independently of one another, dx_i/dx_1 is 0 for $i \neq 1$ and 1 for $i = 1$. So, the sum collapses down to the $\partial f/\partial x_1$ term:

$$\begin{aligned}
F &= \int_{l_n}^{h_n} \cdots \int_{l_1}^{h_1} [f(l_1, x_2, \ldots, x_n) + \int_{l_1}^{x_1} \frac{\partial f}{\partial x_1}(x_1, \ldots, x_n) dx_1] dx_1 \ldots dx_n \\
&\geq \int_{l_n}^{h_n} \cdots \int_{l_1}^{h_1} [(\min_* f(l_1, x_2, \ldots, x_n)) + \int_{l_1}^{x_1} (\min \frac{\partial f}{\partial x_1}(x_1, \ldots, x_n)) dx_1] dx_1 \ldots dx_n \\
&\geq [\int_{l_1}^{h_1} [(\min_* f(l_1, x_2, \ldots, x_n)) + (\min \frac{\partial f}{\partial x_1}(x_1, \ldots, x_n)) \int_{l_1}^{x_1} dx_1] dx_1] \times \\
&\quad [\int_{l_n}^{h_n} dx_n \cdots \int_{l_2}^{h_2} dx_2] \\
&\geq [\int_{l_1}^{h_1} [(\min_* f(l_1, x_2, \ldots, x_n)) + (\min \frac{\partial f}{\partial x_1}(x_1, \ldots, x_n))(x_1 - l_1)] dx_1] \prod_{i=2}^{n} (h_i - l_i) \\
&\geq [[(\min_* f(l_1, x_2, \ldots, x_n)) - l_1(\min \frac{\partial f}{\partial x_1}(x_1, \ldots, x_n))] \int_{l_1}^{h_1} dx_1 \\
&\quad + (\min \frac{\partial f}{\partial x_1}(x_1, \ldots, x_n)) \int_{l_1}^{h_1} x_1 dx_1] \prod_{i=2}^{n} (h_i - l_i) \\
&\geq [(\min_* f(l_1, x_2, \ldots, x_n))(h_1 - l_1) \\
&\quad + (\min \frac{\partial f}{\partial x_1}(x_1, \ldots, x_n))[-l_1(h_1 - l_1) + (\frac{h_1^2 - l_1^2}{2})]] \prod_{i=2}^{n} (h_i - l_i) \\
&\geq [(\min_* f(l_1, x_2, \ldots, x_n))(h_1 - l_1) + (\min \frac{\partial f}{\partial x_1}(x_1, \ldots, x_n)) \frac{(h_1 - l_1)^2}{2}] \prod_{i=2}^{n} (h_i - l_i) \\
&\geq [(\min_* f(l_1, x_2, \ldots, x_n)) + (\min \frac{\partial f}{\partial x_1}(x_1, \ldots, x_n)) \frac{(h_1 - l_1)}{2}] \prod_{i=1}^{n} (h_i - l_i)
\end{aligned}$$

A.3 Bound Using Convexity

This section shows that if $\partial^2 f/\partial x_1^2$ is always ≤ 0 in α (convex down), then

$$F \geq [\prod_{i=1}^{n} (h_i - l_i)][(\min_* f(l_1, x_2, \ldots, x_n)) + (\min_* f(h_1, x_2, \ldots, x_n))]/2,$$

where the minimization of f is over the x_2 through x_n values within α ($\min_*$ means that x_1 is NOT part of the minimization). Within α, f is convex down with respect to x_1, so $f(x_1, \ldots, x_n)$ is $\geq$ than the linear combination of

$$q(x_1)f(l_1, x_2, \ldots, x_n) + (1 - q(x_1))f(h_1, x_2, \ldots, x_n),$$

where $q(x_1) = (x_1 - l_1)/(h_1 - l_1)$. So,

$$\begin{aligned}
F &\geq \int_{l_n}^{h_n} \cdots \int_{l_1}^{h_1} [q(x_1)f(l_1, x_2, \ldots, x_n) + (1 - q(x_1))f(h_1, x_2, \ldots, x_n)]dx_1 \ldots dx_n \\
&\geq \int_{l_n}^{h_n} \cdots \int_{l_1}^{h_1} [q(x_1)(\min_* f(l_1, x_2, \ldots, x_n)) \\
&\quad +(1 - q(x_1))(\min_* f(h_1, x_2, \ldots, x_n))]dx_1 \ldots dx_n \\
&\geq [(\min_* f(l_1, x_2, \ldots, x_n)) \int_{l_1}^{h_1} q(x_1)dx_1 \\
&\quad +(\min_* f(h_1, x_2, \ldots, x_n)) \int_{l_1}^{h_1} (1 - q(x_1))dx_1] \int_{l_n}^{h_n} dx_n \cdots \int_{l_2}^{h_2} dx_2 \\
&\geq [(\min_* f(l_1, x_2, \ldots, x_n))[\frac{x_1^2/2 - l_1x_1}{h_1 - l_1}]_{l_1}^{h_1} \\
&\quad +(\min_* f(h_1, x_2, \ldots, x_n))[x_1 - \frac{x_1^2/2 - l_1x_1}{h_1 - l_1}]_{l_1}^{h_1}] \prod_{i=2}^{n}(h_i - l_i) \\
&\geq [(\min_* f(l_1, x_2, \ldots, x_n))(\frac{h_1 - l_1}{2}) + (\min_* f(h_1, x_2, \ldots, x_n))(\frac{h_1 - l_1}{2})] \prod_{i=2}^{n}(h_i - l_i) \\
&\geq [(\min_* f(l_1, x_2, \ldots, x_n)) + (\min_* f(h_1, x_2, \ldots, x_n))][\prod_{i=1}^{n}(h_i - l_i)]/2
\end{aligned}$$

Appendix B

Cardiovascular Model

This appendix gives a description of a simplified version of the human cardiovascular model [25, 26] that is used in the thesis. The model has 58 variables and constants. They are listed in Section B.1. Inequality constraints on them are given in Section B.2. Equations involving variables and constants are in Section B.3. Section B.4 gives the input for the larger example for HMC shown in Section 6.3. The model in this appendix differs from the one in the references in the following ways:

1. The piecewise linear formulas (in equations for LAP, LVE, LVO, RVE, HR) in the original model were transformed into formulas using exponentiation with constants. The latter formulas are easier to deal with when solving for an arbitrary variable in the formula.

2. The system is assumed always to be in equilibrium: The condition $RVO = CO$ is added.

B.1 Variables and Constants

The model variables and constants are:

AR : A measure of how bad the Aortic Regurgitation (a disease) is. The higher the value, the worse the effects.

ARF : The rate at which Aortic Regurgitation causes blood to Flow back into the heart.

AS : A measure of how bad the Aortic Stenosis (a disease) is. The higher the value, the worse the effects.

BB : A measure of the amount of Beta-Blocker (a type of drug) given. The measure is monotonic in the amount given.

bhr : the Base Heart Rate

BP : Blood Pressure

bpb : the Blood Pressure Base

bsvr : Coefficient for the effect of Beta-blocker on SVR. A constant set to -0.4.

BV: the total Blood Volume in the body

CO: Cardiac Output

DT: Diastolic Time

dtb : Diastolic Time Base, a constant set to 42.7

dv : Dead Volume, a constant set to 3.2

E: A measure of the amount of Exercise. The higher the value, the harder a person is exercising.

evv : Coefficient for the effect of Exercise on Venous-Volume. A constant set to 0.11.

H: A measure of the amount of Hydralazine (a drug) given. The measure is monotonic in the amount given.

hi : Coefficient for the effect of Hydralazine on the Inotropic state. A constant set to 0.3.

HR: Heart Rate

I: Inotropic state

LAP: Left Atrial Pressure

LVC: Left Ventricular Compliance

LVE: How much the Left Ventricle Empties at the end of compressing it compared to normal. 1.0 is normal, less means that less of the chamber is emptied.

$LVEDP$: Left Ventricular End Diastolic Pressure

LVO: Left Ventricular Output

lvok1 : a constant for finding LVO

$LVSF$: Left Ventricular Systolic Function, a measure of the left ventricle's 'strength'. 1.0 is normal, less means a weaker left ventricle.

lvsfb : Left Ventricular Systolic Function Base

MS: A measure of how bad the Mitral Stenosis (a disease) is. The higher the value, the worse the effects.

N: A measure of the amount of Nitroglycerin (a drug) given. The measure is monotonic in the amount given.

nks : Coefficient for the effect of Nitroglycerin on *SVR*. A constant set to 0.

nkv : Coefficient for the effect of Nitroglycerin on Venous constriction. A constant set to 0.12.

PAP : Pulmonary Arterial Pressure

PV : Pulmonary Volume (volume of blood in the pulmonary area)

PVR : Pulmonary Vascular Resistance

pvrk1 : a constant for finding *PVR*

pvrk2 : another constant for finding *PVR*, set to 0.17

RAP : Right Atrial Pressure

RVC : Right Ventricular Compliance

RVE : Like *LVE*, but for the right ventricle.

RVEDP : Right Ventricular End Diastolic Pressure

RVO : Right Ventricular Output

rvok1 : a constant for finding the *RVO*

RVR : the Resistance of the Venous Return

RVSF : Like *LVSF*, but for the right ventricle.

rvsfb : Right Ventricular Systolic Function Base, a constant set to 1.0

SP : Systolic Pressure

SS : Amount of Sympathetic Stimulation. 1.0 is at normal or rest state.

sse : Coefficient for the effect on Sympathetic Stimulation of Exercise. A constant set to 47.5.

ST : Systolic Time

stb : Systolic Time Base, a constant set to $60 - dtb$

SVR : Systemic Vascular Resistance

svrb : Systemic Vascular Resistance Base

svre : Coefficient for the effect on *SVR* of Exercise. A constant set to -4.65.

svrr : *SVR* Response, a constant set to 5.0

ts : Time Slope, a constant set to 0.075

var	lb	ub	var	lb	ub	var	lb	ub
AR	0.0		LVC	0.0		RVE	0.0	2.0
ARF	0.0		LVE	0.0	2.0	$RVEDP$	−2.0	20.0
AS	0.0		$LVEDP$	2.0	40.0	RVO	1.0	30.0
BB	0.0		LVO	1.0	30.0	$rvok1$	−20.0	
bhr	45.0	110.0	$lvok1$	−16.5		RVR	1.0	2.0
BP	50.0	200.0	$LVSF$	0.0	2.0	$RVSF$	0.0	2.0
bpb	50.0	150.0	$lvsfb$	0.0		$rvsfb$	0.0	
BV	3.0	8.0	MS	0.0	10.0	SP	50.0	250.0
CO	1.0	30.0	N	0.0		SS	0.0	2.0
DT	20.0	40.0	PAP	10.0	100.0	ST	20.0	40.0
E	0.0	5.0	PV	0.3	8.0	SVR	5.0	40.0
H	0.0		PVR	0.1	6.0	$svrb$	4.0	20.0
HR	25.0	250.0	$pvrk1$	−5.1		VS	0.0	5.0
I	0.0	2.0	RAP	−2.0	20.0	VC	0.0	
LAP	2.0	60.0	RVC	0.0		VV	0.0	8.0

Table B.1: Lower and Upper Bounds for Cardiovascular Model Variables

VS: Amount of Vagal Stimulation. 1.0 is at normal or rest state.

VC: Venous Constriction

VV: Venous Volume (volume of blood in the veins)

B.2 Inequality Constraints

Inequality constraints between the variables are:

$$PV \leq BV, \quad VV \leq BV, \quad PV + VV \leq BV, \quad LVEDP \leq LAP \leq PAP.$$

Lower and upper bounds on the variables and constants (ones not preset to a particular value) are given in Table B.1. The "var" columns give the variables and constants, the "lb" columns give the corresponding lower bounds, and the "ub" columns give the corresponding upper bounds. The bounds came from a combination of definitions, physiological limits, limits set by the patient being alive, and limits derived from a combination of the equations in Section B.3 and the limits that already exist.

B.3 Equations

The model equations are:

$$\begin{aligned} CO &= RVO \\ DT &= dtb - ts \cdot HR \end{aligned}$$

$$\begin{aligned}
ST &= stb + ts \cdot HR \\
LAP &= 59 \cdot PV^4 + 1.316 \\
LAP &= LVEDP + 620 \cdot MS \cdot (CO/DT)^2 \\
I &= SS \cdot (1 + hi \cdot H)/(1 + BB) \\
LVSF &= I \cdot lvsfb \\
LVE &= LVSF \cdot (1.0 - 0.002 \cdot (SP/70.0)^6) \\
LVO &= LVC \cdot LVE \cdot (17.5 \cdot (1 - (0.01 \cdot LVEDP + 1.011)^{-6}) - 1 + lvok1) \\
ARF &= 0.00137 \cdot (BP - 30.0) \cdot DT \cdot AR \\
CO &= LVO - ARF \\
SVR &= \frac{svrb + SS \cdot svrr \cdot (1 - nks \cdot N) \cdot (1 - bsvr \cdot BB) + svre \cdot E}{1 + 0.64 \cdot H} \\
BP &= CO \cdot SVR \\
SP &= BP + 228 \cdot AS \cdot (CO/ST)^2 \\
SS &= 1.0 - 0.03 \cdot (BP - bpb - sse \cdot E) \\
VC &= 1.0 - nkv \cdot N \\
BV &= VV + PV + \frac{dv}{evv \cdot E + ((0.7 + 0.3 \cdot SS) \cdot VC)} \\
RVR &= SVR \cdot 0.025 + 0.9 \\
PVR &= (pvrk1 + RVO \cdot pvrk2) \cdot (1 + 0.5 \cdot BB)/(1 + 0.625 \cdot H) \\
RAP &= 5.7 \cdot VV - RVR \cdot CO \\
RVEDP &= RAP \\
RVSF &= I \cdot rvsfb \\
RVE &= RVSF \cdot (1.0 - 0.001 \cdot (PAP/15.5)^4) \\
RVO &= RVC \cdot RVE \cdot 1.375 \cdot (RVEDP + rvok1) \\
PAP &= LAP + PVR \cdot RVO \\
VS &= (1.0 + 0.033 \cdot (BP - bpb))/(1 + 7 \cdot E) \\
HR &= bhr + 37.5 \cdot SS/(1 + BB) - 68 \cdot (1 - (0.092 \cdot VS + 1)^{-5})
\end{aligned}$$

B.4 Input for Some of the Larger Examples

This appendix section shows the input for Sections 6.3, 6.4 and 7.5. The input is based on a control run of experiments done on ten patients with mitral stenosis (run A in [14]). The patients are at rest, have no drugs in their systems, and have no diseases other than mitral stenosis. Because of this, the following equations are entered:

$$E = 0,\ BP = bpb,\ RAP = 0,\ AR = AS = 0,\ LVC = RVC = lvsfb = rvsfb = 1,$$
$$BB = H = N = 0.$$

The sample means, standard deviations and correlation coefficients of the six input parameters that can vary in value are given in Table B.2. The statistics for *HR*, *LVEDP* and *PAP* are based on the reference's data on those parameters. Assuming

			Correlation Coefficients					
NAME	MEAN	S.D.	*BP*	*CO*	*HR*	*LAP*	*LVEDP*	*PAP*
BP	87.67	14.48	1.0	-0.116	0.523	0.065	-0.458	0.130
CO	5.7	1.104	-0.116	1.0	0.420	-0.410	-0.126	-0.387
HR	82.9	14.49	0.523	0.420	1.0	-0.182	-0.428	-0.379
LAP	23.0	6.110	0.065	-0.410	-0.182	1.0	0.080	0.565
LVEDP	9.0	3.621	-0.458	-0.126	-0.428	0.080	1.0	-0.329
PAP	32.0	10.18	0.130	-0.387	-0.379	0.565	-0.329	1.0

Table B.2: Mitral Stenosis Patient Statistics

that the wedge pressure (*PWP*) is a good approximation of *LAP*, the *LAP* statistics are based on the reference's *PWP* measurements. Assuming that the patients have a body surface area of $1.74m^2$ (the average for humans), data for *CO* was derived from the cardiac index (*CI*) measurements by using the formula $CO = 1.74 \cdot CI$. Finally, assuming that the patients have a pulse pressure of about $50mmHg$, data for *BP* was derived from the left ventricular systolic pressure (*LVSP*) measurements by using the formula $BP = LVSP/3 + 2 \cdot (LVSP - 50)/3$.

Appendix C

Derivations for GLO

This appendix derives various equations that the GLO method uses to relate moments (means, variances and covariances) in linear combinations, and products & exponentiations of jointly lognormal random variables. The appendix also derives some equations that show some of the differences between lognormal and Gaussian densities with the same means and variances.

C.1 Linear Combinations

First for a linear combination of random variables, $X = \sum_i a_i \cdot X_i$, where the a_i's are constants and the X_i's are random variables, the following formulas hold for all densities [10, Ch. 4]:

$$E[X] = \sum_i a_i \cdot E[X_i] \tag{C.1}$$

$$V[X] = \sum_i a_i^2 \cdot V[X_i] + 2 \cdot \sum_i \sum_{i<k} a_i \cdot a_k \cdot C[X_i, X_k]. \tag{C.2}$$

For a covariance, $C[X,Z]$, use $C[X,Z] = E[X \cdot Z] - E[X] \cdot E[Z]$ and (C.1) to get

$$\begin{aligned} C[X,Z] &= \sum_i a_i \cdot E[X_i \cdot Z] - \sum_i a_i \cdot E[X_i] \cdot E[Z] \\ &= \sum_i a_i \cdot (E[X_i \cdot Z] - E[X_i] \cdot E[Z]) \\ C[X,Z] &= \sum_i a_i \cdot C[X_i, Z]. \end{aligned} \tag{C.3}$$

C.2 Correspondence Between Y and $X = \ln Y$

Next are products and exponentiations of jointly lognormal variables, such as $Y = \prod_i Y_i^{a_i}$, where the a_i's are constants and the Y_i's are jointly lognormal random variables. Before the moment equations for the products and exponentiations can be derived, the correspondence between the moments of the Y_i's and the Y_i's logarithms need to be derived. This section shows these derivations.

Let $X = \ln Y$ and $X_i = \ln Y_i$, then $Y = \prod_i Y_i^{a_i}$ corresponds to $X = \sum_i a_i \cdot X_i$. The following two equations give the relations between the means and variances of X and Y (they also hold for Y_i and X_i) [3, Ch. 2]:

$$\begin{aligned} E[Y] &= \exp(E[X] + V[X]/2) && \text{(C.4)} \\ V[Y] &= E^2[Y] \cdot (e^{V[X]} - 1). && \text{(C.5)} \end{aligned}$$

Rearranging (C.5) yields

$$V[X] = \ln(V[Y]/E^2[Y] + 1). \tag{C.6}$$

Substituting this equation into (C.4) and rearranging yields

$$E[X] = \ln(E^2[Y]/\sqrt{V[Y] + E^2[Y]}). \tag{C.7}$$

To get the covariance relationship between X's and Y's, rearrange $V[X_i + X_k] = V[X_i] + 2 \cdot C[X_i, X_k] + V[X_k]$, the two variable sum version of (C.2), and then exponentiate into

$$\exp(C[X_i, X_k]) = \exp(V[X_i + X_k]/2 - V[X_i]/2 - V[X_k]/2). \tag{C.8}$$

Also, use the correspondence between $Y = Y_i \cdot Y_k$ and $X = X_i + X_k$, and (C.4) to get

$$\begin{aligned} E[Y_i \cdot Y_k] &= \exp(E[X_i + X_k] + V[X_i + X_k]/2) \\ V[X_i + X_k]/2 &= \ln(E[Y_i \cdot Y_k]) - E[X_i + X_k]. && \text{(C.9)} \end{aligned}$$

Substitute $E[Y_i \cdot Y_k] = C[Y_i, Y_k] + E[Y_i] \cdot E[Y_k]$ and the two variable sum version of (C.1) into (C.9) to get

$$V[X_i + X_k]/2 = \ln(C[Y_i, Y_k] + E[Y_i] \cdot E[Y_k]) - E[X_i] - E[X_k]. \tag{C.10}$$

Then, substitute (C.10) into (C.8) and rearrange to get

$$\begin{aligned} \exp(C[X_i, X_k]) &= \exp(\ln(C[Y_i, Y_k] + E[Y_i] \cdot E[Y_k]) \\ &\quad -E[X_i] - E[X_k] - V[X_i]/2 - V[X_k]/2) \\ &= (C[Y_i, Y_k] + E[Y_i] \cdot E[Y_k]) \cdot \exp(-E[X_i] - V[X_i]/2) \cdot \\ &\quad \exp(-E[X_k] - V[X_k]/2) \\ &= \frac{C[Y_i, Y_k] + E[Y_i] \cdot E[Y_k]}{\exp(E[X_i] + V[X_i]/2) \cdot \exp(E[X_k] + V[X_k]/2)}. \end{aligned}$$

Substitute in versions of (C.4) and rearrange to yield

$$\exp(C[X_i, X_k]) = \frac{C[Y_i, Y_k]}{E[Y_i] \cdot E[Y_k]} + 1. \tag{C.11}$$

C.3 Products

This section derives the moment equations for a product of the jointly lognormal variables Y_i's. Let $Y = \prod_i Y_i$. $X_i = \ln Y_i$ means that $Y_i = \exp(X_i)$, so $Y = \exp(\sum_i X_i)$ and $(\ln Y = \sum_i X_i)$.

For $E[Y]$, substitute $(\ln Y = \sum_i X_i)$ into (C.4) and rearrange with versions of (C.1) and (C.2) to get

$$\begin{aligned}
E[Y] &= \exp(E[\ln Y] + V[\ln Y]/2) \\
&= \exp(E[\sum_i X_i] + V[\sum_i X_i]/2) \\
&= \exp(\sum_i E[X_i] + (\sum_i V[X_i] + 2 \cdot \sum_i \sum_{i<k} C[X_i, X_k])/2) \\
&= \exp(\sum_i (E[X_i] + V[X_i]/2)) \cdot \exp(\sum_i \sum_{i<k} C[X_i, X_k]) \\
&= \prod_i \exp(E[X_i] + V[X_i]/2) \cdot \prod_i \prod_{i<k} \exp(C[X_i, X_k]).
\end{aligned}$$

Now substitute in versions of (C.4), (C.11) and $E[Y_i \cdot Y_k] = C[Y_i, Y_k] + E[Y_i] \cdot E[Y_k]$ to get

$$\begin{aligned}
E[Y] &= (\prod_i E[Y_i]) \cdot \prod_i \prod_{i<k} (\frac{C[Y_i, Y_k]}{E[Y_i] \cdot E[Y_k]} + 1) \\
&= (\prod_i E[Y_i]) \cdot \prod_i \prod_{i<k} (\frac{C[Y_i, Y_k] + E[Y_i] \cdot E[Y_k]}{E[Y_i] \cdot E[Y_k]}) \\
E[Y] &= (\prod_i E[Y_i]) \cdot \prod_i \prod_{i<k} (\frac{E[Y_i \cdot Y_k]}{E[Y_i] \cdot E[Y_k]}). \qquad \text{(C.12)}
\end{aligned}$$

For $V[Y]$, exponentiate a version of (C.6), substitute in $(\ln Y = \sum_i X_i)$ and then (C.2), rearrange, and then substitute in versions of (C.11) and exponentiated versions of (C.6):

$$\begin{aligned}
(\frac{V[Y]}{E^2[Y]} + 1) &= \exp(V[\ln Y]) \\
&= \exp(V[\sum_i X_i]) \\
&= \exp(\sum_i V[X_i] + 2 \cdot \sum_i \sum_{i<k} C[X_i, X_k]) \\
&= (\prod_i \exp(V[X_i])) \cdot (\prod_i \prod_{i<k} (\exp(C[X_i, X_k]))^2) \\
(\frac{V[Y]}{E^2[Y]} + 1) &= (\prod_i (\frac{V[Y_i]}{E^2[Y_i]} + 1)) \cdot (\prod_i \prod_{i<k} (\frac{C[Y_i, Y_k]}{E[Y_i] \cdot E[Y_k]} + 1)^2). \qquad \text{(C.13)}
\end{aligned}$$

$(1+\delta_i) \cdot (1+\delta_k) = 1 + \delta_i + \delta_k + \delta_i \cdot \delta_k$. For δ_i's and δ_k's which have magnitudes much less than 1, $\delta_i \cdot \delta_k$ becomes relatively insignificant, and the approximation $(1+\delta_i) \cdot (1+\delta_k) \approx 1 + \delta_i + \delta_k$ holds. Generalizing this approximation to n terms is $\prod_i (1+\delta_i) \approx$

$1 + \sum_i \delta_i$. This generalized approximation assumes both that $\forall i : (1 + \delta_i) \approx 1$ (in other words $\forall i : |\delta_i| << 1$) and that the multiplications can be ordered so that all intermediate terms are ≈ 1. One way to check the latter ordering assumption is to see if the product is ≈ 1.

Assuming that $|\frac{V[Y]}{E^2[Y]}|$ and all the $|\frac{V[Y_i]}{E^2[Y_i]}|$'s and $|\frac{C[Y_i,Y_k]}{E[Y_i]\cdot E[Y_k]}|$'s are much less than 1, one can make the following approximation for (C.13):

$$\begin{aligned}(1 + \frac{V[Y]}{E^2[Y]}) &= \prod_i [(1 + \frac{V[Y_i]}{E^2[Y_i]}) \cdot \prod_{i<k} (1 + \frac{C[Y_i, Y_k]}{E[Y_i] \cdot E[Y_k]})^2] \\ &\approx 1 + \sum_i \frac{V[Y_i]}{E^2[Y_i]} + 2 \cdot \sum_i \sum_{i<k} \frac{C[Y_i, Y_k]}{E[Y_i] \cdot E[Y_k]}.\end{aligned}$$

Subtracting one from all sides yields

$$\frac{V[Y]}{E^2[Y]} \approx \sum_i \frac{V[Y_i]}{E^2[Y_i]} + 2 \cdot \sum_i \sum_{i<k} \frac{C[Y_i, Y_k]}{E[Y_i] \cdot E[Y_k]}. \qquad \text{(C.14)}$$

For $C[Y, Z]$, substitute $(\ln Y = \sum_i X_i)$ and then a version of (C.3) into a version of (C.11). Rearrange and then substitute in versions of (C.11) to yield

$$\begin{aligned}(\frac{C[Y, Z]}{E[Y] \cdot E[Z]} + 1) &= \exp(C[(\ln Y), (\ln Z)]) \\ &= \exp(C[(\sum_i X_i), (\ln Z)]) \\ &= \exp(\sum_i C[X_i, (\ln Z)]) \\ &= \prod_i \exp(C[X_i, (\ln Z)]) \\ (\frac{C[Y, Z]}{E[Y] \cdot E[Z]} + 1) &= \prod_i (\frac{C[Y_i, Z]}{E[Y_i] \cdot E[Z]} + 1). \qquad \text{(C.15)}\end{aligned}$$

Assuming that $|\frac{C[Y,Z]}{E[Y]\cdot E[Z]}|$ and all the $|\frac{C[Y_i,Z]}{E[Y_i]\cdot E[Z]}|$'s are $<< 1$, one can use the same generalized small $|\delta_i|$ approximation used for $\frac{V[Y]}{E^2[Y]}$ to get the approximation (subtract one from both sides)

$$\frac{C[Y, Z]}{E[Y] \cdot E[Z]} \approx \sum_i \frac{C[Y_i, Z]}{E[Y_i] \cdot E[Z]}. \qquad \text{(C.16)}$$

C.4 Exponentiations

Now the moments of exponentiating a lognormal variable Y are derived. This section uses the correspondence $a \cdot X = \ln Y^a$ heavily.

For $E[Y^a]$, take a version of (C.4), substitute in versions of (C.1) and (C.2), and then versions of (C.6) and (C.7). Rearrange to get

$$E[Y^a] = \exp(E[a \cdot X] + V[a \cdot X]/2)$$

$$
\begin{aligned}
&= \exp(a \cdot E[X] + a^2 \cdot V[X]/2) \\
&= \exp(a \cdot \ln(E^2[Y]/\sqrt{V[Y] + E^2[Y]}) + \frac{a^2}{2} \cdot \ln(V[Y]/E^2[Y] + 1)) \\
&= \exp(\ln(\frac{E^a[Y]}{(\sqrt{(V[Y]/E^2[Y]) + 1})^a}) + \ln(\frac{V[Y]}{E^2[Y]} + 1)^{(a^2/2)}) \\
&= \exp(\ln(E^a[Y] \cdot (\frac{V[Y]}{E^2[Y]} + 1)^{(-a/2)} \cdot (\frac{V[Y]}{E^2[Y]} + 1)^{(a^2/2)})) \\
E[Y^a] &= E^a[Y] \cdot (\frac{V[Y]}{E^2[Y]} + 1)^{(a(a-1)/2)} \qquad \text{(C.17)}
\end{aligned}
$$

For $V[Y^a]$, take a version of (C.6) and exponentiate both sides. Substitute in a version of (C.2) and then a version of (C.6) to yield

$$
\begin{aligned}
\ln(V[Y^a]/E^2[Y^a] + 1) &= V[a \cdot X] \\
(V[Y^a]/E^2[Y^a] + 1) &= \exp(V[a \cdot X]) \\
&= \exp(a^2 \cdot V[X]) \\
&= \exp(a^2 \cdot \ln(V[Y]/E^2[Y] + 1)) \\
&= \exp(\ln(V[Y]/E^2[Y] + 1)^{a^2}) \\
(\frac{V[Y^a]}{E^2[Y^a]} + 1) &= (\frac{V[Y]}{E^2[Y]} + 1)^{a^2}. \qquad \text{(C.18)}
\end{aligned}
$$

For $C[Y^a, Z]$, take a version of (C.11), substitute in a version of (C.3) and then the logarithm of a version of (C.11). Rearrange to get

$$
\begin{aligned}
(\frac{C[Y^a, Z]}{E[Y^a] \cdot E[Z]} + 1) &= \exp(C[(a \cdot X), (\ln Z)]) \\
&= \exp(a \cdot C[X, (\ln Z)]) \\
&= \exp(a \cdot \ln(\frac{C[Y, Z]}{E[Y] \cdot E[Z]} + 1)) \\
&= \exp(\ln(\frac{C[Y, Z]}{E[Y] \cdot E[Z]} + 1)^a) \\
(\frac{C[Y^a, Z]}{E[Y^a] \cdot E[Z]} + 1) &= (\frac{C[Y, Z]}{E[Y] \cdot E[Z]} + 1)^a. \qquad \text{(C.19)}
\end{aligned}
$$

C.5 Differences Between Lognormals and Gaussians

This section derives some of the equations that demonstrate the difference between a lognormal and Gaussian density with the same mean and variance. As before, let $X = \ln Y$.

Since a Gaussian density is symmetric, a Gaussian random variable has a coefficient of skewness $\gamma_1 = 0$. From [3, Ch. 2], the coefficient of skewness for a lognormal

Y is $\gamma_1 = \eta^3 + 3 \cdot \eta$, where $\eta^2 = \exp(\sigma^2) - 1$ and σ^2 is $V[X]$. Make the appropriate substitutions to obtain

$$\gamma_1 = (\sqrt{\exp(V[X]) - 1})^3 + 3 \cdot \sqrt{\exp(V[X]) - 1}.$$

Then substitute in (C.6) to get

$$\begin{aligned}
\gamma_1 &= (\sqrt{\exp(\ln(V[Y]/E^2[Y] + 1)) - 1})^3 + 3 \cdot \sqrt{\exp(\ln(V[Y]/E^2[Y] + 1)) - 1} \\
&= (\sqrt{V[Y]/E^2[Y]})^3 + 3 \cdot \sqrt{V[Y]/E^2[Y]} \\
&= (\sqrt{(s.d.)^2/\text{mean}^2})^3 + 3 \cdot \sqrt{(s.d.)^2/\text{mean}^2} \\
\gamma_1 &= |\frac{s.d.}{\text{mean}}|^3 + 3 \cdot |\frac{s.d.}{\text{mean}}|. \qquad \text{(C.20)}
\end{aligned}$$

From [3, Ch. 2], if Y is lognormal, its mean is at $\exp(\mu + \sigma^2/2)$ and its median is at $\exp(\mu)$, where μ is $E[X]$ and σ^2 is $V[X]$. Making the appropriate substitutions, the ratio of Y's mean to its median would be $\exp(V[X]/2)$. If Y is Gaussian, Y's mean and median would be equal. So, the ratio of Y's median if it were Gaussian to Y's median if it were lognormal is also $\exp(V[X]/2)$. Substituting in (C.6) yields

$$\begin{aligned}
\frac{\text{Gauss\_median}}{\text{lognormal\_median}} &= \exp((\ln(V[Y]/E^2[Y] + 1))/2) \\
&= \exp(\ln\sqrt{(s.d.)^2/\text{mean}^2 + 1}) \\
\frac{\text{Gauss\_median}}{\text{lognormal\_median}} &= \sqrt{(\frac{s.d.}{\text{mean}})^2 + 1} \qquad \text{(C.21)}
\end{aligned}$$

So both the coefficients of skewness and the medians indicate that as $|s.d./\text{mean}|$ ratio increases, a Gaussian and lognormal density with the same mean and variance will be more different.

Appendix D

Using the AIS Implementation

This appendix describes how to use the current implementation of AIS, the program that analyzes iterative dynamic systems. AIS has been implemented on Symbolics[1] workstations (Lisp machines) using the Genera 7 version of the Symbolics Common Lisp programming language. The parts of this language used are essentially Common Lisp [46] with the following additions: the LOOP iteration macro, the *flavors* object oriented programming system, some graphics commands, and some system development commands.

The next section describes how to get access to the current implementation. Then the following sections describe the format of input and the commands to get the output respectively.

D.1 Access

The source code for the current implementation is stored as three files on the Zermatt file server in the MIT Lab for Computer Science: `Z:>ay>cycle.lisp`, `Z:>ay>cycle-analyze.lisp`, and `z:>ay>cycle-util.lisp`. The compiled version of the code has the same file names but with `.bin` replacing `.lisp`. The examples used for the thesis are stored in the file `z:>ay>cycle-ex.lisp`. The four `.lisp` files are also listed in Appendix E. The implementation is organized as a system with the name *Cycle* in the package CYCLE.

To define the system, evaluate the following (loading the file `z:>ay>lispm-init.lisp` will do this):

```
;Cycle system
(defpackage cycle (:use symbolics-common-lisp))

(defsystem cycle (:default-pathname "z:>ay>"
  :default-package cycle :package-override cycle)
  (:serial "cycle-util" "cycle" "cycle-analyze"))
```

The Cycle system uses an inequality reasoning system by Elisha Sacks [36, 38]. So before loading or compiling the Cycle system, load the inequality reasoning system

[1]Symbolics, Inc., 8 New England Executive Park, Burlington, MA 01803.

by loading the file `z:>elisha>qm>system`. Then load in my patches to this system by loading the file `z:>ay>qmfix` (this file is also listed in Appendix E).

Now load the Cycle system with the command `load system cycle`.

D.2 Input

To analyze an iterative dynamic system at steady-state, one needs to create an object of the type *cycle* that contains a description of system to be analyzed.

D.2.1 Simplified Example

An example of this creation is the following: Suppose one had the following simplified description of a ventricle: The symbol *HR* gives the rate at which the ventricle beat sequence repeats. The constants are *Pi* and *Po*. The periodic parameters are P and V. The accumulating parameters are the amount of work done by the blood in moving through the ventricle (W), and the amount of blood that has entered the ventricle (Bi) and left the ventricle (Bo). The static conditions on the constants are:

$$Pi < Po, \quad Pi^{1/2} > Po^2, \quad 0 \leq Pi, \quad 0 \leq Po.$$

There are four phases in the sequence. Each phase has a name, condition(s), and equation(s) for value changes. In order, the phases are (as before, π_b and π_e stand for the periodic parameter π's value at the beginning and end of the phase respectively, and α_c stands for the accumulating parameter α's change in value during the phase):

1. Isovolumetric Contraction: $0 \leq V, \quad P_e = Po$.
2. Ejection: $0 \leq V_b$, $0 \leq V_e$, $V_e = Po^2$, $W_c = -P \cdot Bo_c$, $Bo_c = V_b - V_e$.
3. Isovolumetric Relaxation: $0 \leq V, \quad P_e = Pi$.
4. Filling: $0 \leq V_b, \quad 0 \leq V_e, \quad V_e = Pi^{1/2}, \quad W_c = P \cdot Bi_c, \quad Bi_c = V_e - V_b$.

This description would be translated into the AIS implementation's terms by the Lisp code shown in Figure D.1 to create an instance of a `cycle` object (the call to `make-instance`) and then set the variable `c` to point to that object. The quote marks tell the Lisp interpreter/compiler not to evaluate the argument just behind the quote mark, but to use the argument as is.

D.2.2 Explanation of Input

The code in Figure D.1 is an example of a call to the function `make-instance` with `'cycle` as the first argument (for the type of object to be created) and the following keyword arguments:

`:name` A Lisp object of any type to use as a label for the system description on the output.

```
(setq c
  (make-instance 'cycle
    :name 'simplified-ventricle
    :rate 'HR
    :constants '(Pi Po)
    :periodic-vars '(P V)
    :accum-vars '(W Bi Bo)
    :conditions '((< Pi Po) (> (expt Pi 1/2) (expt Po 2.))
                  (<= 0 Pi) (<= 0 Po))
    :phases
    '((iso-volumetric-contraction nil ((<= 0 V) (:= P_e Po)))
      (ejection nil ((<= 0 V_b) (<= 0 V_e) (:= V_e (expt Po 2.))
                     (:= W_c (* -1. P Bo_c)) (:= Bo_c (- V_b V_e))))
      (iso-volumetric-relaxation nil ((<= 0 V) (:= P_e Pi)))
      (filling nil ((<= 0 V_b) (<= 0 V_e) (:= V_e (expt Pi 1/2))
                   (:= W_c (* P Bi_c)) (:= Bi_c (- V_e V_b))))
      )))
```

Figure D.1: Input for Simplified Example for AIS Implementation

`:rate` The *constant* parameter that gives the rate of sequence repetition.

`:constants` A list of all the other *constant* parameters.

`:periodic-vars` A list of all the *periodic* parameters.

`:accum-vars` A list of all the *accumulating* parameters.

`:conditions` A list of all the *conditions* on the constant parameters.

`:phases` A list of the *phases* in the sequence of parameter changes that is iterated. The list is in the order in which the phases occur.

Such a call to `make-instance` will perform the preliminary processing described in Section 2.2 on the call's arguments and (if no errors or inconsistencies are found) return an appropriate *cycle* object. Details on the keyword argument values are as follows:

Constant parameters that are numbers are represented as numbers and those that are symbols are represented as symbols. The notation and representation in Lisp for *constant* parameters that are "arbitrary" functions are dealt with later in Subsection D.2.3. All *periodic* and *accumulating* parameters are represented as symbols.

Conditions on the constant parameters are given as if they were normal Lisp expressions that when evaluated would return true or false. Each condition is a list with three elements. The first is one of the following symbols: `<`, `<=`, `>=`, `>`. The second and third elements are given as if they are normal Lisp expressions that when evaluated would return a number. The allowed functions on these expressions are `+`, `-`, `*`, `/`, `exp`, `expt` and `log`. `-` and `/` are allowed at most two arguments. Besides function

calls, numbers and *constant* parameters (and partial derivatives of such parameters that are "arbitrary" functions) are also allowed in the expressions. For example, the condition $0 < Pi + Po^2$ would turn into the Lisp expression
`(< 0 (+ Pi (expt Po 2.)))`.

Each *phase* is a list of three elements. The first element is a Lisp object of any type to use as a label for the phase on the output. The second object is the Lisp symbol `nil`.[2] The third element is a list of all the *conditions* and *parameter value changes* for the phase.

Before describing the elements in this list, the parameter representations used to describe the value changes in a phase needs to be discussed. *Constant* parameters do not change in value, so they are just referred to normally. The same is true for *periodic* parameters when they are referenced in a phase in which they do not change in value. In phases where a *periodic* parameter p does change in value, one can either refer p's value at the beginning (p_b) or end (p_e) of the phase. To do this in the implementation, one uses a Lisp symbol which has as its name the name of the parameter's Lisp symbol concatenated to `_b` or `_e` respectively. So if p is represented in the implementation by `p`, then p_b is represented by `p_b` and p_e by `p_e`. An *accumulating* parameter a can only have its change in value during a phase (a_c) referred to. Like with changing *periodic* parameters, this is done in the implementation by concatenation, but with `_c` in place of `_b` or `_e`. So if a becomes `a` for the implementation, a_c becomes `a_c`.

Now back to the discussion on the elements in the list of all the *conditions* and *parameter value changes* for a phase. With two exceptions, the notation for the *conditions* in a phase is the same as the notation for the *conditions* for constant parameters described earlier. The first exception is that instead of just being able to refer to *constant* parameters, one can refer to all parameters in the manner described in the previous paragraph. The second exception is that partial derivatives of *constant* parameters are excluded from these conditions.

Parameter value changes are given in the form `(:=` π $\mathcal{E}$`)`, which means the "variable" π has the value given by the expression $\mathcal{E}$. π can either refer to a *periodic* parameter's value at the end of a phase (like V_e given in Subsection D.2.1), or to an *accumulating* parameter's change in value during a phase (like W_c given in the same example). The expression $\mathcal{E}$ has the same form as the expressions in the *conditions* for constant parameters mentioned earlier with again the following two exceptions:

1. Instead of just being able to refer to *constant* parameters, one can refer to all parameters in the manner described two paragraphs ago.

2. Partial derivatives of *constant* parameters are excluded from these expressions.

[2]A value other than `nil` would activate an old style of input that is not documented in this appendix. Unfortunately, due to a lack of time, the steam engine example in Chapter 3 is at present only given in the old notation in the examples file. But one can decode it by looking at the normal ventricle example, which is given in both the new and old notations. Also, some documentation is given in Appendix E.1 in the code that starts with "`(defun new-phase-type`" (definition of the function `new-phase-type`) and the code that starts with "`(defflavor cycle`" (definition of the "flavor" or object type of `cycle`).

D.2.3 Arbitrary Functions

This subsection describes *constant* parameters that are "arbitrary" functions, and also, partial derivatives of these functions. Their description has been put off until now because the present implementation's notation for these functions is quite clumsy and complicated and so I wanted to avoid explaining them until after the rest of the input notation has been explained. This is why the example in Section D.2.1 is "simplified" by having the more specific forms of $Pi^{1/2}$ and Po^2 in place of the more "arbitrary" functions $Vd[Pi]$ and $Vs[Po, HR]$ actually used in the thesis examples in Chapter 3.

A major reason for the clumsiness and complexity is that in order to get the inequality reasoning system to work on these functions without re-implementing it, I represented these functions and their partial derivatives as Lisp symbols. Unfortunately this meant limitations to the notation in order keep it within (what I think are) the legal limits of a name for a symbol. No doubt this notation can be cleaned up quite a bit.

On with the explanation: The basic notation for an arbitrary function of the form $f[x1, \ldots, xn]$ is to replace the commas with dollar signs, so for example, $Vs[Po, HR]$ gets represented in the implementation as `Vs[Po$HR]`. The function name f cannot be a number or itself a function name. The arguments $x1$ through xn have to be either numbers or *constant* parameters. This means that if one wants to use the result of a "standard" function as an argument (for example, $a + b + c$ or $\log(q)$) then one has to put them into "arbitrary" function notation (`+[a$b$c]` and `log[q]`).

If the function f is the inverse of another function g, then $f[x1, \ldots, xn]$ can be denoted by `g^-1[x1$...$xn]` instead of `f[x1$...$xn]`. Do not use more than one level of inversion on a function.

To denote the partial derivative of a function, a series of `%`'s and `!`'s are added in front the notation for that function. The number of `%`'s between two `!`'s indicate which partial derivative (0th, 1st, 2nd, etc.) of an argument is being taken, and the position of these `%`'s relative to the `!`'s indicate with respect to which argument this partial derivative is being taken. The matching process starts from the left and is in a left to right order, so that the `%`'s to the left of all the `!`'s refer to the leftmost argument, the `%`'s between the two leftmost `!`'s refer to the next argument, etc. Some examples:

1. $\partial f[a, b, c, d, e]/\partial a$ translates into `%f[a$b$c$d$e]`.

2. $\partial f[a, b, c, d, e]/\partial d$ translates into `!!!%f[a$b$c$d$e]`.

3. $\partial^3 f[a, b, c, d, e]/\partial a^3$ translates into `%%%f[a$b$c$d$e]`.

4. $\partial^2 f[a, b, c, d, e]/\partial c^2$ translates into `!!%%f[a$b$c$d$e]`.

5. $\partial^5 f[a, b, c, d, e]/(\partial a^3 \cdot \partial d^2)$ translates into `%%%!!!%%f[a$b$c$d$e]`.

6. $\partial^6 f[a, b, c, d, e]/(\partial a^3 \cdot \partial b \cdot \partial e^2)$ translates into `%%%!%!!!%%f[a$b$c$d$e]`.

D.3 Output

Once a `cycle` object (call it *cycle-obj*) has been created, the following function calls (implemented as methods for the `cycle` object type) tell AIS to analyze the iterative dynamic system at steady-state described by *cycle-obj*:

- `(give-phases` *cycle-obj*`)`
  This function call essentially tells AIS to repeat back what was given as input, but with the solved version of the equations.
- `(give-periodic-min/max` *cycle-obj*`)`
  This function call tells AIS to find the extreme values of the periodic parameters.
- `(give-rates` *cycle-obj print-options*`)`
  This function call tells AIS to derive the average rate of accumulation for each accumulating parameter, an upper and lower bound for each rate, each rate's derivatives with respect to each simple constant parameter (symbol), and how altering constant parameters that are functions would affect the rates. The *print-options* argument(s) is optional, and can have 0 or more of the following possible options:

  `:eq` print the equations of the derivatives

  `:curve-shapes` assuming smoothness, draw the possible general curve shapes of each rate versus each simple constant parameter (symbol)

  `:phases` determine the contributions of each phase to a value (making the determinations for the effects of increasing functions has not been implemented yet)

  Some possible versions of this function call are

  1. `(give-rates` *cycle-obj*`)`,
  2. `(give-rates` *cycle-obj* `:phases)`,
  3. `(give-rates` *cycle-obj* `:curve-shapes :eq)`, and
  4. `(give-rates` *cycle-obj* `:eq :phases :curve-shapes)`.
- `(give-ratios` *cycle-obj ratios-to-be-done*`)`
  This function call tells AIS to derive the ratio for the rate of accumulation between pairs of accumulating parameters, each ratio's derivatives with respect to each simple constant parameter (symbol), and how altering constant parameters that are functions would affect the ratios. The *ratios-to-be-done* argument is optional. If it is not given or is not a list, all the ratios are given. If it is a list, it should be a list of accumulating parameters. The function will then give the ratio of the 1st accumulating parameter to the 2nd, 3rd to the 4th, etc.

Shortcomings of the present output are mentioned in Chapter 3 (AIS examples).

Appendix E

AIS Source Code

This appendix contains the source files for the implementation of AIS, the program that analyzes iterative dynamic systems. AIS has been implemented on Symbolics[1] workstations (Lisp machines) using the Genera 7 version of the Symbolics Common Lisp programming language. The parts of this language used are essentially Common Lisp [46] with the following additions: the LOOP iteration macro, the *flavors* object oriented programming system, some graphics commands, and some system development commands. Actually setting up the system and running it is described in Appendix D.1.

The first three sections of this appendix list the main parts (files) of AIS. Following this is a section with documentation on the inequality reasoning system used by AIS. Afterwards is a short section with a listing of a file that provides some patches for that inequality reasoning system. The last section lists the file with the examples run on AIS including the examples described in this thesis. Except for some reformatting to fit the margins and corrections to spelling errors in the comments, the listings in this appendix are as they appear in the corresponding files mentioned in Appendix D.1. Unless otherwise noted, the files are on the Zermatt file server at the MIT Laboratory for Computer Science under the directory path of `Z:>ay>`.

E.1 File Cycle.lisp

This is the main file of the AIS implementation.

```
;;; -*- Mode: LISP; Package: CYCLE -*-
;uses Elisha Sack's inequality reasoning system (loaded by z:>elisha>qm>system.lisp
;into package QM, formly used system CMS in package QM in Z:>elisha>oqm directory).
;When using this system, use PROGV to temporarily dynamically bind whatever context
;is being used to QM::*CONTEXT* because operations like QM::+ and QM::* will make
;calls to PARITY that assume the context is QM::*CONTEXT* regardless of which
;context you are actually using.

(print "LOAD Z:>AY>QMFIX TO FIX SOME INEQUALITY REASONER BUGS.")
```

[1]Symbolics, Inc., 8 New England Executive Park, Burlington, MA 01803.

```
;Would like to be able to use qm::simplify in places before expressions are
;converted to CMS form (in EVAL-IN-CMS)

;Can handle functions applied to other functions.
;Notation: F[A] = 'F' applied to 'A'; %F[A] = dF/dA; %%F[A] = d^2F/dA^2;
; G[X$Y$Z] = 'G' applied to 'X','Y','Z'; %%!%G[X$Y$Z] = d^3G/(dX^2*dY);
; !!%G[X$Y$Z]=dG/dZ; etc.
; Do not put in redundant !'s. Need at least 1 argument.
; A,X,Y,Z can be numbers or names of symbols.
;Can handle functions applied to other functions (but recursive levels may be
; unanalyzable by CYCLE-ANALYZE functions). Except at top level, functions can be
; ordinary ones (+, -, *, etc.).
;If not an ordinary function, can invert it: F^-1[A] = inverse of 'F' applied to
; 'A'. Do not use more than 1 level of inversion on a function.

;Also: variable_X, where X is C - change, B - beginning, E - end

;The implementation of the 'F[A]' stuff is more or less just limited to what is
;needed to make the heart ventricle example work: F[constants] in the constants
;list, and bounds (conditions) on F[constants] and its (partial) derivatives.

;phase-type, CYCLE vs. constant, PERIODIC, accumulating

;In the expressions in the phase-type objects, constant periodic variables can be
;referred to by using their symbol, varying periodic variables can have their
;values at either the beginning or end of the phase refered to by using their
;symbol with a "_b" or "_e" respectively appended to the end, constant
;accumulating variables canNOT be referred to, and varying accumulating variables
;can have the amount they changed by (during the phase) refered to by using their
;symbol with a "_c" appended to the end.
(defstruct phase-type
  name
  ;all below are lists
  condition ;list of 2 argument inequalities between phase parameter expressions
            ;that have to be true for this phase-type (put equalities with 'find'
            ;variables)
  constant ;list of symbols that are constants
  constant-periodic ;list of symbols that are periodic variables that stay
                    ;constant in value during the phase
  varying-periodic-given ;list of symbols that are periodic variables which may
                         ;change during the phase and whose end value in the phase
                         ;is given explicitly by the cycle description as a
                         ;constant
  varying-periodic-find ;List of periodic variables which may change during the
                      ;phase. List in the form of 2 element lists. Each 2 element
                      ;list has in order 1) the symbol for the periodic variable
                      ; and 2) an expression in terms the phase parameters that
                      ;gives the end value of the periodic variable in 1)
  ;the 3 slots below are the accumulating variable versions of the periodic
  ;variable slots above
  constant-accum
  varying-accum-given
  varying-accum-find ;the 2nd elements in the 2 element lists give the amount to
                     ;added (during the phase) to the accumulating variable rather
                    ;than the variable's end value
  )

;key is name, value is a phase-type object
(defvar *phase-type-ht* (make-hash-table))

;Does not make all the needed input checks. The input parameters are as described
;above for the phase-type object
(defun new-phase-type
       (&key name condition constant constant-periodic varying-periodic-given
        varying-periodic-find constant-accum varying-accum-given varying-accum-find
        &aux
        (phase-type-obj
          (make-phase-type
            :name name :condition condition :constant constant
            :constant-periodic constant-periodic
            :varying-periodic-given varying-periodic-given
            :varying-periodic-find varying-periodic-find
            :constant-accum constant-accum :varying-accum-given varying-accum-given
            :varying-accum-find varying-accum-find))
        (solved-symbols (make-hash-table)) ;key is symbol, value is 't
        ;key is a symbol, value is a list of the unsolved symbols in the expression
        ;equal to the symbol in the key
        (unsolved-symbols (make-hash-table))
        )
  ;see what's given and unsolved initially
  (mapcar #'(lambda (parm) (setf (gethash parm solved-symbols) t)) constant)
  (mapcar #'(lambda (parm) (setf (gethash parm solved-symbols) t))
          constant-periodic)
  (mapcar
    #'(lambda (parm &aux (name (symbol-name parm)))
        (setf (gethash (intern (concatenate 'string name '"_B")) solved-symbols) t)
        (setf (gethash (intern (concatenate 'string name '"_E")) solved-symbols) t))
    varying-periodic-given)
  (loop for (parm end-value-expr) in varying-periodic-find
        for name = (symbol-name parm)
        do (setf
             (gethash (intern (concatenate 'string name '"_B")) solved-symbols) t)
           (setf
             (gethash (intern (concatenate 'string name '"_E")) unsolved-symbols)
             (depends end-value-expr)))
  (mapcar #'(lambda (parm)
              (setf (gethash
                      (intern (concatenate 'string (symbol-name parm) '"_C"))
                      solved-symbols) t))
          varying-accum-given)
  (loop for (parm phase-value-addition) in varying-accum-find
        for name = (symbol-name parm)
     do (setf (gethash (intern (concatenate 'string name '"_C")) unsolved-symbols)
              (depends phase-value-addition)))

  ;See if the unsolved can be solved: Do type 1 substitutions for the expressions
  ;giving the values of "find" parameters and type 2 substitutions whenever
  ;possible.
  (loop for no-newly-solved-var? = t
        do (maphash
             #'(lambda (parm list)
                 (if (loop for still-unsolved-dependent-parm in list
                           always (gethash still-unsolved-dependent-parm
                                           solved-symbols))
                     (progn (remhash parm unsolved-symbols)
                            (setf (gethash parm solved-symbols) t)
                            (setq no-newly-solved-var? nil))))
                 unsolved-symbols)
        until no-newly-solved-var?)
  (let ((unsolved-vars nil))     ;list of the symbols still yet to solved
    (maphash #'(lambda (var list) (push var unsolved-vars)) unsolved-symbols)
    (if (not (null unsolved-vars))
        (error "New-phase-type: the phase type ~a has the following unsolvable ~
 parameters: ~a"
               name unsolved-vars)
        (setf (gethash name *phase-type-ht*) phase-type-obj)))

  ;make sure that the conditions have correct form and only proper solved-for
  ;phase parameters
  (mapcar #'(lambda (cond &aux (cond-vars (depends cond)))
              (if (not (and (= (list-length cond) 3.)
                            (member (first cond) '(< <= >= >))))
                  (error "New-phase-type: the phase type ~a condition ~a is ~
 of the wrong form." name cond))
              (mapcar #'(lambda (var)
                          (if (not (gethash var solved-symbols))
                              (error "New-phase-type: the phase type ~a ~
condition ~a has the unknown or unsolved variable ~a." name cond var)))
                      cond-vars))
          condition))

;creating cycles

(defflavor
  cycle (name
         rate ;symbol or number for the number of cycles per time unit, assumed
              ;to be >0
         constants ;list of symbols and numbers that are cycle's constants
                   ;(numbers do not have to be listed)
         periodic-vars ;list of symbols that are the cycle's periodic variables
         accum-vars ;list of symbols that are the cycle's accumulating variables
         conditions ;for now, list of 2 argument inequalities between expressions
                    ;with the constants and/or rate as arguments
         phases ;list of the following (one for each phase in the cycle), in the
                ;order that the phases occur.

                ;When the phase-type is NOT nil:
                ;The parameters in capitals are used when the periodic or accum
                ;parameter involved is 'given'.:
                ; (name phase-type
                ;  ((pt-constant-parm cycle-constant-parm) ...
                ;   (pt-periodic-parm cycle-periodic-parm CYCLE-CONSTANT-PARM)...
        ;   (pt-accum-parm cycle-accum-parm CYCLE-CONSTANT-PARM)...))
        ;For each correspondence between a phase-type (pt) and cycle
        ;parameter there is a 2 or 3 element list to give the
        ;correspondence. The 3rd element exists when a parameter is
        ;"given". This 3rd element is a cycle constant which gives the
        ;needed value.

        ;When the phase-type is NIL, the phase conditions & change
        ;equations are given directly. For each condition & change
        ;equation there is 3 element list (use _b and _e notation for
        ;varying periodic parms, _c notation for accumulating parms):
        ;   (name NIL
        ;    ((:= parm expression) ...    ;change equation for PARM
        ;    ((< expr1 expr2) ...))       ;phase condition ("<" can also be
        ;                                 ; ">", "<=", ">=")

    ;The slots below are not given by the user, the slots are internal and
    ;initialized automatically at object creation time. Each slot is an array
    ;of objects, one object per phase, in the same order as the phases given
    ;by the PHASES slot above.
    corresponding-parms ;list of association lists. Each element of the
                        ;alists marks a correspondence between a phase-type
                        ;parameter (car) and a cycle parameter (cdr) for the
                        ;phase. For varying periodic variables _b and _e
                        ;notation is used, as is _c notation for varying
                        ;accumulating variables.
    solved-parms ;list of association lists. Each element of the alists gives
                 ;the value (cdr) of a cycle periodic or accumulation
                 ;variable (car). The value is an expression of 1 or more
                 ;cycle constants. _b _e _c notation used.
    unsolved-parms ;list of hash tables containing cycle periodic or
                   ;accumulation parameters whose value is unknown. In the
                   ;tables: key is the parameter, value is information needed
                   ;to find the actual value of that parameter (see method
                   ;below that sets up tables and lists). _b _e _c notation
                   ;used.

    ;Note that cycle constants are NOT in the 'key' positions of either
    ;solved-parms or unsolved-parms.

    ;below are internal variables (also automatically initialized at creation
    ;time) which correspond to the cycle as a whole
    accum-per-cycle ;association list of accumulating variables (keys) and
                    ;the amount added to them per cycle
    inequality-reasoner-storage ;storage for a context object from Elisha
                                ;Sack's CMS inequality reasoner
    found-inconsistency? ;true if Elisha Sack's CMS found an inconsistency in
                         ;the conditions (not all conditions may have been
                         ;entered, the system stops after finding the first
                         ;inconsistency). false if not (all conditions will
                         ;be entered).
    )
      ()
(:initable-instance-variables
   name rate constants periodic-vars accum-vars conditions phases)
  (:init-keywords
   :name :rate :constants :periodic-vars :accum-vars :conditions :phases)
  (:required-init-keywords
   :name :rate :constants :periodic-vars :accum-vars :conditions :phases)
  )

;Set-up tables & lists, invoked on creating an instance of a cycle.
;Two levels of substitutions are done:
; 1) the parameters in the phase-type objects are substituted with corresponding
;    ones in the cycle description (correspondences in corresponding-parms), then
; 2) the parameters in the cycle description are substituted with expressions
;    using constants of the cycle (solutions in solved-parm-alists).
;Unfortunately, some 2) substitutions are done before other 1) substitutions
(defmethod (make-instance cycle)
           (&key name rate constants periodic-vars accum-vars conditions phases)
  (create-tables&lists self)
  (find-pt&cycle-parm-correspondences-&-solve-givens self)
  (solve-rest-of-periodic&accum-parms self)
  (warn-of-unsolved-parms self)
  (find-accums-per-cycle self)
  (check&install-conditions self))

;The rest of the file contains INITIALIZATION methods for CYCLE

(defmethod (create-tables&lists cycle) ()
   (setq corresponding-parms (map 'array #'(lambda (phase) nil) phases))
   (setq solved-parms (map 'array #'(lambda (phase) nil) phases))
   (setq unsolved-parms (map 'array #'(lambda (phase) (make-hash-table)) phases))
   (setq accum-per-cycle nil)
   ;in oQM, used (qm::new-context)
   (setq inequality-reasoner-storage (qm::make-context))
   (setq found-inconsistency? nil))


;Do some preliminary checks, get correspondences between phase-type parameters and
;cycle parameters for type (1) substitutions, mark "given" parameters as solved
;(for type 2 substitutions) with their end (periodic) or phase (accumulating)
;values, and mark the other non-constant parameters as unsolved. These unsolved
;parameters are either marked with 'preceding-end' or a cons with
;'no-correspondences-yet' in the cdr. 'preceding-end' means that a periodic
;variable takes the same value as the end value of the same periodic variable in
;the immediate preceding phase. 'no-correspondences-yet' means that the formula
;for a periodic or accumulating variable's value is still in terms of the
;phase-type parameters and not the cycle parameters.)
; For phase-type = NIL, no phase type is given, instead the "correspondence-lists"
;contain the conditions & change equations directly: the type (1) substitutions
;are already "done".
(defmethod (find-pt&cycle-parm-correspondences-&-solve-givens cycle) ()
  (loop for i from 0 ;ith phase
        for (phase-name phase-type correspondence-lists) in phases
        for phase-type-obj = (if phase-type (gethash phase-type *phase-type-ht*))
        for unsolved-parm-ht being the array-elements of unsolved-parms
        do
```

```
    (if (not phase-type)
        (let ;phase type not given, "correspondences" are the actual phase
             ;conditions and change equations
          ;until given a change equation for a periodic parm,
          ;assume that it is unchanged
          ((unchanged-periodic-parms (copy-list periodic-vars)))
          (loop for expression in correspondence-lists
                for (operator left right) = expression do
            (if (not (= (list-length expression) 3.))
                (error "In cycle ~a's phase ~a, the expression ~a is illegal"
                       name i expression))
            ;make sure symbols in EXPRESSION are parameters in the cycle
            (loop for parm in (depends expression)
                  for parm-string = (symbol-name parm)
                  for string-length = (length parm-string) do
              (if (and (> string-length 2.)
                       (eql (char parm-string (- string-length 2.)) '#\_))
                  (let* ((base-string (subseq parm-string 0 (- string-length 2.)))
                         (base-parm-symbol (intern base-string))
                         (last-char (char parm-string (- string-length 1.))))
                    (case last-char
                      (#\C (if (not (member base-parm-symbol accum-vars))
                               (error "In cycle ~a's phase ~a, the symbol ~a is ~
not a cycle accumulating parameter" name i parm)))
                      ((#\B #\E) (if (not (member base-parm-symbol periodic-vars))
                                     (error "In cycle ~a's phase ~a, the symbol ~
~a is not a cycle periodic parameter" name i parm)))
                      (otherwise (if (not (or (eq parm rate)
                                              (member parm constants)
                                              (member parm periodic-vars)))
                                     (error "In cycle ~a's phase ~a, the ~
parameter ~a is neither a cycle constant or periodic parameter" name i parm)))))
                  (if (not (or (eq parm rate) (member parm constants)
                               (member parm periodic-vars)))
                      (error "In cycle ~a's phase ~a, the parameter ~a is ~
neither a cycle constant or periodic parameter" name i parm))))

            (if (eq operator ':=) ;a change equation, process it
                (let* ((parm-string (symbol-name left))
                       (string-length (length parm-string))
                       (base-string (subseq parm-string 0 (- string-length 2.)))
                       (base-parm-symbol (intern base-string))
                       (last-char (char parm-string (- string-length 1.))))
                  (cond
                    ((or (not (eql (char parm-string (- string-length 2.)) '#\_))
                         (not (member last-char '(#\E #\C) :test #'eql)))
                     (error "In cycle ~a's phase ~a, the parameter ~a is illegal"
                            name i left))
                    ((eql last-char '#\C)
                     (if (not (member base-parm-symbol accum-vars))
                         (error "In cycle ~a's phase ~a, the symbol ~a is not a ~
cycle accumulating parameter" name i left))
                     (setf (gethash left unsolved-parm-ht)
                           (cons right (depends right))))
                  (t                              ;last character is a #\E
                   (if (not (member base-parm-symbol periodic-vars))
                       (error "In cycle ~a's phase ~a, the symbol ~a is not a ~
cycle periodic parameter" name i left))
                   (setq unchanged-periodic-parms  ;changing periodic parameter
                         (delete base-parm-symbol unchanged-periodic-parms))
                   (setf (gethash left unsolved-parm-ht)
                         (cons right (depends right)))
                   (setf (gethash (intern
                                    (concatenate 'string base-string '"_B"))
                                  unsolved-parm-ht) 'preceding-end))))))
          (mapcar #'(lambda  (periodic-parm)
                      (setf (gethash periodic-parm unsolved-parm-ht)
                            'preceding-end))
                  unchanged-periodic-parms))

        (progn  ;phase type given
          (if (not phase-type-obj)
              (error "In cycle ~a's phase ~a, the phase-type ~a does not exist."
                     name i phase-type))
          (flet ((check (pt-parm appropriate-cycle-parms desired-cycle-parm-type
                                 &aux
                                 (correspond? (assoc pt-parm correspondence-lists))
                                 (cycle-parm (second correspond?)))
                   (cond
                     ((not correspond?)
                      (format t "~%Warning: In cycle ~a's phase ~a, the phase type~
 parameter ~a has no corresponding cycle parameter." name i pt-parm))
                     ((not (or (member cycle-parm appropriate-cycle-parms)
                               (and (or (numberp cycle-parm) (eq cycle-parm rate))
                                    (eq desired-cycle-parm-type 'constant))))
                      (error "In cycle ~a's phase ~a, the symbol ~a is not a ~
cycle-~a." name i cycle-parm desired-cycle-parm-type)))
                   correspond?))
            (mapcar
              #'(lambda (pt-parm &aux
                          (pt-cycle-pair (check pt-parm constants 'constant)))
                  (if pt-cycle-pair
                      (push (cons pt-parm (second pt-cycle-pair))
                            (aref corresponding-parms i))))
              (phase-type-constant phase-type-obj))
            (mapcar
              #'(lambda (pt-parm &aux
                          (pt-cycle-pair
                            (check pt-parm periodic-vars 'periodic-var)))
                  (if pt-cycle-pair
                      (let ((cycle-parm (second pt-cycle-pair)))
                        (push (cons pt-parm cycle-parm)
                              (aref corresponding-parms i))
                        (setf (gethash cycle-parm unsolved-parm-ht)
                              'preceding-end))))
              (phase-type-constant-periodic phase-type-obj))
            (mapcar
              #'(lambda (pt-parm &aux
                (pt-cycle-value-triple
                  (check pt-parm periodic-vars 'periodic-var)))
            (if pt-cycle-value-triple
                (let* ((cycle-parm (second pt-cycle-value-triple))
                       (value (third pt-cycle-value-triple))
                       (pt-parm-name (symbol-name pt-parm))
                       (cycle-parm-name (symbol-name cycle-parm))
                       (cycle-parm-b
                         (intern
                           (concatenate 'string cycle-parm-name '"_B")))
                       (cycle-parm-e
                         (intern
                           (concatenate 'string cycle-parm-name '"_E"))))
                  (if (and (not (numberp value)) (not (eq value rate))
                           (not (member value constants)))
                      (error "In cycle ~a's phase ~a, the symbol ~a is not ~
a cycle-constant." name i value))
                  (push (cons (intern
                                (concatenate 'string pt-parm-name '"_B"))
                              cycle-parm-b)
                        (aref corresponding-parms i))
                  (push (cons (intern
                                (concatenate 'string pt-parm-name '"_E"))
                              cycle-parm-e)
                        (aref corresponding-parms i))
                  (setf (gethash cycle-parm-b unsolved-parm-ht)
                        'preceding-end)
                  (push (cons cycle-parm-e value) (aref solved-parms i)))))
         (phase-type-varying-periodic-given phase-type-obj))
       (mapcar
         #'(lambda (pt-parm-expr-pair &aux (pt-parm (first pt-parm-expr-pair))
                   (end-value-expr (second pt-parm-expr-pair))
                   (pt-cycle-pair
                     (check pt-parm periodic-vars 'periodic-var)))
            (if pt-cycle-pair
                (let* ((cycle-parm (second pt-cycle-pair))
                       (pt-parm-name (symbol-name pt-parm))
                       (cycle-parm-name (symbol-name cycle-parm))
                       (cycle-parm-b
                         (intern
                           (concatenate 'string cycle-parm-name '"_B")))
                       (cycle-parm-e
                         (intern
                           (concatenate 'string cycle-parm-name '"_E"))))
                  (push (cons (intern
                                (concatenate 'string pt-parm-name '"_B"))
                              cycle-parm-b)
                        (aref corresponding-parms i))
                  (push (cons (intern
                                (concatenate 'string pt-parm-name '"_E"))
                              cycle-parm-e)
                        (aref corresponding-parms i))
                  (setf (gethash cycle-parm-b unsolved-parm-ht)
                        'preceding-end)
```

```
                         (setf (gethash cycle-parm-e unsolved-parm-ht)
                               (cons end-value-expr 'no-correspondences-yet)))))
               (phase-type-varying-periodic-find phase-type-obj))
             ;do not record correspondences because constant accumulating variables
             ;should not be in any formula
             (mapcar #'(lambda (pt-parm) (check pt-parm accum-vars 'accum-var))
                     (phase-type-constant-accum phase-type-obj))
             (mapcar
               #'(lambda (pt-parm &aux (pt-cycle-value-triple
                                         (check pt-parm accum-vars 'accum-var)))
                   (if pt-cycle-value-triple
                       (let* ((cycle-parm (second pt-cycle-value-triple))
                              (value (third pt-cycle-value-triple))
                              (cycle-parm-p
                                (intern (concatenate
                                          'string
                                          (symbol-name cycle-parm) '"_C"))))
                         (if (and (not (numberp value)) (not (eq value rate))
                                  (not (member value constants)))
                             (error "In cycle ~a's phase ~a, the symbol ~a is not ~
a cycle-constant." name i value))
                         (push (cons (intern (concatenate
                                               'string (symbol-name pt-parm) '"_C"))
                                     cycle-parm-p)
                               (aref corresponding-parms i))
                         (push (cons cycle-parm-p value) (aref solved-parms i)))))
               (phase-type-varying-accum-given phase-type-obj))
             (mapcar
               #'(lambda (pt-parm-expr-pair &aux (pt-parm (first pt-parm-expr-pair))
                           (end-value-expr (second pt-parm-expr-pair))
                           (pt-cycle-pair (check pt-parm accum-vars 'accum-var)))
                   (if pt-cycle-pair
                       (let* ((cycle-parm (second pt-cycle-pair))
                              (cycle-parm-p
                                (intern (concatenate
                                          'string (symbol-name cycle-parm) '"_C"))))
                         (push (cons (intern (concatenate
                                               'string (symbol-name pt-parm) '"_C"))
                                     cycle-parm-p)
                               (aref corresponding-parms i))
                         (setf (gethash cycle-parm-p unsolved-parm-ht)
                               (cons end-value-expr 'no-correspondences-yet)))))
               (phase-type-varying-accum-find phase-type-obj)))))))


;See which unsolved periodic & accumulating cycle variables can be solved.
(defmethod (solve-rest-of-periodic&accum-parms cycle) ()
  (loop for cycle-no-newly-solved-var? = t do
    (loop for i from 0  ;ith phase
          for correspondence-alist being the array-elements of corresponding-parms
          for solved-parm-alist being the array-elements of solved-parms
          for unsolved-parm-ht being the array-elements of unsolved-parms
          do
      (loop for phase-no-newly-solved-var? = t do
      (maphash
        #'(lambda (parm where-to-find-value)
            (typecase where-to-find-value
              ;periodic variable, look in the phase given by the number to see
              ;if the end value is solved for
              (number (let* ((parm-looking-for (end-periodic-parameter-of parm)))
                        (if (not (gethash
                                   parm-looking-for
                                   (aref unsolved-parms where-to-find-value)))
                            (let ((value (cdr (assoc
                                               parm-looking-for
                                               (aref solved-parms
                                                     where-to-find-value)))))
                              (remhash parm unsolved-parm-ht)
                              (push (cons parm value) (aref solved-parms i))
                              (setq phase-no-newly-solved-var? nil)
                              ;update current pointer to alist
                              (setq solved-parm-alist (aref solved-parms i))))))
              ;periodic variable, WHERE-TO-FIND-VALUE should be PRECEDING-END.
              ;look for the most recent preceding phase where an end value for
              ;that variable is given
              (symbol (let* ((parm-looking-for (end-periodic-parameter-of parm))
                             (num-phases (length phases)))
                        (loop for phase first (mod (- i 1.) num-phases)
                                        then (mod (- phase 1.) num-phases)
                              repeat num-phases
                              do
                          (if (gethash parm-looking-for
                                       (aref unsolved-parms phase))
                              ;found the preceding phase
                              (progn (setf (gethash parm unsolved-parm-ht) phase)
                                     (return t)))
                          (let* ((parm-value
                                   (assoc parm-looking-for
                                          (aref solved-parms phase)))
                                 (value (cdr parm-value)))
                            (if parm-value
                                ;found the preceding phase & its been solved for
                                (progn
                                  (remhash parm unsolved-parm-ht)
                                  (push (cons parm value) (aref solved-parms i))
                                  (setq phase-no-newly-solved-var? nil)
                                  ;update current pointer to alist
                                  (setq solved-parm-alist (aref solved-parms i))
                                  (return t))))

                             finally ;did not find an end value for the
                                     ;periodic variable
                               (error "In cycle ~a's phase ~a, the periodic ~
variable ~a has no value set for it at any time." name i parm))))
              ;a periodic or accumulating variable where the value is an
              ;expression is in terms of other parameters in the phase
              (cons
                (cond
                          ((eq (cdr where-to-find-value) 'no-correspondences-yet)
                           ;have yet to find the correspondences between the phase-type
                           ;and cycle parameters. Do so.
                           (let* ((expr-in-pt-parms (car where-to-find-value))
                                  (needed-pt-parms (depends expr-in-pt-parms)))
                             (if (loop for pt-parm in needed-pt-parms
                                       thereis
                                         (not (assoc pt-parm correspondence-alist)))
                                 ;Missing an cycle parm to correspond to a needed
                                 ;phase-type parm. Mark as unsolvable.
                                 (setf (cdr where-to-find-value)
                                       'missing-correspondences)
                                 ;potentially solvable, set-up expression with cycle
                                 ;parms substituting for phase-type parms (type 1
                                 ;substitution)
                                 (let* ((expr-in-cycle-parms
                                          (simplify (sublis correspondence-alist
                                                            expr-in-pt-parms)))
                                         (needed-cycle-parms
                                          (depends expr-in-cycle-parms)))
                                   (setf (car where-to-find-value) expr-in-cycle-parms)
                                   ;list of dependent variables
                                   (setf (cdr where-to-find-value) needed-cycle-parms)
                                   ;Have not really solved anything, but set flag to make
                                   ;sure the expression will be examined again with cycle
                                   ;parameters in
                                   (setq phase-no-newly-solved-var? nil)))))
                          ((listp (cdr where-to-find-value)) ;list of dependent variables
                           (if (loop for cycle-parm in (cdr where-to-find-value)
                                     always (not (gethash cycle-parm unsolved-parm-ht)))
                                                      ;solvable, so go solve
                               (let ((value (simplify
                                              (sublis solved-parm-alist
                                                      (car where-to-find-value)))))
                                 (remhash parm unsolved-parm-ht)
                                 (if (not (loop for symbol in (depends value)
                                                always (or (eq symbol rate)
                                                           (member symbol constants))))
                                     (error "In cycle ~a's phase ~a, the solution of ~a ~
for a has some non-constant parameters." name i value parm))
                                 (push (cons parm value) (aref solved-parms i))
                                 (setq phase-no-newly-solved-var? nil)
                                 ;update current pointer to alist
                                 (setq solved-parm-alist (aref solved-parms i)))))))))
             unsolved-parm-ht)
           (if (not phase-no-newly-solved-var?) (setq cycle-no-newly-solved-var? nil))

              until phase-no-newly-solved-var?))
          until cycle-no-newly-solved-var?))

(defun end-periodic-parameter-of
       (parm &aux (parm-name (symbol-name parm)) (name-length (length parm-name))
        (name-looking-for
          (if (and (> name-length 2.)
                  (eql (char parm-name (- name-length 2.)) '#\_))
             (let ((new-name (copy-seq parm-name)))
               (setf (char new-name (- name-length 1.)) '#\E) ;of the form xxxx_B
               new-name)
             (concatenate 'string parm-name '"_E")))) ;of the form xxxx
  (intern name-looking-for))

;check and warn for any unsolved cycle parameters
(defmethod (warn-of-unsolved-parms cycle) ()
  (loop for i from 0 ;phase number
        for unsolved-parm-ht being the array-elements of unsolved-parms do
    (maphash #'(lambda (parameter other)
                 (format t "~%Warning: the parameter ~a in cycle ~a's phase ~a ~
is not solved for." parameter name i))
             unsolved-parm-ht)))


;find out how much accumulating variables gain each cycle
(defmethod (find-accums-per-cycle cycle) ()
  (mapcar
    #'(lambda (var &aux
                (var_c (intern (concatenate 'string (symbol-name var) '"_C")))
                (no-unsolved-found-yet t))
        ;list of additions to the accumulating var from each phase
        (let ((list-of-partial-sums
                (loop for solved-parm-alist
                          being the array-elements of solved-parms
                      for unsolved-parm-ht
                          being the array-elements of unsolved-parms
                      for pair? = (assoc var_c solved-parm-alist)
                      when pair? collect (cdr pair?)
                      else when (gethash var_c unsolved-parm-ht)
                             do (setq no-unsolved-found-yet nil) (return nil))))
          ;only record the accum variable if its accumulation is solved for in
          ;each phase
          (if no-unsolved-found-yet
              (push
                (cons
                  var
                  (cond ((null list-of-partial-sums) 0.) ;no partial sums,
                                                          ;accumulate nothing
                        ((null (rest list-of-partial-sums))     ;1 partial sum
                         (simplify (first list-of-partial-sums)))
                        ;>1 partial sums
                        (t (simplify (cons '+ list-of-partial-sums)))))
                accum-per-cycle))))
    accum-vars))


;check cycle conditions & install all conditions into inequality reasoning object
(defmethod (check&install-conditions cycle)
           (&aux (qm-assert-functions
                   '((< . ,#'qm::assert<)(<= . ,#'qm::assert<=)
                     (>= . ,#'qm::assert>=)(> . ,#'qm::assert>))))
  ;make context object for cycle the current context object while performing
  ;assertions
  (progv '(qm::*context*) (list inequality-reasoner-storage)
    (block found-inconsistency
    (qm::assert< 0 rate) ;install 0<rate
    ;examine & place cycle constraints
    (mapcar
      #'(lambda (cond &aux (type (first cond)) (vars (depends cond)))
          (if (not (and (= (list-length cond) 3.) (member type '(< <= >= >))))
              (error "Cycle ~a's condition ~a has the wrong form." name cond))
          (loop for var in vars
                when (not (or (eq var rate) (member var constants)
                              ;also okay if some derivative of applying a
                              ;function to a constant (this application is also
                              ;declared a constant
                              (if (applying-a-function? var)
                                  (let ((fcn[arg]-wo-derivs
                                          (string-left-trim '(#\% #\!)
                                                            (symbol-name var))))
                                    (member (intern fcn[arg]-wo-derivs)
                                            (cons rate constants))))))
                  do (error "Variable ~a in cycle ~a's condition ~a is not a ~
known cycle constant or a derivative of applying a function to a constant."
                            var name cond))
          (if (not (funcall
                     (cdr (assoc type qm-assert-functions))
                     (eval-in-cms (second cond))
                     (eval-in-cms (third cond))))
              (progn (format t "~%Found a contradiction in conditions. ~
Inequality reasoner storage:")
                     (describe inequality-reasoner-storage)
                     (cerror "Mark as inconsistent, ignore rest of conditions, ~
& return cycle"
                             "Adding cycle ~a's condition ~a revealed a ~
contradiction. The inequality reasoner storage so far is given above." name cond)
                     (setq found-inconsistency? t)
                     (return-from found-inconsistency nil))))

      conditions)

    ;for each phase, install its constraints
    (loop for i from 0
          for (p-name p-type rest) in phases
          for conditions-w/pt-parameters
              = (if p-type (phase-type-condition (gethash p-type *phase-type-ht*)))
          for correspondence-alist being the array-elements of corresponding-parms
          for solved-parm-alist being the array-elements of solved-parms
          for unsolved-parm-ht being the array-elements of unsolved-parms do
      (if (not p-type)
          (loop for cond-w/cycle-parms in rest ;no phase type, giving conditions &
                                                ;change equations directly
                for operator = (first cond-w/cycle-parms)
                unless (eq operator ':=)
                  do (if (not (assoc operator qm-assert-functions))
                                 (error "In cycle ~a's phase ~a, the ~a operator in ~a ~
 is illegal" name i operator cond-w/cycle-parms))
                              (let ((cycle-parms-needed (depends cond-w/cycle-parms)))
                               (if (loop for parm in cycle-parms-needed
                                         never (gethash parm unsolved-parm-ht))
                                   ;all the needed cycle parameters are either constants or
                                   ;have been solved, substitute periodic & accumulating
                                   ;variables with their values (type 2 substitution) &
                                   ;install the resulting condition
                                   (let ((cond-w/subs-done
                                           (sublis solved-parm-alist cond-w/cycle-parms)))
                                     (if (not (funcall
                                                (cdr (assoc (first cond-w/subs-done)
                                                            qm-assert-functions))
                                                (eval-in-cms (second cond-w/subs-done))
                                                (eval-in-cms (third cond-w/subs-done))))
                                         (progn (format t "~%Found a contradiction in ~
 conditions. Inequality reasoner storage:")
                                                (describe inequality-reasoner-storage)
                                                (cerror "Mark as inconsistent, ignore ~
 rest of conditions, & return cycle"
                                                        "In cycle ~a's phase ~a, adding ~
 condition ~a (which became ~a) revealed a contradiction. The inequality reasoner ~
 storage is given above." name i cond-w/cycle-parms cond-w/subs-done)
                                                (setq found-inconsistency? t)
                                                (return-from found-inconsistency nil)))))))

          (mapcar
            #'(lambda (cond-w/pt-parms &aux
                        (pt-parms-needed (depends cond-w/pt-parms)))
                (if (loop for parm in pt-parms-needed
                          always (assoc parm correspondence-alist))
                    ;all condition phase-type parameters have corresponding cycle
                    ;parameters, substitute the former with the latter (type 1
                    ;substitution)
                    (let* ((cond-w/cycle-parms
                             (sublis correspondence-alist cond-w/pt-parms))
                           (cycle-parms-needed (depends cond-w/cycle-parms)))
                      (if (loop for parm in cycle-parms-needed
                                never (gethash parm unsolved-parm-ht))
                          ;all the needed cycle parameters are either constants or
                          ;have been solved, substitute periodic & accumulating
                          ;variables with their values (type 2 substitution) &
                          ;install the resulting condition
                          (let ((cond-w/subs-done
                                  (sublis solved-parm-alist cond-w/cycle-parms)))
                            (if (not (funcall
                                       (cdr (assoc (first cond-w/subs-done)
                                                   qm-assert-functions))
                                       (eval-in-cms (second cond-w/subs-done))
                                       (eval-in-cms (third cond-w/subs-done))))
                                (progn (format t "~%Found a contradiction in ~
 conditions. Inequality reasoner storage:")
                                       (describe inequality-reasoner-storage)
```

```
                                            (cerror "Mark as inconsistent, ignore rest ~
of conditions, & return cycle"
                                                  "In cycle ~a's phase ~a, adding ~
condition ~a (which became ~a) revealed a contradiction. The inequality reasoner ~
storage is given above." name i cond-w/pt-parms cond-w/subs-done)
                                            (setq found-inconsistency? t)
                                            (return-from found-inconsistency nil)))))))
         conditions-w/pt-parameters))))))
```

E.2 File Cycle-Analyze.lisp

This file contains the functions to output analysis of an iterative dynamic system whose description has been entered and processed.

```
;;; -*- Mode: LISP; Package: CYCLE -*-
;Methods to analyze a cycle

;Can only analyze 1 level of applying functions to constants. For example, can
; analyze fcn[arg], but not fcn1[fcn2[arg]], etc.
;The function eps is reserved for internal use.
;If fcn[arg] satisfies a condition, then does fcn[arg]+eps[arg]? At the moment
; this is not forced to be necessarily true.

(defmethod (give-phases cycle) ()
  (if found-inconsistency?
      (print "Inconsistency found in conditions, not all listed conditions have
actually been entered."))
  (format t "~%CONDITIONS of the cycle ~a:~%(< 0 ~a)~{, ~a~}" name rate conditions)
  (format t "~%~%The PHASES:")
  (loop for (phase-name phase-type var-correspondences) in phases
        for conditions-w/pt-parameters
            = (if phase-type
                  (phase-type-condition (gethash phase-type *phase-type-ht*)))
        for correspondence-alist being the array-elements of corresponding-parms
        for solved-parms-for-phase being the array-elements of solved-parms
        for unsolved-parm-ht being the array-elements of unsolved-parms
        for index from 0
        for changed-periodic-alist = nil
        for changed-accum-alist = nil do

    ;pull out & sort changed periodic & accumulating variables for the phase
    (loop for (var . value) in solved-parms-for-phase
          for var-name = (symbol-name var)  for var-name-length = (length var-name)
          when (and (>= var-name-length 3.)
                    (eql (char var-name (- var-name-length 2.)) '#\_))
            do (cond ((eql (char var-name (- var-name-length 1.)) '#\E)
                      (push (cons (subseq var-name 0. (- var-name-length 2.))
                                  value)
                            changed-periodic-alist))
                     ((eql (char var-name (- var-name-length 1.)) '#\C)
                      (push (cons (subseq var-name 0. (- var-name-length 2.))
                                  value)
```

```
                              changed-accum-alist))))
(setq changed-periodic-alist
      (sort changed-periodic-alist #'string< :key #'car))
(setq changed-accum-alist (sort changed-accum-alist #'string< :key #'car))

;print out phase
(format t "~%~%Phase ~a: ~a~@[, a type of ~a~].~%  Added conditions:"
        index phase-name phase-type)
;print out added conditions
(if (not phase-type)
    ;no phase-type, VAR-CORRESPONDENCES are actually expressions
    (loop for expression in var-correspondences
          for operator = (first expression) do
      (if (not (eq operator ':=)) ;not a change equation, so is a condition
          (let ((cycle-parms-needed (depends expression)))
            (if (loop for parm in cycle-parms-needed
                      never (gethash parm unsolved-parm-ht))
                ;all the needed cycle parameters are either constants or have
                ;been solved, substitute periodic & accumulating variables
                ;with their values (type 2 substitution) & print the
                ;resulting condition
                (format t " ~a" (sublis solved-parms-for-phase expression))))))
    ;phase-type exists
    (mapcar
      #'(lambda (cond-w/pt-parms &aux
                 (pt-parms-needed (depends cond-w/pt-parms)))
          (if (loop for parm in pt-parms-needed
                    always (assoc parm correspondence-alist))
              ;all condition phase-type parameters have corresponding cycle
              ;parameters, substitute the former with the latter (type 1
              ;substitution)
              (let* ((cond-w/cycle-parms
                       (sublis correspondence-alist cond-w/pt-parms))
                     (cycle-parms-needed (depends cond-w/cycle-parms)))
                (if (loop for parm in cycle-parms-needed
                          never (gethash parm unsolved-parm-ht))
                    ;all the needed cycle parameters are either constants or
                    ;have been solved, substitute periodic & accumulating
                    ;variables with their values (type 2 substitution) & print
                    ;the resulting condition
                    (format t " ~a" (sublis solved-parms-for-phase
                                            cond-w/cycle-parms))))))
      conditions-w/pt-parameters))
(loop for (name . value) in changed-periodic-alist
      for s1 = (format nil "  ~a changed to" name)
      for s2 = (format nil " ~a." value)
      when (<= (+ (length s1) (length s2)) 105.) do (format t "~%~a~a" s1 s2)
      ;indent properly on output for value
      else do (format t "~%~a~%    ~a." s1 value))
(loop for (name . value) in changed-accum-alist
      for s1 = (format nil "  ~a increased by" name)
      for s2 = (format nil " ~a." value)
      when (<= (+ (length s1) (length s2)) 105.) do (format t "~%~a~a" s1 s2)
      ;indent properly on output for value
```

```
          else do (format t "~%~a~%    ~a." s1 value))
    ))

;find the minimum & maximum values of the periodic parameters
(defmethod (give-periodic-min/max cycle) ()
  (if found-inconsistency?
      (format t "~%An inconsistency was found in the cycle, the results below may ~
be erronous."))
  (format t "~2%For the cycle ~a:" name)
  (mapcar
    #'(lambda (periodic-parm &aux
                (symbol-for-end
                  (intern (concatenate 'string (symbol-name periodic-parm) '"_E")))
                ;lists for solved values
                (s-phase-num nil) (s-phase-name nil) (s-phase-end-value nil)
                ;lists of flags for solved values (<= >= are for things not found
                ;to be <, >, or =). For >it? nth element is T if some other value
                ;is found by Elisha Sacks' CMS to be > than the nth value, nil ow.
                ;Similarly for the other lists.
                >it? =it? <it? >=it? <=it?
                ;lists for unsolved values
                (u-phase-num nil) (u-phase-name nil))
        ;gather periodic-parm values from all the phases
        (loop for phase-num from 0
              for (phase-name . rest) in phases
              for phase-solved-parms being the array-elements of solved-parms
              for phase-unsolved-parms being the array-elements of unsolved-parms
              do
          (if (gethash symbol-for-end phase-unsolved-parms)
              (progn (push phase-num u-phase-num) (push phase-name u-phase-name))
              (let ((pair? (assoc symbol-for-end phase-solved-parms)))
                (if pair?
                    (progn (push phase-num s-phase-num)
                           (push phase-name s-phase-name)
                           (push (cdr pair?) s-phase-end-value)))))
          finally ;Create flag lists
            (let ((num-solved (length s-phase-num)))
              (setq >it? (make-list num-solved :initial-element nil))
              (setq =it? (make-list num-solved :initial-element nil))
              (setq <it? (make-list num-solved :initial-element nil))
              (setq >=it? (make-list num-solved :initial-element nil))
              (setq <=it? (make-list num-solved :initial-element nil))))

        ;Figure out who is greater than who & record in flag lists.
        ;Two values may not have any known relationship between them.
        (progv '(qm::*context*) (list inequality-reasoner-storage)
          (loop for values on s-phase-end-value
                for v1 = (eval-in-cms (first values))
                for >1l on >it?  for =1l on =it?  for <1l on <it?
                for <=1l on <=it?  for >=1l on >=it?  do
            (loop for v2-prelim in (rest values)
                  for v2 = (eval-in-cms v2-prelim)
                  for >2l on (rest >1l)  for =2l on (rest =1l)
                  for <2l on (rest <1l)
                  for <=21 on (rest <=11)  for >=21 on (rest >=11)  do
              (cond ((qm::> v2 v1) (setf (first >11) t) (setf (first <21) t))
                    ((qm::< v2 v1) (setf (first <11) t) (setf (first >21) t))
                    (t (let ((v2>=v1? (qm::>= v2 v1))  (v2<=v1? (qm::<= v2 v1)))
                         (cond ((and v2>=v1? v2<=v1?) (setf (first =11) t)
                                (setf (first =21) t))
                               (v2>=v1? (setf (first >=11) t)
                                        (setf (first <=21) t))
                               (v2<=v1? (setf (first <=11) t)
                                        (setf (first >=21) t)))))))))
        ;Print out
        (format t "~2%For the periodic parameter ~a:" periodic-parm)
        (loop with maybe-highest = nil and may-equal-highest = nil and
                maybe-lowest = nil and may-equal-lowest = nil
              for phase-num in s-phase-num        for phase-name in s-phase-name
              for end-value in s-phase-end-value
              for any-larger-values? in >it?
              for any-potentially-larger-values? in >=it?
              for any-smaller-values? in <it?
              for any-potentially-smaller-values? in <=it?  do
          (if (not any-larger-values?)
              (if any-potentially-larger-values?
                  (setq may-equal-highest
                        (list* end-value phase-name phase-num may-equal-highest))
                  (setq maybe-highest
                        (list* end-value phase-name phase-num maybe-highest))))
          (if (not any-smaller-values?)
              (if any-potentially-smaller-values?
                  (setq may-equal-lowest
                        (list* end-value phase-name phase-num may-equal-lowest))
                  (setq maybe-lowest
                        (list* end-value phase-name phase-num maybe-lowest))))
              finally
                (cond
                  ((null maybe-highest)
                   (format t "~2%   Error, no possible maximum value found."))
                  ((= 3. (list-length maybe-highest))
                   (format t "~2%   The maximum value is ~a, which first occurred ~
in phase ~a (~a)." (car maybe-highest) (cadr maybe-highest) (caddr maybe-highest)))
                  (t ;more than 1 value is possibly the maximum
                   (format t "~2%   The maximum value is one of the following: ~
~{ ~a in phase ~a (~a)~^,~}." maybe-highest)))
                (cond ((null may-equal-highest)) ;none, do nothing
                      ((= 3. (list-length may-equal-highest))
                       (format t "~%   The value ~a, which first occurred in ~
phase ~a (~a), may equal the maximum."
                               (car may-equal-highest) (cadr may-equal-highest)
                               (caddr may-equal-highest)))
                      (t ;more than 1 value may equal the maximum
                       (format t "~%   The the following values may equal the ~
maximum: ~{ ~a in phase ~a (~a)~^,~}." may-equal-highest)))
                (format t "~%")
                (cond
                  ((null maybe-lowest)
                    (format t "~%   Error, no possible minimum value found."))
                   ((= 3. (list-length maybe-lowest))
                    (format t "~%   The minimum value is ~a, which first occurred ~
in phase ~a (~a)." (car maybe-lowest) (cadr maybe-lowest) (caddr maybe-lowest)))
                   (t ;more than 1 value is possibly the minimum
                    (format t "~%   The minimum value is one of the following: ~
~{ ~a in phase ~a (~a)~^,~}." maybe-lowest)))
                 (cond ((null may-equal-lowest)) ;none, do nothing
                       ((= 3. (list-length may-equal-lowest))
                        (format t "~%   The value ~a, which first occurred in ~
phase ~a (~a), may equal the minimum."
                                (car may-equal-lowest) (cadr may-equal-lowest)
                                (caddr may-equal-lowest)))
                       (t ;more than 1 value may equal the minimum
                        (format t "~%   The the following values may equal the ~
minimum: ~{ ~a in phase ~a (~a)~^,~}." may-equal-lowest))))
        (if (not (null u-phase-num))
            (progn (format t "~2%   The following phases have unknown values:")
                   (loop for name in u-phase-name  for num in u-phase-num
                         for items-left on u-phase-name do
                     (if (rest items-left) ;not last item
                         (format t " ~a (~a)," name num)
                         (format t " and ~a (~a)." name num))))))

  periodic-vars))

;Look at the rate of accumulation for each accumulating variable, its derivatives
;w.r.t. each simple constant, and how altering functions affect the rate.
;Possible options include
; :eq -print the equation of the derivative
; :curve-shapes -draw the possible curve shapes of the rate vs. each constant
: :phases -try to figure out the contributions of each phase to a value
;          (not yet implemented for effects of increasing functions)
(defmethod
  (give-rates cycle)
  (&rest print-options &aux
    ;Hashtable of function applications in the expression of interest.
    ;Key is name of function (string),
    ;Value is a list of the names of arguments (strings) for this function.
    (fcn-ht (make-hash-table :test #'equal))
    (simple-constants    ;constants that are NOT function applications or numbers
      (loop with simple-constants = nil
            for constant in (cons rate constants) do   ;treat rate as a constant
        (if (not (numberp constant))
            (multiple-value-bind
              (fcn-name arg-string) (applying-a-function? constant)
              (if fcn-name
                  (push arg-string (gethash fcn-name fcn-ht));function application
                  (push constant simple-constants))))

            finally (return simple-constants)))
    (phase-names (loop for (phase-name . rest) in phases collect phase-name)))

  (progv '(qm::*context*) (list inequality-reasoner-storage)
    (format t "~%RATES of ACCUMULATION for the cycle ~a:" name)
    (if found-inconsistency?
        (print "Inconsistency found in conditions, ignore the bounds."))
    (loop for (accum-var . accum-expr) in accum-per-cycle
          for phase-contribution-symbol =
              (intern (concatenate 'string (symbol-name accum-var) '"_C"))
          for phase-contributions =
              (loop for phase-solved-parms being the array-elements of solved-parms
                    for phase-unsolved-parms
                        being the array-elements of unsolved-parms
                    collect
                      (if (gethash phase-contribution-symbol phase-unsolved-parms)
                          nil
                          (let ((pair? (assoc phase-contribution-symbol
                                              phase-solved-parms )))
                            (if pair? (make-product rate (cdr pair?)) 0.))))
          for rate-of-accum = (make-product rate accum-expr) do
      (analyze-expr rate-of-accum (format nil "Rate of accumulating ~a" accum-var)
                    '"Rate" simple-constants fcn-ht print-options
                    phase-contributions phase-names))))

;Look at the ratio for the rate of accumulation between pairs of accumulating
; variables, its derivatives w.r.t. each simple constant, and how altering
; functions affect the ratio.
;For the optional argument RATIOS-TO-BE-DONE, if it is not a list, all ratios are
; given. If it is a list, the list should be that of accumulating variables.
; GIVE-RATIOS will give the the ratio of the 1st variable to the 2nd, 3rd to the
; 4th, etc.
(defmethod
  (give-ratios cycle)
  (&optional (ratios-to-be-done :all) &aux
    ;Hashtable of function applications in the expression of interest.
    ;Key is name of function (string),
    ;Value is a list of the names of arguments (strings) for this function.
    (fcn-ht (make-hash-table :test #'equal))
    (simple-constants     ;constants that are NOT function applications or numbers
      (loop with simple-constants = nil
            for constant in (cons rate constants) do   ;treat rate as a constant
        (if (not (numberp constant))
            (multiple-value-bind
              (fcn-name arg-string) (applying-a-function? constant)
              (if fcn-name
                  (push arg-string (gethash fcn-name fcn-ht));function application
                  (push constant simple-constants))))

            finally (return simple-constants))))

  (progv '(qm::*context*) (list inequality-reasoner-storage)
    (format t "~%RATIOS of RATES for the cycle ~a:" name)
    (if found-inconsistency?
        (print "Inconsistency found in conditions, ignore the bounds."))
    (if (listp ratios-to-be-done)
        ;look at given pairs
        (loop for pairs on ratios-to-be-done by #'cddr
               for accum-var1 = (first pairs)  for accum-var2 = (second pairs)
               for accum-per-cycle1 = (cdr (assoc accum-var1 accum-per-cycle))
               for accum-per-cycle2 = (cdr (assoc accum-var2 accum-per-cycle)) do
           (analyze-expr (simplify (list '/ accum-per-cycle1 accum-per-cycle2))
                         (format nil "Ratio of accumulating ~a to ~a"
                                 accum-var1 accum-var2)
                         '"Ratio" simple-constants fcn-ht '(:eq :curve-shapes)))
          ;look at all pairs
          (loop for accum-pairs on accum-per-cycle
                for (accum-var1 . accum-per-cycle1) = (first accum-pairs) do
            (loop for (accum-var2 . accum-per-cycle2) in (rest accum-pairs) do
              (analyze-expr (simplify (list '/ accum-per-cycle1 accum-per-cycle2))
                            (format nil "Ratio of accumulating ~a to ~a"
                                    accum-var1 accum-var2)
                            '"Ratio" simple-constants fcn-ht
                            '(:eq :curve-shapes))))))


;Auxillary function for the analysis methods above.
;For EXPR, look at it, its derivatives w.r.t. to each simple constant, and how
; altering functions affect it.
;The inequality context is assumed to be qm::*context*.
;The optional arguments are lists of phase properties. They are only needed if the
; :phases option is being used.
;Print-options is a list of of the desired options. The options are:
; :eq -print the equation of the derivative
; :curve-shapes -draw the possible curve shapes of the rate vs. each constant
: :phases -try to figure out the contributions of each phase to a value
;          (not yet implemented for effects of increasing functions)
(defun analyze-expr (expr long-name short-name simple-constants fcn-ht
                     print-options &optional
                     (phase-subexprs nil) (phase-names nil)
                     &aux (ex (eval-in-cms expr)))
  (multiple-value-bind (lb lb-ex?) (qm::lb ex)
    (multiple-value-bind (ub ub-ex?) (qm::ub ex)
      (let ((string1 (format nil "~a<~:[=~;~] ~a <~:[=~;~]~a. Eq. is"
                             lb lb-ex? long-name ub-ex? ub))
            (string2 (format nil " ~a." expr)))
        (if (<= (+ (length string1) (length string2)) 105.)
            (format t "~%~%~a~a" string1 string2)
            ;indent properly on output for expr
            (format t "~%~%~a~% ~a." string1 expr)))
      (if (member :phases print-options)
          (examine-phase-exprs lb lb-ex? ub ub-ex? phase-subexprs phase-names 4))))

  ;for each simple constant, find derivatives of EXPR w.r.t it
  (mapcar
    #'(lambda (constant)
        (let* ((deriv1 (deriv expr constant)) (d1 (eval-in-cms deriv1))
               (deriv2 (deriv deriv1 constant)) (d2 (eval-in-cms deriv2))
               (phase-subexprs1 (mapcar #'(lambda (subexpr)
                                            (if subexpr (deriv subexpr constant)))
                                        phase-subexprs)))
          (multiple-value-bind (1lb 1lb-ex?) (qm::lb d1)
             (multiple-value-bind (1ub 1ub-ex?) (qm::ub d1)
               (multiple-value-bind (2lb 2lb-ex?) (qm::lb d2)
                 (multiple-value-bind (2ub 2ub-ex?) (qm::ub d2)
                   (if (and (numberp deriv1) (zerop deriv1))
                       (progn
                         (format t "~%~%  ~a independent of ~a."
                                 short-name constant)
                         (if (member :phases print-options)
                                   (examine-phase-exprs
                                     1lb 1lb-ex? 1ub 1ub-ex?
                                     phase-subexprs1 phase-names 6)))
                       (let ((str1 (format nil "  ~a<~:[=~;~] d~a/d(~a) ~
<~:[=~;~]~a. Eq. is" 1lb 1lb-ex? short-name constant 1ub-ex? 1ub))
                             (str2 (format nil " ~a." deriv1)))
                         (if (member ':eq print-options)
                             (progn
                               (if (<= (+ (length str1) (length str2)) 105.)
                                   (format t "~%~%~a~a" str1 str2)
                                   ;indent properly on output for deriv1
                                   (format t "~%~%~a~%    ~a." str1 deriv1))
                               (if (member :phases print-options)
                                   (examine-phase-exprs
                                     1lb 1lb-ex? 1ub 1ub-ex?
                                     phase-subexprs1 phase-names 6))
                               (if (and (numberp deriv2) (zerop deriv2))
                                   (format t "~%  ~a is a linear fcn. of ~a."
                                           short-name constant)
                                   (let ((s1 (format nil "  ~a<~:[=~;~] ~
d^2~a/d(~a)^2 <~:[=~;~]~a. Eq. is" 2lb 2lb-ex? short-name constant 2ub-ex? 2ub))
                                         (s2 (format nil " ~a." deriv2)))
                                     (if (<= (+ (length s1) (length s2)) 105.)
                                         (format t "~%~a~a" s1 s2)
                                         ;indent properly on output for deriv2
                                         (format t "~%~a~%    ~a." s1 deriv2))))
                               (if (member :phases print-options)
                                   (examine-phase-exprs
                                     2lb 2lb-ex? 2ub 2ub-ex?
                                     (mapcar #'(lambda (subexpr)
                                                 (if subexpr
                                                     (deriv subexpr constant)))
                                             phase-subexprs1)
                                     phase-names 6)))
                             (progn
                               (format t "~%~%  ~a<~:[=~;~] d~a/d(~a) <~:[=~;~]~a."
                                           1lb 1lb-ex? short-name
                                           constant 1ub-ex? 1ub)
                               (if (member :phases print-options)
                                   (examine-phase-exprs
                                     1lb 1lb-ex? 1ub 1ub-ex?
                                     phase-subexprs1 phase-names 6))
                               (if (and (numberp deriv2) (zerop deriv2))
                                   (format t "~%  ~a is a linear fcn. of ~a."
                                           short-name constant)
                                   (format
                                  t "~%  ~a<~:[=~;~] d~2~a/d(~a)~2 <~:[=~;~]~a."
                                  2lb 2lb-ex? short-name constant 2ub-ex? 2ub))
                            (if (member :phases print-options)
                                (examine-phase-exprs
                                  2lb 2lb-ex? 2ub 2ub-ex?
                                  (mapcar #'(lambda (subexpr)
                                              (if subexpr
                                                  (deriv subexpr constant)))
                                          phase-subexprs1)
                                  phase-names 6))))
                      (if (member ':curve-shapes print-options)
                          (show-curve-shape 1lb 1lb-ex? 1ub 1ub-ex?
                                            2lb 2lb-ex? 2ub 2ub-ex?
                                            (symbol-name constant) short-name
                                            *standard-output* 5))))))))))
  simple-constants)

;for each function, observe the effects of increasing it on the EXPR (if can be
;easily done)
(maphash
  #'(lambda
      (fcn-name arg-string-list)
      (if
        (multiple-value-bind
          (base-name take-inverse?) (function-inverse? fcn-name)
          (loop for symbol in (depends expr)
                thereis
                  (multiple-value-bind
                    (fcn-name2? arg-string-list2?) (applying-a-function? symbol)
                    (if fcn-name2?
                        (or
                          ;Test to see if there is a top level function
                          ;application in the expression that uses the same
                          ;function as FCN-NAME but is the inverted version when
                          ;FCN-NAME is not or vice-versa.
                          (multiple-value-bind
                            (base-name2 take-inverse2?)
                              (function-inverse? fcn-name2?)
                            (and (not (eq take-inverse? take-inverse2?))
                                 (equal base-name base-name2)))

                          ;Test to see if there is a lower level function
                          ;application in the expression that uses the same
                          ;function name as FCN-NAME
                          (loop with list-of-arg-string-lists
                                  = (list arg-string-list2?)
                                while (not (null list-of-arg-string-lists))
                                for arg-string = (pop list-of-arg-string-lists)
                                thereis
                                  (loop with args = (split-args arg-string)
                                        while (not (null args))
                                        for arg = (pop args)
                                        thereis
                                          (cond
                                                         ((symbolp arg)
                                                          (multiple-value-bind
                                                            (fcn-name3? arg-string-list3?)
                                                              (applying-a-function? arg)
                                                            (if fcn-name3?
                                                                (if
                                                                  (equal
                                                                    base-name
                                                                    (function-inverse?
                                                                      fcn-name3?))
                                                                    t ;function a lower level
                                                                      ;fcn. application of the
                                                                      ;same (modulo inverting)
                                                                      ;as FCN-NAME, return
                                                                      ;true
                                                                    ;not the same function,
                                                                    ;append to arg-string
                                                                    ;list to look further,
                                                                    ;but, return nil for now
                                                                    (progn
                                                                     (push
                                                                      arg-string-list3?
                                                                      list-of-arg-string-lists
                                                                      )
                                                                      nil)))))
                                                         ;ARG a complex expression, insert
                                                         ;the symbols in ARG into ARGS (to
                                                         ;examine later) & return nil
                                                         ;(FCN-NAME not found yet)
                                                         ((listp arg)
                                                          (setq args
                                                                (append (depends arg) args))
                                                          nil)))))))))

          (format t "~%~% Cannot analyze the effects of ~a being different in ~a ~
at this time." fcn-name short-name) ;too weird to analyze easily

          (let* ;can analyze
            ;List of (fcn[arg] . (+ fcn[arg] eps[arg])) for each arg-string (as
            ;arg) mentioned with each fcn. Also assert that each eps[arg] is >0.
            ((substitute-pairs
               (mapcar
                 #'(lambda (arg-string &aux
                             (fcn[arg]
                               (intern (concatenate 'string fcn-name
                                                    '"[" arg-string '"]")))
                             (eps[arg]
                               (intern (concatenate
                                          'string '"eps[" arg-string '"]"))))
                     (if (not (qm::assert< 0 eps[arg])) ;contradiction found
                         (progn (format t "~%Found a contradiction. Inequality ~
reasoner storage:")
                                (describe qm::*context*)
                                (error "Adding condition 0<~a revealed a ~
contradiction. The inequality reasoner storage is given above." eps[arg])))
                    (cons fcn[arg] (list '+ fcn[arg] eps[arg])))
                  arg-string-list))
             (expr2 (sublis substitute-pairs expr)))
            (if (equal expr2 expr)
                (format t "~%~%  ~a independent of the function ~a."
                        short-name fcn-name)
                (let ((dif (eval-in-cms (list '- expr2 expr))))
                  (multiple-value-bind (lb lb-ex?) (qm::lb dif)
                    (multiple-value-bind (ub ub-ex?) (qm::ub dif)
                      (if (member ':eq print-options)
                          (let ((string1 (format nil "  ~a<~:[=~;~] ~a change ~
due to inc. in the ~a fcn. <~:[=~;~]~a. Eq. is"
                                                 lb lb-ex? short-name
                                                 fcn-name ub-ex? ub))
                                (string2 (format nil " ~a." expr2)))
                            (if (<= (+ (length string1) (length string2)) 105.)
                                (format t "~2%~a~a" string1 string2)
                                ;indent properly on output for expr2
                                (format t "~2%~a~%    ~a." string1 expr2)))
                          (format t "~2%  ~a<~:[=~;~] ~a change due to inc. in ~
the ~a fcn. <~:[=~;~]~a." lb lb-ex? short-name fcn-name ub-ex? ub))))))))
    fcn-ht))

;For the expression whose LB LB-outside? UB UB-outside? is given in the arguments,
;examine the subexpressions of that expression from each phase.
;The phase subexpressions and names are given in P-subexprs and P-names
;respectively.
;A P-subexprs entry is nil if the subexpression in that phase is unsolved.
;Assume qm::*context* is the context for the inequality reasoning
(defun examine-phase-exprs
       (lb lb-outside? ub ub-outside? p-subexprs p-names num-spaces-to-indent &aux
        (spaces (make-sequence 'string num-spaces-to-indent :initial-element #\ ))
        ;lists of the phases with a subexpression with the indicated property
        (list-increased nil) (list-may-have-increased nil)
        (list-may-have-decreased nil) (list-decreased nil) (list-no-effect nil)
        (list-unknown-effect nil) (list-unsolved))
  ;Categorize the phase subexpressions
  (loop for phase-name in p-names  for subexpr in p-subexprs
        for se = (eval-in-cms subexpr)  for phase-num from 0 do
    (if subexpr
        (multiple-value-bind (lb lb-ex?) (qm::lb se)
          (multiple-value-bind (ub ub-ex?) (qm::ub se)
            (cond ((more-than-0? lb lb-ex?)
                   (setq list-increased
                         (list* phase-num phase-name list-increased)))
                  ((equal-0? lb lb-ex? ub ub-ex?)
                   (setq list-no-effect
                         (list* phase-num phase-name list-no-effect)))
                  ((less-than-0? ub ub-ex?)
                   (setq list-decreased
                         (list* phase-num phase-name list-decreased)))
                  ((at-least-0? lb)
                   (setq list-may-have-increased
                         (list* phase-num phase-name list-may-have-increased)))
                  ((at-most-0? ub)
                   (setq list-may-have-decreased
                         (list* phase-num phase-name list-may-have-decreased)))
                  (t (setq list-unknown-effect
                           (list* phase-num phase-name list-unknown-effect))))))
       (setq list-unsolved (list* phase-num phase-name list-unsolved))))
  ;Print out categories (they are in reverse order.)
  (let ((expr-cat (cond ((equal-0? lb lb-outside? ub ub-outside?) '0-or-unknown)
                        ((at-least-0? lb) '>=0)
                        ((at-most-0? ub) '<=0)
                        (t '0-or-unknown))))
    (case expr-cat
      (0-or-unknown
        (if (not (null list-increased))
            (format t "~%~aThe following phase(s) increased the value:~
~{ ~a (~a)~^,~}." spaces (reverse list-increased)))
        (if (not (null list-may-have-increased))
            (format t "~%~aThe following phase(s) may have increased the value:~
~{ ~a (~a)~^,~}." spaces (reverse list-may-have-increased)))
        (if (not (null list-may-have-decreased))
            (format t "~%~aThe following phase(s) may have decreased the value:~
~{ ~a (~a)~^,~}." spaces (reverse list-may-have-decreased)))
        (if (not (null list-decreased))
            (format t "~%~aThe following phase(s) decreased the value:~
~{ ~a (~a)~^,~}." spaces (reverse list-decreased))))
      (>=0
        (if (not (null list-increased))
            (format t "~%~aThe following phase(s) helped make the value as large ~
as it is:~{ ~a (~a)~^,~}." spaces (reverse list-increased)))
        (if (not (null list-may-have-increased))
            (format t "~%~aThe following phase(s) may have helped make the value ~
as large as it is:~{ ~a (~a)~^,~}." spaces (reverse list-may-have-increased)))
        (if (not (null list-may-have-decreased))
            (format t "~%~aThe following phase(s) may have decreased the value:~
~{ ~a (~a)~^,~}." spaces (reverse list-may-have-decreased)))
        (if (not (null list-decreased))
            (format t "~%~aThe following phase(s) decreased the value:~
~{ ~a (~a)~^,~}." spaces (reverse list-decreased))))
      (<=0
        (if (not (null list-decreased))
            (format t "~%~aThe following phase(s) helped make the value as ~
negative as it is:~{ ~a (~a)~^,~}." spaces (reverse list-decreased)))
        (if (not (null list-may-have-decreased))
            (format t "~%~aThe following phase(s) may have helped make the value as ~
negative as it is:~{ ~a (~a)~^,~}." spaces (reverse list-may-have-decreased)))
        (if (not (null list-may-have-increased))
            (format t "~%~aThe following phase(s) may have made the value less ~
negative:~{ ~a (~a)~^,~}." spaces (reverse list-may-have-increased)))
        (if (not (null list-increased))
            (format t "~%~aThe following phase(s) made the value less negative:~
~{ ~a (~a)~^,~}." spaces (reverse list-increased))))))
  (if (not (null list-no-effect))
      (format t "~%~aThe following phase(s) had no effect:~{ ~a (~a)~^,~}."
                spaces (reverse list-no-effect)))
  (if (not (null list-unknown-effect))
      (format t "~%~aThe following phase(s) had an unknown effect:~
~{ ~a (~a)~^,~}." spaces (reverse list-unknown-effect)))
  (if (not (null list-unsolved))
      (format t "~%~aThe following phase(s) had an unsolved for effect:~
~{ ~a (~a)~^,~}." spaces (reverse list-unsolved))))



;drawing CURVES from 1st & 2nd derivatives

;For arguments:
; lb & ub stand for lower & upper bound respectively
; 1 stands for 1st derivative, 2 for 2nd derivative
; LBO is true if the corresponding LB is just outside the actual region and is nil
; if LB is at the inside edge of the region. Similarly for UBO. So lb=2 lbo=t ub=4
; ubo=nil means (2,4]. This follows the convention's of Elisha Sacks' bounding
; system.
; the axis-names are strings
(defun
  show-curve-shape
  (lb1 lbo1 ub1 ubo1 lb2 lbo2 ub2 ubo2 X-axis-name Y-axis-name stream
   num-spaces-to-indent &aux
   (spaces (make-sequence 'string num-spaces-to-indent :initial-element #\ ))
   (possible-shape-list ;Assume a smooth curve.
                        ;empty list means no shape possible,
                        ;the symbol '? means the shape is unknown
     (cond ((or (qm::number<+ ub1 ubo1 lb1 lbo1) (qm::number<+ ub2 ubo2 lb2 lbo2))
            nil)
           ((more-than-0? lb2 lbo2)
            (cond ((equal-0? lb1 lbo1 ub1 ubo1) nil)
                  ((at-most-0? ub1) '(backward-J))
                  ((at-least-0? lb1) '(J))
                  (t '(backward-J U J))))
           ((equal-0? lb2 lbo2 ub2 ubo2)
            (cond ((less-than-0? ub1 ubo1) '(ramp-down))
                  ((equal-0? lb1 lbo1 ub1 ubo1) '(flat))
                  ((more-than-0? lb1 lbo1) '(ramp-up))
                  ((at-most-0? ub1) '(ramp-down flat))
                  ((at-least-0? lb1) '(flat ramp-up))
                  (t '(ramp-down flat ramp-up))))
           ((less-than-0? ub2 ubo2)
            (cond ((equal-0? lb1 lbo1 ub1 ubo1) nil)
                  ((at-most-0? ub1) '(upsidedown-J))
                  ((at-least-0? lb1) '(rotate-180-J))
                  (t '(rotate-180-J upsidedown-U upsidedown-J))))
           ((at-least-0? lb2)
            (cond ((less-than-0? ub1 ubo1) '(backward-J ramp-down))
                  ((equal-0? lb1 lbo1 ub1 ubo1) '(flat))
                  ((more-than-0? lb1 lbo1) '(ramp-up J))
                  ((at-most-0? ub1) '(backward-J ramp-down flat))
                  ((at-least-0? lb1) '(flat ramp-up J))
                  (t '(backward-J ramp-down flat ramp-up J U))))
           ((at-most-0? ub2)
            (cond ((less-than-0? ub1 ubo1) '(upsidedown-J ramp-down))
                  ((equal-0? lb1 lbo1 ub1 ubo1) '(flat))
                  ((more-than-0? lb1 lbo1) '(ramp-up rotate-180-J))
                  ((at-most-0? ub1) '(upsidedown-J ramp-down flat))
                  ((at-least-0? lb1) '(flat ramp-up rotate-180-J))
                  (t '(upsidedown-J ramp-down flat ramp-up
                                    rotate-180-J upsidedown-U))))
           (t '?))))
  (cond
    ((not possible-shape-list)
     (format stream "No possible smooth shape for the curve of ~a vs. ~a."
             Y-axis-name X-axis-name))
    ((eq possible-shape-list '?)) ;don't say anything about the shape
    (t ;show list of possible shapes, use Dutch char. style because it seems
       ;the thinnest
     (if (rest possible-shape-list) ;more than 1 possible shape
         (format stream "~%~aThe possible shapes of the curve for ~a vs. ~a ~
are~2% " spaces Y-axis-name X-axis-name)
         (format stream "~%~aThe shape of the curve for ~a vs. ~a is~2% "
                 spaces Y-axis-name X-axis-name))
     (graphics:with-room-for-graphics (stream)
       (multiple-value-bind (width height cmx cmy left top)
           ;simulate partial print-out to find x offset
           (dw:continuation-output-size
             #'(lambda (stream)
                 (graphics:draw-string
                   (concatenate 'string Y-axis-name spaces) 0. 0.
                   :attachment-x :right :stream stream
                   :character-style '(:dutch nil nil)))
             stream)
         ;move so graph's left side is on the screen
         (graphics:with-graphics-translation (stream (max 0. (- left)) 0.)
           (loop with x-distance-between-graph-left-sides = 65.
                 with mxro = 40. ;maximum curve x value relative to x offset
                 with lyc = 30. ;lowest-y-for-curves
                 with hyc = 50. ;highest-y-for-curves
                 for shape in possible-shape-list
                 for shapes-left on possible-shape-list
                 for x-offset = 15.
                              then (+ x-offset x-distance-between-graph-left-sides)
                 initially
                   (graphics:draw-string
                     Y-axis-name 0. hyc :stream stream :attachment-x :right
                     :attachment-y :top :character-style '(:dutch nil nil))
                 do
             (if (and (rest possible-shape-list) (not (rest shapes-left)))
                 ;at last shape of a list of more than 1 shapes, print or
                 (progn (graphics:draw-string "or" x-offset (/ (+ lyc hyc) 2.)
                                              :stream stream :attachment-y :center
                                              :character-style '(:dutch nil nil))
                        (incf x-offset 40.)))
             (case shape
               (ramp-down (graphics:draw-line x-offset hyc (+ x-offset mxro) lyc
                                  :stream stream :thickness 2))
  (flat (graphics:draw-line
         x-offset (/ (+ lyc hyc) 2.) (+ x-offset mxro)
         (/ (+ lyc hyc) 2.) :stream stream :thickness 2))
  (ramp-up (graphics:draw-line x-offset lyc (+ x-offset mxro) hyc
                                :stream stream :thickness 2))
  (U (graphics:draw-cubic-spline
       (list x-offset hyc (+ x-offset (floor (* mxro .15)))
             (how-far-up-from-low lyc hyc .3)
             (+ x-offset (floor mxro 2.)) lyc
             (+ x-offset (floor (* mxro .85)))
             (how-far-up-from-low lyc hyc .3) (+ x-offset mxro) hyc)
       :stream stream :thickness 2))
  (upsidedown-U
    (graphics:draw-cubic-spline
      (list x-offset lyc (+ x-offset (floor (* mxro .15)))
            (how-far-up-from-low lyc hyc .7)
            (+ x-offset (floor mxro 2.)) hyc
            (+ x-offset (floor (* mxro .85)))
            (how-far-up-from-low lyc hyc .7) (+ x-offset mxro) lyc)
      :stream stream :thickness 2))
  (backward-J
    (graphics:draw-cubic-spline
       (list x-offset hyc (+ x-offset (floor (* mxro .3)))
             (how-far-up-from-low lyc hyc .3) (+ x-offset mxro) lyc)
       :stream stream :thickness 2))
  (J (graphics:draw-cubic-spline
       (list x-offset lyc (+ x-offset (floor (* mxro .7)))
             (how-far-up-from-low lyc hyc .3) (+ x-offset mxro) hyc)
       :stream stream :thickness 2))
  (upsidedown-J
    (graphics:draw-cubic-spline
       (list x-offset hyc (+ x-offset (floor (* mxro .7)))
             (how-far-up-from-low lyc hyc .7) (+ x-offset mxro) lyc)
       :stream stream :thickness 2))
  (rotate-180-J
    (graphics:draw-cubic-spline
       (list x-offset lyc (+ x-offset (floor (* mxro .3)))
             (how-far-up-from-low lyc hyc .7) (+ x-offset mxro) hyc)
       :stream stream :thickness 2))
  (otherwise (error "The ~a shape is unknown." shape)))
(if (rest (rest shapes-left)) ;not last or second to last shape
    (graphics:draw-string
      "," (+ x-offset mxro 10.) lyc :stream stream
      :character-style '(:dutch nil nil)))

    finally
      (graphics:draw-string
        '"." (+ x-offset mxro 10.) lyc :stream stream
        :character-style '(:dutch nil nil))
      (graphics:draw-string
        X-axis-name (+ (ceiling x-offset 2.) 3.) 0. :stream stream
        :character-style '(:dutch nil nil))))))))
```

```
(defun how-far-up-from-low (low high frac) (round (+ low (* frac (- high low)))))

;need QM versions to handle :infinity, :-infinity
(defun more-than-0? (lb lbo) (or (qm::plusp lb) (and lbo (qm::zerop lb))))
(defun equal-0? (lb lbo ub ubo)
  (and (not lbo) (not ubo) (qm::zerop lb) (qm::zerop ub)))
(defun less-than-0? (ub ubo) (or (qm::minusp ub) (and ubo (qm::zerop ub))))
(defun at-least-0? (lb) (not (qm::minusp lb)))
(defun at-most-0? (ub) (not (qm::plusp ub)))
```

E.3 File Cycle-Util.lisp

This file contains some low level utility functions for the implementation:

```
;;; -*- Mode: LISP; Package: CYCLE -*-
;basic utilities for cycle
(print "LOAD THE INEQUALITY REASONING SYSTEM BY LOADING FILE
Z:>ELISHA>QM>SYSTEM BEFORE COMPILING OR LOADING THE CYCLE SYSTEM.")

;evaluate expressions using functions in the QM package & simplify
(defun eval-in-cms (expression)
  (if (atom expression) expression ;do not evaluate numbers & symbols
      ;a list, 'apply' function to evaluated arguments
      (let ((evaled-args (mapcar #'eval-in-cms (rest expression))))
        (qm::simplify (apply (intern (symbol-name (first expression))
                                     (find-package 'qm))
                             evaled-args)))))

;basically from Z:>WJL>V7>HF (HF:HF;) KBASE.LISP

(defvar *lst*)
(defvar *integrate*)
(defun depends (expr &aux *lst* *integrate*)
  (depends1 expr)
  (values *lst* *integrate*))
;version of depends that treats statistics functions like symbols
(defun depends-stat (expr &aux *lst* *integrate*)
  (depends1s expr)
  (values *lst* *integrate*))

(defun depends1 (expr)
  (typecase expr (number expr)
            (list (if (eq (car expr) 'integrate)(setq *integrate* 'integrate))
                  (if (eq (car expr) 'control)  ; hack to get values control state
                      ;should have more general mechanism (list of operators maybe)
                      (setq *lst* (adjoin expr *lst*))
                      (loop for x in (cdr expr) do (depends1 x))))
            (symbol (setq *lst* (adjoin expr *lst*)))
            (t (cerror "continue" "strange expression ~A" expr))))

(defun depends1s (expr)
  (typecase expr (number expr)
            (list (if (eq (car expr) 'integrate)(setq *integrate* 'integrate))
                    (if (eq (car expr) 'control)  ; hack to get values control state
                        ;should have more general mechanism (list of operators maybe)
                        (setq *lst* (adjoin expr *lst*))))
              (symbol (setq *lst* (adjoin expr *lst*)))
              (t (cerror "continue" "strange expression ~A" expr))))

;does NOT put variables in alphabetal order, NOR does it put operands which are
; products, sums, etc. into any canonical order
(defvar *canoned* nil)
(defun canonic (expr)
  (cond ((numberp expr) expr)
        ((symbolp expr) expr)
        ((null expr) expr) ;addition
        ((case (car expr)
           ((+ *) (cons (car expr)(canonic1 (cdr expr))))
           ((- / div)
            (list* (car expr)(canonic (cadr expr))(canonic1 (cddr expr))))
           (expt `(expt ,(canonic (cadr expr)) ,(caddr expr)))
           ((exp log) `(,(car expr) ,(canonic (cadr expr))))
           ;(if (list (car expr)(cadr expr)(canonic (caddr expr))
           ;          (canonic (cadddr expr))))
           ;((< >)(list (car expr)(canonic (cadr expr))(canonic (caddr expr))))
           (t (cerror "continue" "don't know how to canonic ~A" expr))))
        (t (cerror "continue" "don't know how to canonic ~A" expr))))

(defun canonic1 (lst)
  (if (loop for x on lst with order = 0 and type do
        (setq type (typecase (car x) (number 0)(symbol 1)(cons 2)
                             (t (cerror "continue" "strange list ~A in canonic"
                                        lst))))
        (if (< type order)(setq *canoned* t)(setq order type))
        (if (= type 2)(setf (car x)(canonic (car x))))
        finally (return *canoned*))
      (nconc (loop for x in lst when (numberp x) collect x)
             (loop for x in lst when (symbolp x) collect x)
             (loop for x in lst when (consp x) collect x))
      lst))

;Does NOT simplify expressions of the form (# some (& ) (# rest) rest), where # is
; + or *, and & is an operator other than #. The matcher does not work properly in
; these cases.
(defun simplify (eqn)
  (loop for y = nil then x
        for x = (canonic eqn) then (simp x)
        when (equal x y) do (return x)))

;altered
(defun simp (expr)
  (cond ((numberp expr) expr)
        ((symbolp expr) expr)
        ((match expr '(+ x))(cadr expr))
        ((match expr '(+ 0 rest)) `(+ ,@(cddr expr)))
        ((match expr '(+ n n rest))
         `(+ ,(+ (cadr expr)(caddr expr)) ,@(cdddr expr)))
```

```
((match expr '(+ some (+ rest) rest))
 '(+ ,@(loop for x in (cdr expr)
             when (and (consp x)(eq (car x) '+))
               append (cdr x)
             else collect x)))
((match expr '(div 0 rest)) 0)
((match expr '(/ 0 rest)) 0)
((match expr '(/ n))(/ (cadr expr)))
((match expr '(div n))(/ (cadr expr)))
((and (match expr '(if x x x))(equal (caddr expr)(cadddr expr)))
 (caddr expr))
((match expr '(- 0 x)) '(- ,(caddr expr)))
((match expr '(- x 0)) (cadr expr))
((match expr '(- n n)) (- (cadr expr)(caddr expr)))
((match expr '(- n n x rest))
 '(- ,(- (cadr expr)(caddr expr)) ,@(cdddr expr)))
((match expr '(- n))(- (cadr expr)))
((match expr '(- (* rest)))'(* -1.0 ,@(cdadr expr)))
((match expr '(* x))(cadr expr))
((match expr '(* 0 rest)) 0)
((match expr '(* some (* rest) rest))
 '(* ,@(loop for x in (cdr expr)
             when (and (consp x)(eq (car x) '*))
               append (cdr x)
             else collect x)))
((match expr '(* 1 rest)) '(* ,@(cddr expr)))
((match expr '(* n n rest))
 '(* ,(* (cadr expr)(caddr expr)) ,@(cdddr expr)))
;added partial distribution of * over +, and expt stuff
;want to limit use of distributions to where form of equations won't be
;disrupted too much
((match expr '(* (+ some (* some n rest) rest) n))
 (cons '+ (loop with multiplier = (third expr)
                for sum-element in (rest (second expr))
                collect '(* ,sum-element ,multiplier))))
((match expr '(* n (+ some (* some n rest) rest)))
 (cons '+ (loop with multiplier = (second expr)
                for sum-element in (rest (third expr))
                collect '(* ,multiplier ,sum-element))))
((match expr '(expt x 0)) 1.)
((match expr '(expt x 1)) (cadr expr))
((match expr '(expt n n)) (expt (second expr) (third expr)))
;may mess up exponentiations of negative values.
;ex.: ((-2)^2)^.5 = - (-2)^1
((match expr '(expt (expt rest) n))
 '(expt ,(cadadr expr) ,(* (third (second expr)) (third expr))))
;only distribute expt over * when taking to an integer power, otherwise
;will mess-up on cases like (expt xy 1/2) when x, y < 0
((match expr '(expt (* rest) int))
 (cons '* (loop with exponent = (third expr)
                for product-element in (rest (second expr))
                collect '(expt ,product-element ,exponent))))
(t
 (loop with *canoned* = nil
            for nexpr = (canonic (cons (car expr)
                                       (loop for x in (cdr expr)
                                             collect (simp x))))
                  then (canonic (cons (car expr)
                                      (loop for x in (cdr nexpr)
                                            collect (simp x))))
            do (if *canoned* (setq *canoned* nil)(return nexpr))))))

;altered to match to integers
;cannot match things like '(+ a b) to patterns like '(+ some x) because 'a matches
;to 'x leaving 'b matching to nothing
(defun match (expr pat)
  (loop for (ex . rex) on expr with (p skip)
        do (or pat (return (null ex)))
           (case (car pat)(rest (return t))
                 (some (setq skip t)(setq pat (cdr pat))))
           (setq p (car pat))
           (if (consp p)
               (if (and (consp ex) (match ex p)) (setq skip nil pat (cdr pat))
                   (or skip (return nil)))
               (case p (x (if ex (setq skip nil pat (cdr pat))(return nil)))
                       (int (if (integerp ex)(setq skip nil pat (cdr pat))
                                (or skip (return nil))))
                       (n (if (numberp ex)(setq skip nil pat (cdr pat))
                              (or skip (return nil))))
                       (t (if (numeql ex p)(setq skip nil pat (cdr pat))
                              (or skip (return nil))))))
        finally (return (or (null pat)(eq (car pat) 'rest)))))

(defun numeql (ex p)
  (or (eql ex p) (and (numberp ex)(numberp p)(= ex p))))

;dealing with DERIVATIVES
;based in part on p.106-8 of Abelson & Sussman's 6.001 text

;symbol notation: F[A] = 'F' applied to 'A'; %F[A] = dF/dA; %%F[A] = d^2F/dA^2;
; G[X$Y$Z] = 'G' applied to 'X','Y','Z'; %%!%G[X$Y$Z] = d^3G/(dX^2*dY);
; !!%G[X$Y$Z]=dG/dZ; etc. Do not put in redundant !'s. Need at least 1 argument.
;Can handle functions applied to other functions. Except at top level, functions
; can be ordinary ones (+, -, *, etc.).
;If not an ordinary function, can invert it: F^-1[A] = inverse of 'F' applied to
; 'A'. Do not use more than 1 level of inversion on a function.

;table of symbols considered constants. The value can be anything but nil.
(defvar *constant-ht* (make-hash-table :test #'equal))

(defun deriv (exp var)
  (cond ((constant? exp) 0.)
        ((symbolp exp)
         (multiple-value-bind (fcn-name arg-string) (applying-a-function? exp)
           (if fcn-name
               ;applying a function
               (deriv-of-fcn-wrt-var exp (split-args arg-string) var)
               (if (same-variable? exp var) 1. 0.))))          ;simple variable
        ((sum? exp) (make-sum (deriv (addend exp) var) (deriv (augend exp) var)))
        ;Use of addend & augend not really correct for differences. Because of
        ;Elisha Sacks' CMS, only really need to handle differences with up to 2
        ;arguments
        ((and (difference? exp) (= (list-length exp) 2.)) ;1 arg.
         (make-product -1. (deriv (arg exp) var)))
        ((difference? exp)
         (make-sum (deriv (addend exp) var)
                   (make-product -1. (deriv (augend exp) var))))
        ((product? exp)
         (make-sum (make-product (multiplier exp) (deriv (multiplicand exp) var))
                   (make-product (multiplicand exp) (deriv (multiplier exp) var))))
        ;Use of multiplier & multiplcand not really correct for differences.
        ;Because of Elisha Sacks' CMS, only really need to handle differences with
        ;up to 2 arguments
        ((and (quotient? exp) (= (list-length exp) 2.)) ;1 arg.
         (make-product (make-product -1. (make-exponential (arg exp) -2.))
                       (deriv (arg exp) var)))
        ((quotient? exp)
         (deriv (make-product (multiplier exp)
                              (make-exponential (multiplicand exp) -1.)) var))
        ((exponential? exp)
         (make-sum
           (make-product
             (make-product (exponent exp)
                           (make-exponential
                             (base exp) (make-sum (exponent exp) -1.)))
             (deriv (base exp) var))
           (make-product exp
                         (make-product (make-log (base exp))
                                       (deriv (exponent exp) var)))))
        ((logarithm? exp) (make-product (make-exponential (arg exp) -1.)
                                        (deriv (arg exp) var)))
        ((to-the-e? exp) (make-product exp (deriv (arg exp) var)))
        (t (error "Deriv: can't handle ~A" exp))
        ))

(defun constant? (x) (or (numberp x) (and (symbolp x) (gethash x *constant-ht*))))

(defun same-variable? (v1 v2) (equal v1 v2)) ;assume both v1 & v2 are variables

(defun applying-a-function?
       (symbol &aux (letters (symbol-name symbol)) (last (- (length letters) 1.))
        ([-&-]-exist? (if (eql '#\] (elt letters last)) (position '#\[ letters))))
  (if (and [-&-]-exist? (< 0. [-&-]-exist? last))
      ;Applying a function to an argument. Return strings of the function name and
      ; arguments (all the arguments are left concatenated together, use
      ; SPLIT-ARGS to split them apart), respectively.
      ;The function SUBSEQ acts funny, the arguments seem to be 1) the sequence,
      ;2) index of the 1st char to be included, 3) index of the 1st char to be
      ;EXCLUDED
      (values (subseq letters 0. [-&-]-exist?)
              (subseq letters (+ [-&-]-exist? 1.) last))
      ;simple variable or constant
      (values nil nil)))

;fcn-name is a string
(defun function-inverse? (fcn-name &aux (position-of-^-for-inverse
                                        (- (length fcn-name) 3.)))
  (if (and (not (minusp position-of-^-for-inverse))
           (equal (subseq fcn-name position-of-^-for-inverse) '"^-1"))
      ;fcn-name is an inverse, return the name of the fcn being inverted and T
      (values (subseq fcn-name 0. position-of-^-for-inverse) t)
      (values fcn-name nil)));not an inverse, return the original fcn-name and NIL

;take a string of arguments (given by APPLYING-A-FUNCTION? and EXPAND-1) and
;return a list of the argument objects (take each argument & feed it to the reader)
(defun split-args (arg-string &aux (level-of-args 0.)
                   (index-of-1st-char-in-next-arg 0.) (reversed-arg-objects nil))
  (loop for i from 0 to (- (length arg-string) 1.)
        for char = (char arg-string i)            do
    (cond ((eql char '#\[) (incf level-of-args));increase level of argument nesting
          ((eql char '#\]) (decf level-of-args)) ;decrease level
          ((and (eql char '#\$) (zerop level-of-args))
           ;new argument coming up, save out current arg. and update pointer to
           ;next arg
           (push (expand-1 (subseq arg-string index-of-1st-char-in-next-arg i))
                 reversed-arg-objects)
           (setq index-of-1st-char-in-next-arg (+ i 1)))))
  ;add last argument & reverse the reversed argument order
  (reverse (cons (expand-1 (subseq arg-string index-of-1st-char-in-next-arg))
                 reversed-arg-objects)))

;If EXPR is a symbol that is a function application in which the function is a
; standard (+, -, *, etc.) one, expand (convert) EXPR into a standard
; s-expression for such a function as much as possible (recurse into the
; arguments). Otherwise, return the expression.
(defun expand (expr) (if (not (symbolp expr)) expr (expand-1 (symbol-name expr))))

;Like EXPAND, but STRING is assumed to represent what applying READ-FROM-STRING to
;it would return.
(defun expand-1 (string &aux (last (- (length string) 1.))
        ([-&-]-exist? (if (eql '#\] (elt string last)) (position '#\[ string))))
  (if (and [-&-]-exist? (< 0. [-&-]-exist? last))
      ;Applying a function to an argument.
      ;The function SUBSEQ acts funny, the arguments seem to be 1) the sequence,
      ;2) index of the 1st char to be included, 3) index of the 1st char to be
      ;EXCLUDED
      (let* ((fcn (subseq string 0. [-&-]-exist?))
             (args (subseq string (+ [-&-]-exist? 1.) last))
             (recognized-fcn?
               (if fcn (member fcn '("+" "-" "*" "/" "EXPT" "LOG" "EXP")
                               :test #'equal))))
        (if (not recognized-fcn?)
            ;applying an unrecognized (ordinary) function to an argument
            (read-from-string string)
            ;applying a special (recognized) function to an argument
            (cons (intern (first recognized-fcn?)) (split-args args))))
      ;simple variable or constant
      (read-from-string string)))

;SUBSEQ needs the position of the 1st element to include (start) & the leftmost
;element to EXCLUDE (end)
(defun deriv-of-fcn-wrt-var (exp args var)
  (loop with current-exp-name = (symbol-name exp)
        for arg in args
        for position-for-%
            first 0
            then (loop for i from position-for-%
                             to (- (length current-exp-name) 1.)
                       for char = (char current-exp-name i)              do
                  (cond ((eql char '#\%))
                        ((eql char '#\!) (return (+ i 1)))
                        (t ;gotten to end of all derivative markers w/o enough "!"
                           ;separaters
                         (setq current-exp-name
                               (concatenate 'string (subseq current-exp-name 0 i)
                                            '"!" (subseq current-exp-name i)))
                         (return (+ i 1)))))
        for deriv-of-fcn-wrt-arg-string =
            (concatenate 'string (subseq current-exp-name 0 position-for-%) '"%"
                         (subseq current-exp-name position-for-%))
        for partial = (make-product (intern deriv-of-fcn-wrt-arg-string)
                                    (deriv arg var))
        for result first partial then (make-sum result partial)
        finally (return result)))


;suppress unneeded 0's, 1's, additional constants, nested +'s *'s
;Place numeric constants out in front
;Later, add suppression of multiple occurrences of non-constants
(defun make-sum (a1 a2)
  (cond ((and (numberp a1) (numberp a2)) (+ a1 a2))
        ((numberp a1) (cond ((zerop a1) a2)
                            ((sum? a2) (if (numberp (addend a2))
                                           (let ((num (+ a1 (addend a2)))
                                                 (rest (cddr a2)))
                                             (if (zerop num)
                                                 (if (= (list-length rest) 1.)
                                                     (first rest) (cons '+ rest))
                                                 (list* '+ num rest)))
                                           (list* '+ a1 (cdr a2))))
                            (t (list '+ a1 a2))))
        ((numberp a2) (cond ((zerop a2) a1)
                            ((sum? a1) (if (numberp (addend a1))
                                           (let ((num (+ a2 (addend a1)))
                                                 (rest (cddr a1)))
                                             (if (zerop num)
                                                 (if (= (list-length rest) 1.)
                                                     (first rest) (cons '+ rest))
                                                 (list* '+ num rest)))
                                           (list* '+ a2 (cdr a1))))
                         (t (list '+ a2 a1))))
        ((and (sum? a1) (sum? a2))
         (cond ((and (numberp (addend a1)) (numberp (addend a2)))
                (list* '+ (+ (addend a1) (addend a2))
                       (append (cddr a1) (cddr a2))))
               ((numberp (addend a2)) (list* '+ (addend a2)
                                             (append (cdr a1) (cddr a2))))
               (t (append '(+) (cdr a1) (cdr a2)))))
        ((sum? a1) (append '(+) (cdr a1) (list a2)))
        ((sum? a2) (append '(+) (cdr a2) (list a1)))
        (t (list '+ a1 a2))
        ))
(defun make-product (m1 m2)
  (cond ((and (numberp m1) (numberp m2)) (* m1 m2))
        ((numberp m1)
         (cond ((zerop m1) 0.)
               ((= m1 1.) m2)
               ((product? m2) (if (numberp (multiplier m2))
                                  (let ((num (* m1 (multiplier m2)))
                                        (rest (cddr m2)))
                                    (if (= num 1.)
                                        (if (= (list-length rest) 1.)
                                            (first rest) (cons '* rest))
                                        (list* '* num rest)))
                                  (list* '* m1 (cdr m2))))
               (t (list '* m1 m2))))
        ((numberp m2)
         (cond ((zerop m2) 0.)
               ((= m2 1.) m1)
               ((product? m1) (if (numberp (multiplier m1))
                                  (let ((num (* m2 (multiplier m1)))
                                        (rest (cddr m1)))
                                    (if (= num 1.)
                                        (if (= (list-length rest) 1.)
                                            (first rest) (cons '* rest))
                                        (list* '* num rest)))
                                  (list* '* m2 (cdr m1))))
               (t (list '* m2 m1))))
        ((and (product? m1) (product? m2))
         (cond ((and (numberp (multiplier m1)) (numberp (multiplier m2)))
                (list* '* (* (multiplier m1) (multiplier m2))
                       (append (cddr m1) (cddr m2))))
               ((numberp (multiplier m2)) (list* '* (multiplier m2)
                                                 (append (cdr m1) (cddr m2))))
               (t (append '(*) (cdr m1) (cdr m2)))))
        ((product? m1) (append '(*) (cdr m1) (list m2)))
        ((product? m2) (append '(*) (cdr m2) (list m1)))
        (t (list '* m1 m2))))
;can use
;((exponential? b) (make-exponential (base b) (make-product (exponent b) e)))
;to suppress redundant expt's, but this may mess-up on negative bases on
;expressions like (n^2)^0.3
(defun make-exponential (b e)
  (cond ((and (numberp e) (zerop e)) 1.)
        ((and (numberp e) (= e 1.)) b)
        ((and (numberp b) (numberp e)) (expt b e))
        (t (list 'expt b e))))
;could use ((exponential? arg) (make-product (exponent arg) (make-log (base arg))))
;to convert log(b^e) into e*log(b), but if b<0 and e is even, will mess up (like in
;log(-1^2)).
(defun make-log (arg)
  (cond ((numberp arg) (log arg))
        ((to-the-e? arg) (arg arg))
        (t (list 'log arg))))

(defun sum? (x) (and (listp x) (eq (car x) '+)))
(defun difference? (x) (and (listp x) (eq (car x) '-)))
(defun addend (s) (cadr s))
;handles arbitrary args
(defun augend (s) (if (null (cdddr s)) (caddr s) (append '(+) (cddr s))))

(defun product? (x) (and (listp x) (eq (car x) '*)))
(defun quotient? (x) (and (listp x) (eq (car x) '/)))
(defun multiplier (p) (cadr p))
;handles arbitrary args
(defun multiplicand (p) (if (null (cdddr p)) (caddr p) (append '(*) (cddr p))))

(defun exponential? (x) (and (listp x) (eq (car x) 'expt) (= (list-length x) 3.)))
(defun base (exp) (cadr exp))
(defun exponent (exp) (caddr exp))

(defun arg (exp) (cadr exp)) ;for 1 argument functions

(defun logarithm? (x) (and (listp x) (eq (car x) 'log) (= (list-length x) 2.)))

(defun to-the-e? (x) (and (listp x) (eq (car x) 'exp) (= (list-length x) 2.)))
```

E.4 Bounder System

The AIS implementation uses an inequality reasoning system by Elisha Sacks. Loading the file `z:>elisha>qm>system.lisp` loads this system, and this should be done before loading or compiling AIS implementation (see Appendix D.1). The ideas behind this system are described in [36, 38]. Documentation on the inequality reasoning system implementation (which comes from the aforementioned file) is as follows:

Comments to Elisha Sacks, `eps@princeton.edu`.

This file loads an algebraic simplifier and an inequality reasoner into package QM. The interface to the simplifier consists of the following functions: `+`, `-`, `*`, `/`, `expt`, `exp`, `log`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `abs`, `min`, and `max`. The functions `+`, `*`, `min`, and `max` take 0 or more arguments, `-` and `/` take 1 or 2, `expt` takes 2, and all other functions take 1. All arguments are expressions: defined as 1) symbols other than T or NIL, 2) lisp numbers and 3) applications of the above functions to expressions. The symbols `:-infinity`, `:infinity`, `pi`, and `%e` have the standard meaning and are self-evaluating. All other symbols are uninterpreted. The function `(make-subsitution new old target)` substitutes into expressions.

The inequality reasoner maintains a set of contexts in which it records assertions and evaluates inequalities. The interface follows. The arguments a and b are expressions. The context argument is optional and defaults to the predefined context `*context*`.

1. `(clear context)` remove all assertions.
2. `(assert+ a strict? context)` assert $a > 0$. Return T if this is consistent with the previous contents of context; return NIL otherwise. Similarly, `assert-`, `assert0`, `(assert< a b context)`, `assert>`, `assert<=`,`assert>=`, and `assert=`.
3. `(< a b context)` returns T if $a < b$ is provable by simple methods, interval arithmetic and substitution. Similarly `>`,`<=`,`>=`. (see "Hierarchical Inequality proving" in AAAI-87 for details.)
4. `(sup exp var-list context)` returns an upper bound for exp in terms of the variables in var-list. Analogously, `inf` returns a lower bound.
5. `(test-sign exp context)` Returns 1,0,-1, or NIL for unknown. Uses simple methods.
6. `(<+ a b context)` returns T is $a < b$ is provable by slower methods, derivative inspection and iterative approximation. Similarly, `>+`, `<=+`, and `>=+`, `test-sign+`.

The following symbols are shadowed (in the QM package, they override the normal definitions): `pi + - * / exp expt log sin cos tan asin acos atan sinh cosh tanh abs signum min max < <= > >= numberp plusp minusp zerop floor ceiling rationalize`

E.5 File Qmfix.lisp

From the viewpoint of the AIS implementation, the inequality reasoning system implementation has some undesirable behavior when presented with an expression like $0 \cdot \infty$ or $\infty - \infty$ that has an ill-defined value: the inequality system returns an upper and lower bound of `nil` on such expressions, instead of returning the loosest bound of $-\infty$ for a lower bound and ∞ for an upper bound. The file below fixes that and should be loaded after the inequality system is loaded, but before the AIS implementation is loaded (see Appendix D.1).

```
;;; -*- Package: QM; Mode: LISP -*-
;fix Bounder bugs

;to fix bugs like (ub (* 0 :infinity)) -> nil, (lb (+ :-infinity :infinity)) -> nil
;  (ub (expt :infinity 0)) -> nil

(defun sup (exp var-set context &aux qform)
  (cond ((null exp) (values :infinity t)) ;line added
        ((subsetp (vars-of exp) var-set)
         (values exp nil))
        ((variablep exp)
         (var-sup exp var-set context))
        ((and (setq qform (quadraticp exp))
              (not (member (car qform) var-set)))
         (quadratic-sup qform var-set context))
        (t
         (let ((op (exp-op exp))
               (args (exp-args exp)))
           (case op
             (+ (+exp-sup args var-set context))
             (* (*exp-sup args var-set context))
             (expt (expt-sup (car args) (cadr args) var-set context))
             ((emin emax) (minmax-sup op args var-set context))
             (t (fun-sup op (car args) var-set context)))))))

(defun inf (exp var-set context &aux qform)
  (cond ((null exp) (values :-infinity t)) ;line added
        ((subsetp (vars-of exp) var-set)
         (values exp nil))
        ((variablep exp)
         (var-inf exp var-set context))
        ((and (setq qform (quadraticp exp))
              (not (member (car qform) var-set)))
         (quadratic-inf qform var-set context))
        (t
         (let ((op (exp-op exp))
               (args (exp-args exp)))
           (case op
             (+ (+exp-inf args var-set context))
             (* (*exp-inf args var-set context))
             (expt (expt-inf (car args) (cadr args) var-set context))
             ((emin emax) (minmax-inf op args var-set context))
             (t (fun-inf op (car args) var-set context)))))))
```

E.6 File: Cycle-ex.lisp

The file listed in this section contains examples run on the AIS implementation. The examples described in Sections 3.1, 3.2 and 3.3 were run by using the code to `setq` the variables `c2`, `c9` and `d`, respectively. Some of the parameter names in the thesis text have been shortened from the names they had in the actual code. For example, in the code for the ventricle examples, *Pi* was shortened from *Pin* in the actual input, *Po* from *Pout*, *Bi* from *Bin*, and *Bo* from *Bout*. In the code to `setq` the variable `c9`, the expression `,(deriv 'Vd2[...] 'Pin)` takes the total derivative of `Vd2[...]` with respect to `Pin`.

Unfortunately, many of the examples use an old style of input that is not described in Appendix D. Some documentation is given in Appendix E.1 in the code that starts with "`(defun new-phase-type`" (definition of the function `new-phase-type`) and the code that starts with "(defflavor cycle" (definition of the "flavor" or object type of cycle).

The file:

```
;;; -*- Mode: LISP; Package: CYCLE -*-
;EXAMPLES for cycle

(new-phase-type
  :name 'produce-seed-stage
  :condition '((<= 0 feed-per-adult) (<= 1 seed-per-adult) (<= 0 seed_b)
               (<= 0 seed_e) (<= 0 adults_b) (<= 0 adults_e) (<= 0 feed_c))
  :constant '(seed-per-adult feed-per-adult)
  :constant-periodic '()
  :varying-periodic-given '(seed)
  :varying-periodic-find '((adults 0))
  :constant-accum '()
  :varying-accum-given '()
  :varying-accum-find '((feed (* feed-per-adult adults_b))
                        (harvest (- (* seed-per-adult adults_b) seed_e)))
  )

(new-phase-type
  :name 'produce-adult-stage
  :condition '((<= 0 fraction-survived) (<= fraction-survived 1)
               (<= 0 seed_b) (<= 0 seed_e) (<= 0 adults_b) (<= 0 adults_e))
  :constant '(fraction-survived)
  :constant-periodic '()
  :varying-periodic-given '()
  :varying-periodic-find '((adults (* fraction-survived seed_b))
                           (seed 0))
  :constant-accum '(feed harvest)
  :varying-accum-given '()
  :varying-accum-find '()
  )

(setq a (make-instance
          'cycle
          :name 'chicken&egg :rate 'cycles/time
          :constants '(desired-egg-stock-level eggs/chicken food/chicken
                       fraction-maturing)
          :periodic-vars '(chickens eggs) :accum-vars '(chicken-feed eggs-sold)
          :conditions '((< 0 desired-egg-stock-level) (<= 1 eggs/chicken)
                        (< 0 food/chicken) (< 0 fraction-maturing)
                        (< fraction-maturing 1))
          :phases
          '((lay&harvest-eggs produce-seed-stage
             ((adults chickens) (seed eggs desired-egg-stock-level)
              (harvest eggs-sold) (feed chicken-feed)
              (seed-per-adult eggs/chicken) (feed-per-adult food/chicken)))
            (hatch-eggs produce-adult-stage
             ((adults chickens) (seed eggs) (harvest eggs-sold)
              (feed chicken-feed) (fraction-survived fraction-maturing)))
            )))
```

```
(setq b (make-instance
          'cycle
          :name 'maize :rate 'once-per-year?
          :constants '(desired-seed-corn-level grain/plant fertilizer/plant
                                               fraction-maturing)
          :periodic-vars '(grain plants) :accum-vars '(fertilizer grain-sold)
          :conditions '((< 0 desired-seed-corn-level) (<= 1 grain/plant)
                        (< 0 fertilizer/plant) (< 0 fraction-maturing)
                        (< fraction-maturing 1))
          :phases
          '((harvest-corn produce-seed-stage
             ((adults plants) (seed grain desired-seed-corn-level)
              (harvest grain-sold)(feed fertilizer)
              (seed-per-adult grain/plant)(feed-per-adult fertilizer/plant)))
            (plant-corn produce-adult-stage
             ((adults plants) (seed grain) (harvest grain-sold) (feed fertilizer)
              (fraction-survived fraction-maturing)))
            )))

(new-phase-type
  :name 'pump-incompressible-stuff-at-constant-p
  :condition '((<= 0 V_b) (<= 0 V_e))
  :constant '()
  :constant-periodic '(P)
  :varying-periodic-given '(V)
  :varying-periodic-find '()
  :constant-accum '()
  :varying-accum-given '()
  :varying-accum-find '((Work-by-stuff (* P Amount-in_c))
                        (Amount-in (- V_e V_b)))
  )

(new-phase-type
  :name 'pump-incompressible-stuff-at-constant-p2
  :condition '((<= 0 V_b) (<= 0 V_e))
  :constant '()
  :constant-periodic '(P)
  :varying-periodic-given '(V)
  :varying-periodic-find '()
  :constant-accum '()
  :varying-accum-given '()
  :varying-accum-find '((Work-by-stuff (* -1. P Amount-out_c))
                        (Amount-out (- V_b V_e)))
  )

(new-phase-type
  :name 'alter-pressure-of-incompressible-stuff-at-constant-v
  :condition '((<= 0 V))
  :constant '()
  :constant-periodic '(V)
  :varying-periodic-given '(P)
  :varying-periodic-find '()
  :constant-accum '(Work-by-stuff Amount-in)
  :varying-accum-given '()
  :varying-accum-find '()
  )

;assume that the heart beat is slow enough to get a full contraction & relaxation
;(period > 0.4 sec, HR < 2.5hz = 180 beats/min)

(setq c (make-instance 'cycle
          :name 'heart-ventricle
          :rate 'HR
          :constants '(Pin Pout Vd[Pin] Vs[Pout])
          :periodic-vars '(P-ventricle V-ventricle)
          :accum-vars '(Work-by-blood amount-of-blood-in amount-of-blood-out)
          :conditions
          '((< Pin Pout) (> Vd[Pin] Vs[Pout]) (<= 0 Vd[Pin]) (< 0 %Vd[Pin])
            (> 0 %%Vd[Pin]) (<= 0 Vs[Pout]) (< 0 %Vs[Pout]) (< 0 %%Vs[Pout]))
          :phases
          '((iso-volumetric-contraction
              alter-pressure-of-incompressible-stuff-at-constant-v
             ((V V-ventricle) (P P-ventricle Pout) (Work-by-stuff Work-by-blood)
              (Amount-in amount-of-blood-out)))
            (ejection pump-incompressible-stuff-at-constant-p2
             ((P P-ventricle) (V V-ventricle Vs[Pout])
              (Work-by-stuff Work-by-blood) (Amount-out amount-of-blood-out)))
            (iso-volumetric-relaxation
              alter-pressure-of-incompressible-stuff-at-constant-v
             ((V V-ventricle) (P P-ventricle Pin) (Work-by-stuff Work-by-blood)
              (Amount-in amount-of-blood-in)))
            (filling pump-incompressible-stuff-at-constant-p
             ((P P-ventricle) (V V-ventricle Vd[Pin])
              (Work-by-stuff Work-by-blood) (Amount-in amount-of-blood-in)))
            )))

(setq c1 (make-instance ;include positive inotropic effect of increasing HR
          'cycle
          :name 'heart-ventricle-with-HR-affecting-Vs
          :rate 'HR :constants '(Pin Pout Vd[Pin] Vs[Pout$HR])
          :periodic-vars '(P-ventricle V-ventricle)
          :accum-vars '(Work-by-blood amount-of-blood-in amount-of-blood-out)
          :conditions
          '((< Pin Pout) (> Vd[Pin] Vs[Pout$HR]) (<= 0 Vd[Pin])
            (< 0 %Vd[Pin]) (> 0 %%Vd[Pin]) (<= 0 Vs[Pout$HR]) (< 0 %Vs[Pout$HR])
            (< 0 %%Vs[Pout$HR]) (> 0 !%Vs[Pout$HR]))
          :phases
          '((iso-volumetric-contraction
              alter-pressure-of-incompressible-stuff-at-constant-v
            ((V V-ventricle) (P P-ventricle Pout) (Work-by-stuff Work-by-blood)
             (Amount-in amount-of-blood-out)))
            (ejection pump-incompressible-stuff-at-constant-p2
             ((P P-ventricle) (V V-ventricle Vs[Pout$HR])
              (Work-by-stuff Work-by-blood) (Amount-out amount-of-blood-out)))
            (iso-volumetric-relaxation
              alter-pressure-of-incompressible-stuff-at-constant-v
             ((V V-ventricle) (P P-ventricle Pin) (Work-by-stuff Work-by-blood)
              (Amount-in amount-of-blood-in)))
              (filling pump-incompressible-stuff-at-constant-p
               ((P P-ventricle) (V V-ventricle Vd[Pin])
                (Work-by-stuff Work-by-blood) (Amount-in amount-of-blood-in)))
              )))

(setq c2 (make-instance ;version without phase types
           'cycle
           :name 'heart-ventricle-with-HR-affecting-Vs
           :rate 'HR :constants '(Pin Pout Vd[Pin] Vs[Pout$HR])
           :periodic-vars '(P V)
           :accum-vars '(W Bin Bout)
           :conditions
           '((< Pin Pout) (> Vd[Pin] Vs[Pout$HR]) (<= 0 Vd[Pin]) (< 0 %Vd[Pin])
             (> 0 %%Vd[Pin]) (<= 0 Vs[Pout$HR]) (< 0 %Vs[Pout$HR])
             (< 0 %%Vs[Pout$HR]) (> 0 !%Vs[Pout$HR]))
           :phases
           '((iso-volumetric-contraction nil ((<= 0 V) (:= P_e Pout)))
             (ejection nil ((<= 0 V_b) (<= 0 V_e) (:= V_e Vs[Pout$HR])
                            (:= W_c (* -1. P Bout_c)) (:= Bout_c (- V_b V_e))))
             (iso-volumetric-relaxation nil ((<= 0 V) (:= P_e Pin)))
             (filling nil ((<= 0 V_b) (<= 0 V_e) (:= V_e Vd[Pin])
                           (:= W_c (* P Bin_c)) (:= Bin_c (- V_e V_b))))
             )))

;time stuff messed-up
(setq c4
  (make-instance ;CT is for contraction time
    'cycle
    :name 'left-heart-ventricle-with-mitral-stenosis
    :rate 'HR
    :constants
    '(Pin Pout Vd[Pin] Vs[Pout$HR] CT T0[Pin$Pout] T2[Pin$Pout]
      Vd^-1[Vs[Pout$HR]]
      Vd2[Vs[Pout$HR]$Vd[Pin]$-[/[HR]$+[CT$T2[Pin$Pout]]]$Pin]
      Vd^-1[Vd2[Vs[Pout$HR]$Vd[Pin]$-[/[HR]$+[CT$T2[Pin$Pout]]]$Pin]])
    :periodic-vars '(P V)
    :accum-vars '(Bin Bout T)
    :conditions
    '((< Pin Pout) (> Vd[Pin] Vs[Pout$HR]) (<= 0 Vd[Pin]) (< 0 %Vd[Pin])
      (> 0 %%Vd[Pin]) (<= 0 Vs[Pout$HR]) (< 0 %Vs[Pout$HR])
      (< 0 %%Vs[Pout$HR]) (> 0 !%Vs[Pout$HR])
      (<= 0 T0[Pin$Pout]) (< %T0[Pin$Pout] 0) (< 0 !%T0[Pin$Pout])
      (<= 0 T2[Pin$Pout]) (< %T2[Pin$Pout] 0) (< 0 !%T2[Pin$Pout])
      (<= 0 %Vd2[Vs[Pout$HR]$Vd[Pin]$-[/[HR]$+[CT$T2[Pin$Pout]]]$Pin])
      (<= 0 !%Vd2[Vs[Pout$HR]$Vd[Pin]$-[/[HR]$+[CT$T2[Pin$Pout]]]$Pin])
      (<= 0 !!%Vd2[Vs[Pout$HR]$Vd[Pin]$-[/[HR]$+[CT$T2[Pin$Pout]]]$Pin])
      (<= 0 !!!%Vd2[Vs[Pout$HR]$Vd[Pin]$-[/[HR]$+[CT$T2[Pin$Pout]]]$Pin])
      (< Vd^-1[Vs[Pout$HR]] Pin)
      (< 0 %Vd^-1[Vs[Pout$HR]]) (< 0 %%Vd^-1[Vs[Pout$HR]])
      (< 0 %Vd^-1[Vd2[Vs[Pout$HR]$Vd[Pin]$-[/[HR]$+[CT$T2[Pin$Pout]]]$Pin]])
      (< 0 %%Vd^-1[Vd2[Vs[Pout$HR]$Vd[Pin]$-[/[HR]$+[CT$T2[Pin$Pout]]]$Pin]])
      )
    :phases
    '((iso-volumetric-contraction nil ((<= 0 V) (:= P_e Pout)
                                    (<= 0 T_c) (:= T_c TO[Pin$Pout])))
   (ejection nil
     ((<= 0 V_b) (<= 0 V_e) (:= V_e Vs[Pout$HR]) (:= Bout_c (- V_b V_e))
      (<= 0 T_c) (:= T_c (- CT TO[Pin$Pout]))))
   (iso-volumetric-relaxation nil ((<= 0 V) (:= P_e Pin)
                                   (<= 0 T_c) (:= T_c T2[Pin$Pout])))
   ;P may not monotonically dec/increase while filling
   ;(probably down then back-up)
   (filling nil
     ((<= 0 V_b) (<= 0 V_e) (:= Bin_c (- V_e V_b))
      (<= 0 T_c) (:= T_c (- (/ HR) (+ CT T2[Pin$Pout])))
      (<= V_b V_e) (<= V_e Vd[Pin])
      (:= V_e Vd2[Vs[Pout$HR]$Vd[Pin]$-[/[HR]$+[CT$T2[Pin$Pout]]]$Pin])
      (<= Vd^-1[Vs[Pout$HR]] P_e) (<= P_e Pin)
      (:= P_e Vd^-1[Vd2[Vs[Pout$HR]$Vd[Pin]$-[/[HR]$+[CT$T2[Pin$Pout]]]$Pin]])
       )))))

;Derivatives of Tf now unknown (which is true). 1st TOTAL derivative of Vd2 wrt to
;Pi is >0
(Setq c9
  (make-instance
    ;TF is time in filling phase, TR is time to go from fully contracted to
    ;fully relaxed (HR may be either too fast for ventricle to fully relax, or
    ;so slow that full relaxtion is reached before the end of the beat cycle),
    ;TC is time to go from full relaxation to full contraction (HR may be too
    ;fast for full contraction to occur)
    'cycle
    :name 'l-heart-ventricle-w/mitral-stenosis-&-asymptotic-filling-&-new-time
    :rate 'HR
    :constants
    '(Pin Pout Vd[Pin] Vs[Pout$HR]
      Vd^-1[Vs[Pout$HR]] TR[HR] TC[HR] TF[Pin$Pout$HR$TC[HR]$TR[HR]]
      Vd2[Vs[Pout$HR]$Vd[Pin]$TF[Pin$Pout$HR$TC[HR]$TR[HR]]$TR[HR]$Pin]
      Vd^-1[Vd2[Vs[Pout$HR]$Vd[Pin]$TF[Pin$Pout$HR$TC[HR]$TR[HR]]$TR[HR]$Pin]])
    :periodic-vars '(P V)
    :accum-vars '(Bin Bout)
    :conditions
    '((< Pin Pout) (> Vd[Pin] Vs[Pout$HR]) (<= 0 Vd[Pin]) (< 0 %Vd[Pin])
      (> 0 %%Vd[Pin]) (<= 0 Vs[Pout$HR]) (< 0 %Vs[Pout$HR])
      (< 0 %%Vs[Pout$HR]) (> 0 !%Vs[Pout$HR])
      (<= 0 TR[HR]) (> 0 %TR[HR]) (<= 0 TC[HR]) (> 0 %TC[HR])
      (<= 0 TF[Pin$Pout$HR$TC[HR]$TR[HR]])
      (<= TF[Pin$Pout$HR$TC[HR]$TR[HR]] (/ HR))
      ;all derivatives of TF are unknown
      ;(< 0 %TF[Pin$Pout$HR$TC[HR]$TR[HR]])
      ;(> 0 !%TF[Pin$Pout$HR$TC[HR]$TR[HR]])
      ;(> 0 !!%TF[Pin$Pout$HR$TC[HR]$TR[HR]])
      ;(> 0 !!!%TF[Pin$Pout$HR$TC[HR]$TR[HR]])
      ;(> 0 !!!!%TF[Pin$Pout$HR$TC[HR]$TR[HR]])
      (< 0 %Vd2[Vs[Pout$HR]$Vd[Pin]$TF[Pin$Pout$HR$TC[HR]$TR[HR]]$TR[HR]$Pin])
      (< 0 !%Vd2[Vs[Pout$HR]$Vd[Pin]$TF[Pin$Pout$HR$TC[HR]$TR[HR]]$TR[HR]$Pin])
      (< 0 !!%Vd2[Vs[Pout$HR]$Vd[Pin]$TF[Pin$Pout$HR$TC[HR]$TR[HR]]$TR[HR]$Pin])
      (> 0
         !!!%Vd2[Vs[Pout$HR]$Vd[Pin]$TF[Pin$Pout$HR$TC[HR]$TR[HR]]$TR[HR]$Pin])
```

```
        (< 0
           !!!!%Vd2[Vs[Pout$HR]$Vd[Pin]$TF[Pin$Pout$HR$TC[HR]$TR[HR]]$TR[HR]$Pin])
        (< 0 ,(deriv
                '\Vd2[Vs[Pout$HR]$Vd[Pin]$TF[Pin$Pout$HR$TC[HR]$TR[HR]]$TR[HR]$Pin]
                'Pin))
        (< Vd^-1[Vs[Pout$HR]] Pin)
        (< 0 %Vd^-1[Vs[Pout$HR]]) (< 0 %%Vd^-1[Vs[Pout$HR]])
        (<
          0
          %Vd^-1[Vd2[Vs[Pout$HR]$Vd[Pin]$TF[Pin$Pout$HR$TC[HR]$TR[HR]]$TR[HR]$Pin]])
        (<
          0
          %%Vd^-1[Vd2[Vs[Pout$HR]$Vd[Pin]$TF[Pin$Pout$HR$TC[HR]$TR[HR]]$TR[HR]$Pin]])
        )
      :phases
      '((iso-volumetric-contraction nil ((<= 0 V) (:= P_e Pout)))
        (ejection nil
          ((<= 0 V_b) (<= 0 V_e) (:= V_e Vs[Pout$HR]) (:= Bout_c (- V_b V_e))))
        (iso-volumetric-relaxation nil ((<= 0 V) (:= P_e Pin)))
        ;P may not monotonically dec/increase while filling
        ;(probably down then back-up)
        (filling nil
         ((<= 0 V_b) (<= 0 V_e) (:= Bin_c (- V_e V_b))
          (< V_b V_e) (< V_e Vd[Pin])
          (:= V_e
           Vd2[Vs[Pout$HR]$Vd[Pin]$TF[Pin$Pout$HR$TC[HR]$TR[HR]]$TR[HR]$Pin])
          (< Vd^-1[Vs[Pout$HR]] P_e) (< P_e Pin)
          (:= P_e
           Vd^-1[Vd2[Vs[Pout$HR]$Vd[Pin]$TF[Pin$Pout$HR$TC[HR]$TR[HR]]$TR[HR]$Pin]]
           ))))))

(new-phase-type
  :name 'pump-in-ideal-gas-at-constant-v
  :condition '((<= P_b P_e) (<= 3/2 k) (< 0 R) (< 0 Tin) (<= 0 P_b) (<= 0 P_e)
               (<= 0 V) (<= 0 n_b) (<= 0 n_e) (< 0 T_b) (< 0 T_e) (<= 0 Ein_c)
               (<= 0 Ain_c))
  :constant '(k R Tin)
  :constant-periodic '(V)
  :varying-periodic-given '(P)
  :varying-periodic-find '((n (+ n_b Ain_c))
                           (T (/ (+ (* n_b T_b) (* Ain_c Tin)) n_e)))
  :constant-accum '(W-on-device)
  :varying-accum-given '()
  :varying-accum-find '((Ein (* k (- P_e P_b) V))
                        (Ain (/ (* (- P_e P_b) V) (* R Tin))))
  )

(new-phase-type
  :name 'pump-out-ideal-gas-at-constant-v
  :condition '((>= P_b P_e) (<= 3/2 k) (< 0 R) (< 0 T) (<= 0 P_b) (<= 0 P_e)
               (<= 0 V) (<= 0 n_b) (<= 0 n_e) (< 0 T) (<= 0 Eout_c)
               (<= 0 Aout_c))
  :constant '(k R)
  :constant-periodic '(V T)
```

```
   :varying-periodic-given '(P)
   :varying-periodic-find '((n (/ (* P_e V) (* R T))))
   :constant-accum '(W-on-device)
   :varying-accum-given '()
   :varying-accum-find '((Eout (* k (- P_b P_e) V))
                         (Aout (- n_b n_e)))
   )

 (new-phase-type
   :name 'pump-in-ideal-gas-at-constant-p
   :condition '((<= V_b V_e) (<= 3/2 k) (< 0 R) (< 0 Tin) (<= 0 P) (<= 0 V_b)
                (<= 0 V_e) (<= 0 n_b) (<= 0 n_e) (< 0 T_b) (< 0 T_e)
                (<= 0 W-on-device_c) (<= 0 Ein_c) (<= 0 Ain_c))
   :constant '(k R Tin)
   :constant-periodic '(P)
   :varying-periodic-given '(V)
   :varying-periodic-find '((n (+ n_b Ain_c))
                            (T (/ (+ (* n_b T_b) (* Ain_c Tin)) n_e)))
   :constant-accum '()
   :varying-accum-given '()
   :varying-accum-find '((W-on-device (* P (- V_e V_b)))
                         (Ein (* (+ 1. k) W-on-device_c))
                         (Ain (/ W-on-device_c (* R Tin))))
   )

 (new-phase-type
   :name 'pump-out-ideal-gas-at-constant-p
   :condition '((>= V_b V_e) (<= 3/2 k) (< 0 R) (<= 0 P) (<= 0 V_b) (<= 0 V_e)
                (<= 0 n_b) (<= 0 n_e) (< 0 T) (>= 0 W-on-device_c) (<= 0 Eout_c)
                (<= 0 Aout_c))
   :constant '(k R)
   :constant-periodic '(P T)
   :varying-periodic-given '(V)
   :varying-periodic-find '((n (/ (* P V_e) (* R T))))
   :constant-accum '()
   :varying-accum-given '()
   :varying-accum-find '((W-on-device (* P (- V_e V_b)))
                         (Eout (* (+ 1. k) P (- V_b V_e)))
                         (Aout (- n_b n_e)))
   )

 (new-phase-type
   :name 'adiabatic-expand-ideal-gas
   :condition '((<= 3/2 k) (< 0 R) (<= 0 n) (<= 0 V_b) (<= 0 V_e) (< 0 T_b)
                (< 0 T_e) (<= 0 P_b) (<= 0 P_e))
   :constant '(k R)
   :constant-periodic '(n)
   :varying-periodic-given '(V)
   :varying-periodic-find '((T (* T_b (expt (/ V_b V_e) (/ k))))
                            (P (* P_b (expt (/ V_b V_e) (+ 1. (/ k))))))
   :constant-accum '(Ein Eout Ain Aout)
   :varying-accum-given '()
   :varying-accum-find
   '((W-on-device (* k P_b V_b (- 1. (expt (/ V_b V_e) (/ k))))))
```

```
    )

  ;because they have no reference to an external constant, T, N, & amount-out stay
  ;unsolved
  (setq
    d
    (make-instance
      'cycle
      :name 'single-acting-steam-piston :rate 'RPM
      :constants '(Pin Pout V1 Vex Vcomp Vh Tin R k-steam)
      :periodic-vars '(P-chamber V-chamber n-chamber T-chamber)
      :accum-vars '(W-on-piston E-of-steam-in E-of-steam-out amount-of-steam-in
                              amount-of-steam-out)
      :conditions
      '((< 0 Pout) (> Pin Pout) (< 0 V1) (< V1 Vex) (< Vex Vh) (< V1 Vcomp)
        (< Vcomp Vh) (< 0 Tin) (< 0 R) (<= 3/2 k-steam)) ;many others
      :phases
      '((open-admission pump-in-ideal-gas-at-constant-v
          ((k k-steam) (R R) (Tin Tin) (P P-chamber Pin) (V V-chamber) (n n-chamber)
           (T T-chamber) (W-on-device W-on-piston) (Ein E-of-steam-in)
           (Ain amount-of-steam-in)))
        (steam-admission pump-in-ideal-gas-at-constant-p
          ((k k-steam) (R R) (Tin Tin) (P P-chamber) (V V-chamber Vex) (n n-chamber)
           (T T-chamber) (W-on-device W-on-piston) (Ein E-of-steam-in)
           (Ain amount-of-steam-in)))
        (steam-expansion adiabatic-expand-ideal-gas
          ((k k-steam) (R R) (P P-chamber) (V V-chamber Vh) (n n-chamber)
           (T T-chamber) (W-on-device W-on-piston) (Ein E-of-steam-in)
           (Eout E-of-steam-out) (Ain amount-of-steam-in) (Aout amount-of-steam-out)))
        (open-exhaust pump-out-ideal-gas-at-constant-v
          ((k k-steam) (R R) (P P-chamber Pout) (V V-chamber) (n n-chamber)
           (T T-chamber) (W-on-device W-on-piston) (Eout E-of-steam-out)
           (Aout amount-of-steam-out)))
        (exhaust pump-out-ideal-gas-at-constant-p
          ((k k-steam) (R R) (P P-chamber) (V V-chamber Vcomp) (n n-chamber)
           (T T-chamber) (W-on-device W-on-piston) (Eout E-of-steam-out)
           (Aout amount-of-steam-out)))
        (steam-compression adiabatic-expand-ideal-gas
          ((k k-steam) (R R) (P P-chamber) (V V-chamber V1) (n n-chamber)
           (T T-chamber) (W-on-device W-on-piston) (Ein E-of-steam-in)
           (Eout E-of-steam-out) (Ain amount-of-steam-in) (Aout amount-of-steam-out)))
        )))

;add (< 0 Pout) ?
(setq e
  (make-instance
    'cycle
    :name 'single-acting-steam-piston-no-expand/compress
    :rate 'RPM :constants '(Pin Pout V1 Vh Tin R k-steam)
    :periodic-vars '(P-chamber V-chamber n-chamber T-chamber)
    :accum-vars '(W-on-piston E-of-steam-in E-of-steam-out amount-of-steam-in
                            amount-of-steam-out)
    :conditions  ;many others
    '((> Pin Pout) (< 0 V1) (< V1 Vh) (< 0 Tin) (< 0 R) (<= 3/2 k-steam))
:phases
'((open-admission pump-in-ideal-gas-at-constant-v
   ((k k-steam) (R R) (Tin Tin) (P P-chamber Pin) (V V-chamber) (n n-chamber)
    (T T-chamber) (W-on-device W-on-piston) (Ein E-of-steam-in)
    (Ain amount-of-steam-in)))
  (steam-admission pump-in-ideal-gas-at-constant-p
   ((k k-steam) (R R) (Tin Tin) (P P-chamber) (V V-chamber Vh) (n n-chamber)
    (T T-chamber) (W-on-device W-on-piston) (Ein E-of-steam-in)
    (Ain amount-of-steam-in)))
  (open-exhaust pump-out-ideal-gas-at-constant-v
   ((k k-steam) (R R) (P P-chamber Pout) (V V-chamber) (n n-chamber)
    (T T-chamber) (W-on-device W-on-piston) (Eout E-of-steam-out)
    (Aout amount-of-steam-out)))
  (exhaust pump-out-ideal-gas-at-constant-p
   ((k k-steam) (R R) (P P-chamber) (V V-chamber Vl) (n n-chamber)
    (T T-chamber) (W-on-device W-on-piston) (Eout E-of-steam-out)
    (Aout amount-of-steam-out)))
  )))
```

Bibliography

[1] Sanjaya Addanki, Roberto Cremonini, and J. Scott Penberthy. Reasoning about assumptions in graphs of models. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 1432–1438, August 1989.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffery D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, 1974.

[3] J. Aitchison and J. Brown. *The Lognormal Distribution.* Cambridge at the University Press, Great Britain, 1957.

[4] M. Avriel, editor. *Advances in Geometric Programming.* Plenum Press, New York, 1980.

[5] Oyvind Bjőrke. *Computer-Aided Tolerancing.* Tapir Publishers, Norwegian Institute of Technology, N-7034, Trondheim, Norway, 1978.

[6] Daniel G. Bobrow, editor. *Qualitative Reasoning about Physical Systems.* MIT Press, 1985. Reprinted from *Artificial Intelligence*, vol. 24, 1984.

[7] Eugene Braunwald, editor. *Heart Disease.* W. B. Saunders Co., Philadelphia, third edition, 1988.

[8] Terrell Croft, editor. *Steam-Engine Principles and Practice.* McGraw-Hill Book Co., Inc., New York, 2nd edition, 1939. Revised by E. J. Tangerman.

[9] Wilbur Davenport, Jr. *Probability and Random Processes.* McGraw-Hill Book Co., 1970.

[10] Morris H. DeGroot. *Probability and Statistics, 2nd ed.* Addison-Wesley Publishing Company, 1986.

[11] J. E. Dennis, Jr. and Robert B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations.* Prentice-Hall, Inc., 1983.

[12] Brian Falkenhainer and Kenneth Forbus. Setting up large-scale qualitative models. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 301–306, August 1988.

[13] Kenneth D. Forbus. Intelligent computer-aided engineering. *AI Magazine*, 9(3):23–36, 1988.

[14] G. Giuffrida, G. Bonzani, S. Betocchi, F. Piscione, P. Giudice, D. Miceli, F. Mazza, and M. Condorelli. Hemodynamic response to exercise after propranolol in patients with mitral stenosis. *The American Journal of Cardiology*, 44:1076–1082, November 1979.

[15] C. V. Greenway. Mechanisms and quantitative assessment of drug effects on cardiac output with a new model of the circulation. *Pharmacological Reviews*, 33(4), 1982.

[16] G. Hahn and S. Shapiro. *Statistical Models in Engineering.* John Wiley & Sons, Inc., 1967.

[17] David Halliday and Robert Resnick. *Physics.* John Wiley and Sons, Inc., New York, 1960.

[18] John H. Halton. A retrospective and prospective survey of the monte carlo method. *SIAM Review*, 12(1):1–63, 1970.

[19] J. M. Hammersley and D. C. Handscomb. *Monte Carlo Methods.* Barnes and Noble, Inc., 1965.

[20] Eric J. Horvitz, David E. Heckerman, and Curtis P. Langlotz. A framework for comparing alternative formalisms for plausible reasoning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 210–214. American Association for Artificial Intelligence, 1986.

[21] Ronald L. Iman and W. J. Conover. A distribution-free approach to inducing rank correlation among input variables. *Commun. Statist.-Simula. Computa.*, 11(3):311–334, 1982.

[22] Mark Johnson. *Multivariate Statistical Simulation.* John Wiley and Sons, New York, 1987.

[23] Jan Koch-Weser. Effect of rate changes on strength and time course of contraction of papillary muscle. *American Journal of Physiology*, 204(3):451–457, 1962.

[24] Veng-Kin Lau and Kiichi Sagawa. Model analysis of the contribution of atrial contraction to ventricular filling. *Annals of Biomedical Engineering*, 7:167–201, 1979.

[25] W. J. Long, S. Naimi, M. G. Criscitiello, and R. Jayes. Using a physiological model for prediction of therapy effects in heart disease. In *Proc. of the Computers in Cardiology Conf.* IEEE, October 1986.

[26] William J. Long, Shapur Naimi, M. G. Criscitiello, and Robert Jayes. The development and use of a causal model for reasoning about heart failure. In *Symposium on Computer Applications in Medical Care*, pages 30–36. IEEE, November 1987.

[27] Dean T. Mason. *Congestive Heart Failure, Mechanisms, Evaluation, and Treatment.* Dun-Donnelley, New York, 1976.

[28] Seshashayee Murthy and Sanjaya Addanki. Prompt: An innovative design tool. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 637–642, 1987.

[29] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, San Mateo, CA, 1988.

[30] Charles S. Peskin and Cheng Tu. Hemodynamics in congenital heart disease. *Comput. Biol. Med.*, 16(5):331–359, 1986.

[31] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes.* Cambridge University Press, Cambridge, England, 1987.

[32] Olivier Raiman. Order of magnitude reasoning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 100–104. American Association for Artificial Intelligence, 1986.

[33] H. Ratschek and J. Rokne. *Computer Methods for the Range of Functions.* Halsted Press: a division of John Wiley and Sons, New York, 1984.

[34] D. Richardson. Some unsolvable problems involving elementary functions of a real variable. *Journal of Symbolic Logic*, 33:511–520, 1968.

[35] John Ross. Cardiovascular system (sec. 2). In *Best and Taylor's Physiological Basis of Medical Practice.* Williams and Wilkins, Baltimore, eleventh edition, 1985.

[36] Elisha P. Sacks. Hierarchical reasoning about inequalities. In *Proceedings of the National Conference on Artificial Intelligence*, pages 649–654. American Association for Artificial Intelligence, 1987.

[37] Elisha P. Sacks. Qualitative sketching of parameterized functions. In D. Sriram and R. A. Adey, editors, *Knowledge Based Expert Systems for Engineering: Classification, Education and Control*, pages 1–13. Computational Mechanics Publications, Boston, 1987.

[38] Elisha P. Sacks. Automatic qualitative analysis of ordinary differential equations using piecewise linear approximations. TR 416, Massachusetts Institute of Technology, Laboratory for Computer Science, 545 Technology Square, Cambridge, MA, 02139, March 1988. Also appears as AI-TR-1031.

[39] Elisha P. Sacks. Automatic qualitative analysis of ordinary differential equations using piecewise linear approximations. *Artificial Intelligence*, 41(3):313–364, January 1990.

[40] Kiichi Sagawa. Analysis of the ventricular pumping capacity as a function of input and output pressure loads. In E. Reeve and A. Guyton, editors, *Physical Bases of Circulatory Transport: Regulation and Exchange*, chapter 9. W. B. Sanders Co., Philadephia, 1967.

[41] T. Sato, A. Takeuchi, J. Yamagami, H. Yamamoto, S. Akiyama, K. Endou, M. Shirataka, N. Ikeda, and H. Tsuruta. Computer assisted instruction for therapy of heart failure based on simulation of cardiovascular system. In R. Salamon, B. Blum, and M. Jørgensen, editors, *MEDINFO 86: Proceedings of the Fifth Conference on Medical Informatics*, pages 761–765, Washington, October 1986. North-Holland.

[42] Ross D. Shachter. A linear approximation method for probabilistic inference. In R. Shachter, T.S. Levitt, L.N. Kanal, and J. Lemmer, editors, *Uncertainty in Artificial Intelligence 4*, pages 93–103. Elsevier Science Publishers B.V. (North-Holland), 1990. Revised version of the paper in The Fourth Workshop on Uncertainty in Artificial Intelligence (1988).

[43] Ross D. Shachter, David M. Eddy, Vic Hasselblad, and Robert Wolpert. A heuristic bayesian approach to knowledge acquisition: Application to analysis of tissue-type plasminogen activator. In *Third Workshop on Uncertainty in Artificial Intelligence*, pages 229–236, July 1987.

[44] William C. Shoemaker. Physiology, monitoring and therapy of critically ill general surgical patients. In William C. Shoemaker and Edward Abraham, editors, *Diagnostic Methods in Critical Care*, chapter 4, pages 47–86. Marcel Dekker, Inc., New York, 1987.

[45] Robert Spence and Randeep Singh Soin. *Tolerance Design of Electronic Circuits.* Addison-Wesley Publishing Co., Reading, MA, 1988.

[46] Guy L. Steele Jr. *Common LISP.* Digital Press, 1984.

[47] A. H. Stroud. *Approximate Calculation of Multiple Integrals.* Prentice-Hall, Inc., 1971.

[48] Peter Szolovits and Stephen G. Pauker. Categorical and probabilistic reasoning in medical diagnosis. *Artificial Intelligence*, 11:115–144, 1978.

[49] G. Thomas, Jr. *Calculus and Analytic Geometry, 4th edition.* Addison-Wesley Publishing Co., 1968.

[50] Daniel S. Weld. The use of aggregation in causal simulation. *Artificial Intelligence*, 30(1):1–34, October 1986.

[51] Daniel S. Weld. Comparative analysis. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 959–965, 1987.

[52] Daniel S. Weld. Exaggeration. In *Proceedings of the National Conference on Artificial Intelligence.* American Association for Artificial Intelligence, 1988.

[53] Daniel S. Weld. Theories of comparative analysis. AI-TR 1035, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 545 Technology Square, Cambridge, MA, 02139, May 1988.

[54] Brian C. Williams. MINIMA: A symbolic approach to qualitative algebraic reasoning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 264–269. American Association for Artificial Intelligence, August 1988.

[55] Patrick Henry Winston. *Artificial Intelligence.* Addison-Wesley, Reading, MA, second edition, 1984.