

# Tools for Modelling and Identification with Bond Graphs and Genetic Programming

by

Stefan Wiechula

A thesis  
presented to the University of Waterloo  
in fulfilment of the  
thesis requirement for the degree of  
Master of Applied Science  
in  
Mechanical Engineering

Waterloo, Ontario, Canada, 2006

©Stefan Wiechula 2006

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

# Abstract

The contributions of this work include genetic programming grammars for bond graph modelling and for direct symbolic regression of sets of differential equations; a bond graph modelling library suitable for programmatic use; a symbolic algebra library specialized to this use and capable of, among other things, breaking algebraic loops in equation sets extracted from linear bond graph models. Several non-linear multi-body mechanics examples are presented, showing that the bond graph modelling library exhibits well-behaved simulation results. Symbolic equations in a reduced form are produced automatically from bond graph models. The genetic programming system is tested against a static non-linear function identification problem using typeless symbolic regression. The direct symbolic regression grammar is shown to have a non-deceptive fitness landscape: perturbations of an exact program have decreasing fitness with increasing distance from the ideal. The planned integration of bond graphs with genetic programming for use as a system identification technique was not successfully completed. A categorized overview of other modelling and identification techniques is included as context for the choice of bond graphs and genetic programming.

# Acknowledgements

I'd like to thank my supervisor, Dr. Jan Huissoon, for his support.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	System identification . . . . .	1
1.2	Black box and grey box problems . . . . .	3
1.3	Static and dynamic models . . . . .	6
1.4	Parametric and non-parametric models . . . . .	8
1.5	Linear and nonlinear models . . . . .	9
1.6	Optimization, search, machine learning . . . . .	11
1.7	Some model types and identification techniques . . . . .	12
1.7.1	Locally linear models . . . . .	13
1.7.2	Nonparametric modelling . . . . .	14
1.7.3	Volterra function series models . . . . .	15
1.7.4	Two special models in terms of Volterra series . . . . .	16
1.7.5	Least squares identification of Volterra series models . . . . .	18
1.7.6	Nonparametric modelling as function approximation . . . . .	21
1.7.7	Neural networks . . . . .	22
1.7.8	Parametric modelling . . . . .	23
1.7.9	Differential and difference equations . . . . .	23
1.7.10	Information criteria–heuristics for choosing model order . . . . .	25
1.7.11	Fuzzy relational models . . . . .	26
1.7.12	Graph-based models . . . . .	28
1.8	Contributions of this thesis . . . . .	30
<b>2</b>	<b>Bond graphs</b>	<b>31</b>
2.1	Energy based lumped parameter models . . . . .	31
2.2	Standard bond graph elements . . . . .	32
2.2.1	Bonds . . . . .	32
2.2.2	Storage elements . . . . .	33
2.2.3	Source elements . . . . .	34

2.2.4	Sink elements . . . . .	35
2.2.5	Junctions . . . . .	35
2.3	Augmented bond graphs . . . . .	39
2.3.1	Integral and derivative causality . . . . .	41
2.3.2	Sequential causality assignment procedure . . . . .	44
2.4	Additional bond graph elements . . . . .	46
2.4.1	Activated bonds and signal blocks . . . . .	46
2.4.2	Modulated elements . . . . .	46
2.4.3	Complex elements . . . . .	47
2.4.4	Compound elements . . . . .	48
2.5	Simulation . . . . .	51
2.5.1	State space models . . . . .	51
2.5.2	Mixed causality models . . . . .	52
<b>3</b>	<b>Genetic programming</b>	<b>54</b>
3.1	Models, programs, and machine learning . . . . .	54
3.2	History of genetic programming . . . . .	55
3.3	Genetic operators . . . . .	57
3.3.1	The crossover operator . . . . .	58
3.3.2	The mutation operator . . . . .	58
3.3.3	The replication operator . . . . .	58
3.3.4	Fitness proportional selection . . . . .	59
3.4	Generational genetic algorithms . . . . .	59
3.5	Building blocks and schemata . . . . .	61
3.6	Tree structured programs . . . . .	63
3.6.1	The crossover operator for tree structures . . . . .	67
3.6.2	The mutation operator for tree structures . . . . .	67
3.6.3	Other operators on tree structures . . . . .	68
3.7	Symbolic regression . . . . .	69
3.8	Closure or strong typing . . . . .	70
3.9	Genetic programming as indirect search . . . . .	72
3.10	Search tuning . . . . .	73
3.10.1	Controlling program size . . . . .	73
3.10.2	Maintaining population diversity . . . . .	74
<b>4</b>	<b>Implementation</b>	<b>76</b>
4.1	Program structure . . . . .	76
4.1.1	Formats and representations . . . . .	77

4.2	External tools . . . . .	81
4.2.1	The Python programming language . . . . .	81
4.2.2	The SciPy numerical libraries . . . . .	82
4.2.3	Graph layout and visualization with Graphviz . . . . .	82
4.3	A bond graph modelling library . . . . .	83
4.3.1	Basic data structures . . . . .	83
4.3.2	Adding elements and bonds, traversing a graph . . . . .	84
4.3.3	Assigning causality . . . . .	84
4.3.4	Extracting equations . . . . .	86
4.3.5	Model reductions . . . . .	86
4.4	An object-oriented symbolic algebra library . . . . .	100
4.4.1	Basic data structures . . . . .	100
4.4.2	Basic reductions and manipulations . . . . .	102
4.4.3	Algebraic loop detection and resolution . . . . .	104
4.4.4	Reducing equations to state-space form . . . . .	107
4.4.5	Simulation . . . . .	110
4.5	A typed genetic programming system . . . . .	110
4.5.1	Strongly typed mutation . . . . .	112
4.5.2	Strongly typed crossover . . . . .	113
4.5.3	A grammar for symbolic regression on dynamic systems	113
4.5.4	A grammar for genetic programming bond graphs . . . . .	118
<b>5</b>	<b>Results and discussion</b>	<b>120</b>
5.1	Bond graph modelling and simulation . . . . .	120
5.1.1	Simple spring-mass-damper system . . . . .	120
5.1.2	Multiple spring-mass-damper system . . . . .	125
5.1.3	Elastic collision with a horizontal surface . . . . .	128
5.1.4	Suspended planar pendulum . . . . .	130
5.1.5	Triple planar pendulum . . . . .	133
5.1.6	Linked elastic collisions . . . . .	139
5.2	Symbolic regression of a static function . . . . .	141
5.2.1	Population dynamics . . . . .	141
5.2.2	Algebraic reductions . . . . .	143
5.3	Exploring genetic neighbourhoods by perturbation of an ideal	151
5.4	Discussion . . . . .	153
5.4.1	On bond graphs . . . . .	153
5.4.2	On genetic programming . . . . .	157

<b>6</b>	<b>Summary and conclusions</b>	<b>160</b>
6.1	Extensions and future work . . . . .	161
6.1.1	Additional experiments . . . . .	161
6.1.2	Software improvements . . . . .	163



# List of Figures

1.1	Using output prediction error to evaluate a model. . . . .	4
1.2	The black box system identification problem. . . . .	5
1.3	The grey box system identification problem. . . . .	5
1.4	Block diagram of a Volterra function series model . . . . .	17
1.5	The Hammerstein filter chain model structure . . . . .	17
1.6	Block diagram of the linear, squarer, cuber model . . . . .	18
1.7	Block diagram for a nonlinear moving average operation . . . .	21
2.1	A bond denotes power continuity . . . . .	32
2.2	A capacitive storage element: the 1-port capacitor, C . . . . .	34
2.3	An inductive storage element: the 1-port inertia, I . . . . .	34
2.4	An ideal source element: the 1-port effort source, Se . . . . .	35
2.5	An ideal source element: the 1-port flow source, Sf . . . . .	35
2.6	A dissipative element: the 1-port resistor, R . . . . .	36
2.7	A power-conserving 2-port element: the transformer, TF . . . .	37
2.8	A power-conserving 2-port element: the gyrator, GY . . . . .	37
2.9	A power-conserving multi-port element: the 0-junction . . . . .	38
2.10	A power-conserving multi-port element: the 1-junction . . . . .	38
2.11	A fully augmented bond includes a causal stroke . . . . .	40
2.12	Derivative causality: electrical capacitors in parallel . . . . .	44
2.13	Derivative causality: rigidly joined masses . . . . .	45
2.14	A power-conserving 2-port: the modulated transformer, MTF . . .	47
2.15	A power-conserving 2-port: the modulated gyrator, MGY . . . .	47
2.16	A non-linear storage element: the contact compliance, CC . . . .	48
2.17	Mechanical contact compliance, an unfixed spring . . . . .	48
2.18	Effort-displacement plot for the CC element . . . . .	49
2.19	A suspended pendulum . . . . .	50
2.20	Sub-models of the simple pendulum . . . . .	50

2.21	Model of a simple pendulum using compound elements . . . . .	51
3.1	A computer program is like a system model. . . . .	54
3.2	The crossover operation for bit string genotypes . . . . .	58
3.3	The mutation operation for bit string genotypes . . . . .	58
3.4	The replication operation for bit string genotypes . . . . .	59
3.5	Simplified GA flowchart . . . . .	61
3.6	Genetic programming produces a program. . . . .	66
3.7	The crossover operation for tree structured genotypes . . . . .	68
3.8	Genetic programming as indirect search . . . . .	73
4.1	Model reductions step 1: remove trivial junctions . . . . .	88
4.2	Model reductions step 2: merge adjacent like junctions . . . . .	89
4.3	Model reductions step 3: merge resistors . . . . .	90
4.4	Model reductions step 4: merge capacitors . . . . .	91
4.5	Model reductions step 5: merge inertias . . . . .	91
4.6	A randomly generated model . . . . .	92
4.7	A reduced model: step 1 . . . . .	93
4.8	A reduced model: step 2 . . . . .	94
4.9	A reduced model: step 3 . . . . .	95
4.10	Algebraic dependencies from the original model . . . . .	96
4.11	Algebraic dependencies from the reduced model . . . . .	97
4.12	Algebraic object inheritance diagram . . . . .	101
4.13	Algebraic dependencies with loops resolved . . . . .	108
5.1	Schematic diagram of a simple spring-mass-damper system . . . . .	123
5.2	Bond graph model of the system from figure 5.1 . . . . .	123
5.3	Step response of a simple spring-mass-damper model . . . . .	124
5.4	Noise response of a simple spring-mass-damper model . . . . .	124
5.5	Schematic diagram of a multiple spring-mass-damper system . . . . .	126
5.6	Bond graph model of the system from figure 5.5 . . . . .	126
5.7	Step response of a multiple spring-mass-damper model . . . . .	127
5.8	Noise response of a multiple spring-mass-damper model . . . . .	127
5.9	A bouncing ball . . . . .	128
5.10	Bond graph model of a bouncing ball with switched C-element . . . . .	129
5.11	Response of a bouncing ball model . . . . .	129
5.12	Bond graph model of the system from figure 2.19 . . . . .	132
5.13	Response of a simple pendulum model . . . . .	133

5.14	Vertical response of a triple pendulum model . . . . .	136
5.15	Horizontal response of a triple pendulum model . . . . .	137
5.16	Angular response of a triple pendulum model . . . . .	137
5.17	A triple planar pendulum . . . . .	138
5.18	Schematic diagram of a rigid bar with bumpers . . . . .	139
5.19	Bond graph model of the system from figure 5.18 . . . . .	140
5.20	Response of a bar-with-bumpers model . . . . .	140
5.21	Fitness in run A . . . . .	144
5.22	Diversity in run A . . . . .	144
5.23	Program size in run A . . . . .	145
5.24	Program age in run A . . . . .	145
5.25	Fitness of run B . . . . .	146
5.26	Diversity of run B . . . . .	146
5.27	Program size in run B . . . . .	147
5.28	Program age in run B . . . . .	147
5.29	A best approximation of $x^2 + 2.0$ after 41 generations . . . . .	149
5.30	The reduced expression . . . . .	150
5.31	Mean fitness versus tree-distance from the ideal . . . . .	153
5.32	Symbolic regression “ideal” . . . . .	154

# List of Tables

1.1	Modelling tasks in order of increasing difficulty. . . . .	13
2.1	Units of effort and flow in several physical domains. . . . .	33
2.2	Causality options by element type: 1- and 2-port elements. . .	42
2.3	Causality options by element type: junction elements. . . . .	43
4.1	Grammar for symbolic regression on dynamic systems . . . . .	117
4.2	Grammar for genetic programming bond graphs . . . . .	119
5.1	Neighbourhood of a symbolic regression “ideal” . . . . .	152

# Listings

3.1	Protecting the fitness test code against division by zero . . . .	71
4.1	A simple bond graph model in .bg format . . . . .	78
4.2	A simple bond graph model constructed in Python code . . .	80
4.3	Two simple functions in Python . . . . .	81
4.4	Graphviz DOT markup for the model defined in listing 4.4 . .	83
4.5	Equations from the model in figure 4.6 . . . . .	98
4.6	Equations from the reduced model in figure 4.9 . . . . .	99
4.7	Resolving a loop in the reduced model . . . . .	106
4.8	Equations from the reduced model with loops resolved . . . .	107

# Chapter 1

## Introduction

### 1.1 System identification

System identification refers to the problem of finding models that describe and predict the behaviour of physical systems. These are mathematical models and, because they are abstractions of the system under test, they are necessarily specialized and approximate.

There are many uses for an identified model. It may be used in simulation to design systems that interoperate with the modelled system, a control system for example. In an iterative design process it is often desirable to use a model in place of the actual system if it is expensive or dangerous to operate the original system. An identified model may also be used online in parallel with the original system for monitoring and diagnosis or model predictive control. A model may be sought that describes the internal structure of a closed system in a way that plausibly explains the behaviour of the system giving insight into its working. It is not even required that the system exist. An experimenter may be engaged in a design exercise and have only functional (behavioural) requirements for the system but no prototype or even structural details of how it should be built. Some system identification techniques are able to run against such functional requirements and generate a model for the final design even before it is realized. If the identification technique can be additionally constrained to only generate models that can be realized in a usable form (akin to finding not just a proof, but a generative proof in mathematics), then the technique is also an automated design method [45].

In any system identification exercise, the experimenter must specialize the model in advance by deciding what aspects of the system behaviour will be represented in the model. For any system that might be modelled, from a rocket to a stone, there are an inconceivably large number of environments that it *might* be placed in, and interactions that it might have with the environment. In modelling the stone, one experimenter may be interested to describe and predict its material strength while another is interested in its specular qualities under different lighting conditions. Neither will (or should) concern themselves with the other's interests when building a model of the stone. Indeed, there is no obvious point at which to stop if either experimenter chose to start including other aspects of a stone's interactions with its environment. One could imagine modelling a stone's structural resonance, chemical reactivity, and aerodynamics only to realize one has not modelled its interaction with the solar wind. Every practical model is specialized, addressing itself to only some aspects of the system behaviour. Like a 1:1 scale map, the only "complete" or "exact" model is a replica of the original system. It is assumed that any experimenter engaged in system identification has reason to not use the system directly at all times.

Identified models can only ever be approximate since they are based on necessarily approximate and noisy measurements of the system behaviour. In order to build a model of the system behaviour, it must first be observed. There are sources of noise and inaccuracy at many levels of any practical measurement system. Precision may be lost and errors accumulated by manipulations of the measured data during the process of building a model. Finally, the model representation itself will always be finite in scope and precision. In practice, a model's accuracy should be evaluated in terms of the purpose for which the model is built. A model is adequate if it describes and predicts the behaviour of the original system in a way that is satisfactory or useful to the experimenter.

In one sense, the term "system identification" is nearly a misnomer since the result is at best an abstraction of some aspects of the system and not at all a unique identifier for it. There can be many different systems which are equally well described in one aspect or another by a single model and there can be many equally valid models for different aspects of a single system. Often there are additional equivalent representations for a model, such as a differential equation and its Laplace transform, and tools exist that assist in moving from one representation to another [28]. Ultimately, it is perhaps best to think of an identified model as a second system that, for some purposes,

can be used in place of the system under test. They are identical in some useful aspect. This view is most clear when using analog computers to solve some set of differential equations, or when using a software on a digital computer to simulate another digital system ([59] contains many examples that use MATLAB to simulate digital control systems).

## 1.2 Black box and grey box problems

The black box system identification problem is specified as, after observing a time sequence of input and measured output values, find a model that as closely as possible reproduces the output sequence given only the input sequence. In some cases the system inputs may be a controlled test signal chosen by the experimenter. A typical experimental set-up is shown schematically in figure 1.2. It is common, and perhaps intuitive, to evaluate models according to their output prediction error, as shown schematically in figure 1.1. Other formulations are possible but less often used. Input prediction error, for example, uses an inverse model to predict test signal values from the measured system output. Generalized prediction error uses a two part model consisting of a forward model of input processes and an inverse model of output processes; their meeting corresponds to some point internal to the system under test [85].

Importantly, the system is considered to be opaque (black), exposing only a finite set of observable input and output ports. No assumptions are made about the internal structure of the system. Once a test signal enters the system at an input port it cannot be observed any further. The origin of output signals cannot be traced back within the system. This inscrutable conception of the system under test is a simplifying abstraction. Any two systems which produce identical output signals for a given input signal are considered equivalent in a black box system identification. By making no assumptions about the internal structure or physical principles of the system, an experimenter can apply familiar system identification techniques to a wide variety of systems and use the resulting models interchangeably.

It is not often true, however, that nothing is known of the system internals. Knowledge of the system internals may come from a variety of sources including

- Inspection or disassembly of a non-operational system



- Theoretical principles applicable to that class of systems
- Knowledge from the design and construction of the system
- Results of another system identification exercise
- Past experience with similar systems.

In these cases, it may be desirable to use an identification technique that is able to use and profit from a preliminary model incorporating any prior knowledge of the structure of the system under test. Such a technique is loosely called “grey box” system identification to indicate that the system under test is neither wholly opaque nor wholly transparent to the experimenter. The grey box system identification problem is shown schematically in figure 1.3.

Note that in all the schematic depictions above the measured signal available to the identification algorithm and to the output prediction error calculation is a sum of the actual system output and some added noise. This added noise represents random inaccuracies in the sensing and measurement apparatus as are unavoidable when modelling real, physical processes. A zero mean Gaussian distribution is usually assumed for the added noise signal. This is representative of many physical processes and a common simplifying assumption for others.

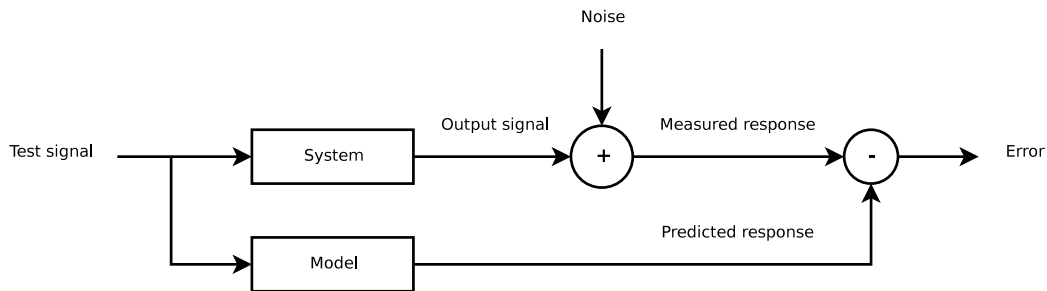


Figure 1.1: Using output prediction error to evaluate a model.

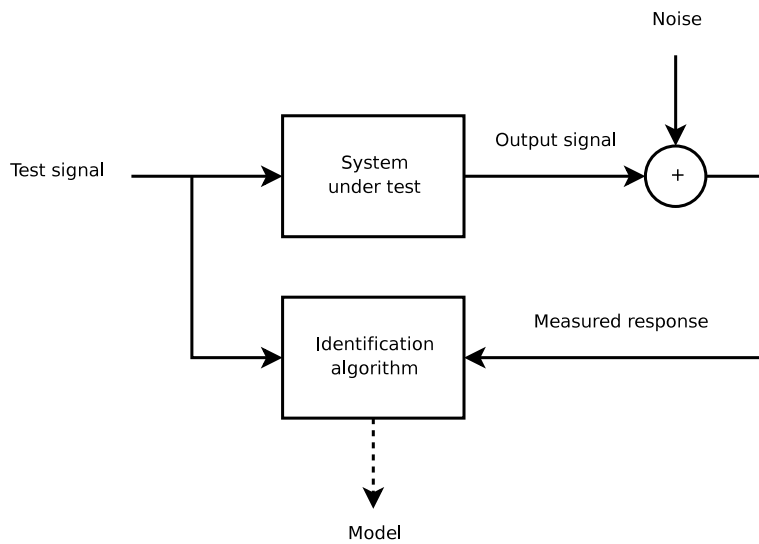


Figure 1.2: The black box system identification problem.

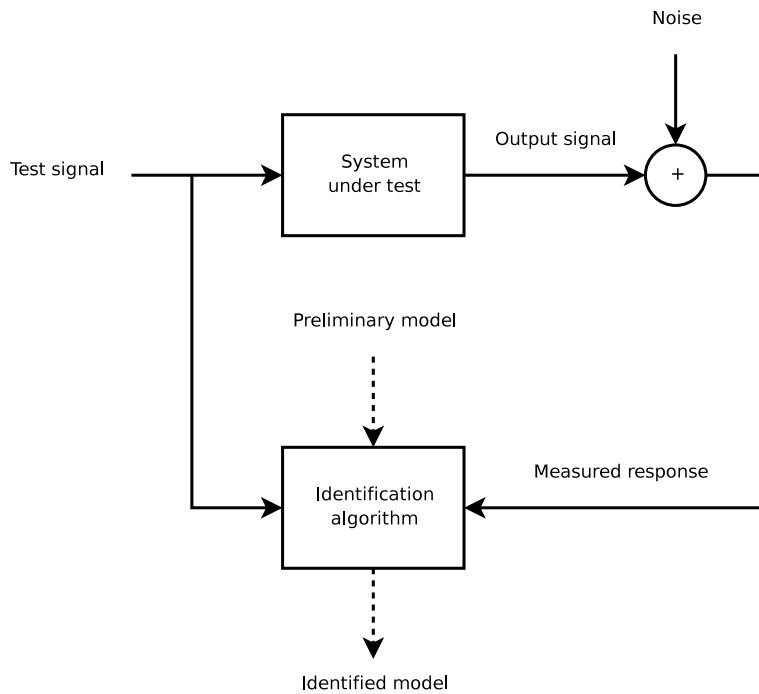


Figure 1.3: The grey box system identification problem.

### 1.3 Static and dynamic models

Models can be classified as either static or dynamic. The output of a static model at any instant depends only on the value of the inputs at that instant and in no way on any past values of the input. Put another way, the input–output relationships of a static model are independent of order and rate at which the inputs are presented to it. This prohibits static models from exhibiting many useful and widely evident behaviours such as hysteresis and resonance. As such, static models are not of much interest on their own but often occur as components of a larger compound model that is dynamic over all. The system identification procedure for static models can be phrased in visual terms as: fit a line (plane, hyper-plane), curve (sheet), or some discontinuous function to a plot of system input–output data. The data may be collected in any number of experiments and is plotted without regard to the order in which it was collected. Because there is no way for past inputs to have a persistent effect within static models, these are also sometimes called “stateless” or “zero-memory” models.

Dynamic models have “memory” in some form. The observable effects at their outputs of changes at their inputs may persist beyond subsequent changes at their inputs. A very common formulation for dynamic models is called “state space form”. In this formulation a dynamic system is represented by a set of input, state, and output variables that are related by a set of algebraic and first order ordinary differential equations. The input variables are all free variables in these equations. Their values are determined outside of the model. The first derivative of each state variable appears alone on the left-hand side of a differential equation, the right-hand side of which is an algebraic expression containing input and state variables, but not output variables. The output variables are defined algebraically in terms of the state and input variables. For a system with  $n$  input variables,  $m$  state variables and  $p$  output variables, the model equations would be

$$\begin{aligned}\dot{u}_1 &= f_1(x_1, \dots, x_n, u_1, \dots, u_m) \\ &\vdots \\ \dot{u}_m &= f_m(x_1, \dots, x_n, u_1, \dots, u_m)\end{aligned}$$

$$\begin{aligned}
y_1 &= g_1(x_1, \dots, x_n, u_1, \dots, u_m) \\
&\vdots \\
y_p &= g_p(x_1, \dots, x_n, u_1, \dots, u_m)
\end{aligned}$$

where  $x_1 \dots x_n$  are the input variables,  $u_1 \dots u_m$  are the state variables, and  $y_1 \dots y_p$  are the output variables.

The main difficulties in identifying dynamic models (over static ones) are first, because the data order of arrival within the input and output signals matter, typically much more data must be collected and second, that the system may have unobservable state variables. Identification of dynamic models typically requires more data to be collected since, to adequately sample the behaviour of a stateful system, not just every interesting combination of the inputs but every path through the interesting combinations of the inputs must be exercised. Here “interesting” means simply “of interest to the experimenter”.

Unobservable state variables are state variables that do not coincide exactly with an output variable. That is a  $u_i$  for which there is no  $y_j = g_j(x_1, \dots, x_n, u_1, \dots, u_m) = u_i$ . The difficulty in identifying systems with unobservable state variables is twofold. First, there may be an unknown number of unobservable state variables concealed within the system. This leaves the experimenter to estimate an appropriate model order. If this estimate is too low then the model will be incapable of reproducing the full range of system behaviour. If this estimate is too high then extra computation costs will be incurred during identification and use of the model and there is an increased risk of over-fitting (see section 1.6). Second, for any system with an unknown number of unobservable state variables it is impossible to know for certain the initial state of the system. This is a difficulty to experimenters since, for an extreme example, it may be futile to exercise any path through the input space during a period when the outputs are overwhelmingly determined by persisting effects of past inputs. Experimenters may attempt to work around this by allowing the system to settle into a consistent, if unknown, state before beginning to collect data. In repeated experiments the maximum time between application of a stationary input signal and the arrival of the system outputs at a stationary signal is recorded. Then while collecting data for model evaluation the experimenter will discard the output signal for at least as much time between start of excitation and start of data collection.

## 1.4 Parametric and non-parametric models

Models can be divided into those described by a small number of parameters at distinct points in the model structure, called parametric models, and those described by an often quite large number of parameters all of which have similar significance in the model structure, called non-parametric models.

System identification with parametric model entails choosing a specific model structure and then estimating the model parameters to best fit model behaviour to system behaviour. Symbolic regression (discussed in section 3.7) is unusual because it is able to identify the model structure automatically and the corresponding parameters at the same time.

Parametric models are typically more concise (contain fewer parameters and are shorter to describe) than similarly accurate non-parametric models. Parameters in a parametric model often have individual significance in the system behaviour. For example, the roots of the numerator and denominator of a transfer function model indicate poles and zeros that are related to the settling time, stability, and other behavioural properties of the model. Continuous or discrete transfer functions (in the Laplace or Z domain) and their time domain representations (differential and difference equations) are all parametric models. Examples of linear structures in each of these forms is given in equations 1.1 (continuous transfer function), 1.2 (discrete transfer function), 1.3 (continuous time-domain function), and 1.4 (discrete time-domain function). In each case, the experimenter must choose  $m$  and  $n$  to fully specify the model structure. For non-linear models, there are additional structural decisions to be made, as described in Section 1.5.

$$Y(s) = \frac{b_m s^m + b_{m-1} s^{m-1} + \dots + b_1 s + b_0}{s^n + a_{n-1} s^{n-1} + \dots + a_1 s + a_0} X(s) \quad (1.1)$$

$$Y(z) = \frac{b_1 z^{-1} + b_2 z^{-2} + \dots + b_m z^{-m}}{1 + a_1 z^{-1} + a_2 z^{-2} + \dots + a_n z^{-n}} X(z) \quad (1.2)$$

$$\frac{d^n y}{dt^n} + a_{n-1} \frac{d^{n-1} y}{dt^{n-1}} + \dots + a_1 \frac{dy}{dt} + a_0 y = b_m \frac{d^m x}{dt^m} + \dots + b_1 \frac{dx}{dt} + b_0 x \quad (1.3)$$

$$y_k + a_1 y_{k-1} + \dots + a_n y_{k-n} = b_1 x_{k-1} + \dots + b_m x_{k-m} \quad (1.4)$$

Because they typically contain fewer identified parameters and the range of possibilities is already reduced by the choice of a specific model structure,

parametric models are typically easier to identify provided the chosen specific model structure is appropriate. However, system identification with an inappropriate parametric model structure, particularly a model that is too low order and therefore insufficiently flexible, is sure to be frustrating. It is an advantage of non-parametric models that the experimenter need not make this difficult model structure decision.

Examples of non-parametric model structures include: impulse response models (equation 1.5 continuous and 1.6 discrete), frequency response models (equation 1.7), and Volterra kernels. Note that equation 1.7 is simply the Fourier transform of equation 1.5 and that equation 1.6 is the special case of a Volterra kernel with no higher order terms.

$$y(t) = \int_0^t u(t - \tau)x(\tau)d\tau \quad (1.5)$$

$$y(k) = \sum_{m=0}^k u(k - m)x(m) \quad (1.6)$$

$$Y(j\omega) = U(j\omega)X(j\omega) \quad (1.7)$$

The lumped-parameter, graph-based models discussed in chapter 2 are compositions of sub-models that may be either parametric or non-parametric. The genetic programming based identification technique discussed in chapter 3 relies on symbolic regression to find numerical constants and algebraic relations not specified in the prototype model or available graph components. Any non-parametric sub-models that appear in the final compound model must have been identified by other techniques and included in the prototype model using a grey box approach to identification.

## 1.5 Linear and nonlinear models

There is a natural ordering of linear parametric models from low-order models to high-order models. That is, linear models of a given structure they can be enumerated from less complex and flexible to more complex and flexible. For example, models structured as a transfer function (continuous or discrete) can be ordered by the number of poles first and by the number of zeros for models with the same number of poles. Models having a greater number of poles and zeros are more flexible, that is, they are able to reproduce more

complex system behaviours than lower order models. Two or more models having the same number of poles and zeros would be considered equally flexible.

Linear here means linear-in-the-parameters, assuming a parametric model. A parametric model is linear-in-the-parameters if, for input vector  $u$ , output vector  $y$ , and parameters to be identified  $\beta_1 \dots \beta_n$ , the model can be written as a polynomial containing non-linear functions of the inputs but not of the parameters. For example, equation 1.8 is linear in the parameters  $\beta_i$  while equation 1.9 is not. Both contain non-linear functions of the inputs (namely  $u_2^2$  and  $\sin(\dots u_3)$ ) but equation 1.8 forms a linear sum of the parameters, each multiplied by some factor that is unrelated in any way to the parameters. This structure permits the following direct solution to the identification problem.

$$y = \beta_1 u_1 + \beta_2 u_2^2 + \beta_3 \sin(u_3) \quad (1.8)$$

$$y = \beta_1 \beta_2 u_1 + u_2^2 + \sin(\beta_3 u_3) \quad (1.9)$$

For models that are linear-in-the-parameters, the output prediction error is also a linear function of the parameters. A best fit model can be found directly (in one step) by taking the partial derivative of the error function with respect to the parameters and testing each extrema to solve for the parameters vector that minimizes the error function. When the error function is a sum of squared error terms, this technique is called a linear least squares parameter estimation. In general, there are no such direct solutions to the identification problem for non-linear model types. Identification techniques for non-linear models all resort to an iterative, multi-step process of some sort. These iterations can be viewed as a search through the model parameter and, in some cases, model structure space. There is no guarantee of finding a globally optimal model in terms of output prediction error.

There is a fundamental trade-off between model size, or order, and the model's expressive power, or flexibility. In general it is desirable to identify the most compact model that adequately represents the system behaviour. Larger models incur greater storage and computational costs during identification and use. Excessively large models are also more vulnerable to over fitting (discussed in section 1.6), a case of misapplied flexibility.

One approach to identifying minimal-order linear models is to start with an arbitrarily low order model structure and repeatedly identify the best

model using the linear least squares technique discussed below while incrementing the model order between attempts. The search is terminated as soon as an adequate representation of system behaviour is achieved. Unfortunately, this approach or any like it are not applicable to general non-linear models since any number of qualitatively distinct yet equally complex structures are possible.

One final complication to the identification of non-linear models is that superposition of separately measured responses applies to linear models only. One can not measure the output of a non-linear system in response to distinct test signals on different occasions and superimpose the measurements latter with any accuracy. This prohibits conveniences such as adding a test signal to the normal operating inputs and subtracting the normal operating outputs from the measured output to estimate the system response to the test signal alone.

## 1.6 Optimization, search, machine learning

System identification can be viewed as a type of optimization or machine learning problem. In fact most system identification techniques draw heavily from the mathematics of optimization and in as much as the “best” model structure and parameter values are sought, system identification *is* an optimization problem. When automatic methods are desired, system identification is a machine learning problem. Finally, system identification can also be viewed as a search problem where a “good” or “best” model is sought within the space of all possible models.

Overfitting and premature optimization are two pitfalls common to many machine learning, search, and global optimization problems. They might also be phrased as the challenge of choosing a rate at which to progress and of deciding when to stop. The goal in all cases is to generalize on the training data, find a global optimum, and to not give too much weight to unimportant local details.

Premature optimization occurs when a search algorithm narrows in on a neighbourhood that is locally, but not globally optimal. Many system identification and machine learning algorithms incorporate tunable parameters that adjust how aggressive the algorithm is in pursuing a solution. In machine learning, this is called a learning rate and controls how fast the algorithm will accept new knowledge and discard old. In heuristic search algorithms it



is a factor that trades off exploration of new territory with exploitation of local gain made in a neighbourhood. Back propagation learning for neural networks encodes the learning rate in a single constant factor. In fuzzy logic systems it is implicit in the fuzzy rule adjustment procedures. Learning rate corresponds to step size in a hill climbing or simulated annealing search and in (tree structured) genetic algorithms the representation (function and terminal set) determines the topography of the search space and the operator design (crossover and mutation functions) determine the allowable step sizes and directions.

Ideally, the identified model will respond to important trends in the data, and not to random noise or aspects of the system that are uninteresting to the experimenter. Successful machine learning algorithms generalize on the training data and good identified models give good results to unseen test signals as well as those used during identification. A model which does not generalize well may be “overfitting”. To check for overfitting, a model is validated against unseen data. This data must be held aside strictly for validation purposes. If the validation results are used to select a best model, then the validation data implicitly becomes training data. Stopping criteria typically include a quality measure (a minimum fitness or maximum error threshold) and computational measures (a maximum number of iterations or amount of computer time). Introductory descriptions of genetic algorithms and back propagation learning in neural networks mention both criteria and a “first met” arrangement. The Bayesian and Akaike information criteria are two heuristic stopping criteria mentioned in every machine learning textbook. They both incorporate a measure of parsimony for the identified model and weight that against the fitness or error measure. They are therefore applicable to techniques that generate a variable length model the complexity or parsimony of which is easily quantifiable.

## 1.7 Some model types and identification techniques

Table 1.1 divides all models into four classes labelled in order of increasing difficulty of their identification problem. The linear problems each have well known solution methods that achieve an optimal model in one step. For linear static models (class 1 in table 1.1) the problem is to fit a line, plane,

or hyper-plane that is “best” by some measure (such as minimum squared or absolute error) to the input–output data. The usual solution method is to form the partial derivative of the error measure with respect to the model parameters (slope and offset for a simple line) and solve for model parameter values at the minimum error extremum. The solution method is identical for linear-in-the-parameters dynamic models (class 2 in table 1.1). Non-linear static modelling (class 3 in table 1.1) is the general function approximation problem (also known as curve fitting or regression analysis). The fourth class of system identification problems, identifying models that are both non-linear and dynamic, is the most challenging and the most interesting. Non-linear dynamic models are the most difficult to identify because, in general, it is a combined structure and parameter identification problem. This thesis proposes a technique–bond graph models created by genetic programming–to address black- and grey-box problems of this type.

Table 1.1: Modelling tasks in order of increasing difficulty.

	linear	non-linear
static	(1)	(3)
dynamic	(2)	(4)

(1), (2) are both linear regression problems and (3) is the general function approximation problem. (4) is the most general class of system identification problems.

Two models using local linear approximations of nonlinear systems are discussed first, followed by several nonlinear nonparametric and parametric models.

### 1.7.1 Locally linear models

Since nonlinear system identification is difficult, two categories of techniques have arisen attempting to apply linear models to nonlinear systems. Both rely on local linear approximations of nonlinear systems which are accurate for small changes about a fixed operating point. In the first category several linear model approximations are identified, each at different operating points of the nonlinear system. The complete model interpolates between these local linearizations. An example of this technique is the Linear Parameter-Varying

(LPV) model. In the second category a single linear model approximation is identified at the current operating point and then updated as the operating point changes. The Extended Kalman Filter (EKF) is an example of this technique.

Both these techniques share the advantage of using linear model identification, which is fast and well understood. LPV models can be very effective and are popular for industrial control applications [12], [7]. For a given model structure, the extended Kalman filter estimates state as well as parameters. It often used to estimate parameters of a known model structure given noisy and incomplete measurements (where not all states are measurable) [47], [48]. The basic identification technique for LPV models consists of three steps:

1. Choose the set of base operating points
2. Perform a linear system identification at each of the base operating points
3. Choose an interpolation scheme (e.g. linear, polynomial, or b-spline)

In step 2 it is important to use low amplitude test signals such that the system will not be excited outside a small neighbourhood of the chosen operating point for which the system is assumed linear. The corollary to this is that with any of the globally nonlinear models discussed later it is important to use a large test signal which excites the nonlinear dynamics of the system otherwise only the local, approximately linear behaviour will be modelled. [5] presents a one-shot procedure that combines all of the three steps above.

Both of these techniques are able to identify the parameters, but not the structure of a model. For EKF, the model structure must be given in a state space form. For LPV the model structure is defined by the number and spacing of the base operating points and the interpolation method. Recursive or iterative forms of linear system identification techniques can track (update the model online) mildly nonlinear systems that simply drift over time.

### 1.7.2 Nonparametric modelling

Volterra function series models and artificial neural networks are two general nonparametric nonlinear models about which quite a lot has been written. Volterra models are linear in the parameters but they suffer from an extremely large number of parameters which leads to computational difficulties and overfitting. Dynamic system identification can be posed as a

function approximation problem and neural networks are a popular choice of approximator. Neural networks can also be formed in so-called recurrent configurations. Recurrent neural networks are dynamic systems themselves.

### 1.7.3 Volterra function series models

Volterra series models are an extension of linear impulse response models (first order time-domain kernels) to nonlinear systems. The linear impulse response model is given by:

$$y(t) = y_1(t) = \int_{-T_s}^0 g(t - \tau) \cdot u(\tau) d\tau \quad (1.10)$$

Its discrete-time version is:

$$y(t_k) = y_1(t_k) = \sum_{j=0}^{N_s} W_j \cdot u_{k-j} \quad (1.11)$$

The limits of integration and summation are finite since practical, stable systems are assumed to have negligible dependence on the very distant past. That is, only finite memory systems are considered. The Volterra series model extends the above into an infinite sum of functions. The first of these functions is the linear model  $y_1(t)$  above.

$$y(t) = y_1(t) + y_2(t) + y_3(t) + \dots \quad (1.12)$$

The continuous-time form of the additional terms (called bilinear, trilinear, etc. functions) is:

$$\begin{aligned} y_2(t) &= \int_{-T_1}^0 \int_{-T_2}^0 g_2(t - \tau_1, t - \tau_2) \cdot u(\tau_1) \cdot u(\tau_2) d\tau_1 d\tau_2 \\ y_3(t) &= \int_{-T_1}^0 \int_{-T_2}^0 \int_{-T_3}^0 g_3(t - \tau_1, t - \tau_2, t - \tau_3) \cdot u(\tau_1) \cdot u(\tau_2) \cdot u(\tau_3) d\tau_1 d\tau_2 d\tau_3 \\ &\vdots \\ y_p(t) &= \int_{-T_1}^0 \int_{-T_2}^0 \cdots \int_{-T_p}^0 g_p(t - \tau_1, t - \tau_2, \dots, t - \tau_p) \cdot u(\tau_1) \cdot u(\tau_2) \cdots \\ &\quad \cdots \cdot u(\tau_p) d\tau_1 d\tau_2 \cdots d\tau_p \\ &\vdots \end{aligned}$$

Where,  $g_p$  is called the  $p^{th}$  order continuous time kernel. The rest of this discussion will use the discrete-time representation.

$$\begin{aligned}
y_2(t_n) &= \sum_{i=0}^{M_2} \sum_{j=0}^{M_2} W_{i,j}^{(2)} \cdot u_{n-i} \cdot u_{n-j} \\
y_3(t_n) &= \sum_{i=0}^{M_3} \sum_{j=0}^{M_3} \sum_{k=0}^{M_3} W_{i,j,k}^{(3)} \cdot u_{n-i} \cdot u_{n-j} \cdot u_{n-k} \\
&\vdots \\
y_p(t_n) &= \sum_{i_1=0}^{M_p} \sum_{i_2=0}^{M_p} \cdots \sum_{i_p=0}^{M_p} W_{i_1,i_2,\dots,i_p}^{(p)} \cdot u_{n-i_1} \cdot u_{n-i_2} \cdots u_{n-i_p} \\
&\vdots
\end{aligned}$$

$W^{(p)}$  is called the  $p$ 'th order discrete time kernel.  $W^{(2)}$  is a  $M_2$  by  $M_2$  square matrix and  $W^{(3)}$  is a  $M_3$  by  $M_3$  by  $M_3$  cubic matrix and so on. It is not required that the memory lengths of each kernel ( $M_2$ ,  $M_3$ , etc.) are the same, but they are typically chosen equal for consistent accounting. If the memory length is  $M$ , then the  $p$ 'th order kernel  $W^{(p)}$  is a  $p$  dimensional matrix with  $M^p$  elements. The overall Volterra series model has the form of a sum of products of delayed values of the input with constant coefficients coming from the kernel elements. The model parameters are the kernel elements, and the model is linear in the parameters. The number of parameters is  $M_{(1)} + M_{(2)}^2 + M_{(3)}^3 + \cdots + M_{(p)}^p + \dots$  or infinite. Clearly the full Volterra series with an infinite number of parameters cannot be used in practice; rather it is truncated after the first few terms. In preparing this thesis, no references were found to applications using more than 3 terms in the Volterra series. The number of parameters is still large even after truncating at the third order term. A block diagram of the full Volterra function series model is shown in figure 1.4.

#### 1.7.4 Two special models in terms of Volterra series

Some nonlinear systems can be separated into a static nonlinear function followed by a linear time-invariant (LTI) dynamic system. This form is called a Hammerstein model and is shown in figure 1.5 The opposite, an LTI dynamic system followed by a static nonlinear function is called a Weiner model.

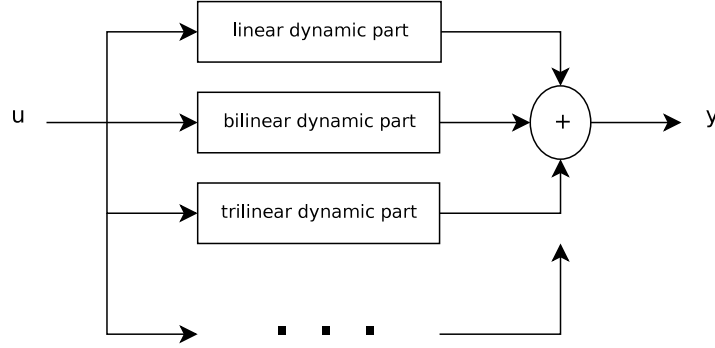


Figure 1.4: Block diagram of a Volterra function series model

This section discusses the Hammerstein model in terms of Volterra series and presents another simplified Volterra-like model recommended in [6].

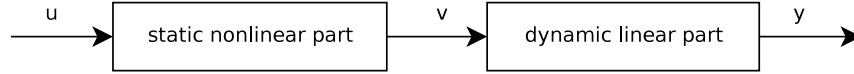


Figure 1.5: The Hammerstein filter chain model structure

Assuming the static nonlinear part is a polynomial in  $u$  and given the linear dynamic part as an impulse response weighting sequence  $W$  of length  $N$ .

$$v = a_0 + a_1u + a_2u^2 + \dots + a_q u^q \quad (1.13)$$

$$y(t_k) = \sum_{j=0}^N W_j \cdot v_{k-j} \quad (1.14)$$

Combining these gives:

$$\begin{aligned} y(t_k) &= \sum_{j=0}^N W_j \sum_{i=0}^q a_i u_i^k \\ &= \sum_{i=0}^q a_i \sum_{j=0}^N W_j u_i^k \\ &= a_0 \sum_{j=0}^N W_j + a_1 \sum_{j=0}^N W_j u_1 + a_2 \sum_{j=0}^N W_j u_2^2 + \dots + a_q \sum_{j=0}^N W_j u_q^q \end{aligned}$$

Comparing with the discrete time Volterra function series shows that the Hammerstein filter chain model is a discrete Volterra model with zero for all the off-diagonal elements in higher order kernels. Elements on the main diagonal of higher order kernels can be interpreted as weights applied to powers of the input signal.

Bendat [6] advocates using a simplification of the Volterra series model that is similar to the special Hammerstein model described above. It is suggested to truncate the series at 3 terms and to replace the bilinear and trilinear operations with a square- and cube-law static nonlinear map followed by LTI dynamic systems. This is shown in figure 1.6. Obviously it is less general than the system in figure 1.4, not only because it is truncated at the third order term but also because, like the special Hammerstein model above, it has no counterparts to the off-diagonal elements in the bilinear and trilinear Volterra kernels. In [6] it is argued that the model in figure 1.6 is still adequate for many engineering systems and examples are given in automotive, medical, and naval applications including a 6-DOF model for a moored ship. The great advantage of this model form is that the signals  $u_1 = u$ ,  $u_2 = u^2$ , and  $u_3 = u^3$  can be calculated which reduces the identification of the dynamic parts to a multi-input, multi-output (MIMO) linear problem.

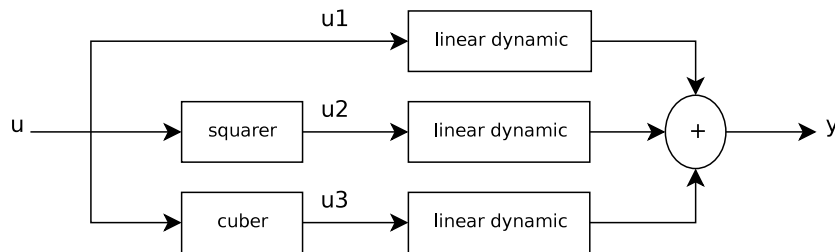


Figure 1.6: Block diagram of the linear, squarer, cuber model

### 1.7.5 Least squares identification of Volterra series models

To illustrate the ordinary least squares formulation of system identification with Volterra series models, just the first 2 terms (linear and bilinear) will

be used.

$$\hat{y}(t_n) = y_1(t_n) + y_2(t_n) + e(t_n) \quad (1.15)$$

$$\underline{\hat{y}} = \sum_{i=0}^{N_1} W_i^1 \cdot u_{n-i} + \sum_{i=0}^{N_2} \sum_{j=0}^{N_2} W_{i,j}^{(2)} \cdot u_{n-i} \cdot u_{n-j} + e(t_n) \quad (1.16)$$

$$= (\underline{u}_1)^T \underline{W}^{(1)} + (\underline{u}_1)^T \underline{W}^{(2)} \underline{u}_2 + \underline{e} \quad (1.17)$$

Where the vector and matrix quantities  $u_1$ ,  $u_2$ ,  $W^{(1)}$ , and  $W^{(2)}$  are:

$$\underline{u}_1 = \begin{bmatrix} u_{n-1} \\ u_{n-2} \\ \vdots \\ u_{n-N_1} \end{bmatrix} \quad \underline{u}_2 = \begin{bmatrix} u_{n-1} \\ u_{n-2} \\ \vdots \\ u_{n-N_2} \end{bmatrix}$$

$$\underline{W}^{(1)} = \begin{bmatrix} W_1^{(1)} \\ W_2^{(1)} \\ \vdots \\ W_{N_1}^{(1)} \end{bmatrix} \quad \underline{W}^{(2)} = \begin{bmatrix} W_{1,1}^{(2)} & \cdots & W_{1,N_2}^{(2)} \\ \vdots & \ddots & \vdots \\ W_{N_1,1}^{(2)} & \cdots & W_{N_1,N_2}^{(2)} \end{bmatrix}$$

Equation 1.17 can be written in a form (equation 1.18) amenable to solution by the method of least squares if the input vector  $U$  and the parameter vector  $\beta$  are constructed as follows. The first  $N_1$  elements in  $U$  are the elements of  $u_1$  and this is followed by elements having the value of the products of  $u_2$  appearing in the second term of equation 1.16. The parameter vector,  $\beta$ , is constructed from elements of  $W^{(1)}$  followed by products of the elements of  $W^{(2)}$  as they appear in the second term of equation 1.16.

$$\hat{y} = U\beta + e \quad (1.18)$$

Given a sequence of  $N$  measurements,  $U$  and  $\beta$  become rectangular, and  $y$  and  $e$  become  $N$  by 1 vectors. The ordinary least squares solution is found by forming the pseudo-inverse of  $U$ .

$$\beta = (U^T U)^{-1} U \hat{y} \quad (1.19)$$

The ordinary least squares formulation has the advantage of not requiring a search process. However, the number of parameters is large,  $O(N^P)$  for a model using the first  $P$  Volterra series terms and a memory length of  $N$ .



A slight reduction in the number of parameters is available by recognizing that not all elements in the higher order kernels are distinguishable. In the bilinear kernel for example, there are terms of the form  $W_{i,j}(u_{n-i})(u_{n-j})$  and  $W_{j,i}(u_{n-j})(u_{n-i})$ . However, since  $(u_{n-i})(u_{n-j}) = (u_{n-j})(u_{n-i})$ , the parameters  $W_{i,j}$  and  $W_{j,i}$  are identical. Only  $N(N+1)/2$  unique parameters need to be identified in the bilinear kernel but this reduction by approximately half leaves a kernel size that is still of the same order,  $O(N^P)$ .

One approach to reducing the number of parameters in a Volterra model is to approximate the kernels with a series of lower order functions, transforming the problem from estimating kernel parameters to estimating a smaller number of parameters to the basis functions. The kernel parameters can then be read out by evaluating the basis functions at  $N$  equally spaced intervals. Treichl et al. [81] use a weighted sum of distorted sinus functions as orthonormal basis functions (OBF) to approximate Volterra kernels. The parameters to this formula are  $\zeta$ , a form factor affecting the shape of the sinus curve,  $j$  which iterates over component sinus curves, and  $i$  the read-out index. There are  $m_r$  sinus curves in the weighted sum. The values of  $\zeta$  and  $m_r$  are design choices. The read-out index varies from 1 to  $N$ , the kernel memory length. In the new identification problem, only the weights,  $w_j$ , applied to each OBF in the sum must be estimated. This problem is also linear in the parameters and the number of parameters has been reduced to  $m_r$ . For example if  $N$  is 40 and  $m_r$  is 10, then there is a 4 times compression of the number of parameters in the linear kernel.

Another kernel compression technique is based on the discrete wavelet transform (DWT). Nikolaou and Mantha [53] show the advantages of a wavelet based compression of the Volterra series kernels while modelling a chemical process in simulation. Using the first two terms of the Volterra series model with  $N_1 = N_2 = 32$  gives the uncompressed model a total of  $N_1 + N_2(N_2 + 1)/2 = 32 + 528 = 560$  parameters. In the wavelet domain identification, only 18 parameters were needed in the linear kernel and 28 in the bilinear kernel for a total of 46 parameters. The 1- and 2-D DWT used to compress the linear and bilinear kernels respectively are not expensive to calculate so the compression ratio of  $560/46$  results in a much faster identification. Additionally, it is reported that the wavelet compressed model shows much less prediction error on validation data not used in the identification. This is attributed to overfitting in the uncompressed model.

## 1.7.6 Nonparametric modelling as function approximation

Black box models identified from input/output data can be seen in very general terms as mappings from inputs and delayed values of the inputs to outputs. Boyd and Chua [13] showed that “any time invariant operation with fading memory can be approximated by a nonlinear moving average operator”. This is depicted in figure 1.7, where  $P(\dots)$  is a polynomial in  $u$  and  $M$  is the model memory length (number of  $Z^{-1}$  delay operators). Therefore, one way to look at nonlinear system identification is as the problem of finding the polynomial  $P$ . The arguments to  $P$ ,  $uZ^{-1}$ ,  $uZ^{-2}$ ,  $\dots$ ,  $uZ^{-M}$ , can be calculated. What is needed is a general and parsimonious function approximator. The Volterra function series is general but not parsimonious, unless the kernels are compressed.

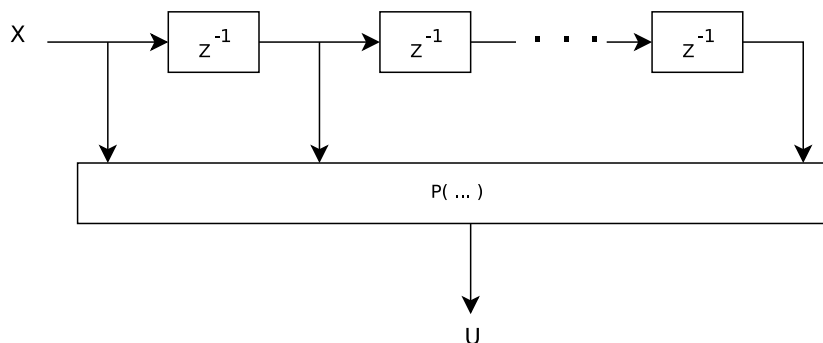


Figure 1.7: Block diagram for a nonlinear moving average operation

Juditsky et al. argue in [38] that there is an important difference between uniformly smooth and spatially adaptive function approximations. Kernels work well to approximate functions that are uniformly smooth. A function is not uniformly smooth if it has important detail at very different scales. For example, if a Volterra kernel is smooth except for a few spikes or steps, then the kernel order must be increased to capture these details and the “curse of dimensionality” (computational expense) is incurred. Also, the resulting extra flexibility of the kernel in areas where the target function is smooth may lead to overfitting. In these areas, coefficients of the kernel approximation should be nearly uniform for a good fit but the kernel encodes more information than is required to represent the system.

Orthonormal basis functions are a valid compression of the model parameters that form smooth, low order representations. However, since common choices such as distorted sinus functions are combined by weighted sums and are not offset or scaled, they are not able to adapt locally to the smoothness of the function being approximated. Wavelets, which are used in many other multi-resolution applications, are particularly well suited to general function approximation and the analytical results are very complete [38]. Sjoberg et al. [73] distinguish between local basis functions, whose gradient vanishes rapidly at infinity, and global basis functions, whose variations are spread across the entire domain. The Fourier series is given as an example of a global basis function; wavelets are a local basis function. Local basis functions are key to building a spatially adaptive function approximator. Since they are essentially constant valued outside of some interval, their effect can be localized.

### 1.7.7 Neural networks

Multilayer feed-forward neural networks have been called a “universal approximator” in that they can approximate any measurable function [35]. Here at the University of Waterloo, there is a project to predict the position of a polishing robot using delayed values of the position set point from 3 consecutive time steps as inputs to a neural network. In the context of figure 1.7, this is a 3rd order non-linear moving average with a neural network taking the place of  $P$ ,

Most neural network applications use local basis functions in the neuron activations. Hofmann et al. [33] identify a Hammerstein-type model using a radial basis function network followed by a linear dynamic system. Scott and Millgrew [70] develop a multilayer feed-forward neural network using orthonormal basis functions and show its advantages over radial basis functions (whose basis centres must be fixed) and multilayer perceptron networks. These advantages appear to be related to the concept of spatial adaptability discussed above.

The classical feed-forward multilayer perceptron (MLP) networks are trained using a gradient descent search algorithm called back-propagation learning (BPL). For radial basis networks this is preceded by a clustering algorithm to choose the basis centres, or they may be chosen manually. One other network configuration is the so-called recurrent or dynamic neural network in which the network output or some intermediate values are used also

as inputs creating a feedback loop. There are any number of ways to configure feedback loops in neural networks. Design choices include the number and source of feedback signals, and where (if at all) to incorporate delay elements. Recurrent neural networks are themselves dynamic systems and just function approximators. Training of recurrent neural networks involves unrolling the network over several time steps and applying the back propagation learning algorithm to the unrolled network. This is even more computationally intensive than the notoriously intensive training of simple feed-forward networks.

### **1.7.8 Parametric modelling**

The most familiar parametric model is a differential equation since this is usually the form of model created when a system is modelled from first principles. Identification methods for differential and difference equation models always require a search process. The process is called equation discovery or sometimes symbolic regression. In general, the more qualitative or heuristic information that can be brought to bear on reducing the search space, the better. Some search techniques are in fact elaborate induction algorithms, others incorporate some generic heuristics called information criteria. Fuzzy relational models can conditionally be considered parametric. Several graph-based models are discussed, including bond graphs which in some cases are simply graphical transcriptions of differential equations but can incorporate arbitrary functions.

### **1.7.9 Differential and difference equations**

Differential equations are very general and are the standard representation for models not identified but derived from physical principles. “Equation discovery” is the general term for techniques which automatically identify an equation or equations that fit some observed data. Other model forms, such as bond graph, are converted into a set of differential and algebraic equations prior to simulation. If no other form is needed then an identification technique that produces such equations immediately is perhaps more convenient. If a system identification experiment is to be bootstrapped (grey-box fashion) from an existing model then the existing model usually needs to be in a form compatible with what the identification technique will produce.

Existing identification methods for differential equations are called “symbolic regression” and “equation discovery”. The first usually refers to methods that rely on simple fitness guided search algorithms such as genetic programming, “evolution strategie”, or simulated annealing. The second term has been used to describe more constraint-based algorithms. These use qualitative constraints to reduce the search space by ruling out whole classes of models by their structure or the acceptable range of their parameters. There are two possible sources for these constraints:

1. They may be given to the identification system from prior knowledge
2. They may be generated online by an inference system.

LAGRAMGE ([23], [24], [78], [79]) is an equation discovery program that searches for equations which conform to a context free grammar specified by the experimenter. This grammar implicitly constrains the search to permit only a (possibly very small) class of structures. The challenge in this approach is to write meaningful grammars for differential equations, grammars that express *a priori* knowledge about the system being modelled. One approach is to start with a grammar that is capable of producing only one unique equation and then modify it interactively to experiment with different structures. Todorovski and Dzeroski [80] describe a “minimal change” heuristic for limiting model size. It starts from an existing model and favours variations which both reduce the prediction error and are structurally similar to the original. This is a grey-box identification technique.

PRET ([74], [75], [14]) is another equation discovery program that incorporates expert knowledge in several domains collected during its design. It is able to rule out whole classes of equations based on qualitative observations of the system under test. For example, if the phase portrait has certain geometrical properties, then the system must be chaotic and therefore at least a second order differential equation. Qualitative observations can also suggest components which are likely to be part of the model. For example output signal components at twice the input signal frequency suggest there may be a squared term in the equation. PRET performs as much high level work as possible eliminating candidate model classes before resorting to numerical simulations.

Ultimately if the constraints do not specify a unique model, all equation discovery methods resort to a fitness guided local search. The first book in the Genetic Programming series [43] discusses symbolic regression and empirical

discovery (yet another term for equation discovery), giving examples from econometrics and mechanical dynamics. The choice of function and terminal sets for genetic programming can be seen as externally provided constraints. Additionally, although Koza uses a typeless genetic programming system, so-called “strongly typed” genetic programming provides constraints on the crossover and mutation operations.

Saito et al. [69] present two unusual algorithms. The RF5 equation discovery algorithm transforms polynomial equations into neural networks, trains the network with the standard back-propagation learning algorithm, then converts the trained network back into a polynomial equation. The RF6 discovery algorithm augments RF5 using a clustering technique and decision-tree induction.

One last qualitative reasoning system is called General Systems Problem Solving [42]. It describes a taxonomy of systems and an approach to identifying them. It is not clear if general systems problem solving has been applied and no references beyond the original author were found.

### 1.7.10 Information criteria—heuristics for choosing model order

Information criteria are heuristics that weight model fit (reduction in error residuals) against model size to evaluate a model. They try to guess whether a given model is sufficiently high order to represent the system under test but not so flexible as to be likely to over fit. These heuristics are not a replacement for the practice of building a model with one set of data and validating it with a second. The Akaike and Bayesian information criteria are two examples. The Akaike Information Criteria (AIC) is:

$$AIC = -2 \ln(L) + 2k \tag{1.20}$$

where  $L$  is the likelihood of the model, and  $k$  is the order of the model (number of free parameters). The likelihood of a model is the conditional probability of a given output,  $u$ , given the model,  $M$ , and model parameter values  $U$ .

$$L = P(u|M, U) \tag{1.21}$$

Bayesian Information Criteria (BIC) considers the size of the identification data set, reasoning that models with a large number of parameters but which

were identified from a small number of observations are very likely to be over fit and have spurious parameters. The Bayes information criterion is:

$$BIC = -2 \ln(L) + k \ln(n) \quad (1.22)$$

where  $n$  is the number of observations (i.e. sample size). The likelihood of a model is not always known however, and so must be estimated. One suggestion is to use a modified version based on the root mean square error (RMSE) instead of the model likelihood [3].

$$AIC \cong -2n \ln\left(\frac{RMSE}{n}\right) + 2k \quad (1.23)$$

Çinar [17] reports on building In building the nonlinear polynomial models in by increasing the number of terms until the Akaike information criteria (AIC) is minimized. This works because there is a natural ordering of the polynomial terms from lowest to highest order. Sjoberg et al. [73] point out that not all nonlinear models have a natural ordering of their components or parameters.

### 1.7.11 Fuzzy relational models

According to Sjoberg et al. [73], fuzzy set membership functions and inference rules can form a local basis function approximation (the same general class of approximations as wavelets and splines). A fuzzy relational model consists of two parts:

1. input and output fuzzy set membership functions
2. a fuzzy relation mapping inputs to outputs

Fuzzy set membership functions themselves may be arbitrary nonlinearities (triangular, trapezoidal, and Gaussian are common choices) and are sometime referred to as “linguistic variables” if the degree of membership in a set has some qualitative meaning that can be named. This is the sense in which fuzzy relational models are parametric: the fuzzy set membership functions describe the degrees to which specific qualities of the system are true. For example, in a speed control application there may be linguistic variables named “fast” and “slow” whose membership function need not be mutually exclusive. However, during the course of the identification method

described below the centres of the membership functions will be adjusted. They may be adjusted so far that their original linguistic values are not appropriate (for example if “fast” and “slow” became coincident). In that case, or when the membership functions, or the relations are initially generated by a random process, then fuzzy relational models should be considered non-parametric by the definition given in section 1.4 above. In the best case, however, fuzzy relational models have excellent explanatory power since the linguistic variables and relations can be read out in qualitative English-like statements such as “when the speed is fast, decrease the power output” or even “when the speed is somewhat fast, decrease the power output a little”.

Fuzzy relational models are evaluated to make a prediction in 4 steps:

1. The raw input signals are applied to the input membership functions to determine the degree of compatibility with the linguistic variables.
2. The input membership values are composed with the fuzzy relation to determine the output membership values.
3. The output membership values are applied to the output variable membership functions to determine a degree of activation for each.
4. The activations of the output variables is aggregated to produce a crisp result.

Composing fuzzy membership values with a fuzzy relation is often written in notation that makes it look like matrix multiplication when in fact it is something quite different. Whereas in matrix multiplication corresponding elements from one row and one column are multiplied and then those products are added together to form one element in the result, composition of fuzzy relations uses other operations in place of addition and subtraction. Common choices include minimum and maximum.

An identification method for fuzzy relational models is provided by Branco and Dente [15] along with its application to prediction of a motor drive system. An initial shape is assumed for the membership functions as are initial values for the fuzzy relation. Then the first input datum is applied to the input membership functions, mapped through the fuzzy relation and output membership functions to generate an output value from which an error is measured (distance from the training output datum). The centres of the membership functions are adjusted in proportion to the error signal integrated across the value of the immediate (pre-aggregation) value of that



membership function. This apportioning of error or “assignment of blame” resembles a similar step in the back-propagation learning algorithm for feed-forward neural networks. The constant of proportionality in this adjustment is a learning rate: too large and the system may oscillate and not converge, too slow and identification takes a long time. All elements of the relation are updated using the adjusted fuzzy input and output variables, and the relation from the previous time step.

Its tempting to view the membership functions as nonlinear input and output maps, similar to a combined Hammerstein-Wiener model, but composition with the fuzzy relation in the middle differs in that max-min composition is a memoryless nonlinear operation. The entire model, therefore, is static-nonlinear.

### 1.7.12 Graph-based models

Models developed from first principles are often communicated as graph structures. Familiar examples include electrical circuit diagrams and block diagrams or signal flow graphs for control and signal processing. Bond graphs ([66], [39], [40]) are a graph-based model form where the edges (called bonds) denote energy flow between components. The nodes (components) may be sources, dissipative elements, storage elements, or translating elements (transformers and gyrators, optionally modulated by a second input). Since these graph-based model forms are each already used in engineering communication, it would be valuable to have an identification technique that reports results in any one of these forms.

Graph-based models can be divided into 2 categories: direct transcriptions of differential equations, and those that contain more arbitrary components. The first includes signal flow graphs and basic bond graphs. The second includes bond graphs with complex elements that can only be described numerically or programmatically. Bond graphs are described in great detail in chapter 2.

The basic bond graph elements each have a constitutive equation that is expressed in terms of an effort variable ( $e$ ) and a flow variable ( $f$ ) which together quantify the energy transfer along connecting edges. The constitutive equations of the basic elements have a form that will be familiar from linear circuit theory but they can also be used to build models in other domains. When all the constitutive equations for each node in a graph are written out, with nodes linked by a bond sharing their effort and flow variables, the

result is a coupled set of differential and algebraic equations. The constitutive equations of bond graph nodes need not be simple integral formulae. Arbitrary functions may be used so long as they are amenable to whatever analysis or simulation the model is intended. In these cases, the bond graph model is no longer a simply a graphic transcription of differential equations.

There are several commercial software packages for bond graph simulation and bond graphs have been used to model some large and complex systems, including a 10 degree of freedom planar human gait model developed at the University of Waterloo [61]. A new software package for bond graph simulation is described in chapter 4 and several example simulations are presented in section 5.1.

Actual system identification using bond graphs requires a search through candidate models. This is combined structure and parameter identification. The structure is embodied in the types of elements (nodes) that are included in the graph and the interconnections between them (arrangement of the edges or bonds). The genetic algorithm and genetic programming have both been applied to search for bond graph models. Danielson et al. [20] use a genetic algorithm to perform parameter optimization on an internal combustion engine model. The model is a bond graph with fixed structure; only the parameters of the constitutive equations are varied. Genetic programming is considerably more flexible since, whereas the genetic algorithm uses a fixed length string, genetic programming breeds computer programs (that in turn output bond graphs of variable structure when executed). See [30] or [71] for recent applications of genetic programming and bond graphs to identifying models for engineering systems.

Another type of graph-based model used with genetic programming for system identification could be called a virtual analog computer. Streeter, Keane and Koza [76] used genetic programming to discover new designs for circuits that perform as well or better than patented designs given a performance metric but no details about the structure or components of the patented design. Circuit designs could be translated into models in other domains (e.g. through bond graphs, or by directly comparing differential equations from each domain). This is expanded on in book form [46] with an emphasis on the ability of genetic programming to produce parametrized topologies (structure identification) from a black-box specification of the behavioural requirements.

Many of the design parameters of a genetic programming implementation can also be viewed as placing constraints on the model space. For example, if

a particular component is not part of the genetic programming kernel, cannot arise through mutation, and is not inserted by any member of the function set, then models containing this component are excluded. In a strongly typed genetic programming system the choice of types and their compatibilities constrains the process further. Fitness evaluation is another point where constraints may be introduced. It is easy to introduce a “parsimony pressure” by which excessively large models (or those that violate any other *a priori* constraint) are penalized and discouraged from recurring.

## 1.8 Contributions of this thesis

This thesis contributes primarily a bond graph modelling and simulation library capable of modelling linear and non-linear (even non-analytic) systems and producing symbolic equations where possible. In certain linear cases the resulting equations can be quite compact and any algebraic loops are automatically eliminated. Genetic programming grammars for bond graph modelling and for direct symbolic regression of sets of differential equations are presented. The planned integration of bond graphs with genetic programming as a system identification technique is incomplete. However, a function identification problem was solved repeatedly to demonstrate and test the genetic programming system and the direct symbolic regression grammar is shown to have a non-deceptive fitness landscape—perturbations of an exact program have decreasing fitness with increasing distance from the ideal.

# Chapter 2

## Bond graphs

### 2.1 Energy based lumped parameter models

Bond graphs are an abstract graphical representation of lumped parameter dynamic models. Abstract in that the same few elements are used to model very different systems. The elements of a bond graph model represent common roles played by the parts of any dynamic system, such as providing, storing, or dissipating energy. Generally, each element models a single physical phenomenon occurring in some discrete part of the system. To model a system in this way requires that it is reasonable to “lump” the system into a manageable number of discrete parts. This is the essence of *lumped* parameter modelling: model elements are discrete, there are no continua (though a continuum may be approximated in a finite element fashion). The standard bond graph elements model just one physical phenomenon each, such as the storage or dissipation of energy. There are no combined dissipative–storage elements, for example, among the standard bond graph elements, although one has been proposed [55].

Henry M. Paynter developed the bond graph modelling notation in the 1950s [57]. Paynter’s original formulation has since been extended and refined. A formal definition of the bond graph modelling language [66] has been published and there are several textbooks [19], [40], [67] on the subject. Linear graph theory is a similar graphical energy based lumped parameter modelling framework that has some advantages over bond graphs including more convenient representation of multi-body mechanical systems. Birkett and Roe [8], [9], [10], [11] explain bond graphs and linear graph theory in

terms of a more general framework based on matroids.

## 2.2 Standard bond graph elements

### 2.2.1 Bonds

In a bond graph model, system dynamics are expressed in terms of the power transfer along “bonds” that join interacting sub-models. Every bond has two variables associated with it. One is called the “intensive” or “effort” variable. The other is called the “extensive” or “flow” variable. The product of these two is the amount of power transferred by the bond. Figure 2.1 shows the graphical notation for a bond joining two sub-models. The effort and flow variables are named on either side of the bond. The direction of the half arrow indicates a sign convention. For positive effort ( $e > 0$ ) and positive flow ( $f > 0$ ), a positive quantity of power is transferred from A to B. The points at which one model may be joined to another by a bond are called “ports”. Some bond graph elements have a limited number of ports, all of which must be attached. Others have unlimited ports, any number of which may be attached. In figure 2.1, systems A and B are joined at just one point each so they are “one-port elements”.

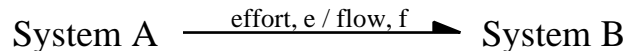


Figure 2.1: A bond denotes power continuity between two systems

The effort and flow variables on a bond may use any units so long as the resulting units of power are compatible across the whole model. In fact, different parts of a bond graph model may use units from another physical domain entirely. This makes bond graphs particularly useful for multi-domain modelling, such as in mechatronic applications. A single bond graph model can incorporate units from any row of table 2.1 or, indeed, any other pair of units that are equivalent to physical power. It needs to be emphasized that bond graph models are based firmly on the physical principal of power continuity. The power leaving A in figure 2.1 is exactly equal to the power entering B and this represents, with all the caveats of a necessarily inexact model, real physical power. When the syntax of the bond graph modelling

language is used in domains where physical power does not apply (as in economics) or where lumped-parameters are a particularly weak assumption (as in many thermal systems) the result is called a “pseudo bond graph”.

Table 2.1: Units of effort and flow in several physical domains.

	Intensive variable	Extensive variable	
Generalized terms	Effort	Flow	Power
Linear mechanics	Force ( $N$ )	Velocity ( $m/s$ )	( $W$ )
Angular mechanics	Torque ( $Nm$ )	Velocity ( $rad/s$ )	( $W$ )
Electrics	Voltage ( $V$ )	Current ( $A$ )	( $W$ )
Hydraulics	Pressure ( $N/m^2$ )	Volume flow ( $m^3/s$ )	( $W$ )

Two other generalized variables are worth naming. Generalized displacement,  $q$ , is the first integral of flow,  $f$ , and generalized momentum,  $p$ , is the integral of effort,  $e$ :

$$q = \int_0^T f dt \quad (2.1)$$

$$p = \int_0^T e dt \quad (2.2)$$

## 2.2.2 Storage elements

There are two basic storage elements: the capacitor and the inertia.

A capacitor, or C element, stores potential energy in the form of generalized displacement proportional to the incident effort. A capacitor in mechanical translation or rotation is a spring, in hydraulics it is an accumulator (a pocket of compressible fluid within a pressure chamber), in electrical circuits it is a charge storage device. The effort and flow variables on the bond attached to a linear one-port capacitor are related by equations 2.1 and 2.3. These are called the ‘constitutive equations’ of the C element. The graphical notation for a capacitor is shown in figure 2.2.

$$q = Ce \quad (2.3)$$

An inertia, or I element, stores kinetic energy in the form of generalized momentum proportional to the incident flow. An inertia in mechanical translation or rotation is a mass or flywheel, in hydraulics it is a mass-flow effect, in electrical circuits it is a coil or other magnetic field storage device. The constitutive equations of a linear one-port inertia are 2.2 and 2.4. The graphical notation for an inertia is shown in figure 2.3.

$$p = I f \tag{2.4}$$

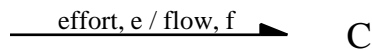


Figure 2.2: A capacitive storage element: the 1-port capacitor, C

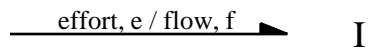


Figure 2.3: An inductive storage element: the 1-port inertia, I

### 2.2.3 Source elements

Ideal sources mandate a particular schedule for either the effort or flow variable of the attached bond and accept any value for the other variable. Ideal sources will supply unlimited energy as time goes on. Two things save this as a realistic model behaviour. First, models will only be run in finite length simulations and second, the environment is assumed to be vastly more capacious than the system under test in any respects where they interact.

The two types of ideal source element are called the effort source and the flow source. The effort source (Se, figure 2.4) mandates effort according to a schedule function  $E$  and ignores flow (equation 2.5). The flow source (Sf, figure 2.5) mandates flow according to  $F$  ignores effort (equation 2.6).

$$e = E(t) \tag{2.5}$$

$$f = F(t) \tag{2.6}$$

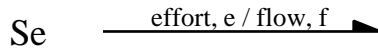


Figure 2.4: An ideal source element: the 1-port effort source, Se

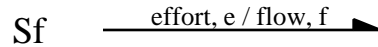


Figure 2.5: An ideal source element: the 1-port flow source, Sf

Within a bond graph model, ideal source elements represent boundary conditions and rigid constraints imposed on a system. Examples include: a constant valued effort source representing a uniform force field such as gravity near the earth’s surface, and a zero valued flow source representing a grounded or otherwise absolutely immobile point. Load dependence and other non-ideal behaviours of real physical power sources can often be modelled by bonding an ideal source to some other standard bond graph elements so that they together expose a single port to the rest of the model showing the desired behaviour in simulation.

### 2.2.4 Sink elements

There is one type of element that dissipates power from a model. The resistor, or R element, is able to sink an infinite amount of energy as time goes on. This energy is assumed to be dissipated into a capacious environment outside the system under test. The constitutive equation of a linear one-port resistor is 2.7, where R is a constant value called the “resistance” associated with a particular resistor. The graphical notation for a resistor is shown in figure 2.6.

$$e = Rf \tag{2.7}$$

### 2.2.5 Junctions

There are four basic multi-port elements that do not source, store, or sink power but are part of the network topology that joins other elements to-



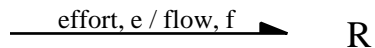


Figure 2.6: A dissipative element: the 1-port resistor, R

gether. They are called the transformer, the gyrator, the 0-junction, and the 1-junction. Power is conserved across each of these elements. The transformer and gyrator are two-port elements. The 0- and 1-junctions permit any number of bonds to be attached (but are redundant in any model where they have less than 3 bonds attached).

The graphical notation for a transformer, or TF-element, is shown in figure 2.7. The constitutive equations for a transformer, relating  $e_1, f_1$  to  $e_2, f_2$  are given in equations 2.8 and 2.9 where  $m$  is a constant value associated with the transformer, called the “transformer modulus”. From these equations it can be easily seen that this ideal, linear element conserves power since (incoming power)  $e_1 f_1 = m e_2 f_2 / m = e_2 f_2$  (outgoing power). In mechanical terms, a TF-element may represent an ideal gear train (with no friction, backlash, or windup) or lever. In electrical terms, it is a common electrical transformer (pair of windings around a single core). In hydraulic terms it could represent a coupled pair of rams with differing plunger surface areas.

$$e_1 = m e_2 \tag{2.8}$$

$$f_1 = f_2 / m \tag{2.9}$$

The graphical notation for a gyrator, or GY-element, is shown in figure 2.8. The constitutive equations for a gyrator, relating  $e_1, f_1$  to  $e_2, f_2$  are given in equations 2.10 and 2.11 where  $m$  is a constant value associated with the gyrator, called the “gyrator modulus”. From these equations it can be easily seen that this ideal, linear element conserves power since (incoming power)  $e_1 f_1 = m f_2 e_2 / m = f_2 e_2$  (outgoing power). Physical interpretations of the gyrator include gyroscopic effects on fast rotating bodies in mechanics, and Hall effects in electro-magnetics. These are less obvious than the TF-element interpretations but the gyrator is more fundamental in one sense: two GY-elements bonded together are equivalent to a TF-element, whereas two TF-elements bonded together are only equivalent to another transformer. Models from a reduced bond graph language having no TF-elements are called “gyro-bond graphs” [65].

$$e_1 = m f_2 \tag{2.10}$$

$$f_1 = e_2/m \quad (2.11)$$



Figure 2.7: A power-conserving 2-port element: the transformer, TF



Figure 2.8: A power-conserving 2-port element: the gyrator, GY

The 0- and 1-junctions permit any number of bonds to be attached directed power-in ( $b_{in_1} \dots b_{in_n}$ ), and any number directed power-out ( $b_{out_1} \dots b_{out_m}$ ). These elements “join” either the effort or the flow variable of all attached bonds, setting them equal to each other. To conserve power across the junction the other bond variable must sum to zero over all attached bonds with bonds directed power-in making a positive contribution to the sum and those directed power-out making a negative contribution. These constitutive equations are given in equations 2.12 and 2.13 for a 0-junction, and in equations 2.14 and 2.15 for a 1-junction. The graphical notation for a 0-junction is shown in figure 2.9 and for a 1-junction in figure 2.10.

$$e_1 = e_2 \dots = e_n \quad (2.12)$$

$$\sum f_{in} - \sum f_{out} = 0 \quad (2.13)$$

$$f_1 = f_2 \dots = f_n \quad (2.14)$$

$$\sum e_{in} - \sum e_{out} = 0 \quad (2.15)$$

The constraints imposed by 0- and 1-junctions on any bond graph model they appear in are analogous to Kirchhoff’s voltage and current laws for electric circuits. The 0-junction sums flow to zero across the attached bonds just like Kirchhoff’s current law requires that the net electric current (a flow variable) entering a node is zero. The 1-junction sums effort to zero across the

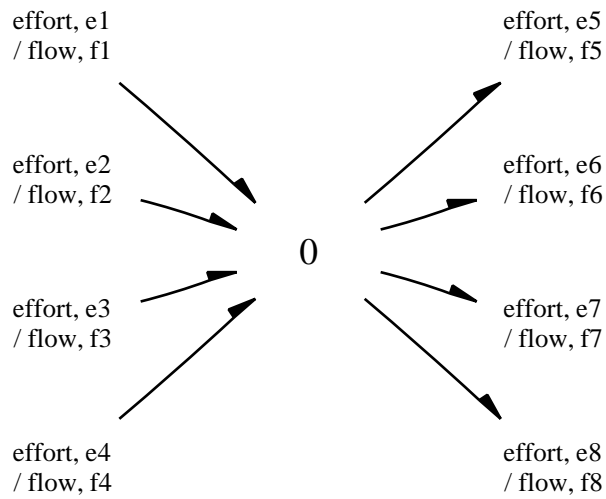


Figure 2.9: A power-conserving multi-port element: the 0-junction

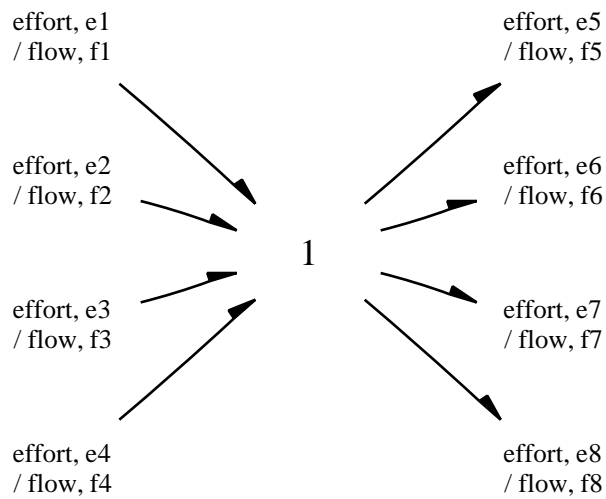


Figure 2.10: A power-conserving multi-port element: the 1-junction

attached bonds just like Kirchhoff’s voltage law requires that the net change in electric potential (an effort variable) around any loop is zero. Naturally, any electric circuit can be modelled with bond graphs.

The 0- and 1-junction elements are sometimes also called f-branch and e-branch elements, respectively since they represent a branching points for flow and effort within the model [19].

## 2.3 Augmented bond graphs

Bond graph models can be augmented with an additional notation on each bond to denote “computational causality”. The constitutive equations of the basic bond graph elements in section 2.2 could easily be rewritten to solve for the effort or flow variable on any of the attached bonds in terms of the remaining bond variables. There is no physical reason why they should be written one way or another. However, in the interest of solving the resulting set of equations, it is helpful to choose one variable as the “output” of each element and rearrange to solve for it in terms of the others (the “inputs”). The notation for this choice is a perpendicular stroke at one end of the bond, as shown in figure 2.11. By convention, the effort variable on that bond is considered an input to the element closest to the causal stroke. The location of the causal stroke is independent of the power direction half arrow. In both parts of figure 2.11 the direction of positive power transfer is from A to B. In the first part effort is an input to system A (output from system B) and flow is an input to system B (output from system A). The equations for this system would take the form:

$$\begin{aligned} flow &= A(effort) \\ effort &= B(flow) \end{aligned}$$

In the second part of figure 2.11 the direction of computational causality is reversed. So, although the power sign convention and constitutive equations of each element are unchanged, the global equations for the entire model are rewritten in the form:

$$\begin{aligned} flow &= B(effort) \\ effort &= A(flow) \end{aligned}$$

The concept discussed here is called *computational* causality to emphasize the fact that it is an aid to extracting equations from the model in a form that

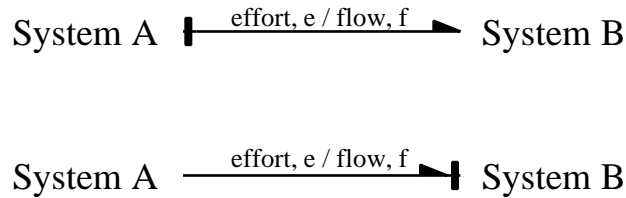


Figure 2.11: The causal stroke and power direction half-arrow on a fully augmented bond are independent

is easily solved in a computer simulation. It is a practical matter of choosing a convention and does not imply any philosophical commitment to an essential ordering of events in the physical world. Some recent modelling and simulation tools that are based on networks of ported elements, but not exactly bond graphs, do not employ any concept of causality [2] [56]. These systems enjoy a greater flexibility in composing subsystem models into a whole as they are not burdened by what some argue is an artificial ordering of events<sup>1</sup>. Computational causality, they argue, should be determined automatically by bond graph modelling software and the software’s user not bothered with it [16]. The cost of this modelling flexibility is a greater complexity in formulating a set of equations from the model and solving those in simulation (in particular very general symbolic manipulations may be required resulting in a set of implicit algebraic-differential equations to be solved rather than a set of explicit differential equations in state-space form; see section 2.5.1). The following sections show how a few simple causality conventions permit the efficient formulation of state space equations where possible and the early identification of cases where it is not possible.

<sup>1</sup>Consider this quote from Francois E. Cellier at <http://www.ece.arizona.edu/~cellier/psml.html> “Physics, however, is essentially acausal (Cellier et al., 1995). It is one of the most flagrant myths of engineering that algebraic loops and structural singularities in models result from neglected fast dynamics. This myth has its roots in the engineers’ infatuation with state-space models. Since state-space models are what we know, the physical realities are expected to accommodate our needs. Unfortunately, physics doesn’t comply. There is no physical experiment in the world that could distinguish whether I drove my car into a tree, or whether it was the tree that drove itself into my car.”

### 2.3.1 Integral and derivative causality

Tables 2.2 and 2.3 list all computational causality options and the resulting equations for each of the standard bond graph elements.

Source elements, since they mandate a particular schedule for one variable, have only one causality option: that in which the scheduled variable is an output of the source element. The sink element (R) and all power conserving elements (TF, GY, 0, 1) have two causality options. Either option will do for these elements since there is no mandate (as for source elements) and since the constitutive equations are purely algebraic and (for linear elements, at least) easily inverted.

The storage elements (C, I) have two causality options as well. In this case, however, there are reasons to prefer one form over the other. The preference is always for the form that yields differential expressions on the left-hand side and not the right-hand side of any equations. Table 2.2 shows that this is causal effort-out for capacitors and causal effort-in for inductors. Since the variables on the right-hand side of these preferred differential equations are inputs to the element (flow for a capacitor and effort for an inductor), the variable in the differential expression can be evaluated by integration over time. For this reason, these preferred forms are called “integral causality” forms. In physical terms, the opposing forms, called “derivative causality”, occur when the energy variable of a storage element depends directly, that is without another type of storage element or a resistor between them, on another storage element (storage-storage dependency) or on a source element (storage-source dependency) of the model [51]. Two examples of derivative causality follow.

The electric circuit shown schematically in figure 2.12 contains two capacitors connected in parallel to some larger circuit. A corresponding bond graph model fragment is shown next to it. If one capacitor is first assigned the preferred, effort-out causality, that bond is the effort-in to the junction. Since a 0-junction permits only one attached bond to be causal effort-in, the rest must be effort-out forcing the bond attached to the second capacitor to be effort-in with respect to the capacitor. In electrical terms, the charges stored in these capacitors are not independent. If the charge on one is known, then the voltage across it is known ( $e = q/C$ ). Since they are connected directly in parallel with on another, it is the same voltage across both capacitors. Therefore the charge on the second capacitor is known as well ( $q = eC$ ). They act in effect as a single capacitor (with summed capacitance)

Table 2.2: Causality options by element type: 1- and 2-port elements.

















Element	Notation	Equations
Effort source	Se 	$e = E(t)$
Flow source	Sf 	$f = F(t)$
Capacitor	 C	$q = Ce$ $f = \frac{dq}{dt}$
	 C	$e = q/C$ $\frac{dq}{dt} = f$
Inertia	 I	$f = p/I$ $\frac{dp}{dt} = e$
	 I	$p = If$ $e = \frac{dp}{dt}$
Resistor	 R	$f = e/R$
	 R	$e = Rf$
Transformer	 TF 	$e_1 = me_2$ $f_2 = mf_1$
	 TF 	$f_1 = mf_2$ $e_2 = me_1$
Gyrator	 GY 	$e_1 = rf_2$ $e_2 = rf_1$
	 GY 	$f_1 = e_2/r$ $f_2 = e_1/r$

Table 2.3: Causality options by element type: junction elements.

Element	Notation	Equations
0-Junction	( $b^*$ directed power-in)	$e^* \equiv e_{in_1}$ $f^* \equiv f_{in_1}$ $e_{in_2} = \dots = e_{in_n} = e^*$ $e_{out_1} = \dots = e_{out_m} = e^*$ $f^* = \sum_1^m f_{out} - \sum_2^n f_{in}$
	( $b^*$ directed power-out)	$e^* \equiv e_{out_1}$ $f^* \equiv f_{out_1}$ $e_{in_1} = \dots = e_{in_n} = e^*$ $e_{out_2} = \dots = e_{out_m} = e^*$ $f^* = \sum_1^n f_{in} - \sum_2^m f_{out}$
1-Junction	( $b^*$ directed power-in)	$e^* \equiv e_{in_1}$ $f^* \equiv f_{in_1}$ $f_{in_2} = \dots = f_{in_n} = f^*$ $f_{out_1} = \dots = f_{out_m} = f^*$ $e^* = \sum_1^m e_{out} - \sum_2^n e_{in}$
	( $b^*$ directed power-ou)	$e^* \equiv e_{out_1}$ $f^* \equiv f_{out_1}$ $f_{in_1} = \dots = f_{in_n} = f^*$ $f_{out_2} = \dots = f_{out_m} = f^*$ $e^* = \sum_1^n e_{in} - \sum_2^m e_{out}$

- 0- and 1-junctions permit any number of bonds directed power-in ( $b_{in_1} \dots b_{in_n}$ ), and any number directed power-out ( $b_{out_1} \dots b_{out_m}$ ).
- A 0-junction permits only one connected bond ( $b^*$ ) to be causal effort-in (with the causal stroke adjacent to the junction). The rest must be effort-out.
- A 1-junction permits only one connected bond ( $b^*$ ) to be causal effort-out (with the causal stroke away from the junction). The rest must be effort-in.



and have only one degree of freedom between them.

For a mechanical example, consider two masses rigidly joined such as the point masses on a rigid lever shown in figure 2.13. The bond graph model shown in that figure is applicable for small rotations of the lever about horizontal. The dependency between storage elements is not as direct here as in the previous example. The dependent elements (two I-elements) are linked through three other elements, not just one. Nevertheless, choosing integral causality for one inertia forces derivative causality on the other as a consequence of the causality options for the three intervening elements. Here again there are two storage elements in the model but only one degree of freedom.

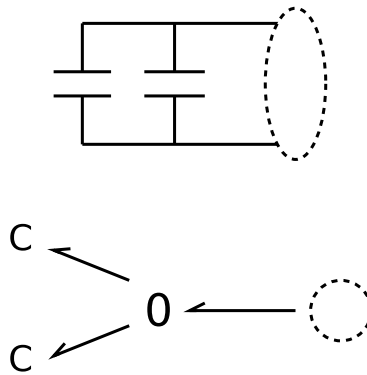


Figure 2.12: A system having dependent dynamic parts and a corresponding model having storage elements with derivative causality: electrical capacitors in parallel and C elements with common effort

### 2.3.2 Sequential causality assignment procedure

There is a systematic method of assigning computational causality to bond graph models so as to minimize computational complexity and consistently arrive at a state space formulation where possible. This method is called the sequential causality assignment procedure (SCAP) [40], [67] and it proceeds in three stages. After each stage, the consequences of currently assigned causality are propagated as far as possible through the graph. For example, if a bond has been assigned effort-in causality to a 0-junction then the other bonds attached to that junction must all be causal effort-out from that junc-

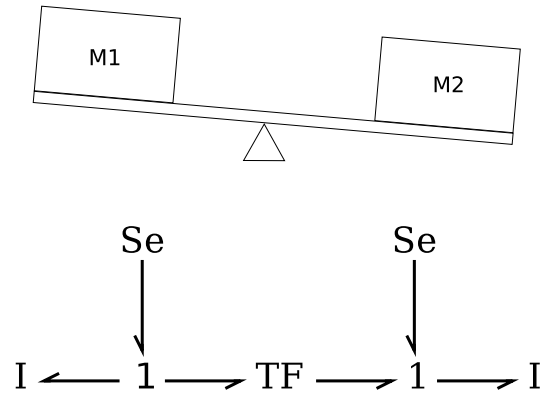


Figure 2.13: A system having dependent dynamic parts and a corresponding model having storage elements with derivative causality: rigidly joined masses and I elements with proportionally common flow

tion and any of those bonds that do not yet have a causal assignment can be assigned at that time. The three stages of SCAP are:

1. Assign the required causality to all bonds attached to sources.
2. Assign the preferred causality to any unassigned bonds attached to storage elements.
3. Assign arbitrary (perhaps random) causality to any unassigned bonds attached to sink elements.

If unassigned bonds attached to nodes of any one type are always handled in a consistent order then the assigned causalities, and therefore also the collected equations, will be stable (identical for identical graphs).

Consider the fully augmented Se-C-I-R model in figure 5.2. In the first stage of SCAP, bond 4 would be assigned causality effort-out from the source (effort-in to the junction). Since 1-junctions allow any number of attached bonds to be causal effort-in, there are no further consequences of this assignment. In the second stage, storage elements are assigned their preferred causality. If the I-element is chosen first and the attached bond is assigned the preferred causality (effort-in to the inertia, effort-out from the junction) then the remaining bonds attached to the junction must be causal effort-in and the graph is completely augmented. If instead the C-element is chosen

before the I-element, it is still assigned the preferred causality. Consequences do not propagate any further and the I-element is selected after all. In either case, as is often the case, all bonds attached to sink elements are assigned by propagation of consequences and stage 3 is never reached.

## 2.4 Additional bond graph elements

### 2.4.1 Activated bonds and signal blocks

An activated bond is one on which either the effort or flow variable is considered negligibly small and the other is called the signal. Activated bonds carry information in the signal but no power, no matter how large a value the signal may take on. An activated bond is drawn with a full arrow at one end indicating the signal-flow direction.

Any bond graph can be transformed into a signal-flow graph by splitting each bond into a pair of activated bonds carrying the effort and flow signals. The reverse is not true since bond graphs encode physical structure in ways that a signal graph is not required to conform to. Arbitrary signal-flow graphs may contain odd numbers of signals or topologies in which the signals can not be easily or meaningfully paired.

Signal blocks are elements that do not source, store, or sink power but instead perform operations on signals and can only be joined to other elements by activated bonds. In this way, activated bonds form a bridge between bond graphs and signal-flow graphs allowing the techniques to be used together in a single model.

### 2.4.2 Modulated elements

Activated bonds join standard bond graph elements at a junction, where convention or the signal name denotes which of the effort or flow variable is carried, or at modulated elements. If the modulus of a transformer (TF) or gyrator (GY) is not constant but provided instead by the signal on an activated bond, a modulated transformer (MTF) or modulated gyrator (MGY) results.



Figure 2.14: A power-conserving 2-port: the modulated transformer, MTF



Figure 2.15: A power-conserving 2-port: the modulated gyrator, MGY

### 2.4.3 Complex elements

Section 2.4.2 showed how certain non-linear effects can be modelled using modulated transformer and gyrator elements. [40] presents a multi-port MTF node with more than two power bonds. Single- and multi-port elements with complex behaviour, such as discontinuous or even non-analytic effort–flow relations are also possible. In the non-analytic case, element behaviour would be determined empirically and implemented as a lookup table with interpolation for simulation. These effort–flow relations are typically not invertible, and so these elements, like the ideal source elements introduced in section 2.2, have a required causality.

Figure 2.16 shows an example of an element with discontinuous constitutive equations. The element is labelled *CC* for *contact compliance*; the mechanical system from which this name derives (perpendicular impact between an elastic body and a rigid surface) is shown in figure 2.17. Constitutive equations for the *CC* element in the shown causal assignment are given in equations 2.16 and 2.17. The notation  $(q < r)$  is used to indicate 0 when  $q > r$  and 1 otherwise, where  $r$  is called the *threshold* and is a new parameter particular to this specialized variant of the C element. In the mechanical

interpretation of figure 2.17, these equations are such that the force exerted by the spring is proportional to the compression of the spring when it is in contact with the surface, and zero otherwise. The effort–displacement relationship of the CC element is plotted in figure 2.18. Behaviour of the 1-port CC element with effort-in causality is undefined. Section 5.1.3 includes an application of the CC element.

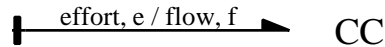


Figure 2.16: A non-linear capacitive storage element: the 1-port contact compliance, CC

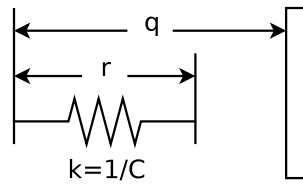


Figure 2.17: Mechanical contact compliance: an unfixed spring exerts no force when  $q$  exceeds the unsprung length  $r$

$$e = (q < r)(r - q)/C \quad (2.16)$$

$$\frac{dq}{dt} = f \quad (2.17)$$

#### 2.4.4 Compound elements

The above example shows how a new bond graph element with arbitrary properties can be introduced. It is also common to introduce new compound elements that stand in place of a particular combination of other bond graph elements. In this way, complex models representing complex systems can be built in a modular fashion by nesting combinations of simpler models representing simpler subsystems. At simulation time, a preprocessor may replace compound elements by their equivalent set of standard elements or,

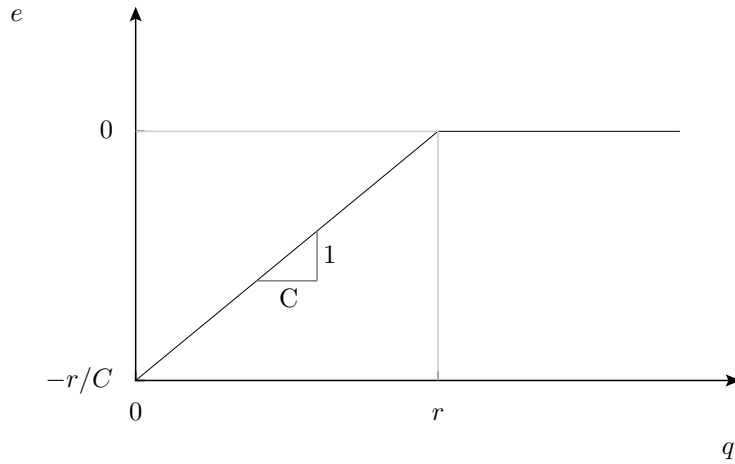


Figure 2.18: Effort-displacement plot for the CC element shown in figure 2.16 and defined by equations 2.16 and 2.17

compound elements may be implemented such that they directly output a set of constitutive equations that is entirely equivalent to those of the standard elements being abstracted.

A library of three-dimensional joints for use in modelling multibody mechanics is given in [86]. Other libraries of reusable sub-models are presented in [19] and [4].

For a modelling application in mechanics, consider the planar mechanism shown in figure 2.19. A slender mass is pin jointed at one end. The pin joint itself is suspended in the horizontal and vertical directions by a spring-damper combination. A bond graph model of this system is given in figure 5.12. The graph is reproduced as figure 2.20 and divided into two parts representing the swinging mass (“link”) and the suspended pin joint (“joint”). The sub-models are joined at only two ports, bonds 20 and 23, representing the horizontal and vertical coupling of the pin joint to point A on the link. If two new, compound bond graph elements were introduced subsuming the sub-graphs marked in figure 2.20, the entire model could be drawn as in figure 2.21.

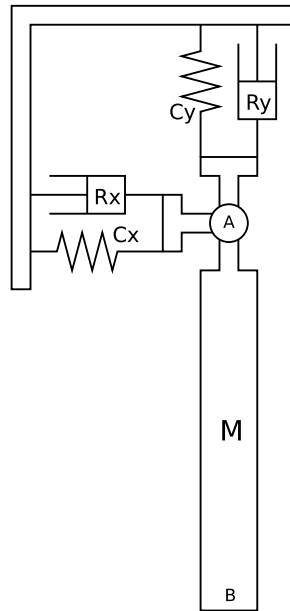


Figure 2.19: A pendulum with damped-elastic suspension

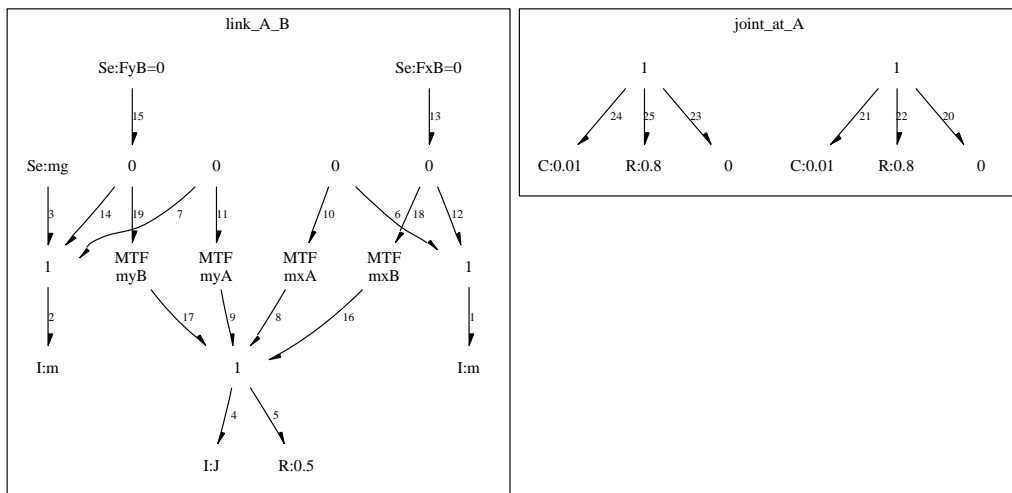


Figure 2.20: Sub-models of the simple pendulum depicted in figure 2.19 and modelled in figure 5.12

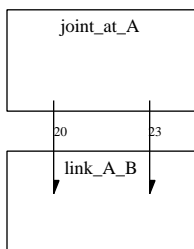


Figure 2.21: Model of a simple pendulum as in figures 5.12 and 2.20, rewritten using specialized, compound elements

## 2.5 Simulation

### 2.5.1 State space models

The instantaneous state of an energy-based model is precisely described by a full accounting of the distribution of energy within the system. For a bond graph built from the standard elements, this is simply the value of the energy variable of each storage element in the model. If these values are assembled into a vector, that vector locates the model in a “state space”.

The “state space equations” express the first derivative of the state vector in terms of state and input variables only (called the “state update” equation) and express any non-state, non-input variables in terms of the state variables and the input variables (the “readout” equation). For a system with inputs  $\vec{x}$ , state vector  $\vec{u}$ , and outputs  $\vec{y}$ , the “state update” equation is 2.18 and the “readout” equation is 2.19.

$$\frac{d\vec{u}}{dt} = f(\vec{x}, \vec{u}) \quad (2.18)$$

$$\vec{y} = g(\vec{x}, \vec{u}) \quad (2.19)$$

The procedure to form the state space equations of a bond graph model is straight forward provided the constitutive equations for each element come from table 2.2 or 2.3 and all storage elements have integral causality. Note that every equation in tables 2.2 and 2.3 is written in an assignment statement form, having just one variable (or the first time-derivative of a variable) alone on the left-hand side. If, following the procedure described in Section 2.3.2, causality has been assigned to all bonds and the result shows integral



causality for all storage elements, then every variable occurring in the constitutive equations of the elements will appear on the left-hand side of one and only one of the collected equations. The state update equation is formed from those constitutive equations having a derivative on the left-hand side. The energy variable of any storage element with integral causality will become a state variable (one element in the state vector). The procedure is to replace each variable on the right-hand side of these equations with their “definition” (the right-hand side of the equation for which the variable appears on the left-hand side) repeatedly back-substituting until only input and state variables appear on the right-hand side. Applying the same procedure to the remaining (non-state update) equations yields the readout equations for every non-state variable.

## 2.5.2 Mixed causality models

When a model contains one or more storage elements with derivative causality, it is not possible to reduce the constitutive equations of that model to state space form. The result instead is a set of implicit differential-algebraic equations in which either some differential expressions must occur on the right-hand side of some equations or, some variables must occur only on the right-hand sides and never on the left-hand side of any equations. This is inconvenient since there are ready and well known techniques for solving state space equation sets in simulation.

An experimenter faced with such a model has two choices: either reach for a more powerful (and inevitably more computationally expensive) solution method for the simulation, or rewrite the model to eliminate any occurrence of derivative causality. In the first case, the numerical solution techniques are more expensive than those for state space equations since a set of implicit algebraic equations must be solved iteratively (e.g. by a root finding algorithm) at each time step. In the second case, the approach varies with the system, the purpose of the model, and the preference of the experimenter.

One approach to revising a mixed or differential causality model is to combine dependent elements. In the case of the parallel electrical capacitors model shown in figure 2.12 it is a simple matter of replacing the two C-elements with a single one having the sum of their capacitance. For the linked masses in figure 2.13 or models having even longer and more complicated linkages, possibly spanning sub-models of different physical domains, the choice of which elements to combine and how is not at all obvious.

A second approach is to decouple the dependent storage elements by introducing additional decoupling elements between them, raising the number of degrees of freedom of the model. For example, if a small resistance were added between the capacitors in figure 2.12 then they would no longer be directly linked. The charge stored in each would be independent and the model would have as many degrees of freedom as storage elements. In the rigidly coupled masses example (figure 2.13) the addition of an R-element would, in effect, make the lever plastic. A C-element could make the lever elastic. Either way the inertias would be freed to move independently and a state-space formulation would then be possible.

# Chapter 3

## Genetic programming

### 3.1 Models, programs, and machine learning

Genetic programming is the application of genetic algorithms to the task of creating computer programs. It is a “biologically inspired, domain-independent method... that automatically creates a computer program from a high-level statement of a problem’s requirements” [45].

A method that automatically creates computer programs will be of interest to anyone doing system identification since a computer program and a model (model in the sense of chapter 1) are very similar at their most abstract and formal description. A computer program is something that receives informational input and performs computations on that input to produce an informational output. A system model receives input signals and performs transformations on those signals to produce output signals.

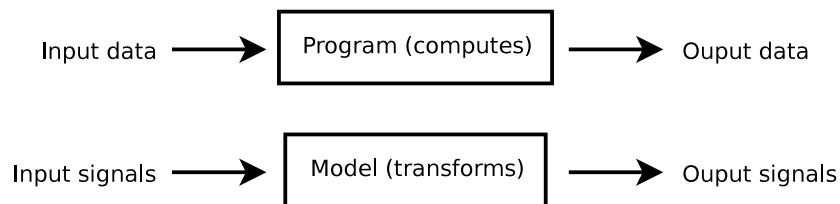


Figure 3.1: A computer program is like a system model.

The method popularized by Koza in [43], [44], [45], [46] and used here is a genetic algorithm applied to computer programs stored as concrete syntax trees in a functional programming language. Genetic algorithms are one

form of “evolutionary computing”, which includes also evolution strategies and others. All of evolutionary computing is in turn part of the broader class of adaptive heuristic search algorithms inspired by natural processes, which includes also simulated annealing, swarm optimization, ant colony optimization, and others.

The canonical genetic algorithm is similar to some other common machine learning algorithms, such as back propagation learning in multilayer perceptron networks, in that it performs a local search for minima on an error surface. However, whereas back-propagation learning in neural networks tests one point on the error surface at a time, a genetic algorithm explores many points on the surface in parallel. The objective of a genetic algorithm need not have any particular structure, such as a network or a tree, so long as the candidate solutions can be encoded in a form that is amenable to the operations described in the next section.

## 3.2 History of genetic programming

Early experiments in evolving computer programs were performed by Fogel, Owens and Walsh under the title Evolutionary Programming [26]. In that work, populations of finite state automata were evolved to solve discrete time series prediction problems. The reproduction and mutation genetic operators (asexual reproduction) were used to search for individual solutions inhabiting minima on an error surface (fitness maxima). Crossover (sexual reproduction) was not used and a problem-specific encoding of solutions as finite state automata limited the generality of the method.

In 1981, Richard Forsyth published a pattern recognition system using genetic principals to breed tree structured Boolean proposition statements as classifiers [27]. In this work Forsyth notes the generality of “Naturalistic Selection” in contrast to the many limiting assumptions inherent in more specialized statistical classifiers and also makes the important observation that the evolved rules are plainly and comprehensibly expressed, which facilitates human-machine collaboration.

Using a tree structured encoding of solutions allows the size and shape of individuals to vary so that these need not be known *a priori*<sup>1</sup>. [18] also

---

<sup>1</sup>In an interesting early link to other soft computing techniques, under “future work” Forsyth suggests using fuzzy logic in place of crisp Boolean propositions and references L.A. Zadeh “Fuzzy Sets” in Information and Control 8, 1965.

used genetic operations to search for tree structured programs in a special purpose language. Here the application was in symbolic regression.

A tree structured encoding of individuals would later prove to be very important. Generating machine code or FORTRAN without regard to syntax fails to yield acceptable results ([22]) because the distribution of syntactically correct, and therefore executable, programs within the search space of all possible combinations of, say, FORTRAN key words is far too sparse. Rather, it is preferable to limit the search to finding a program which solves the problem from within the (very much smaller) space of all syntactically correct programs.

Program syntax may be checked by generating the parse tree as would a compiler. Genetic operations may then be chosen which operate on the programs parse, or syntax tree directly and in syntax preserving ways. For fitness testing, the parse tree is converted into executable code. In 1992 John Koza published a textbook “Genetic Programming: on the programming of computers by means of natural selection” [43] which very much set the standard for the future work in Genetic Programming. In it he shows an implementation of a program induction system in LISP, a general purpose programming language, using the reproduction, crossover and mutation operators on individual program syntax trees. LISP was a particularly convenient choice for two reasons. First, the syntax or parse tree of LISP programs is very directly expressed in LISP source code. An example of this dual representation is shown in figure 6.

Second, LISP is a highly dynamic language with introspective capabilities. That is, a program written in LISP may read and modify its own source code while it is running. This allowed John Koza to write a Genetic Programming framework in LISP which operated on other LISP programs (the candidate solutions) and could load and run them as needed for fitness testing. For these reasons most genetic programming implementations since have been written in LISP although C is also used [31], [63] and many experimenter still use problem-specific languages. Improved performance has been shown by using genetic programming to evolve low level yet general purpose machine code for a stack-based virtual machine [58]. For demonstrating human competitive artificial intelligence it will be desirable to use a general purpose high level programming language popular among human programmers. “Grammatical evolution” is a genetic algorithm that can evolve complete programs in any recursive language using a variable-length linear chromosome to encode mappings from a Backus Naur Form (BNF) language

definition to individual programs [68]. This technique has been demonstrated using a BNF language definition consisting of a subset of C where the resulting programs were compiled with a standard C compiler and executed to test their fitness [54].

Today there are several journals and annual conferences dedicated to genetic programming and algorithms in general.

### 3.3 Genetic operators

In an analogy to natural evolution, genetic algorithms maintain a population of candidate solutions to the problem that are selectively reproduced and recombined from one generation to the next. The initial generation is created by arbitrary, and usually random, means. There are three operations by which the next generation individual candidate solutions are created from the current population.

1. Sexual reproduction (“crossover”)
2. Asexual reproduction (“replication”)
3. Mutated asexual reproduction (“mutation”)

The words “sexual” and “asexual” do not mean that there is any analogy to gender in the genetic algorithm. They simply refer to operations in which two “parent” candidate solutions come together to create a “child” candidate and to operations in which only one “parent” candidate is used.

These operations are applied to an encoded version of the candidate solutions. Candidates are maintained in this form from generation to generation and only converted back to their natural expression in the problem domain when they are evaluated, once per generation. The purpose of this encoding is so that the operations listed above need not be redesigned to apply to the form of the solution to each new type of problem. Instead, they are designed to apply to the encoded form and an encoding is designed for each new problem type. The usual encoded form is a bit string, a fixed-length sequence of zeros and ones. The essential property that a bit string encoding must have for the genetic operators to apply is that every possible combination of zeros and ones represents a syntactically valid, if hopelessly unfit, solution to the problem.

### 3.3.1 The crossover operator

The crossover operator is inspired by sexual reproduction in nature. Specifically, genetic recombination in which parts of the genetic material of both parents are combined to form the genetic material of their offspring. For fixed-length bit string genotypes, “single point” crossover is used. In this operation, a position along either of the equal-length parents is chosen randomly. Both parent bit strings are split at this point (hence *single* point) and the latter parts are swapped. Two new bit strings are created that are complimentary offspring of the parents. The operation is depicted in figure 3.3.1.

Parents:		aaaaaa		bbbbbb
		aaaa	aa	bbbb bb
Children:		aaaabb		bbbbaa

Figure 3.2: The crossover operation for bit string genotypes

### 3.3.2 The mutation operator

The mutation operator is inspired by random alterations in genetic material that give rise to new, previously unseen traits in nature. The mutation operator applied to bit string genotypes replaces one randomly selected bit with its compliment. The operation is depicted in figure 3.3.

Parent:		aaaaaa
		aaaa a a
Child:		aaaaba

Figure 3.3: The mutation operation for bit string genotypes

### 3.3.3 The replication operator

In the replication operation a bit string is copied verbatim into the next generation.

Parent:	aaaaaa
Child:	aaaaaa

Figure 3.4: The replication operation for bit string genotypes

### 3.3.4 Fitness proportional selection

The above three operators (crossover, mutation, replication) apply to directly genetic material and describe how new genetic material is created from old. Also needed is some way of selecting which old genes to operate on. In an analogy to “survival of the fittest” in nature, individual operands are chosen at random with a bias towards selecting the more fit individuals. Selection in exact proportion to the individual fitness scores is often visualized as a roulette wheel on which each individual has been allocated a wedge whose angular extent is proportional to the individual’s portion of the sum of all fitness scores in the population. A uniform random sample from the circumference of the wheel, equivalent to dropping a ball onto this biased wheel, will have a chance of selecting each individual that is exactly proportional to that individual’s fitness score.

This operation (fitness proportional selection) is applied before each occurrence of a genetic operator to choose the operands. In this way it tends to be the more fit individuals which are used in crossover, mutation, and replication to create new individuals. No individual in any generation is entirely ruled out of being selected occasionally. However, less fit individuals are less likely to be selected and sequences within the bit string genome which are present in less fit individuals but not in any particularly fit individuals will tend to be less prevalent in subsequent generations.

## 3.4 Generational genetic algorithms

Genetic algorithms can be divided into two groups according to the order in which new individuals are added to the population and others removed. In a “steady state” genetic algorithm, new individuals are created one or two at a time by applying one of the genetic operators to a selection from the existing population. Thus, the population evolves incrementally without a clear demarcation of one generation from the next. In a “generational” genetic algorithm, the entire population is replaced at once. A new population



of the same size as the old is built by selecting individuals from the old and applying genetic operators to them. Then the old population is discarded. The more popular generational form is described below and used throughout. A comparison of the performance of steady state and generational algorithms is given in [64].

The generational form of a genetic algorithm proceeds as follows [52]:

1. Choose a representation for solutions to the problem as bit strings of some fixed length  $N$
2. Choose the relative probabilities of crossover  $P_C$ , mutation  $P_M$ , and replication  $P_R = 1 - (P_C + P_M)$
3. Choose the population size,  $M$
4. Define termination criteria in terms of best fitness, number of generations, or a length of time
5. Generate an initial, random population of  $M$  individuals
6. Evaluate the fitness of each individual in the population
7. If any of the termination criteria are met, halt
8. Make a weighted random selection of a genetic operator
9. Select one or two (dependent on the operator) individuals at random weighted in proportion to their fitness.
10. Apply the operator to the selected individuals, creating a new individual.
11. Repeat steps 8–10 until  $M$  new individuals have been created
12. Replace the old population with the newly created individuals
13. Go to step 6

This is the form originally presented by Holland [34] and is also called the canonical genetic algorithm [84]. See also figure 3.4 for a simplified flowchart.

It is customary for the members of the initial population to be randomly generated but they may just as well be drawn from any configuration that

needs to be adapted to requirements encoded in a fitness test. Genetic algorithms are adaptive. They respond to selective pressures encoded in the fitness test at the start of each generation, even if the fitness test has changed between generations. In this way, genetic algorithms can also be used to track, and home in on, a moving objective.

Genetic algorithms are also open ended searches. They may be stopped at any time to yield their best-of-generation candidate. Typically, the termination criteria are expressed in terms of both a desired solution fitness level (performance criteria) and either a number of generations or a length of time (resource criteria). The fitness level is set to stop the search as soon as a satisfactory result is achieved. The running time or number of generations represent the length of time an experimenter is willing to wait for a solution before restarting the algorithm with different parameters. For example, Koza et al. ran their patent-busting genetic programming experiments for one month each [46].

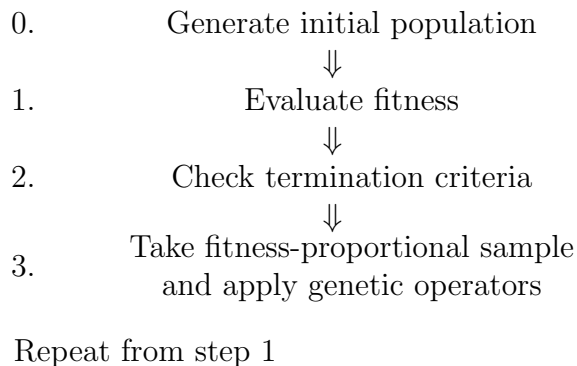


Figure 3.5: Simplified flowchart for the Genetic Algorithm or Genetic Programming

### 3.5 Building blocks and schemata

The fundamental qualitative argument for why genetic algorithms are successful is expressed in the so-called “building blocks” hypothesis [29], [34]. It says that good solutions to a problem (fit individuals) are assembled from

good parts. In particular, good solutions have “good” partial solutions assembled in a “good” way to form a “good” whole. This seems almost tautological at first. Of course the whole thing must be “good” if it is a useful solution. However, it implies a more specific quality of the problem that must be true for genetic algorithms to be more successful than a random search. It implies that good building blocks, while not sufficient for a good solution, are necessary. For the right type of problem, genetic algorithms will keep good building blocks in the population until they can later be assembled into a good whole. The right type of problem is one in which a good building block can add value to an otherwise flawed solution.

Consider a function identification problem. It is desired to find an algebraic expression for an unknown function of one variable. Genetic programming applied to this class of problems is known as “symbolic regression”. All the requirements can be met for the application of genetic algorithms to this problem. Namely, a suitable encoding and fitness measure are possible. The encoding may be as a bit string but another, more natural encoding is shown in section 4.5.3. The fitness function shall be the reciprocal of the sum mean square difference between the solution expression and a sampling of the “true” function in whatever form it is available. There are some sub-expressions, or building blocks, that will bring the solution expression dramatically closer to the true function even if other parts of the solution are incorrect. For the example of an offset and scaled parabola such as  $1 + 2(x + 3)^2$  it is clear that solutions containing a positive quadratic term will be more fit than solutions containing only linear expressions given a sufficiently large range of samples in the fitness test. All linear expressions are exponentially worse fitting near at least one extremity.

A more formal complement to the building blocks hypothesis is schema theory [34], [29]. Schemata for bit string genomes are strings from the alphabet  $\{0, 1, *\}$  that describe sets of bit strings. A 0 or a 1 anywhere in a schema indicates that bit string members of the corresponding set *must* have a 0 or a 1 respectively in the given position. The \* character is a wild card, indicating a degree of freedom. Members of the schema set may have either value in that position. The number of bits that are specified exactly (not by wild cards) in a schema is called the “order” of the schema. The number of positions between the outermost exactly specified bits is called the “defining length” of the schema. For example,  $01 * 1*$  is a length 5, order 3 schema with a defining length of 4 that describes the following set of bit strings:

01010  
01011  
01110  
01111

Schema theory states that genetic algorithms truly operate in the space of schemata. It predicts that low-order schemata with a short defining length and above average fitness receive exponentially more trials in future generations. Poli [60] gives an adaptation to tree-structured genetic programming.

### 3.6 Tree structured programs

The key to applying genetic algorithms to the problem of generating a computer program is to choose a genetic encoding such that even randomly generated individuals decode to a valid, if incorrect, computer program. Alternatively, it is possible to use any encoding at all (the one preferred by human programmers perhaps) but assign an arbitrarily low (i.e. zero) fitness score to invalid programs. However, in most languages the distribution of valid computer programs among the set of random character strings is so astronomically low that the algorithm will spend virtually all available time constructing, checking, and discarding invalid programs.

The canonical genetic programming system by Koza [43] was written in LISP. The LISP programming language [50] has an extremely minimal syntax and a long history of use in artificial intelligence research. Program source code and data structures are both represented in a fully bracketed prefix notation called s-expressions. Nesting bracket expressions gives a hierarchical or tree structure to both code and data. As an early step in compilation or interpretation, programs written in most languages are fit into a tree structure according to the way they are expressed in the language (called the “concrete parse tree” of the program). The LISP s-expression syntax is a very direct representation of this structure (i.e. parsing LISP programs is near trivial).

This immediate representation and the ability of LISP code and data to be interchanged and manipulated freely by other LISP code makes it relatively

straight forward to write LISP programs that create and manipulate other LISP programs. Since LISP code and data are both stored as tree structures (s-expressions), any valid data structure is also valid code. Data structures can be created and manipulated according to the genetic operators using standard LISP constructs and, when the time comes, they can be evaluated as program code to test their fitness.

Genetic programming systems are now often written in C [63], [46] for performance reasons. For this thesis Python [83] is used but the essence of the process is still the creation, manipulation, and evaluation of programs in the form of their concrete parse trees. Often the genetic programs themselves are created in a special purpose language that is a reduced version of, or completely distinct from, the language in which the genetic programming system is written. For example, in this thesis the genetic programs are created in a statically typed functional language that is interpreted by the genetic programming system, itself written in Python. The language is statically typed in that every element of a program has a particular type that determines where and how it can be used, and the type of an element cannot be changed. It is functional in the same sense as LISP or Haskell whereas Python, in which the genetic programming system is written, is an imperative language, like C or Java.

They are “imperative” in the sense that they consist of a sequence of commands, which are executed strictly one after the other. [...] A functional program is a single expression, which is executed by evaluating the expression. <sup>2</sup>

The canonical form of genetic programming as given in [43] shares the basic procedure of the generational genetic algorithm. New generations are created from old by fitness proportional selection and the crossover, mutation, and replication operations. Genetic programming is precisely the genetic algorithm described above with the following important differences:

1. Whereas a bit string genetic algorithm encodes candidate solutions to a problem as fixed length bit strings, genetic programming individuals are programs (typically encoded as their parse trees) that solve the problem when executed.

---

<sup>2</sup>[http://haskell.org/haskellwiki/Introduction#What\\_is\\_functional\\_programming.3F](http://haskell.org/haskellwiki/Introduction#What_is_functional_programming.3F)

2. Whereas bit string genetic algorithm chromosomes have no syntax (only bitwise semantics) the genetic programming operators preserve program syntax. Thus only the reduced space of syntactically correct programs that is being searched, not the space of all possible sequences of tokens from the programming language.
3. Whereas bit string genetic algorithm solutions have a fixed-length encoding which entails predetermining, to some degree, the size and shape of the solution before searching<sup>3</sup>, tree structured genetic programming solutions have variable size and shape. Thus, in a system identification application, genetic programming can perform structural and parametric optimization simultaneously.
4. Whereas bit string genetic algorithm individuals typically decode to a solution directly, genetic programming individuals decode to an executable program. Evaluation of genetic programming solutions involves running these programs and testing their output against some metric.
5. There are some operators on tree structured programs that have no simple analogy in nature and are either not available or are not commonly used in bit string genetic algorithms.

Preparation for a run of genetic programming requires the following 5 decisions from the experimenter:

1. Define the terminal node set. This is the set of all variables, constants, and zero argument functions in the target language. It is the pool from which leaf nodes (those having no children) may be selected in the construction and manipulation of program trees.
2. Define the function node set. This is the set of operators in the target language. Members are characterized by the number of arguments they take (their arity). It is the pool from which internal (non-leaf) nodes may be selected in the construction and manipulation of program trees. Canonical genetic programming requires that the function set satisfy closure, meaning that every member of the function set must accept as

---

<sup>3</sup>This requirement for *a priori* knowledge of upper limits on the size and shape of the solution is common to many other machine learning techniques. For example, with neural network applications the number of layers and nodes per layer must be decided by the experimenter.

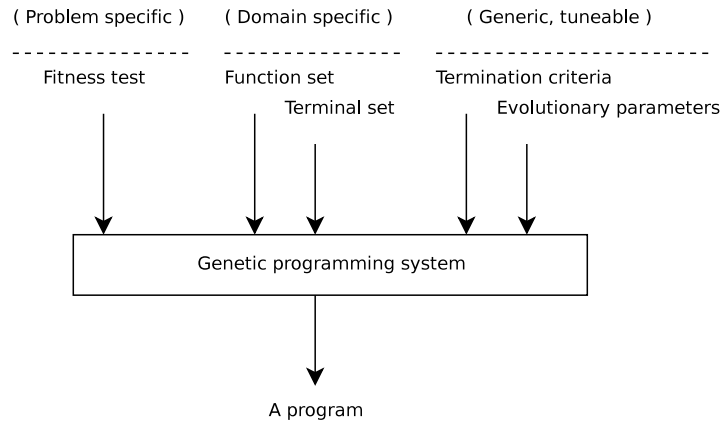


Figure 3.6: Genetic programming produces a program. Some inputs to the framework are generic, others are specific to the problem domain but only one, the fitness test, is specific to any particular problem.

input the output of any other member. This allows crossover points to be chosen anywhere in the program syntax tree without constraint.

3. Define the fitness function. Typically, the fitness function executes an individual program with a fixed suite of test inputs and calculates an error metric based on the program output.
4. Choose the evolutionary control parameters  $M$ ,  $G$ ,  $P_R$ ,  $P_C$ ,  $P_M$ .
  - $M$  population size (number of individuals)
  - $G$  maximum number of generations to run
  - $P_R$  probability of selecting the replication operator
  - $P_C$  probability of selecting the crossover operator
  - $P_M$  probability of selecting the mutation operator
5. Choose termination criteria. Typically a run of genetic programming is terminated when a sufficiently fit individual is found or a fixed number of generations have elapsed.

The first two decisions (definition of the function and terminal set) define the genetic encoding. These need to be done once per computer language in which programs are desired. Specialized languages can be defined to better

exploit some prior knowledge of the problem domain, but in general this is an infrequent task. The fitness function is particular to the problem being solved. It embodies the very definition of what the experimenter considers desirable in the sought program and so changes with each new problem. The evolutionary control parameters are identical to those of the canonical genetic algorithm and are completely generic. They can be tuned to optimize the search, and different values may excel for different programming languages, encoding, or problems. However, genetic programming is so very computationally expensive that they are general let alone once adequate performance is achieved. The cost of repeated genetic programming runs usually prohibits extensive iterative adjustment of these parameters except on simple problems for the study of genetic programming itself.

Specialized genetic programming techniques such as strongly typed genetic programming will require additional decisions by the experimenter.

### **3.6.1 The crossover operator for tree structures**

The canonical form of genetic programming operates on program parse trees instead of bit strings and the genetic operations of crossover and mutation differ necessarily from those used in the canonical bit string genetic algorithm. Selection and replication, which operate on whole individuals, are identical.

The crossover operator discussed in section 3.3.1 swaps sub-strings between two parent individuals to create a pair of new offspring individuals. In genetic programming with tree structured programs, sub-trees are swapped. To do this, a node is selected at random on each parent then the sub-trees rooted at these nodes are swapped. The process is illustrated in figure 3.7. Closure of the function set ensures that both child programs are syntactically correct. They may, however, have different size and shape from the parent programs, which differs significantly from the fixed-length bit-strings discussed previously.

### **3.6.2 The mutation operator for tree structures**

Two variations on the mutation operator are possible using tree structured individuals. The first, called “leaf node mutation” selects at random one leaf node (a node at the periphery of the tree having no children) and replaces it with a new node randomly selected from the terminal set. Like the single



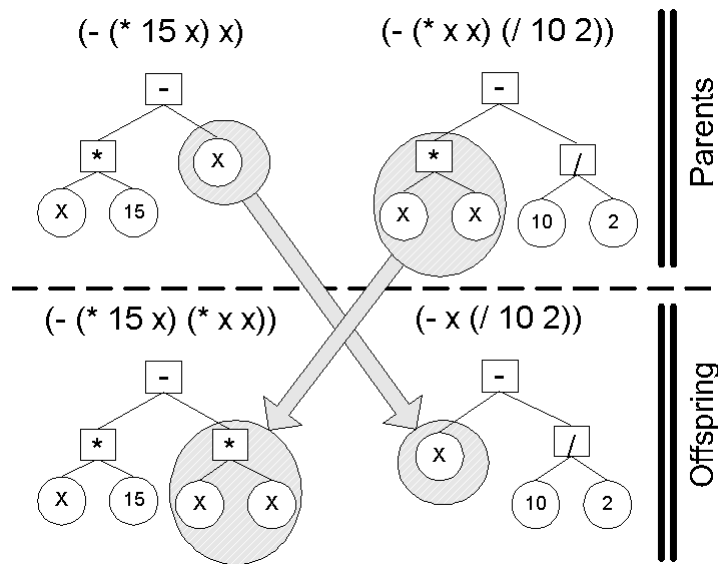


Figure 3.7: The crossover operation for tree structured genotypes

bit mutation discussed in section 3.3.2, this is the smallest possible random change within the overall structure of the individual.

The second mutation operator is called “sub-tree mutation”. In this variation, a node is selected at random from anywhere within the tree. The sub-tree rooted at this node is discarded and a new sub-tree of random size and shape is generated in its place. This may create much more dramatic changes in the program. In the extreme, if the root node of the whole tree is selected then the entire program is replaced by a new randomly generated one.

### 3.6.3 Other operators on tree structures

Other more exotic operations such as “hoist” and “automatically defined functions” have been suggested [43]. Not all have counterparts in bit string genetic algorithms or in nature. The hoist operator creates a new individual asexually from a randomly selected sub-tree of the parent—a partial replication. The child will be smaller and differently shaped than the parent tree.

The automatically defined function operator adds a new member to the function set by encapsulating part of the parent tree. An internal (non-leaf)

node is selected at random to be the root of the new function. The sub-tree rooted at this node is traversed to some depth. Un-traversed sub-trees below this depth are considered arguments to the function. A new node is added to the function set having the required number of arguments (that is, requiring that number of children). This new function stores a copy of the traversed portion. Within the parent individual, the sub-tree at the root of the new function is replaced with an instance of the automatically defined function with the untraversed sub-trees attached below. In subsequent generations the automatically defined function is available for use within the descendants of this program just like any other member of the function set. Immediately prior to fitness evaluation of any program containing automatically defined functions, those functions are replaced by their stored originals. Thus, an individual created by the automatically defined function operator is exactly as fit as its parent but is has a different, and simpler, structure using a new function that was not available to the parent. Automatically defined functions allow a portion of a program from anywhere within its tree to be encapsulated as a unit unto itself and thus protected from the destructive effects of crossover and mutation.

### 3.7 Symbolic regression

Symbolic regression is an application of genetic programming where the “programming” language in use is restricted to just algebraic expressions. Symbolic regression solves the problem of generating a symbolic expression that is a good fit to some sampled data. The task performed by symbolic regression is sometimes also known as “function identification” or “equation discovery”. A variation that generates dimensions (units) along with the expressions is called “empirical discovery” [41]. Traditional regression methods such as linear, quadratic or polynomial regression search only over for the coefficients of a preformed expression. They cannot automatically discover the form of expression which best suits the data.

Symbolic regression and its variants are widely reported in the literature. In its simplest form it is easily understood and example problems are easily generated. Symbolic regression is often used in benchmarking implementations of the genetic programming technique. Symbolic regression on a single-input, single-output (SISO) static, nonlinear function is presented in section 5.2. Symbolic regression on multiple-input, multiple-output (MIMO) static

functions can be treated in the same way using vector quantities in place of scalars. Symbolic regression on SISO or MIMO *dynamic* systems, however requires a different approach. Modelling of dynamic systems requires some sort of memory operation, or differential equations. Section 4.5.3 presents a design for using symbolic regression to discover sets of differential and algebraic equations in state space form.

### 3.8 Closure or strong typing

Genetic programming systems must be strongly typed or mono-typed with closure. The type system of a functional programming language is said to have closure if the output of every function is suitable input for any function in the language. The canonical form presented by Koza is mono-typed with closure. To enforce closure in symbolic regression, care must be taken with some very common operators that one is almost certain to want to include in the function set.

For example, division must be protected from division by zero. Koza handles this by defining a “protected division” operator “%” which returns the constant value 0.0 whenever its second operand is zero.

Measures such as this ensure that every candidate program will execute without error and produce a result within the domain of interest. The outcomes, then, are always valid programs that produce interpretable arithmetic expressions. They may, however, exploit the exceptions made to ensure closure in surprising ways. A candidate program might exploit the fact that  $x/0 = 0$  for any  $x$  in a particular genetic programming system to produce the constant 1.0 through a sub-expression of the form  $\cos(x/0)$  where  $x$  is a sub-expression tree of any size and content. In this case the result is interpretable by replacing the division node with a 0.0 constant value node. A more insidious example arises if one were to take the seemingly more reasonable approach of evaluating  $x/0$  as equal to some large constant (perhaps with the same sign as  $x$ ). That is,  $x/0$  has a value as close to infinity as is representable within the programming system. This has the disadvantage that  $\sin(x/0)$  and  $\cos(x/0)$  evaluate to unpredictable values on  $[-1,1]$  which vary between computer platforms based on the largest representable value and the implementation of the sin and cos functions.

A more robust solution is to assign a very low fitness value to any candidate program which employs division by zero. Using the standard Python

Listing 3.1: Protecting the fitness test code against division by zero

```
def fitTest(candidate):
    error = 0.0
    try:
        for x in testPoints:
            error += abs(candidate(x) - target(x))
    except ZeroDivisionError:
        return 0.0
    # NaN (undefined) compares equal to any number.
    if error==0 and error==1:
        return 0.0
    # High error —> low fitness
    # Zero error —> perfect fitness (1.0)
    return 1.0/(1.0+error)
```

division operator will throw a runtime exception when the second operand is zero. That exception is trapped and the individual in which it occurred is awarded a fitness evaluation of zero. The following source code excerpt shows how division by zero and occurrences of the contagious NaN value<sup>4</sup> result in a zero fitness evaluation. Comments in Python begin with a # character and continue to the end of the line.

Another very common yet potentially problematic arithmetic operation

---

<sup>4</sup>The standard Python math library relies on the underlying platform's implementation. For example in cPython (used in this project), the Python standard library module "math" is a fairly transparent wrapper on the C standard library header "math.h". All operating systems used in this project (Microsoft Windows 2000, GNU/Linux, and SGI IRIX) at least partially implement IEEE standard 754 on binary floating point arithmetic. Of particular interest, IEEE 754 defines two special floating point values called "INF" (-INF is also defined) and "NaN". INF represent any value larger than can be held in the floating point data type. INF arises through such operations as the addition or multiplication of two very large numbers. NaN is the IEEE 754 code for "not a number". NaN arises from operations which have undefined value in common arithmetic. Examples include cos(INF) or sin(INF) and INF/INF. These special values, INF and NaN, are contagious in that operations involving one of these values and any ordinary number evaluates to the special value. For example  $INF - 123.456 = INF$  and  $NaN * 2.0 = NaN$ . Of these two special values, NaN is the most "virulent" since in addition to the above, operations involving NaN and INF evaluate to NaN. See <http://grouper.ieee.org/groups/754/> for further reading on the standard.

is fractional powers. If the square root of a negative value is to be allowed, then a mono-typed GP system with closure must use complex numbers as its one type. If one is not interested in evolving complex valued expressions then this also must be trapped. Again, the more robust option is to assign prohibitively low fitness to programs that employ complex values.

GP trees that build graphs from a kernel are strongly typed since there may be several types of modifiable sites on the graph which can only be modified in certain (possibly non-overlapping) ways. For a simple example, a node and an edge may both be modifiable in the kernel yet there are not many operations that a GP function node could sensibly perform in exactly the same way on either a node or an edge.

### **3.9 Genetic programming as indirect search**

Genetic programming refers to the use of genetic algorithms to produce computer programs. An indirect use of genetic programming uses the genetic algorithm to find programs that, when run, produce a solution from the search space. These programs are judged not on their own structure but on the quality of their output. This generalizes the applicability of the technique from simple bit-strings or tree structures to any possible output of a computer program in the given programming language. There is a computational cost to all of this indirection however, and a risk that it contorts the fitness landscape in a way that will slow down progress of the genetic algorithm.

A genetic programming system can be arranged that gives each genetic program as input a representation of some prototypical solution (called a kernel) that incorporates any prior knowledge about the problem. If the problem is system identification, then the kernel may be a rough system model arrived at by some other identification technique. If the problem is symbolic regression, then the kernel may be a low order approximation of the target function, for example. In this arrangement, a program is sought which modifies the kernel and returns a better solution. This makes genetic programming useful in “grey-box” system identification problems, for example, where a preliminary model is already known.

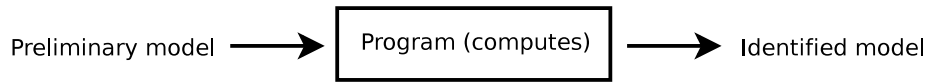


Figure 3.8: Genetic programming as indirect search for use in “greybox” system identification

## 3.10 Search tuning

### 3.10.1 Controlling program size

Although it is often listed as an advantage of genetic programming that there is no formal requirement for *a priori* knowledge of the size or form of an optimal or even correct solution, there are practical reasons for setting an upper bound on the size of evolved programs.

Namely, the algorithm must be implemented with finite computational resources. If individuals are allowed to grow without bound then one or more exceptionally large individuals can quickly consume so much memory and processor time as to virtually stall the search.

Common resource usage limitation options include a fixed population size and maximum number of generations, which are near ubiquitous in genetic programming. As to program size, a fixed ceiling may be set either as a constraint on crossover or (as implemented in this project) by assigning zero fitness during program evaluation for individuals that exceed the ceiling.

Another program size limiting option is to incorporate program size as a negative factor in fitness evaluation, creating a sort of evolutionary “parsimony pressure”. Here there is a difficulty in deciding the “pressure schedule” that maps between program size and the penalty factor. In particular a schedule that is monotonic and steep, tending to penalize large programs and reward small programs to an extent that overwhelms the test-suite based performance factor will limit the size and diversity of subprograms available for inclusion in a solution through crossover. This extreme case would obviously have a detrimental effect on the performance of the genetic programming algorithm. Even if the extreme is not reached, the very act of designing a “parsimony pressure schedule” requires us to estimate the size of a correct solution to our problem. Specifying how big a program must be in order to solve a given problem will usually be more difficult than picking some generous value which is obviously adequate.

A “parsimony pressure” measure may also be implemented by limiting the computational resources (e.g. memory or processing time) available to each program during evaluation. Programs which are not able to complete the fitness evaluation test suite within their assigned resource limits are culled from the population. While more complex to implement than, for example, a simple ceiling on program tree size, this approach is stronger. The strength of this approach arises when the programs being evolved may incorporate iteration or recursion for which the bounds are a parameter of the search or when the programs being evolved are allowed to allocate resources for their own use. None of these aspects apply to the programs of interest in this work however, and so the simpler approach was taken.

It is also interesting to note that the approach of limiting computation resources in general is perhaps more naturalistic. Biological organisms which have very extensive material needs will, on occasion, find those needs unmet and consequently perish.

### **3.10.2 Maintaining population diversity**

It is a fact of performing a local search on complex error surfaces that local extrema are sometimes encountered. Genetic programming can become “stuck” for considerable time on a non-global fitness maximum if the population comes to consist predominately of individuals that are structurally identical. That is, if one solution occurs which is very much more fit than all others in the current population, then reproduction with reselection will cause the next generation to consist primarily of copies of this one individual. Although, subtree swapping crossover can produce offspring of differing size and shape from both parents (for example by swapping a large portion of one program for a small portion of another), if both parents are substantially or exactly structurally equivalent, then the range of possible offspring is greatly reduced. In the extreme, a population consisting entirely of individuals having exactly the same structure, a solution containing function nodes not present in that structure is not reachable except through mutation, which is typically employed sparingly.

There are many techniques available for population diversity maintenance such as tagging or distance measures based limits on the total number of individuals per generation sharing any particular structure, demes with migration, and adaptive crossover or mutation rates (e.g. make mutation rate proportional to the inverse of some population diversity measure).

No active population diversity maintenance techniques were employed but a measure of population diversity was included in the measured genetic programming run aspects. This measure employed tagging, a sort of structural hashing in which each individual is assigned a “tag” (which could be a simple integer) such that structurally equivalent individuals share a tag value while tag values differ for structurally different individuals. A test for structural equivalence must be designed for the particular function and terminal sets in use. As an example, in a symbolic regression system, two program trees might be considered equivalent if their root nodes are equivalent where the equivalence of two nodes is defined recursively as follows: two nodes are equivalent if they share the same type, the same number of children, and corresponding pairs of their children are all equivalent. Thus two constant value node would be considered structurally equivalent regardless of any difference in their values (a parametric, not structural difference) since they have the same type (constant value) and the same number of children (zero).

It may seem natural to consider any subtree having only constant value leaf nodes to be structurally equivalent to a single constant value node. A side effect of this would be to bias the search towards exploring towards exploring structural variations and away from parametric optimization.

Tagging is less computationally expensive and conceptually simpler than numerically valued distance measures. A conceptually simple recursive algorithm for evaluating structural equivalence is described above. This evaluation can be made tail-recursive and short circuiting. Short circuit evaluation is possible in this case because the algorithm will stop recursion and return false immediately upon encountering any two corresponding nodes which are not equivalent. The implementation in Python developed for this project achieves this automatically since all of the Python built-in logical operators are short circuiting. A LISP or Scheme implementation would benefit since modern optimizing LISP compilers perform very well with tail-recursion.



# Chapter 4

## Implementation

In the course of producing this thesis, software was written for bond graph modelling and simulation, symbolic algebra, and strongly typed genetic programming. This software was written in the Python [83] programming language with a numerical solver provided by the additional SciPy [37] libraries. Visualization tools were written that use Graphviz [25], Pylab [36] and PyGame [72] to show bond graphs, genetic trees, algebraic structures, numerical results, and animations of multi-body mechanics simulations.

### 4.1 Program structure

The bond graph modelling and simulation and related symbolic algebra code is divided into 4 modules.

**Bondgraph.py** The *Bondgraph* module defines a set of objects representing basic and complex bond graph elements, bonds, and graphs. Graphs can be assembled programmatically or prescriptively (from text files). Automatic causality assignment and equation extraction are implemented here as are some bond graph reduction routines.

**Components.py** The *Components* module contains a collection of compound bond graph elements.

**Algebra.py** The *Algebra* module provides facilities for constructing and manipulating symbolic expressions and equations. It produces textual representations of these in Graphviz “DOT” markup or L<sup>A</sup>T<sub>E</sub>X. It implements algorithms for common algebraic reductions and recipes for

specialized actions such as finding and eliminating algebraic loops from sets of equations where possible and identifying where it is not possible.

**Calculus.py** The *Calculus* module provides extensions to the **Algebra** module for constructing and manipulating differential equations. In particular, for reducing coupled sets of algebraic and first order differential equations to a set of differential equations involving a minimal set of state variables, and a set of algebraic equations defining all the remaining variables in terms of that minimal set.

The genetic programming code is divided into 4 modules as well.

**Genetics.py** The *Genetics* module implements an abstract genetic programming type system from which useful specializations can be derived. It implements the genetic operators for tree structured representations.

**Evolution.py** The *Evolution* module executes the generational genetic algorithm in a generic way, independent of the particular genetic representation in use.

**SymbolicRegression.py** The *SymbolicRegression* module implements a specialized genetic programming type system for symbolic regression on dynamic systems on top of the *Genetics* module.

**BondgraphEvolution.py** The *BondgraphEvolution* module implements a specialized genetic programming type system for evolving bond graphs on top of the *Genetics* module. Uses a subset of the symbolic regression module to find constant valued expressions for bond graph element parameters.

A number of other modules were written. Most of these automate particular experiments or implement small tools for visualization of bond graphs and genetic trees, generating test signals and visualizing numerical results. A framework for distributing genetic programming across a local or wide area network in the island model was also begun but is yet incompletd.

### 4.1.1 Formats and representations

Simulation results and test signals used as input to simulations are stored in comma separated value (CSV) text files for portability. For particularly long

or detailed (small time-step) simulations, there is a noticeable delay while the plotting and animation visualization tools parse and load the results. A more compact non-text format might run quicker but the delay is acceptable in exchange for the ability to inspect the results with a text editor and the ease of importing CSV data into off-the-shelf tools such as MATLAB, Octave, and even spreadsheet programs.

There are several graph definition markup languages in use already, including GraphML (an XML schema) and DOT (used by Graphviz). An expedient custom markup format for storing bond graphs was defined as part of this work. The `Bondgraph.py` module is able to load models from files written in this format and write them back out, possibly after some manipulation. The distributed computing modules use this format to send bond graph models across the network. It is not the intention that this format become widely used but an example is included as listing 4.1 for completeness.

Any line beginning with a `#` is a comment intended for human eyes and ignored by the software. Nodes and bonds are defined within `begin` and `end` lines. Nodes referred to within the definition of a bond must themselves have been defined earlier in the file. The direction of positive power flow along bonds is from the tail to the head (i.e. the half arrow would be drawn at the head). Bond graphs defined in this way are acausal, since `Bondgraph.py` is able to assign causality automatically when it is needed.

This format can be criticized as needlessly verbose. For comparison, Python code to create the same model then extract and print the corresponding state space equations is included as listing 4.2.

Listing 4.1: A simple bond graph model in `.bg` format

```
title single mass/spring/damper chain.

## nodes ##

# Spring, damper and mass.
begin node
    id spring
    type C
    parameter 1.96
end node

begin node
```

```

    id damper
    type R
    parameter 0.02
end node

begin node
    id mass
    type I
    parameter 0.002
end node

# Common flow point joining s/d/m
begin node
    id point1
    type cfj
end node

# Input force , acting on spring and damper
begin node
    id force
    type Se
    parameter Y
end node

## bonds ##

# Spring and damper joined to mass
begin bond
    tail point1
    head spring
end bond

begin bond
    tail point1
    head damper
end bond

```

```

begin bond
    tail point1
    head mass
end bond

begin bond
    tail force
    head point1
end bond

# EOF

```

Listing 4.2: A simple bond graph model constructed in Python code

```

from Bondgraph import *

g = Bondgraph('single_mass/spring/damper_chain')
c = g.addNode(Capacitor(1.96, 'spring'))
r = g.addNode(Resistor(0.02, 'damper'))
i = g.addNode(Inertia(0.002, 'mass'))
j = g.addNode(CommonFlowJunction())
s = g.addNode(EffortSource('E'))

g.bondNodes(j, c)
g.bondNodes(j, r)
g.bondNodes(j, i)
g.bondNodes(s, j)

g.assignCausality()
g.numberElements()

print StateSpace(g.equations())

```

## 4.2 External tools

### 4.2.1 The Python programming language

Python is a highly dynamic, interpreted, object-oriented programming language originally written by Guido van Rossum. It has been distributed under a series of open source licenses [62] and has thus gained contributions from hundreds of individuals world-wide. Similar to LISP, the dynamic and introspective features of Python facilitate the creation, manipulation and evaluation of blocks of code in an automated way at runtime, all of which are required steps in genetic programming.

In addition to personal preference and familiarity, Python was chosen because it is more legible to a general audience than LISP and faster and less error prone than writing programs in C. The historical choice of syntax and keywords for Python has been strongly influenced by the “Computer Programming for Everyone” philosophy [82] which strives to show that powerful programming environments enjoyed by experts need not be inaccessible to others. The language design choices made to date have been successful in this respect to the extent that executable Python source code from real applications has occasionally been mistaken for pseudo code. Take for example the following two functions which implement the universal and existential logical qualifiers, respectively:

Listing 4.3: Two simple functions in Python

```
def forAll(sequence, condition):
    for item in sequence:
        if not condition(item):
            return False
    return True

def thereExists(sequence, condition):
    for item in sequence:
        if condition(item):
            return True
    return False
```

These are succinct, easily understood by English readers familiar with procedural programming and, due to the name-based polymorphism inherent in Python’s object system [49], these operate on any iterable object,

“sequence” (which may in fact be a list, tuple, dictionary etc. or even a custom object-type implementing the iterator interface) and any callable object “condition”.

Since this research addresses automated dynamic system identification topics which are primarily of interest to engineers and not computer scientists, the ease and accuracy with which an audience of non-specialists in the implementation language can understand the software is important.

In addition to its legibility, Python code is portable without change across several important platforms. Development was carried out at times on each of Mac OS X, GNU/Linux, and Microsoft Windows with no change to the code base. Later, dozens of genetic programming runs were performed with the same code on a 4 processor SGI machine under the IRIX operating system. Python compiles easily for most Unix variants making Python code portable to some very powerful machines.

## 4.2.2 The SciPy numerical libraries

SciPy [37] is a collection of libraries and extensions to the Python programming language that are likely to of use to scientists and engineers. In this work, SciPy is used for fast matrix operations and to access a numerical solver. Both capabilities are made possible by code from netlib [1] included in SciPy. The solver is LSODA, an adaptive solver that automatically switches between stiff and non-stiff routines. It is part of ODEPACK [32], which SciPy exposes as `scipy.integrate.odeint`. `StateSpace` objects from the `Calculus.py` module conform to the interface provided by `scipy.integrate.odeint`.

## 4.2.3 Graph layout and visualization with Graphviz

Graphviz [25] is an open source suit of graph layout and visualization software. It was used in this work to visualize many automatically generated graph and tree structures including tree-structured genetic programs, bond graphs, and tree-structured equation representations. The importance of an automated graph layout tool is keenly felt when handling large, automatically generated graphs. If Graphviz or Springgraph [21] (another open source graph layout suite, it reads the Graphviz DOT file format but supports only one layout algorithm) were not available, an automated layout tool would need to have been written. Instead, `Bondgraph.py` is able to produce a

graph definition in the DOT format used by Graphviz. For example, listing 4.4 is the DOT representation produced by `Bondgraph.py` from the graph defined in listing 4.1. From this, Graphviz produces the hierarchical layout (ordered by direction of positive power flow) shown in figure 5.2.

Graphviz DOT files are also produced for genetic trees, nested algebraic expressions, and algebraic dependency graphs.

Listing 4.4: Graphviz DOT markup for the model defined in listing 4.4

```

digraph "single mass/spring/damper chain."
{
  rankdir=LR;
  node [shape=plaintext];
  C1 [label="C: spring"];
  R2 [label="R: damper"];
  I3 [label="I: mass"];
  11 [label="1"];
  Se4 [label="Se: force"];
  edge [fontsize=10];
  11 -> C1 [arrowhead=half, arrowtail=tee, label="1"];
  11 -> R2 [arrowhead=half, arrowtail=tee, label="2"];
  11 -> I3 [arrowhead=teehalf, label="3"];
  Se4 -> 11 [arrowhead=teehalf, label="4"];
}

```

## 4.3 A bond graph modelling library

### 4.3.1 Basic data structures

Formally, a graph is a set of nodes and a set of edges between the nodes. `Bondgraph.py` implements a bond graph object that maintains a list of elements and another of the bonds between them. Each element has list of bonds and counts of the maximum number of causal-in and causal-out bonds allowed. Each bond has head and a tail (references to elements within the bond graph) and a causality (initially nil but assigned equal to the head or the tail during sequential causality assignment). Activated bonds are a subclass of bonds. Activated bonds are never assigned a causality. Signal blocks are elements that only accept activated bonds at their ports.



### 4.3.2 Adding elements and bonds, traversing a graph

The graph class has an `addNode` and a `removeNode` method, a `bondNodes` and a `removeBond` method. Nodes are created independently then added to a graph. The `addNode` graph method checks first that the node is not already part of this or any other graph and that it has no attached bonds. Bonds are created only within the context of a particular graph by calling the `bondNodes` graph method. This method takes two elements (which are to become the tail and the head of the bond, respectively), checks that both elements are part of this graph and can accommodate an additional bond, then joins them and returns a reference to the new bond.

Each element maintains a list of attached bonds. Each bond has a reference to its head and tail elements. In this way it is possible to walk the graph starting from any element or bond. For convenience, the bond graph object has methods to iterate over all elements, all bonds, or a filtered subset thereof.

### 4.3.3 Assigning causality

The sequential causality assignment procedure (SCAP) described in Section 2.3.2 is automated by the bond graph code. This procedure is invoked by calling the `assignCausality` method on a bond graph object. An `unassignCausality` method exists as well which strips any assigned causality from all bonds in the graph.

The `assignCausality` method implementation begins by segregating the graph elements into four lists:

1. one-port elements with a required causality (source elements  $S_e$  and  $S_f$ , switched storage element  $CC$ ),
2. one-port elements with a preferred causality (storage elements  $C$  and  $I$ ),
3. one-port elements with arbitrary causality (sink element  $R$ ),
4. others ( $0$ ,  $1$ ,  $GY$ ,  $TF$ , signal blocks).

As a side effect of the way bond graph elements are stored and retrieved, elements appear in these lists in the order that they were added to the graph.

For each element in the source element list, the attached bond is assigned the required causality. Each time a bond is assigned causality the consequences of that assignment, if any, are propagated through the element at the other end of the bond. Causality propagates when:

- a bond attached to a TF or GY element is assigned either causality,
- a bond attached to a 0 element is assigned effort-in (to the 0 junction) causality,
- the last-but-one bond attached to a 0 element is assigned effort-out (from the 0 junction) causality,
- a bond attached to a 1 element is assigned effort-out (from the 1 junction) causality,
- the last-but-one bond attached to a 1 element is assigned effort-in (to the 1 junction) causality.

The rules for causality propagation are implemented in the `extendCausality` method of each element object type so the propagation of causality assignment consequences is a simple matter of looking up the element at the other end of the just-assigned bond and invoking its `extendCausality` method. That element object itself is then responsible for examining the causality assignment of all attached bonds, making any new assignments that may be implied, and invoking `extendCausality` on its neighbours after any new assignments.

If, as a consequence of the assignment of a required causality, another causality requirement is violated then the `extendCausality` method on the second element will raise an exception and abort the original invocation of `assignCausality`. This could occur if, for example, two like source elements are bonded directly together.

After all required causality one-port elements have been handled, the preferred causality list is examined. Any elements whose attached bond has been assigned a causality, preferred or otherwise, as a consequence of earlier required or preferred causality assignments is skipped over. Others are assigned their preferred causality and consequences are propagated.

Finally, the list of arbitrary causality elements is examined. Again, any whose attached bond has been assigned a causality are skipped over. The rest are assigned a fixed causality (effort-in to R elements) and consequences

are propagated. If any non-activated bonds remain without a causality assignment, these are given an arbitrary assignment (effort in the direction of positive power transfer). Once all bonds have been assigned a causality, the list of preferred causality elements is re-examined and any instances of differential causality are noted in a new list attached to the bond graph object.

#### 4.3.4 Extracting equations

All bond graph element objects have an `equations` method that, when invoked, returns a list of constitutive equations for that element. This method fails if the attached bonds have not yet been assigned a causality and a number. Bonds are assigned a causality either manually or automatically as described in the previous section. Bonds are assigned a number in the order they were added to the graph by invoking the `numberBonds` method of the bond graph object. Bond numbers are used to name effort and flow variables of each bond uniquely. Bond graph objects also have an `equations` method. It returns an equation set object containing all the constitutive equations of all the elements in the graph. It does this by first invoking `assignCausality` and `numberBonds` on the graph object itself, then invoking the `equations` method on each element and collecting the results. Bond graph element objects will, in general, return different constitutive equations depending on the causality assignment of the attached bonds. Elements with a required causality such as `Se`, `Sf`, and `CC` will raise an exception and abort the collection of equations if their `equations` method detects other than the required causality since the constitutive equations of these elements are not defined in that case.

Equations, the algebraic expressions which they contain, and the equation sets into which they are collected are all implemented by the symbolic algebra library described in Section 4.4.

#### 4.3.5 Model reductions

There are certain structural manipulations of a bond graph model that do not change its behaviour in simulation. These may be applied to reduce the complexity of a model, just as the rules of algebra are applied to reduce complex expressions to simpler ones. Five such manipulations are automated by the bond graph library and all bond graph objects built by the library have a `reduce` method that applies them in the order given below. It is advisable to

invoke the `reduce` method before extracting equations for simulation since it is simple (computationally inexpensive) to apply them. Although detecting where these reductions are applicable requires an examination of every junction in the graph and all the junction's neighbours, that is outweighed by the algebraic manipulations it obviates. These bond graph reductions produce a model with fewer constitutive equations and often, if there are loops, with fewer algebraic loops. The latter is a particularly large gain since resolving algebraic loops (the implementation of the procedure for which is described in Section 4.4.3) involves extensive manipulations of sets of equations that may be very large. The implemented bond graph reductions are:

1. Remove junctions with less than 3 bonds
2. Merge adjacent like junctions
3. Merge resistors bonded to a common junction
4. Merge capacitors bonded to a common effort junction
5. Merge inertias bonded to a common flow junction

These five model reduction steps are illustrated in figures 4.1 to 4.5.

In the first manipulation, any 0 or 1 junction having only one bond is removed from the model along with the attached bond. Any junction having only two bonds is likewise removed along with both its bonds and a single new bond put in their place. For this and the second manipulation, the equivalence of the model before and after follows directly from the constitutive equations of the 0 and 1 junctions elements.

In the third manipulation, some number of R elements, all bonded to the same junction and removed and a single new R element is put in their place. For elements with the linear constitutive equation given by equation 4.1 there are two cases, depending on the junction type. Handling of non-linear elements is not automated by the library. If the attached junction is a 1 element, the resistance parameter of the new R element is simply the sum of the parameters of the replaced elements. If the attached junction is a 0 element, the resistance parameter of the new R element is  $r_{eq}$  where

$$r_{eq} = \frac{1}{\frac{1}{r_1} + \frac{1}{r_2} + \dots + \frac{1}{r_n}} \quad (4.1)$$

These are the familiar rules for electrical resistances in series and parallel.

The fourth and fifth manipulations handle the case shown in 2.12 and an even simpler case than is shown in 2.13, respectively. The same electrical and mechanical analogies apply. Capacitors in parallel can be replaced by a single capacitor have the sum of the replaced capacitances. Rigidly joined inertias are effectively one mass.

To illustrate the application of the model reduction rules described above, figure 4.6 shows a randomly generated bond graph model that contains: junctions with less than 3 bonds, adjacent like junctions, and multiple resistors bonded to a common junction. Figures 4.7, 4.8, and 4.9 show the application of the first 3 steps in the automated model reduction method. The 4th and 5th are not applicable to this model. Listing 4.5 gives the collected constitutive equations of the original model (figure 4.6) and figure 4.10 shows the algebraic dependencies between them. Listing 4.6 gives the collected constitutive equations of the reduced model (figure 4.9) and figure 4.11 shows the algebraic dependencies between them.

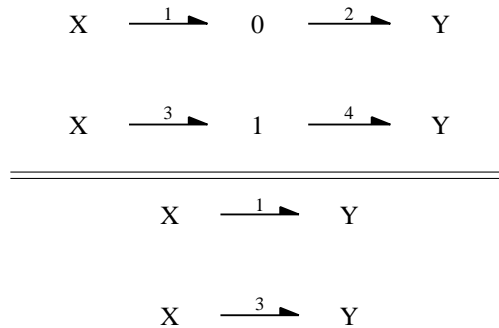


Figure 4.1: Model reductions step 1: remove trivial junctions

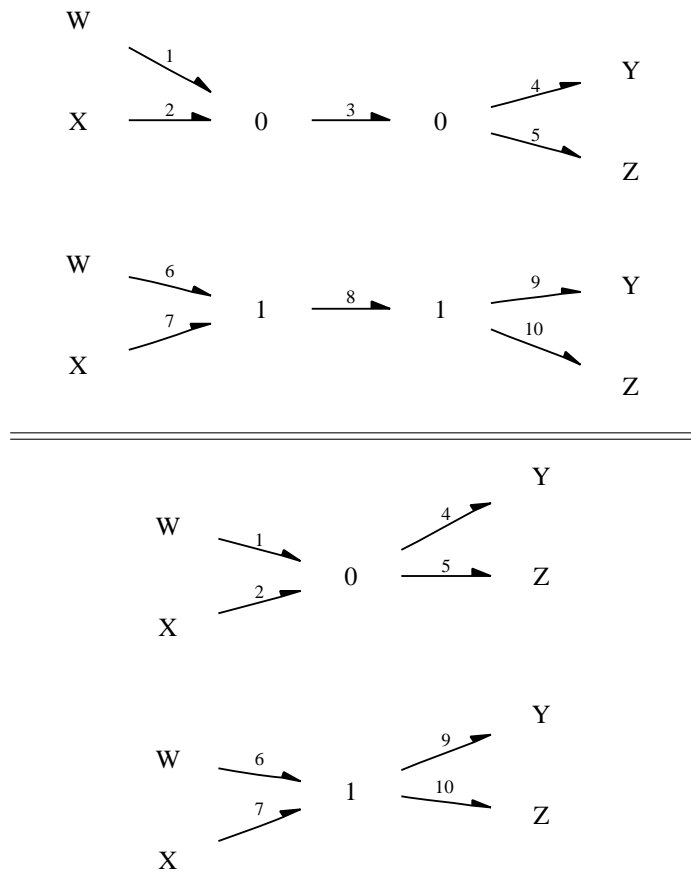


Figure 4.2: Model reductions step 2: merge adjacent like junctions

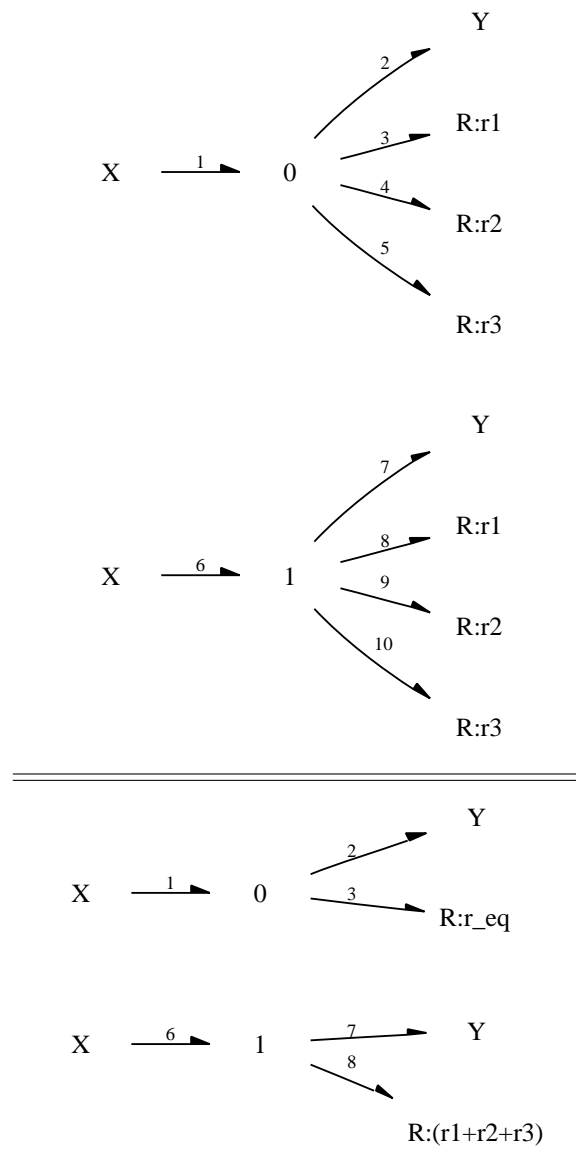


Figure 4.3: Model reductions step 3: merge resistors bonded to a common junctions

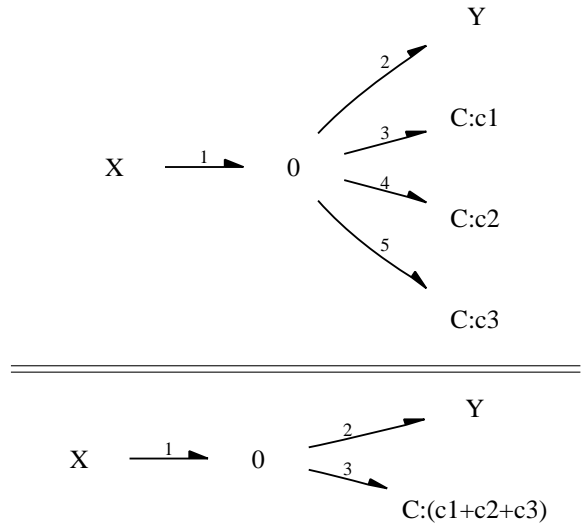


Figure 4.4: Model reductions step 4: merge capacitors bonded to a common effort junctions

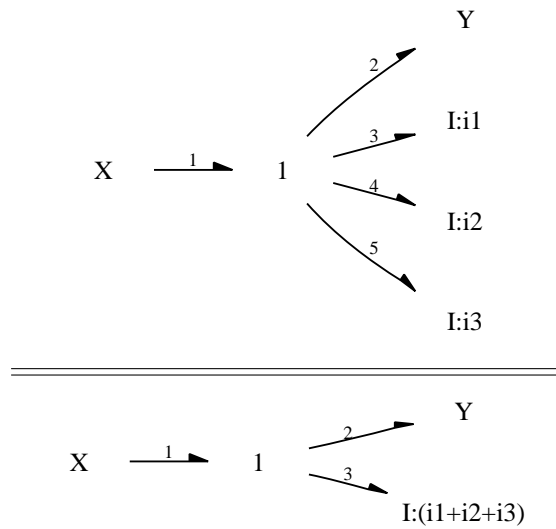


Figure 4.5: Model reductions step 5: merge inertias bonded to a common flow junctions



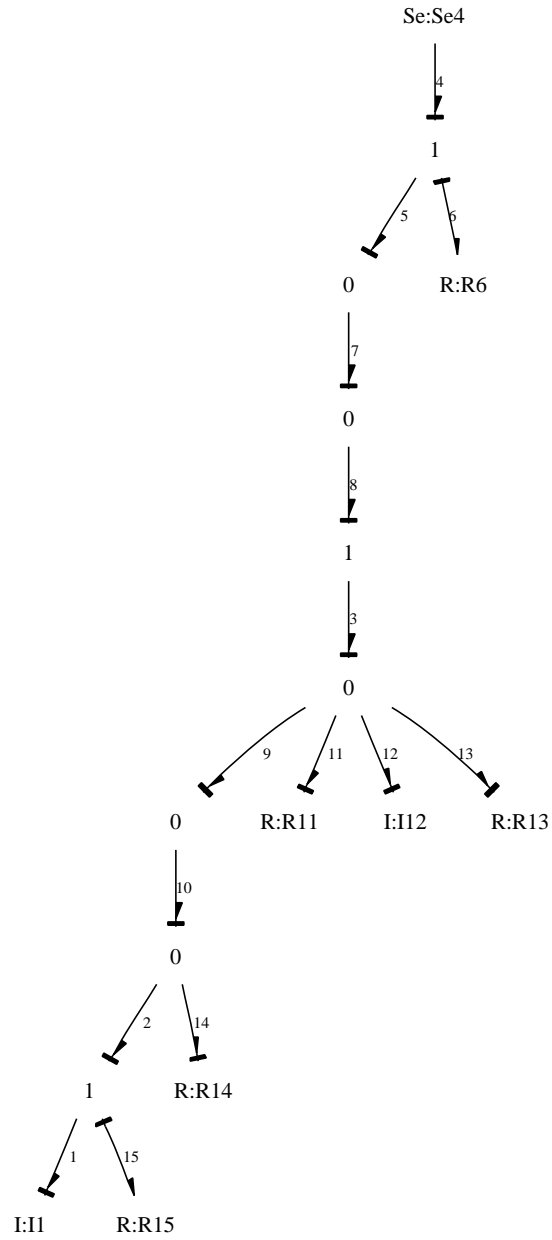


Figure 4.6: A randomly generated model containing: trivial junctions, adjacent like junctions, and multiple resistors bonded to a common junction

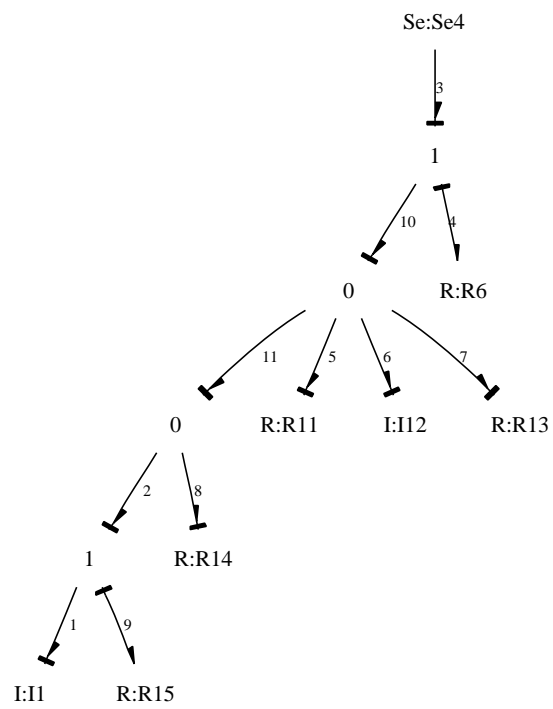


Figure 4.7: The model from figure 4.6, with trivial junctions removed

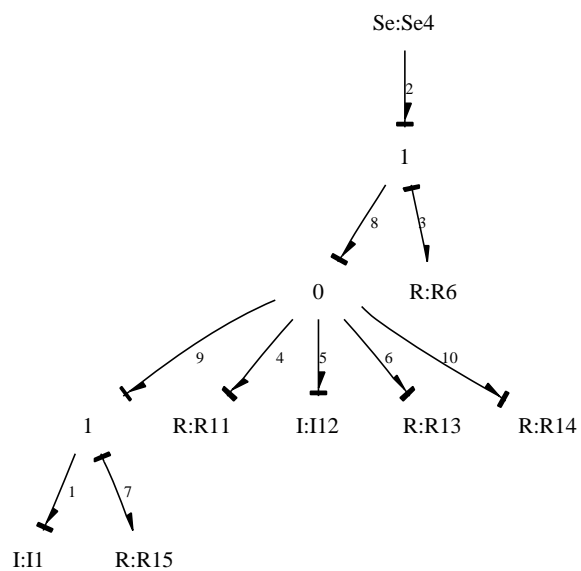


Figure 4.8: The model from figure 4.6, with trivial junctions removed and adjacent like junctions merged

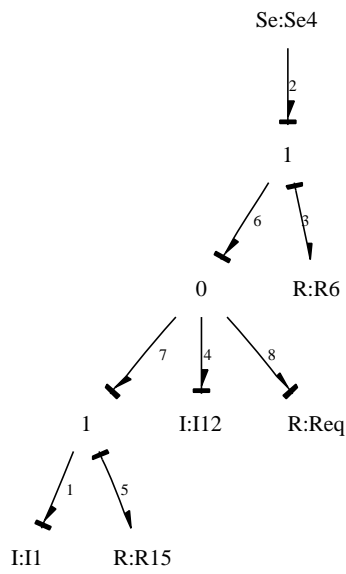


Figure 4.9: The model from figure 4.6, with trivial junctions removed, adjacent like junctions merged and resistors bonded to a common junction merged

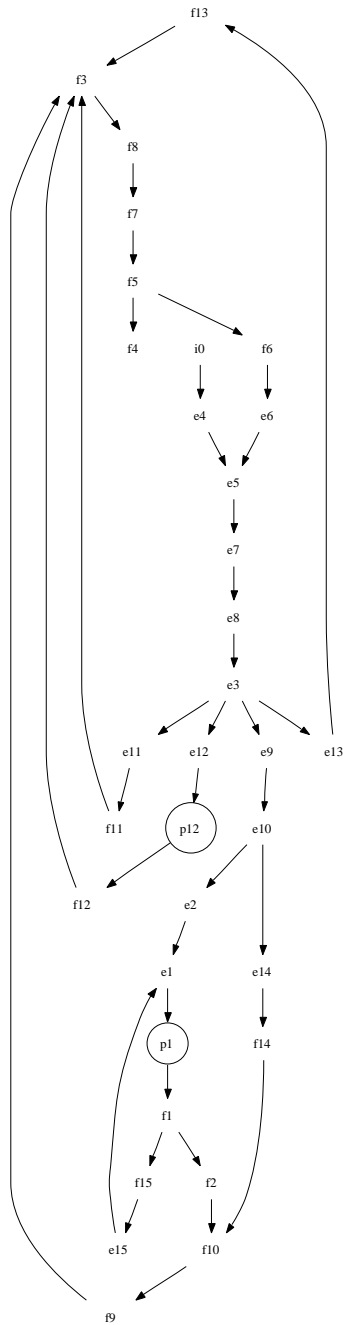


Figure 4.10: Algebraic dependencies within equations extracted from the model of figure 4.6 and listing 4.5

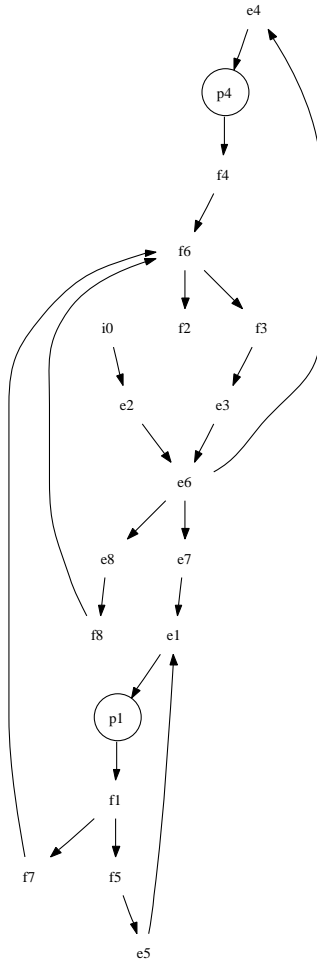


Figure 4.11: Algebraic dependencies within equations extracted from the reduced model of figure 4.9 and listing 4.6

Listing 4.5: Equations from the model in figure 4.6

```
e1 = (e2+(-1.0*e15))
e2 = e10
e3 = e8
e4 = i0
e5 = (e4+(-1.0*e6))
e6 = (f6*32.0)
e7 = e5
e8 = e7
e9 = e3
e10 = e9
e11 = e3
e12 = e3
e13 = e3
e14 = e10
e15 = (f15*45.0)
f1 = (p1/-78.0)
f2 = f1
f3 = (f9+f11+f12+f13)
f4 = f5
f5 = f7
f6 = f5
f7 = f8
f8 = f3
f9 = f10
f10 = (f2+f14)
f11 = (e11/45.0)
f12 = (p12/0.0429619641442)
f13 = (e13/7137.88689084)
f14 = (e14/93.0)
f15 = f1
p1_dot = e1
p12_dot = e12
```

Listing 4.6: Equations from the reduced model in figure 4.9

```
e1 = (e7+(-1.0*e5))
e2 = i0
e3 = (f3*32.0)
e4 = e6
e5 = (f5*45.0)
e6 = (e2+(-1.0*e3))
e7 = e6
e8 = e6
f1 = (p1/-78.0)
f2 = f6
f3 = f6
f4 = (p4/0.0429619641442)
f5 = f1
f6 = (f4+f7+f8)
f7 = f1
f8 = (e8/30.197788376)
p1_dot = e1
p4_dot = e4
```



## 4.4 An object-oriented symbolic algebra library

An object-oriented symbolic algebra library was written as part of this work. It is used by the bond graph library to produce equations in a modular way depending on causality assignment, and by the symbolic regression code for similar reasons. It is capable of several simple and widely applicable algebraic manipulations as well as some more specialized recipes such as reducing equations produced by the bond graph library to state space form even if they contain algebraic loops.

### 4.4.1 Basic data structures

The library implements three basic objects and specializations thereof: an algebraic expression, an equation, and an equation set.

Expression objects can be evaluated given a table in which to look for the value of any variables they contain. Expression objects have methods that list any variables they contain, test if the expression is constant (contains no variables), list any nested sub-expressions (each variable, constant and operator is an expression in itself), replace any one sub-expression with another, and other more specialized algebraic manipulations. Base expression objects are never used themselves, leaf nodes from the object inheritance graph shown in figure 4.12 are used. That figure indicates that `Operator` is a type of `Expression` and `Div` (division) is a type of `Operator`. `Operator` has all the attributes of `Expression` but may modify or add some, and so on for `CommutativeOperator` and all of the practical objects (leaf nodes in the graph). Each type of practical object accepts some number of sub-expressions which may be of the same or any other type. `Constant` and `Variable` objects accept no sub-expressions. `Cos` and `Sin` operators accept one sub-expression each. `Div`, `Greater`, and `Lesser` accept two, ordered sub-expressions. The commutative operators `Add` and `Mul` accept any number of sub-expressions, the order of which is immaterial.

Equation objects are composed of two expression objects (a left-hand side and a right) and methods to list all variables, expressions, and sub-expressions in the equations; replace any expression appearing in the equation with another; as well as several more elaborate algebraic tests and manipulations described in the sections that follow.

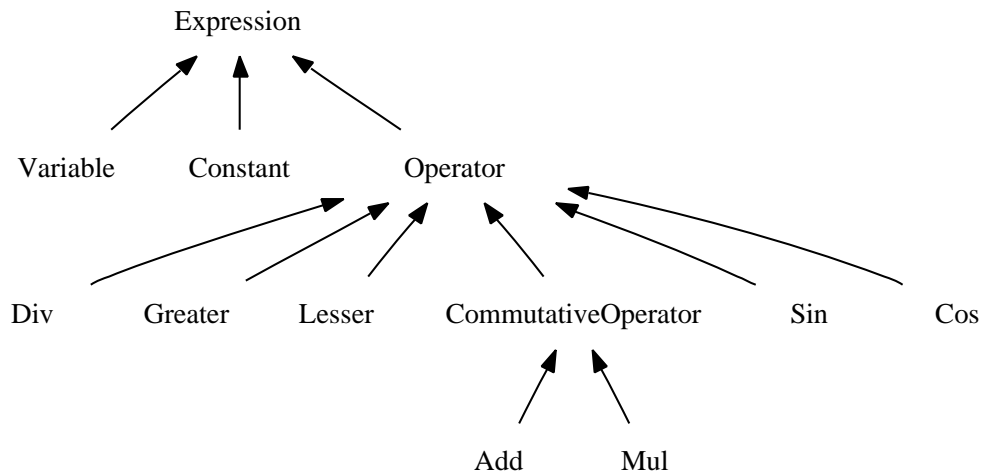


Figure 4.12: Algebraic object inheritance diagram

Equation sets are collections of equations that contain no duplicates. Equation set objects support similar methods to equations and expressions (list all variables, replace one expression with another across all equations in the set, etc.) as well as a number of methods that are specific to collections of equations. Those include recursive backsubstitution of the equations from the set into any other equation, provided the original set is in a certain “declarative” form; detection of this form; detection of algebraic loops; and resolution of algebraic loops. Easily the most complex manipulation that is automated by the library, algebraic loops are discussed in section 4.4.3.

The `Calculus.py` module extends the basic algebra library (`Algebra.py`) with a `Diff` operator and a type of equation set called `StateSpace`. A `StateSpace` object is constructed from an ordinary equation set by a process described in Section 2.5.1. `Diff` takes a single sub-expression and is a placeholder for first order differentiation of that expression with respect to time (a variable called “t” is always assumed to exist in any state space equation set).

## 4.4.2 Basic reductions and manipulations

Expression objects support the following basic manipulations.

- Substitutions (search and replace)
- Get list of variables
- Aggregate constants
- Collapse constant expressions
- Convert constant divisors to multiplication
- Aggregate commutative operators
- Remove single operand commutative operators
- Expand distributive operators

Substitutions and getting a list of variables are self-explanatory. “Aggregate constants” replaces multiple `Constant` objects that appear as direct sub-expressions of a commutative operator with one equivalent `Constant` object. “Collapse constant expressions” detects expressions that contain no variables, evaluates them on the spot and replaces the entire expression with one `Constant` object.

“Convert constant divisors to multiplication” detects `Div` objects that have a `Constant` object as their second sub-expression (the divisor) and replaces the `Div` with a `Mul` and the `Constant` with a new `Constant` whose value is the reciprocal of the original. “Aggregate commutative operators” detects occurrences of a commutative operator within the direct sub-expressions of a like operator. It joins them into one like operator having all the sub-expressions of both.

`CommutativeOperator` objects accept any number of sub-expressions (the operands). “Remove single operand commutative operators” detects occurrences of a commutative operator with just one operand and replaces them with their one sub-expression. `CommutativeOperator` objects with only one sub-expression are null or “do nothing” operations. A single operand `Add` is assumed to add zero to its operand and a single operand `Mul` is assumed to multiply by one. These may arise after aggregating constants if the operator

had only constant sub-expressions. Removing them allows other manipulations to proceed.

“Expand distributive operators” detects a `Mul` object with at least one sub-expression that is an `Add` object. It selects the first such sub-expression it finds and replaces the `Mul` object with a new `Add` object. The operands to this new `Add` object are `Mul` objects each of which has as operands one of the operands from the original `Add` object as well as all of the operands of the original `Mul` object except for the original `Add` object.

`Expression` objects also support the following more specialized manipulations which make use of the above basic manipulations.

- Drive expression towards polynomial (iterative)
- Collect like terms (from a polynomial)

“Drive expression towards polynomial” is a sequence of basic manipulations that, when applied repeatedly, is a heuristic for putting expressions into polynomial form. Polynomial form is defined here to mean an `Add` object, the operands of which are all either `Constant`, `Variable`, or `Mul` objects. If they are `Mul` objects then they have only two operands themselves, one of which is a `Constant` and the other of which is a `Variable` object. Not all expressions can be manipulated into this form but the `isPolynomial` method on `Expression` objects detects when it has been achieved. “Drive expression towards polynomial” is primarily used while resolving algebraic loops in equation sets extracted from linear bond graph models, in which case the preconditions exist to make a polynomial form always achievable. The `towardsPolynomial` method on `Expression` objects applies the following sequence of operations each time it is invoked:

1. Aggregate constants
2. Convert constant divisors
3. Aggregate commutative operators
4. Expand distributive operators.

`Equation` objects support the following manipulations which make use of the above `Expression` object manipulations.

- Reduce equation to polynomial form

- Collect like terms (from a polynomial form equation)
- Solve polynomial form equation for any variable.

“Reduce equation to polynomial form” invokes `towardsPolynomial` on the left and right-hand side expressions of the equation repeatedly until `isPolynomial` gives a true result for each. “Collect like terms” examines the variables in the operands of the `Add` objects on either side of a polynomial form equation and collects them such that there is only one operand between those `Add` objects that contains any one variable. The result may not be in the polynomial form defined above but can be driven there again by invoking the equation’s `towardsPolynomial` method. `Equation` objects have a `solveFor` method that, given a variable appearing in the equation attempts to put the equation in polynomial form then collects like terms, isolates the term containing that variable on one side of the equation and divides through by any constant factor. Of course this works only for equations that can be put in the polynomial form defined above. The `solveFor` method attempts to put the equation in polynomial form by invoking the `towardsPolynomial` repeatedly, stopping either when the `isPolynomial` method gives a true result or after a number of iterations proportional to the size of the equation (number of `Expression` objects it contains). If the iteration limit is reached, then it is presumed not possible to put the equation in polynomial form, and the `solveFor` method fails.

`EquationSet` objects support the following manipulations which make use of the above `Equation` and `Expression` object manipulations.

- Detect and resolve algebraic loop(s)
- Recursive backsubstitution from declarative form equations
- Reduce equation set to state space form

These are described in detail in the next two sections.

### 4.4.3 Algebraic loop detection and resolution

Equation sets extracted from bond graph models sometimes contain algebraic loops in which dependencies of a variable include itself. Here, the dependencies of a variable are defined to be the variables on the right-hand side of its declarative form equation plus the dependencies of all of those. Recall

that equations extracted from a bond graph are all in declarative form. Algebraic loops, if not identified and resolved, will cause infinite regression in the process of reducing equations to state space form by back-substitution of variables declared by algebraic equations into the differential equations, as described in Section 2.5.1. Loops in the dependency graph which pass through a differential operator are not problematic to this process. These differential-algebraic loops encode the linkage among state variables and need not be resolved before hand.

Algebraic loops do not represent any general physical attributes of the model as do the causal linkages described in Section 2.3.1 (figures 2.12 and 2.12). They are simply computational inconveniences. Fortunately, for linear systems (bond graph models containing only the standard elements described in Section 2.2), it is always possible to resolve (break) algebraic loops and the process has been automated as part of this work. Less fortunately, the process is not easily generalized to non-linear systems which may contain complex trigonometric expressions and non-invertible functions.

The process, in brief, is to identify algebraic loops with a depth-first search of the dependency graphs. The search is started once from each node (variable) in the graph that appears as the left-hand side of an algebraic declarative equation. Once identified, a purely algebraic loop is resolved by

1. Traversing the loop and collapsing the declarative equations for all loop variables into one equation by back substitution into the declarative equation for the first loop variable
2. Expanding distributive operators, aggregating constants and commutative operators
3. Collecting like terms
4. Isolating the first loop variable

The result of the first step is a new declarative equation for the first loop variable which contains only the variable itself and non-loop variables on the right-hand side. It is always possible, by applying the operations listed in the second step, to reduce the right-hand expression of the collapsed loop equation to the form of a sum of products where each product is between a variable and a constant. This is possible because the algebraic declarative equations extracted from bond graph models have one of only two forms

$$A = B \cdot C$$

and

$$D = E_1 + E_2 + \dots + E_n$$

where  $C$  is a constant and any of  $E_1 \dots E_n$  may be prefixed by a negative sign. The third and fourth steps bring the product containing the first loop variable out of the sum and onto the right-hand side, then factor out the variable and divide through by the constant part.

The result of this process is a new declarative equation for one loop variable in terms only of variables from outside the loop. This equation is used to replace the original declarative equation in the set, breaking the loop. For sets of equations that contain multiple (and possibly overlapping) algebraic loops, the process can be applied repeatedly to break one loop at a time, substituting the result back into the set before searching again for loops. It is important that this is done sequentially since attempting to solve all loops are once before substituting the collapsed and refactored equations back into the set often results in an equal number of different loops in the now changed equation set.

Listing 4.7 outlines the process for the loop in the reduced model shown in figure 4.9. The resulting set of equations is given in listing 4.8. The dependency graph for those equations is given in figure 4.13 and shows no remaining algebraic loops. The entire process is automated.

Listing 4.7: Resolving a loop in the reduced model

```
loop: [ 'e6 ', 'e8 ', 'f8 ', 'f6 ', 'f3 ', 'e3 ' ]
      e8 = e6
      f8 = (e8/30.197788376)
      e3 = (f3*32.0)
      f3 = f6
      f6 = (f4+f7+f8)
      e6 = (e2+(-1.0*e3))
collapsing
e6 = (e2+(-1.0*((f4+f7+(e6/30.197788376))*32.0)))
expanding and aggregating
iterations: 3 True True
e6 = (e2+(f4*-32.0)+(f7*-32.0)+(e6*-1.05968025213))
collecting
```

```

(e6+(-1.0*(e6*-1.05968025213)))
  = (e2+(f4*-32.0)+(f7*-32.0))

expanding and aggregating
iterations: 2 True True
(e6+(e6*1.05968025213)) = (e2+(f4*-32.0)+(f7*-32.0))

isolating
e6 = (((e2*1.0)+(f4*-32.0)+(f7*-32.0))/2.05968025213)
done

```

Listing 4.8: Equations from the reduced model with loops resolved

```

e1 = (e7+(-1.0*e5))
e2 = i0
e3 = (f3*32.0)
e4 = e6
e5 = (f5*45.0)
e6 = (((e2*1.0)+(f4*-32.0)+(f7*-32.0))/2.05968025213)
e7 = e6
e8 = e6
f1 = (p1/-78.0)
f2 = f6
f3 = f6
f4 = (p4/0.0429619641442)
f5 = f1
f6 = (f4+f7+f8)
f7 = f1
f8 = (e8/30.197788376)
p1_dot = e1
p4_dot = e4

```

#### 4.4.4 Reducing equations to state-space form

Equations generated by the bond graph modelling library described in Section 4.3 always appear in “declarative form” meaning that they have on the left-hand side either a single, bare variable or the first time derivative of a variable. Moreover, no variable ever appears on the left-hand side of more



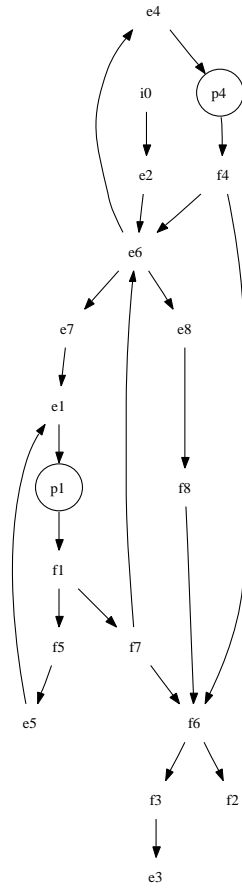


Figure 4.13: Algebraic dependencies within the reduced model with loops resolved

than one equation. These two properties are crucial to the procedure for reducing sets of equations to state space form. If either condition does not exist (for example because the bond graph model has integral causality for some elements) then the procedure is aborted. After checking that all equations are in declarative form, checking for algebraic loops and resolving any that appear, the procedure has three steps.

1. Identify *state*, *input*, and *other* variables
2. Back-substitute until only *state* and *input* variables on right-hand side
3. Split *state* and *readout* equation sets

*State* variables are those that appear within a differential. These are the energy variables from each storage element in the bond graph model. The value of these variable at any point in time encodes the state of the system, that is the precise distribution of energy within it. Since these variables together entirely describe the state of the system, a vector formed by them positions the system within a state-space. The state-space form being sought is one in which a numerical integrator can be used to solve for the value of this state vector over time. That is, for the systems trajectory through state space.

*Input* variables are those that do not appear on the left-hand side of any declarative form equation. Since there is no explicit definition for them in the equation set, and there will not be one in the reduced set either, these variables represent inputs to the system. *Other* variables are those that appear on the left-hand side of a purely algebraic (non-differential) equation. These are parameters of the system that can later be solved for in terms of the state variables.

In the second step, the right-hand side of non-differential equations is substituted into the right-hand sides of the all equations wherever the left-hand side variable of those non-differential equations appears. This continues recursively until all equations have only input and state variables on their right-hand sides. If the equation set contains algebraic loops then this process will never terminate, so it is very important to detect and resolve and algebraic loops before attempting reduction to state space form.

The result of this recursive back-substitution is a set of declarative form equations, some differential and some purely algebraic, and all having only state and input variables on the right-hand side. The third step is to segregate

the differential equations from the purely algebraic ones. The differential equations are in declarative form for the state vector and will be passed to a numerical integration routine to find the system trajectory in state space given a sequence of input variable values over time. The remaining purely algebraic equations can be used to solve for any of their left-hand side variables given the system trajectory and the input sequences.

#### 4.4.5 Simulation

Once a state space equation set has been constructed, it can be solved over any range of time for which a sequence of input values is available. Autonomous system, with no inputs can be solved for arbitrary time ranges of course. The differential equations from a state space equation set are solved by numerical integration using the SciPy [37] provided interface to the LSODA [32] integrator. The integration routine solves for the system trajectory in state space by integrating both sides of the state variable declarative form differential equations. This is the fundamental reason why integral causality is preferred to differential causality in bond graph models. Numerical differentiation with a centred difference implies knowledge of the future, whereas numerical integration is a simple summing up over the past.

Inputs, if any, are read from a file then interpolated on-the-fly to allow the integration routine to evaluate the input at any point within the available range of simulation time. Zeroth order (constant) and a first order (linear) interpolators are implemented. The zeroth order hold function was used for all simulation results presented in chapter 5.

Once the system trajectory is in hand, it and the interpolated inputs are used to solve the remaining algebraic equations across the same range of simulated time for any interesting non-state variables. This is a straightforward application of the algebraic expression solving capabilities described in previous sections.

### 4.5 A typed genetic programming system

A strongly typed genetic programming system requires extra attention to the design of the crossover and mutation operators. These operators must ensure that the type system is adhered to so that the result of any operation is still a valid program in the language.

The design described here isolates the crossover and mutation operators from any details of the problem being solved by the genetic algorithm. To solve a different class of problem (e.g. graph-based modelling instead of symbolic regression), a new set of program node types is implemented. That is, the grammar for a different genetic programming language is defined. Each new node type derives from a base class implementing the crossover and mutation mechanisms in a generic way. To interoperate with those mechanisms, each node type must have the following attributes:

**type** a unique type identifier

**min children** an integer, the minimum number of children it will accept

**max children** an integer, the maximum number of children it will accept

**child types** a list of types at least *max children* long

Each node must also provide a method (function) that may be invoked to compose a partial solution to the problem from the results of invoking the same method on its children (the nodes below it in the program tree). The “composed partial solution” returned by the root node of a genetic program is taken to be a solution to the whole problem (e.g. a set of differential-algebraic equations or a complete bond graph).

Optionally, a kernel of a solution may be passed in to the root node when it is invoked. That kernel is a prototype solution to the problem incorporating whatever *a priori* knowledge the experimenter has. It might be a set of equations that form a low order approximation of the system behaviour, or an idealized graph model, or a graph model that is highly detailed in parts that have been identified by other methods and sketched in very simply in other parts. The invoked method of each node may pass all or part of the kernel on to its children when calling them, use all or part of the kernel when composing its result, or even ignore the kernel entirely (removing part of the original kernel from the ultimate result). These details are specific to the design of the node types with a particular genetic programming language grammar and the overall genetic programming framework is not affected by these choices. For example, the genetic programming type system for symbolic regression presented in section 4.5.3 does not use a kernel while the system for bond graphs presented in section 4.5.4 does use a kernel in the form of a bond graph model.

### 4.5.1 Strongly typed mutation

The implemented mutation operator for strongly typed tree structured representations proceeds in three steps.

1. Choose a node at random from the parent
2. Generate a new random tree whose root has the same type as the chosen node
3. Copy the parent but replace the subtree rooted at the chosen node with the new random tree

In the first step a choice is made uniformly among all nodes. It is not weighted by the position of the node within the tree or by the type of the node. In the second step it is always possible to generate a tree rooted with a node of the same type as the chosen node since all nodes in the parent (including the chosen node) come from the function and terminal set of which the replacement tree will be built. In the third step of the mutation operator for strongly typed tree structured representations, the parent tree is copied with replacement of the chosen subtree.

An attempt is made also to create the new tree at approximately the same size (number of nodes) and depth as the subtree to be replaced. The tree is constructed breadth first. Each time a new node is added, the required type is first checked and a list of functions and terminals conforming to that type is assembled. A function or terminal is chosen from the set at random, with functions preferred over terminals by a configurable ratio. If the tree exceeds either the size or depth of the tree to be replaced, then terminals are used wherever possible (i.e. a function will only be used if no terminal conforms to the required type). For the type systems (genetic programming language grammars) described in sections 4.5.3 and 4.5.4 this most often results in not more than one more layer of leaf nodes being added to the tree. This works out so well because the tree is constructed breadth first and in these type systems there are very few function types that accept only other function types as children and these are closely tied to the one type used as the root of a full genetic program so their children all tend to be filled in before the tree hits a size or depth limit.

The tree creation procedure described here is also used to create random individuals at the start of a genetic programming run. In that case the size

and depth limits imposed on the initial population are arbitrary and left to the experimenter to choose for each run.

### 4.5.2 Strongly typed crossover

The implemented mutation operator for strongly typed tree structured representations proceeds in three steps.

1. Find common node types by intersecting the node type sets of the parents
2. List all nodes from the first parent conforming to one of the common types
3. Choose at random from the list
4. List all nodes from the second parent conforming to the same type as the chosen node
5. Choose at random from the list
6. Copy the first parent but replace the subtree rooted at the first chosen node with a copy of that rooted at the second

If the set of common types is empty, crossover cannot proceed and the genetic algorithm selects a new pair of parents.

### 4.5.3 A grammar for symbolic regression on dynamic systems

This section describes some challenges encountered while designing a genetic programming grammar (type system) for symbolic regression targeted at modelling dynamic systems and presents the finished design.

Symbolic regression refers to the use of genetic algorithms to evolve a symbolic expression that fits some given sampled data. In contrast to linear regression and other types of curve fitting, the structure of the expression is not specified before hand. The grammar for this conventional form of symbolic regression consists of random constant values, a node representing the independent variable, and standard algebraic operators (with the possible replacement of division by an operator protected against division by zero).

This grammar is typeless or “mono-typed”. The result of symbolic regression is a tree structured algebraic expression, a scalar function of one variable. The genetic programming grammar for bond graphs given in section 4.5.3 uses this conventional form of symbolic regression, with the omission of any independent variable nodes, to evolve constant valued expressions for any needed parameters of graph elements.

Symbolic regression as a technique for identifying dynamic systems presents additional challenges. The system under test may be multi-input and multi-output, and being dynamic requires that the regressed symbolic model have some representation of state. The chosen (and obvious, perhaps) representation is a set of equations in state space form, such as those extracted and reduced from a bond graph model with all integral causality (see section 4.4.4).

Recall that a state space equation set consists of two subsets. The state equations are first order differential equations. The differential operands on the left-hand side of these equations make up the state vector. The readout equations are a set of declarative form purely algebraic equations that define a number of output variables in terms of the state and input variables. For a black-box system identification problem, it is desired to specify the number of inputs and the number of outputs but nothing further. The number of state variables (length of the state vector) should not need to be pre-specified. It is simple enough to draw up a tree-structured representation of a state space equation set, but the desire to not pre-specify the number of states requires that the representation contain some facility by which the genetic algorithm can vary the number and interconnection of the state equations in the course of applying the usual genetic operators (crossover and mutation).

In addition, the operation that adds a new state equation should disrupt as little as possible any existing structure in a partially successful program. Simply making the new state available for later integration into the model is enough. Given a tree structured representation in which a “state space equation set” sits above a “state equation set” and a “readout equation set” each of which sit above a number of equations, there are several ways in which new state equations might be added to the “state equation set”.

1. A new “blank” state equation could be added where “blank” means, for example, that the right-hand side is constant
2. A new random state equation could be added where the right-hand side is a random expression in terms of input and state variables

3. An existing state equation could be copied, leaving the copies untied and free to diverge in future generations.

These all leave open the question of how the new state variable is linked (eventually) into the existing set of state equations. Also, if a state equation is removed (because a crossover or mutation operator deletes that part of the program tree) there is the question of what to do with references to the deleted state variable within the other state equations and within the readout equations.

The implemented solution assigns to each state variable an index on some interval (a real number between 0 and 1, for example). References to state variables made from the right-hand side of state and readout equations do not link directly to any program object representing the state variable. Rather they link to an index on the interval. The indices assigned to state variables and indices assigned to references to state variables need not align exactly. When it comes time to evaluate the program, references to a state on the right-hand side of any equations are resolved to state variables defined on the left-hand side of state equations by starting from the reference index and searching in one direction, modulo the length of the interval, until the first state variable index is encountered.

When a new random state equation is introduced (e.g. through mutation) it is given a random index on the interval. When a new state equation is introduced by copying another (e.g. through crossover) it gets exactly the same index as the original. When the program is evaluated, immediately prior to resolving state variable references, the duplicate state variable indices are resolved as follows. Starting from one end of the interval and proceeding in the opposite direction to the variable resolution search, any duplicate index is replaced by a number half way between its original index and the next state equation index (with wrap around at the interval boundaries). Overall, this design has some desirable properties, including:

- Random expressions can be generated that refer to any number of state variables and all references will resolve to an actual state variable, no matter how many states the program describes at this or any later point in the genetic programming run.
- Adding an identical copy of a state equation has no affect until the original or the copy is modified, so crossover that simply adds a state is not destructive to any advantages the program already has.



- When one of those equations is modified, the change has no effect on parts of the system that were not already dependent on the original state equation (but half, on average, of the references to the original state variable become references to the new state variable).

The second and third points are important since other designs considered before arriving at this one, such as numbering the state equations and using very large numbers modulo the final number of equations to reference the state variables, would reorganize the entire system every time a new state equation was introduced. That is, every change to the number of states would be a major structural change across the entire system even if the new state is not referenced anywhere in the readout or other state equations. This new scheme allows both small, incremental changes (such as the introduction of a new state equation by copying and later modifying in some small way an old one) and large changes (such as part or all of the state equation set being swapped with an entirely different set) both through the standard crossover operation.

The type system is illustrated in table 4.5.3. The first column gives a name to every every possible node. The second column lists the type to which each node conforms. This determines which positions within a valid program that node may legally occupy. The third column lists the types of the new positions (if any) that the node opens up within a program. These are the types that nodes appearing below this one in a program tree must conform to.

Every program tree is rooted at a `DynamicSystem` node. The program is evaluated by calling on the root node (a `DynamicSystem` instance) to produce a result. It will return a state space equation set using the symbolic algebra library described earlier in this chapter (i.e. the same implementation used by the bond graph modelling library). The `DynamicSystem` node does this by first calling on its children to produce a set of state and readout equations, then resolving variable references using the index method described above, and finally combining all the equations into one state space equation set object. `StateEquationSet` and `OutputEquationSet` nodes produce their result by calling on their first child to produce an equation and their second child, if present, to produce a set of equations, then returning the aggregate of both. `StateEquation` and `OutputEquation` nodes call on their own children for the parts to build an equation compatible with the simulation code, and so on.

Node	Type	Child types
DynamicSystem	DynamicSystem	StateEquationSet, OutputEquationSet
StateEquationSet	StateEquationSet	StateEquation, StateEquationSet
StateEquation	StateEquation	StateVariableLHS, Expression
OutputEquationSet	OutputEquationSet	OutputEquation, OutputEquationSet
OutputEquation	OutputEquation	OutputVariable, Expression
StateVariableLHS	StateVariableLHS	
StateVariableRHS	Expression	
OutputVariable	OutputVariable	
InputVariable	Expression	
Constant	Expression	
Add	Expression	Expression, Expression
Mul	Expression	Expression, Expression
Div	Expression	Expression, Expression
Cos	Expression	Expression
Sin	Expression	Expression
Greater	Expression	Expression, Expression
Lesser	Expression	Expression, Expression

Table 4.1: Grammar for symbolic regression on dynamic systems

#### 4.5.4 A grammar for genetic programming bond graphs

This section presents a genetic programming grammar (type system) that can be used to generate bond graph models including any constant parameters of the elements. Used for system identification this does combined structural and parametric identification. Genetic programs from this language are each rooted at a `Kernel` node. This node embeds an ordinary bond graph model, some elements of which have been marked as replaceable by the program. The grammar, as given, only allows bonds and constant parameters of elements to be replaced but could be extended to other elements or even connected subgraphs to be replaced.

If a constant parameter of some element (for example the capacitance of a `C` element) is marked as replaceable in the kernel, that creates an opening for a new node conforming to the `Expression` type below the kernel in the program. The `Expression` types are all borrowed from the symbolic regression grammar presented in the previous section. Importantly, all of the variable nodes have been omitted so the expressions generated here are always constant valued.

If a bond is marked as replaceable in the kernel, that creates an opening for a new node conforming to the `bond` type below the kernel in the program. Two nodes in table 4.5.4 conform to that type. When the program is evaluated, these nodes replace the given bond with two bonds joined by a 0- or a 1-junction. Consequently, they create openings for two new nodes conforming to the `bond` type, one conforming to either the `effort-in` or the `effort-out` type and an unlimited number of nodes conforming to the complementary type.

The `effort-in` and `effort-out` types apply to nodes that will insert some number of elements joined to the rest of the graph by one bond and having preferred causality such that the bond has causality `effort-in` or `effort-out` from the rest of the graph. The nodes included in table 4.5.4 each insert just one element plus the attaching bond. These considerations ensure that, if sequential causality assignment (SCAP) applied to the kernel model would produce an integral causality model, then SCAP will produce an integral causality model from the output of any genetic program in the grammar as well. If an integral causality model is desired (because, for example, only a differential equation solver is available, not a differential-algebraic solver, and integral causality models produce state-space models), then this is a great efficiency advantage. The grammar will not produce models incompatible with

the solver, constraining the genetic algorithm to not waste time generating models that will only be discarded.

The child types column for the `Kernel` node is left blank since the list of openings below the kernel is specific to the kernel and which elements have been marked replaceable. The child types column for the `Constant` node is left blank because it is a terminal node and presents no openings below itself in a program tree.

Node	Type	Child types
Kernel	kernel	
InsertCommonEffortJunction	bond	bond, bond, effort-in, effort-out*
InsertCommonFlowJunction	bond	bond, bond, effort-in*, effort-out
AttachInertia	effort-out	Expression, bond
AttachCapacitor	effort-in	Expression, bond
AttachResistorEffortIn	effort-in	Expression, bond
AttachResistorEffortOut	effort-out	Expression, bond
Constant	Expression	
Add	Expression	Expression, Expression
Mul	Expression	Expression, Expression
Div	Expression	Expression, Expression

Table 4.2: Grammar for genetic programming bond graphs

# Chapter 5

## Results and discussion

### 5.1 Bond graph modelling and simulation

Included below are two simple, linear models using only standard, linear bond graph elements and four non-linear models using modulated, complex, and compound elements. The models are each named after a mechanical analog. Simulations are performed using step and noise inputs or, for the autonomous models, starting from a non-equilibrium state.

#### 5.1.1 Simple spring-mass-damper system

Figure 5.1 shows a simple, idealized, linear spring-mass-damper system in schematic form. The system consists of a mass,  $m$ , attached to a spring,  $c$ , and a dashpot,  $r$ , in parallel. An equivalent bond graph model for this system is shown in figure 5.2. The element parameters are

$$\begin{aligned}C &= 1.96 \\I &= 0.002 \\R &= 0.02\end{aligned}$$

Equations 5.1 to 5.10 are the collected constitutive equations from each node in the system. The  $e$ ,  $f$ ,  $p$ , and  $q$  variables measure effort, flow, momentum, and displacement on the like numbered bonds. The applied effort from the source element is marked  $E$ .

$$\dot{p}_3 = e_3 \quad (5.1)$$

$$\dot{q}_1 = f_1 \quad (5.2)$$

$$e_1 = (q_1/1.96) \quad (5.3)$$

$$e_2 = (f_2 * 0.02) \quad (5.4)$$

$$e_3 = (e_4 + (-1.0 * e_1) + (-1.0 * e_2)) \quad (5.5)$$

$$e_4 = E \quad (5.6)$$

$$f_1 = f_3 \quad (5.7)$$

$$f_2 = f_3 \quad (5.8)$$

$$f_3 = (p_3/0.002) \quad (5.9)$$

$$f_4 = f_3 \quad (5.10)$$

Equations 5.11 and 5.12 are the differential state equations and equations 5.13 to 5.20 are the readout equations for the remaining non-state variables. Both are produced by the automated procedure described in Section 4.4.4. There are two state variables  $p_3$  and  $q_1$ . These are the energy variables of the two storage elements (C and I). Note that variable  $E$  does not appear on the left-hand side of any state or readout equation. It is an input to the system.

$$\dot{p}_3 = (E + (-1.0 * (q_1/1.96)) + (-1.0 * ((p_3/0.002) * 0.02))) \quad (5.11)$$

$$\dot{q}_1 = (p_3/0.002) \quad (5.12)$$

$$e_1 = (q_1/1.96) \quad (5.13)$$

$$e_2 = ((p_3/0.002) * 0.02) \quad (5.14)$$

$$e_3 = (E + (-1.0 * (q_1/1.96)) + (-1.0 * ((p_3/0.002) * 0.02))) \quad (5.15)$$

$$e_4 = E \quad (5.16)$$

$$f_1 = (p_3/0.002) \quad (5.17)$$

$$f_2 = (p_3/0.002) \quad (5.18)$$

$$f_3 = (p_3/0.002) \quad (5.19)$$

$$f_4 = (p_3/0.002) \quad (5.20)$$

The state variable values are found by numerical integration of the state equations. The value of  $E$  at any point in simulated time is interpolated from a prepared data file as needed. The remaining non-state and non-input variables are evaluated across simulated time by substituting state variable values into the right-hand side of readout equations as needed.

Figure 5.3 shows the model responding to a unit step in the applied force,  $E$ . The displacement and flow variables of the 1-C bond are plotted. Since the junction is a common flow (“1”) junction, all 4 bonds have the same displacement and flow value (equations 5.7, 5.8, 5.10).  $f_1$  and  $q_1$  shown in figure 5.3 are the speed and position of the spring, the mass, and the damper. Predictably, the speed rises immediately as the force is applied. It crosses zero again at each of the position extrema. The position changes smoothly, with inflection points at the speed extrema. As the work done by the applied force is dissipated through the damper, the system reaches a new steady state with zero speed and a positive position offset where the spring and the applied force reach an equilibrium.

Figure 5.4 shows the model responding to unit magnitude uniform random noise in the applied force,  $E$ . This input signal and model response were used for the experiments described in Section 5.3. A broad spectrum signal, such as uniform random noise, exercises more of the dynamics of the system, which is important for identification.

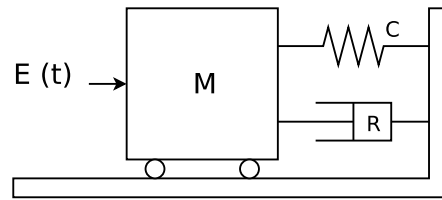


Figure 5.1: Schematic diagram of a simple spring-mass-damper system

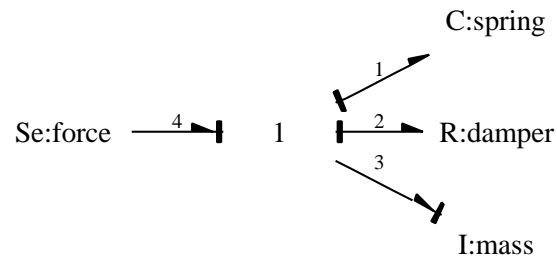


Figure 5.2: Bond graph model of the system from figure 5.1



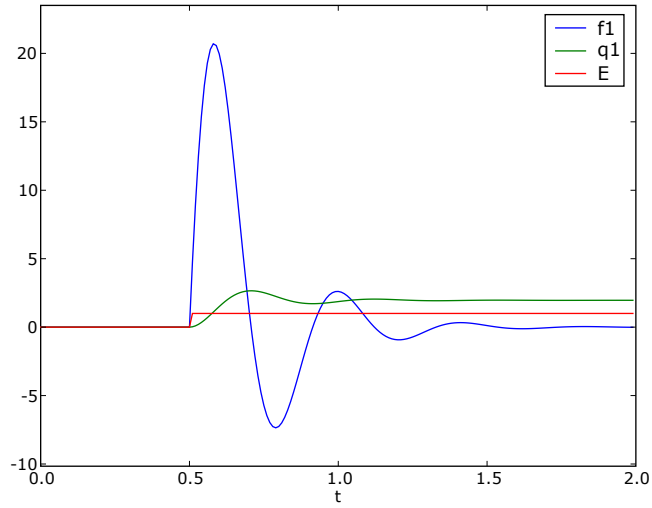


Figure 5.3: Response of the spring-mass-damper model from figure 5.2 to a unit step input

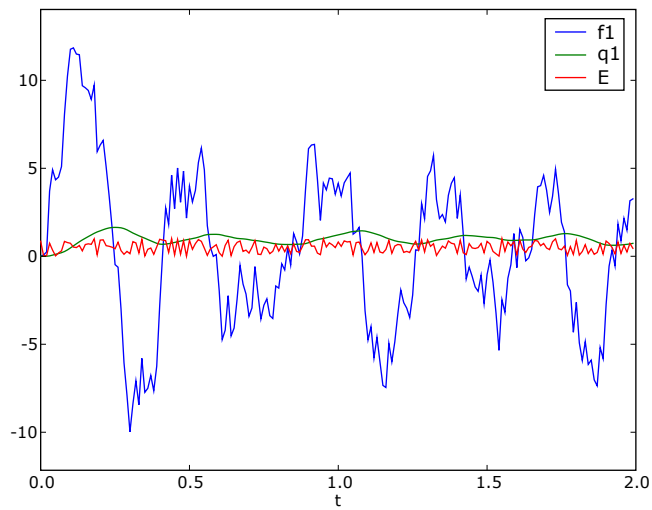


Figure 5.4: Response of the spring-mass-damper model from figure 5.2 to a unit magnitude uniform random noise input

### 5.1.2 Multiple spring-mass-damper system

Figure 5.5 shows, in schematic form, a series of point masses connected by idealized, linear spring-dashpot shock absorbers. An equivalent bond graph model for this system is given in figure 5.6. The element parameters are

$$\begin{aligned} C_1 = C_2 = C_3 &= 1.96 \\ I_1 = I_2 = I_3 &= 0.002 \\ R_1 = R_2 = R_3 &= 0.02 \end{aligned}$$

There are 38 constitutive equations in the model. When reduced to form, the differential state equations are as given in equations 5.21 to 5.21.

$$\begin{aligned} \dot{p}_2 &= (E + (-1.0 * (((p_2/0.002) + (-1.0 * (p_8/0.002))) * 0.02) \\ &\quad + (q_6/1.96)))) \\ \dot{p}_8 &= (((((p_2/0.002) + (-1.0 * (p_8/0.002))) * 0.02) + (q_6/1.96)) \\ &\quad + (-1.0 * (((p_8/0.002) + (-1.0 * (p_{14}/0.002))) * 0.02) \\ &\quad + (q_{12}/1.96)))) \\ \dot{p}_{14} &= (((((p_8/0.002) + (-1.0 * (p_{14}/0.002))) * 0.02) + (q_{12}/1.96)) \\ &\quad + (-1.0 * ((p_{14}/0.002) * 0.02)) + (-1.0 * (q_{16}/1.96))) \\ \dot{q}_6 &= ((p_2/0.002) + (-1.0 * (p_8/0.002))) \\ \dot{q}_{12} &= ((p_8/0.002) + (-1.0 * (p_{14}/0.002))) \\ \dot{q}_{16} &= (p_{14}/0.002) \end{aligned}$$

Figure 5.7 shows the multiple spring-mass-damper system modelled in figure 5.6 responding to a unit step in the applied force,  $E$ . Variables  $q_6$ ,  $q_{12}$ , and  $q_{16}$  are plotted, showing the displacement of the 3 springs from their unsprung lengths. When the step input is applied,  $q_6$  rises first as it is most directly coupled to the input, followed by  $q_{12}$  then  $q_{16}$ . The system is damped, but not critically, and so oscillates before settling at a new steady state. Since the capacitances are identical, all springs are displaced by an equal amount at steady state.

Figure 5.8 shows the multiple spring-mass-damper system modelled in figure 5.6 responding to unit magnitude uniform random noise in the applied force,  $E$ . The spring (capacitor) displacements are shown again. Here, the

system acts as a sort of mechanical filter. The input signal,  $E$ , is broad spectrum but the output signal at subsequent stages ( $q_6, q_{12}, q_{16}$ ) is increasingly smooth.

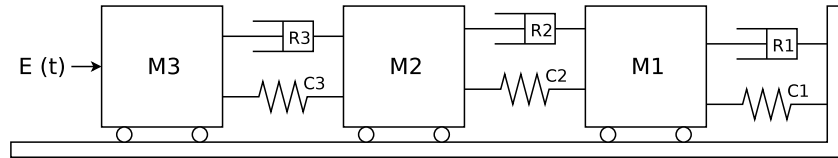


Figure 5.5: Schematic diagram of a multiple spring-mass-damper system

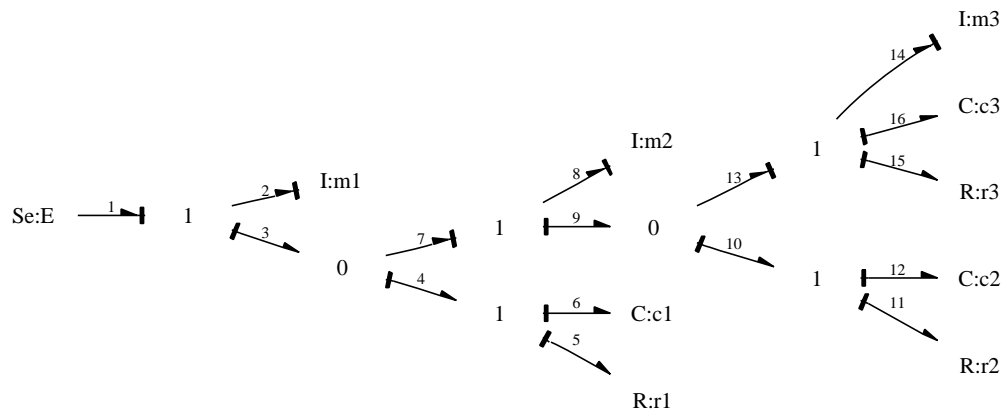


Figure 5.6: Bond graph model of the system from figure 5.5

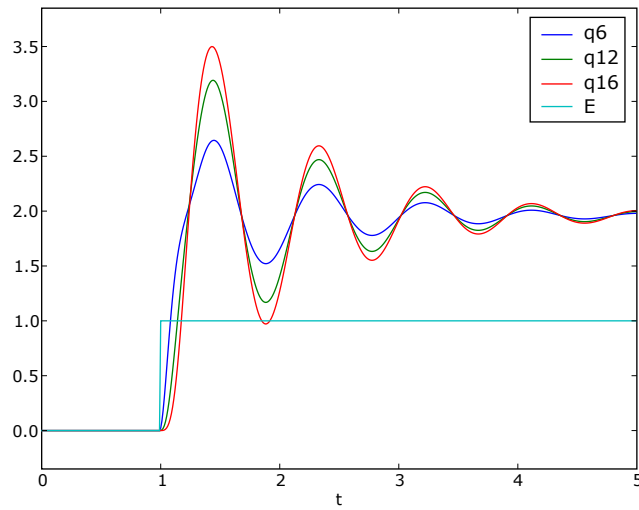


Figure 5.7: Response of the multiple spring-mass-damper model from figure 5.6 to a unit step in the applied force,  $E$

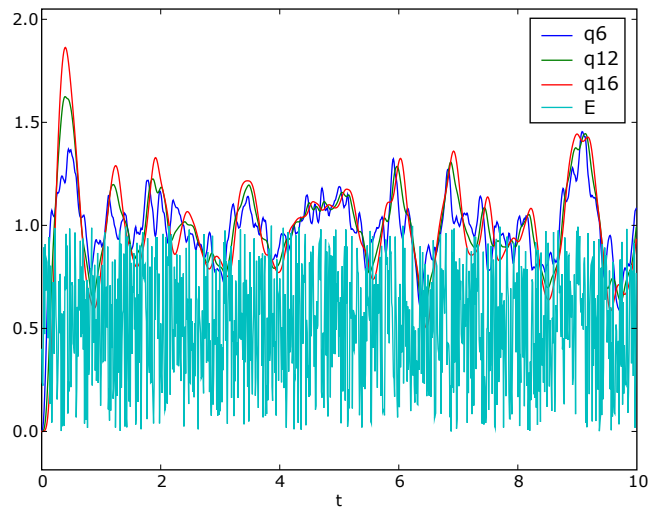


Figure 5.8: Response of the multiple spring-mass-damper model from figure 5.6 to unit magnitude uniform random noise in the applied force,  $E$

### 5.1.3 Elastic collision with a horizontal surface

Figure 5.9 shows an elastic ball released from some height above a rigid horizontal surface. Figure 5.10 is a bond graph model of the system using the CC element defined in section 2.4.3. The included R element represents viscous drag on the ball from the environment.

Parametric details of the model are as follows. The inertia,  $I$ , (mass of the ball) is 0.002; the capacitance,  $C$ , (inverse spring constant of the ball) is 0.25; the capacitance switching threshold,  $r$ , (radius of the ball) is 1.0; the resistance,  $R$ , (drag coefficient) is 0.001; and the effort source is a constant  $-9.81$  times the inertia (force of gravity on the ball).

Figure 5.11 shows the motion of centre of mass (capacitor displacement) of the bouncing ball modelled in figure 5.10 plotted against time. The ball is released from a position above the surface, falls, and bounces repeatedly. Each bounce is shorter in time and displacement than the last, as energy is dissipated through the sink element,  $R$ . The low point of each bounce is not as low as the last since the ball strikes the surface ever more gently and is not compressed as much.

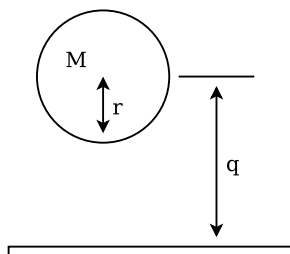


Figure 5.9: An elastic ball bouncing off a rigid horizontal surface

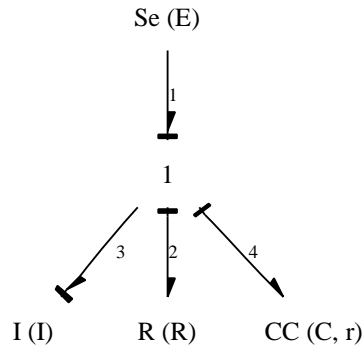


Figure 5.10: Bond graph using a special switched C-element to model an elastic ball bouncing off a rigid horizontal surface

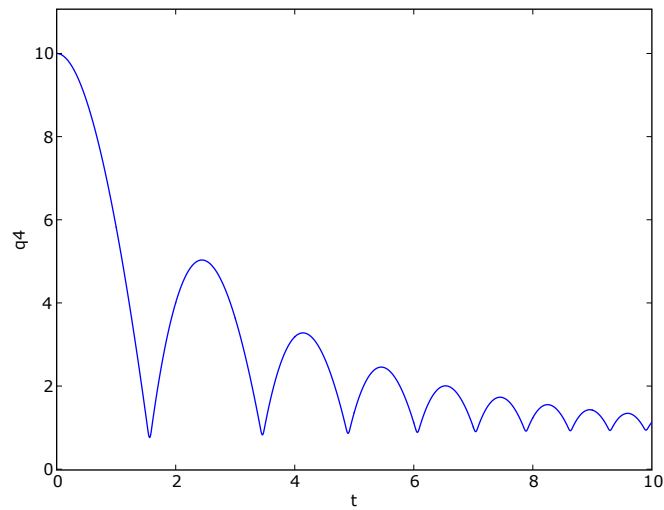


Figure 5.11: Response of the bouncing ball model from figure 5.10 released above a rigid horizontal surface

### 5.1.4 Suspended planar pendulum

Figure 2.19 shows a planar pendulum in schematic form. An equivalent bond graph model is shown in figure 5.12.

Note that the pin-joint is suspended by a parallel damper and spring in the horizontal and vertical directions. The model in figure 5.12 avoids differential causality by not rigidly coupling the joined end to the ground. The joint subgraph of the bond graph model represents a pin joint with some horizontal and vertical compliance. The spring constant of this compliance can be made arbitrarily high, but the resulting differential equations used in simulation will be accordingly stiff.

If not connected at the joint, the pendulum-body would enjoy 3 degrees of freedom: horizontal and vertical translation of the centre of mass, plus rotation about the centre. These are reflected in the pendulum body subgraph of the bond graph model by 3 inertia elements, the flow variables associated with which represent the horizontal, vertical and angular momentum of pendulum body. An ideal pin joint, perfectly rigid in the horizontal and vertical directions, would constrain the pendulum-body to have only one degree of freedom. The horizontal and vertical position (and speed) of the pendulum-body would be entirely dependent on its angular position (and speed). A literal model is easily drawn up in bond graph notation by bonding constant zero flow sources to the common flow junctions representing the horizontal and vertical flow of the fixed end of the pendulum-body. That model has 3 storage elements (horizontal, vertical, and rotational inertias) but only 1 degree of freedom so it will necessarily show derivative causality on 2 of the inertia elements. Exactly which two are dependent and which one is independent will be determined by the node ordering used in causality assignment.

Three common flow (“1”) junctions track the horizontal, vertical, and rotational speed of the pendulum body. Attached to these are activated bonds leading to integrating signal blocks (“INT”). Integrating the horizontal, vertical, and rotational flow gives the corresponding position signals. The rotational position signal is fed into four other signal blocks (not shown) each of which performs a scaling and trigonometric function to produce the modulating signal for one of the four modulated transformer (“MTF”) elements. Those unseen signal blocks perform the following operations, where  $\theta$  is the output of signal block “INT1” (that is, rotational position: the integral of

rotational flow signal  $a3$ ).

$$\begin{aligned}
 m_{xA} &= 10 * \cos(\theta) \\
 m_{yA} &= 10 * \sin(\theta) \\
 m_{xB} &= -10 * \cos(\theta) \\
 m_{yB} &= -10 * \sin(\theta)
 \end{aligned}$$

There are 65 other constitutive equations in the model. When reduced to state space form, the differential state equations are:

$$\begin{aligned}
 \dot{n}_{a1} &= (p_1/10.0) \\
 \dot{n}_{a2} &= (p_2/10.0) \\
 \dot{n}_{a3} &= (p_4/11.0) \\
 \dot{p}_1 &= (((-1.0 * (q_{21}/0.01)) + (-1.0 * ((p_1/10.0) \\
 &\quad + ((10.0 * \cos(n_{a3})) * (p_4/11.0))) * 0.8))) + 0.0) \\
 \dot{p}_2 &= (-98.1 + ((-1.0 * (q_{24}/0.01)) + (-1.0 * ((p_2/10.0) \\
 &\quad + ((10.0 * \sin(n_{a3})) * (p_4/11.0))) * 0.8))) + 0.0) \\
 \dot{p}_4 &= (((10.0 * \cos(n_{a3})) * ((-1.0 * (q_{21}/0.01)) \\
 &\quad + (-1.0 * ((p_1/10.0) + ((10.0 * \cos(n_{a3})) * (p_4/11.0))) * 0.8)))) \\
 &\quad + ((10.0 * \sin(n_{a3})) * ((-1.0 * (q_{24}/0.01)) + (-1.0 * ((p_2/10.0) \\
 &\quad + ((10.0 * \sin(n_{a3})) * (p_4/11.0))) * 0.8)))) + ((-10.0 * \cos(n_{a3})) * 0.0) \\
 &\quad + ((-10.0 * \sin(n_{a3})) * 0.0) + (-1.0 * ((p_4/11.0) * 0.5))) \\
 \dot{q}_{21} &= ((p_1/10.0) + ((10.0 * \cos(n_{a3})) * (p_4/11.0))) \\
 \dot{q}_{24} &= ((p_2/10.0) + ((10.0 * \sin(n_{a3})) * (p_4/11.0)))
 \end{aligned}$$

Figure 5.13 shows the planar pendulum modelled in figure 5.12 responding to an initial displacement. The vertical displacement of the centre of mass ( $int\_a2$ ) and the extension of the vertical suspending spring ( $q_{24}$ ) are plotted. The joint stiffness has been reduced to clearly illustrate the horizontal and vertical compliance in the pin joint. The upper extremum of the centre position cycle decreases over time as energy is dissipated from the system and the pendulum swings through lesser angles. The initial impact of the pendulum's downward momentum being arrested by the spring causes a low frequency vibration in the spring that can be seen in the variable lower extremum on the pendulum centre position and spring extension near the



start of the plot. The more persistent cycle shows the spring extending during the lower part of the pendulum swing and contracting to near zero at the high points.

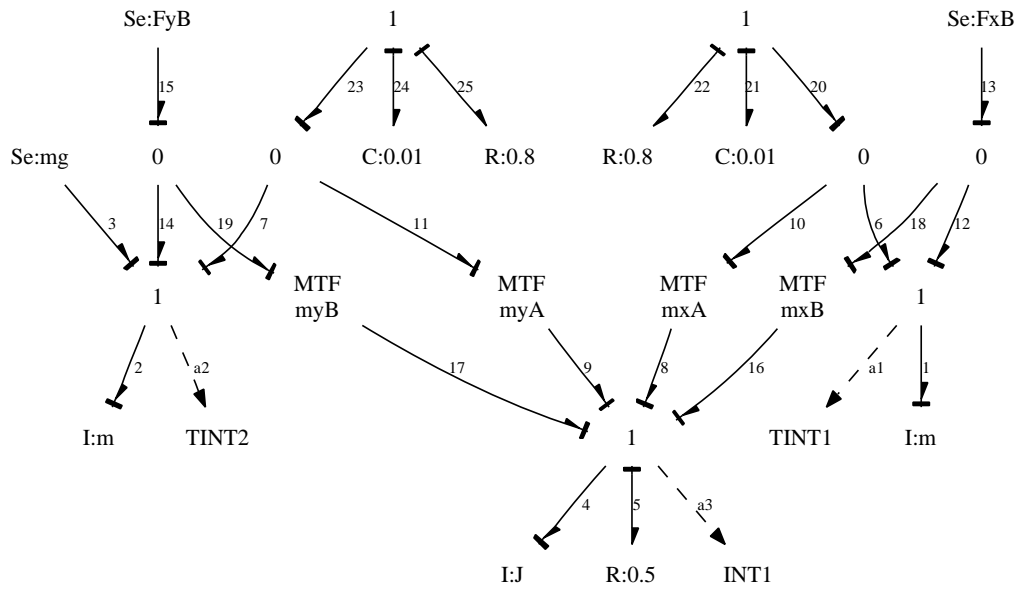


Figure 5.12: Bond graph model of the system from figure 2.19

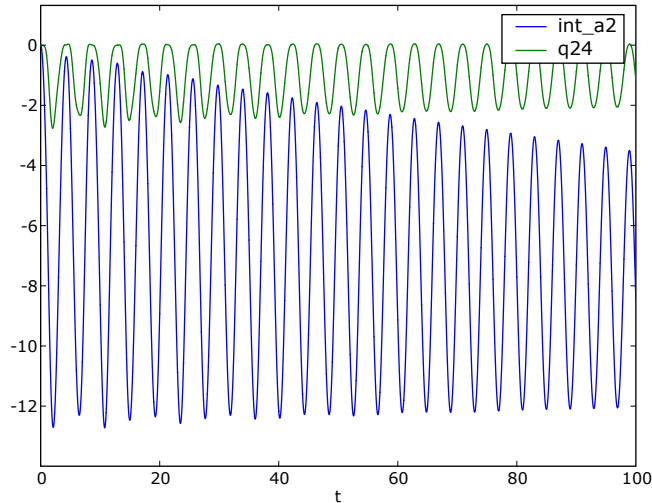


Figure 5.13: Response of the pendulum model figure 5.12 released from an initial horizontal position

### 5.1.5 Triple planar pendulum

Figure 5.17 shows a triple planar pendulum with damped-elastic joints in schematic form on the left and an equivalent bond graph model using compound elements on the right. There are 104 constitutive equations in the model. When reduced to state space form, there are 24 differential state equations (equations 5.21 to 5.21).

Figures 5.14 through 5.16 show the response of the triple planar pendulum modelled in figure 5.17 after being released from an initial position with all 3 links laid out horizontally. Figure 5.14 plots the vertical position of the centres of mass of the 3 pendulum bodies. Figure 5.15 plots the horizontal positions. Figure 5.16 plots the angular displacement of each link from horizontal.

$$\begin{aligned}
V\dot{p}_{hi_b0} &= ((0.0 + (((x_j0/0.001) + ((0.0 + (-1.0 * (Vx C_b0 + (10.0 * Vp_{hi_b0} \\
&\quad * \sin(p_{hi_b0})))))/0.001)) * 10.0 * \sin(p_{hi_b0}) + (-1.0 * ((y_j0/0.001) \\
&\quad + ((0.0 + (-1.0 * (Vy C_b0 + (-1.0 * 10.0 * Vp_{hi_b0} * \cos(p_{hi_b0})))))) \\
&\quad /0.001)) * 10.0 * \cos(p_{hi_b0}) + (-1.0 * 0.0) + ((x_j1/0.001) \\
&\quad + (((Vx C_b0 + (-1.0 * 10.0 * Vp_{hi_b0} * \sin(p_{hi_b0}))) + (-1.0 \\
&\quad * (Vx C_b1 + (10.0 * Vp_{hi_b1} * \sin(p_{hi_b1})))))/0.001)) \\
&\quad * 10.0 * \sin(p_{hi_b0}) + (-1.0 * ((y_j1/0.001) + ((Vy C_b0 + (10.0 \\
&\quad * Vp_{hi_b0} * \cos(p_{hi_b0}))) + (-1.0 * (Vy C_b1 + (-1.0 * 10.0 \\
&\quad * Vp_{hi_b1} * \cos(p_{hi_b1})))))/0.001)) * 10.0 * \cos(p_{hi_b0}))/50.0) \\
V\dot{p}_{hi_b1} &= ((0.0 + (((x_j1/0.001) + (((Vx C_b0 + (-1.0 * 10.0 * Vp_{hi_b0} \\
&\quad * \sin(p_{hi_b0}))) + (-1.0 * (Vx C_b1 + (10.0 * Vp_{hi_b1} \\
&\quad * \sin(p_{hi_b1})))))/0.001)) * 10.0 * \sin(p_{hi_b1}) + (-1.0 \\
&\quad * ((y_j1/0.001) + (((Vy C_b0 + (10.0 * Vp_{hi_b0} * \cos(p_{hi_b0}))) \\
&\quad + (-1.0 * (Vy C_b1 + (-1.0 * 10.0 * Vp_{hi_b1} * \cos(p_{hi_b1})))))) \\
&\quad /0.001)) * 10.0 * \cos(p_{hi_b1}) + (-1.0 * 0.0) + ((x_j2/0.001) \\
&\quad + (((Vx C_b1 + (-1.0 * 10.0 * Vp_{hi_b1} * \sin(p_{hi_b1}))) + (-1.0 \\
&\quad * (Vx C_b2 + (10.0 * Vp_{hi_b2} * \sin(p_{hi_b2})))))/0.001)) * 10.0 \\
&\quad * \sin(p_{hi_b1}) + (-1.0 * ((y_j2/0.001) + ((Vy C_b1 + (10.0 \\
&\quad * Vp_{hi_b1} * \cos(p_{hi_b1}))) + (-1.0 * (Vy C_b2 + (-1.0 * 10.0 \\
&\quad * Vp_{hi_b2} * \cos(p_{hi_b2})))))/0.001)) * 10.0 * \cos(p_{hi_b1}))/50.0) \\
V\dot{p}_{hi_b2} &= ((0.0 + (((x_j2/0.001) + (((Vx C_b1 + (-1.0 * 10.0 * Vp_{hi_b1} \\
&\quad * \sin(p_{hi_b1}))) + (-1.0 * (Vx C_b2 + (10.0 * Vp_{hi_b2} \\
&\quad * \sin(p_{hi_b2})))))/0.001)) * 10.0 * \sin(p_{hi_b2}) + (-1.0 \\
&\quad * ((y_j2/0.001) + (((Vy C_b1 + (10.0 * Vp_{hi_b1} * \cos(p_{hi_b1}))) \\
&\quad + (-1.0 * (Vy C_b2 + (-1.0 * 10.0 * Vp_{hi_b2} * \cos(p_{hi_b2})))))))/0.001)) \\
&\quad * 10.0 * \cos(p_{hi_b2}) + (-1.0 * 0.0) + (0.0 * 10.0 * \sin(p_{hi_b2})) \\
&\quad + (-1.0 * 0.0 * 10.0 * \cos(p_{hi_b2}))/50.0)
\end{aligned}$$

$$\begin{aligned}
Vx\dot{C}_b0 &= (((((x_j0/0.001) + ((0.0 + (-1.0 * (VxC_b0 + (10.0 * Vp_{hi_b0} \\
&\quad * \sin(p_{hi_b0})))))/0.001)) + (-1.0 * ((x_j1/0.001) \\
&\quad + (((VxC_b0 + (-1.0 * 10.0 * Vp_{hi_b0} * \sin(p_{hi_b0}))) \\
&\quad + (-1.0 * (VxC_b1 + (10.0 * Vp_{hi_b1} * \sin(p_{hi_b1})))))) \\
&\quad /0.001))))/10.0) \\
Vx\dot{C}_b1 &= (((((x_j1/0.001) + (((VxC_b0 + (-1.0 * 10.0 * Vp_{hi_b0} \\
&\quad * \sin(p_{hi_b0}))) + (-1.0 * (VxC_b1 + (10.0 * Vp_{hi_b1} \\
&\quad * \sin(p_{hi_b1})))))/0.001)) + (-1.0 * ((x_j2/0.001) \\
&\quad + (((VxC_b1 + (-1.0 * 10.0 * Vp_{hi_b1} * \sin(p_{hi_b1}))) \\
&\quad + (-1.0 * (VxC_b2 + (10.0 * Vp_{hi_b2} * \sin(p_{hi_b2})))))) \\
&\quad /0.001))))/10.0) \\
Vx\dot{C}_b2 &= (((((x_j2/0.001) + (((VxC_b1 + (-1.0 * 10.0 * Vp_{hi_b1} * \sin(p_{hi_b1}))) \\
&\quad + (-1.0 * (VxC_b2 + (10.0 * Vp_{hi_b2} * \sin(p_{hi_b2})))))))/0.001)) \\
&\quad + (-1.0 * 0.0))/10.0) \\
Vy\dot{C}_b0 &= (((((y_j0/0.001) + ((0.0 + (-1.0 * (VyC_b0 + (-1.0 * 10.0 * Vp_{hi_b0} \\
&\quad * \cos(p_{hi_b0})))))/0.001)) + (-1.0 * ((y_j1/0.001) + (((VyC_b0 \\
&\quad + (10.0 * Vp_{hi_b0} * \cos(p_{hi_b0}))) + (-1.0 * (VyC_b1 + (-1.0 * 10.0 \\
&\quad * Vp_{hi_b1} * \cos(p_{hi_b1})))))))/0.001))) + (-1.0 * 10.0 * 9.81))/10.0) \\
Vy\dot{C}_b1 &= (((((y_j1/0.001) + (((VyC_b0 + (10.0 * Vp_{hi_b0} * \cos(p_{hi_b0}))) \\
&\quad + (-1.0 * (VyC_b1 + (-1.0 * 10.0 * Vp_{hi_b1} * \cos(p_{hi_b1})))))))/0.001)) \\
&\quad + (-1.0 * ((y_j2/0.001) + (((VyC_b1 + (10.0 * Vp_{hi_b1} * \cos(p_{hi_b1}))) \\
&\quad + (-1.0 * (VyC_b2 + (-1.0 * 10.0 * Vp_{hi_b2} * \cos(p_{hi_b2})))))))/0.001))) \\
&\quad + (-1.0 * 10.0 * 9.81))/10.0) \\
Vy\dot{C}_b2 &= (((((y_j2/0.001) + (((VyC_b1 + (10.0 * Vp_{hi_b1} * \cos(p_{hi_b1}))) \\
&\quad + (-1.0 * (VyC_b2 + (-1.0 * 10.0 * Vp_{hi_b2} * \cos(p_{hi_b2})))))))/0.001)) \\
&\quad + (-1.0 * 0.0) + (-1.0 * 10.0 * 9.81))/10.0) \\
p_{hi_b0} &= Vp_{hi_b0} \\
p_{hi_b1} &= Vp_{hi_b1} \\
p_{hi_b2} &= Vp_{hi_b2} \\
x\dot{C}_b0 &= VxC_b0 \\
x\dot{C}_b1 &= VxC_b1 \\
x\dot{C}_b2 &= VxC_b2
\end{aligned}$$

$$\begin{aligned}
\dot{x}_j0 &= (0.0 + (-1.0 * (VxC_b0 + (10.0 * Vp_{hi_b0} * \sin(p_{hi_b0})))))) \\
\dot{x}_j1 &= ((VxC_b0 + (-1.0 * 10.0 * Vp_{hi_b0} * \sin(p_{hi_b0}))) \\
&\quad + (-1.0 * (VxC_b1 + (10.0 * Vp_{hi_b1} * \sin(p_{hi_b1})))))) \\
\dot{x}_j2 &= ((VxC_b1 + (-1.0 * 10.0 * Vp_{hi_b1} * \sin(p_{hi_b1}))) \\
&\quad + (-1.0 * (VxC_b2 + (10.0 * Vp_{hi_b2} * \sin(p_{hi_b2})))))) \\
y\dot{C}_b0 &= VyC_b0 \\
y\dot{C}_b1 &= VyC_b1 \\
y\dot{C}_b2 &= VyC_b2 \\
y_j0 &= (0.0 + (-1.0 * (VyC_b0 + (-1.0 * 10.0 * Vp_{hi_b0} * \cos(p_{hi_b0})))))) \\
y_j1 &= ((VyC_b0 + (10.0 * Vp_{hi_b0} * \cos(p_{hi_b0}))) \\
&\quad + (-1.0 * (VyC_b1 + (-1.0 * 10.0 * Vp_{hi_b1} * \cos(p_{hi_b1})))))) \\
y_j2 &= ((VyC_b1 + (10.0 * Vp_{hi_b1} * \cos(p_{hi_b1}))) \\
&\quad + (-1.0 * (VyC_b2 + (-1.0 * 10.0 * Vp_{hi_b2} * \cos(p_{hi_b2}))))))
\end{aligned}$$

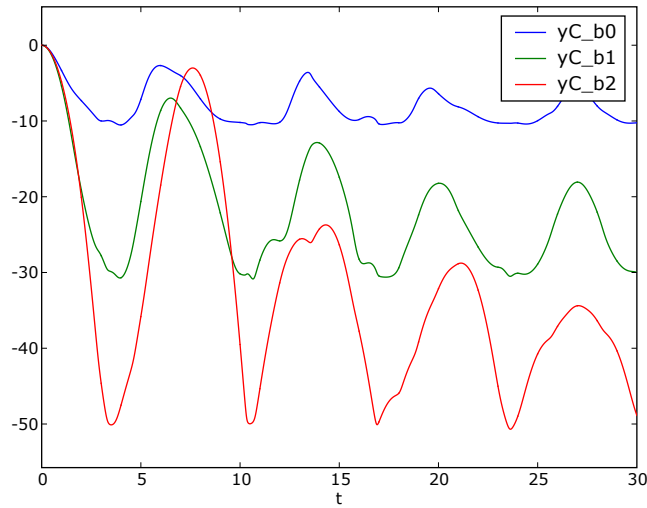


Figure 5.14: Response of the triple pendulum model figure 5.17 released from an initial horizontal arrangement – vertical displacement of pendulum body centres

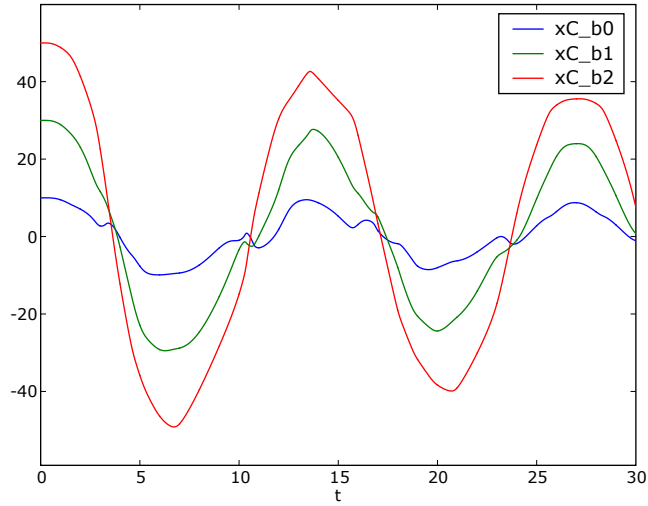


Figure 5.15: Response of the triple pendulum model figure 5.17 released from an initial horizontal arrangement – horizontal displacement of pendulum body centres

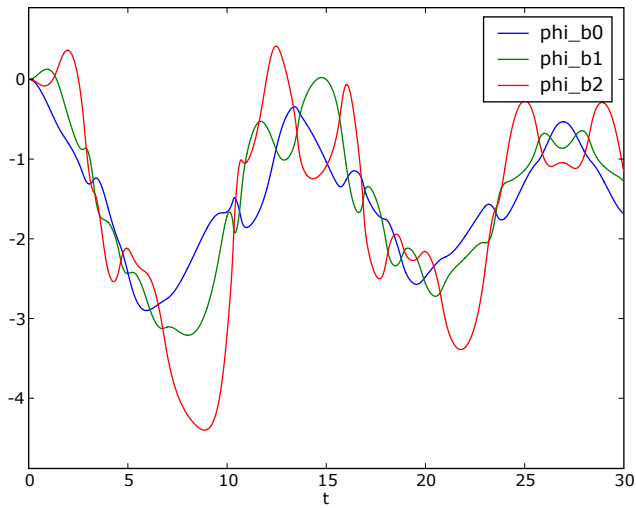


Figure 5.16: Response of the triple pendulum model figure 5.17 released from an initial horizontal position – angular position of pendulum bodies

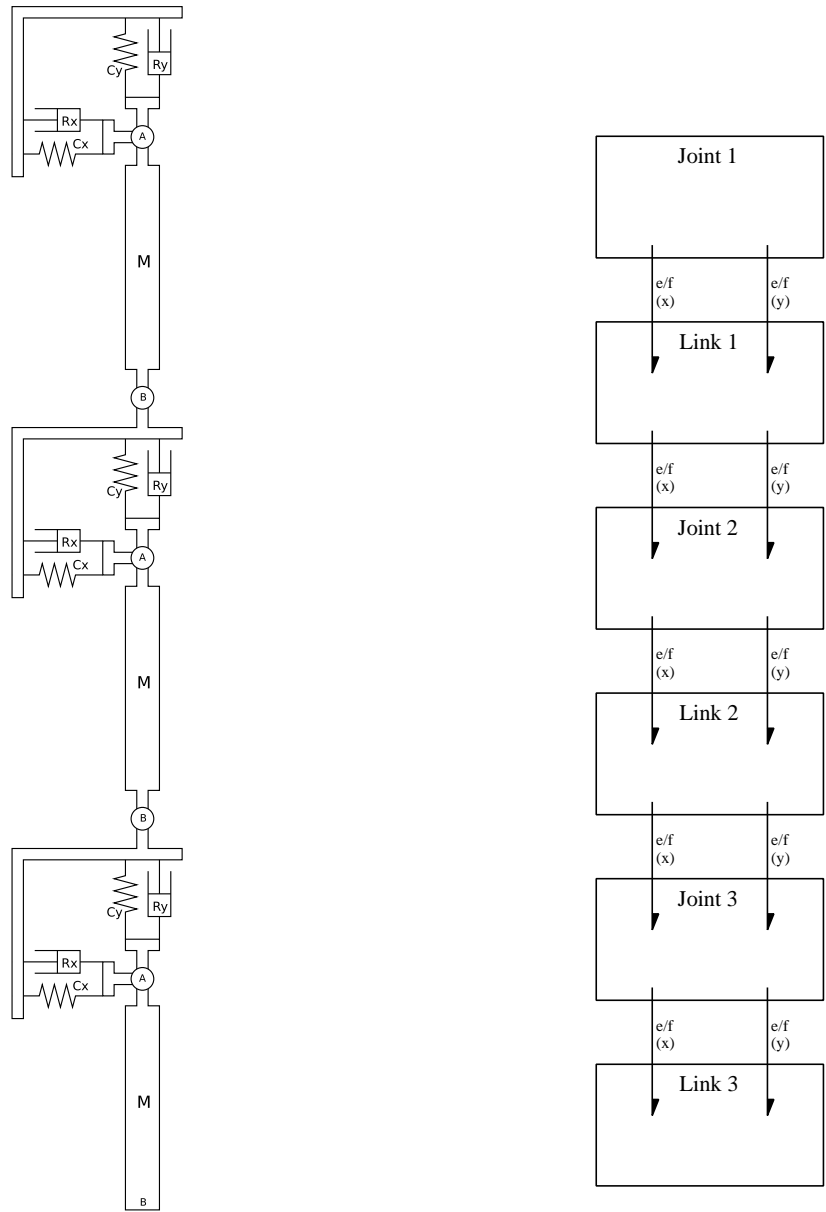


Figure 5.17: Schematic diagram and compound element bond graph model of a triple planar pendulum

### 5.1.6 Linked elastic collisions

Figure 5.18 shows a rigid bar with damped-elastic bumpers at each end in schematic form. The system consists of a rigid link element,  $L$ , as used above in 5.1.4 representing the bar; two switched capacitor elements,  $CCa$ ,  $CCb$  representing elastic collision between the tips and a rigid horizontal surface; unswitched vertical damping at each end and on rotation (not shown in the schematic) representing drag on the bar as it tumbles. An equivalent bond graph model for this system is shown in figure 5.19.

Figure 5.20 shows the bar-with-bumpers modelled in figure 5.19 released above a rigid horizontal surface.  $int_a2$  is the vertical displacement of the centre of mass above the horizontal surface.  $int_a3$  is the angular displacement of the bar from vertical.  $q21$  and  $q24$  are the vertical displacements of the two end points above the surface. The bar is released level to the horizontal surface so  $int_a2$ ,  $q21$ , and  $q24$  (height of the centre, left tip, and right tip) are coincident and  $int_a3$  (angular displacement) is steady at  $\pi/2$  before the bar tips strike the ground at height 1, equal to the unsprung length,  $q$ , of the contact compliances. The contact compliance on one side has twice the capacitance (half the stiffness) of the other so it compresses further on impact then bounces higher as the bar rotates in the air.

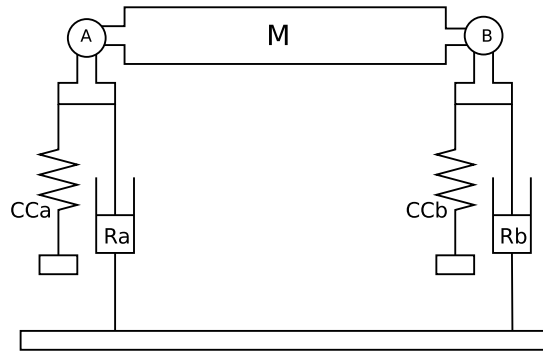


Figure 5.18: Schematic diagram of a rigid bar with contact compliance and viscous drag at each end



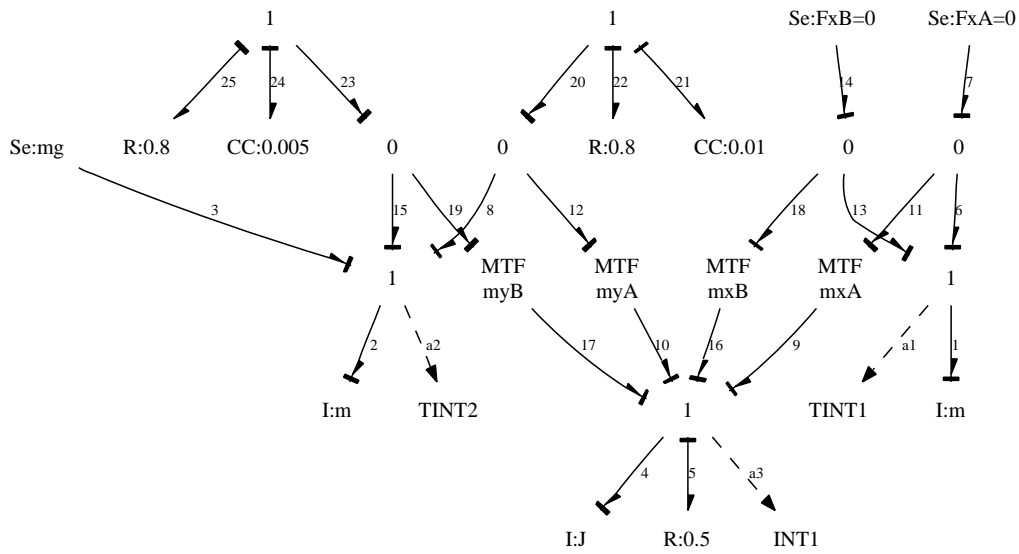


Figure 5.19: Bond graph model of the system from figure 5.18

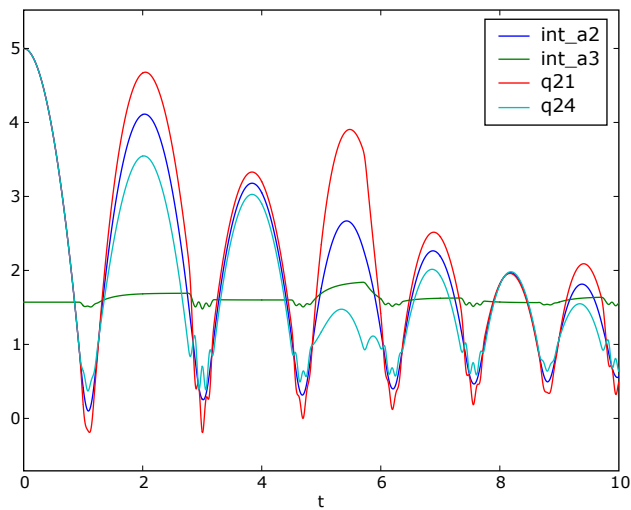


Figure 5.20: Response of the bar-with-bumpers model from figure 5.19 released above a rigid horizontal surface

## 5.2 Symbolic regression of a static function

As an early proof of the genetic programming software a static function approximation problem was solved by symbolic regression and some observations of the system performance were made. The genetic programming system at that time was mono-typed with a special division operator to protect against divide-by-zero errors. Four aspects of the genetic programming system performance were measured.

1. Population fitness (summarized as minimum, mean, and maximum program fitness each generation)
2. Population diversity (the number of structurally unique programs in each generation)
3. Program age (the number of generations that each program had been present in the population)
4. Program size (summarized as minimum, mean, and maximum number of nodes in a program)

The particular symbolic regression problem chosen consisted of 100 samples of the function  $x^2 + 2$  at integer values of  $x$  between  $-50$  and  $+50$ . The fitness evaluation consisted of finding the total absolute error between the candidate and the target samples and normalizing that value onto the interval  $(0, 1]$  where a perfect candidate with zero error was awarded a fitness of 1. Exceptionally, where the candidate was found to perform division by zero or to return the special value NaN (“not a number”) a fitness value of zero was awarded. Over 50 genetic programming runs were performed, all with the nominal crossover probability set at 0.8, the mutation probability set at 0.2, and the number of generations limited to 500 but with a variety of population sizes.

### 5.2.1 Population dynamics

Plots of the four measured aspects of genetic programming run performance are shown for two particularly interesting runs (run “A” and run “B”) in figures 5.21 to 5.28. For both runs the population size was fixed at 500 individuals.

The fitness plot for run “A” shows a very typical slow initial progress and low initial fitness. Given that the number of programs that could be randomly generated even within the size limits placed on members of the initial random population, it would be very rare indeed for a sizable number or even one individual to have a fitness value that is not very low. The initial best, mean and worst fitness values in both runs are all extremely low. The other characteristic of run “A” fitness that is very typical is the long initial stretch of slow progress. Runs “A” and “B” similarly take tens of generations for the best of generation fitness to exceed 0.1.

The mean fitness trend in run “A” can be seen on close inspection to make its first break from the “floor” near generation 20. At this same generation the beginning of a sharp decline in population diversity can be seen. From this, one can conclude that around generation 20 a new individual appeared with significantly greater than average fitness and quickly came to dominate the population. In run “A” the mean program size begins to decrease at generation 20 and reaches a minimum around generation 30 corresponding to the population diversity minimum for this run. From this, one can infer that the newly dominant individual was of less than average size when introduced. By generation 30 there were less than 40 unique program structures in a population of 500. As there came to be more and more individuals of the dominant (and smaller than average ) type, the mean program size necessarily decreased as well. When the population diversity later rose, many of these newly introduced program structures must have been larger than the recently dominant one since the mean program size also increased.

Figure 5.24 (“Program age in run A”) has been marked to indicate the generation in which the oldest ever individual for that run was created, near generation 25. The oldest (maximum age), most fit (maximum fitness), and largest (maximum size) individuals are of course not necessarily or even likely the same program.

Run “B” experiences an even more severe loss of diversity falling to just 20 unique program structures in its 100th generation. The sharp decrease in population diversity between generations 50 and 100 corresponds exactly to the sharp increase in mean program fitness over the same interval. At the end of that time, the mean program fitness has risen to almost match that of the most fit program where it stays for over 300 generations. The sudden rise in mean fitness and decline in diversity is a dramatic example of one extremely successful design overwhelming all others. The entirely flat fitness trend for the ensuing 300 generations is an equally dramatic example of the

adverse effects of low population diversity on genetic programming search progress. In effect the inherent parallelism of the method has been reduced. Although this is not the same as reducing the population size since multiple concurrent crossovers with identical parents are still likely to produce varied offspring due to the random choice of crossover points, it has a similar effect. During this 300 generation “slump” the mean program age rises steadily with an almost perfectly linear slope of just less than unity indicating that there is some, but very little population turnover. The algorithm is exploiting one program structure and small variations in the neighbourhood that differs only by leaf node values. There is very little exploration of new program structures happening.

It was often observed that crises in one of the measured aspects of genetic programming were accompanied by sudden changes in other aspects. In general not much can be inferred about one aspect by observing the others with the possible exceptions of inferring the arrival of an individual with dramatically higher than average fitness from sudden decreases in population diversity or changes in mean program size. In both cases it is the dominance of the new program structure in the population that is being observed and this inference would not be possible if the genetic programming system took active measures to ensure population diversity.

Note also that although the peak program size is sometimes recorded in excess of the declared ceiling of 100 nodes, this is simple an artifact of the order in which fitness evaluation and statistics logging were performed each generation. Individuals of excessive size were recorded at the point of fitness evaluation were barred from participating in crossover, mutation, or reproduction due to their zero fitness score and so could not enter the population of the following generation.

### **5.2.2 Algebraic reductions**

The non-parsimonious tendencies of typical genetic programming output is well known and often commented on. A successful result, however, is always functionally equivalent in some testable ways to the target or goal. Specifically, it excels at the particular fitness test in use. Sometimes there are structural equivalences that can be exploited to produce a simplified or reduced representation which is nonetheless entirely equivalent to the original in a functional sense. In symbolic regression the exploitable structural equivalences are simply the rules of algebra. Even a very small set of alge-

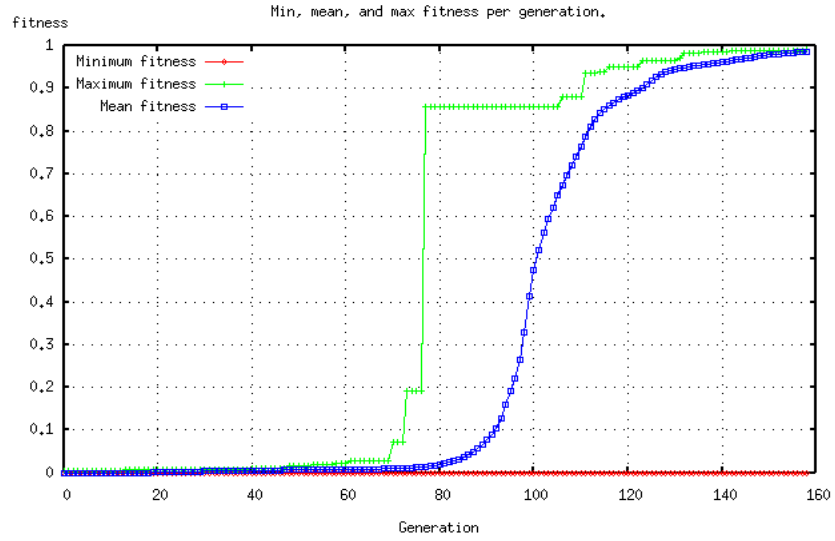


Figure 5.21: Fitness in run A – progress slow initially and while size limiting in effect

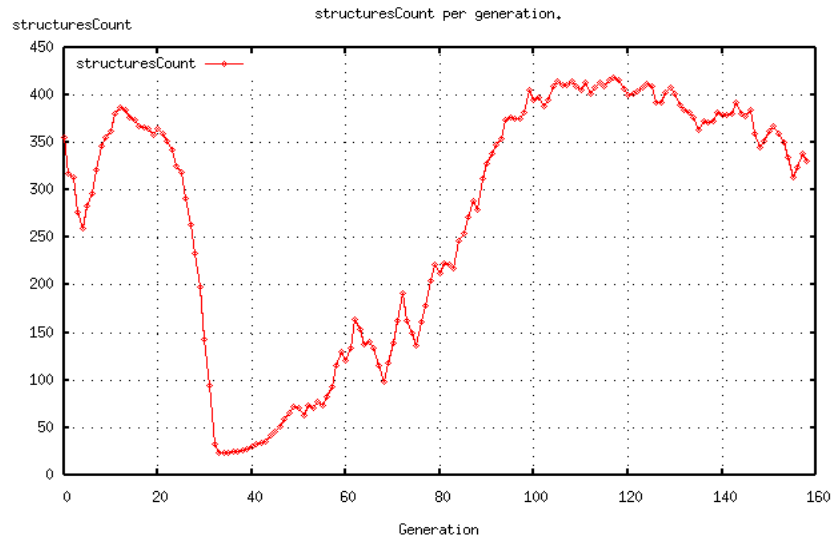


Figure 5.22: Diversity in run A – early crisis is reflected in size and age trends

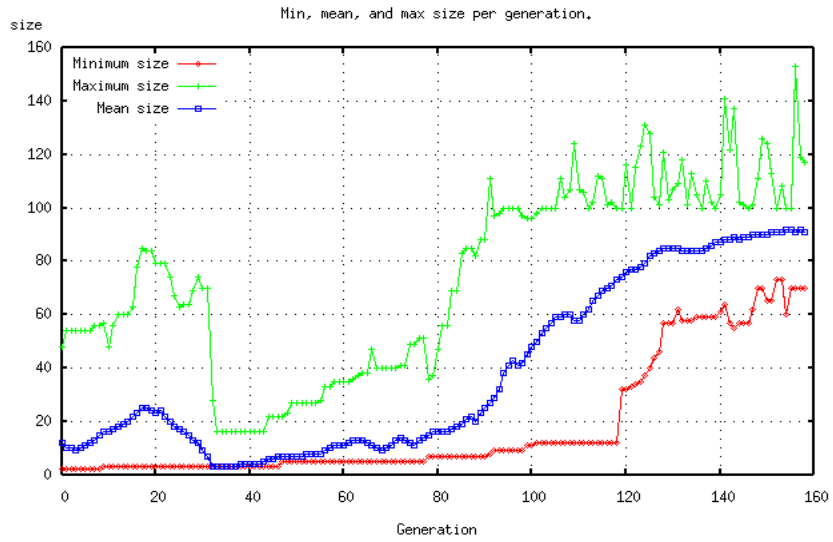


Figure 5.23: Program size in run A – mean size approaches ceiling (100 nodes)

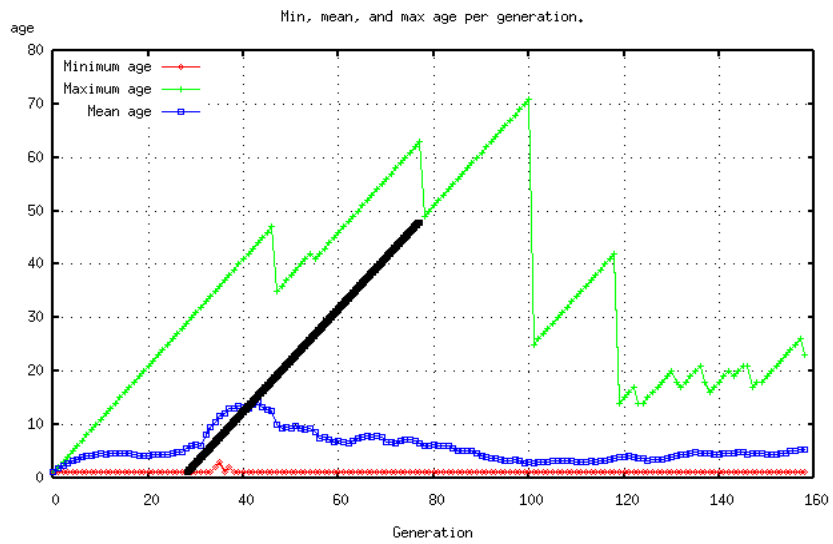


Figure 5.24: Program age in run A – bold line extrapolates birth of oldest program

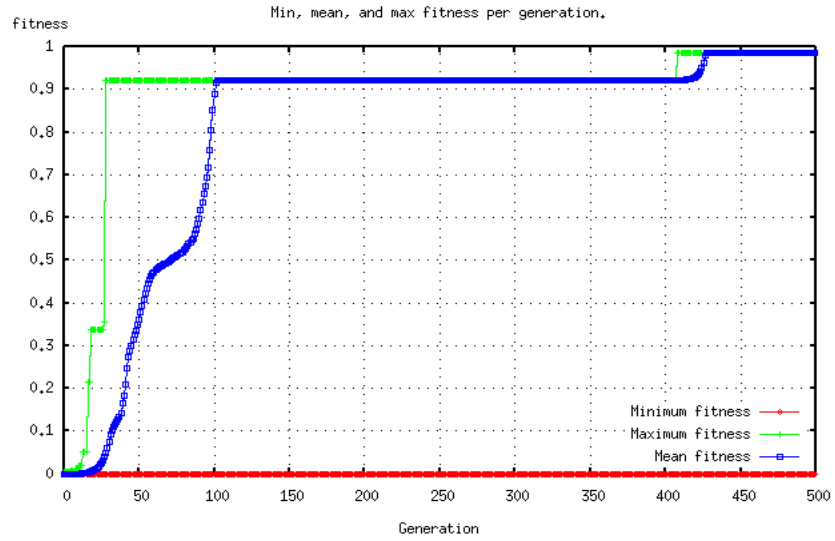


Figure 5.25: Fitness of run B – progress stalled by population convergence

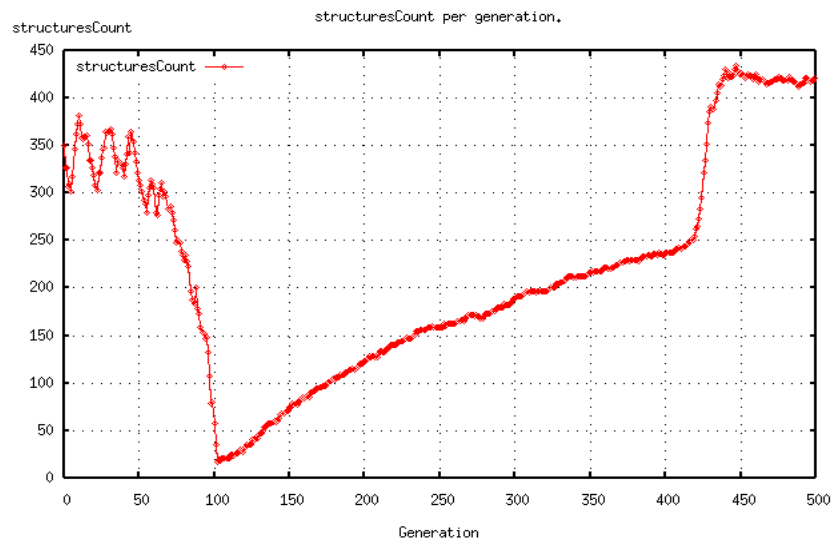


Figure 5.26: Diversity of run B – early loss of diversity causing stalled progress

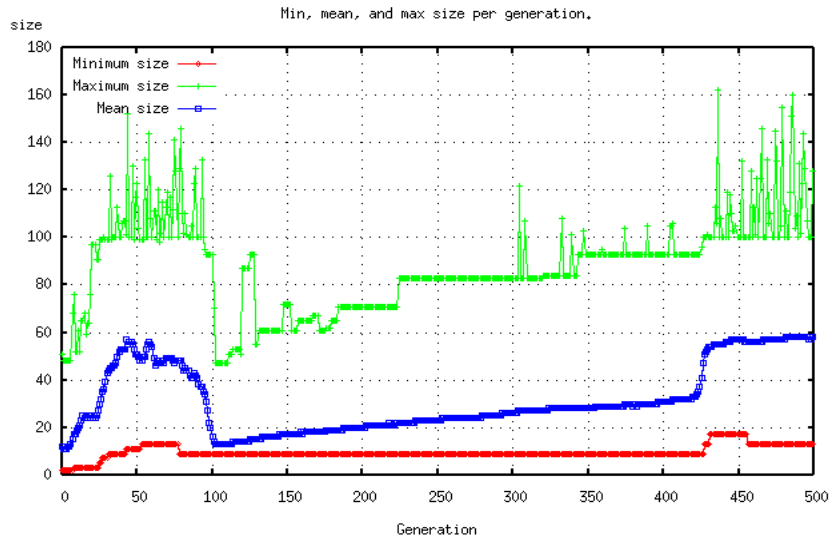


Figure 5.27: Program size in run B – early domination by smaller than average schema

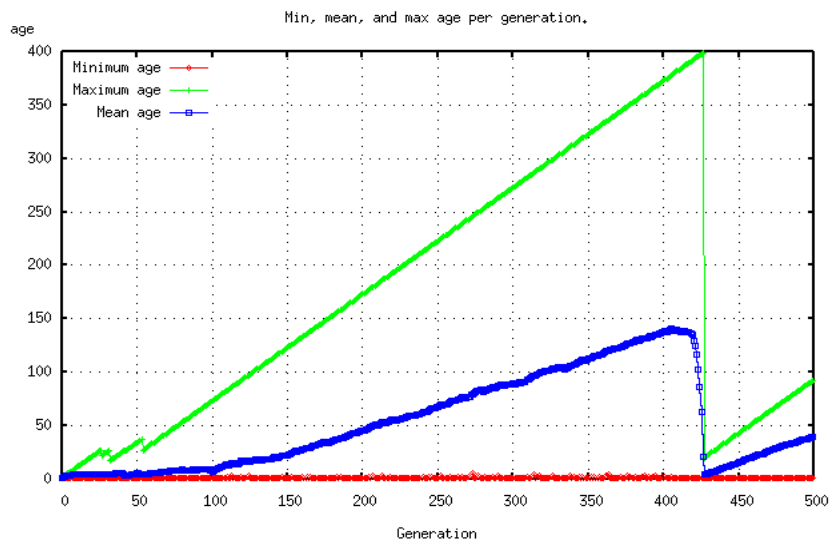


Figure 5.28: Program age in run B – low population turnover mid-run between crisis



braic reduction rules capable of producing reduced results which are in many cases vastly more legible. The algebraic reduction rules implemented at the time these experiments were performed are listed below in s-expression form (LISP prefix notation).

(- x x) replaced by (0)  
(- x 0) replaced by (x)  
(+ x 0) replaced by (x)  
(+ 0 x) replaced by (x)  
(/ x x) replaced by (1)  
(/ 0 x) replaced by (0)      for all x.

All of these reductions operate on a single function node and its children. Only two properties of the children are checked: first is either child a constant value node and equal to zero or one, and second, are the children equal to each other.

One other type of simplification was performed as well. Any subtree containing only function nodes and constant value terminal nodes was replaced by a single constant value node. The value of this new node is found by evaluating the subtree at an arbitrary value of the independent variable. This “trick” alone is responsible for many reductions that are more dramatic than those produced by all the other rules listed above.

It is not at all clear by inspecting the raw expression in figure 5.29, a best approximation that successfully terminated one run after 41 generations, that it approximates  $x^2 + 2$ . The raw expression is simply too large, and contains misleading operations (trigonometry and division). However, the reduced expression in figure 5.30 is very obviously a good approximation to the target expression. All of these gains in clarity are had by recognizing that the entire right branch of the root *Add* node is a constant expression containing no variables. It evaluates to very nearly 2.

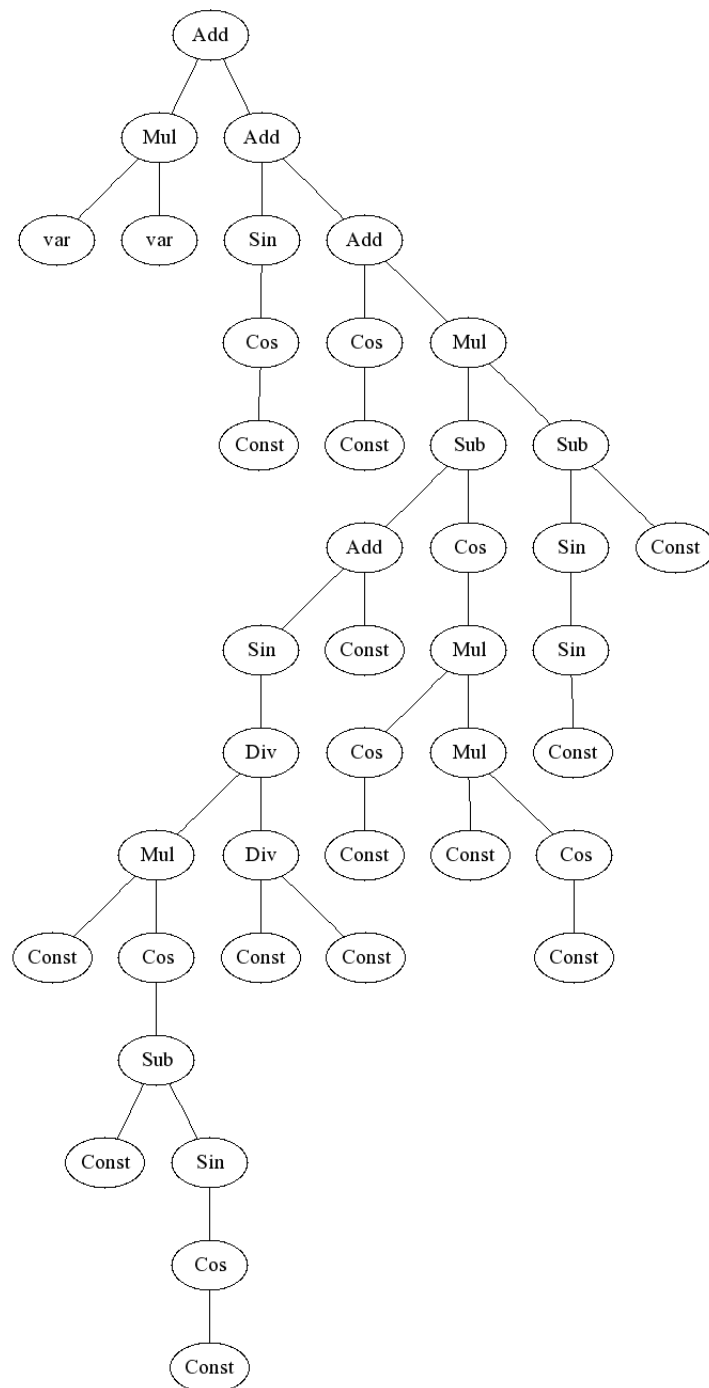


Figure 5.29: A best approximation of  $x^2 + 2.0$  after 41 generations

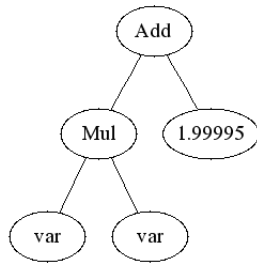


Figure 5.30: The reduced expression

### 5.3 Exploring genetic neighbourhoods by perturbation of an ideal

The symbolic regression for dynamic systems genetic programming grammar presented in section 4.5.3 is here evaluated by sampling the neighbourhood of a manually constructed “ideal” program. The ideal is constructed so as to have exactly the same differential state update equations and one of the output equations of the model described in Section 5.1.1. The neighbourhood of this ideal is sampled by repeatedly applying the mutation operator to fresh copies of the ideal. This generates a large number of new genetic programs that are “near” to the ideal (they differ by only the subtree rooted at the mutation point). All of the generated programs are fitness tested against the response of the ideal to a uniform random noise input, as shown in figure 5.4.

To quantify nearness, a metric for pairwise distance between tree structures is defined. In this metric, distance is equal to the size (number of nodes) of the subtree rooted at the first node that does not match in a breadth-first traversal of both trees. Since the corresponding subtrees in the original and the mutant differ, the size of the largest subtree is used. This metric is symmetric (the distance between two trees is independent of the order they are given in). Under this metric identical trees have a distance of 0, trees that differ by only one leaf node have a distance of 1, and so on. This is a form of “edit distance”. It is the number of nodes that would have to be added, deleted, or replaced to transform one program into the other.

Figure 5.32 shows the “hand crafted” ideal symbolic regression tree. Table 5.3 shows the mean fitness of programs by distance and the total number of programs found at that distance within the sample. Figure 5.31 plots mean fitness versus tree-distance from the ideal. Distances at which fewer than 10 programs were found are omitted.

The point of these results is that the representation in question produces a fitness landscape that rises on approaching the ideal. If edit distance as defined above is taken as an estimate of the amount of work the genetic algorithm will have to do to reach the ideal starting from the mutant, then these results suggest the given representation will be well-behaved in genetic programming experiments. If instead the fitness landscape had fallen off approaching the ideal, the representation would be called “deceptive”. The heuristic by which genetic algorithms operate (“good” solutions have good parts and “goodness” is measured by the fitness test) would be misled in

Distance	Mean fitness	Count
1.0	0.000469	2212.0
3.0	0.000282	503.0
5.0	0.000346	66.0
7.0	0.000363	36.0
9.0	0.000231	25.0
11.0	0.000164	16.0
13.0	0.000143	10.0
15.0	0.000071	10.0
17.0	0.000102	9.0
19.0	0.000221	1.0
23.0	0.0	2.0
25.0	0.0	2.0
27.0	0.0	1.0
29.0	0.000286	5.0
35.0	0.0	1.0
51.0	0.000715	1.0

Table 5.1: A sampling of the neighbourhood of a “hand crafted” dynamic system symbolic regression tree that outputs equations 5.1, 5.2, and 5.7

that case. Another pathologically difficult problem for genetic programming would be one in which the fitness landscape is entirely flat and low at all points not exactly coincident with the ideal. In such cases fitness-guided search such as genetic programming can do no better than a random search.

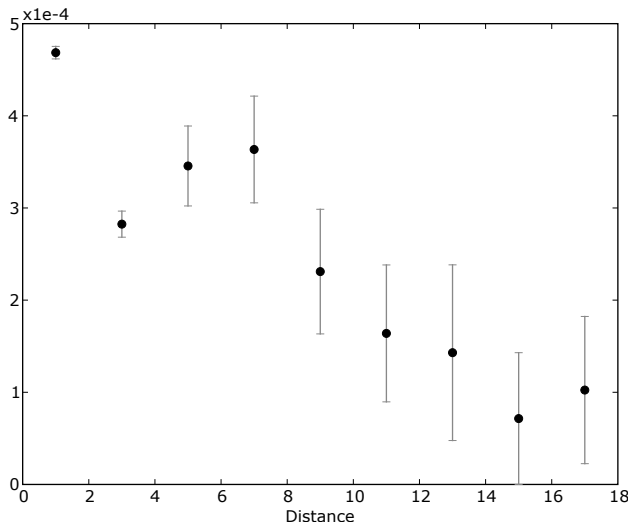


Figure 5.31: Mean fitness versus tree-distance from the ideal. Bars show standard error. Series truncated at first point with only one sample.

## 5.4 Discussion

### 5.4.1 On bond graphs

Bond graphs are a good choice of modelling language for several reasons. In short, bond graphs encode structure in physically meaningful and computationally useful ways. Bond graphs are a modular modelling language in which a subsystem, once modelled, can be reused elsewhere. Every element in the basic bond graph modelling language models a single physical phenomena. These can be interconnected freely but the language provides qualitative insight into the physical consequences of some modelling decisions. These insights come from the available model reductions (a sort of algebra for bond graphs) and from the augmented bond graph in which a causal direction

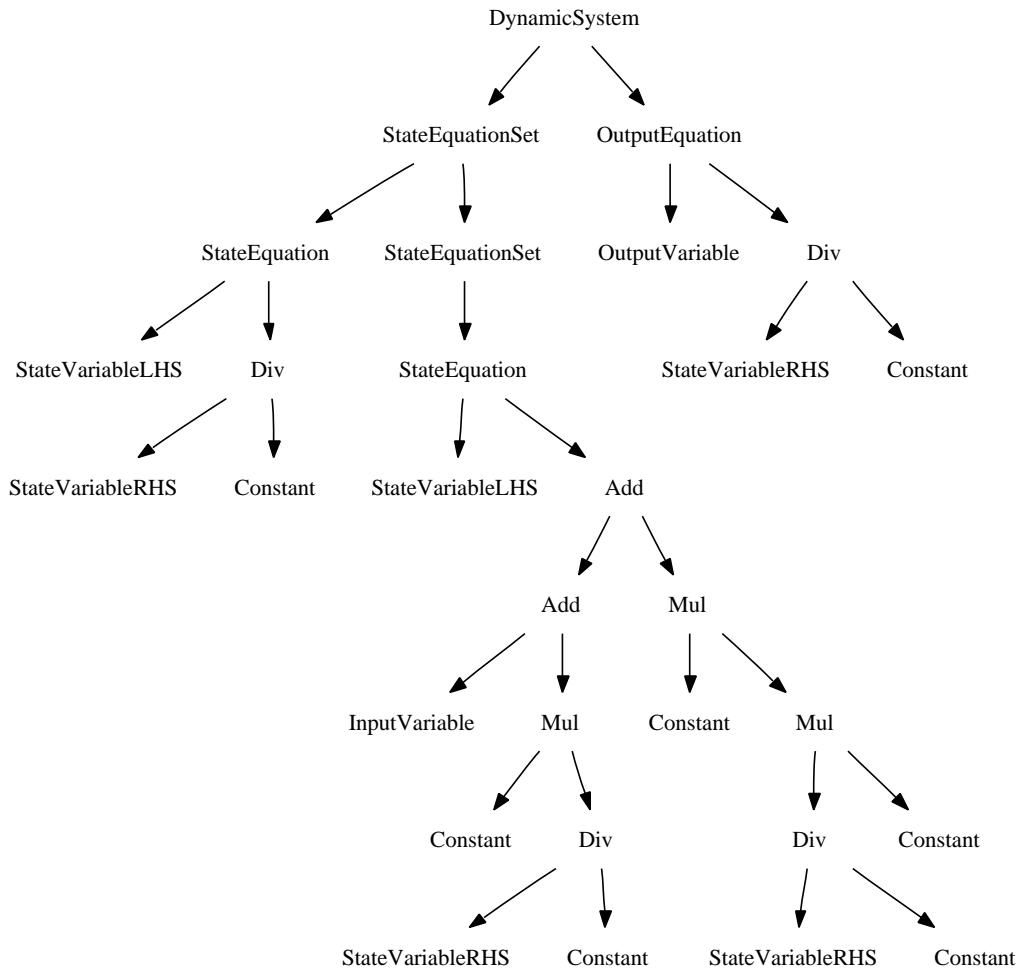


Figure 5.32: The hand crafted “ideal” symbolic regression tree

has been assigned to every bond. These qualitative insights may be helpful during an interactive or iterative grey-box system identification exercise.

Model reductions generate alternate, behaviourally equivalent but structurally simplified models. This may highlight areas where multiple parts in a model have been used to represent one physical part in the system. Model reductions are also computationally valuable when using bond graphs with genetic programming since the reduced model will have fewer equations. The algebraic manipulations required between generating the bond graph model and solving the equations numerically are more complex than the bond graph reductions so it is best to simplify as early as possible. A smaller set of equations will also be quicker to evaluate numerically (provided it is no more stiff than the original system) simply because the right-hand side of the state equations will be shorter and faster to evaluate at each point along the integration.

Causality assignment produces an augmented bond graph from which it is simple to tell whether a state-space formulation of the model equations will be possible. If not, then the model is over specified and has fewer state variables than discrete energy storing elements. Some storage elements are dependent on others and the reason (inertias with common flow, capacitors with common effort, etc.) can often be linked to physically intuitive notions about the system under test and the way it is abstracted into the model.

As a corollary to the fact that bond graph models encode structure in physically meaningful ways, bond graphs present a smaller search space for genetic algorithms (or any other search technique) to explore than do some other representations. There is always a physical interpretation for any bond graph in a choice of domain (electrical, solid body mechanical, hydraulic). The interpretation may not be realizable in size or scale, but it exists. There are signal flow graphs and sets of differential equations, to give two examples, for which no physical interpretation exists in a given domain. A signal-flow graph and a set of differential equations can each be extracted from any bond graph model (the former simply by splitting each bond into an effort signal and a flow signal) but the reverse is not true. In both transformations some information about the physical structure is lost.

Finally, bond graphs are an attractive modelling language because they scale smoothly from simple linear systems, where additional automated transformations are possible and for which a great body of analysis exists, to complex nonlinear and even non-analytic (numerical or computational only) models without losing any of the above advantages. A model can start out



small, simple, and linear but grow large, complex, and non-linear over the course of some experiments without ever having to stop and recode the model in a new language.

Bond graphs have distinct disadvantages, however, all of which come down to being the wrong sort of abstraction for some applications. As mentioned in the introduction to this thesis, there are no perfect models. All models are abstractions that trade off fidelity for simplicity. The art of modelling is to decide or discover which aspects of the system are important to the exercise and so must be included in a model, and which aspects are not important and can be dropped for simplicity. A modelling language that presents abstractions at just the right level of detail can help this task along. Bond graphs present abstractions (elements and bond) with more physical detail than a signal-flow graph or a set of differential equations but with less detail than, for example a free body diagram for solid-body mechanics applications.

A modelling language which is too abstract will be inconveniently distracting both to human experimenters and to search algorithms such as genetic programming. Planar and 3-D solid body mechanics applications are one area where bond graphs can be faulted in this regard.

Fundamental to the bond graph modelling paradigm is the requirement for power bond variables to be given in compatible units. For mechanics applications they must also either be given with respect to one global frame of reference or complicated subgraphs must be introduced for coordinate transformation purposes. Although libraries of such joint transformations have been published, the result of this technique is that parts of a model do not represent power transfer within the system but rather happen to produce the correct equations to transform coordinates between adjacent parts of the model. These inconvenient work-arounds for limitations of the modelling language can be distracting. All models presented in this thesis use a single, global frame of reference which precludes, for example, modelling body-aligned compliance in the triple-pendulum pin joints.

The highly abstract representations presented by the bond graph modelling language can also be “distracting” to a search algorithm such as genetic programming and lower its efficiency by allowing many models that are valid within the language but physically senseless with respect to a particular application. Consider the rigid link with connection points at either end used in the single and triple pendulum models and in the linked elastic collisions model. If this subgraph were presented to the genetic programming system

as part of a larger model, there is every chance that the algorithm would at some point generate models containing internal modifications that destroyed the symmetry of that subgraph in a way physically senseless in terms of rigid bodies. The rate at which such valid-but-unhelpful models are generated will be in direct proportion to the ratio of the size of the rigid body sub-model to the whole model within the genetic program.

Encapsulating the rigid-link sub-model as was done for the triple pendulum model alleviates these problems at the cost of effectively inventing a new and very specialized modelling language. In order to be reusable, the rigid link sub-model includes three state variables. This leads directly to the need for “suspended” pin joints in the pendulum models to avoid differential causality and the need for a differential-algebraic solver. A modelling language wholly dedicated to rigid body mechanical applications would have the context on hand to be able to insert a single degree of freedom model where appropriate. It would also be much less abstract and therefore also much less widely applicable (for example to electrical or hydraulic applications) than are bond graphs.

### 5.4.2 On genetic programming

Genetic programming has several desirable properties as a search algorithm to be used for system identification. Overall, genetic programming requires very little problem-specific information in order to begin but it also provides two mechanisms by which any additional information about the problem can be taken advantage of. The basic requirements for the application of genetic programming to system identification are a generative grammar for models and a fitness test. Additional problem-specific insight can be provided either within the fitness test (to reject undesirable models) or in the form of a more restrictive grammar (to reduce the size of the search space, which is preferable). Some modelling languages make it easier to specify restrictive genetic grammars than do others because of the way each encodes physical structure.

The use of indirection through a genetic grammar means that the exact structure of need not be pre-specified. This allows genetic programming to perform simultaneous structure and parameter identification. It also allows the structure to grow or shrink in the course of identification. Many other modelling and identification schemes require that the model size be specified ahead of time (for example, the number of layers and nodes per layer must

be chosen for a neural network).

In contrast to some other bio-mimetic search algorithms such as swarm and ant colony algorithms, and the venerable simplex algorithm, genetic programming solutions need not exist within a metric space. Some concept of distance between genetic programs may be useful in evaluating a particular genetic representation or genetic programming implementation, but one is not required.

A grey-box system identification is easily implemented by seeding the initial genetic population with instances of a model encoding any prior knowledge or by providing this model as input to each genetic program when it is evaluated by the fitness function. In fact the algorithm can be interrupted and subjected to this kind of external “help” at any point. Genetic programming is extensible in many ways. It is easily hybridized with other search techniques such as parametric hill climbing or simulated annealing by introducing new operators that are applied either occasionally or before every fitness evaluation and by varying the choice of operators over time. The latter choice (applying some new optimizing operator before each fitness evaluation) is a sort of search within a search algorithm. A simulated annealing with genetic programming hybrid is made by varying the likelihood of choosing each genetic operator over the course of a run according to some schedule.

Genetic programming is easily parallelizable for distribution across a local or wide area network in one of several ways. One particular scheme (island model) has been attributed in the literature with performance gains that are super-linear in the degree of parallelism. It is fortunate that these parallelization options are available since genetic algorithms are notoriously computationally expensive.

Computational expense is chief among the detractions of genetic programming but not the only one. Because they use so little problem-specific knowledge, genetic algorithms may be needlessly inefficient when such knowledge is available, unless it can be used to reduce the search space by refining the structural grammar. This has, in effect, been stated already in this section but it is worth repeating while underlining that it may not always be a simple matter to encode knowledge of the system under test in a suitably restrictive genetic grammar. Encoding this knowledge in the fitness test is easier (simply assign arbitrarily low fitness to models that do not meet expectations) but much less efficient since these rejected genetic programs are produced, stored in the population, and evaluated at some expense in processing time

and memory.

Finally, some problems are deceptively hard, and genetic algorithms offer no help in identifying them. Genetic algorithms can be envisioned as exploring a “fitness landscape” and seeking a global maximum thereon. The fitness landscape of deceptive problems provide no, or negative, feedback about proximity to the global optimum. They may also use a genetic representation in which the crossover and mutation operators are often destructive, cutting through and breaking apart useful sub-models. The argument that bond graphs are too abstract for some rigid body mechanics applications is an example. Since genetic algorithms, including tree-structured genetic programming, offer no special feedback or guidance when a hard problem is encountered, and since empirical studies of genetic programming are so expensive in computer time, it can be difficult to predict in which situations genetic programming will bear fruit.

# Chapter 6

## Summary and conclusions

Black-box identification of general non-linear dynamic systems with any number of inputs and outputs is a hard problem. It is the most general and encompassing class of problems in system identification. The project behind this thesis set out, over-ambitiously without question, to create a software tool that would address this most general class of identification problems in a very flexible way. The aim was to create a system that would allow an experimenter to insert as much or as little information as was available about the system under test before or during an identification, to allow this information to be specified in a way that is natural to the task, and to have the algorithm make good use of it. Additionally, it was desired that the resulting model should be in a form that is “natural” to the experimenter. Questions of efficient identification were deliberately set aside even from the beginning with one exception: all structural identification problems are at heart search problems within a space defined by the modelling language. These spaces can be very large, infinite if the model size is not bounded, so the most important step that can be taken towards an efficient structural identification algorithm is to provide an easy mechanism by which an experimenter can limit the solution space by adding any information they may have about the system.

Bond graphs are a flexible and “natural” modelling language for many applications and genetic programming is a general purpose heuristic search algorithm that is flexible in useful ways. The work leading up to this thesis produced software tools for modelling with bond graphs, a strongly typed genetic programming implementation, and genetic programming grammars that define search spaces for linear and non-linear bond graph models and

symbolic regression of differential equations in state-space form. The bond graph modelling implementation is demonstrated with several simulations of multi-body mechanical systems. The genetic programming implementation is checked by applying it to symbolic regression of a static function. The neighbourhood of a manually constructed “ideal” genetic program modelling a particular dynamic system is examined to show that the fitness landscape is shaped in a way that will drive the genetic algorithm towards the given ideal.

## 6.1 Extensions and future work

### 6.1.1 Additional experiments

It should be very clear that many of the original goals of this project (design of a flexible software system for black- and grey-box identification of non-linear dynamic systems) were not met. It is expected, for example, that bond graphs are a much more efficient representation for use with genetic programming in identifying dynamic system models than is direct symbolic regression of differential equations. This remains untested. A suitable experiment would involve many genetic programming runs with each representation on one or more identification problems. Population size times average number of generations until a solution is found would be a suitable metric for comparison. This metric is the average number of fitness evaluations performed in the course of the search. Since fitness evaluation (involving simulation and comparison of simulation results to stored signals) is by far the most computationally intensive aspect of the search, this is a good and easily calculated measure of the total amount of computer time required.

It would be interesting to repeat the genetic neighbourhood explorations with the genetic programming grammar for bond graphs. For comparison, the ideal should be designed such that its constitutive equations, when extracted and reduced to state space form, are identical to the equation set produced by the symbolic regression ideal.

There should be no great difference in run time for the fitness evaluation of bond graph models and symbolic regression programs that address the same identification problem. If some method were available to automatically create a genetic program of each type that produced identical behaviour (one is suggested in the next section) then this could also be tested with a much

larger (and better randomized) sample size than could reasonably be created by hand.

Parametric identification with bond graphs by symbolic regression of constant valued expressions for element parameters is expected to be faster than black box identification of the entire structure and parameter set. It is not expected to be less computationally intensive than conventional direct methods (such as linear least squares) when they are available. Also, the performance of parametric identification by symbolic regression is independent of the representation chosen for the unchanging structure, be it bond graph, (unchanging) symbolic regression tree, or anything else.

Starting from the accomplished static function approximation results, a good course would be to attempt:

1. Parametric identification of a linear dynamic system by symbolic regression
2. Structural and parametric identification of a linear dynamic system using bond graphs
3. Parametric identification of a non-linear dynamic system using bond graphs.
4. Structural and parametric identification of a non-linear dynamic system using bond graphs.

At each stage the simplest imaginable system should be attempted first: a spring-mass system before a spring-mass-damper system before a multi-body spring-mass-damper system. For a simple non-linear system, the “bouncing ball” example would be interesting or a non-linear (e.g. progressive stiffening) spring could be modelled. In the first case the two parameters of the CC element need to be discovered by the algorithm. In the later case the spring curve, that is the effort-displacement relation for the non-linear C element, would need to be discovered (by symbolic regression even though the remainder of the model is a bond graph).

If they were to find any more use as a modelling tool in their own right, outside of the system identification tasks that were the goal of this project, then the bond graph library would benefit from verification against one or more established modelling tools; either a bond graph modelling application such as 20sim or domain specific tools such as Spice and Adams.

### 6.1.2 Software improvements

To make the tools more widely interoperable, the “.bg” file format for bond graph model storage should be replaced with a file format known to and in use by other software. GraphML, an XML schema for exchanging graphs of all types, may be a good choice.

To facilitate grey-box identification experiments, a routine could be written which produces a minimal program from any phenotypic expression. For example, given an equation set, return a symbolic regression program that would generate it or, given a bond graph model, return a program that would generate it from the empty kernel. This would be a useful tool for bootstrapping experiments from existing models. Presently, such initial genetic programs must be formulated manually.

The software written in the course of producing this thesis contains incomplete implementations of schedulers and network protocols for both island model distributed genetic programming and centralized genetic programming with distributed fitness evaluation. If large problems are tackled, one of these (or equivalent parallel processing capabilities) will be needed.

Differential-algebraic equation solvers (DAE solvers) are available. The numerical simulation code could be extended to use one when reduction to state-space form fails on mixed-causality bond graph models.

Several possible software optimizations were noted during development but not implemented. Fitness evaluations could be cached or memoized such that duplicate genetic programs appearing in the population do not incur duplicate work by the fitness test (easily the most expensive part of the process in system identification applications). Memoization is a classic time/space computing trade-off. Execution time is reduced at the cost of storing past results. To limit the total cost, not all cached results are kept, or they are not kept indefinitely. The implementation can be as simple as a finite table sorted in least-recently-used order and mapping genetic programs (or some short, unique key derived from them) to cached results.

Most tree methods in the software are implemented recursively. Examples of trees include algebraic expressions, equations, and equation sets and genetic programs. Methods on these trees include evaluating an algebraic expression or a genetic program, listing all the variables contained in an algebraic expression, and so on. These methods are implemented such that the root node invokes the like-named method on the nodes below it aggregating their results in some way as they do to the nodes below them. The stan-



standard Python interpreter (called cPython when it is necessary to distinguish it from alternates) is implemented in C and uses the underlying stack to store context frames each time a new method is invoked. While the software is processing very large tree structures, the Python interpreter will use quite a bit of memory, to the detriment of other processes. This is exacerbated by the Python interpreter being not especially aggressive about returning previously allocated but currently unused memory to the operating system. It is always possible to rewrite a recursive algorithm in an iterative form, though the expression may not be as clear or concise. This could be done in many areas of the software if memory use becomes an issue during larger experiments. If the trees become so large as to overflow the stack, the software could also be run on top of Stackless Python [77], a python interpreter that does not use the underlying C stack to store context frames.

# Bibliography

- [1] Netlib repository at UTK and ORNL. <http://www.netlib.org/>.
- [2] Dynasim AB. Dymola, dynamic modeling laboratory. <http://www.dynasim.se/dymola.htm>, 2006.
- [3] Paul-Michael Agapow. Information criteria. [http://www.agapow.net/science/maths/info\\_criteria.html](http://www.agapow.net/science/maths/info_criteria.html), 2003.
- [4] R. R. Allen. Multiport models for the kinematic and dynamic analysis of gear power transmissions. *Transactions of the ASME*, 101:258–267, April 1979.
- [5] B. Bamieh and L. Giarré. Identification of linear parameter varying models. *International Journal of Robust and Nonlinear Control*, 12(9):841–853, 2002.
- [6] J.S. Bendat. *Nonlinear Systems Techniques and Applications*. Wiley Interscience, New York, NY, 2002.
- [7] P. Bendotti and B. Bodenheimer. Linear parameter-varying versus linear time-invariant control design for a pressurized water reactor. *International Journal of Robust and Nonlinear Control*, 9(13):969–995, 1999.
- [8] S. H. Birkett and P. H. Roe. The mathematical foundations of bond graphs-i. algebraic theory. *Journal of the Franklin Institute*, (326):329–250, 1989.
- [9] S. H. Birkett and P. H. Roe. The mathematical foundations of bond graphs-ii. duality. *Journal of the Franklin Institute*, (326):691–708, 1989.

- [10] S. H. Birkett and P. H. Roe. The mathematical foundations of bond graphs-iii. matroid theory. *Journal of the Franklin Institute*, (327):87–108, 1990.
- [11] S. H. Birkett and P. H. Roe. The mathematical foundations of bond graphs-iv. matrix representation and causality. *Journal of the Franklin Institute*, (327):109–128, 1990.
- [12] B. Bodenheimer, P. Bendotti, and M. Kantner. Linear parameter-varying control of a ducted fan engine. *International Journal of Robust and Nonlinear Control*, 6(9-10):1023–1044, 1996.
- [13] S. Boyd and L. O. Chua. Fading memory and the problem of approximating nonlinear operators with volterra series. *IEEE Transactions on Circuits and Systems*, pages 1150–1171, 1985.
- [14] Elizabeth Bradley, Matthew Easley, and Reinhard Stolle. Reasoning about nonlinear system identification. *Artificial Intelligence*, 133(1-2):139–188, 2001.
- [15] P.J. Costa Branco and J.A. Dente. A fuzzy relational identification algorithm and its application to predict the behaviour of a motor drive system. *Fuzzy Sets and Systems*, pages 343–354, 2000.
- [16] F.E. Cellier, M. Otter, and H. Elmqvist. Bond graph modeling of variable structure systems. *Proc. ICBGM'95, 2nd SCS Intl. Conf. on Bond Graph Modeling and Simulation*, pages 49–55, 1995.
- [17] Ali Çinar. Nonlinear time series models for multivariable dynamic processes. *InCINC'94 – The first International Chemometrics InterNet Conference, session on chemometrics in dynamic systems*, 1994.
- [18] Michael Lynn Cramer. A representation for the adaptive generation of simple sequential programs. In *Proceedings of the First International Conference on Genetic Algorithms and their Applications*, pages 183–187, Pittsburgh USA, July 24-26 1985. Carnegie Mellon University.
- [19] Vjekoslav Damić and John Montgomery. *Mechatronics by bond graphs: an object-oriented approach to modelling and simulation*. Springer-Verlag, Berlin Heidelberg, 2003.

- [20] B. Danielson, J. Foster, and D. Frincke. Gabsys: Using genetic algorithms to breed a combustion engine. *IEEE International Conference on Evolutionary Computation*, 1998.
- [21] Darxus. Springgraph. <http://www.chaosreigns.com/code/springgraph/>, 2002.
- [22] K. De Jong. On using genetic algorithms to search program spaces. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 210–216, Cambridge, MA, July 28-31 1987.
- [23] S. Dzeroski and L. Todorovski. Discovering dynamics. *Proc. Tenth International Conference on Machine Learning*, pages 97–103, 1993.
- [24] S. Dzeroski and L. Todorovski. Discovering dynamics: from inductive logic programming to machine discovery. *Journal of Intelligent Information Systems*, 4:89–108, 1993.
- [25] John Ellson et al. Graphviz - graph visualization software. <http://www.graphviz.org/>.
- [26] L. Fogel, A. Owens, and M. Walsh. *Artificial Intelligence Through Simulated Evolution*. John Wiley and Sons, New York, USA, 1966.
- [27] Richard Forsyth. Beagle - a darwinian approach to pattern recognition. *Kybernetes*, 10:159–166, 1981.
- [28] Peter Gawthrop and Lorcan Smith. *Metamodelling: Bond Graphs and Dynamic Systems*. Prentice-Hall Inc., 1996.
- [29] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison Wesley, 1989.
- [30] E. D. Goodman, K. Seo, Z. Fan, J. Hu, and R. C. Rosenberg. Automated design of mechatronic systems: Novel search methods and modular primitives to enable real world applications. *2003 NSF Design, Service and Manufacturing Grantees and Research Conference*, 1994.
- [31] Erik Goodman. Galopps 3.2.4 - the “genetic algorithm optimized for portability and parallelism system”. <http://garage.cse.msu.edu/software/galopps/>, August 25 2002.

- [32] Alan C. Hindmarsh. Odepack, a systematized collection of ode solvers. pages 55–64, North-Holland, Amsterdam, August 1983. MIT Press.
- [33] S. Hofmann, T. Treichl, and D. Schroder. Identification and observation of mechatronic systems including multidimensional nonlinear dynamic functions. *7th International Workshop on Advanced Motion Control*, pages 285–290, 2002.
- [34] John Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.
- [35] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks 2*, pages 359–366, 1989.
- [36] John Hunter. Pylab (matplotlib): a python 2d plotting library. <http://matplotlib.sourceforge.net/>, 2003.
- [37] Eric Jones, Travis Oliphant, Pearu Peterson, et al. Scipy: Open source scientific tools for python. <http://www.scipy.org/>, 2001.
- [38] A. Juditsky, H. Hjalmarsson, A. Benveniste, B. Delyon, L. Ljung, J. Sjöberg, and Q. Zhang. Nonlinear black-box modelling in system identification: mathematical foundations. *Automatica*, 31, 1995.
- [39] Dean C. Karnopp, Donald L. Margolis, and Ronald C. Rosenberg. *System dynamics: a unified approach*. John Wiley & Sons, second edition, 1990.
- [40] Dean C. Karnopp, Donald L. Margolis, and Ronald C. Rosenberg. *System Dynamics: Modeling and Simulation of Mechatronic Systems*. John Wiley & Sons, third edition, 2000.
- [41] Maarten Keijzer. *Scientific Discovery Using Genetic Programming*. PhD thesis, Danish Technical University, Lyngby, Denmark, March 25 2002.
- [42] G.J. Klir. *Architecture of Systems Problem Solving*. Plenum Press, 1985.
- [43] John R. Koza. *Genetic Programming, on the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, Massachusetts, 1992.

- [44] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, Massachusetts, 1994.
- [45] John R. Koza, Forrest H. Bennett III, David Andre, and Martin A. Keane. *Genetic Programming III; Darwinian Invention and Problem Solving*. Morgan Kauffman Publishers, Inc., 340 Pine Street, Sixth Floor, 1999.
- [46] John R. Koza, Martin A. Keane, Matthew J. Streeter, William Mydlowec, Jessen Yu, and Guido Lanza. *Genetic Programming IV; Routine Human-competitive Machine Intelligence*. Kluwer Academic Publishers, 2003.
- [47] V. Leite, R. Araujo, and D. Freitas. A new online identification methodology for flux and parameters estimation of vector controlled induction motors. *IEEE International Electric Machines and Drives Conference*, 1:449–455, 2003.
- [48] Fei Chun Ma and Sok Han Tong. Real time parameters identification of ship dynamic using the extended kalman filter and the second order filter. *Proceedings of 2003 IEEE Conference on Control Applications*, pages 1245–1250, 2003.
- [49] Aahz Maruch. Typing: Strong vs. weak, static vs. dynamic. <http://www.artima.com/weblogs/viewpost.jsp?thread=7590>, July 2003.
- [50] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 1960.
- [51] Pieter J. Mosterman. *Hybrid Dynamic Systems: A hybrid bond graph modeling paradigm and its application in diagnosis*. PhD thesis, Vanderbilt University, May 1997.
- [52] Michael Negnevitsky. *Artificial Intelligence*. Addison Wesley, 2002.
- [53] M. Nikolaou and D. Mantha. Efficient nonlinear modeling using wavelets and related compression techniques. *NSF Workshop on Nonlinear Model Predictive Control*, 1998.
- [54] Michael O’Neill and Conor Ryan. Evolving multi-line compilable c programs. In *Proceedings of the Second European Workshop on Genetic Programming*, pages 83–92, London, UK, 1999. Springer-Verlag.

- [55] George F. Oster and David M. Auslander. The memristor: A new bond graph element. *Transactions of the ASME, Journal of Dynamic Systems, Measurement, and Control*, 94(3):249–252, 1972.
- [56] Chris Paredis. Composable simulation and design, project overview. Technical report, CompSim Project Group, Carnegie Mellon University, 2001.
- [57] Henry M. Paynter. *Analysis and Design of Engineering Systems*. The M.I.T. Press, Cambridge, Massachusetts, 1961.
- [58] Timothy Perkis. Stack-based genetic programming. In *Proceedings of the IEEE World Congress on Computational Intelligence*, 1994.
- [59] Charles L. Phillips and H. Troy Nagle. *Digital Control System Analysis and Design*. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 3rd edition, 1995.
- [60] R. Poli and W. B. Langdon. A new schema theory for genetic programming with one-point crossover and point mutation. In *International Conference on Genetic Programming, GP'97*, pages 278–285, Stanford, 1997. Morgan Kaufmann.
- [61] Cristian Pop, Amir Khajepour, Jan P. Huissoon, and Aftab E. Patla. Bondgraph modeling and model evaluation of human locomotion using experimental data. March 2000.
- [62] PSF. Python copyright notice and license. <http://www.python.org/doc/Copyright.html>, 2005.
- [63] Bill Punch and Douglas Zongker. lil-gp genetic programming system. <http://garage.cse.msu.edu/software/lil-gp/>, September 1998.
- [64] A. Rogers and A. Prügel-Bennett. *Theoretical Aspects of Evolutionary Computing*, chapter Modelling the Dynamics of a Steady State Genetic Algorithm, pages 57–68. Springer, 1999.
- [65] R. C. Rosenberg. The gyrobondgraph: A canonical form for multiport systems models. *Proceedings ASME Winter Annual Conference*, 1976.

- [66] R. C. Rosenberg and D. C. Karnopp. A definition of the bond graph language. *Journal of Dynamic Systems, Measurement, and Control*, pages 179–182, September 1972.
- [67] Ronald C. Rosenberg and Dean Karnopp. *Introduction to Physical System Dynamics*. McGraw-Hill Publishing Company, New York, New York, 1983.
- [68] Conor Ryan, J. J. Collins, and Michael O Neill. Grammatical evolution: Evolving programs for an arbitrary language. In Wolfgang Banzhaf, Riccardo Poli, Marc Schoenauer, and Terence C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391, pages 83–95, Paris, France, 1998. Springer-Verlag.
- [69] Kazumi Saito, Pat Langley, Trond Grenager, and Christopher Potter. Computational revision of quantitative scientific models. *Proceedings of the 3rd International Conference on Discovery Science (DS2001)*, LNAI2226:336–349, 2001.
- [70] I. Scott and B. Mlllgrew. Orthonormal function neural network for nonlinear system modeling. *IEEE International Conference on Neural Networks*, 4:1847–1852, 1996.
- [71] K. Seo, Z. Fan, J. Hu, E. D. Goodman, and R. C. Rosenberg. Toward an automated design method for multi-domain dynamic systems using bond graphs and genetic programming. *Mechatronics*, 13(8-9):851–885, 2003.
- [72] Pete Shinnars et al. Pygame: Python game development. <http://pygame.org/>, 2000.
- [73] J. Sjöberg, Q. Zhang, L. Ljung, A. Benveniste, B. Delyon, P-Y. Glorennec, H. Hjalmärsson, and A. Juditsky. Nonlinear black-box modelling in system identification: mathematical foundations. *Automatica*, 31, 1995.
- [74] Reinhard Stolle and Elizabeth Bradley. Opportunistic modeling. *Proceedings IJCAI Workshop Engineering Problems for Qualitative Reasoning*, 1997.



- [75] Reinhard Stolle and Elizabeth Bradley. Multimodal reasoning for automatic model construction. *Proceedings Fifteenth National Conference on Artificial Intelligence*, 1998.
- [76] Matthew J. Streeter, Martin A. Keane, and John R. Koza. Routine duplication of post-2000 patented inventions by means of genetic programming. In James A. Foster, Evelyne Lutton, Julian Miller, Conor Ryan, and Andrea G. B. Tettamanzi, editors, *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of *LNCS*, pages 26–36, Kinsale, Ireland, 2002. Springer-Verlag.
- [77] Christian Tismer. Stackless python, a python implementation that does not use the c stack. <http://stackless.com/>, 1999.
- [78] L. Todorovski. Declarative bias in equation discovery. Master’s thesis, Faculty of Computer and Information Science, Ljubljana, Slovenia, 1998.
- [79] L. Todorovski and S. Dzeroski. Declarative bias in equation discovery. *Proc. Fourteenth International Conference on Machine Learning*, pages 376–384, 1997.
- [80] L. Todorovski and S. Dzeroski. Theory revision in equation discovery. *Lecture Notes in Computer Science*, 2001.
- [81] T. Treichl, S. Hofmann, and D. Schroder. Identification of nonlinear dynamic miso systems with orthonormal base function models. *Proceedings of the 2002 IEEE International Symposium on Industrial Electronics*, 1, 2002.
- [82] Guido van Rossum. Computer programming for everybody: A scouting expedition for the programmers of tomorrow. CNRI Proposal 90120-1a, Corporation for National Research Initiatives, July 1999.
- [83] Guido van Rossum et al. Python programming language. <http://python.org/>, 1990.
- [84] Darrell Whitley. A genetic algorithm tutorial. *Statistics and Computing*, pages 65–85, 1994.
- [85] W. J. Wilson. *System Identification (E&CE 683 course notes)*. University of Waterloo, Waterloo, Canada, September 2004.

- [86] Ashraf A. Zeid and Chih-Hung Chung. Bond graph modeling of multi-body systems: A library of three-dimensional joints. *Journal of the Franklin Institute*, (329):605–636, 1992.