

DATA-DRIVEN, FREE-FORM MODELING OF BIOLOGICAL SYSTEMS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

In Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Theodore William Cornforth III

January 2014

© 2014 Theodore William Cornforth III

DATA-DRIVEN, FREE-FORM MODELING OF BIOLOGICAL SYSTEMS

Theodore William Cornforth III, Ph. D.

Cornell University 2014

The quantity of data available to scientists in all disciplines is increasing at an exponential rate, yet the insight necessary to distill data into scientific knowledge must still be supplied by human experts. This widening gap between data and insight can be bridged with data-driven modeling, in which computational methods shift much of the work in creating models from humans to computers. Traditional approaches to data-driven modeling require that the form of the model be fixed in advance, which requires substantial human effort and limits the complexity of problems that can be addressed. In contrast, a newer approach to automated modeling based on evolutionary computation (EC) removes such restrictions on the form of models. This free-form modeling has the potential both to reduce human effort for routine modeling and to make complex problems more tractable. Although major advances in EC-based modeling have been made in recent years, many challenges remain. These challenges include three features often seen in biological systems: complex nonlinear behavior, multiple time scales, and hidden variables.

This work addresses these challenges by developing new approaches to EC-based modeling, with applications to neuroscience, systems biology, ecology, and other fields. The contributions of this work consist of three primary lines of research. In the first line of research, EC-based methods for the automated design of analog

electrical circuits are adapted for the modeling of electrical systems studied in neurophysiology that display complex, nonlinear behavior, such as ion channels. In the second line of research, EC-based methods for symbolic modeling are extended to facilitate the modeling of dynamical systems with multiple time scales, such as those found throughout ecology and other fields. Finally, in the third line of research, established EC-based algorithms are extended with the capability to model dynamical systems as systems of differential equations with hidden variables, which can contribute in an essential way to the observed dynamics of a physical system yet historically have presented a particularly difficult challenge to automated modeling.

BIOGRAPHICAL SKETCH

Theodore William Cornforth III was born in Orange, California on October 5th, 1978. His parents are Laurie Ann Cornforth and Theodore William Cornforth II, and his siblings are Anne Michelle Cornforth and Jonathan Francis Cornforth. Some of Theodore's earliest memories are of being fascinated by minerals, constellations, insects, archeological relics, and other aspects of the natural and artificial world. As he grew older, his interests gradually came to include what is arguably the loftiest ambition in all of science: the creation of an artificial intelligence to rival or even surpass the capabilities of any human. As frightening as this idea can be, it would surely be a watershed moment in history, ushering in a new era of unprecedented advances in all areas of science, medicine, and technology.

While this goal will remain firmly in the realm of science fiction for the foreseeable future, Theodore decided to join the ranks of the many thousands of scientists from a variety of disciplines who are contributing to its realization. Computational neuroscience is one field at the heart of this dream of human-level artificial intelligence, and in 2003, Theodore entered graduate school at the University of Oregon to pursue study in that area. After completing an M.S. degree under the direction of Professor Terry Takahashi in 2007, Theodore's interests shifted in focus from biological to more purely computational areas. This prompted another course of study at the University of Oregon and resulted in a second M.S. degree, in the area of computing and information science, under the guidance of Professor John Conery in 2009.

Theodore then decided to continue his education at Cornell University in the field of computational biology, which resulted in his joining the Creative Machines Lab under the direction of Professor Hod Lipson in 2010. His work at Cornell was primarily focused on the development of artificial intelligence for automating the process of mathematical modeling, and the sum total of this work is presented here. It is his hope that this body of work contributes in some small way to the ongoing endeavor from which a new kind of truly intelligent entity will someday emerge.

For Bernice Modell Cornforth DeBolt, Theodore William Cornforth,
Mary Colletta Mayes, and Francis Edward Mayes, Jr.

ACKNOWLEDGMENTS

I would like to thank Professors Hod Lipson, David Christini, Bart Selman, Christiane Linster, Kyung-Joong Kim, Jim Torresen, Terry Takahashi, John Conery, and Anthony Long for their mentorship and guidance. I owe a debt of gratitude to Margie Hinonangan-Mendoza, Sue Bishop, and all members of the Cornell Creative Machines Lab for their help, scientific and otherwise. Financial resources were provided by the Tri-Institutional Training Program in Computational Biology and Medicine, National Science Foundation grant ECCS 0941561 on Cyber-enabled Discovery and Innovation (CDI), National Institutes of Health NIDA grant RC2 DA028981, and the U.S. Defense Threat Reduction Agency (DTRA) grant HDTRA 1-09-1-0013.

I have been blessed by the love and support of an amazing family: my parents Laurie and Ted, who taught me the value of knowledge, my siblings Annie and Jon with whom I shared a childhood, and my grandparents: TWC I (Grandpa), Bernice (Grandma), Frank (Gumpy), and Colletta (Grammy).

To Kristin, I would like to express my love and gratitude that goes beyond any words. You make life worth living.

Creator of All Things, may my infinitesimal efforts at unlocking the mysteries of Your universe serve to honor You and to magnify the humility and wonder with which we, humanity, behold Your creation.

TABLE OF CONTENTS

BIOGRAPHICAL SKETCH	iii
DEDICATION	v
ACKNOWLEDGMENTS	vi
TABLE OF CONTENTS	vii
LIST OF FIGURES	xi
LIST OF TABLES	xvi
PREFACE	xvii
CHAPTER 1. INTRODUCTION	1
CHAPTER 2. NONLINEAR AND NEUROPHYSIOLOGICAL MODELS	11
Introduction	11
Background	11
Initial Experiments	19
Circuit Evolution	19
Neurophysiological Data	22
Results	23
Discussion	23
Evolving Robust Circuit Models	31
Neurophysiological Data	31
Circuit Evolution	33
Results	39
Discussion	41
Reverse-Engineering Nonlinear Analog Circuits	47
Introduction	47
Background	48
Genetic Algorithm	51
Stimulus Selection	54
Target Circuits	56
Results	60
Discussion	61
Comparison of Approaches to Analog Circuit Design	65
Introduction	65
Background	65
Representation	67

Compared Algorithms	68
Standard Genetic Algorithm	69
Evolution Strategies	69
Immune Programming	70
Age-Fitness Pareto Optimization	70
Naive Algorithms	71
Results	72
CHAPTER 3. DYNAMICAL SYSTEMS WITH MULTIPLE TIME SCALES	78
Introduction	78
Background	78
Methods	83
Algorithm Step 1	85
Algorithm Step 2	87
Algorithm Step 3	89
Symbolic Regression	89
Results	92
Application to Synthetic Systems	92
Application to Physical Systems	98
Discussion	101
CHAPTER 4. INFERENCE OF HIDDEN VARIABLES	106
Introduction	106
Background	106
Related Work	110
Methods	114
Problem Statement	114
GP Algorithm for ODE Model Inference	117
Fitness Evaluation	121
Results	124
Initial Experiments	124
Ideal Rocket Equations	134
Chemical Reaction Network	137
Physical 5-azacytidine Degradation Kinetics	144
Physical Pendulum	148
Random Systems of ODEs	151
Discussion	153
CHAPTER 5. CONCLUSION	162
APPENDIX 1. EVOLUTION OF SORTING ALGORITHMS	164
Introduction	164
Methods	171
Overview	171
The Evolver and Mapper	173

Building Blocks	174
Fitness Calculations	179
Experiments	180
Initial Experiments	180
for () Loops	182
Seeding Experiments	185
Evolution of Sorting Algorithms	190
Conclusion	192
APPENDIX 2.C API FOR SORTING PROGRAMS	194
APPENDIX 3.IMPLEMENTATION OF 14 SORTING ALGORITHMS	196
APPENDIX 4.GRAMMAR FOR SORTING PROGRAMS	205
APPENDIX 5.MACHINE LEARNING OF OPTIONS TRADING STRATEGIES	208
Introduction	208
Methods	210
Data Collection	210
Data Processing with Sliding Windows	212
Feature Construction	214
Classifier Models	217
Results	218
Discussion	221
APPENDIX 6.RANDOM FUNCTIONS WITH MULTIPLE TIME SCALES	224
APPENDIX 7.COMPARISON OF EVOLUTIONARY ALGORITHMS FOR OPTIMIZATION AND SYMBOLIC REGRESSION	226
Introduction	226
Genetic Algorithms for the Traveling Salesman Problem	226
Genetic Programming Algorithms for the Symbolic Regression Problem	230
APPENDIX 8.GATING OF DENDRODENDRITIC INHIBITION OF MITRAL CELLS BY GRANULE CELLS IN THE MAMMALIAN OLFACTORY SYSTEM	241
Introduction	241
Methods	241
Results and Discussion	243
APPENDIX 9.OVERVIEW OF MACHINE LEARNING	248
Introduction	248
Training and Testing	250

Bayesian Perspectives on Learning	257
Linear Models	264
Artificial Neural Networks	267
REFERENCES	270

LIST OF FIGURES

Figure 1.1. Symbolic regression.	4
Figure 1.2. Pathological features of search spaces.	4
Figure 1.3. Genetic programming for a symbolic regression application.	8
Figure 1.4. Bursting activity in a mouse olfactory neuron.	10
Figure 2.1. Simple electrophysiology setup.	12
Figure 2.2. Llapicque’s analog circuit model for a neuron.	13
Figure 2.3. The Hodgkin-Huxley model.	15
Figure 2.4. The embryonic circuit.	20
Figure 2.5. Step response of voltage-gated potassium and sodium channels.	24
Figure 2.6. Schematic of an evolved potassium channel circuit.	25
Figure 2.7. Schematic of an evolved sodium channel circuit.	26
Figure 2.8. Fitness as a function of the number of evaluated circuits.	27
Figure 2.9. A closed loop experimental system for modeling neurophysiological systems with analog circuits.	30
Figure 2.10. Comparison of target ion channel behavior and evolved model circuit behavior.	34
Figure 2.11. Error vs. number of fitness evaluations for each of the six primary evolution techniques, each of which was combined with Age-Fitness optimization as the secondary algorithm.	40
Figure 2.12. Performance of evolved circuits on previously unseen test data.	41
Figure 2.13. Overfitting of potassium channel circuit models.	41
Figure 2.14. Fitness evaluations for linear and non-linear analog circuit evolution.	50
Figure 2.15. Circuit representation.	52

Figure 2.16. RTL inverter target and evolved circuits.	57
Figure 2.17. Random target and evolved circuits.	59
Figure 2.18. Generalization error for the five stimulus selection methods with the RTL inverter target.	61
Figure 2.19. Generalization error for the five stimulus selection methods with random targets.	62
Figure 2.20. RTL inverter target and evolved circuit behavior.	63
Figure 2.21. Random target and evolved circuit behavior.	64
Figure 2.22. Basic GA compared with naive algorithms.	73
Figure 2.23. Naive crossover.	74
Figure 2.24. Variants of the basic GA compared with naive algorithms, evolution strategies, and immune programming.	74
Figure 2.25. Basic GA with naive 2 point crossover compared with a basic GA using topology-aware crossover.	75
Figure 2.26. Size of evolved circuits as a function of number of evaluations for the experiment shown in Figure 2.25.	75
Figure 2.27. Size of evolved circuits obtained with the Age-Fitness GA compared with the basic GA and naive algorithms.	76
Figure 2.28. Error of evolved circuits as a function of number of evaluations for the experiment shown in Figure 2.27.	77
Figure 3.1. A forced Van der Pol oscillator time series and its corresponding frequency spectrum.	79
Figure 3.2. Natural dynamical systems with multiple time scales.	80
Figure 3.3. Reverse-engineering multiple-time-scale dynamical systems.	81
Figure 3.4. Proposed algorithm pseudocode.	84
Figure 3.5. Algorithm Step 1: modeling fast dynamics.	86
Figure 3.6. Algorithm Step 2: separating slow dynamics from the data.	88

Figure 3.7. Algorithm Step 3: modeling slow dynamics and completing model inference.	90
Figure 3.8. The four systems used to test the proposed algorithm.	93
Figure 3.9. Effects of noise, magnitude of time scale differences, window size and computational effort on algorithm performance.	97
Figure 3.10. Algorithm performance on random target functions of different complexities.	99
Figure 3.11. Modeling process for the physical cantilever beam.	99
Figure 4.1. The system of ODEs given in Equation 4.7 encoded as a forest of two binary trees.	118
Figure 4.2. Summary of dynamical systems studied.	124
Figure 4.3. Behavior of the first target system and an evolved model from the Initial Experiments section.	127
Figure 4.4. Summary of evolutionary runs for the first example in the Initial Experiments section.	128
Figure 4.5. Behavior of the second target system and an evolved model from the Initial Experiments section.	132
Figure 4.6. Summary of evolutionary runs for the second example in the Initial Experiments section.	133
Figure 4.7. Behavior of the target system and an evolved model from the Ideal Rocket Equations section.	135
Figure 4.8. Summary of evolutionary runs for the Ideal Rocket Equations section.	136
Figure 4.9. Behavior of the target system and an evolved model from the Chemical Reaction Network section.	139
Figure 4.10. Summary of evolutionary runs for the Chemical Reaction Network section with measured variables $\{X_1\}$, $\{X_2\}$, $\{X_1, X_2\}$	140
Figure 4.11. Summary of evolutionary runs for the Chemical Reaction Network section with measured variables $\{X_3\}$, $\{X_1, X_3\}$, $\{X_2, X_3\}$	143

Figure 4.12. Behavior of the target system and an evolved model from the Physical 5-azacytidine Degradation Kinetics section.	146
Figure 4.13. Summary of evolutionary runs for the Physical 5-azacytidine Degradation Kinetics section.	147
Figure 4.14. Behavior of the target system and an evolved model from the Physical Pendulum section.	149
Figure 4.15. Summary of evolutionary runs for the Physical Pendulum section.	150
Figure 4.16. Summary of experiments with random systems of ODEs of two variables.	153
Figure 4.17. Summary of experiments with random systems of ODEs of three variables.	154
Figure 4.18. Ambiguity in nonlinear systems of ODEs.	155
Figure 4.19. Non-identifiability in linear systems of ODEs.	157
Figure A1.1. Grammar-based genetic programming.	172
Figure A1.2. Three examples of randomly generated, valid programs obtained in the first experiment.	183
Figure A1.3. Three examples of randomly generated, invalid programs obtained in the first experiment.	184
Figure A1.4. Three examples of valid programs randomly generated using the grammar from the <code>for ()</code> loops experiment.	185
Figure A1.5. Mean fitness of the population in the seed experiment as a function of generation.	189
Figure A5.1. One month of raw data.	213
Figure A5.2. Cross validation experiments.	215
Figure A7.1. Results for traveling salesman Test Problem 1.	230
Figure A7.2. Results for traveling salesman Test Problem 2.	232
Figure A7.3. Results for traveling salesman Test Problem 3.	234

Figure A7.4. Results for symbolic regression Test Problem 4.	237
Figure A7.5. Results for symbolic regression Test Problem 5.	238
Figure A7.6. Results for symbolic regression Test Problem 6.	239
Figure A8.1. The $Mi \rightarrow Gr \rightarrow Mi$ model.	242
Figure A8.2. Effect of relative timing of proximal (APC) and distal (mitral cell soma) inputs to the granule onto mitral DDI.	244
Figure A8.3. Repeated proximal and distal stimuli at the same frequency, Part I.	245
Figure A8.4. Repeated proximal and distal stimuli at the same frequency, Part II.	246
Figure A8.5. Synchronization of Mi cell oscillatory activity.	247

LIST OF TABLES

Table 2.1. Components and parameter ranges used in circuit evolution.	21
Table 2.2. Components and parameter values for the evolved potassium channel circuit.	28
Table 2.3. Components and parameter values for the evolved sodium channel equivalent circuit.	29
Table 3.1. Models produced by the proposed algorithm and by symbolic regression alone for the two synthetic systems: the sum-of-sines system and a Van der Pol Oscillator.	95
Table 3.2. Models produced by the proposed algorithm and by symbolic regression alone for the two physical systems: a cantilever beam and the 5-azacytidine reaction system.	102
Table 4.1. Evolved models for the first system in the Initial Experiments section.	129
Table 4.2. Evolved models for the second system in the Initial Experiments section.	134
Table 4.3. Evolved models for the Ideal Rocket Equations section.	137
Table 4.4. Evolved models for the Chemical Reaction Network section.	141
Table 4.5. Evolved models for the Physical 5-azacytidine Degradation Kinetics section.	148
Table A1.1. Time complexity of different sorting algorithms.	181
Table A5.1. Parameter values used to train the final classifiers.	220
Table A5.2. Performance of classifiers on test data measured using gain.	220
Table A5.3. Performance of classifiers on test data measured using the expectancy score.	222
Table A6.1. Ten example randomly generated target functions with multiple time scales for each of six different complexities.	224

PREFACE

As perhaps with all science, my work at Cornell took an unexpected and circuitous path. Nonetheless, my efforts in many different directions gradually coalesced into three main lines of research. These are introduced in Chapter 1 and then described in detail in Chapters 2, 3, and 4. Some closing remarks appear in Chapter 5. A significant amount of additional research not directly related to these three main lines of research is described in the various appendices and included for the sake of completeness. Although my contributions pale in comparison to those of the many who have come before me, I present this record so that those who follow may find inspiration within its pages.

CHAPTER 1

INTRODUCTION

Models that organize, explain, and predict empirical observations play a central role in science. The form taken by a model is problem specific and must be designed by a human expert, usually at the expense of considerable time and effort. At the same time, the rate at which scientists are able to generate empirical data is expected to steadily increase for the foreseeable future, as is the complexity of problems routinely studied in all fields (Clery and Voss 2005, Szalay and Gray 2006, Strogatz 2001, Strogatz 2007). These two factors suggest there is a widening gap between our ability to generate data and our ability to distill data into useful scientific models. New computational methods will be necessary to close this gap by assisting humans in the creation of scientific models (King et al. 2004, Waltz and Buchanan 2009, Evans and Rzhetsky 2010).

Many methods have previously been proposed and successfully used for automating the modeling process. An important class of scientific models applies to quantitative experiments in which independent variables are manipulated to produce changes in the observed dependent variables. Models of this type consist of a function f that maps values in the domain of independent variables \mathbf{x} to values in the range of dependent variables \mathbf{y} . This is a difficult problem in general, as the function f must be found based on a limited sample of paired values from \mathbf{x} and \mathbf{y} . Automated methods for finding f in this setting are usually considered to be part of the supervised machine

learning or statistical modeling fields. In machine learning terms, the process of finding f is known as training the model. The set of measurements of the input variables \mathbf{x} and the output variables \mathbf{y} that is used for training is called the training data set. The space of possible functions is called the problem representation. Because the training set is finite, only an approximation to the true function f can be found. The quality of this approximating function can be estimated by measuring its ability to map input data to output data in the training set. However, a more reliable estimate of quality is the function's generalization ability, that is, its ability to map input data to output data when that data were not present in the training set. Both estimates of quality are important for the training process, as will be discussed in more detail below. This work is primarily concerned with regression models, in which both the input and output variables are continuous, real-valued numbers.

The vast majority of supervised machine learning methods in use today assume representations in which the form of the function f is substantially fixed in advance. These methods are well established and have been used with great success in many applications. Highly efficient training algorithms are usually associated with such "fixed-form models," either because closed form expressions for the optimal values of the adjustable parameters have been derived or because classical optimization methods such as gradient descent are applicable. Familiar techniques such as linear regression, Generalized Linear Models, artificial neural networks, Markov models, and Kalman filters all arguably fall into this category (Draper and Smith 1998, Bishop 2006).

Consider as an example the case of linear regression with one dependent variable. Using the terminology introduced above, the linear regression task is to find

functions of the form

$$f(\mathbf{x}) = w_0 + \sum_{j=1}^{M-1} w_j x_j \quad (1.1)$$

where M is the number of adjustable weights $(w_0, w_1, \dots, w_{M-1})$, which are multiplied by the $(M - 1)$ independent variables $(x_1, x_2, \dots, x_{M-1})$. The automated portion of linear regression occurs during training when values of the weights are found that are optimal given a set of statistical assumptions (Bishop 2006). In the absence of any prior knowledge that a simple weighted sum is a meaningful model for the data, it is unlikely that even an f with these optimal weights would generalize well or that it would provide insight into the natural phenomenon being studied.

Consider instead a “free-form” representation such as $f \in \{\text{all functions that can be constructed with algebraic operations over the independent variables}\}$. If models using such a representation could be effectively trained with automated methods, this would free the human modeler from having to search the vast space of possible functional forms themselves and from acquiring and encoding the additional domain knowledge that would be needed to constrain this space. This type of free-form, data-driven modeling is the main focus of this work. The focus will primarily be on algebraic models, and free-form, data-driven modeling with an algebraic representation is usually called “symbolic regression” (Figure 1.1).

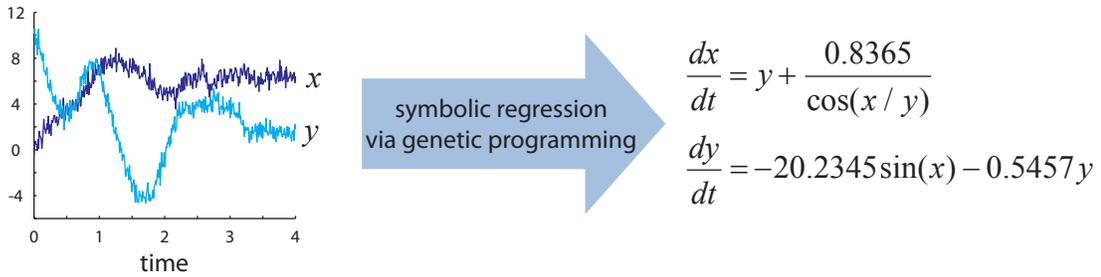


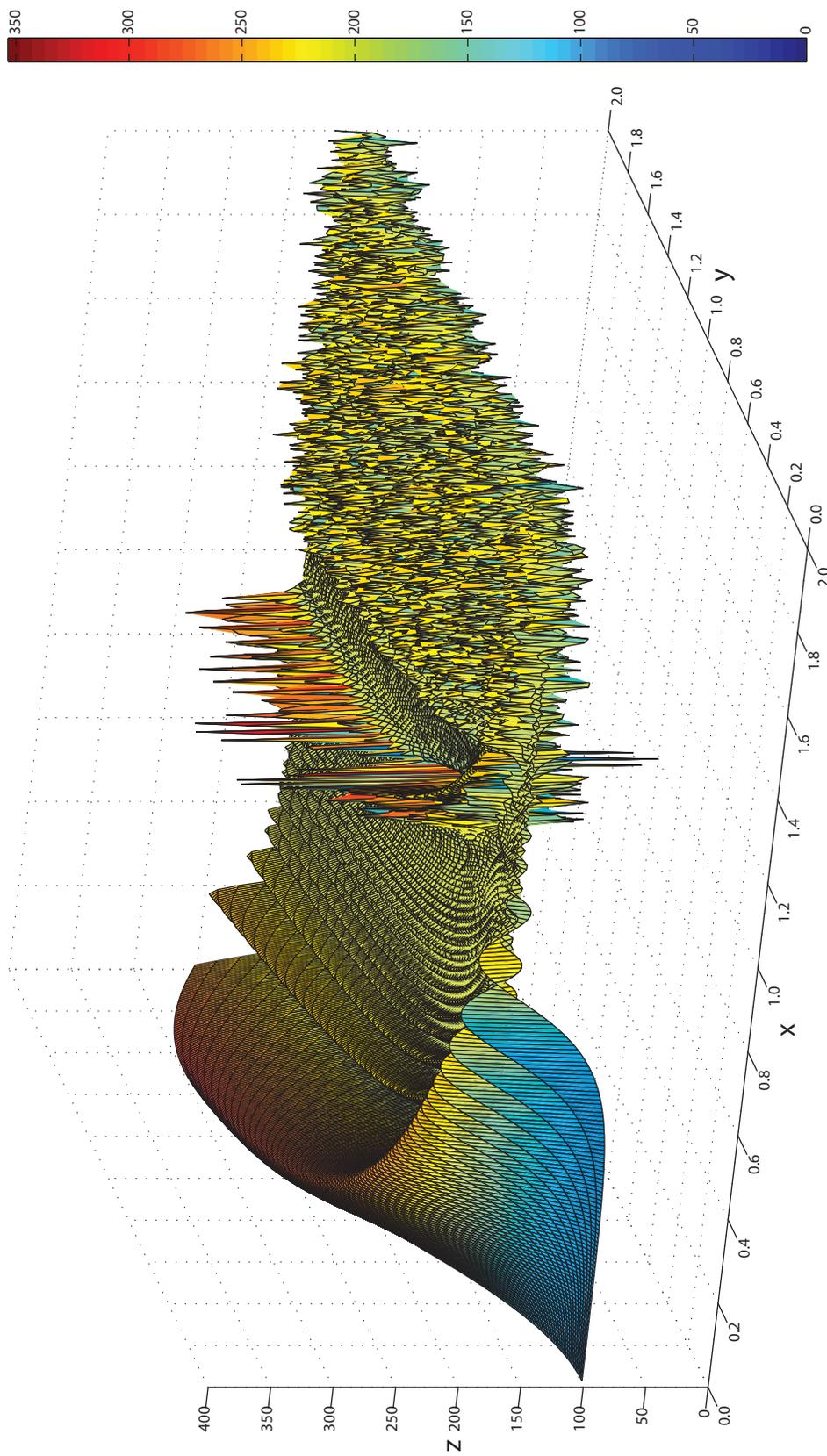
Figure 1.1. Symbolic regression. Symbolic regression algorithms take raw data as input (left), and without any additional human-supplied information, generate free-form algebraic models that explain and predict the data (right).

The primary disadvantage of free-form modeling is that training of the model is considerably more difficult than for fixed-form models. As noted above, fixed-form models can usually be trained with relatively efficient algorithms that take advantage of closed form or differentiable expressions for model quality as a function of the adjustable parameters. In contrast, with free-form models, it is usually the case that little or nothing is known about how to efficiently find good functions. This problem compounds issues present in all optimization and machine learning applications, such as complex patterns of local optima in the search space (Figure 1.2) and the approximation-generalization tradeoff.

Figure 1.2. Pathological features of model search spaces (next page). This surface represents the value z as a function of $\mathbf{x} = (x_1, x_2)$, where

$$z = f(\mathbf{x}) = \sum_{t=1}^{200} [\sin(t) - \sin(x_1 t^{x_2})]^2$$

and $x_i \in [0, 2]$. Over the range plotted, the global minimum occurs at $\mathbf{x}^* = (1, 1)$, where $f(\mathbf{x}^*) = 0$. Despite the apparent simplicity with which f can be expressed, a variety of so-called search space pathologies are present. These include plateaus, numerous local maxima and minima, and deceptive gradients. In addition, the local gradient containing the global minimum is extremely narrow relative to the size of the search space, a phenomenon often referred to as the “needle-in-a-haystack” problem.



Although little can be done about pathological search spaces in most cases, the approximation-generalization tradeoff is a fundamental concept in machine learning that can be approached with a wide variety of different methods. Essentially, the approximation-generalization tradeoff is a universal phenomenon whereby as the complexity of the space of possible models considered increases, inferred model error on a training data set will tend to decrease but error on a test data set will tend to increase (Abu-Mostafa et al. 2012). While a complete review of the theory and practice related to the approximation-generalization tradeoff is beyond the scope of the present work, see Appendix 9 for a further discussion of these and other fundamental machine learning issues.

The primary means of addressing the approximation-generalization tradeoff that is used in this work is to consider a Pareto hall of fame throughout the process of a model search. This hall of fame contains all the best models found so far, as judged by their accuracy on the training data and the complexity of the model. At any point in the search, there is no known model with lower accuracy and lower complexity than any of the models in the Pareto hall of fame. At the end of the model search, the Pareto hall of fame is taken as the results of the search. Further techniques can then be used to identify models of interest within this hall of fame, although these techniques are largely qualitative and problem-specific.

Many attempts have been made to address search space pathologies, the approximation-generalization tradeoff, and other issues in machine learning applications. In the context of free-form modeling, some historical approaches have addressed these issues by requiring that a human expert encode a large amount of

domain-specific knowledge related to the problem. These systems include BACON (Langley et al. 1987) and LAGRANGE (Džeroski and Todorovski 1995), as well as the more recent systems PROMETHEUS (Bridewell et al. 2006) and Adam the “robot scientist” (King et al. 2009).

A potentially more powerful approach to training free-form models in a data-driven setting is through stochastic optimization (Luke 2011). Stochastic optimization algorithms search through the space of possible functions f in a semi-random fashion and typically require large amounts of computing power. The most powerful stochastic optimization methods include evolutionary algorithms, in which a population of candidate functions is maintained and used to bias the search through function space in various ways (Fogel et al. 1966, Fogel 1995, De Jong 2006). When evolutionary algorithms are used for free-form modeling tasks such as symbolic regression, the approach is usually called “genetic programming” after the influential body of work by John Koza in which many of the applications of evolving programs were first explored (Koza 1992, Koza 1994, Koza et al. 1999, Koza et al. 2003).

Genetic programming for a symbolic regression application works by starting with a “primordial soup” of building blocks (Figure 1.3a). These building blocks are the set of possible mathematical and algebraic primitives with which functions can be constructed. The building blocks are used to make an initial population of several candidate functions at random (Figure 1.3b). A fitness value is assigned to each of these functions by measuring how accurately each function maps the value of independent variables to the value of dependent variables, as gauged by performance on a training data set. Individual members of the population with the highest fitness

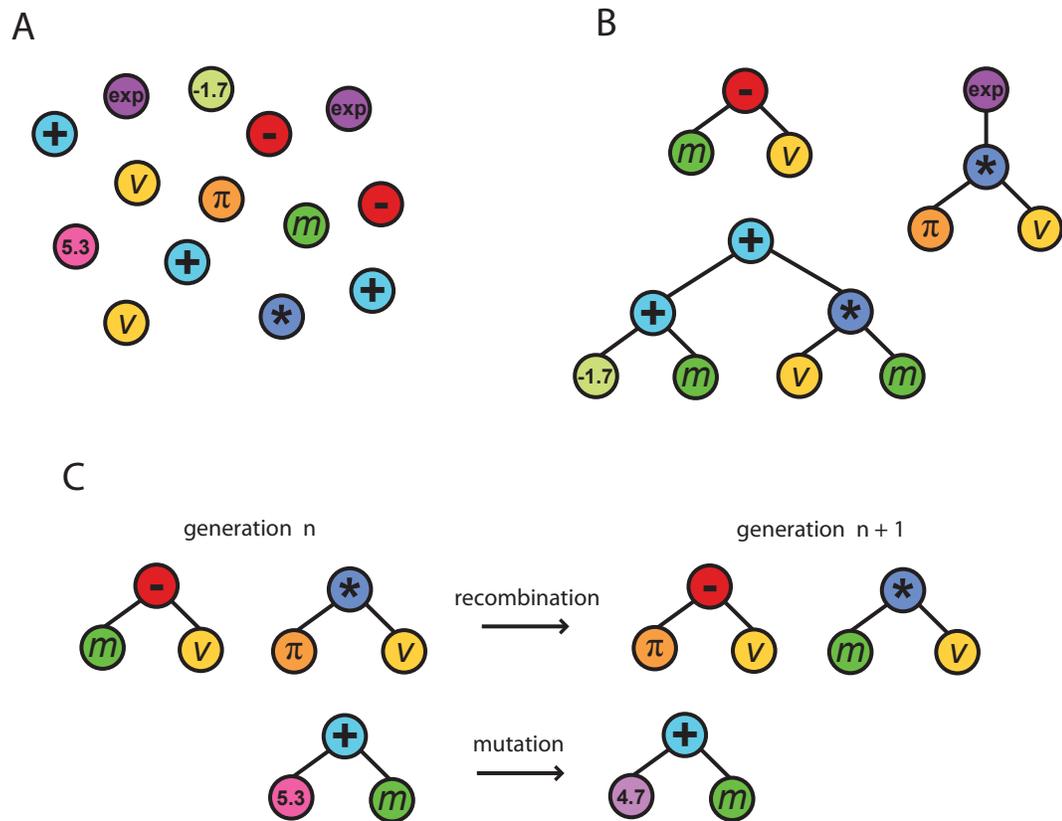


Figure 1.3. Genetic programming for a symbolic regression application. A) A primordial soup of algebraic building blocks. B) A population of simple functions is randomly created from the building blocks in generation 0. C) As evolution proceeds, variation is introduced and maintained with recombination and mutation.

are selected to survive into the next generation. However, before this next generation is formed, variation is introduced into the survivors with two operations: mutation, in which portions of single individuals are randomly altered, and recombination, in which portions of two or more individuals are randomly exchanged (Figure 1.3c). The cycle of selection, mutation, and recombination continues for several generations until an individual of acceptable quality has been obtained or until the allotted time for training has elapsed.

In the examples of Figure 1.1 and Figure 1.3, the building blocks used by evolution are algebraic and the rules for combining them restrict individuals in the population to taking the form of structures called expression trees. Expression trees have been extensively studied as part of the Lisp programming language and it is known that every computable function is captured with such a representation (Boyer and Moore 1984). However, for other problems, a different set of building blocks may be more appropriate. For example, electrical building blocks are widely used. Such a set of electrical blocks used in a particular problem might consist of resistors, capacitors, transistors, etc. (Koza et al. 1996a, Koza et al. 1997). In that case, the representation is the space of possible analog circuits instead of the space of algebraic expressions.

In this work, analog circuits, algebraic expressions, pseudocode fragments, hidden Markov models, and other representations are used for different applications. These applications primarily fall into three main lines of research, each of which focuses on a particular challenge faced by algorithms for the automated inference of free-form models. Consider the membrane voltage data from a mouse olfactory neuron shown in Figure 1.4. Several features of this empirical data are apparent that would make it difficult for any algorithm to infer a model that explains the data. These include complex and nonlinear dynamics, the presence of dynamics at multiple time scales (the overall rise and fall of the bursting regions in addition to the intervals separating action potentials within individual bursts), and the likelihood that various unmeasured or “hidden” variables are involved in the dynamics and required for an interpretable and parsimonious model of the data. Complicating factors such as these are the rule

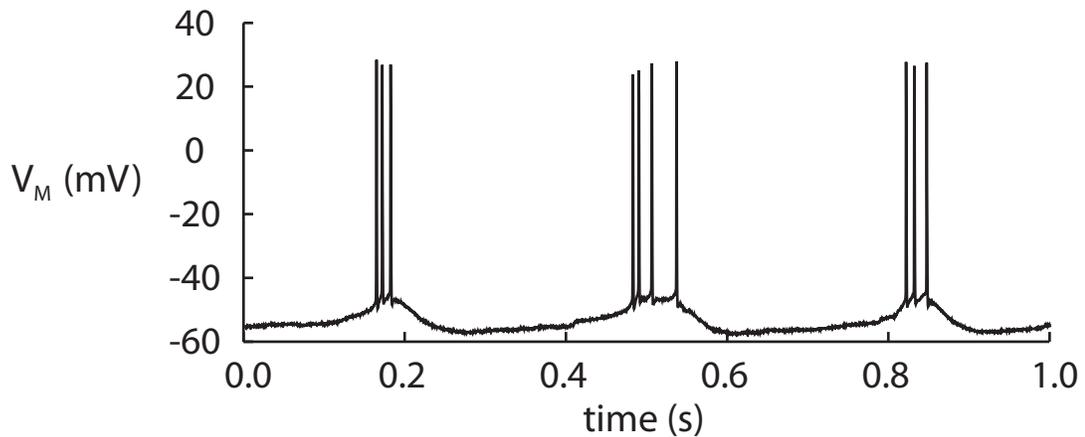


Figure 1.4. Bursting activity in a mouse olfactory neuron. Shown is a 1-second portion of membrane voltage data collected by and provided courtesy of Abdallah Hayar.

rather than the exception in naturally-occurring dynamical systems from all areas of science and engineering, and the need is great for improved modeling algorithms that take these factors into account.

This work addresses these three main challenges by developing data-driven, free-form modeling algorithms for complex and nonlinear electrical systems, for ordinary differential equation models of dynamical systems with multiple time scales, and for ordinary differential equations models of dynamical systems with hidden variables. Chapters 2, 3, and 4 describe these three lines of research in detail. A conclusion and summary of the contributions of this work are included in Chapter 5. Additional research not directly related to these three main lines of research is included in several of the appendices that follow Chapter 4. This additional research covers a broad range of machine learning topics, including the automated design of sorting algorithms, the inference of financial models, and comparisons of different methods for optimization and symbolic regression problems.

CHAPTER 2

NONLINEAR AND NEUROPHYSIOLOGICAL MODELS

Introduction

In this chapter, analog electrical circuits are used to model the types of complex, nonlinear dynamical systems commonly observed in neurophysiological systems. Both the use of circuits for this purpose and the automation of analog circuit design are well established techniques. Here, these two ideas are combined with the goal of automatically constructing analog circuit models of physiological systems that might otherwise present an intractable challenge to automated modeling algorithms.

Background

Analog electrical circuits have long been used as tools to understand the behavior of biological neurons. The work of Lapicque in 1907 was perhaps the first successful application of this idea (Lapicque 1907, Abbott 1999). He used the experimental setup shown in Figure 2.1, and on the basis of observations made with such an apparatus, he reasoned that the lipid bilayer membrane separating the intracellular space from the extracellular fluid is capable of storing charge and can act like an electrical capacitor. In addition, measurements of the membrane voltage in response to applied currents suggested the presence of a “leak” conductance that acts

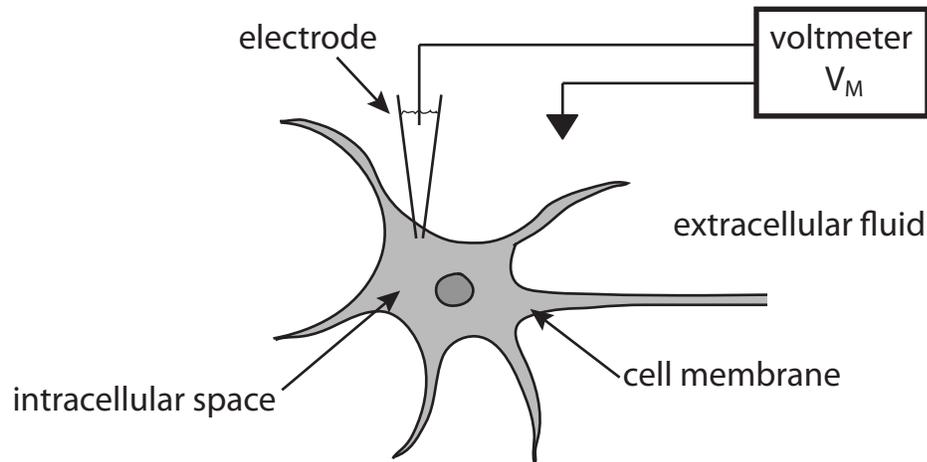


Figure 2.1. Simple electrophysiology setup. Using a micromanipulator and microscope, a glass electrode can be positioned inside an individual neuron in such a way as to minimize damage to the cell membrane. This creates a simple circuit for measuring the potential difference between the intracellular space and the extracellular fluid. This potential difference is called the membrane voltage, V_M . A stimulus current (the “input” to the cell) can be delivered through the electrode and the resulting changes in V_M (the “output” from the cell) then recorded for later study.

like a resistor-battery combination in its tendency to slowly return the membrane voltage to a baseline resting value. An analog circuit matching this description is a capacitor in parallel with a resistor and battery, and is perhaps the simplest circuit capable of reproducing the fundamental electrical properties of a neuron (Figure 2.2). Such a circuit is often referred to as an “equivalent” circuit for a neuron in that it reproduces the essential behavior of a more complicated electrical system in relatively simple form.

The concept of an equivalent circuit for a neuron has proven very useful for the development of more sophisticated models of neuron behavior. Lapique himself used the circuit in Figure 2.2 as the basis for the widely used integrate-and-fire model, in which all-or-nothing spikes in membrane voltage called action potentials are triggered when the capacitor is charged to some threshold potential (Lapique 1907).

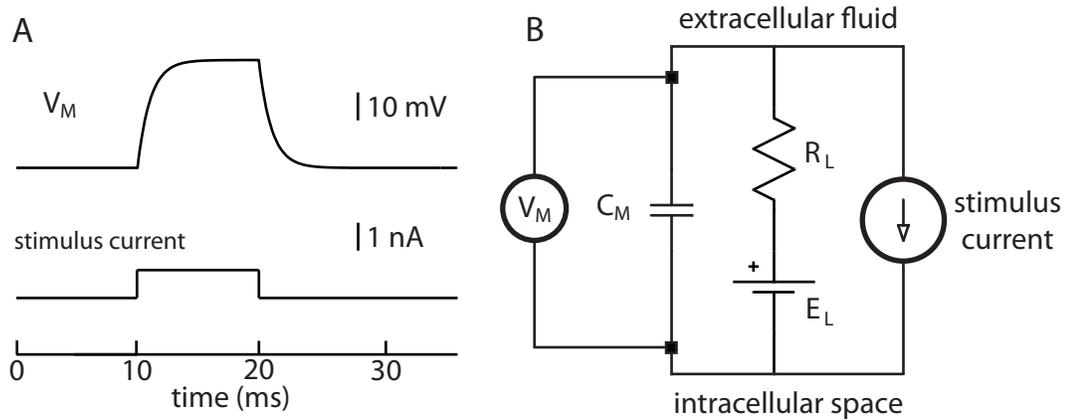


Figure 2.2. Lapicque’s analog circuit model for a neuron. **A)** Stimulation of a cell with a small current applied through the electrode results in a membrane voltage (V_M) response similar to that of an RC circuit. **B)** Lapicque developed a simple, intuitive analog circuit that explains and predicts this behavior. The node at the top of the circuit represents the extracellular fluid and the node at the bottom represents the intracellular space. The capacitor represents the lipid bilayer cell membrane and the resistor in series with a battery represents the “leak current” that slowly returns V_M to a baseline resting value after stimulation with the electrode. With properly adjusted parameter values for the components, this circuit reproduces the fundamental passive electrical behavior of a neuron. Typical values for the components are on the order of 1 G Ω for the leak resistance (R_L), -70 mV for the leak potential (E_L), 1 pF for the membrane capacitance (C_M), and 1 nA for the stimulus current. A major simplifying assumption in this and similar models is that both the extracellular and intracellular spaces are isopotential.

Although useful for many purposes, this model does not describe the complex dynamics of the numerous membrane conductances in a typical neuron, of which the above-mentioned leak conductance is only the simplest.

Today, these different conductances are known to correspond to different types of membrane-spanning pores in the cell membrane called ion channels (Hille 2001). Ion channels are key players in a wide variety of physiological processes, especially those involving dynamic electrical activity. They are typically characterized by the type of ion that flows through them as well as by the factors that influence the degree

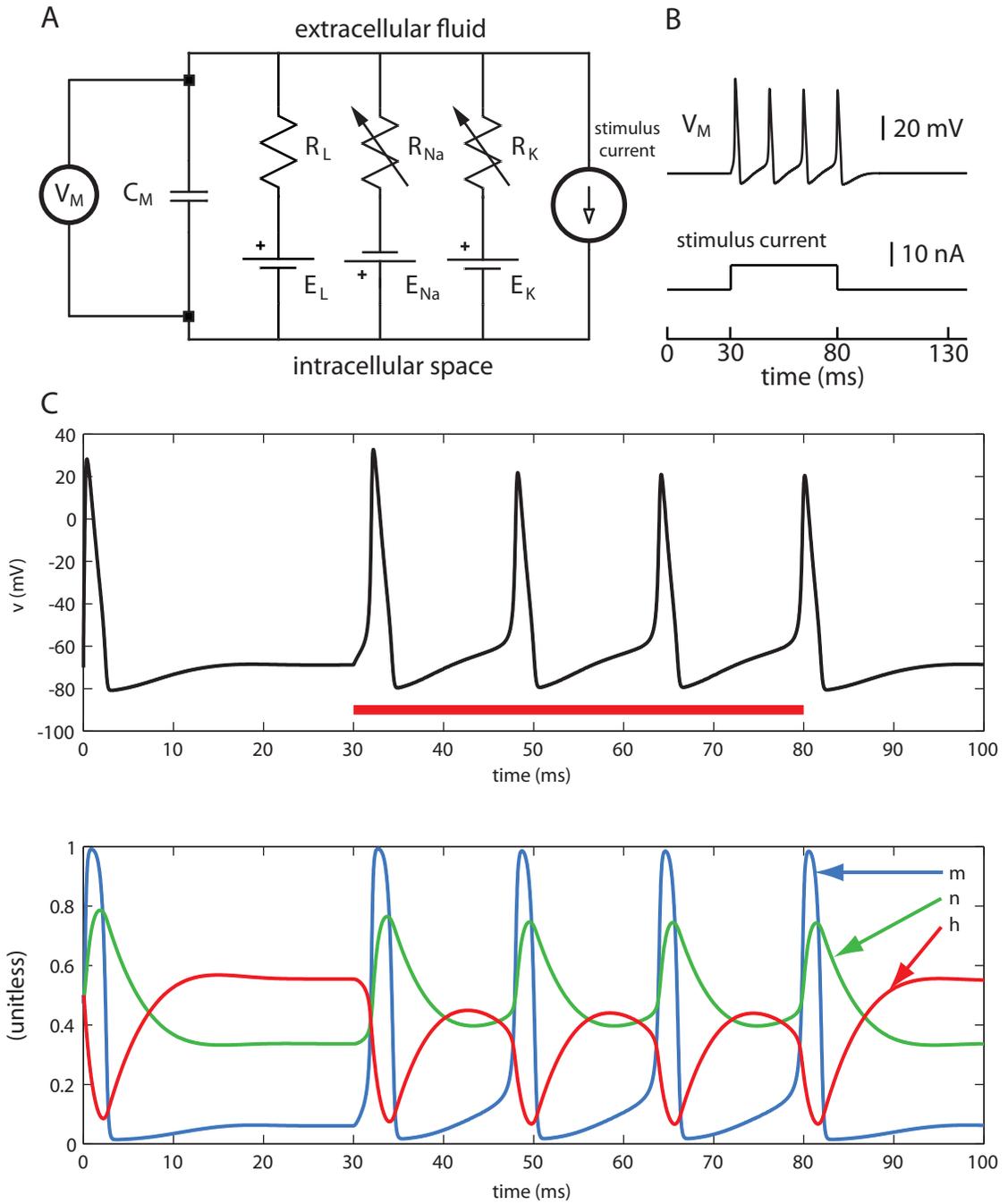
to which the channel admits that ion. For example, a voltage-gated sodium channel only admits sodium ions and the intrinsic rate of sodium ion passage through the channel depends on membrane voltage. The behavior of individual channel types and even individual channel molecules can be studied in isolation through different pharmacological and electromechanical techniques, including the type of simple “wet lab” experiment shown in Figure 2.1 (Kandel et al. 2000).

Even to this day, Lapicque’s experimental setup and circuit model remain useful tools for studying the physiology of single neurons. However, as noted above, his basic circuit as shown in Figure 2.2 does not explain or predict any of the more complex electrical behaviors displayed by neurons, such as action potentials. A major breakthrough in computational neuroscience was the detailed mathematical description of the primary conductances involved in action potential generation by Hodgkin and Huxley. In their series of papers from the 1950’s, they used an analog circuit to model the action potentials observed in the giant axons of squid (Hodgkin and Huxley 1952a-d, Hodgkin et al. 1952). Their work, for which they were awarded the 1963 Nobel Prize in Physiology or Medicine, remains highly influential. The equivalent circuit used by Hodgkin and Huxley, with components for sodium and potassium conductances in addition to the leakage conductance, is shown in Figure 2.3. In this case, the behavior of the nonlinear resistors is described with a complex system of coupled differential equations:

$$\begin{aligned}
\frac{dv}{dt} &= f(v, h, m, n, i) = -120m^3h(v - 45) - 36n^4(v + 82) - 0.5(v + 60) \\
\frac{dh}{dt} &= f(v, h) = 0.07e^{-\frac{1}{20}v - \frac{7}{2}}(1 - h) - \frac{h}{1 + e^{-\frac{1}{10}v - 4}} \\
\frac{dm}{dt} &= f(v, m) = \frac{1}{10} \frac{(v + 45)(1 - m)}{1 - e^{-\frac{1}{10}v - \frac{9}{2}}} - 4e^{-\frac{1}{18}v - \frac{35}{9}}m \\
\frac{dn}{dt} &= f(v, n) = \frac{0.1 \left(\frac{1}{10}v + 6 \right) (1 - n)}{1 - e^{-\frac{1}{10}v - 6}} - 0.125e^{-\frac{1}{80}v - \frac{7}{8}}n
\end{aligned} \tag{2.1}$$

where v is the observed membrane voltage, i is the stimulus current, and the hidden (unobservable) variables h , m , n are the gating parameters of the sodium and potassium conductances. Although this system of equations is typically not presented in its full form in this manner without extensive explanation, the intent here is to emphasize the complexity of the task faced by Hodgkin and Huxley and the magnitude of their achievement. For details, see the description in Hodgkin and Huxley (1952d) or the highly accessible introduction in Hoppensteadt and Peskin (2002).

Figure 2.3. The Hodgkin-Huxley model (next page). **A)** The basis for Hodgkin and Huxley’s 1952 model was a circuit similar to that of Lapique except for the conductances R_{Na} and R_K . These nonlinear conductances were not modeled with circuits explicitly, but with the complex system given in Equation 2.1. **B)** The addition of components representing voltage-gated sodium and potassium channels allows the circuit to capture more complex neural behavior such as these action potentials. **C)** This more complex behavior, such as the action potentials in the upper panel, emerges from the interaction of several parameters, including the gating variables h , m , and n as shown in Equation 2.1. The dynamics of these gating variables are shown in the lower panel on the same time axis as the membrane voltage. The red bar indicates the duration of application of an input current pulse.



Authors often cite Hodgkin and Huxley's work as a particularly impressive example of how human insight can distill raw data into a useful model. Hodgkin and Huxley had almost none of the knowledge of molecular biology and ion channels that we take for granted today, yet their model proved exceptionally powerful at explaining and predicting experimental observations (Häusser 2000). If artificial intelligence were capable of such extraordinary leaps of creativity and insight in the face of little or no mechanistic knowledge of the underlying physiology, it would be a very powerful tool in the neuroscientist's toolbox. Although action potentials are important, they are just one among the hundreds if not thousands of dynamical systems that would need to be modeled to develop a reasonably complete picture of single cell neurophysiology. These include the dynamical systems underlying synaptic transmission, vesicle cycling, axon and dendrite growth, apoptosis, and general cell metabolism (Kandel et al. 2000). This vast undertaking is of great interest not only for purely theoretical reasons, but for biomedical applications that require accurate neurophysiological models (Ellner and Guckenheimer 2006). A primary goal of this work is to shift as much of the burden of creating these models as possible from humans to computers.

Even if only ion channels are considered, hundreds of channel types are known in vertebrates alone and it is likely that hundreds more remain to be discovered, let alone reduced to useful models (Gabashvili et al. 2001). A means of automatically modeling ion channels based on easily obtained experimental data could be of great benefit to both basic research and applied areas including the design of neuromorphic circuits (Douglas et al. 1995, Sicard et al. 1999), drug design (Noble 2008), and other biomedical applications requiring sophisticated physiological models (Iniewski 2008,

Smith 2010).

Different fixed-form, EC-based approaches to the automated modeling of ion channels have been proposed recently. These include models based on kinetic Markov models (Menon et al. 2009) and others based on systems of differential equations (Gurkiewicz and Korngreen 2007, Kherlopian et al. 2011). In contrast, the goal of the research described in this chapter is to take the first steps toward free-form, data-driven modeling of ion channels. Drawing inspiration from the work of Lapicque, Hodgkin, and Huxley described above, the central working hypothesis of this chapter will be that analog electrical circuits are a natural representation for this task and may facilitate the modeling process as compared to other representations, such as systems of differential equations. In addition, EC is a powerful and well established technique for optimizing the design of circuits in a free-form context (Koza 1992, Koza et al. 1996a, Koza et al. 1997, Keymeulen et al. 2000). The natural combination of these ideas considered here is the application of analog circuit evolution to the automated modeling of ion channels. An important feature of this modeling approach is that it is data-driven, that is, only the type of data that can easily be obtained in a typical neuroscience wet lab are used to guide the process and no additional domain specific knowledge needs to be encoded.

Initial Experiments

For initial experiments, the aim was to apply the established technique of analog circuit evolution (Koza et al. 1997) to the task of creating equivalent circuit models for simple neurophysiological systems. The systems were measured through very simple, direct experiments of the kind that can easily be performed in a typical neuroscience “wet” lab. It was found that with only this input data and with only the simplest electrical components such as resistors and capacitors, circuit evolution can automatically generate accurate equivalent circuits for the complex ion channel systems of the type studied by Hodgkin and Huxley.

Circuit Evolution

A simple version of Koza’s circuit evolution technique was employed (Koza et al. 1996a, Koza et al. 1997, Kim et al. 2010). Initially, a population of candidate equivalent circuits was created with two randomly selected, randomly connected electrical components. These randomly chosen components were placed in a variable portion of an otherwise invariant embryonic circuit as shown in Figure 2.4. The fitness of individuals in this random population was evaluated by translating each individual into an equivalent Spice netlist, simulating its behavior using NG-Spice20¹, and comparing this behavior to that of the target neurophysiological system (see **Neurophysiological Data** below). A steady-state population-updating method was used in which the least fit half of the population was subject to mutation before the

¹ <http://ngspice.sourceforge.net>

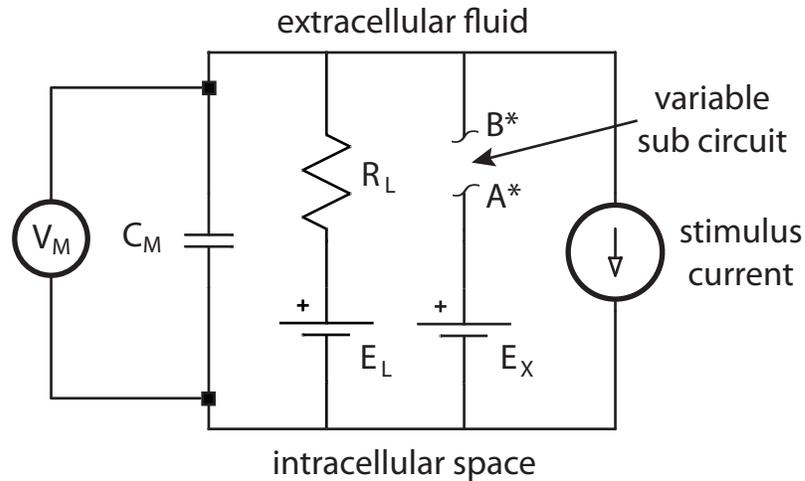


Figure 2.4. The embryonic circuit. This circuit is essentially a model of a cell membrane with no ion channels embedded except for those responsible for the leak current. It was the task of evolution to find a sub circuit such that the entire circuit reproduces the behavior of a target ion channel. The nodes labeled A* and B* were used to identify connection points for the evolved sub circuits shown in **Results** below. The intrinsic driving force for the modeled ion channel (E_X) was included as this is easy to measure experimentally and is not difficult to model, unlike the portion of the circuit that would be equivalent to the nonlinear resistors in Figure 2.4.

next generation of fitness evaluation and selection occurs. No recombination was used. For all results reported, population sizes of 96 were evolved for 10,000 generations.

Circuits were represented with a direct schematic-based encoding, in which components and their connections are stored as flat lists (Floreano and Mattiussi 2008). Seven low-level electrical components were used as the raw material or building blocks with which evolution operates. These components and the allowed ranges of their variable parameters are shown in Table 2.1. All models were default NG-Spice20 models with the exception of the MOSFET models, which were generously provided by Mario Simoni (Simoni et al. 2004). From each circuit that was chosen to be mutated, one randomly selected component underwent one of eight

Table 2.1. Components and parameter ranges used in circuit evolution.

Component	Parameter range
inductor	$L = 1 - 1 \times 10^9$ kH
capacitor	$C = 1 - 1 \times 10^9$ fF
resistor	$R = 1 - 1 \times 10^9$ k Ω
diode	D (not a variable parameter)
EMF	$V = 0 - 20$ V
p-type MOSFET	length = 10 μ M, width = [5, 10, 20] μ M
n-type MOSFET	length = 10 μ M, width = [5, 10, 20] μ M

possible mutations:

- 1) Parameter change: the numerical value of the component's parameter is assigned a new value drawn from a uniform random distribution over the range of allowed values (Table 2.1).
- 2) Type change: the component type is changed to another type chosen randomly.
- 3) Parallel addition: a new component is added in parallel to the randomly chosen component already present in the circuit. The component type and parameters are randomly chosen, except that the type must be different than the component already present in the circuit.
- 4) Serial addition: a new component is added in serial to the randomly chosen component already present. Otherwise, this mutation is the same as parallel addition.
- 5) Deletion: the randomly chosen component is deleted.
- 6) Grounding: the randomly chosen component is connected to ground.
- 7) Replacement: the randomly chosen component is replaced with another component, possibly of the same type.
- 8) Addition: instead of selecting a component already present in the circuit, two nodes

are randomly selected and a new component is created connecting the two nodes.

Neurophysiological Data

In some of the 1952 Hodgkin-Huxley studies, the contribution of the primary sodium and potassium currents to the squid giant axon action potential were studied in isolation using pharmacological techniques (Hodgkin et al. 1952, Hodgkin and Huxley 1952 a-d). These experimental conditions were recreated in the present work. Using the NEURON 7.0 simulation environment (Hines and Carnevale 2001), default HH sodium and potassium ion channels were inserted into a single compartment. Such a simulated “cell” has the equivalent circuit shown in Figure 2.3. The conductance values for sodium were then set to 0 to simulate pharmacological isolation of the voltage-gated sodium channels. The model cell was then stimulated with a small step current of 1 nA. The membrane voltage response measured before, during, and after stimulation represents the target input/output behavior of idealized Hodgkin-Huxley voltage-gated potassium ion channels. Evolution was tasked with finding a variable sub circuit as in Figure 2.4 that behaved in the same way as the Hodgkin-Huxley nonlinear potassium resistance R_K . The voltage-gated sodium channel was targeted for evolution in a similar manner except that the conductance values for potassium were set to 0.

To evaluate the fitness of an evolved circuit, it was stimulated with a step current of 1.0 nA in NG-Spice20 and the resulting membrane voltage time series was compared with the simulated membrane voltage time series obtained from NEURON. Spice simulation data were recorded at 0.1 ms resolution for 40 ms and each of those

401 time points were compared with the corresponding time points from the target NEURON data. Error was defined as the sum of the absolute errors at each time point. To reduce fitting errors due simply to voltage offset or scaling, both of the membrane voltage time series involved in the comparison were normalized to the range 0 - 1. Results shown below are plotted on this relative voltage scale.

Results

Evolution produced circuits that mimicked the behavior of both the idealized Hodgkin-Huxley voltage-gated potassium channel and voltage-gated sodium channel. Figure 2.5a shows the response of a typical evolved circuit for the voltage gated potassium channel compared with the target potassium channel response. Figure 2.6 is the schematic for this evolved circuit. Similarly, Figure 2.5b shows the response of a typical evolved circuit for the voltage gated sodium channel and Figure 2.7 is the schematic for this evolved circuit. Performance of the evolutionary algorithm for potassium channel circuit evolution and for sodium channel circuit evolution are shown in Figure 2.8a and Figure 2.8b, respectively.

Discussion

In this initial set of experiments, the goal was to determine whether analog circuit evolution can be used to automatically construct equivalent circuits for neurophysiological systems. To test this, two systems were used: voltage-gated sodium and potassium ion channels. Results show that evolved circuits can reproduce the behavior of the modeled systems to a large degree. However, additional work will

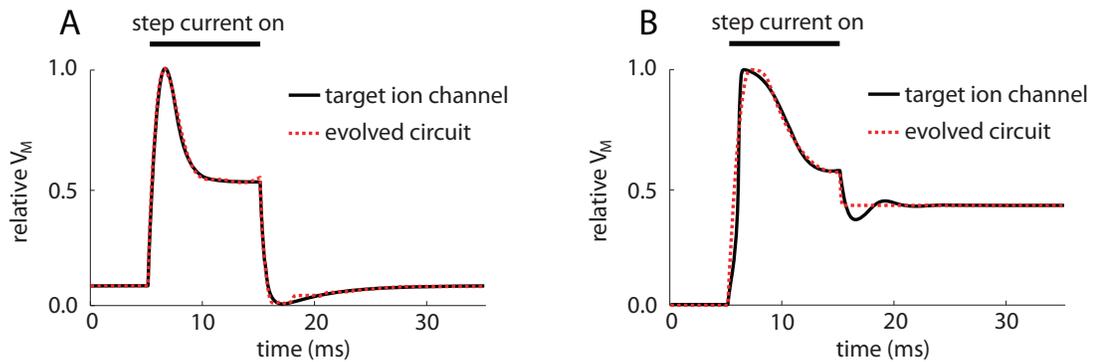


Figure 2.5. Step response of voltage-gated potassium (A) and sodium (B) channels. The step currents used to probe the target ion channel and the evolved circuit were both 1 nA in amplitude and 10 ms in duration. The responses of the idealized channels obtained with NEURON are shown with solid lines and the responses of the embryonic circuit plus evolved sub circuits are shown with dashed lines.

be required to confirm that the evolved equivalent circuits are accurate and robust models of the sodium and potassium ion channels across a wide range of realistic physiological conditions. The behavior of the evolved sodium channel circuits in particular deviate somewhat from the desired behavior. One possible explanation is that Hodgkin-Huxley voltage-gated sodium channel dynamics involve both activation and inactivation processes. This contrasts with the dynamics of voltage-gated potassium channels, which can be modeled with only activation. It is possible that recombination, which was not used in these initial experiments, would allow a promising sub circuit for only the activation or only the inactivation process to replicate and then differentiate during evolution. Future work is also required to investigate the performance of more sophisticated representations for analog circuit evolution such as Analog Genetic Encoding (Mattiussi and Floreano 2007) and graph grammar-based approaches (Das and Vemuri 2009).

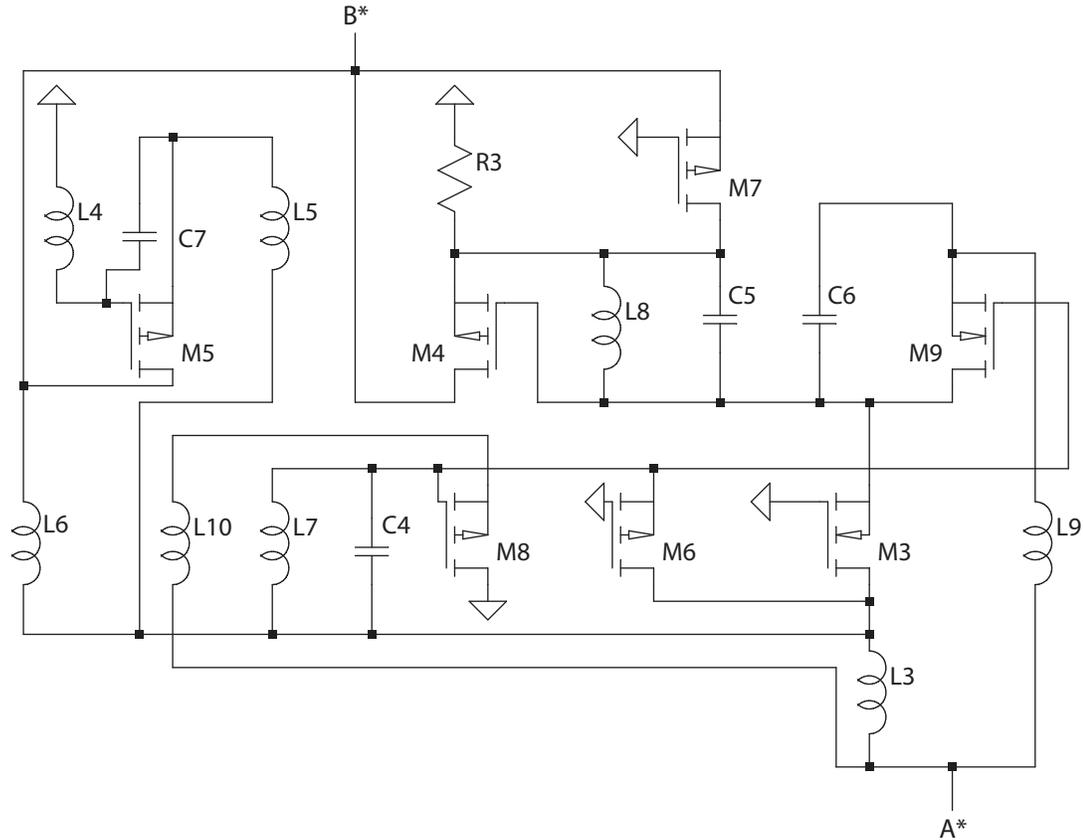


Figure 2.6. Schematic of an evolved potassium channel circuit. No post processing or simplification of the circuit was performed. To conserve space, only the evolved variable sub circuit is shown and the nodes labeled A* and B* connect with the embryonic circuit as shown in Figure 2.4. Component parameter values are shown in Table 2.2.

In these initial experiments, step current inputs to the respective systems were used to characterize their output behavior, but it will be important to confirm that other types of input/output pairings can be reproduced by the equivalent circuits as well. Coevolution of input functions and circuit models is one possible way to maintain selection pressure for robustness (Torresen 2002). Such a coevolutionary approach was tried in Kim et al. 2010 with good results. A longer-term goal is to move beyond

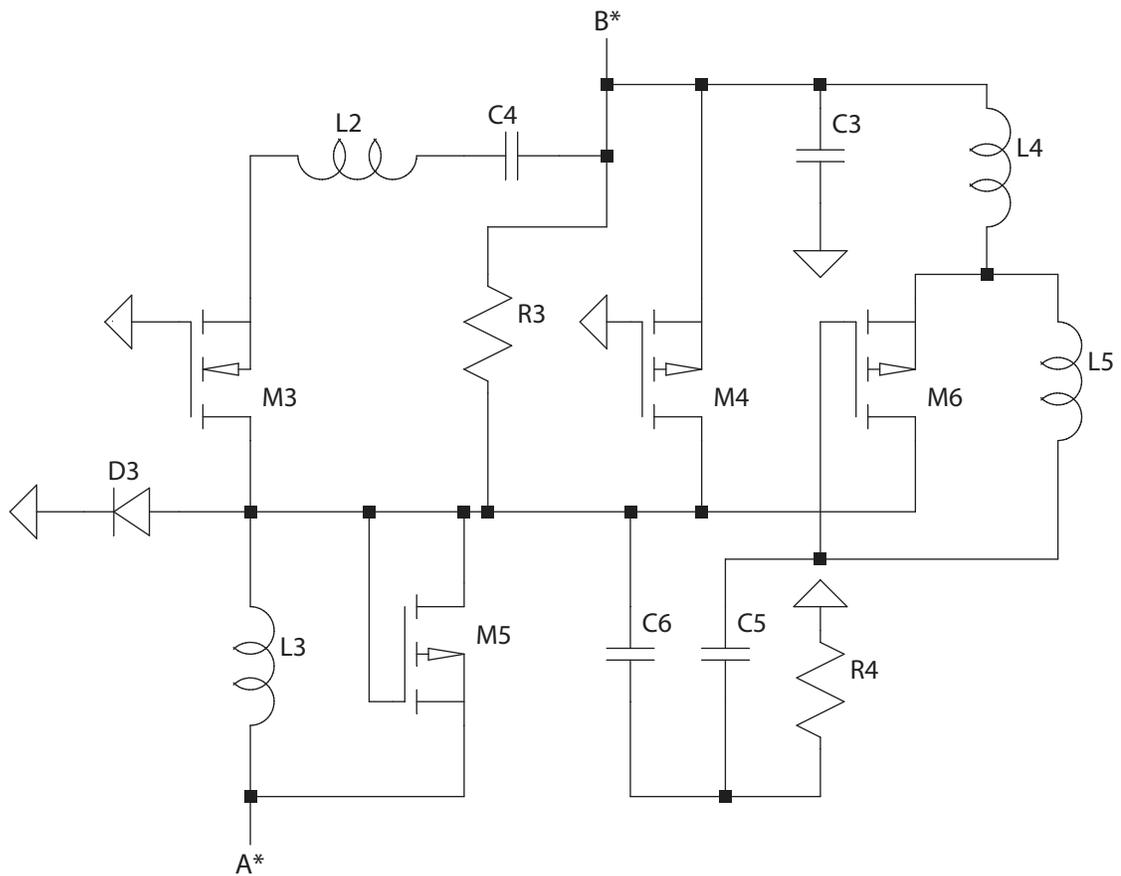


Figure 2.7. Schematic of an evolved sodium channel circuit. As in Figure 2.6, no post processing or simplification of the circuit was performed and only the evolved variable sub circuit is shown. Component parameter values are shown in Table 2.3.

ion channels simulated in NEURON and to use circuit evolution as one component of a closed-loop automated experiment system as proposed in Bongard and Lipson 2007. Active learning could be used to probe a physical system of interest, such as a single neuron as shown in Figure 2.1. The inputs that cause the most disagreement between predicted and observed outputs would be used to evolve equivalent circuits. The accuracy and robustness of the model could then be refined by further cycles of active learning probes and circuit evolution (Figure 2.9).

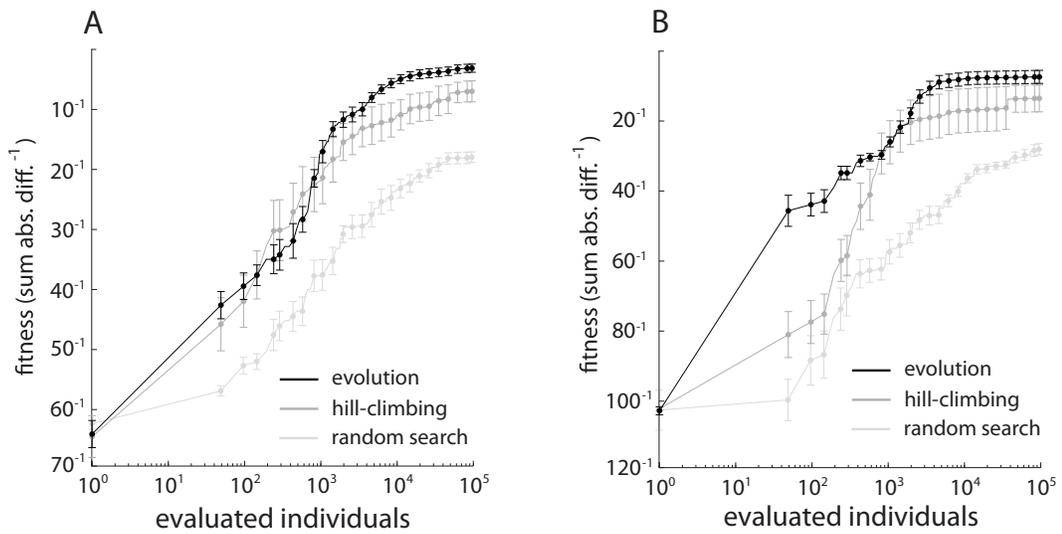


Figure 2.8. Fitness as a function of the number of evaluated circuits. Potassium (A) and sodium (B) model circuits were searched for with an evolutionary algorithm (black), a simple hill-climbing algorithm (dark gray), and random search (light gray) for ten independent trials each. The mean fitness across trials of the best circuit model is shown as a dot with error bars representing mean fitness \pm standard error of the mean.

The use of circuit evolution to model neurophysiological systems has potential applications beyond those presented here. For example, the design of neuromorphic circuits that interface living tissue with electrical components is of growing importance in the medical field, yet as with the design of all novel analog circuits, neuromorphic circuit design is largely performed by hand and then only by experienced electrical engineers (Douglas et al. 1995, Smith 2010). The complexity of hand-designed hardware implementations of the Hodgkin Huxley model attests to the difficulty of the task (Kohno and Aihara 2007, Simoni et al. 2004). The assistance of circuit evolution could be invaluable for this application, especially as neuromorphic circuits customized to the needs of individual patients become a reality (Sicard et al. 1999). Although discrete components were used in these initial experiments, the

Table 2.2. Components and parameter values for the evolved potassium channel circuit.

Component	Parameter value
L3	3.50×10^2 kH
M3	width = 10 μ M
M4	width = 5 μ M
R3	3.93×10^6 k Ω
M5	width = 10 μ M
M6	width = 20 μ M
L4	1.79×10^5 kH
M7	width = 5 μ M
C3	1.01×10^3 fF
L5	4.58×10^5 kH
L6	3.48×10^6 kH
L7	4.38×10^2 kH
C4	2.34×10^3 fF
M8	width = 10 μ M
M9	width = 5 μ M
L8	1.48×10^1 kH
L9	1.34×10^4 kH
C5	3.14×10^2 fF
C6	1.90×10^1 fF
L10	1.64×10^2 kH

approach proposed here is applicable to the design of integrated circuits, which would almost certainly be used in any practical neuromorphic system. One drawback of the equivalent circuit approach in general is the lack of a mechanistic correspondence to the underlying neurophysiological system. However, many applications may not require an understanding of the underlying biology in detail. For example, simulations used in drug design might benefit from an accurate and easily-produced model of a neurophysiological system with only the requirement that certain observed behavior be reproduced (Noble 2008).

Table 2.3. Components and parameter values for the evolved sodium channel equivalent circuit.

Component	Parameter value
L3	3.52×10^3 kH
M3	width = 5 μ M
R3	2.16×10^6 k Ω
M4	width = 10 μ M
M5	width = 10 μ M
C3	7.83×10^1 fF
R4	5.93×10^7 k Ω
C4	3.24×10^3 fF
M6	width = 5 μ M
C5	7.07×10^8 fF
C6	4.03×10^1 fF
L4	2.21 kH
L5	2.76×10^4 kH
L6	3.48 kH

Given the inherently electrical nature of systems in neurophysiology, the electrical components in equivalent circuits are likely to be natural building blocks with which evolution can construct accurate models. However, equivalent circuits are only one of many ways to represent neurophysiological systems. Others include differential equations as in Hodgkin and Huxley's work, Markov kinetic models (Hawkes 2003), and artificial neural networks (Lockery et al. 1989). Evolution-based search has now been successfully applied to the optimization of models using all those representations (Gurkiewicz and Korngreen 2007, Menon et al. 2009, Stanley and Miikkulainen 2002). The more general idea of using software and hardware-based techniques to automatically study systems in biology, chemistry, and physics has also had noteworthy successes (King et al. 2009, Lindsay et al. 1980, Schmidt and Lipson 2009). It is likely that many nontrivial scientific tasks currently requiring significant

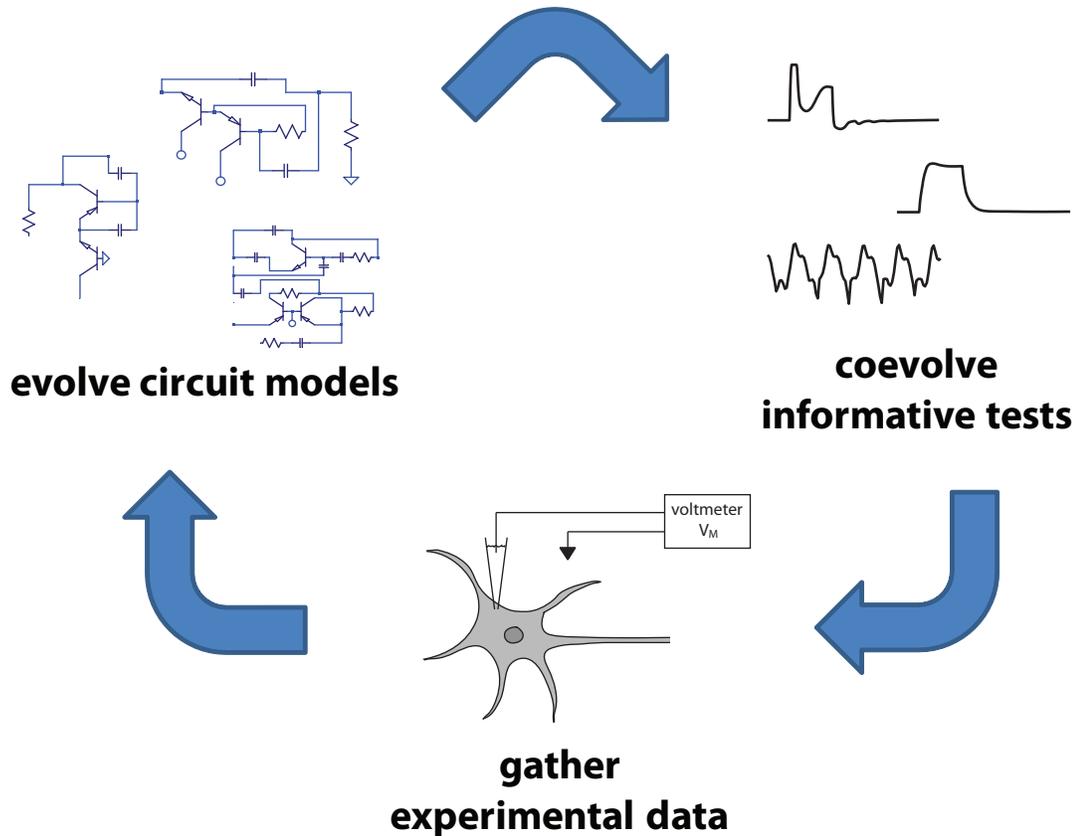


Figure 2.9. A closed loop experimental system for modeling neurophysiological systems with analog circuits. To start, an arbitrary current input can be used to probe a neuron (bottom). The resulting output behavior is then used as the target for circuit evolution (left). After a population of candidate models is evolved, test inputs are coevolved (right). For this coevolutionary process, the fitness of a test is determined by using it as input to the population of circuit models and assessing the corresponding outputs. The more disagreement there is between those outputs, the more useful the test will be at disambiguating between the competing circuit models. The most informative test coevolved in this manner is then used to probe the neuron (bottom), and the experimental loop continues.

human effort will begin to be augmented by sophisticated artificial intelligence and robotic systems in the near future.

Evolving Robust Circuit Models

Although the results obtained with the preliminary experiments described above were promising, they were limited in the sense that the evolutionary algorithm used was very simple and evolved circuits were not rigorously tested and compared with the target systems under a variety of different stimulus conditions. To address these shortcomings, a further set of experiments was performed as described in this section. For these experiments, the performance of several different evolutionary techniques was compared. Most of these techniques have not been previously applied to circuit evolution. In addition, these experiments took into account the need to robustly model ion channel behavior under many physiological conditions instead of just one and how this need can be balanced against the computationally expensive Spice simulations necessary for nonlinear analog circuit evolution.

Neurophysiological Data

The methods used were substantially similar to those described in *Initial Experiments* above, except where otherwise noted. To obtain the “ground truth” physiological data necessary to assess the fitness of evolved circuit models, the conditions of the 1952 Hodgkin-Huxley studies were again recreated. In some of those studies, the contribution of sodium and potassium currents to the squid giant axon action potential were studied by isolating the respective currents using pharmacological techniques (Hodgkin and Huxley 1952a-d, Hodgkin et al. 1952). Using the NEURON 7.0 simulation environment (Hines and Carnevale 2001), an “empty neuron” was

generated with one compartment that is precisely modeled by the circuit shown in Figure 2.2b. Default HH potassium ion channels were added and the cell was then stimulated with small currents while measuring the cell's membrane voltage. The membrane voltage response obtained in this way represented the target input/output behavior of an idealized Hodgkin-Huxley voltage-gated potassium ion channel. The evolutionary algorithm was tasked with creating a circuit that displays the same input/output behavior as this potassium ion channel.

To evaluate the fitness of a candidate circuit produced during evolution, the circuit was stimulated with the same currents used to stimulate the simulated cell in NEURON. The resulting membrane voltage time series was then compared with the simulated membrane voltage time series obtained from NEURON. Circuit behavior was recorded at 0.1 ms resolution for 40 ms and each of those 401 time points were compared with the corresponding time points from the target NEURON data. The inverse of fitness, or error, was defined here as the sum of the absolute differences in the two voltage traces at each time point. To eliminate error due simply to voltage offset or scaling, both of the membrane voltage time series involved in each of these comparisons were normalized to the range 0 - 1. Results below are plotted on this relative voltage scale.

To reduce to a manageable size the infinite space of potential electrophysiological conditions under which the evolved circuit must accurately reproduce the behavior of the potassium ion channel, the universe was defined as consisting of six possible conditions. These different electrophysiological conditions correspond to different time domain waveforms of a current pulse applied to the cell.

Initially, random waveforms were tried for this purpose, but it was found that standard engineering waveforms such as step and sinusoidal pulses gave the most meaningful results. As shown in Figure 2.10, the six stimulus waveforms used for “training” the circuit models were three step pulses of amplitude 0.3 nA, 1.0 nA, and 3.0 nA and three 500 Hz sinusoidal pulses with amplitudes of 0.3 nA, 1.0 nA, and 3.0 nA.

Circuit Evolution

A variation on Koza’s analog circuit evolution technique was employed (Koza 1992, Koza et al. 1996a, Koza et al. 1996b, Koza et al. 1997). Initially, a population of candidate circuits was created with between two and twenty randomly selected, randomly connected analog electrical components. These randomly chosen components were placed in a variable portion of the otherwise invariant embryonic circuit shown in Figure 2.4. Circuits were represented with a direct schematic-based encoding, in which components and their connections are stored as flat lists (Floreano and Mattiussi 2008). Seven low-level electrical components were used as the raw material or building blocks with which evolution operates. These components and the ranges of their parameter values are shown in Table 2.1.

The fitness of candidate circuit models was evaluated by translating each individual into an equivalent Spice netlist, simulating its behavior using NG-Spice22², and comparing this behavior to that of the target ion channel under the same stimulus conditions. All component models used were default NG-Spice22 models with the exception of the MOSFET models, which were generously provided by Mario Simoni

² <http://ngspice.sourceforge.net>

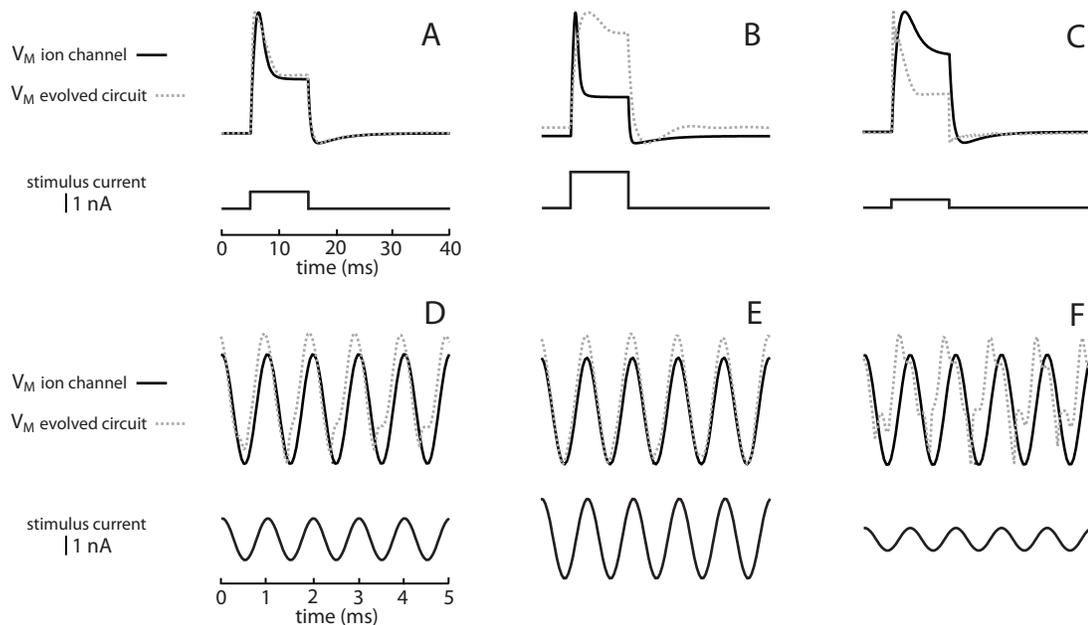


Figure 2.10. Comparison of target ion channel behavior and evolved model circuit behavior. **A)** The 1 nA input current shown was used to stimulate a HH potassium ion channel in NEURON while measuring membrane voltage. The resulting membrane voltage is shown as the solid black trace above the stimulus current. This voltage trace was used as the target response to gauge the fitness of circuits stimulated with the same 1 nA input current. After 1200 generations of evolution using the Single Objective technique, the output voltage response of the circuit with behavior that most closely matched the target response was plotted as the dashed gray line. **B-F)** The output voltage responses of that same circuit to five other stimulus currents are shown. Only a portion of the 40 ms long 500 Hz sinusoidal currents in **D-F)** are shown for clarity. The much greater deviation from target behavior can be seen with these targets that were not used to evaluate fitness during evolution.

and designed for special use in neuromorphic circuits (Simoni et al. 2004).

The performance of six different multiobjective and coevolutionary techniques were evaluated. These six techniques are referred to below as the “primary evolution techniques.” In addition, four secondary variations on each of the six primary techniques were employed, for a total of 24 distinct analog circuit evolution algorithms. The primary techniques are discussed first below, followed by the

“secondary techniques.” In all cases, the vast majority of the computational effort required for evolutionary runs was devoted to Spice simulations and any differences in the efficiency between other portions of the 24 algorithms were negligible by comparison.

The six primary evolution techniques will be referred to as: 1) Single Objective, 2) Active Coevolution, 3) Multiobjective Simple Fitness, 4) Multiobjective Pareto Fitness, 5) Incremental Simple Fitness, and 6) Incremental Pareto Fitness. The Single Objective technique involved a single stimulus waveform used to obtain the target ground truth input/output behavior of the cell in NEURON. This input/output behavior was then used as the sole target for an evolutionary search through analog circuit space and as the basis for evaluating a circuit’s fitness. Each generation, the population was subjected to tournament selection with tournaments of size 2 and replacement until 128 parents were selected. These 128 parents were randomly chosen to undergo mutation or crossover until 128 offspring were generated. The population of the subsequent generation then consisted of 256 individuals: the 128 parents plus the 128 offspring. Pilot experiments suggested that mutation was generally more effective than crossover at promoting diversity without creating invalid and unsimulatable circuits, and so the probability that selected parents would undergo crossover was set to 10% and the mutation probability was set to 90%. Simple one point crossover was used and mutation involved the random selection of one component in the circuit that then underwent one of eight possible mutations. The eight possible mutations are the same as those described in *Initial Experiments* above.

The Active Coevolution technique was inspired by the closed experimental loop concept proposed in Bongard and Lipson 2007. Instead of one static target input/output relationship used as the basis for evaluating circuit fitness, the target antagonistically coevolved in order to maintain selection pressure on the circuit population. One initial target from the universe of six possible training targets was randomly chosen at the start of a run and evolution proceeded as in the Single Objective technique above for N generations. At that point, the behavior of each circuit in the population was probed with all six possible input currents. The input type that produced the greatest disagreement in circuit output behavior among the population of circuits was then selected as the single evolutionary target for the next N generations of evolution, whereupon another input type was chosen, and so on. As the cycle repeated, the same target could be chosen more than once. The rationale behind this type of active learning through coevolution of stimuli is that the best stimulus to use as a target at any point in the evolutionary process is expected to be the one that is most helpful in disambiguating between competing models in a population of models (Bongard and Lipson 2007).

In the Multiobjective Simple Fitness and Multiobjective Pareto Fitness techniques, all six training targets in the universe of possible targets were used to evaluate the fitness of every circuit in every generation. Although this is an obvious strategy to enhance the robustness of the circuit models, using all six targets requires six separate Spice evaluations for each circuit instead of only one as for the Single Objective and Active Coevolution techniques. The ideal tradeoff between increased computational effort per generation and the evolution of potentially more robust

circuit models was not clear a priori. In the Multiobjective Simple Fitness technique, the fitness of a circuit was quantified as the sum of the differences between the circuit's behavior and the target behavior for each of the six targets. The Multiobjective Simple Fitness technique was otherwise the same as the Single Objective technique above. For the Multiobjective Pareto Fitness technique, the six fitness cases were considered to be distinct, equally important objectives. As a result, the outcome of tournaments during tournament selection could no longer be decided based on the simple fitness-derived ranks of the individuals in the population. Instead, the NSGA-II multiobjective optimization algorithm was used to optimize for all six fitness objectives simultaneously (Deb et al. 2000). The rationale for this approach was that treating the fitness cases as separate fitness objectives might maintain population diversity more effectively than using a single, monolithic fitness objective as in the Multiobjective Simple Fitness technique.

The Incremental Simple Fitness and Incremental Pareto Fitness techniques were inspired by previous observations that a single target input/output relationship can be modeled relatively easily, but that the modeling of multiple targets seems to be disproportionately difficult (Figure 2.10). For both these incremental techniques, an evolutionary run started with a single target. After N generations a second target was added, after N more generations a third target was added, and so on. The next target to be added was chosen every N generations using the same criteria as in the Active Coevolution technique above, although any given target could only be added once. In most of the experiments shown below, all six targets were eventually added after $6 \cdot N$ generations. For the Incremental Simple Fitness technique, when targets were added,

they were combined into one fitness score as in the Multiobjective Simple Fitness technique above. For the Incremental Pareto Fitness technique, when targets were added, they were added as separate objectives and multiobjective optimization was used as in the Multiobjective Pareto Fitness technique above.

Recent work has shown that using genotypic age as an explicit metaobjective in addition to fitness can be a powerful yet simple way to reduce bloat and maintain diversity in an evolving population (Schmidt and Lipson 2010a-b). A standard multiobjective algorithm such as NSGA-II can be used to perform this Age-Fitness Pareto optimization (Deb et al. 2000). Here, four variations on this idea were employed: 1) Fitness Only optimization, 2) Age-Fitness optimization, 3) Size-Fitness optimization, and 4) Age-Size-Fitness optimization. Age was defined here as the number of generations the individual has been present in the population, which was to be minimized. Size was defined simply as the number of components in the circuit and as with age, it was to be minimized. An individual descended from another individual, whether by mutation or recombination, inherited that parent's age. In the case of recombination, the age of the older parent was inherited. To provide a steady flow of low-age genotypes, one new circuit was added to the population each generation. This new individual was randomly constructed in the same way as all 256 members of the initial population.

When one of these four secondary evolution techniques was combined with one of the six primary evolution techniques above that uses multiple fitness objectives, the additional objectives were simply added as equally weighted, independent objectives. For example, in the Multiobjective Pareto Fitness technique with Age-

Size-Fitness optimization, there were eight separate, equally important objectives to be optimized by the NSGA-II algorithm: age, size, and six fitness objectives.

Results

The ability of analog circuit evolution to reverse engineer the Hodgkin-Huxley potassium ion channel was tested by performing 10 trials for each of the 24 circuit evolutionary algorithms. Each trial lasted for a total of 7200 circuit evaluations. Stimuli were changed every 1200 generations for Active Coevolution and added every 343 generations for the incremental algorithms. All population sizes were 256 as described above. These results are shown in Figure 2.11. The error plotted in Figure 2.11 is the sum of the errors on each of the six training stimuli, regardless of which of the six stimuli were used as targets for that evolutionary algorithm. As a reference for interpreting the meaning of the error values, the total error for all six stimuli for the circuit shown in Figure 2.10 is approximately 350, or 58.3 per stimulus on average. Age-Fitness optimization was found to perform as well or better than any other secondary evolution technique for each of the six primary evolution techniques. When comparing among the six primary evolution techniques that utilized Age-Fitness optimization, the overall error measure after 7200 fitness evaluations was lowest for the Multiobjective Simple Fitness and Incremental Simple Fitness algorithms, as summarized in Figure 2.11.

Because the experiments described so far were essentially the results of training circuit models using one or more of the six stimulus types shown in Figure 2.10, it was also important to evaluate the evolved models on test data not used in the

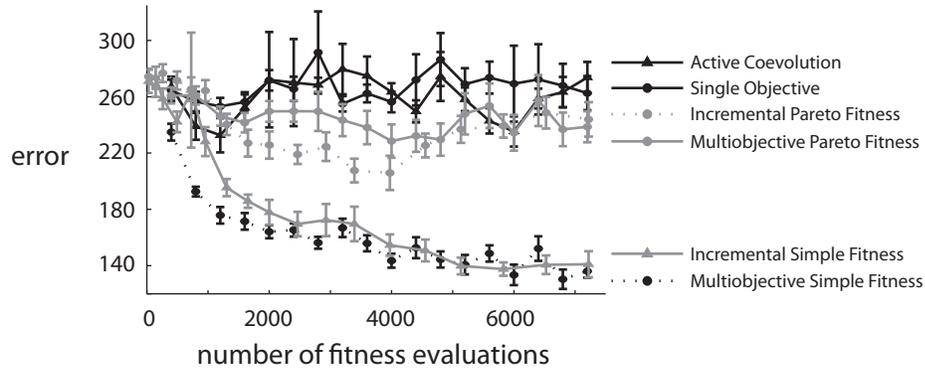


Figure 2.11. Error vs. number of fitness evaluations for each of the six primary evolution techniques, each of which was combined with Age-Fitness optimization as the secondary algorithm.

training process. Figure 2.12 shows that the evolved circuits generalize to such previously unseen test data and can accurately reproduce the potassium ion channel’s response to these novel stimuli. For this experiment, the best circuit evolved during one run of Incremental Simple Fitness was compared with the best circuit evolved during one run of Incremental Pareto Fitness. These best circuits were defined as those with the lowest error on the six training targets in their respective populations at the end of the run.

A simple approach to managing the problem of overfitting is to use some stimuli as validation data and to stop the training process early, that is, when error on the validation data begins to increase (Bishop 2006). If such an approach were successful, it would greatly decrease the number of fitness evaluations needed to evolve circuits with good generalization as compared to the other methods that involve potentially evaluating the fitness of an evolved circuit for several stimuli. Although this idea was not tested exhaustively, pilot experiments suggested that simple early stopping would not substantially improve results (Figure 2.13).

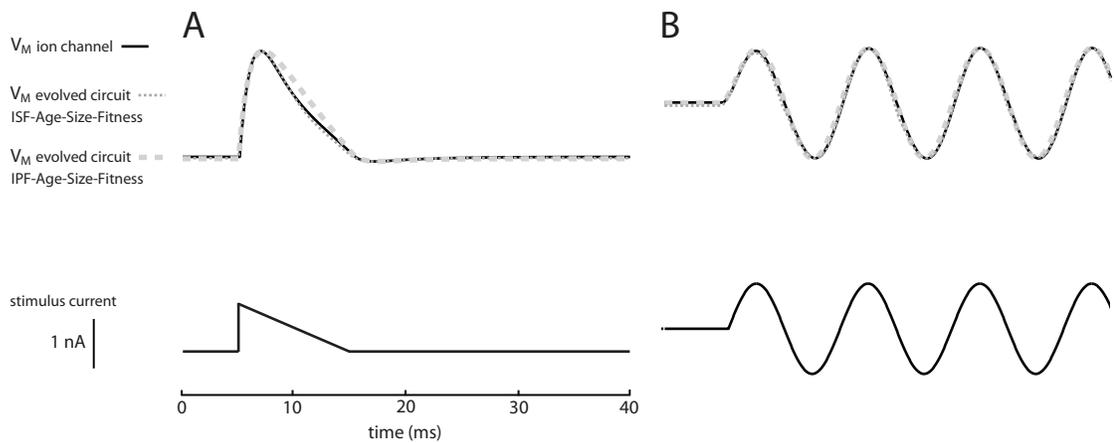
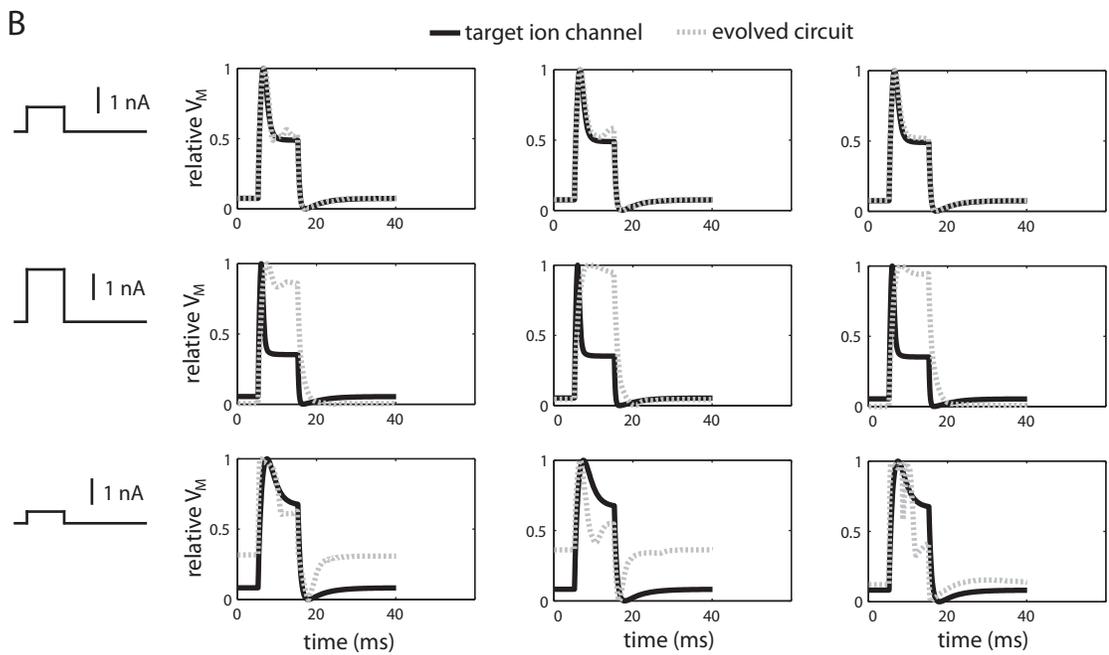
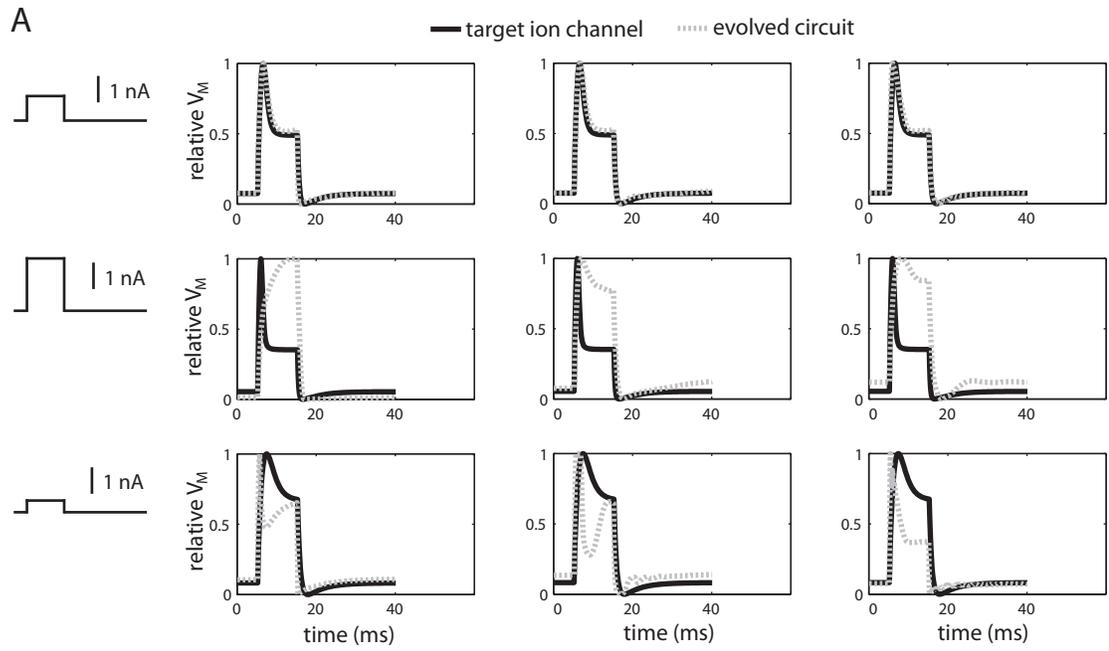


Figure 2.12. Performance of evolved circuits on previously unseen test data. **A)** Response of the target ion channel, one of the best evolved Incremental Simple Fitness circuits, and one of the best evolved Incremental Pareto Fitness circuits to a previously-unseen ramp stimulus current. **B)** Same as in A except the responses are to a previously unseen 100 Hz sinusoid stimulus current. Time scales are the same in panel A and panel B.

Discussion

Formally characterizing the complete input/output behavior of a nonlinear dynamical system such as the analog circuits or ion channels in this work would be feasible with enough computational effort (Hedrich and Barke 1995). This suggests that the fitness of a candidate analog circuit during evolution could be exhaustively evaluated by determining how well its output behavior matches the target system's output behavior for every possible input that the systems could receive. However, it would not be practical to include such analysis as part of the fitness calculations in circuit evolution

Figure 2.13. Overfitting of potassium channel circuit models (next page). **A)** The three columns of membrane voltage plots show the response of three different circuits to the three different stimuli in the first column, but only the 1 nA stimulus in the first row was used to evaluate fitness during evolution. **B)** Similar to panel A except that the circuits were evolved with early-stopping. Evolution was stopped when error on the other stimuli (2 nA and 0.5 nA) increased, but these other stimuli were not used to evaluate fitness.



without considerably more computing power than that provided by an ordinary workstation or cluster.

The best practical alternative for the reverse engineering application studied in the present work appeared to be the selection of a few different training stimuli to characterize the target neurophysiological system and evolved circuits. From a neuroscientist's point of view, the most natural comparison between ion channel behavior and an evolved circuit model would be the membrane voltage measured in the time domain in response to a stimulus current as employed here (Hille 2001, Kandel et al. 2000). However, it is possible that other means of characterizing and comparing the dynamic behavior of these systems can more efficiently guide evolution.

Circuits in this work were represented as a genotype with one gene for each component, where each gene included a sizing parameter and two or more connection parameters. More sophisticated representations such as Analog Genetic Encoding (Mattiussi and Floreano 2007) or a graph grammar-based representation (Das and Vemuri 2009) could potentially improve results, but to date, only limited work has been done performing systematic comparisons of different representations for analog circuit evolution (Zebulum et al. 1998). Also, 7200 fitness evaluations and population sizes of 256 as used here are very small when compared to for example, the experiments by Koza and colleagues (Koza 1992, Koza et al. 1996a, Koza et al. 1996b, Koza et al. 1997). However, here the concern was primarily with comparing different evolutionary algorithms as a prelude to larger experiments that would use only one or a smaller number of algorithms.

The secondary evolution techniques that attempted to optimize size (Fitness-Size and Age-Size-Fitness) generally performed worse than Fitness or Age-Fitness techniques. The exception to this was for the Single Objective and Active Coevolution techniques, the two techniques that relied on only one training target at a time. One explanation for these observations is that the use of size as a metaobjective is particularly effective at reducing bloat (Schmidt and Lipson 2010a-b), and smaller circuits means that they are better at generalizing to unseen targets. This would be a relative advantage when there is only one training target as in the Single Objective and Active Coevolution algorithms, but would be neutral or harmful when the entire universe of six targets were available for at least a portion of the evolutionary run as in the other four algorithms.

The Multiobjective Simple Fitness and Incremental Simple Fitness techniques appeared to outperform the other techniques for this difficult hardware evolution problem. This suggests that fitness evaluations including several different input/output targets are important to prevent overfitting for any one target. However, there was no clear advantage to presenting these multiple targets incrementally or all at once from the beginning of the evolutionary run. As the scale of this problem grows to encompass the fully automated design of complex neuromorphic circuits used in biomedical applications, understanding the advantages and disadvantages of the incremental vs. “all at once” approaches will likely be of increasing importance.

A longer-term goal is to first validate this reverse engineering approach using ion channels and other systems simulated in NEURON but to eventually use circuit evolution as only one portion of a closed-loop experimental system (Bongard and

Lipson 2007). A particularly relevant application of such a system would be the fully automated design of neuromorphic circuits that can mimic large portions of a biological nervous system. These types of circuits are of increasing importance in science and medicine, but as with most analog or mixed-mode circuits, neuromorphic circuit design must currently be performed by human experts (Douglas et al. 1995, Smith 2010). Successful neuromorphic circuit designs deployed in the real world are often highly complex, such as the various hand-designed implementations of the full Hodgkin-Huxley model reported in the literature (Kohn and Aihara 2007, Simoni et al. 2004). The assistance of automated reverse engineering for these design tasks is likely to be of increasing importance in the future, especially as the need for neuromorphic circuits customized to individual patients begins to increase (Sicard et al. 1999). A practical neuromorphic system would probably be implemented with integrated circuits in VLSI (Iniewski 2008), but as with almost all studies related to analog circuit evolution reported in the literature, discrete electrical components were used here. However, this approach would translate easily to the automated design of integrated circuits.

One drawback of analog circuit evolution, as well as evolutionary design in general, is that evolved designs can be much more difficult to interpret than human designs (Kashtan and Alon 2005). However, many biomedical applications of evolved circuits may not require such an interpretation or may not require an obvious correspondence between circuit elements and modeled biological function. For example, some aspects of drug design and testing only require that the qualitative behavior of a neurophysiological system be reproduced (Noble 2008).

In this work, the focus was on analog circuits as a representation for models of neurophysiological systems. Some preliminary work suggested that this representation has an advantage over other representations when evolving both the form and parameters of models for neurophysiological systems. These other representations include differential equations (Ellner and Guckenheimer 2006, Hoppensteadt and Peskin 2002) and Markov models (Gurkiewicz and Korngreen 2007, Hawkes 2003, Menon et al. 2009). An interesting avenue for future work will be to more thoroughly examine the pros and cons of these different representations and how they interact with different types of evolutionary search.

In summation, this section describes the results of a study designed to evaluate different approaches to the evolution of analog circuit models of the Hodgkin-Huxley potassium ion channel. The motivation for a study encompassing many different evolution techniques was the observation that circuits can be relatively easily evolved that reproduce ion channel behavior as characterized by a single input/output relationship, but that generalizing to arbitrary ion channel behavior is surprisingly difficult. The main finding was that potentially numerous input/output training targets are needed to evolve circuits that generalize well to previously unseen data.

Reverse-Engineering Nonlinear Analog Circuits

Introduction

To continue progress toward the overall goal of automated modeling of ion channels with analog circuits, the problem of poor model generalization revealed in the experiments described in the previous sections of this chapter would have to be addressed. A serious difficulty any evolutionary algorithm is likely to experience for this application is that fitness evaluations involve Spice simulations, a very expensive procedure from a computational standpoint. The time domain simulations here take on the order of 100 ms - 1 s per CPU core, per evaluation, which is orders of magnitude more than fitness evaluations in other common genetic programming applications such as symbolic regression. As a result, good generalization must be achieved with a relatively small number of training and/or validation data points.

Nonetheless, many successful approaches to automating the design of analog circuits based on evolutionary computation have been proposed. Active learning, in which only the most informative tests are used for fitness evaluation (Krogh and Vedelsby 1995, Bongard and Lipson 2007), and incremental learning, in which tests are gradually added over the course of evolution (Torresen 2002, Bongard 2011) are two approaches that have been proposed for difficult hardware evolution problems in which data must be used as efficiently as possible. As a circuit that generalizes well must satisfy potentially conflicting demands in order to mimic ion channel behavior for a wide range of stimulus values, it is possible that another technique, multiobjective optimization (Deb et al. 2000), might also be an efficient way to

improve generalization.

The fitness evaluations necessary to evolve linear analog circuits are relatively straightforward. However, as detailed above, this is not the case for nonlinear analog circuits, especially for the most general class of design tasks of which ion channel modeling is a special case: reverse-engineering an arbitrary nonlinear “black box” circuit. Different approaches to fitness evaluations in this setting are investigated in this section. Results show that an incremental algorithm outperforms naive approaches and that it is possible to evolve robust nonlinear analog circuits with time-domain output behavior that closely matches that of black box circuits for any time-domain input.

Background

Analog circuit design is a challenging task that typically requires a domain expert with years of experience (Rutenbar et al. 2002). As a result, the costs associated with developing circuits for new applications can be very high. Many studies reported in the literature have attempted to automate the circuit design process in order to reduce the human effort required (Rutenbar 1993, Harjani et al. 2006, El-Turky and Nordin 1986, Koh et al. 1990). Approaches involving evolutionary computation have been particularly successful, and these generally allow both the topology and component sizes of a circuit to be optimized. Perhaps the earliest work in this area was by Koza and colleagues, who employed genetic programming and a developmental encoding (Koza et al. 1997). Later work showed that simpler methods based on genetic algorithms could also obtain good results (Kruiskamp and Leenaerts 1995, Lohn and

Colombano 1998).

All approaches to the automation of analog circuit design involving evolutionary computation rely on fitness evaluations, in which the behavior of a candidate circuit is simulated and compared with a design specification. For linear circuits, such as those comprised only of passive resistors, capacitors, and inductors, these fitness evaluations are relatively straightforward because the behavior of a linear circuit is fully characterized by its frequency and phase responses (Schaumann and Van Valkenburg 2001). In a typical application, the AC frequency response of an evolved circuit can be compared with the desired frequency response using a measure such as the mean-squared error (Koza et al. 1997). This error then serves as the basis for assigning a fitness value to the evolved circuit (Figure 2.14a). For nonlinear circuits, such as those with transistors, diodes or other nonlinear components, fitness evaluations can be considerably more difficult as it is generally not possible to fully characterize the behavior of a nonlinear circuit with a single measurement such as AC frequency or phase response (Hedrich and Barke 1995).

Despite this complicating factor, some studies have successfully “forward-engineered” nonlinear analog circuits with evolutionary computation (Koza et al. 1997, Sapargaliyev and Kalganova 2012). In these studies, a problem-specific design specification was developed and the task was to evolve a circuit that meets this design specification. Depending on the nature of the target design, Fourier analysis, DC sweeps, time-domain transient analysis, or other analyses can be applied to an evolved circuit and used to compare its behavior to that of the design specification. Although time-domain analyses are the most computationally expensive, these are often

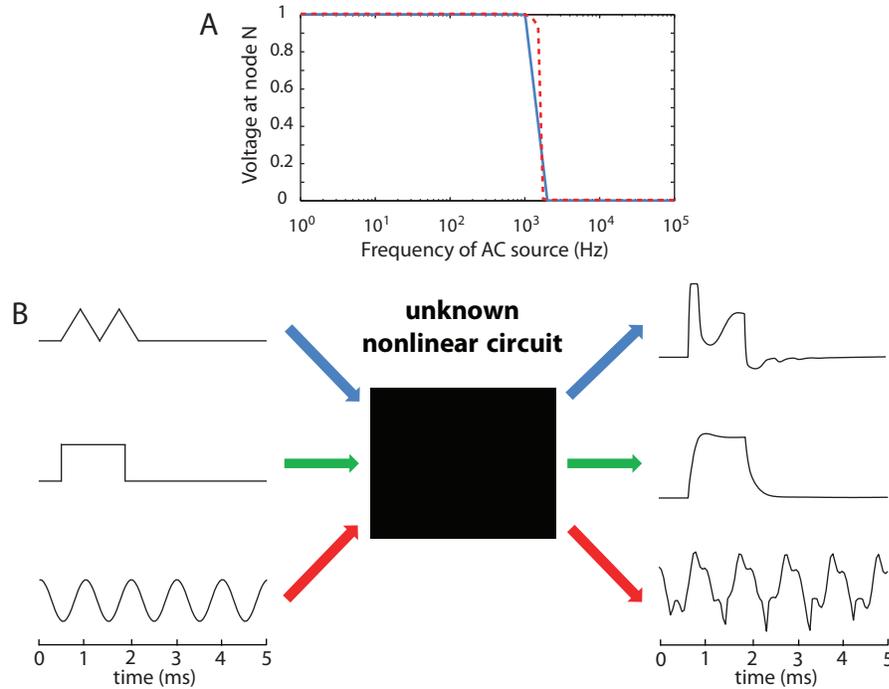


Figure 2.14. Fitness evaluations for linear and non-linear analog circuit evolution. **A)** The fitness of an evolved linear analog circuit can easily be assessed by comparing its frequency response (dashed line) to a desired frequency response (solid line). In this case the desired behavior is that of a low pass filter. **B)** To reverse-engineer an unknown “black box” nonlinear analog circuit, time domain inputs (left) are used to probe the black box (center). The resulting outputs (right) are compared with the outputs of an evolved circuit in response to the same inputs as a means of assigning a fitness value to the evolved circuit.

considered necessary to evolve the most robust circuit designs (Mydlowec and Koza 2000, Sapargaliyev and Kalganova 2012).

The forward-engineering tasks considered in previous studies of nonlinear analog circuit evolution are fundamentally different from a reverse-engineering task in which a nonlinear circuit must be evolved without a design specification. Instead, the circuit is evolved to match the behavior of an unknown “black box” nonlinear circuit (Figure 2.14b). The black box circuit must be probed with different voltage or current inputs in order to obtain information used as the basis for evaluating the fitness of

evolved circuits, yet the appropriate type of stimuli to use for these probes would generally not be known in advance. This is especially true if the goal is to evolve a circuit that generalizes well and reproduces the time-domain output behavior of a black box circuit for arbitrary time-domain inputs not used during evolution.

In this section, the goal was to investigate different algorithms for selecting the type of probe stimuli to use for the efficient evolutionary design of nonlinear analog circuits in the most general reverse-engineering setting. Many possibilities exist for such an algorithm, including the incremental approaches that have proven very powerful in other applications of evolutionary hardware (Torresen 2002). Using randomly-generated nonlinear circuits in order to evaluate different approaches in as unbiased a manner as possible, the results show that incrementally presenting probe stimuli is a particularly efficient means of guiding nonlinear circuit evolution. Importantly, the evolved circuits generalize well and match target circuit behavior for arbitrary stimuli.

Genetic Algorithm

The experiments here employed a genetic algorithm in which circuits were represented as variable-length linear chromosomes, where each gene contained four elements (Figure 2.15). The circuits considered in this work were composed of five component types: resistors, capacitors, inductors, pnp bipolar transistors, and npn bipolar transistors. In the case of resistors, capacitors, and inductors, the elements in a gene specified the type, connection nodes, and parameter value of the component. To mimic the limitations of physical hardware, these components had one of 96 possible

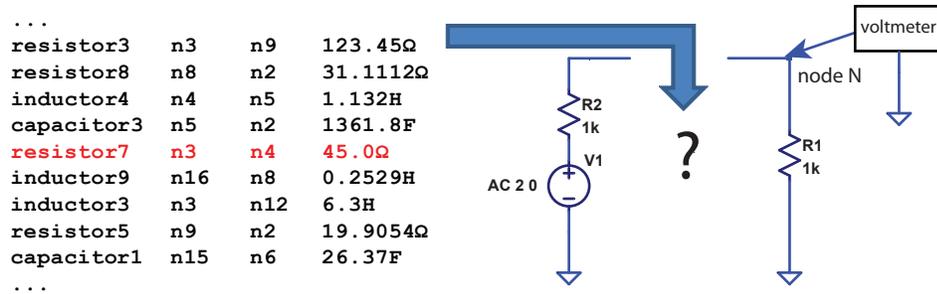


Figure 2.15. Circuit representation. Circuits were represented as linear chromosomes, with each gene (such as the one highlighted) corresponding to one component. The list as a whole specifies the components in a variable portion of an otherwise invariant embryonic circuit (right). Note that the embryonic circuit contains both a voltage source and an output node at which voltage measurements are made.

parameter values taken from a \log_{10} scale with 12 values per decade. These values ranged from 1.0 - 8.2×10^7 pF, nH, and Ω for capacitors, inductors, and resistors respectively. In the case of bipolar transistors, NG-Spice22³ default models were used and the four-element gene specified transistor type (pnp or npn) and the connection nodes for emitter, base, and collector. Each individual in the population of 64 circuits was initialized by concatenating 1 - 9 randomly chosen genes. The resulting circuit was checked for topological validity. This validation process included checks for dangling nodes, disconnected subcircuits, inductor loops, the lack of capacitor paths to ground, and other common problems that would have made the circuit unsimulatable in NG-Spice22.

As in standard genetic algorithms, crossover and mutation were used to create new members of the population. Parents were selected at random from the population of 64 and used to generate two offspring with a 90% chance of crossover. Instead of one point, two point, uniform or another standard type of crossover for linear

³ <http://ngspice.sourceforge.net>

chromosomes, a type of topology-aware crossover was used (Das and Vemuri 2007). Regardless of whether the two offspring were generated by crossover, each gene in each offspring was mutated with a probability of 5%. A mutation consisted of the change from one type of element to another, a change of connections from one node to another, or a change in parameter values. For these experiments, it did not appear to be necessary to use mutations to add or delete genes, as crossover in the variable-length chromosomes appeared to provide sufficient variation in offspring size. Offspring were created in this manner until 64 offspring were obtained. Both crossover and mutation were designed to ensure that only topologically valid circuits resulted from the operation.

Genotypic age was used to perform survival selection with the Age-Fitness Pareto algorithm for some experiments (Schmidt and Lipson 2010a-b). Survival selection was performed by culling the combined population of parents and offspring using tournament selection with replacement and tournaments of size 2. This process was implemented with a Pareto tournament scheme in which two random members of the combined population were selected. If one of the pair had both lower fitness and higher age than the other, it was discarded. The survivor was then returned to the pool. This continued until the population size was reduced back to 64. If a large number of tournaments failed to result in any discarded individuals, individuals would have been randomly discarded until the population size was reduced back to 64. However, the need to randomly discard individuals in this manner never arose. Age was defined as the number of generations in which an individual had been present in the population, where parents passed on their age to offspring. Offspring inherited the

age of the older parent in the case of crossover. One new, randomly generated individual with an age of 0 and created in the same way as members of the initial population was added to the population each generation.

When evaluating the fitness of a circuit, the linear chromosome was translated into a netlist recognizable by the NG-Spice22⁴ circuit simulator. This netlist specified the variable portion of an otherwise invariant embryonic circuit common to all evolved and target circuits (Figure 2.15). Time-domain transient analysis was used to measure the voltage of the circuit at the output node in response to a voltage source with an arbitrary waveform as the input. The differences between evolved circuit outputs and target outputs were compared by calculating the sum of squared errors in the time domain. Random voltage source waveforms were specified by 20 parameters: 18 random parameters corresponding to the first nine coefficients of the Fourier series, one parameter controlling the scaling of the waveform with respect to time, and one parameter controlling the phase offset of the waveform.

Stimulus Selection

As the focus of this study was on comparing different methods of selecting probe stimuli for obtaining information about target circuits and evaluating the fitness of evolved circuits, several different methods for doing so were compared. The simplest method, “Fixed Single,” was to probe the target black box circuit with a single randomly generated voltage source waveform. The resulting voltage output was then used as the basis for assigning fitness values to evolved circuits. Each fitness

⁴ <http://ngspice.sourceforge.net>

evaluation consisted of applying the same input to the evolved circuit and comparing the resulting output in the time domain with that originally obtained for the target circuit. Fitness was calculated as the sum of squared errors between target output and the output of the evolved circuit.

A second method for stimulus selection, “Fixed Four,” was inspired by the fact that voltage ramp and step functions are very common hand-designed waveforms used to probe circuits (Mydlowec and Koza 2000). Two fixed but different voltage ramps and two fixed but different step functions were each used to probe the target and evolved circuits. Fitness was assigned as the sum of the errors on each of the four targets.

The third stimulus selection method, “Switching Single,” was the same as Fixed Single except that a new randomly generated target input/output pair was generated periodically. In other words, the stimulus switched at regular intervals during evolution. This method was used to determine whether overfitting to a fixed single input/output pair was a significant problem.

Incremental stimulus selection was also employed. Inspired by work in digital circuit evolution (Torresen 2002) and other evolvable hardware domains, this method involved periodically generating a new random stimulus, applying it to the black box target circuit, and then adding this new input/output pair to the set of test cases used to evaluate fitness of an evolved circuit. The fitness of an evolved circuit was then defined simply as the sum of its fitness values for each individual stimulus. The “Switching Incremental” algorithm employed this approach without Age-Fitness Pareto optimization whereas the “Age-Fitness Incremental” algorithm was identical

except that Age-Fitness Pareto optimization was used.

For all five algorithms, the time required to evolve circuits was almost entirely spent on the computationally expensive NG-Spice22 simulations required for fitness evaluation. As there were significant differences in the number of fitness evaluations required for the five algorithms for a given number of evolutionary generations, all experiments below accounted for these differences and use the same total number of fitness evaluations (Spice simulations) for all algorithm comparisons. Given the computational expense of Spice simulations, evolutionary runs were performed for a predetermined number of fitness evaluations instead of following the standard practice of letting the genetic algorithm run until stagnation. However, the number of fitness evaluations used appeared to be sufficient to draw preliminary conclusions about the performance of the various algorithms.

Target Circuits

A hand-designed RTL inverter circuit was used as the reverse-engineering target for initial experiments. A RTL inverter is a simple transistor switch that implements logical negation. Not counting the elements of the embryonic circuit, this circuit had three elements as shown in (Figure 2.16). Each of 40 evolutionary trials used this same simple circuit as a target to obtain baseline information about the performance of the algorithms.

In order to evaluate different stimulus selection methods in as unbiased a manner as possible, further experiments employed randomly-generated nonlinear circuits as the black box targets. As part of the process of generating a random circuit,

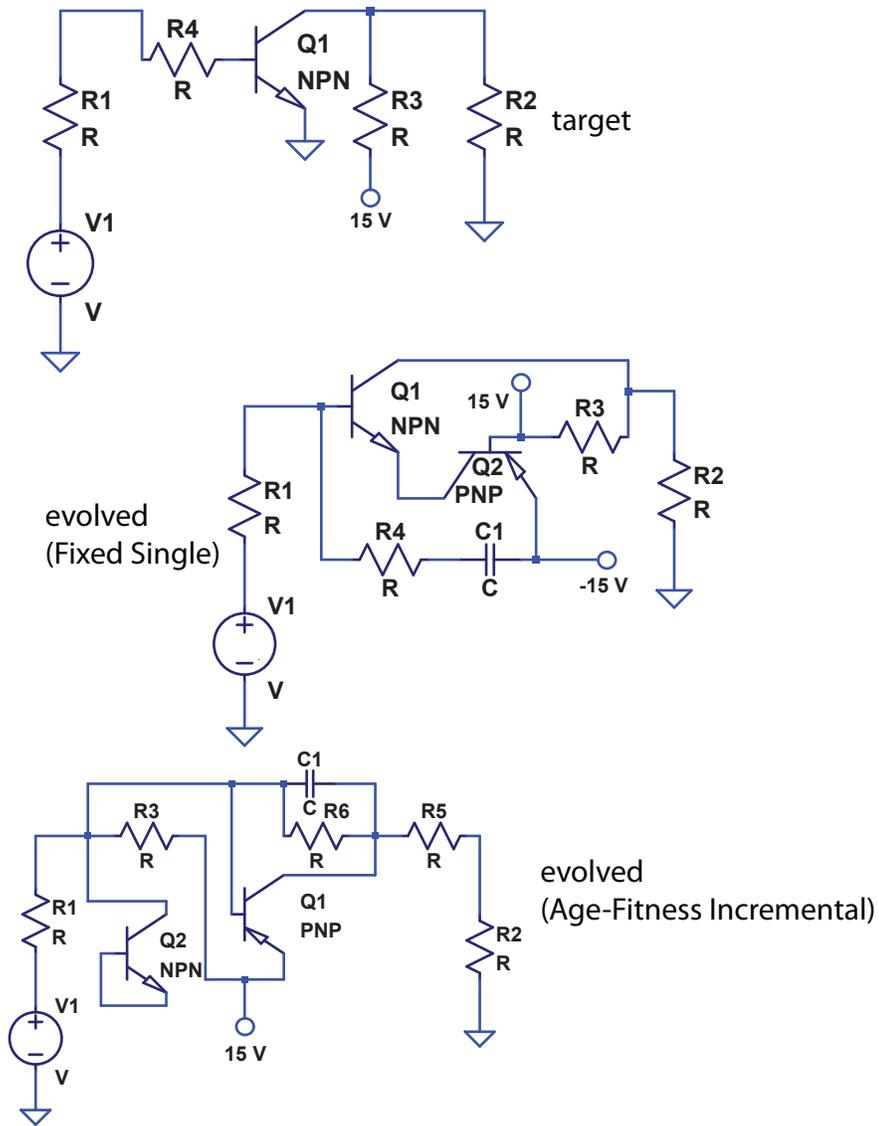


Figure 2.16. RTL inverter target and evolved circuits. The target circuit is shown at top. The other circuits are those evolved to match the behavior of this target circuit using the Fixed Single stimulus algorithm (middle) and the Age-Fitness Incremental algorithm (bottom).

extensive tests were performed to ensure that the size of the final circuit accurately reflected the somewhat qualitative notion of circuit complexity. In particular, every possible combination of components was systematically removed to determine whether a simpler circuit had virtually the same behavior. This procedure was

performed in addition to standard simplification methods such as combining linear elements in series or parallel that are guaranteed not to affect the behavior of the circuit. An example of a random nonlinear circuit is shown in Figure 2.17.

Random circuits were generated with an algorithm consisting of several steps that were repeated until 40 random circuits with a variety of sizes were eventually produced:

Step 1) A new circuit is formed by creating a random netlist with 3 - 11 elements and at least one nonlinear element (one of the two types of bipolar transistors). The number of elements is chosen randomly using a uniform distribution and each component is randomly selected from all possible components, with each type equally likely.

Step 2) The new circuit is inserted into the embryonic circuit (Figure 2.15) and simulated in NG-Spice22. In the simulation, the circuit is stimulated with 25 separate voltage source waveforms randomly generated as described in **Genetic algorithm** above.

Step 3) If the circuit is simulatable and the voltage response of the circuit measured at “node N” as shown in Figure 2.15 is of sufficient variability, go to Step 4. Otherwise, discard the circuit and go to Step 1. Sufficient variability is defined here as having a variance of at least 0.025 on all 25 test stimuli after the response is scaled to the range [0, 1]. This step was motivated by the observation that many randomly created circuits were valid and simulatable but were not interesting candidates for reverse-engineering in that they generated flat or nearly flat responses for any inputs.

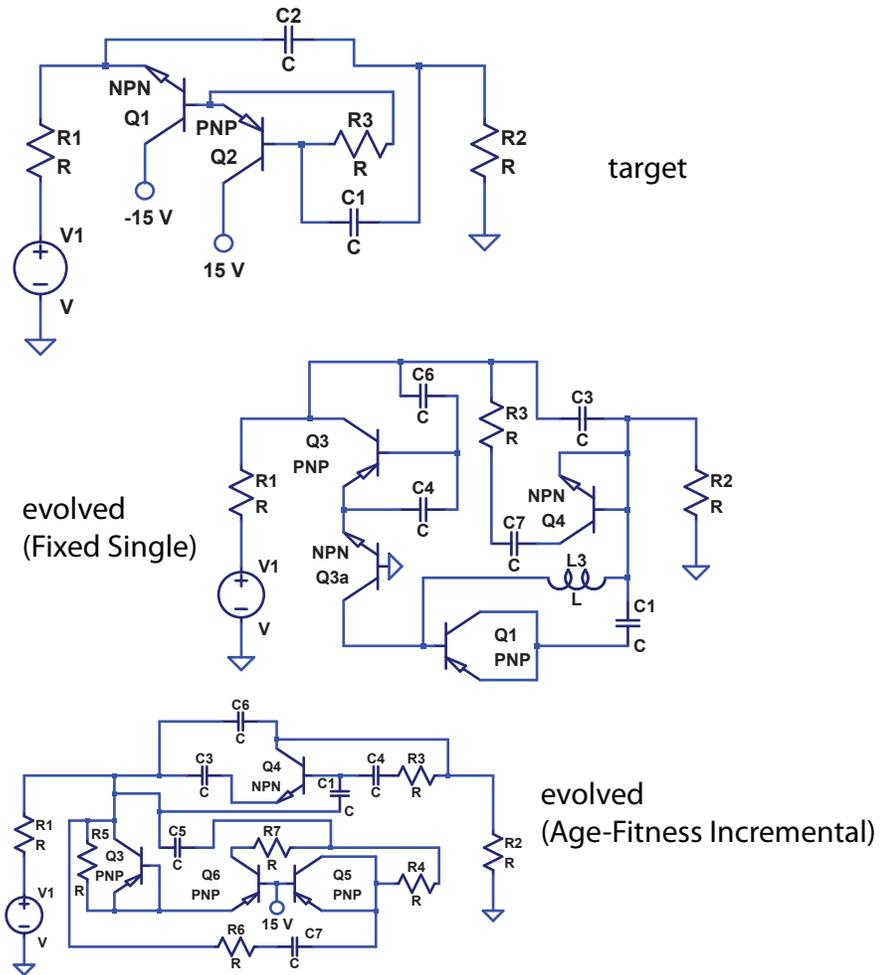


Figure 2.17. Random target and evolved circuits. One randomly generated target circuit is shown at the top. The other circuits are those evolved to match the behavior of this target circuit using the Fixed Single stimulus algorithm (middle) and the Age-Fitness Incremental algorithm (bottom).

Step 4) Call the circuit at this point C1. C1 is simplified without altering its functionality by combining linear components of the same type in series and parallel.

Step 5) Every possible combination of components is deleted from the variable portion of the circuit, including the combination in which no components are deleted. For each combination of deleted components, if the remaining circuit is valid,

it is simulated and tested with the same 25 waveforms used in Step 2.

Step 6) Among all the circuits formed in step 5, any that are not valid circuits or are not simulatable are discarded. Among the remaining circuits, the response of each is compared to the response of circuit C1 for all 25 input waveforms. If the total sum of squared differences between the truncated circuit's response and the response of C1 is greater than or equal to 0.001 (when all responses are scaled to between 0 and 1), then the truncated circuit is discarded. Designate the smallest truncated circuit that has not been discarded as C2. If C2 has fewer components than C1, set C1 equal to C2 and go to Step 4. Otherwise, return C2.

Results

Each of the five stimulus selection algorithms were applied to the target circuits in 40 trials. The results summarizing evolutionary progress across all 40 trials for all five algorithms with the RTL inverter as the target are shown in Figure 2.18. Similar results were obtained for random circuits as shown in Figure 2.19. In both cases, presenting stimuli with the incremental algorithm was clearly the most effective method of evolving circuits that generalized well and that matched the performance of the target circuits for several random stimuli not used during evolution. In contrast, little benefit was gained by using different non-incremental approaches given a fixed computational budget.

Figure 2.16 shows two examples of circuits obtained with the Fixed Single algorithm and the Age-Fitness Incremental algorithm, as well as the RTL inverter

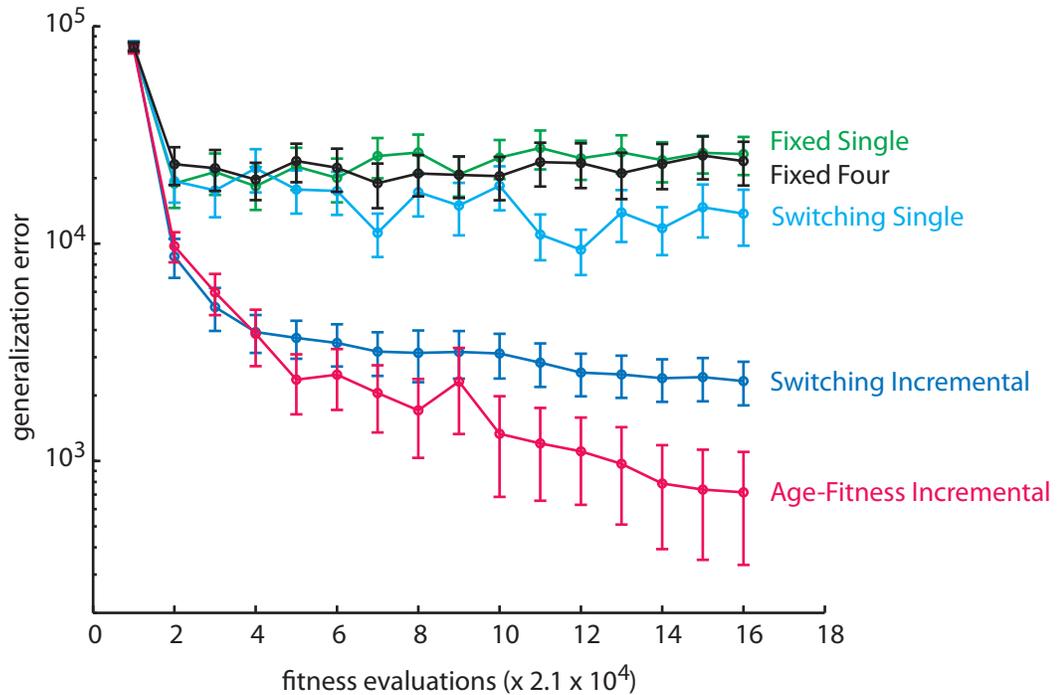


Figure 2.18. Generalization error for the five stimulus selection methods with the RTL inverter target. At regular intervals (approximately once every 2.1×10^4 fitness evaluations), each member of the current population of circuits was evaluated on 25 randomly chosen stimuli not seen during training. The circuit with the lowest total error was then recorded. The mean value of this error across each of 40 trials is plotted, with error bars representing \pm the standard error of the mean.

target circuit they were evolved to mimic. Although the evolved circuits do not match the target circuit in size or structure, the circuit evolved using the Age-Fitness Incremental algorithm closely matched the behavior of the target circuit for arbitrary stimuli (Figure 2.20). A similar conclusion can be drawn from the results for random circuits as shown in Figure 2.17 and Figure 2.21.

Discussion

In this section, different methods of fitness evaluation for reverse-engineering nonlinear analog circuits with a genetic algorithm were compared. Stimulus

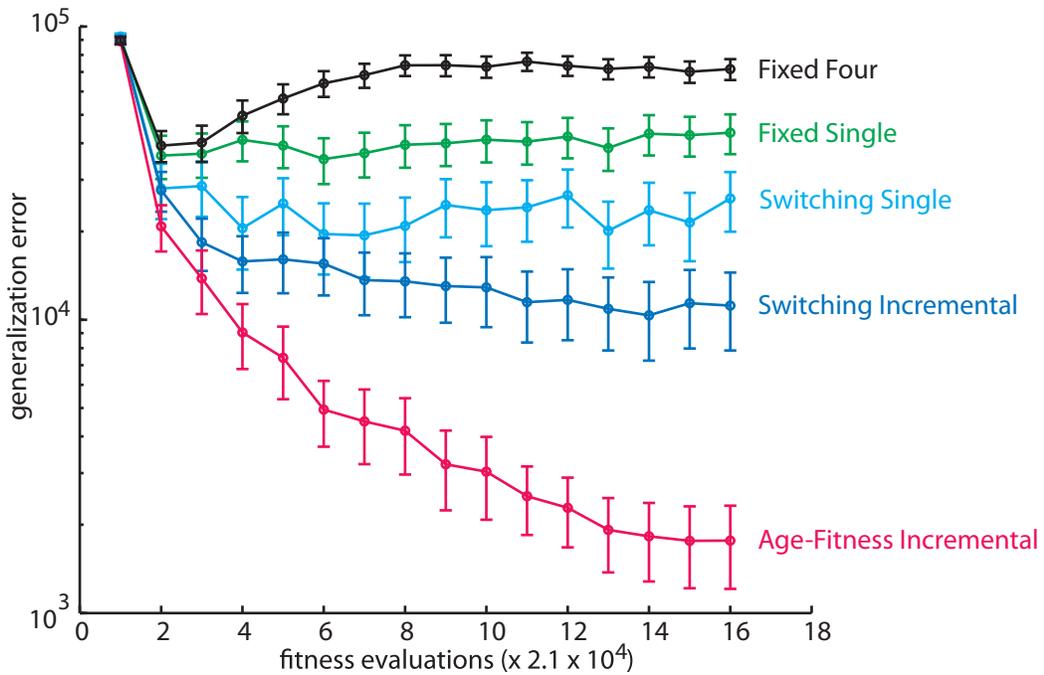


Figure 2.19. Generalization error for the five stimulus selection methods with random targets. Axes and bars are the same as in Figure 2.18.

presentation with an incremental algorithm proved to be a particularly effective means of performing these fitness evaluations. This method successfully produced circuits that closely matched the behavior of black box target circuits.

One weakness of the approaches studied here is that the evolved circuits fail to match their respective target circuits in size and structure, even when they match them in behavior. It could then be argued that the evolved circuits are no more valuable as models of the target circuit than a neural network or other universal function approximator that can also reproduce the input/output behavior of the target circuit. However, using circuits as the function approximator arguably confers two advantages. First, circuits are almost certainly the most useful function approximators for a reverse-engineered circuit if the function approximator is to be implemented in

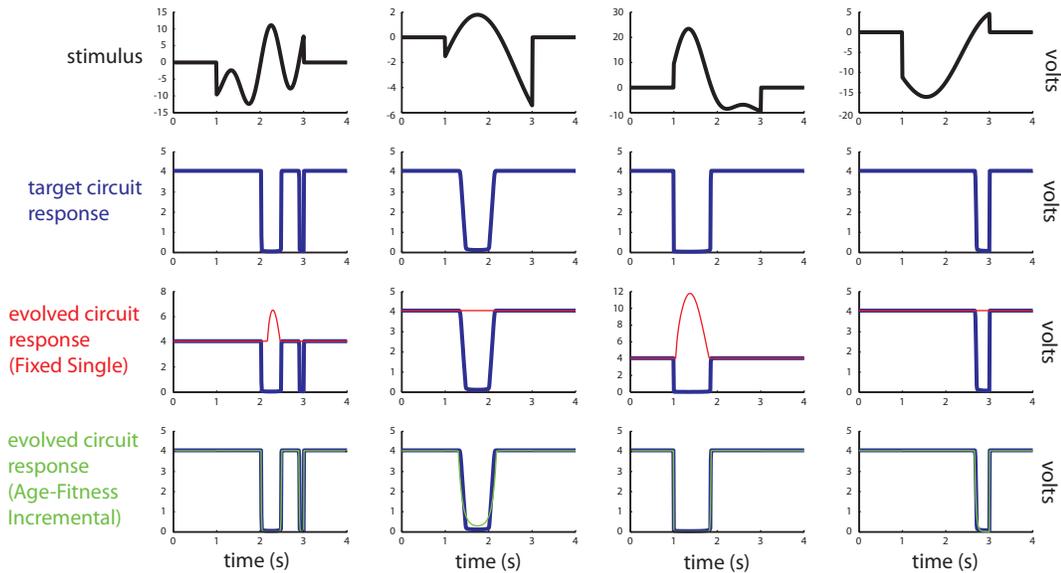


Figure 2.20. RTL inverter target and evolved circuit behavior. Four randomly chosen stimuli not used during training (top row) were applied to the target circuit, which generated the responses shown in the second row. The response of the circuit evolved with the Fixed Single algorithm shown in Figure 2.18 to those same four stimuli is shown with the thin line in the third row. These thin lines are plotted over the target responses (thick lines) for reference. The response of the circuit evolved with the Age-Fitness Incremental algorithm shown in Figure 2.18 to those four stimuli is shown with the thin line in the bottom row plotted over the target responses.

hardware. Second, circuits and electrical components, although computationally expensive to simulate, are likely to be a natural representation with which to approximate the input/output functions embodied by the target nonlinear circuits and so might be easier to identify than neural networks or other non-domain specific representations.

Future work could extend the results obtained here in several ways. First, it will be important to study random target circuits of larger sizes, as reverse-engineering targets in real life applications tend to contain considerably more components than the

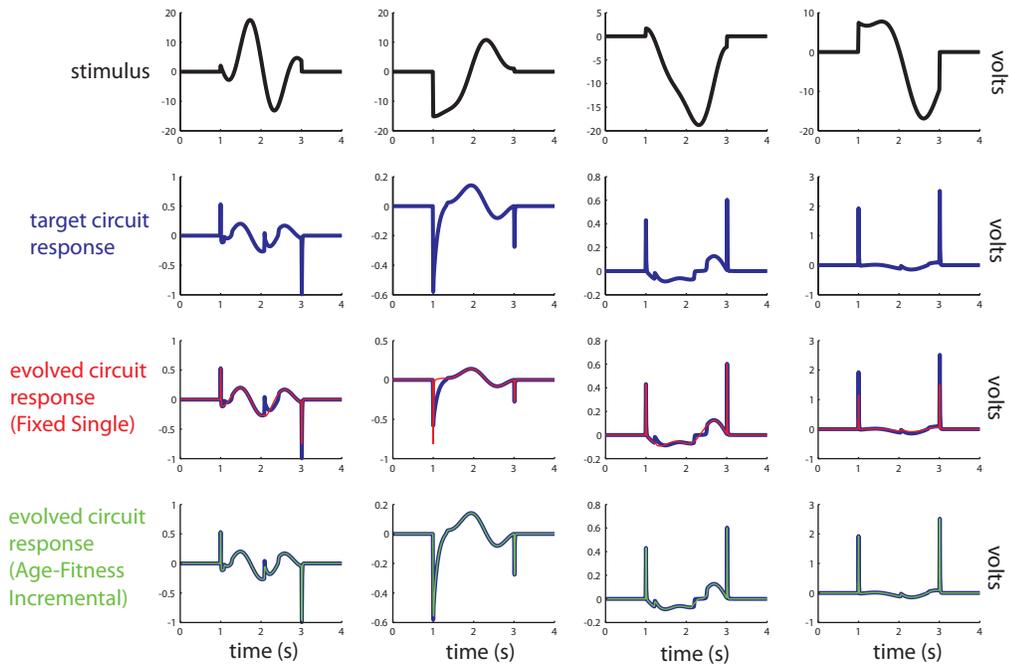


Figure 2.21. Random target and evolved circuit behavior. Panels are the same as in Figure 2.20 except that results are for circuits shown in Figure 2.19.

circuits considered here, which had 3 - 11 components. Second, physical circuits implemented in hardware should be used as targets as an important means of confirming the results obtained here with simulations. Third, the prospect of coevolving stimuli instead of randomly generating them as in the current study should be investigated. Such an approach has proven effective in other evolutionary computation problem domains such as symbolic regression (Bongard and Lipson 2007). Finally, the approaches here should be applied to additional circuits with specific, known functions such as cube root circuits (Koza et al. 1997). Although the results would not be as general as when the targets are random circuits, this would be useful as a baseline comparison of the approaches considered here to the more common evolutionary hardware methods oriented to forward-engineering tasks.

Comparison of Approaches to Analog Circuit Design

Introduction

Many evolutionary approaches to automating the design of analog electrical circuits have been proposed, yet few studies have compared these approaches in a rigorous and unbiased manner. In this section several distinct evolutionary algorithms are tested on a standard linear lowpass filter design problem. Results reveal several characteristics of evolutionary algorithms that are important for success in this difficult evolvable hardware task.

Background

Analog circuits continue to be an essential component of many electrical systems, especially when an interface with the external world is required. Analog technology is also preferred over digital when high signal processing speed and low power consumption is important for the application (Vittoz 1985). However, unlike with digital circuits, the design of analog circuits is still performed largely by hand. This requires an expert in analog design and is often a lengthy and expensive process (Rutenbar et al. 2002). As a result, many approaches to automating one or more aspects of the analog circuit design process have been proposed (Rutenbar 1993, Harjani et al. 2006, El-Turky and Nordin 1986, Koh et al. 1990). Among these are the powerful approaches that use efficient evolutionary search through extremely large design spaces to optimize both the topology and device parameters of a circuit with minimal human guidance (Koza et al. 1997, Kruiskamp and Leenaerts 1995, Lohn and

Colombano 1998, Mattiussi and Floreano 2007).

As part of any evolutionary approach to circuit design, the behavior of a candidate circuit must be measured and compared to a design specification. Circuit behavior is usually measured by simulation and analysis with the industry-standard Spice circuit simulator. Although Spice simulations of analog circuits are a robust and accurate means of assessing circuit behavior, they are computationally very expensive and generally require orders of magnitude more CPU time than simulations of digital circuits. Perhaps because of the computational effort involved, few studies of analog circuit evolution reported in the literature perform several independent evolutionary trials to rigorously evaluate the proposed approach or to compare it to other approaches on the same design problem. Furthermore, these studies typically consider only one or a few specific design problems, which makes general performance trends difficult to identify.

In this section, a systematic comparison is made of several approaches to analog circuit evolution that have been proposed in the literature: a standard genetic algorithm, Evolution Strategies, Immune Programming, and Age-Fitness Pareto optimization. This latter is a multiobjective variant on the genetic algorithm that has achieved state-of-the-art performance in other evolutionary computation domains but has not previously been applied to analog circuit design. Within these four general approaches, the influence of topology-aware crossover is also studied. Like substructure reuse and the role of simplification, topology-aware crossover is sometimes mentioned anecdotally in the analog circuit evolution literature, but is rarely if ever studied in a rigorous manner.

Representation

As in the previous section of this chapter, circuits were represented as variable-length linear chromosomes, where each gene contains four elements (Figure 2.15). However, the linear circuits considered in this section did not include transistors or nonlinear elements and were composed of three component types: resistors, capacitors, and inductors. Simulating the frequency response of these linear circuits is far less computationally intensive than time-domain transient analysis of nonlinear circuits; linear circuits were used for this study so that a rigorous, large-scale comparison of several analog design methods would be feasible. The elements in a gene specified the type, connection nodes, and parameter value of the component. Each individual in the population of 64 circuits was initialized by concatenating 1 - 9 randomly chosen genes. The resulting circuit was checked for topological validity. This included checks for dangling nodes, disconnected subcircuits, inductor loops, the lack of capacitor paths to ground, and other common problems that would make the circuit unsimulatable.

Many studies of circuit evolution do not restrict the parameter values that components can be assigned and treat them as arbitrary floating point numbers. However, only a restricted range of component values is generally available for hardware implementations of analog circuits. Here, as in the previous section of this chapter, it is assumed that it is important for any circuit evolved circuit design to be suitable for eventual implementation in hardware. Parameter ranges used were the same as in the previous section.

When evaluating the fitness of a circuit, the linear chromosome was translated into a netlist recognizable by the NG-Spice22⁵ circuit simulator. This netlist specified the variable portion of an otherwise invariant embryonic circuit common to all evolved and target circuits (Figure 2.15). AC frequency analysis was used to measure the voltage of the circuit at the output node in response to an AC voltage source as the input. All experiments below involve the attempt to automatically design an idealized lowpass filter with a cutoff frequency of 2 kHz. The exact error measure used was chosen to match that in Koza et al. 1996c, who hand designed a fairly complex error metric that was found to perform well. Briefly, the frequency response of an evolved circuit was compared with the target specification at 101 points across the frequency spectrum. The differences between the observed and target values were weighted differently for different points, and the error metric plotted in results below is the sum of the differences at each of these points. See Koza et al. 1996c for details.

Compared Algorithms

Four main evolutionary algorithms were compared: a standard genetic algorithm (GA), Evolution Strategies, Immune Programming, and Age-Fitness Pareto optimization. In addition, to obtain baseline performance information, two naive algorithms were included in all experiments: a hill climbing algorithm and random search.

⁵ <http://ngspice.sourceforge.net>

Standard Genetic Algorithm

The distinguishing feature of the standard genetic algorithm is that crossover and mutation are used to create new members of the population and that maximization of fitness or minimization of error is the sole objective. Each generation, parents were selected at random from the population of 64 and used to generate two offspring with a 90% chance of crossover. Either two point crossover or topology-aware crossover (Das and Vemuri 2007) was used depending on the experiment. Regardless of whether the two offspring were generated by crossover, each gene in each offspring was mutated with a probability of 5%. A mutation consisted of the change from one type of element to another, a change of connections from one node to another, or a change in parameter values. Offspring were created in this manner until 64 total offspring were obtained. Topology-aware crossover and mutation ensured that only topologically valid circuits resulted from the operation, although 2 point crossover did not. Survival selection was performed by culling the combined population of parents and offspring using tournament selection with replacement and tournaments of size 2.

Evolution Strategies

As two point crossover is potentially a very disruptive strategy for circuit evolution, many studies have focused on mutation-only algorithms that do not use crossover. Evolution Strategies is one such relatively simple algorithm that differs from the standard genetic algorithm primarily in that no crossover is used and selection is typically performed differently. In the implementation here, at each generation, each parent was simply cloned once to create one offspring. This offspring was then

subjected to mutation as in the standard genetic algorithm. The pooled population of parents and offspring was then ranked by fitness and the bottom performing half of the pooled population was discarded. The surviving members of the population were then the parents for the next generation of evolution.

Immune Programming

Like Evolution Strategies, Immune Programming is a mutation-only evolutionary algorithm. Unlike Evolution Strategies however, selection in Immune Programming is performed with a complex scheme inspired by the antibody selection process in the vertebrate immune system. Instead of every parent contributing an equal number of offspring to the next generation as in Evolution Strategies, parents contribute more or less than others depending on their fitness. Here, the algorithm for determining the number of offspring contributed by a given parent was the same as that described in Conca et al. 2008. See that work for details.

Age-Fitness Pareto Optimization

Age-Fitness Pareto optimization is a recently-proposed approach to evolutionary computation in which the standard fitness objective is combined with an additional objective called genotypic age to create a multiobjective optimization problem (Schmidt and Lipson 2010a-b). In the implementation here, the algorithm proceeded in the same manner as the standard GA except that survival selection was performed by culling the combined population of parents and offspring using tournament selection with replacement and tournaments of size 2. This process was implemented

with a Pareto tournament scheme in which two random members of the combined population were selected. If one of the pair had both lower fitness and higher age than the other, it was thrown out. The survivor was then returned to the pool. This continued until the population size was reduced back to 64. Age was defined as the number of generations in which an individual had been present in the population. Parents passed on their age to offspring, and offspring inherited the age of the older parent in the case of crossover. One new, randomly generated individual with an age of 0 and created in the same way as members of the initial population was added to the population each generation.

Naive Algorithms

Two naive algorithms, a hill climbing algorithm and random search, were included in all experiments as a way of establishing a performance baseline against which the other four algorithms could be judged. Both of these naive algorithms eliminate the population-based searches used by the other algorithms. For each trial of the hill climbing algorithm, the search was initialized by randomly creating an individual with 1 - 9 genes in the same way as individuals in the initial population were created for the population-based methods. Mutation was repeatedly applied to this individual. If a mutated version had higher fitness than the “parent,” the mutated version replaced the parent and the algorithm continued. For the random search algorithm, individuals with 1 - 9 genes were repeatedly generated at random while a record was kept of the best individual found since the beginning of the trial.

Results

As a first experiment, a baseline comparison was made between the basic GA with 2 point crossover, the hill climbing algorithm, and the random search algorithm to see if results were consistent with those reported in the literature. Figure 2.22 shows that, as expected, the GA outperforms the two naive algorithms.

One of the strengths of the more complicated developmental encodings that have been reported in the literature for analog circuit evolution is that crossover always produces a valid offspring. This is not the case with naive 2 point crossover in the GA approach used here and there were no guarantees that a resulting offspring was a valid circuit. In fact, naive 2 point crossover with the unordered list of components in a linear representation appears to be equivalent to randomly taking several components from one parent, randomly taking several from another parent, and then combining these components at random in the resulting offspring circuits (Figure 2.23). In other words, the naive crossover operation is potentially very disruptive and may even result in the GA being functionally equivalent to a random search. It was hypothesized that if this were in fact the case, the basic GA with crossover only and no mutation might have performance similar to random search.

This hypothesis was tested and results showed that the GA with 2 point crossover only (and no mutation) performs worse than the full GA with crossover and mutation but does not perform as poorly as random search or hill climbing, and in fact performs better than the GA with mutation only (and no crossover). These results are shown in Figure 2.24. Also shown in Figure 2.24 are results for Evolution Strategies and Immune Programming, neither of which use crossover. Both were outperformed

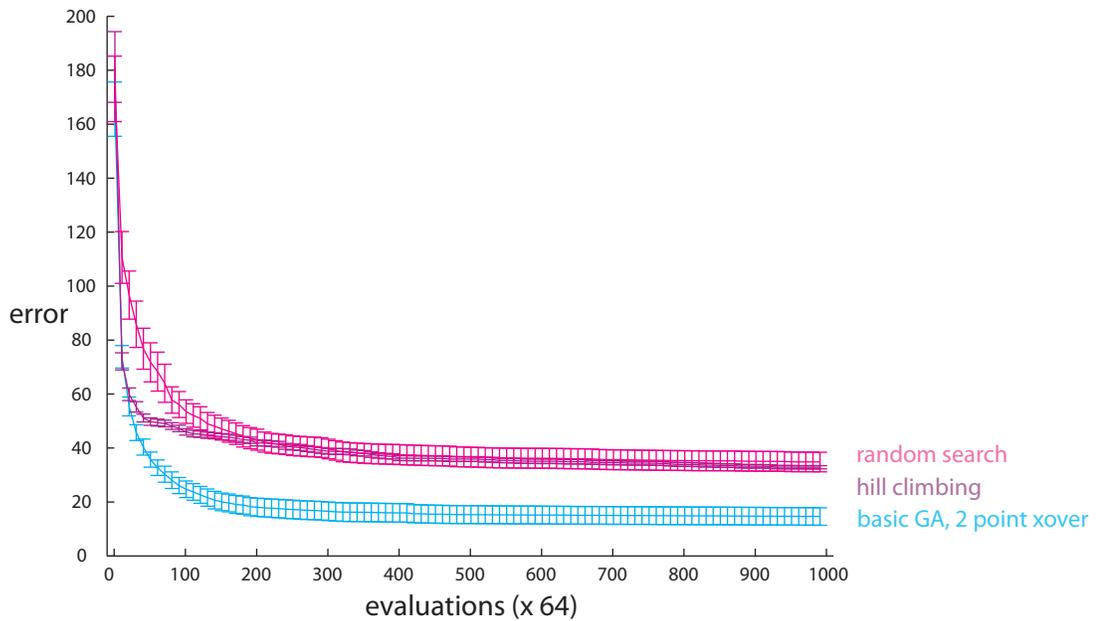


Figure 2.22. Basic GA compared with naive algorithms. As in all the experiments in this section, results are shown for 50 replicate trials. The curves plotted represent the mean \pm the standard error of the best circuits found after the indicated number of evaluations.

by either version of the GA that used crossover. These results indicate that even naive crossover appears to be an important part of how the GA searches through circuit space effectively. Nonetheless, topology aware crossover is a priori, an attractive alternative to the disruptive 1 point or 2 point naive crossover. A GA using topology-aware crossover did appear to significantly outperform a GA that was identical except that it used naive 2 point crossover (Figure 2.25).

Another important consideration for the automated design of analog circuits is the size of any evolved circuits. The size of circuits throughout evolution was also tracked in the experiments shown in Figure 2.25. This size data is plotted in Figure 2.26. The size of circuits in these experiments was artificially limited to 35, and hill climbing quickly found bloated solutions with 35 components without achieving good

resistor3	n5	n8	123.45 Ω	capacitor7	n3	n2	44.85F
capacitor8	n7	n4	3.56F	inductor12	n1	n5	1.22H
inductor2	n3	n9	12.91H	resistor6	n8	n2	1.783 Ω
capacitor3	n8	n3	0.043F	inductor8	n6	n1	90.34H
resistor7	n4	n2	2.729 Ω	capacitor2	n5	n8	872.9F
resistor2	n6	n5	8.823 Ω	capacitor8	n9	n4	2.395F
inductor5	n6	n3	573.82H	resistor3	n2	n3	0.003 Ω
inductor4	n2	n6	0.736H	resistor2	n1	n9	73.913 Ω
resistor4	n3	n7	44.28 Ω	capacitor3	n4	n3	4.162F
capacitor7	n1	n4	0.873F	inductor5	n8	n7	0.093H

Figure 2.23. Naive crossover. Standard 2 point crossover (and 1 point as shown here) applied to the linear representation is potentially very disruptive to the topology of the circuits involved.

performance. As random search by definition can never increase the size of any circuit, the circuits obtained with random search stayed small but again were unable to achieve good performance. The basic GA with either type of crossover gave more reasonable results, with higher performance than the hill climbing algorithm and with

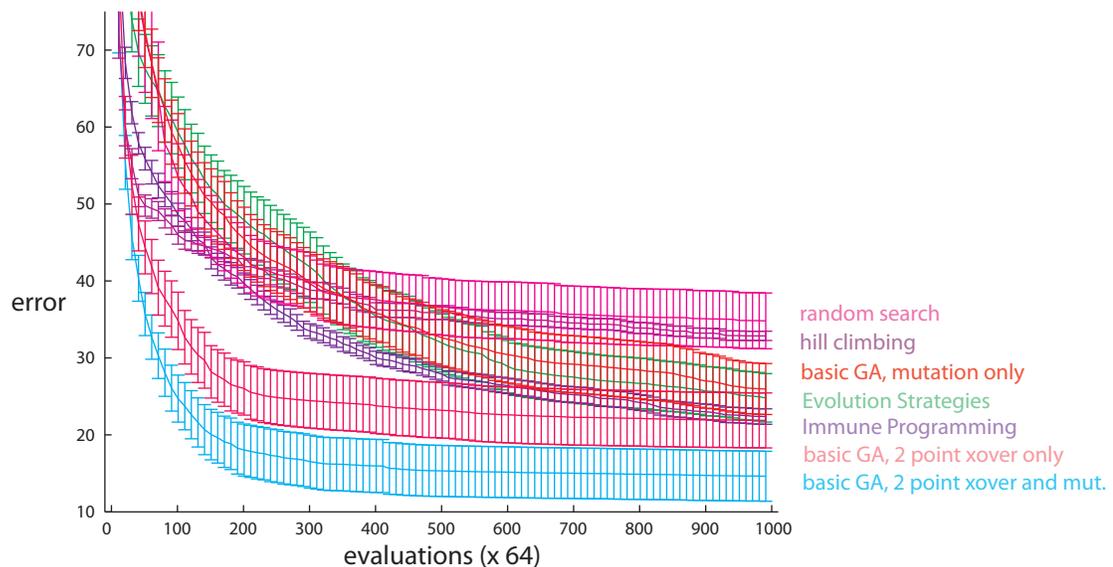


Figure 2.24. Variants of the basic GA compared with naive algorithms, Evolution Strategies, and Immune Programming.

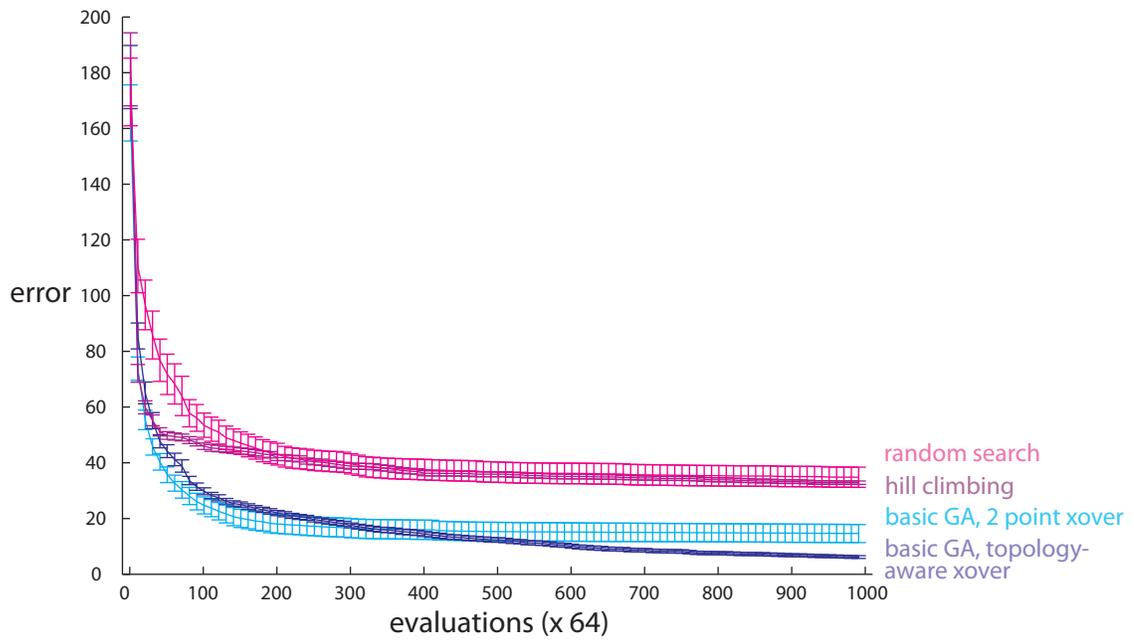


Figure 2.25. Basic GA with naive 2 point crossover compared with a basic GA using topology-aware crossover.

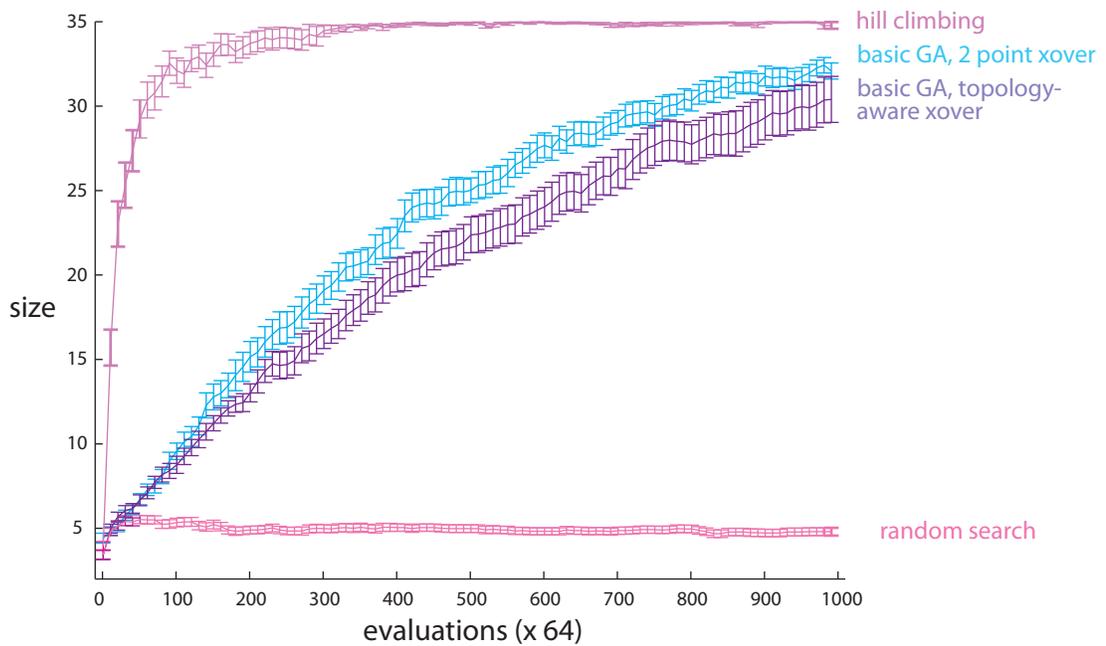


Figure 2.26. Size of evolved circuits as a function of number of evaluations for the experiment shown in Figure 2.25.

less bloat. However, circuits with approximately 25 - 30 components as obtained by both variants of the GA are roughly 2 - 3 times the size of human-designed low pass filters that show similar performance, and so it was hypothesized that even the GA variants in this experiment were finding bloated solutions.

Age-Fitness Pareto optimization is a newly proposed method for bloat control not previously applied to analog circuit evolution. The GA with the Age-Fitness Pareto optimization technique incorporated appeared to control bloat more effectively than the otherwise identical basic GA (Figure 2.27). Smaller circuit sizes are not necessarily better though, especially if their performance is inferior. However, the error of circuits evolved with the Age-Fitness GA outperformed the basic GA as well as naive algorithms (Figure 2.28).

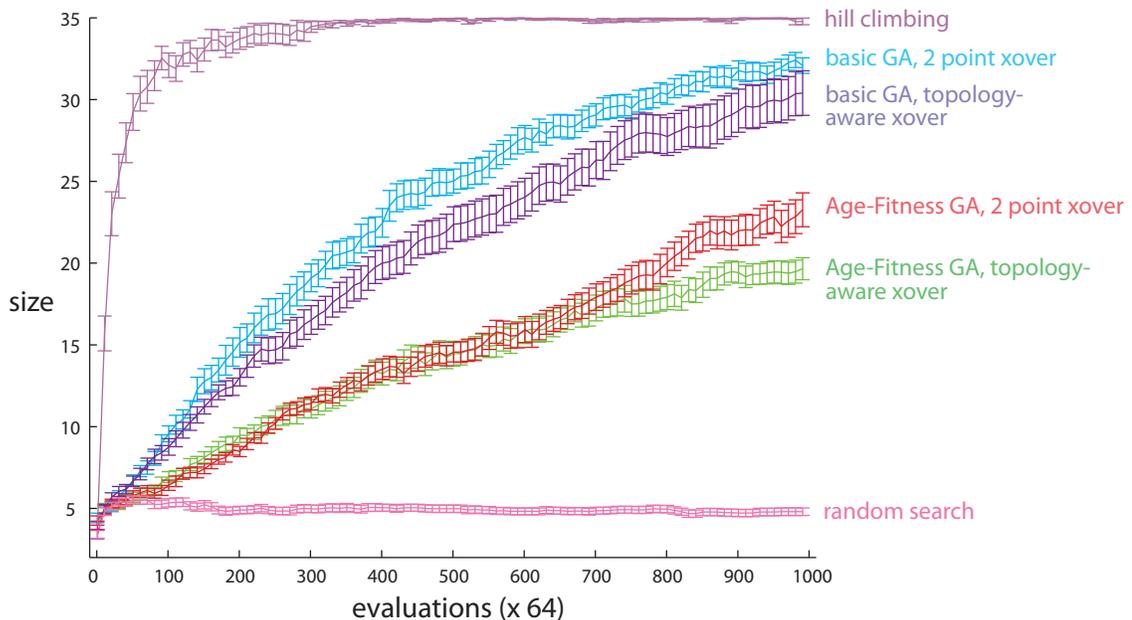


Figure 2.27. Size of evolved circuits obtained with the Age-Fitness GA compared with the basic GA and naive algorithms.

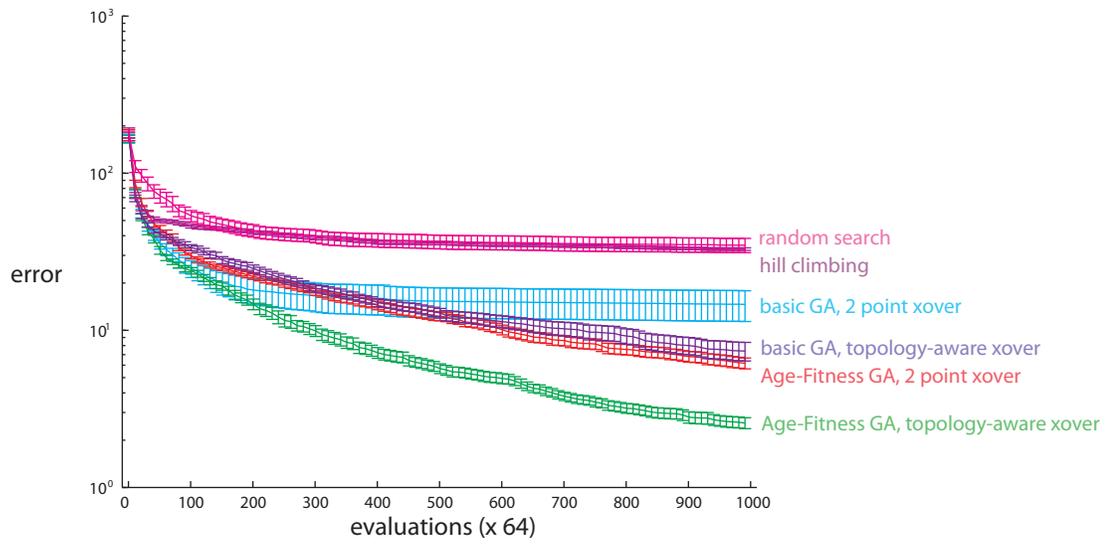


Figure 2.28. Error of evolved circuits as a function of number of evaluations for the experiment shown in Figure 2.27.

CHAPTER 3

DYNAMICAL SYSTEMS WITH MULTIPLE TIME SCALES

Introduction

The presence of dynamics at multiple time scales is a common feature of dynamical systems that contributes to their complexity and increases the difficulty with which they can be analyzed. In this section, an algorithm is proposed for creating a symbolic multi-scale dynamical model of a system based on experimental observations. Its applicability on a variety of both physical and synthetic systems is demonstrated, and results are shown for experiments designed to evaluate its performance and scalability with various levels of noise and time scales.

Background

The mathematical modeling of dynamical systems plays a central role in science, yet it is increasingly difficult for the human capability to construct models to keep pace with the growth in complexity of the systems under study in all areas of science (Clery and Voss 2005, Szalay and Gray 2006, Strogatz 2001). A common feature of many natural systems that contributes to their complexity is the presence of dynamics simultaneously present at multiple time scales. Dynamical systems with multiple time scales are those in which different patterns of behavior predominate

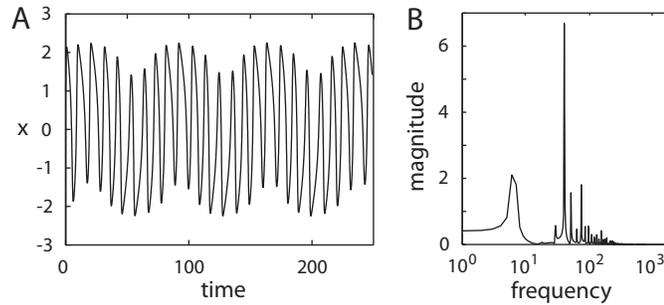


Figure 3.1. A forced Van der Pol oscillator time series (A) and its corresponding frequency spectrum (B).

when the system is viewed at different scales (Jones 2001). For example, the forced Van der Pol oscillator is used to model many natural and synthetic oscillating systems (Figure 3.1). The fast self-oscillations and slower forced oscillations are readily visible in the time series data for the dynamic variable x as shown in Figure 3.1a. The frequency spectrum of these data confirms the qualitative observation that the dynamics consist predominantly of those at two different time scales (Figure 3.1b).

Phenomena with multiple time scales arise in many different areas of science and engineering (Figure 3.2, Figure 3.3). Some examples include bursting activity in neurons (Harris-Warrick and Flamm 1987), the reaction kinetics of enzyme catalysis (Henzler-Wildman et al. 2007), the motion of the cytoskeleton within cells (Deng et al. 2006), and the behavior of flapping structures (Chung et al. 2004). Dynamics at multiple spatio-temporal scales are also frequently observed, such as in fluid mechanics (Klein 2001, Chen and Fish 2001) and patterns of ecological diversity (Pandolfi 2002, Borcard et al. 2004). There is a widening gap between our ability to generate data from complex systems such as these and our ability to distill that data into useful scientific models. Practitioners frequently note that automation will be necessary to close this gap by assisting scientists in the creation of interpretable

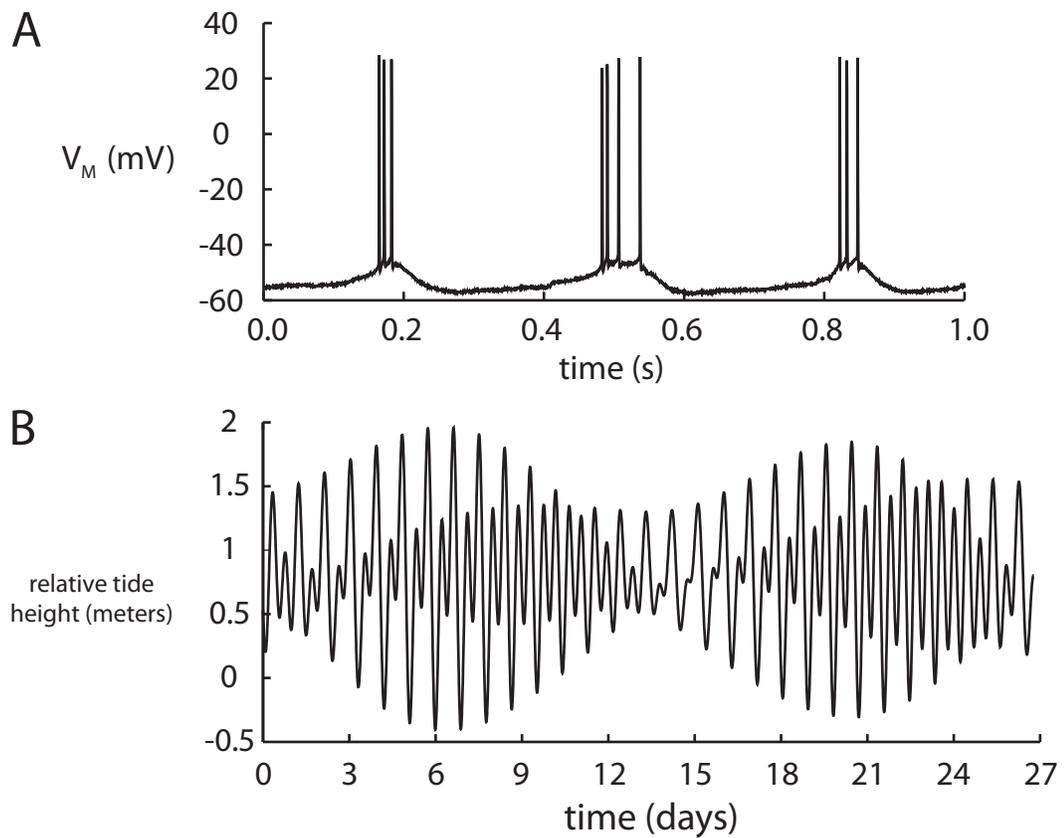
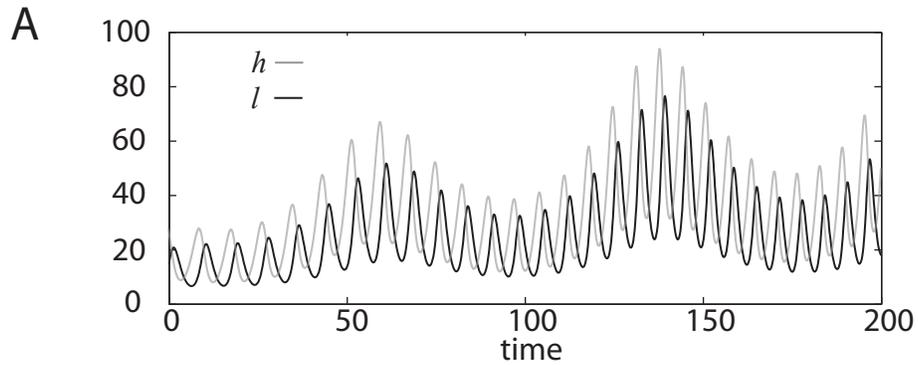


Figure 3.2. Natural dynamical systems with multiple time scales. **A)** One second of membrane voltage data collected from a mouse olfactory neuron. Data collected by and provided courtesy of Abdallah Hayar. **B)** Tide levels recorded at a beach over approximately one month. Data from <http://tidesandcurrents.noaa.gov>.

models (King et al. 2004, Waltz and Buchanan 2009, Evans and Rzhetsky 2010).

In fact, a wide variety of methods do exist for automating the modeling process. Familiar techniques such as linear and nonlinear regression are relatively easy to apply to empirical data, yet they assume that the structure of the model is substantially fixed in advance (Draper and Smith 1998). Other techniques such as neural networks and support vector machines can be more powerful, yet they tend to



B

$$\frac{dh}{dt} = \left\{ \frac{\cos(0.05t)}{32.1} + 0.434 + \frac{t^{0.456}}{23.456} \right\} h - \left\{ \frac{\sin(0.08t)}{100} + 0.051 + \frac{t^{0.201}}{122.456} \right\} hl$$

$$\frac{dl}{dt} = -(0.0020t + 0.8439)l + \left\{ \frac{\sin(0.08t)}{100} + 0.051 + \frac{t^{0.201}}{122.456} \right\} hl$$

Figure 3.3. Reverse-engineering multiple-time-scale dynamical systems. **A)** A simulation of a predator-prey system in which hare (h) and lynx (l) populations display fast oscillations as well as slower overall changes in the magnitude of those oscillations. **B)** The goal of the work in this chapter is to use data such as that shown in panel **A** to automatically infer an algebraic model for the system. The model here is equivalent to a classic Lotka-Volterra predator-prey system except that the constant coefficients of the dynamic variables are replaced by complex time-dependent functions (gray). These time-dependent coefficients are responsible for the slow changes in population size.

produce models that are difficult to interpret (Bishop 2006). To avoid such restrictions, “robot scientist” approaches have been developed that use inductive reasoning over databases of human-supplied background knowledge (Langley et al. 1987, King et al. 2009). Here, the focus is on symbolic regression, a machine learning technique that represents models as human-interpretable algebraic expressions and that places few constraints on the form of those expressions (Koza 1992). In addition, symbolic regression requires minimal human-supplied knowledge and can

automatically design models de novo directly from empirical data.

Different varieties of symbolic regression have been successfully used to find explicit linear and nonlinear models (Koza 1992, Koza 1994, Koza et al. 1999, Koza et al. 2003), implicit equations (Schmidt and Lipson 2009), and differential equations (Bongard and Lipson 2007) for problems in many fields of science and engineering. In addition, many studies have focused specifically on applications of symbolic regression to the analysis of time series data, with the goal of forecasting (Güven 2009, Mulloy et al. 1996) or the goal of reverse-engineering human-interpretable and parsimonious descriptive models (Bongard and Lipson 2007, Sakamoto and Iba 2001). Typically, symbolic regression is used to model time series as explicit linear or nonlinear algebraic equations (Jara 2011, Wagner et al. 2007), but recent advances have also improved the capabilities of symbolic regression for modeling time series as invariant equations (Schmidt and Lipson 2009) and ordinary differential equations (Schmidt et al. 2011).

Although these previously proposed algorithms are generally suitable for application to multiple-time-scale systems, they do not take advantage of the fact that dynamics present at multiple time scales are potentially separable. This discards important information about the system that could potentially be used to improve the accuracy and explanatory power of the constructed models. In this section, a new method is proposed to leverage information about multiple time scales in symbolic regression algorithms. The overall goal is to take measurements from a system, such as the observed hare and lynx population sizes shown in Figure 3.3a, and then use these data to reverse-engineer an algebraic model for the system (Figure 3.3b). The

proposed method specifically focuses on systems with multiple time scales and is based on the idea that dynamics at different scales can be separated and independently modeled. A complete model describing dynamics at all scales can then be composed from these independent models. To study the proposed algorithm, experiments were performed in which both synthetic and physical systems were modeled. In addition, the factors that influence performance of the algorithm were studied in detail.

Methods

A summary of the proposed algorithm is shown in pseudocode in Figure 3.4. The algorithm involves three main steps. In Step 1, an observed time series is divided into small windows such that slow dynamics in the data are approximately constant. The data in each window are modeled with symbolic regression and the models obtained from several different windows are stored and ranked. In Step 2, the best models from Step 1 are considered one at a time. Any numerical constants in each model are replaced with “placeholder” variables. Ordinary nonlinear regression is then used to find values of the placeholder variables such that the expression fits the observed data for a small portion of the original data set. This process of finding values for the placeholder variables is repeated for several samples taken from different portions of the original data set. In Step 3, the values of the placeholder variables found in Step 2 are processed with an additional round of symbolic regression. As the fitted values of the placeholder variables are time-dependent, they represent time series data sets that can themselves be modeled with symbolic

```

Begin Algorithm_1(input DATA)

  Step 1:

    Divide DATA into a set W of (possibly overlapping) windows.

    For each window in W, perform symbolic regression and add output candidate
    models to bag of models B.

    Rank models in bag B and select a subset M of the best models. Denote the
    number of coefficients in model  $M_i$  as  $k_i$ .

  Step 2:

    For each model  $M_i$ 

      Replace the  $k_i$  coefficients with  $k_i$  placeholder variables  $\{c_1, c_2, c_3, \dots, c_{k_i}\}$ .

      Divide DATA into a set V of (possibly overlapping) windows.

      For each window  $V_j$ 

        Fit model  $M_i$  to the data in the window with nonlinear regression.

        Store fitted placeholder variables in row j, columns 1 ...  $k_i$  of a
        table  $T_i$ . Also store the time value at the midpoint of window  $V_j$  in row
        j, column  $(k_i + 1)$  of table  $T_i$ .

      End For

    End For

  Step 3:

    For each table  $T_i$ 

      For each column  $q \in \{1, 2, 3, \dots, k_i\}$  in table  $T_i$ 

        Perform symbolic regression with column q as the dependent variable and
        the time value in column  $(k_i + 1)$  as the independent variable. Rank
        the candidate output models and choose the best to replace coefficient
         $c_q$  in model  $M_i$ .

      End For

    End For

  Return M

End Algorithm_1

```

Figure 3.4. Proposed algorithm pseudocode.

regression. The models resulting from this step are finally substituted back into the corresponding placeholder variables from which they were derived. This three step process of separating and independently modeling dynamics at different time scales can be repeated as needed for data with more than two time scales.

In the next three subsections, the algorithm will be informally illustrated using the multiple-time-scale dynamical system shown in Figure 3.3. For the purposes of this explanation, it will be assumed that both the hare (h) and lynx (l) data are measured and that the task of finding a model $dh/dt = f(h, l, t)$ can be considered separately from the task of finding a model $dl/dt = f(h, l, t)$. In other words, the modeling problem is partitioned, and numerical solution of the system of differential equations is not necessary as part of the modeling process (Bongard and Lipson 2007). Also for purposes of illustration, only the task of using the observed hare and lynx data to find a model for dl/dt will be described.

Algorithm Step 1

The algorithm begins by dividing the time derivative of the observed lynx data into small windows as shown in Figure 3.5a. These windows can overlap and their size is chosen such that all but the fastest dynamics are approximately constant over the time span of the window. Simple inspection of the data appeared to be sufficient for choosing the size of the window in this example and in the examples presented in the **Results** section below. However, more sophisticated potential methods for selecting the size of the window are considered in the **Discussion** section below. In the example shown in Figure 3.5, one of these windows consists of the data between the dashed lines. Within this window, the time-dependent coefficients of the l and hl terms are roughly constant, and so the complex ground truth model can be approximated by a much simpler local model (Figure 3.5b). Any symbolic regression

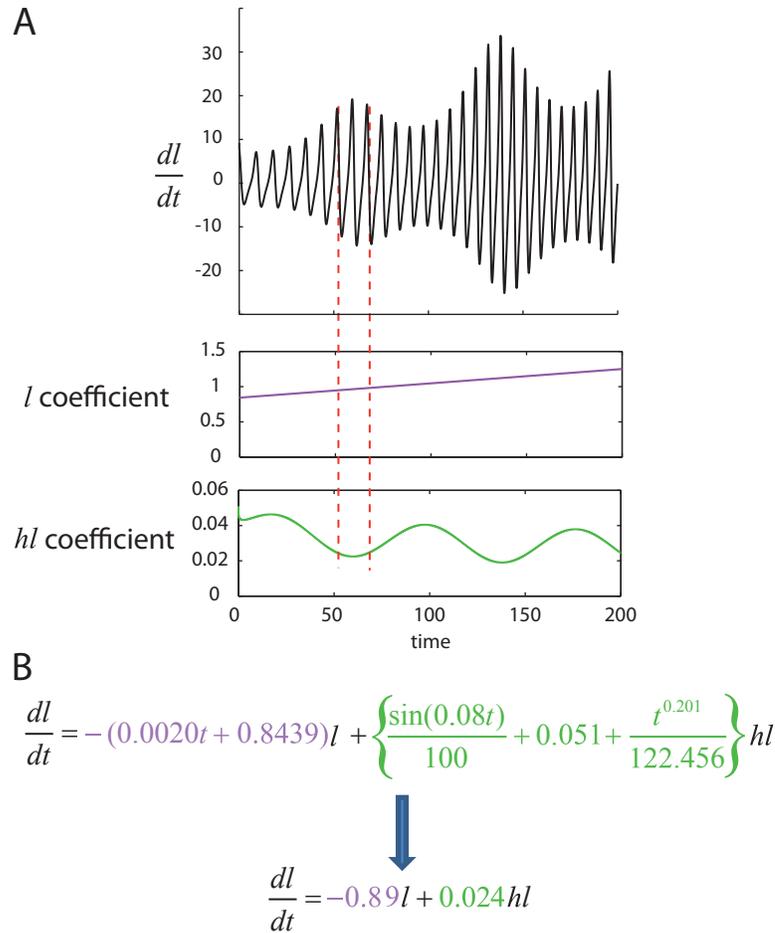


Figure 3.5. Algorithm Step 1: modeling fast dynamics. **A)** The time derivative dl/dt is estimated from the original lynx data shown in Figure 3.3. Windows are then used to divide dl/dt into smaller data sets over which slower components of the dynamics are approximately constant. **B)** Symbolic regression is used to find the relatively simple model that approximates the lynx dynamics within the window bounded by the two dashed lines. This is repeated for several windows, giving a bag of models.

algorithm can then be used find this relatively simple expression when given the dl/dt data from within the window. Repeating this process for each window in the original data results in several different candidate models of the simpler form shown in Figure 3.5b. This set of candidate models will be referred to below as the “bag” of models produced by Step 1 of the algorithm.

Algorithm Step 2

In the next step of the algorithm, the bag of models generated in Step 1 is processed. Although models in the bag could be merged or otherwise processed in many different ways, here individual candidate models are simply selected without regard to the window from which they were obtained. Once a model is selected from the bag (Figure 3.6a), all numerical constants in the model are replaced with placeholder variables $\{a, b, c, \dots\}$. The resulting model will be referred to as a “general form expression” (Figure 3.6b). The placeholder variables are then taken to be the adjustable parameters in a series of nonlinear regression problems in which the general form expression is locally fit to the dl/dt data at several different points (Figure 3.6c). For these experiments, the size of the local windows for which these local nonlinear regression fits were performed was set to the size of the windows used in Step 1 of the algorithm, although more sophisticated methods of choosing window size are considered below.

The Levenberg-Marquardt algorithm (Nocedal and Wright 2006) with random-restarts was found to be an efficient means of performing the large number of nonlinear regression runs that are necessary. To accelerate the rate of achieving good local fits, the initial estimates used at the start of each nonlinear regression run were set to the fitted values of the placeholder variables found with nonlinear regression in the previous segment of the dl/dt data. The initial estimates for the first nonlinear regression run were randomly chosen. Overall, the computational effort required for these repeated nonlinear regression fits was found to be much lower than that required for symbolic regression, especially when considered on a per-CPU-core basis.

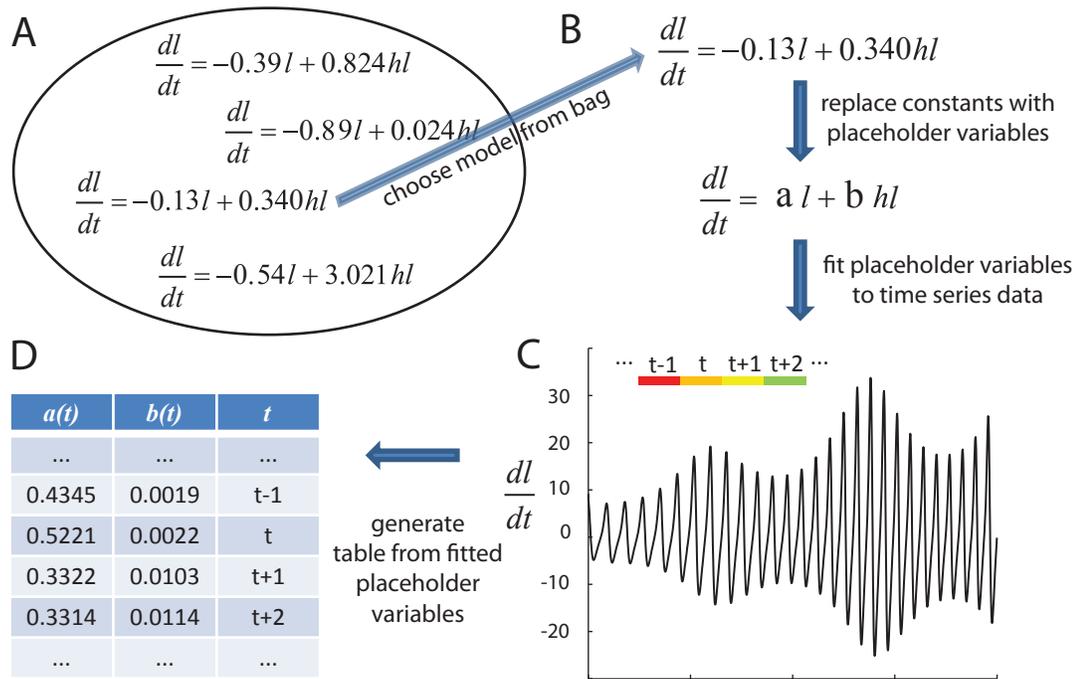


Figure 3.6. Algorithm Step 2: separating slow dynamics from the data. Expressions are taken from the bag of models obtained in Step 1 (A). The coefficients in these expressions are then replaced with placeholder variables (a , b , c , etc.), resulting in general form expressions (B). Ordinary nonlinear regression is used to locally fit these placeholder variables for several locations (\dots , $t - 1$, t , $t + 1$, \dots) across the time span of the original data (C). The result of Step 2 is a table (D) that contains the fitted value for each placeholder variable as a function of time t .

The result of performing local nonlinear regression for several windows spanning the observed dl/dt data can be viewed as a table as shown in Figure 3.6d. Each row of this table contains the results of fitting the general form expression to one small window within the data. The first N columns of a given row in the table are the fitted values of the N placeholder variables, and column $(N + 1)$ is the time point at the center of the window for which the local nonlinear regression was performed.

Algorithm Step 3

In Step 3 of the algorithm, the tables produced in Step 2 are processed with additional rounds of symbolic regression as shown in Figure 3.7. In a given table with $(N + 1)$ columns, each of the first N columns represent the evolution of a placeholder variable as a function of time, where time is given in column $(N + 1)$. Together these two columns constitute a new set of time series data that can be modeled with symbolic regression just as the original data were in Step 1 (Figure 3.7a). The resulting model can then be substituted into the general form expression where it replaces the corresponding placeholder variable wherever it appears (Figure 3.7b). This procedure is repeated for additional columns in the table and finally results in a complete model for the observed lynx data (Figure 3.7c).

Symbolic Regression

The algorithm proposed in this chapter relies on both nonlinear regression and symbolic regression as sub-algorithms. Here, the details of the symbolic regression algorithm used will be described in more detail. Like other types of regression, symbolic regression is concerned with the identification of a mathematical model of a system based on experimental observations of the system. However, symbolic regression identifies both the algebraic structure and parameters of these models, and this distinguishes it from other methods such as linear and nonlinear regression that are capable only of optimizing the parameters of a model with a pre-defined structure (Koza 1992). The primary downside of symbolic regression compared with other types of regression is the much greater computational effort involved in the search

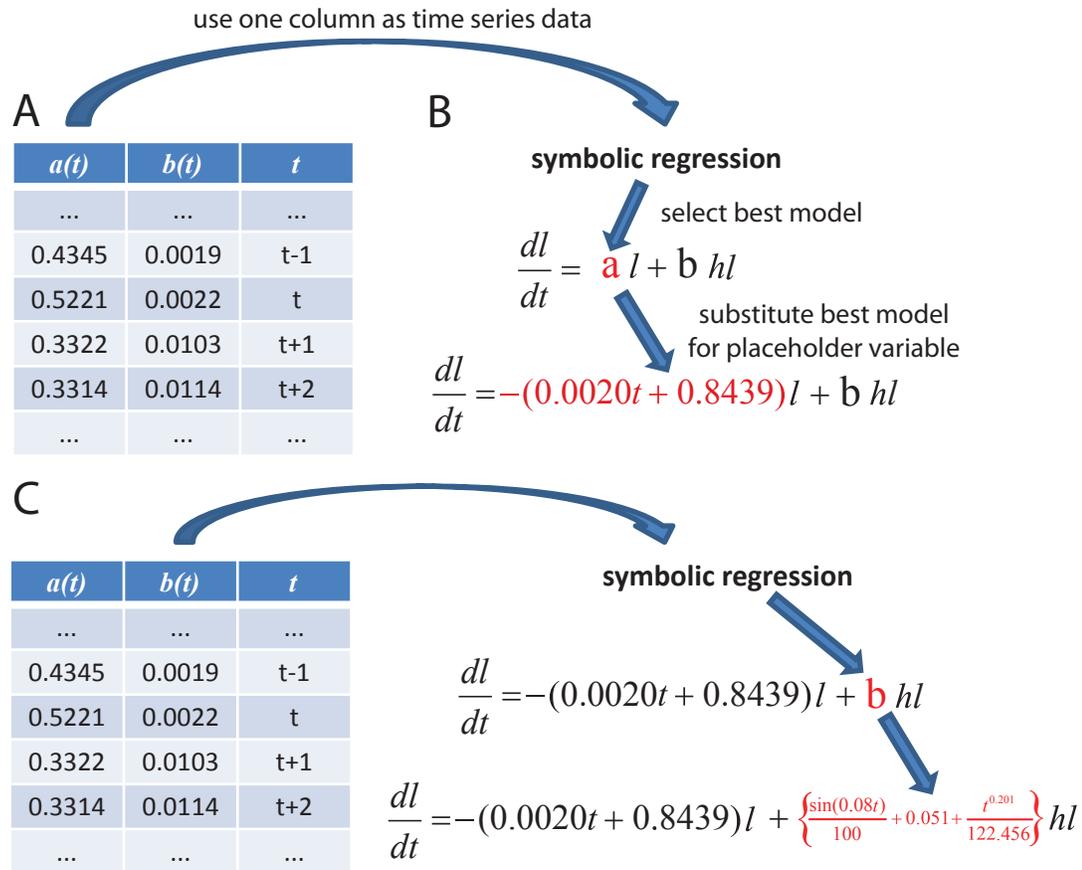


Figure 3.7. Algorithm Step 3: modeling slow dynamics and completing model inference. **A)** As each column of a table obtained previously in Step 2 is a new set of time series data, it can be modeled with symbolic regression. **B)** The best model resulting from this symbolic regression step is a model of the slow dynamics that is used to replace the corresponding placeholder variable in the general form expression. **C)** Repeating this process for every column in the table results in the completed expression for dl/dt .

through the space of possible algebraic expressions. Population-based stochastic optimization techniques such as genetic programming have proven to be particularly effective means of performing this search (Koza 1992, De Jong 2006).

In genetic programming, expressions are typically represented as trees or other similar types of graphs. The nodes of these graphs correspond to mathematical building blocks from which candidate models can be constructed. These building

blocks may include primitive operations such as $+$ or \cos , as well as operands such as the independent variables of the model or floating point constants. Biologically-inspired operations such as mutation and crossover are used to introduce and maintain variation in the population of candidate models as the search proceeds. Mutation is used to change individual nodes in an expression graph whereas crossover exchanges sub-graphs between two different models. Candidate models in the population are selected to survive and reproduce on the basis of a fitness objective that measures the difference between the behavior predicted by the model and that observed in the target system. In this work, fitness is defined as the mean absolute error (MAE):

$$\text{fitness}(f) = \frac{1}{N} \sum_{i=1}^N |f(x_i) - y_i| \quad (3.1)$$

where f is a candidate model, x_i and y_i are the independent and dependent variables of the training data, respectively, and N is the number of examples in the training data.

Some experiments described below involved random target functions. To generate random target functions, random binary trees of operations and operands were generated substantially as described in Schmidt and Lipson 2008. The expressions encoded by these trees were then symbolically simplified using MATLAB's Symbolic Math Toolbox⁶ in order to obtain an accurate measure of the random functions' complexity. This procedure was repeated until a large number of unique functions across a range of several complexities were obtained. Appendix 6 shows several examples of random functions with multiple time scales that were

⁶ <http://www.mathworks.com/products/matlab>

generated in this manner.

Throughout this work, the Eureqa algorithm was used for all symbolic regression (Schmidt and Lipson 2009). Eureqa implements the basic features of genetic programming as described above and extends it with several non-standard features. These additional features include the use of directed acyclic graphs to represent expressions (Schmidt and Lipson 2007), the coevolution of rank predictors to accelerate evaluation of candidate models (Schmidt and Lipson 2008, Schmidt and Lipson 2010c), and the use of Age-Fitness Pareto optimization for bloat control and diversity maintenance (Schmidt and Lipson 2010a-b). See these references for more details about Eureqa.

Results

To test the proposed algorithm, it was applied to four systems with multiple time scales. Two of these were synthetic systems with simulated data and two were physical systems.

Application to Synthetic Systems

Two synthetic systems were studied: the one-dimensional “sum-of-sines” test system and a more complex three-dimensional oscillating system. The ground truth equations for the two synthetic problems and samples of data generated from them are shown in Figure 3.8 (first and second rows). 50 independent trials of the proposed algorithm were run with the goal of reverse-engineering the target expressions. As symbolic

regression alone is applicable to the same task, 50 independent trials of symbolic regression alone were also run with the same targets as a control to gauge whether the proposed algorithm was able to obtain better models with the same amount of computational effort. In all 100 trials for one of the modeling problems, the total number of candidate model evaluations was the same. The ground truth equations were used to generate training data, which were used during model inference. These ground truth equations were also used to generate separate test data, which were not used during model inference but were used as the basis for all performance results reported below. Gaussian noise $\sim N(0, 0.01)$ was added to the training data for the oscillator problem and varying amounts of noise were added to the training data for the sum-of-sines problem as described below.

The best models produced in each of the 50 trials were pooled and ranked by mean absolute error on the test data. The target, best, and median models obtained by the proposed algorithm and by symbolic regression alone are compared in Table 3.1. In addition, the best models produced by the proposed algorithm and by symbolic

Figure 3.8. The four systems used to test the proposed algorithm (next page). The first column shows the target systems, which are only approximate theoretical models in the case of the two physical systems: the cantilever beam and the 5-azacytidine system. The second column shows the data measured from the corresponding system, which are the input to the proposed algorithm or to symbolic regression alone. The third column shows the best model inferred by the proposed algorithm. Finally, the fourth column summarizes the results of 50 trials of the proposed algorithm and symbolic regression alone. Light gray bars are results for the proposed algorithm, dark gray bars are for symbolic regression alone, and the error bars are \pm standard error of the mean for 50 independent trials of the algorithms. All differences between algorithms except for the 5-azacytidine kinetics problem are significant by an unpaired t -test, $p < 0.05$. Values for the 5-azacytidine kinetics problem are multiplied by 1000 for easier visual comparison with the other results.

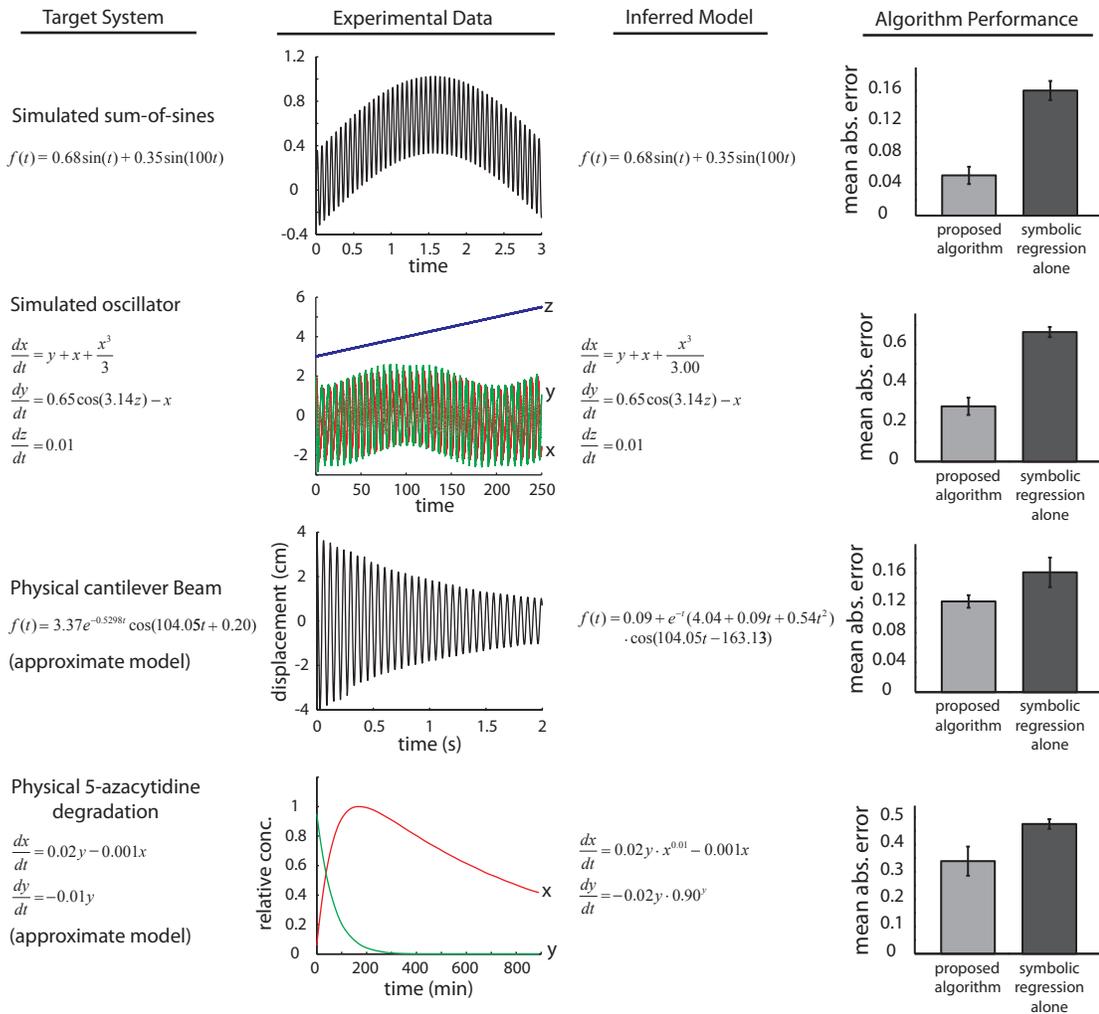


Table 3.1. Models produced by the proposed algorithm and by symbolic regression alone for the two synthetic systems: the sum-of-sines system (A) and a Van der Pol Oscillator (B).

A) sum-of-sines

target
 $f(t) = 0.67890 * \sin(t) + 0.34560 * \sin(100 * t)$

best, proposed algorithm
 $f(t) = 0.67890 * \sin(t) + 0.34560 * \sin(100 * t)$

median, proposed algorithm
 $f(t) = 0.67879 * \sin(0.00011 + t) + 0.34580 * \sin(99.98260 * t - 6.14881)$

best, symbolic regression alone
 $f(t) = 0.67890 * \sin(t) + 0.34560 * \sin(100 * t)$

median, symbolic regression alone
 $f(t) = \sin(t) - 0.65618 * \sin(t) * \sin(-49.55108 * t) * \sin(-49.55108 * t)$

B) Van der Pol oscillator

target
 $dx/dt = y + x - x^3/3$
 $dy/dt = 0.65430 * \cos(3.14159 * z) - x$
 $dz/dt = 0.01000$

best, proposed algorithm
 $dx/dt = y + x - x^3/3.00049$
 $dy/dt = 0.65432 * \cos(-3.14181 * z) - x$
 $dz/dt = 0.00995$

median, proposed algorithm
 $dx/dt = y + x - x^3/3.00041$
 $dy/dt = -0.65948 * \cos(9.13489 - 3.06381 * z) - 0.99125 * x$
 $dz/dt = 0.00997$

best, symbolic regression alone
 $dx/dt = y + 1.00084 * x - x^3/3.00147$
 $dy/dt = 0.65309 * \cos(-3.14095 * z) - x$
 $dz/dt = 0.00996$

median, symbolic regression alone
 $dx/dt = y + 0.99894 * x - x^3/3.00302$
 $dy/dt = z + 3.73327 * \cos(z) - x$
 $dz/dt = 0.01001$

regression alone are shown in the third column of Figure 3.8. For both synthetic problems, the proposed algorithm outperformed symbolic regression alone. Although the inferred models with lowest error found with both algorithms closely matched the target models, modeling with the proposed algorithm appeared to be more robust. In particular, the structure of models with higher error tended to differ from the target to a greater extent with symbolic regression alone as can be seen by comparing the median models in Table 3.1. This is also reflected in the mean absolute error of the models produced by the proposed algorithm and by symbolic regression alone as shown in the fourth column of Figure 3.8.

Several factors could potentially influence the accuracy and efficiency of the proposed algorithm. These include two factors beyond the control of the investigator: the presence of noise in the training data and the magnitude of differences between time scales inherent in the system. In addition, the choice of window size and the total amount of computational effort used are two factors under the control of the investigator that are likely to impact the performance of the algorithm. To investigate the effects of these four factors, the sum-of-sines test problem shown above was used except that the influence of the four factors was systematically tested. As before, the performance of the proposed algorithm was compared with that of symbolic regression alone. 50 independent trials were performed for each and the same amount of computational effort used in all trials. A summary of these results is shown in Figure 3.9. For relatively low noise levels (Figure 3.9a) and a large difference between the two time scales (Figure 3.9b), the proposed algorithm outperforms symbolic regression alone. However, at higher noise levels and less separation between time

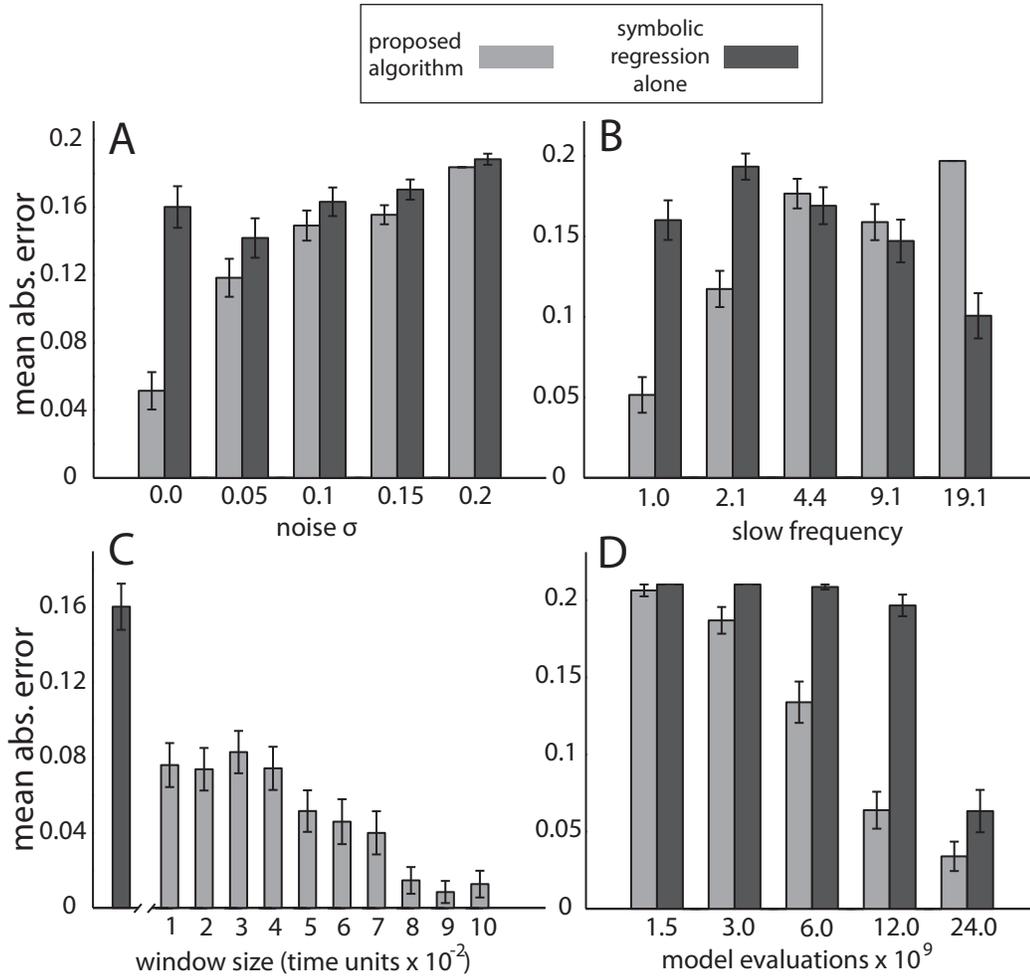


Figure 3.9. Effects of noise, magnitude of time scale differences, window size, and computational effort on algorithm performance. For these experiments, the target for the sum-of-sines problem was changed from that shown in Figure 3.8 to:

$$f(t) = 0.67890 \cdot \sin(\Omega \cdot t) + 0.34560 \cdot \sin(100 \cdot t) + N(0, \sigma)$$

where Ω is the frequency of the slow component of the dynamics and σ is the standard deviation of Gaussian noise added to the signal. The effect of varying σ on the mean absolute error of the best generated models is shown in panel **A** and the effect of varying Ω is shown in panel **B**. For the sum-of-sines problem with no noise and maximum separation of time scales, the effect on mean absolute error of varying window size is shown in panel **C** and the effect of varying the amount of computational effort used is shown in panel **D**. Light gray bars are results for the proposed algorithm, dark gray bars are for symbolic regression alone, and all error bars represent \pm standard error of the mean for 50 independent trials of the algorithms.

scales, this advantage decreases. For the sum-of-sines problem with no noise and maximum separation between time scales, the advantage of the proposed algorithm over symbolic regression alone is robust over a wide range of window sizes (Figure 3.9c) and levels of total computational effort (Figure 3.9d).

Finally, to test the performance and scalability of the proposed algorithm in as unbiased a manner as possible, hundreds of random target problems with two time scales and different complexities were generated. An example of a random target problem is shown in Figure 3.10a. The proposed algorithm outperformed symbolic regression alone across a range of target complexities (Figure 3.10b).

Application to Physical Systems

Two physical systems were studied: a vibrating cantilever beam modeled as a one-dimensional function of time (Le Pourhiet et al. 2003), and the two-dimensional 5-azacytidine kinetic system (Beisler 1978, Argemí and Saurina 2007). The application of the proposed algorithm to the vibrating cantilever beam was straightforward as the vibrations of the beam represent the fast component of the dynamics and the gradual decline of the magnitude of these vibrations due to damping represents the slow component of the dynamics (Figure 3.11). In contrast, the kinetic system displays a distinctive type of behavior known as fast-slow dynamics (Palsson 1987, Chen et al. 2010). The fast phase dominates very early in the chemical reaction while the slow phase dominates later. This contrasts with a system such as the sum-of-sines problem in which the fast and slow dynamics are superimposed throughout the observed time series. However, a variation on the proposed algorithm for reverse-engineering

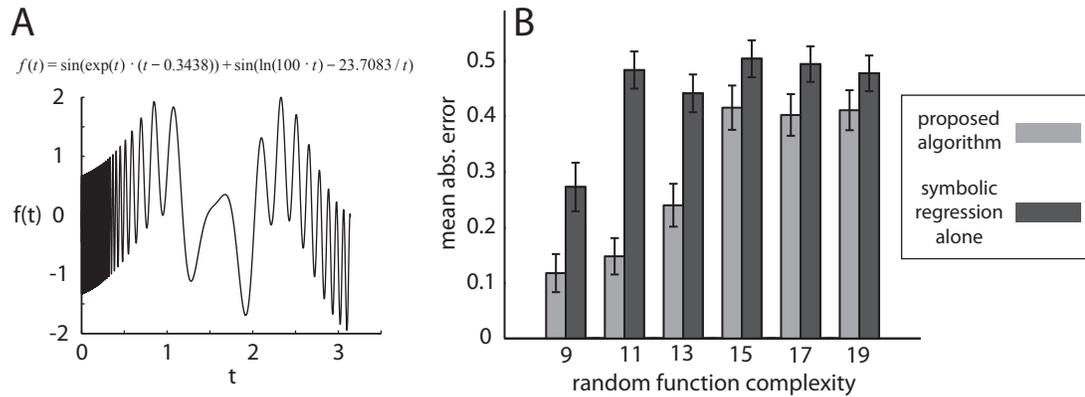
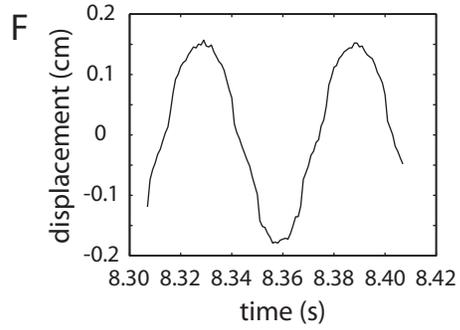
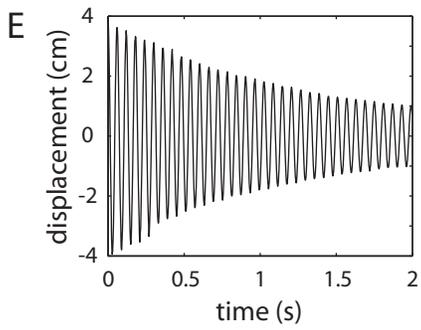
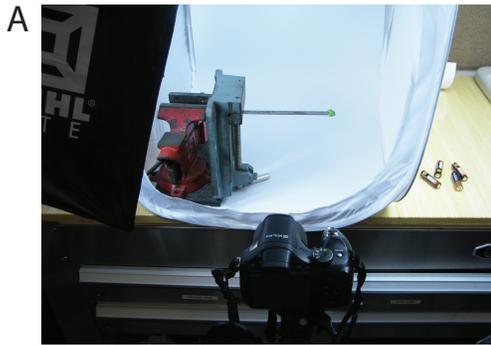


Figure 3.10. Algorithm performance on random target functions of different complexities. **A)** An example of a random target function with a complexity of 17. Complexity is quantified as the number of operators and operands in a function. **B)** The effects of varying random target function complexity on algorithm performance. Light gray bars are results for the proposed algorithm, dark gray bars are for symbolic regression alone, and all error bars represent \pm standard error of the mean for 50 independent trials of the algorithms. The same number of model fitness evaluations was used in all trials. A unique random target function was generated for each trial within each complexity class.

Figure 3.11. Modeling process for the physical cantilever beam (next page). **A)** A simple beam was fixed firmly at one end with a vise and allowed to freely vibrate at the other end. To minimize noise that might interfere with image processing algorithms, the apparatus was positioned inside an enclosure with which background and lighting could be controlled easily. **B)** A green rubber band was used as a marker to facilitate optical tracking of the tip of the beam. **C)** The tip of the beam was displaced approximately 4 cm from its resting position and then released. The resulting motion of the beam was continuously photographed for approximately 30 seconds at 1000 frames per second and a resolution of 224x56 pixels. The green marker is not shown in this photograph to allow for observation of the entire beam's motion. **D)** One frame from the raw image data. The green marker is visible at the tip of the beam on the right. **E)** Offline analysis was used to process the image data and the displacement of the tip of the beam relative to the neutral position (0 cm) was optically tracked. Only the first 2 seconds of the resulting data after the initial perturbation are shown here. **F)** A different portion of the same image data is shown here on a smaller time scale to illustrate the noise level in this empirical data. **G)** With no information other than this raw data, the proposed algorithm generated parsimonious and interpretable symbolic models such as this.



G

↓

$$f(t) = 5.06e^{-t} \cos(104.05t - 31.19)$$

multiple-time-scale dynamical systems can be employed for fast-slow systems as well. Instead of using windows to decompose superimposed dynamics at different time scales, the data are decomposed into a very early phase when the fast dynamics are expected to dominate and a later phase when the slow dynamics are expected to dominate. A model of each of these components is inferred separately and then finally combined into a complete model.

Approximate theoretical models for the two physical systems derived from first principles are shown in Figure 3.8, column 1. The experimental data from the systems are shown in Figure 3.8, column 2. Reverse-engineering was performed as for the synthetic systems with 50 independent trials, and the same amount of computational effort was used for the proposed algorithm and symbolic regression alone. As with the synthetic systems, the best models produced in each of the 50 trials were pooled and ranked by mean absolute error on the test data. The target, best, and median models obtained by the proposed algorithm and by symbolic regression alone are compared in Table 3.2. Along with the theoretical models, the best models from 50 trials of the proposed algorithm are shown in Figure 3.8, column 3. The proposed algorithm outperformed symbolic regression alone for both physical systems as shown in Table 3.2 and Figure 3.8, column 4.

Discussion

In this chapter, a divide-and-conquer approach to the automated modeling of dynamical systems with multiple time scales was proposed. For both synthetic

Table 3.2. Models produced by the proposed algorithm and by symbolic regression alone for the two physical systems: a cantilever beam (A) and the 5-azacytidine reaction system (B).

A) cantilever beam

approximate theoretical model
 $f(t) = 3.3672 * \exp(-0.5298 * t) * \cos(104.0514 * t + 0.19914)$

best, proposed algorithm
 $f(t) = (0.09382 + \exp(-t) * (4.03615 + 0.09041 * t + 0.54199 * t^2)) * \cos(104.0481 * t - 163.1311)$

median, proposed algorithm
 $f(t) = (0.46968 + 3.59458 * \exp(-0.98795 * t)) * \cos(-104.04810 * t - 31.18420)$

best, symbolic regression alone
 $f(t) = (1.47585 + 0.97938 * \exp(0.99979 - 1.35988 * t))$

median, symbolic regression alone
 $f(t) = 2.57905 / (0.58074 + t) * \cos(104.52440 * t) + 0.30671 * \cos((\cos(0.01356 / (t^2)) + 3.86792 / t + 104.92515) * t)$

B) 5-azacytidine kinetics

approximate theoretical model
 $dx/dt = 0.01871 * y - 0.00125 * x$
 $dy/dt = -0.01490 * y$

best, proposed algorithm
 $dx/dt = 0.01799 * y * x^{0.00946} - 0.00132 * x$
 $dy/dt = -0.01628 * y * 0.90006^y$

median, proposed algorithm
 $dx/dt = 0.01778 * y - 0.00132 * x$
 $dy/dt = -0.01695 * y * 0.82759^y$

best, symbolic regression alone
 $dx/dt = (0.12748 * y - 0.00847 * x) / (6.43765 + y^{2.73063})$
 $dy/dt = -0.01660 * y^2 / (y - 7.33560) - 0.01660 * y$

median, symbolic regression alone
 $dx/dt = 0.01906 * y^2 + 0.01906 * x * y - 0.00139 * y - 0.00139 * x$
 $dy/dt = 0.00283 * y^2 - 0.01684 * y$

problems in which an exact underlying model is known and for physical systems with only approximate theoretical models, the algorithm consistently found models that matched the accuracy and parsimony of the targets. Although symbolic regression is

an important sub-algorithm within the approach proposed here, results suggest that the proposed algorithm can outperform symbolic regression alone with little additional computational effort. By improving the scalability of the automated modeling process, this algorithm helps to meet the challenge posed by the ever-increasing complexity of systems studied throughout science and engineering.

Most theoretical and computational work in the area of multiple-time-scale dynamical systems has focused on a technique known as singular perturbation theory and its related techniques (Jones 2001, Verhulst 2005). The goal in this body of work is to decompose models of multiple-time-scale systems into independent single-scale systems that approximate the behavior of one of the scales in the original system (Chen et al. 2010, Lee and Othmer 2009). Although singular perturbation and its extensions are powerful techniques, they are used for a fundamentally different purpose than the algorithm proposed here. Singular perturbation starts with a complete, known model and then proceeds to analyze and reduce that model, whereas the focus of the present work is on the *de novo*, automated design of the complete model using only empirical observations. This makes the algorithm proposed here more closely related to the divide-and-conquer curve fitting technique MFIT, in which high dimensional regression problems with several independent variables are reduced to a series of single-dimensional regression problems (Lawlor 1973).

One of the algorithm's main assumptions is that windows of an appropriate size are chosen in Step 1 and Step 2, yet a priori information for making this choice would generally not be available. The results shown in Figure 3.9c suggest a degree of insensitivity to the exact choice of window size, and in the current work it was

found that simple inspection of the data was sufficient to determine reasonable window sizes. However, many possibilities exist for a more principled method of identifying window size. For systems with periodic dynamics, one such approach is to find the frequency spectrum of the signal as in Figure 3.1b and use it to estimate the optimal window length that would separate faster components of the signal from slower components. This could be repeated for the next fastest component of the dynamics, and then the next fastest, etc. until window sizes appropriate for dynamics at each time scale are identified. More advanced techniques such as wavelet analysis (Akansu and Haddad 2000) could also be used to decompose signals into components that dominate at different time scales.

Systems with multiple spatial scales, multiple spatio-temporal scales, or with more than two time scales were not considered in the examples here but should be amenable to analysis by the proposed algorithm with little or no modification. Rather, any limitations of the algorithm would likely be due to noise in the data, the resolution of the data, the magnitude of separation between spatial or temporal scales, and the computational effort required for the model search. The resolution of the data set and the magnitude of separation between spatial or temporal scales would generally not be under the control of the experimenter, but sophisticated denoising techniques such as unscented Kalman filtering (Julier et al. 2000) could reduce problems posed by noise in the data.

The problems here were chosen to be relatively small so that several replicate trials could be performed with both the proposed algorithm and symbolic regression alone. It is likely that the relative benefits of the proposed algorithm over symbolic

regression alone will increase as the difficulty of the problem increases. However, further work is needed to rigorously test this hypothesis and to gain insight into how the performances of both algorithms scale with problem difficulty.

CHAPTER 4

INFERENCE OF HIDDEN VARIABLES

Introduction

The data-driven modeling of dynamical systems is an important scientific activity, and many studies have applied genetic programming (GP) to the task of automatically constructing such models in the form of systems of ordinary differential equations (ODEs). These previous studies assumed that data measurements were available for all variables in the system, whereas in real-world settings, it is typically the case that one or more variables are unmeasured or “hidden.” In this chapter, the prospect of automatically constructing ODE models of dynamical systems from time series data with GP in the presence of hidden variables is investigated. Several examples with both synthetic and physical systems demonstrate the unique challenges of this problem and the circumstances under which it is possible to reverse-engineer both the form and parameters of ODE models with hidden variables.

Background

Models that organize, explain, and predict empirical observations play a central role in science. The form taken by a model is generally problem-specific and must be designed by a human expert, usually at the expense of considerable time and

effort. In contrast, the rate at which scientists are able to automatically generate empirical data is expected to steadily increase for the foreseeable future, as is the complexity of problems routinely studied in all fields (Clery and Voss 2005, Strogatz 2001, Strogatz 2007, Szalay and Gray 2006). As a result, there is a widening gap between our ability to generate data and our ability to distill data into useful scientific models. New computational methods will be necessary to close this gap by assisting humans in the creation of scientific models (Evans and Rzhetsky 2010, King et al. 2004, Waltz and Buchanan 2009).

Genetic programming (GP) and related machine learning techniques based on evolutionary computation have been proposed as a means of automatically creating models with minimal human effort and minimal supplied background knowledge (Koza 1992, Banzhaf et al. 1997). Traditional modeling techniques such as nonlinear regression and Hidden Markov Model inference typically require the model to have a generic structure substantially specified in advance by a domain expert. Other techniques such as neural networks and autoregressive models are predictive of unseen data but can be difficult to interpret. In contrast, GP places minimal constraints on model structure, requires little or no domain knowledge, and is capable of automatically inferring parsimonious explanatory models such as those that would ordinarily be hand-crafted from first principles by a human expert (Schmidt and Lipson 2009). Such models are particularly valuable as they explicitly represent scientific knowledge and hypotheses as interpretable analytical expressions (Schwabacher and Langley 2001).

GP has been successfully applied to many symbolic modeling problems, including those involving data with measurements that vary over time (Kaboudan 2000, Mulloy et al. 1996, Neely et al. 1997, Oakley 1994, Rodriguez-Vázquez and Fleming 2005, Schmidt and Lipson 2009). The models of these dynamical systems are in many cases most naturally represented as a system of ordinary differential equations (ODEs). In fact, several previous studies have used GP and closely related techniques to infer ODEs based on experimental data and to garner insight into the dynamical systems under study (Andrew 1996, Angeline and Fogel 1997, Babovic and Keijzer 2000, Bernardino and Barbosa 2011, Bongard and Lipson 2007, Koza 1002). Many authors have also combined GP for structural optimization with local search algorithms for improved estimation of parameters in the context of ODE model inference (Ahalpara and Sen 2011, Ando et al. 2002, Cao et al. 2000, Gray et al. 1997, Gray et al. 1998, Iba et al. 1994, Iba 2008, Qian et al. 2008, Sakamoto and Iba 2001, Winkler et al. 2005).

It is often the case that dynamical systems encountered in the real world are modeled with systems of differential equations in which one or more of the dynamic variables are unmeasured or “hidden.” As an example, consider the ideal rocket equations, which describe the basic Newtonian mechanics of rocket motion. When written explicitly as a system of ordinary differential equations, these rocket equations take the form:

$$\begin{cases} \frac{dX}{dt} = V \\ \frac{dV}{dt} = \frac{mV_e}{M} \\ \frac{dM}{dt} = -m \end{cases} \quad (4.1)$$

Here, X is the position of the rocket at time t , V is the velocity of the rocket at time t , M is the mass of the rocket at time t , m is the (constant) rate at which mass is lost by the rocket due to fuel expenditure, and V_e is the (constant) velocity of the mass lost by the rocket. The position over time, and by extension, the velocity of the rocket are easy to measure by an outside observer, whereas the mass of the rocket cannot be measured directly. However, it is essential to include the concept of mass for accurately and parsimoniously describing the mechanics of rocket motion, and so the mass is a hidden variable. Hidden variables are often a component of models such as the rocket equations that provide an explanation of the modeled phenomenon. This is fundamentally different from a model such as an autoregressive polynomial that would be merely descriptive of the observed data (Bridewell et al. 2008).

Hidden variables frequently arise in real world settings in all fields of science, usually when it is difficult or impossible to measure one or more of the system's dynamic variables. A classic example from biology is the set of ion channel gating variables in the Hodgkin-Huxley ODE system describing action potentials, which is shown in Equation 2.1 (Hodgkin and Huxley 1952d). Other examples include the latent prey subpopulations in predator-prey systems with rapid evolution (Yoshida et al. 2003), the concentration of several of the species in a large chemical reaction

network (Cho et al. 2003), and the components of a dynamical model of DC motors (Mamani et al. 2008).

Any practical system that uses GP to automatically construct ODE models of real world dynamical systems must take into account the presence of hidden variables. However, to date no previous study has addressed this issue, possibly because hidden variables present several challenges to automated modeling that are not present when all variables in the system are measured. Chief among these challenges is the inherent difficulty of identifying unique ODE models containing hidden variables that match given experimental observations. The possibility of obtaining meaningful models despite this and other difficulties is explored here. Using examples from both synthetic and physical dynamical systems, it will be shown that under many circumstances, it is possible to automatically construct both the form and parameters of ODE models with hidden variables using GP.

Related Work

The inference of ODE models both with and without hidden variables has a long history. This section briefly reviews this work and contrasts it with the GP-based approach that is the main contribution of this chapter. Although no single review article encompasses all work in this large area, comprehensive reviews covering much of the area include Sjöberg et al. 1995, Voss et al. 2004, Aguirre and Letellier 2009, and Hong et al. 2008.

Much of the previous work in the inference of ODE models has been undertaken in the broad field of linear and nonlinear system identification (Åström and Eykhoff 1971, Baake et al. 1992, Ljung 1999). Similarly to traditional regression, most system identification approaches to ODE inference involve the optimization of parameter values to match observed data while assuming the structure of the model is fixed in advance. These structures are typically polynomials or Taylor series (Cremers and Hübler 1987, Gouesbet 1991). More recently, system identification approaches often do not assume a fixed structure and instead use structure selection techniques, although these are restricted to combinations of generic basis functions or do not involve ODEs (Aguirre et al. 2001, Hong et al. 2008, Wei and Billings 2008).

Work in the area of system identification frequently assumes that all dynamic variables are measured, even if only in noise-corrupted form, but a significant body of work has also addressed dynamical systems with hidden variables. These techniques, often categorized as belonging to the area of nonlinear dynamics and chaos, are generally based on the observation that a complete phase space representation of a system with hidden variables can be reconstructed using only one or a few observed variables (Packard et al. 1980, Takens 1981). The reconstructed phase space is qualitatively similar to that of the true unknown system and previous studies have taken advantage of that fact to construct approximate models of systems with hidden variables using ODEs (Breden and Hübler 1990) as well as other model formulations (Bakker et al. 2000, Crutchfield and McNamara 1987). However, the trajectories of hidden variables obtained with phase space reconstruction techniques are not equivalent to those that would be obtained by measuring them directly in an

experiment. As result, these trajectories are unsuitable as targets for GP in this work, where the goal is to reverse-engineer the exact analytical model of a system. Nonetheless, it is possible that information obtained from the phase space reconstruction of a system could be incorporated into the approach proposed in this chapter. For example, the search for a hidden variable expression could be seeded with the results of symbolic regression performed with the trajectories obtained from phase space reconstruction as targets.

Another significant body of work on ODE inference can be broadly categorized as knowledge-based, in contrast to the type of free-form modeling possible with GP. While both approaches are essentially a search through the space of possible model structures and parameter values, knowledge-based approaches use human-supplied domain knowledge to constrain the search space (Adachi et al. 2006, Bradley et al. 2001, Džeroski and Todorovski 1993, Langley 1981, Langley et al. 1987, Todorovski and Džeroski 1997, Washio et al. 2000). An example of this approach is inductive process modeling, in which differential equation models for continuous time data are constructed using predefined algebraic expressions and other constraints that represent expert domain knowledge (Langley et al. 2002, Bridewell et al. 2008). This and similar knowledge-based approaches have been successfully used to construct accurate and interpretable ODE models, even when hidden variables are present. However, a serious potential drawback of all knowledge-based approaches is that substantial domain knowledge must be supplied in the form of predefined model structures. For example, in a population dynamics modeling task, inductive process modeling as described in Bridewell et al. 2008 exhaustively enumerates all possible

combinations of one of a few different algebraic structures that each correspond to known models of population growth and decline.

Knowledge-based modeling is a valid approach that has achieved notable success in a variety of problem domains, yet it requires significant human involvement in the modeling process to specify domain knowledge and supply constraints. In light of the explosion of data being gathered in all scientific fields, it is of great interest to develop automated free-form modeling approaches that require no human-supplied domain knowledge aside from the specification of a set of algebraic primitives such as $+$, $-$, \sin , \cos , etc. Both the structure and parameters of the resulting model are constructed de novo from the primitives. Free-form approaches, including the GP-based approach used in this chapter, rely on computationally-intensive stochastic searches through model space, yet recent studies have successfully induced complex ODE models and natural laws without reliance on domain knowledge (Bongard and Lipson 2007, Bongard and Lipson 2009).

GP and other free-form approaches have not been applied to the modeling of systems with hidden variables in any prior work. The motivation of the work presented here is to take the first steps toward combining GP's freedom from supplied domain knowledge with the capability to model ODEs with hidden variables. This combination could extend the power of machine learning to assist scientists in the task of translating raw data into meaningful analytical models.

Methods

Problem Statement

In the general case with no hidden variables, this chapter considers dynamical systems empirically described by n time series $y_1(t), y_2(t), \dots, y_n(t)$, where each time series consists of m observations. The observed data Y from such a system can be written as an $n \times m$ matrix:

$$Y = \begin{pmatrix} y_1(t_0), y_1(t_1), \dots, y_1(t_{m-1}) \\ y_2(t_0), y_2(t_1), \dots, y_2(t_{m-1}) \\ \vdots \\ y_n(t_0), y_n(t_1), \dots, y_n(t_{m-1}) \end{pmatrix} \quad (4.2)$$

where $y_i(t_j)$, ($i = 1, 2, \dots, n$), ($j = 0, 1, \dots, m - 1$) is the observed value of variable y_i at time t_j . The dynamical system can be modeled as a system of ODEs X with the general form:

$$X = \begin{cases} \frac{dX_1}{dt} = f_1(t, X_1, X_2, \dots, X_n) \\ \frac{dX_2}{dt} = f_2(t, X_1, X_2, \dots, X_n) \\ \vdots \\ \frac{dX_n}{dt} = f_n(t, X_1, X_2, \dots, X_n) \end{cases} \quad (4.3)$$

where X_i is the i th state variable and f_i is an arbitrary function of the n state variables and time t . Together with initial values for each of the state variables $\hat{y}_1(0), \hat{y}_2(0), \dots, \hat{y}_n(0)$, system of ODEs X can be solved with numerical integration

techniques to yield the predicted time series data. Written as an $n \times m$ matrix as with Y above, the predicted data \hat{Y} is:

$$\hat{Y} = \begin{pmatrix} \hat{y}_1(t_0), \hat{y}_1(t_1), \dots, \hat{y}_1(t_{m-1}) \\ \hat{y}_2(t_0), \hat{y}_2(t_1), \dots, \hat{y}_2(t_{m-1}) \\ \vdots \\ \hat{y}_n(t_0), \hat{y}_n(t_1), \dots, \hat{y}_n(t_{m-1}) \end{pmatrix} \quad (4.4)$$

To measure the accuracy of system of ODEs X as a model of the observed data Y , this work employed the scaled mean absolute error:

$$\text{error (all variables measured)} = \frac{1}{mn} \sum_{i=1}^n \left(\frac{\sum_{j=0}^{m-1} |Y_{ij} - \hat{Y}_{ij}|}{\sigma_i} \right) \quad (4.5)$$

where σ_i is the standard deviation of the i th observed time series. This scaling factor was used so that error values could be meaningfully compared for any studied dynamical system. The present work is concerned with the “inverse problem” of identifying the form and parameters of system of ODEs X so as to minimize this prediction error.

When inferring an ODE model with hidden variables, one or more of the time series $y_i(t)$ are not measured and are unavailable to guide the model inference process. In this case, the inference process is instead guided by the prediction error calculated solely on the basis of the measured variables. Out of the n total variables in the system, a subset of size o variables are measured or observed and $(n - o)$ variables are

unmeasured or hidden. To calculate the error of a system of ODEs X in this case, the scaled mean absolute error was employed as above except that only measured variables are involved in the error calculation:

$$\text{error} = \frac{1}{mO} \sum_{i=1}^o \left(\frac{\sum_{j=0}^{m-1} |Z_{ij} - \hat{Z}_{ij}|}{\sigma_i} \right) \quad (4.6)$$

Here, σ_i is the standard deviation of the i th observed time series and Z is a matrix equal to Y except that rows corresponding to hidden variables are omitted. Similarly, \hat{Z} is a matrix equal to \hat{Y} except that rows corresponding to hidden variables are omitted. Note that numerical integration of candidate ODE model X with one or more hidden variables always produces n predicted time series; the predicted time series corresponding to hidden variables are simply not used to calculate the error of X .

In some experiments below, noise was added to the observed time series. For each predicted data point, a random number was drawn from a Gaussian distribution with mean 0 and standard deviation equal to 0.01 times the standard deviation of the observed time series. This noise was also added to the initial values of the observed time series that were used as the starting points for numerical integration of candidate ODE models, thereby making evaluation of models considerably more difficult than in a standard symbolic regression task (Bongard and Lipson 2007, Koza 1992). Although sophisticated techniques such as unscented Kalman filtering exist for filtering out this kind of noise (Julier and Uhlmann 1997, Julier et al. 2000), the intent was to test the robustness of model inference in the face of reasonable amounts of

measurement noise. Whenever several GP trials were performed in experiments, noise was added independently for each trial.

GP Algorithm for ODE Model Inference

This work employed a variation on Koza’s tree-based GP to identify both the form and parameters of ODEs (Koza 1992). Individuals were encoded as a forest of n binary trees, where n was the number of equations and dynamic variables in the system of ODEs. For example, the forest of two trees shown in Figure 4.1 corresponds to the system of ODEs:

$$\begin{cases} \frac{dX_1}{dt} = X_1^2 + \sin(X_2) \\ \frac{dX_2}{dt} = \frac{1.75}{(X_1 - X_2)} \end{cases} \quad (4.7)$$

Here, autonomous ODEs were considered, and each tree is composed of terminals from the set $\{a, X_1, X_2, \dots, X_n\}$, where X_i is the i th state variable in a system of ODEs with n variables and a represents ephemeral random constants in the range $(-9.0, 9.0)$. For all examples except **Physical Pendulum** below, the set of possible arithmetic operations used in the binary trees was $\{+, -, *, /\}$. For experiments in the **Physical Pendulum** section, this set was augmented to include trigonometric functions: $\{+, -, *, /, \sin, \cos\}$. Along with the trees representing individual ODEs, each individual was composed of a set of initial values $\hat{y}_1(0), \hat{y}_2(0), \dots, \hat{y}_n(0)$, one for each of the n equations in the system. For observed variables, these values were simply

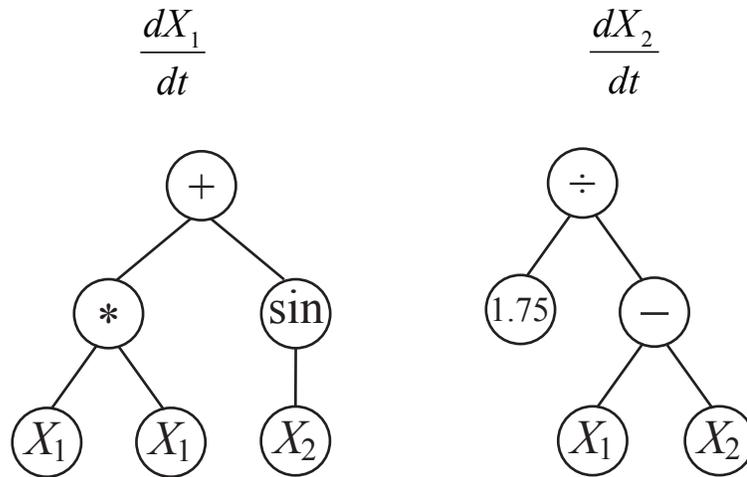


Figure 4.1. The system of ODEs given in Equation 4.7 encoded as a forest of two binary trees.

fixed to the known initial values obtained from the observed time series data, whereas for hidden variables, the values were optimized as if they were trees containing a single ephemeral random constant node. The maximum allowed depth of the trees was five. The population of 96 trees was initialized by randomly adding nodes to the leaves of an initially empty tree until a pre-chosen depth was reached. This depth was chosen randomly from the range (1, 5), with each possible depth equally likely.

As in standard GP, crossover and mutation were used to generate novel individuals. Each generation, two parents from the population of 96 were selected at random. With 90% probability, they generated two offspring by subtree crossover with internal nodes selected as crossover points 90% of the time and terminals selected 10% of the time (Koza 1992). With 5% probability, the parents generated two offspring by each undergoing mutation in which a node in the tree was randomly selected and replaced by a new randomly generated subtree. Finally, with 5% probability, the two parents generated two offspring without modification. No

recombination or mutation to a node of a different type was allowed for the trees in the forest that represented initial values. Reproduction continued into this manner until 96 children were generated, at which point the errors of the offspring trees were evaluated and survival selection took place as described below.

Recent work has shown that using genotypic age as an explicit objective in addition to fitness (or error) can be a powerful yet simple way to reduce bloat and maintain diversity in an evolving GP population (Schmidt and Lipson 2010a-b). Here, age was defined as the number of generations the individual had been present in the population, which was to be minimized. An individual descended from another individual, whether by mutation or recombination, inherited that parent's age. In the case of recombination, the age of the older parent was inherited. To provide a steady flow of low-age genotypes, one new individual with an age of 0 was added to the population each generation. This new individual was randomly constructed in the same way as all 96 members of the initial population.

In contrast to standard GP, genotypic age was used to perform survival selection with the Age-Fitness Pareto algorithm (Schmidt and Lipson 2010a-b). With this algorithm, the minimization of genotypic age was used as a second objective in addition to the minimization of error, making this a multiobjective optimization problem. Survival selection with elitism was implemented in this multiobjective setting with a Pareto tournament of size 2, in which individuals from the combined population of parents and offspring were randomly selected to participate. If either of the pair had both higher error and higher age than the other, it was discarded. This continued until the desired population size of 96 was reached, at which point

reproduction took place. In the event that a large number of tournaments failed to discard any individuals, individuals were to be randomly discarded until the desired population size was reached, although this never occurred in practice.

Many authors have noted that standard GP can have difficulty evolving floating point constants effectively. The solutions that have been proposed generally involve embedding local search algorithms or other optimization techniques within a standard GP algorithm (Iba et al. 1994). Here, the simple strategy of hill-climbing as a local search technique was employed (Russell and Norvig 2009). Following Schmidt and Lipson 2009, each generation, one hill-climbing step was performed in which each constant in each individual was perturbed with a small amount of 0 mean Gaussian noise. The standard deviation of the noise was set to 0.1 times the absolute value of the constant. The error of each individual was then reevaluated and if error increased, the old parameter values were restored.

As in all machine learning, the search for symbolic models with GP involves an inherent tradeoff between model complexity and accuracy (Bishop 2006, Abu-Mostafa 2012). The most accurate models, or those with lowest error on the training data set, tend to be unnecessarily complex and overfit the data in the sense that they perform poorly on test data not seen during training (Montana 1995, Sakamoto and Iba 2001). For this reason, tracking summary statistics such as the training error of the best model over the course of the evolutionary search can be misleading. This is especially the case in the present study as the interest is primarily in recovering the exact structure of synthetic systems and in inferring parsimonious, explanatory models of physical systems. In light of this goal and instead of recording only the model with

lowest error on the training or test data, a Pareto front of models was tracked throughout an evolutionary run. This Pareto front represented the best-so-far tradeoffs between model complexity and training error. After the completion of a trial, the models in this Pareto hall of fame were taken as a whole to be the results of model evolution. Model complexity was measured simply as the sum of the number of terminals and operations used in all equations of the system of ODEs. For example, the model in Equation 4.7 and depicted in Figure 4.1 has a complexity of 11, with 6 terminals and operations in the first equation of the system and 5 terminals and operations in the second equation.

Fitness Evaluation

Instead of maximizing the fitness of evolved models, the goal of evolution in the GP algorithm employed here was to minimize the error as defined in **Problem Statement** above. For the error calculation, the predicted time series were found by numerically integrating the system of ODEs encoded by an individual using the common fourth-order Runge-Kutta method with fixed step size (Press et al. 2007). In most of the examples described in detail below, three different variations on this basic means of evaluating candidate models were compared. In the first method, each candidate model was integrated for 128 time steps and all 128 resulting predicted data points were compared against the 128 corresponding observed data points using Equation 4.6. This method is referred to as “128 All” below. In the second method, each candidate model was integrated for only 8 time steps and the 8 resulting predicted data points were compared against the 8 corresponding observed data points. This method

is referred to as “8 All” below.

Although the evaluations of candidate models using only 8 data points are likely to be much less accurate than those using all 128 data points, each of the individual evaluations using 8 points required only 0.0625 times as much computational effort as when all 128 points were calculated. As computational effort in this application of GP is dominated by the evaluation of candidate models, 16 times as many individuals can be evaluated for a given amount of computational effort with the 8-point method. A priori it was not clear which method would perform better given this evaluation accuracy vs. number of evaluations tradeoff. For purposes of fair comparison between methods, the total computational effort as measured by number of integration steps performed was held constant within each example shown below.

In the third method of evaluating candidate models, each candidate model was numerically integrated for 128 time steps as before, but only 8 of the resulting predicted time series data points were compared against the corresponding observed time series data points. This method is referred to as “8 Coev.” below. The selection of these 8 points from out of the 128 total generated data points was performed using the Coevolution of Fitness Predictors algorithm detailed in Schmidt and Lipson 2008. That study found that only a small number (approximately 8) of the data points from a complete training data set are necessary to accurately evaluate individual fitnesses on a symbolic regression task. Although those points can be selected randomly, coevolving the location of points based on the accuracy of the resulting fitness estimates was found to accelerate evolution the most. Although this main result does not apply here because the number of evaluated data points is defined by the number

of integration steps taken, the authors also found that even when the advantage of decreased computational effort is not taken into account, coevolving the location of evaluation points accelerated evolution relative to other methods. For this work, it was hypothesized that using only 8 out of the 128 obtained data points but selecting them with coevolution might confer a similar advantage. Note that there is no advantage in terms of computational effort of this coevolutionary method over simply using all 128 points to evaluate a candidate model, as in both cases an individual candidate model must be integrated for 128 time steps.

Depending on the method, either 128 integration points or 8 integration points were used as training data to calculate the error used to guide evolution of the models. However, in the results below, error is instead reported on test data not used to guide evolution in order to better evaluate how well the evolved models could make predictions. Regardless of the method used to train the models, this test error was obtained by integrating the evolved model for 256 time steps and comparing the last 128 points to the corresponding points in the target model using the error calculation in Equation 4.6.

In the examples described below in sections **Initial Experiments**, **Ideal Rocket Equations**, **Chemical Reaction Network**, and **Random Systems of ODEs**, the number of point evaluations and integration steps were as described above. However, due to the nature of the empirical data, in the section **Physical 5-azacytidine Degradation Kinetics**, 256 instead of 128 integration steps were used during training and 320 instead of 256 integration steps were used for computing test error. The 8 Coev. and 8 All methods still employed only 8 points for these

experiments. In the section **Physical Pendulum**, 128 integration steps were used during training, but only 192 integration steps were used for computing test error. Again, the 8 Coev. and 8 All methods employed only 8 points. The differences between sections **Initial Experiments**, **Ideal Rocket Equations**, and **Chemical Reaction Network** as compared with sections **Physical 5-azacytidine Degradation Kinetics** and **Physical Pendulum** are reflected in the figures illustrating the data sets. For consistency the three evaluation methods are referred to as “128 All,” “8 Coev.,” and “8 All” for all examples below.

Results

The performance of the GP algorithm was studied for several systems with hidden variables. The six main systems studied are shown in Figure 4.2, which summarizes the target system, the behavior of the system, and some results obtained with the algorithm.

Initial Experiments

For initial experiments, two simple synthetic nonlinear dynamical systems with known ODE models were used. The first was given by:

Figure 4.2. Summary of dynamical systems studied (next page). For the two physical systems (5-azacytidine degradation kinetics and pendulum), the target model is the approximate theoretical model derivable from first principles.

Target System

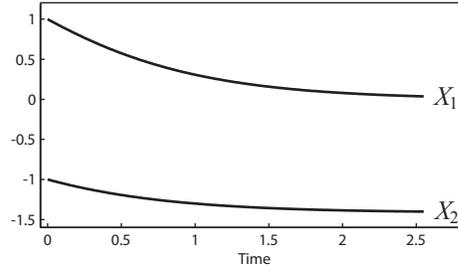
Experimental Data

Example Inferred Model

synthetic system 1

$$\frac{dX_1}{dt} = X_1 X_2$$

$$\frac{dX_2}{dt} = -0.5 X_1$$



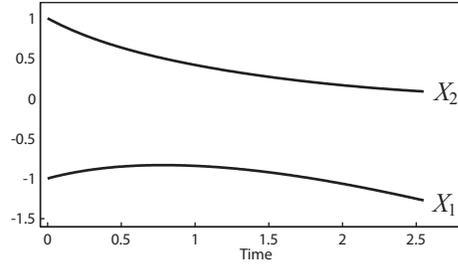
$$\frac{dX_1}{dt} = X_1 X_2$$

$$\frac{dX_2}{dt} = -0.5000 X_1$$

synthetic system 2

$$\frac{dX_1}{dt} = X_2 - 0.5$$

$$\frac{dX_2}{dt} = X_1 X_2$$



$$\frac{dX_1}{dt} = X_2 - 0.5002$$

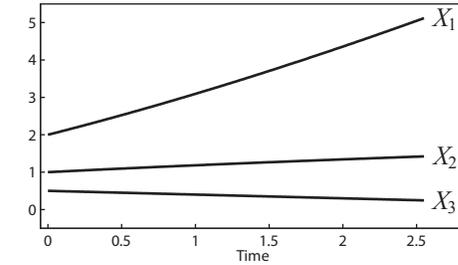
$$\frac{dX_2}{dt} = X_1 X_2$$

ideal rocket equations

$$\frac{dX_1}{dt} = X_2$$

$$\frac{dX_2}{dt} = \frac{0.2}{X_3}$$

$$\frac{dX_3}{dt} = -0.1$$



$$\frac{dX_1}{dt} = X_2$$

$$\frac{dX_2}{dt} = \frac{0.2060}{X_3}$$

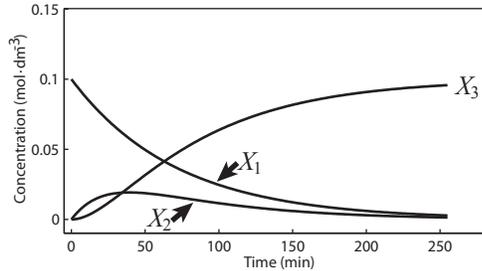
$$\frac{dX_3}{dt} = -0.1030$$

chemical reaction network

$$\frac{dX_1}{dt} = -1.4 X_1$$

$$\frac{dX_2}{dt} = 1.4 X_1 - 4.2 X_2$$

$$\frac{dX_3}{dt} = 4.2 X_2$$



$$\frac{dX_1}{dt} = -1.3906 X_1$$

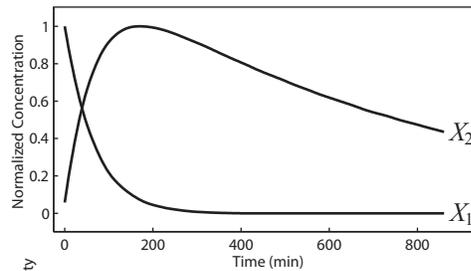
$$\frac{dX_2}{dt} = X_1 - 4.2451 X_2$$

$$\frac{dX_3}{dt} = 4.1547 X_2 + X_2 X_3$$

physical 5-azacytidine degradation kinetics

$$\frac{dX_1}{dt} = -0.02 X_1$$

$$\frac{dX_2}{dt} = 0.02 X_1 - 0.001 X_2$$



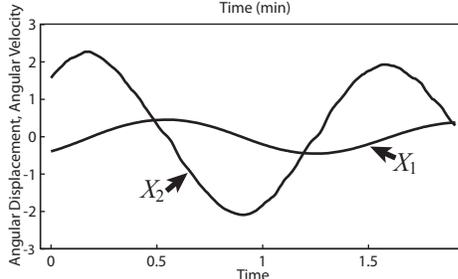
$$\frac{dX_1}{dt} = -0.0013 X_1$$

$$\frac{dX_2}{dt} = 0.0236 X_1 - 0.0150 X_2$$

physical pendulum

$$\frac{dX_1}{dt} = X_2$$

$$\frac{dX_2}{dt} = -20.0 \sin(X_1) - 0.2 X_2$$



$$\frac{dX_1}{dt} = X_2$$

$$\frac{dX_2}{dt} = -19.8412 X_1 - 0.2070 X_2$$

$$\begin{cases} \frac{dX_1}{dt} = X_1 X_2 \\ \frac{dX_2}{dt} = -0.5 X_1 \end{cases} \quad (4.8)$$

with initial values $y_1(0)=1.0, y_2(0)=-1.0$. Data were generated from this target model, both with and without noise (Figure 4.3). X_2 was designated as a hidden variable and the data available to the GP algorithm consisted only of the time series data for X_1 . An attempt was then made to reverse-engineer the exact model given in Equation 4.8 using only this time series data for X_1 . This was repeated for 50 independent trials using each of the three fitness evaluation methods described in **Fitness Evaluation** above. Figure 4.4 summarizes the results of these runs and shows the test error of the best model found as a function of computational effort for each of the three methods.

For both noise-free data (Figure 4.4a) and data to which noise was added (Figure 4.4b), evaluating a candidate model by integrating for 128 time steps and then using all 128 resulting points (128 All) resulted in models that on average generalized better to unseen test data. Less successful were the methods of integrating for 128 steps but using only 8 coevolved points (8 Coev.) and of integrating for only 8 steps (8 All).

Examining the evolved ODE models revealed that in many of the 50 trials, the exact form of the target model was found:

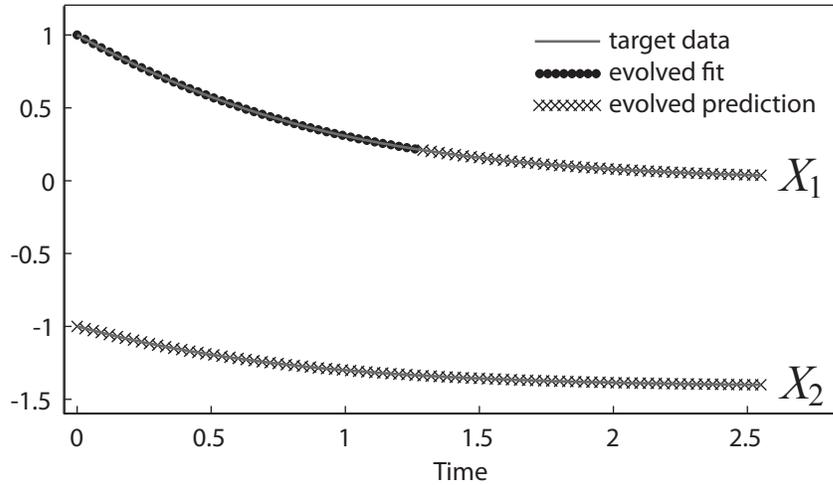


Figure 4.3. Behavior of the first target system and an evolved model from the Initial Experiments section. Equation 4.8 was used to generate the target data shown in the solid gray line. The behavior of an evolved ODE model in the region covered by the training data (“evolved fit”) is shown in solid circles and the behavior of that evolved model in a region not seen during training (“evolved prediction”) is shown with x’s. The behavior of the hidden variable (X_2) in the evolved model is plotted entirely with x’s as none of the time series data from X_2 were available for training.

$$\begin{cases} \frac{dX_1}{dt} = X_1 X_2 \\ \frac{dX_2}{dt} = \alpha X_1 \end{cases} \quad (4.9)$$

with initial values $\hat{y}_1(0) = 1.0, \hat{y}_2(0) = \beta$, where α and β are floating point constants.

Table 4.1 shows the frequency with which this form was identified and the variability of the two constants across trials in which this form was identified. Figure 4.3 compares the response of a typical evolved model having the target form with the response of the target model, showing a very close match on both the observed time series for X_1 and the hidden time series for X_2 .

Looking more closely at the trials in which the exact form of the target model was not identified, it was frequently observed that equally accurate and parsimonious

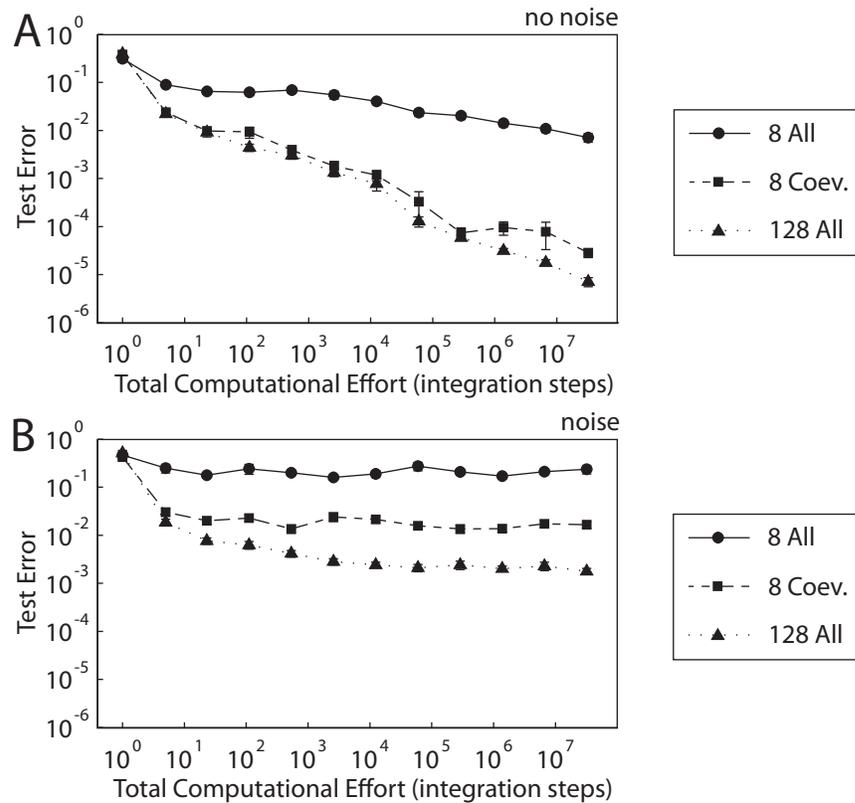


Figure 4.4. Summary of evolutionary runs for the first example in the Initial Experiments section. The error of the best model on test data not seen during training was calculated each generation. In selected generations, the mean of this test error was plotted for each of the three evaluation methods with symbols as shown in the legend. Error bars represent \pm the standard error of the mean. In most cases the error bars are smaller than the height of the markers and are not visible. Panel **A** shows results for noise-free data and panel **B** shows results for data to which noise was added.

models with a different structure were identified. For example, in 38 out of 50 trials without noise and using the 128 All evaluation method, the exact target model form was identified (Table 4.1). In all 12 out of the remaining 12 trials, accurate models with this alternative structure were identified:

Table 4.1. Evolved models for the first system in the Initial Experiments section. A successful trial indicates that the form given in Equation 4.9 was evolved in the trial. α and β refer to the floating point constants in Equation 4.9. The variability of the constants across trials in which the target form in Equation 4.9 was evolved is given as mean \pm standard deviation. No data for a given measurement is indicated by “NA.”

Evaluation Method, Noise	Num. Successful Trials Out of 50	α	β
128 All, no noise	38	-0.5000 ± 0.0003	-1.0000 ± 0.0000
128 All, noise	21	-0.5160 ± 0.0226	-0.9928 ± 0.0092
8 Coev., no noise	38	-0.5000 ± 0.0002	-1.0000 ± 0.0000
8 Coev., noise	13	-0.5239 ± 0.0294	-0.9904 ± 0.0113
8 All, no noise	8	-0.5006 ± 0.0004	-1.0000 ± 0.0000
8 All, noise	0	NA	NA

$$\begin{cases} \frac{dX_1}{dt} = \alpha X_1 X_2 \\ \frac{dX_2}{dt} = X_1 \end{cases} \quad (4.10)$$

with initial values $\hat{y}_1(0) = 1.0, \hat{y}_2(0) = \beta$, where α and β are floating point constants. In all cases, α was approximately -0.5 and β was approximately 2.0 . Both these models and the target model in Equation 4.8 have a complexity of 6, and in most cases the training error for models with this structure was approximately the same as the training error for models with the target structure. In other words, models with the structure shown in Equation 4.10 appear to represent a local optimum in the space of possible models that was very attractive to the evolutionary search, thereby making the primary goal of recovering the exact target structure difficult.

One approach to recovering the exact target structure despite the presence of local optima was motivated by the observation that the local optimum represented by models with the structure in Equation 4.10 appeared to be associated with initial

values $\hat{y}_1(0) = 1.0, \hat{y}_2(0) = 2.0$, whereas evolved models matching the true target structure tended to have initial values very close to the true values of $\hat{y}_1(0) = 1.0, \hat{y}_2(0) = -1.0$. As the initial value of the hidden variable ($\hat{y}_2(0)$) was treated as a free parameter to be optimized, it is possible to simply constrain this initial value to a range such as $(-1.1, -0.9)$, which may prevent the evolutionary search from becoming trapped in the local optimum associated with initial values of approximately 2.0. A priori, there would be no way to know that the initial values should be constrained in this way. However, assuming that the empirical time series data can be normalized to approximately order unity and at the expense of additional computational effort, several trials could be used to “scan” through the space of possible initial values.

Such scans were performed with 50 trials each in which the initial values were constrained to a range of 0.2 and in which the center of the ranges was successively set to each of $(-6.0, -5.8, \dots, 3.6, 3.8)$. When the range of initial values was set to $(-1.1, -0.9)$, models of the form shown in Equation 4.8 were found with parameters closely matching those of the target. An example of one such model is:

$$\begin{cases} \frac{dX_1}{dt} = X_1 X_2 \\ \frac{dX_2}{dt} = -0.5000 X_1 \end{cases} \quad (4.11)$$

with initial values $y_1(0) = 1.0, y_2(0) = -1.0000$. This initial value scanning approach successfully recovered the target model when the 128 All and 8 Coev. methods were used, from both noise-free data and from data with added noise. Although the

alternative model in Equation 4.10 was also found in some trials of the scans, this approach is one potential means of reliably recovering the exact target model structure despite the presence of highly attractive local optima in model space.

In these initial experiments, this target system was also employed:

$$\begin{cases} \frac{dX_1}{dt} = X_2 - 0.5 \\ \frac{dX_2}{dt} = X_1 X_2 \end{cases} \quad (4.12)$$

with initial values $y_1(0) = -1.0, y_2(0) = 1.0$. Again X_2 was designated as a hidden variable and data were generated both with and without noise (Figure 4.5). This system is very similar to that in Equation 4.8 except that now the hidden variable is described by a nonlinear expression whereas in Equation 4.8, the hidden variable was described by a linear expression. Figure 4.6 summarizes the results of 50 independent trials using only the time series data from X_1 in Equation 4.12 as the basis for model identification. The results show that as before, evaluating a candidate model by integrating for 128 time steps and then using all 128 resulting points (128 All) was the most successful evaluation method.

The target form of this second system in general is:

$$\begin{cases} \frac{dX_1}{dt} = X_2 - \alpha \\ \frac{dX_2}{dt} = X_1 X_2 \end{cases} \quad (4.13)$$

with initial values $\hat{y}_1(0) = -1.0, \hat{y}_2(0) = \beta$, where α and β are floating point constants

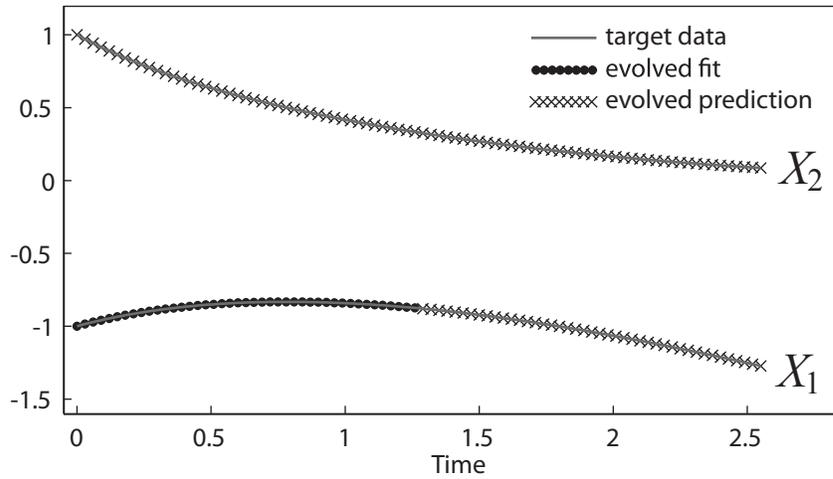


Figure 4.5. Behavior of the second target system and an evolved model from the Initial Experiments section. Equation 4.12 was used to generate the target data shown in the solid gray line.

Table 4.2 shows the frequency with which this form was identified and the variability of the two constants across trials in which this form was identified. Figure 4.5 compares the response of a typical evolved model matching the target form with that of the target model.

As with the previous test case, alternative model structures were found that were difficult for the evolutionary search to reject on the basis of accuracy or parsimony. One frequently encountered alternative structure was:

$$\begin{cases} \frac{dX_1}{dt} = X_1 \\ \frac{dX_2}{dt} = X_1(X_2 + \alpha) \end{cases} \quad (4.14)$$

with initial values $\hat{y}_1(0) = -1.0$, $\hat{y}_2(0) = \beta$, where α and β are floating point constants.

In all cases, α and β were approximately 0.5. Models with this structure have complexity equal to the target model in Equation 4.12, and depending on the exact

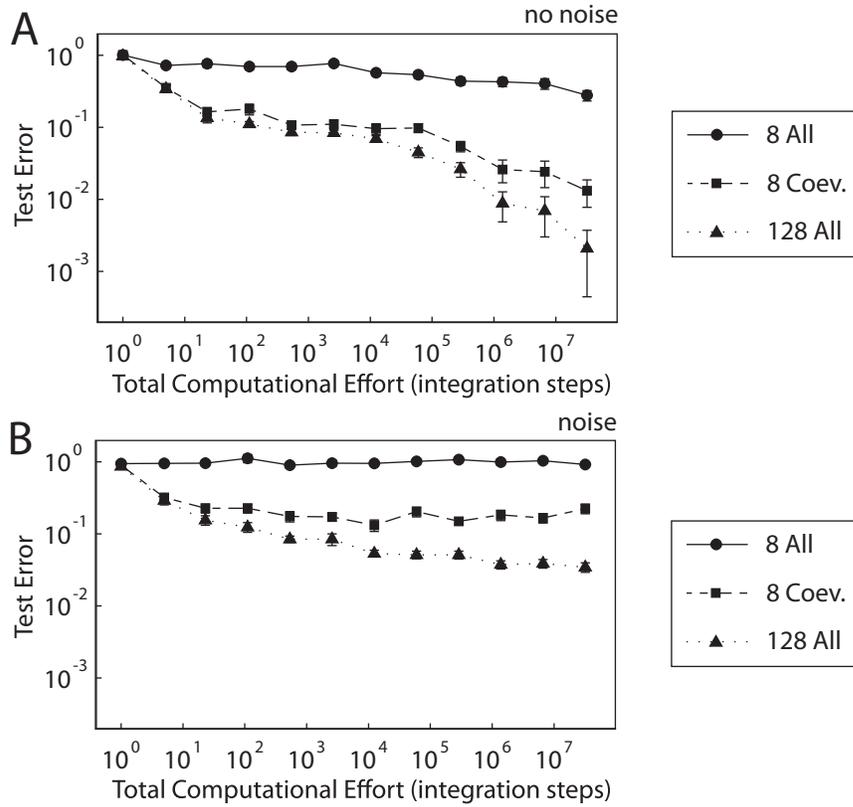


Figure 4.6. Summary of evolutionary runs for the second example in the Initial Experiments section. Panel **A** shows results for noise-free data and panel **B** shows results for data with noise.

values of the constants present, often had similar or even lower training errors than models with the target structure. To overcome this attractive local optimum in the space of possible models, the initial value scanning approach described above was used again. When the range of initial values was set to (0.9, 1.1), models of the form shown in Equation 4.12 were found with parameters closely matching those of the target. An example of one such model is:

$$\begin{cases} \frac{dX_1}{dt} = X_2 - 0.5002 \\ \frac{dX_2}{dt} = X_1 X_2 \end{cases} \quad (4.15)$$

Table 4.2. Evolved models for the second system in the Initial Experiments section. A successful trial indicates that the form given in Equation 4.13 was evolved in the trial.

Evaluation Method, Noise	Num. Successful Trials Out of 50	α	β
128 All, no noise	25	0.5000 ± 0.0004	1.0000 ± 0.0005
128 All, noise	11	0.5036 ± 0.0101	1.0017 ± 0.0043
8 Coev., no noise	24	0.4999 ± 0.0006	0.9998 ± 0.0008
8 Coev., noise	11	0.5016 ± 0.0029	1.0031 ± 0.0057
8 All, no noise	1	0.5001 ± 0.0000	1.0001 ± 0.0000
8 All, noise	0	NA	NA

with initial values $y_1(0) = -1.0, y_2(0) = 1.0003$. This initial value scanning approach worked successfully for the 128 All and 8 Coev. methods both with and without noise.

Ideal Rocket Equations

In this section, an instance of the ideal rocket equations given in Equation 4.1 is considered:

$$\begin{cases} \frac{dX_1}{dt} = X_2 \\ \frac{dX_2}{dt} = \frac{0.2}{X_3} \\ \frac{dX_3}{dt} = -0.1 \end{cases} \quad (4.16)$$

with initial values $y_1(0) = 2.0, y_2(0) = 1.0, y_3(0) = 0.5$. X_3 corresponds to the mass variable in Equation 4.1 and so it was taken as a hidden variable, whereas position (X_1) and velocity (X_2) were both observed variables. Target data from this system were simulated both with and without measurement noise as shown in Figure 4.7. Figure 4.8 summarizes the results of 50 independent trials using the time series data from X_1

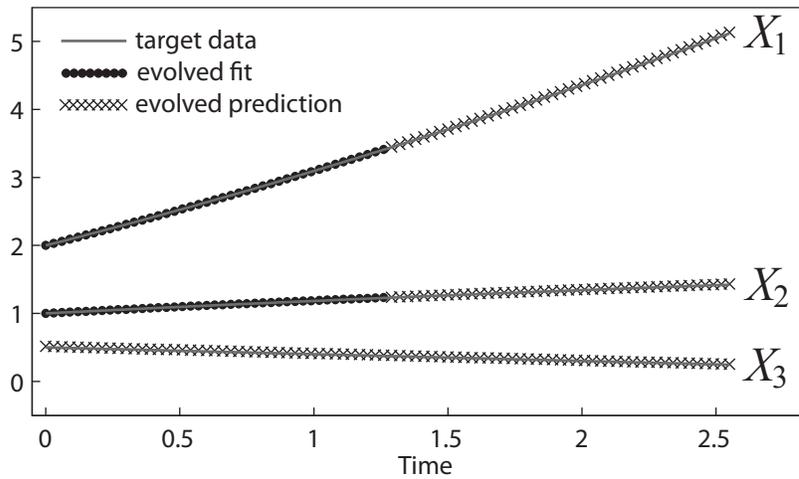


Figure 4.7. Behavior of the target system and an evolved model from the Ideal Rocket Equations section. Equation 4.16 was used to generate the target data shown in the solid gray line.

and X_2 in Equation 4.16 as the basis for model identification.

The target form of the ODE models for this example in general is:

$$\left\{ \begin{array}{l} \frac{dX_1}{dt} = X_1 \\ \frac{dX_2}{dt} = \frac{\alpha}{X_3} \\ \frac{dX_3}{dt} = \beta \end{array} \right. \quad (4.17)$$

with initial values $y_1(0) = 2.0, y_2(0) = 1.0, y_3(0) = \chi$, where α, β, χ are floating point constants. Table 4.3 shows the frequency with which this form was identified and the variability of the floating point constants across trials in which this form was identified. Unlike with the examples in the **Initial Experiments** section above, the true values of the floating point constants were difficult for the GP algorithm to identify. This was the case even when the correct form of the model was identified.

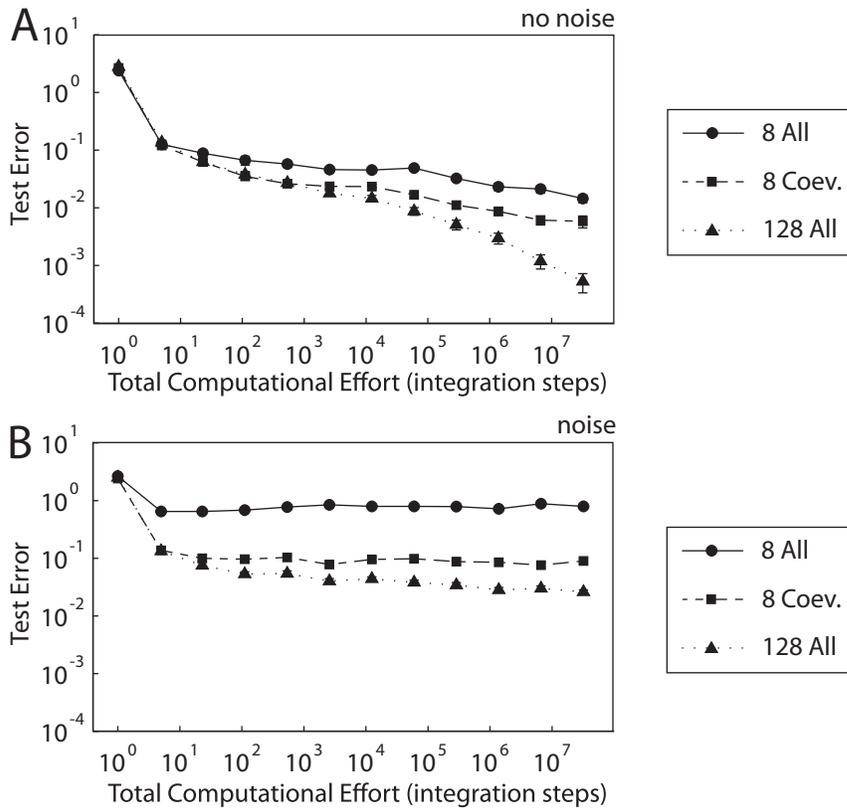


Figure 4.8. Summary of evolutionary runs for the Ideal Rocket Equations Section. Panel **A** shows results for noise-free data and panel **B** shows results for data with noise.

Although this difficulty in identifying parameter values appears to be due to a different type of local optimum in model space than that encountered in the examples in the **Initial Experiments** section, the initial value scanning approach used there is again appropriate for attempting to recover the target model. When the range of initial values for the hidden variable was set to $(0.4, 0.6)$, models of the form shown in Equation 4.16 were found with parameters closely matching those of the target. This initial value scanning approach worked successfully for the 128 All and 8 Coev. methods without noise and the 128 All method with noise added. Figure 4.7 compares the response of the target model to the response of this typical evolved model obtained

Table 4.3. Evolved models for the Ideal Rocket Equations section. A successful trial indicates that the form given in Equation 4.17 was evolved in the trial.

Evaluation Method, Noise	Num. Successful Trials Out of 50	α	β	χ
128 All, no noise	42	0.7136 ± 2.5092	-0.3565 ± 1.2542	0.0011 ± 0.0014
128 All, noise	0	NA	NA	NA
8 Coev., no noise	40	-0.3243 ± 2.9437	0.1881 ± 1.4694	0.0013 ± 0.0012
8 Coev., noise	5	1.1016 ± 2.3599	-0.6617 ± 1.1341	0.0299 ± 0.0287
8 All, no noise	8	0.5073 ± 3.1272	-0.2541 ± 1.5642	0.0005 ± 0.0005
8 All, noise	0	NA	NA	NA

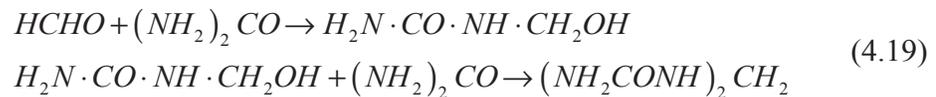
with initial value scanning:

$$\begin{cases} \frac{dX_1}{dt} = X_2 \\ \frac{dX_2}{dt} = \frac{0.2060}{X_3} \\ \frac{dX_3}{dt} = -0.1030 \end{cases} \quad (4.18)$$

with initial values $y_1(0) = 2.0, y_2(0) = 1.0, y_3(0) = 0.5151$.

Chemical Reaction Network

Several previous studies that evolved ODE models have used a particular chemical reaction problem as a benchmark (Ahalpara and Sen 2011, Ando et al. 2002, Bernardino and Barbosa 2011, Cao et al. 2000). This problem involves the reaction between formaldehyde and carbamide, which produces methylol urea and methylene urea according to:



The concentrations of formaldehyde (X_1), methylol urea (X_2), and methylene urea (X_3) change over time in a way that satisfies the ODE:

$$\begin{cases} \frac{dX_1}{dt} = -1.4X_1 \\ \frac{dX_2}{dt} = 1.4X_1 - 4.2X_2 \\ \frac{dX_3}{dt} = 4.2X_2 \end{cases} \quad (4.20)$$

with initial values $y_1(0) = 0.1, y_2(0) = 0.0, y_3(0) = 0.0$. The reaction rate parameters and initial values are typical observed experimental values given in Cao et al. 2000. Target data from this system were simulated both with and without measurement noise (Figure 4.9).

For the present work, the problem of interest is to model the complete reaction network when data for one or more of the three concentration time series are not available. Due to the nature of the dependencies in the reaction network, the presence of hidden variables can drastically change the modeling problem. For example, if only the concentration of formaldehyde (X_1) is measured, there is no way to infer expressions for the rate of change of the concentration of methylol urea (X_2) or methylene urea (X_3). This is because the rate of change of the concentration of formaldehyde is effectively independent of the concentrations of all species other than formaldehyde itself, as can be seen in Equation 4.20. Aside from measuring all three concentrations, there are six possible combinations of measurements that can be taken: $\{X_1\}, \{X_2\}, \{X_3\}, \{X_1, X_2\}, \{X_1, X_3\}, \{X_2, X_3\}$. Each of these cases will be considered in turn and will be referred to using only the relevant variable names for brevity. Due

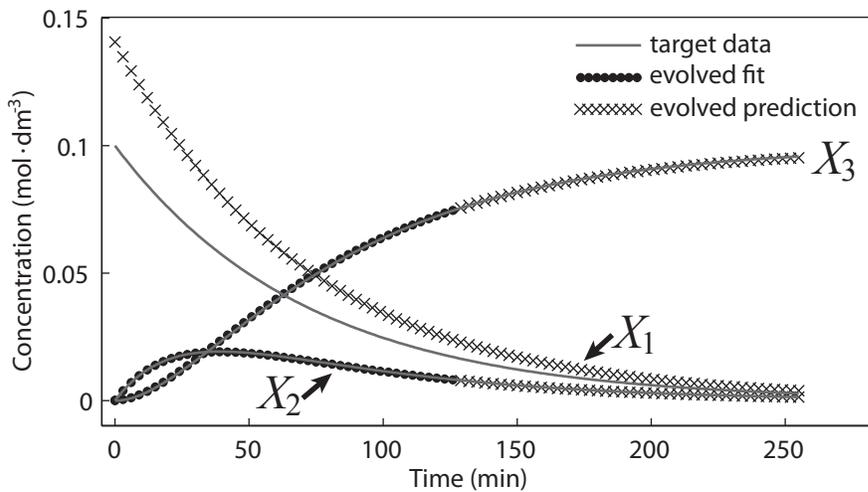


Figure 4.9. Behavior of the target system and an evolved model from the Chemical Reaction Network section. Equation 4.20 was used to generate the target data shown in the solid gray line.

to the large number of cases, for these experiments only the 128 All evaluation method was used, as previous results suggested it was the most effective method.

When only $\{X_1\}$ is measured, the expression for the rate of change of X_1 is constrained in the ODE model for the system, whereas no information exists as the basis for finding an expression for the rate of change of X_2 or X_3 . Figure 4.10 summarizes the results of 50 independent trials using the time series data $\{X_1\}$ as the basis for model identification. The target form of the ODE models for this problem in general is:

$$\begin{cases} \frac{dX_1}{dt} = \alpha X_1 \\ \frac{dX_2}{dt} = \sim \\ \frac{dX_3}{dt} = \sim \end{cases} \quad (4.21)$$

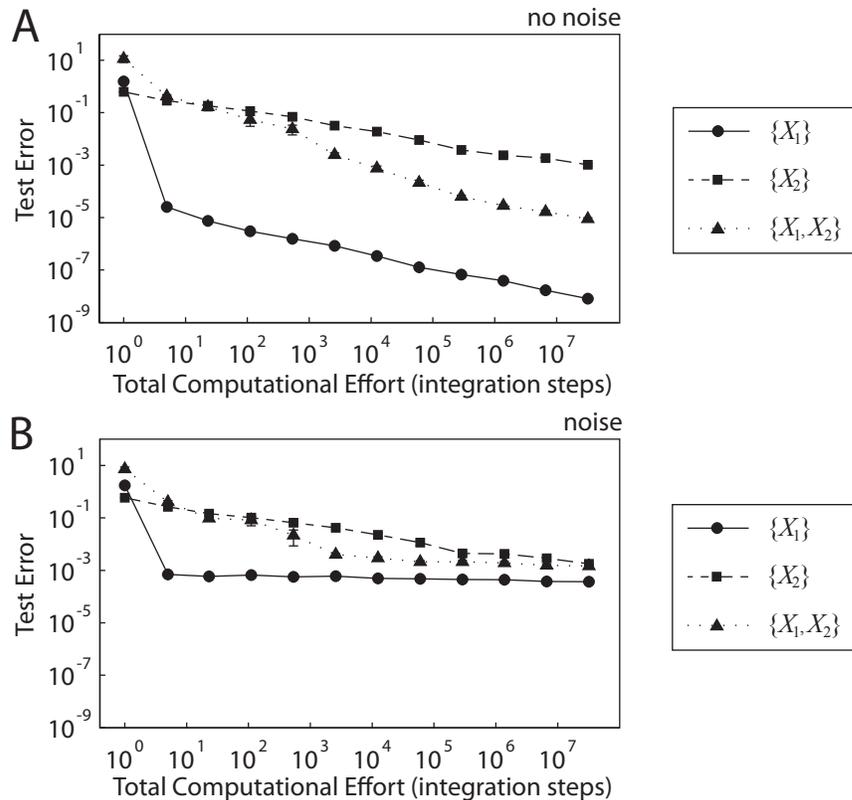


Figure 4.10. Summary of evolutionary runs for the Chemical reaction network section with measured variables $\{X_1\}$, $\{X_2\}$, $\{X_1, X_2\}$. All runs used the 128 All evaluation method. Panel **A** shows results for noise-free data and panel **B** shows results for data with noise.

with initial values $y_1(0) = 0.1, y_2(0) = \sim, y_3(0) = \sim$, where α is a floating point constant and \sim is arbitrary. Table 4.4 shows the frequency with which this form was identified and the variability of the floating point constant across trials in which this form was identified.

When the measurements consist of $\{X_2\}$ or $\{X_1, X_2\}$, the expressions for the rate of change of X_1 and X_2 are constrained in the ODE model for the system, whereas no information exists as the basis for finding an expression for the rate of change of X_3 . Figure 4.10 summarizes the results of 50 independent trials using the time series data $\{X_2\}$ or $\{X_1, X_2\}$ as the basis for model identification. The target form of the

Table 4.4. Evolved models for the Chemical Reaction Network section. For trials with $\{X_1\}$ measured, a successful trial indicates that the form given in Equation 4.21 was evolved. For trials with $\{X_2\}$ or $\{X_1, X_2\}$ measured, a successful trial indicates that the form given in Equation 4.22 was evolved. For the remaining trials, a successful trial indicates that the form given in Equation 4.23 was evolved.

{Measured Variables}, Noise	Num. Successful Trials Out of 50	α	β	χ	δ	ε	φ
$\{X_1\}$, no noise	50	-1.4000 \pm 0.0000	NA	NA	NA	NA	NA
$\{X_1\}$, noise	50	-1.4000 \pm 0.0048	NA	NA	NA	NA	NA
$\{X_1, X_2\}$, no noise	50	-1.4000 \pm 0.0000	1.4000 \pm 0.0000	4.1999 \pm 0.0001	NA	NA	NA
$\{X_1, X_2\}$, noise	49	-1.4006 \pm 0.0048	1.3992 \pm 0.0056	4.1974 \pm 0.0161	NA	NA	NA
$\{X_2\}$, no noise	25	-3.1863 \pm 1.3691	1.0880 \pm 0.6903	2.4107 \pm 0.3729	NA	0.1715 \pm 0.1203	NA
$\{X_2\}$, noise	18	-2.6312 \pm 1.3873	1.0301 \pm 0.9361	2.6135 \pm 1.4852	NA	0.5836 \pm 1.0006	NA
$\{X_2, X_3\}$, no noise	3	-1.3882 \pm 0.0129	0.8377 \pm 0.2811	4.2209 \pm 0.0389	4.1974 \pm 0.0506	0.1840 \pm 0.0757	NA
$\{X_2, X_3\}$, noise	6	-2.7514 \pm 1.5170	1.2043 \pm 1.8326	2.8550 \pm 1.5655	4.1916 \pm 0.0543	0.1220 \pm 0.0437	NA
$\{X_3\}$, no noise	1	-2.4779 \pm 0.0000	1.0000 \pm 0.0000	2.0000 \pm 0.0000	0.0503 \pm 0.0000	8.7270 \pm 0.0000	0.2894 \pm 0.0000
$\{X_3\}$, noise	0	NA	NA	NA	NA	NA	NA
$\{X_1, X_3\}$, no noise	1	-1.4000 \pm 0.0000	-4.5143 \pm 0.0000	4.5143 \pm 0.0000	-1.4119 \pm 0.0000	NA	0.0048 \pm 0.0000
$\{X_1, X_3\}$, noise	0	NA	NA	NA	NA	NA	NA

ODE models for this problem in general is:

$$\begin{cases} \frac{dX_1}{dt} = \alpha X_1 \\ \frac{dX_2}{dt} = \beta X_1 - \chi X_2 \\ \frac{dX_3}{dt} = \sim \end{cases} \quad (4.22)$$

with initial values $y_1(0) = \varepsilon, y_2(0) = 0.0, y_3(0) = \sim$, where $\alpha, \beta, \chi, \varepsilon$ are floating point constants and \sim is arbitrary. (In the case when X_1 and X_2 were measured, ε was set to the observed initial value.) Table 4.4 shows the frequency with which this form was identified and the variability of the four constants across trials in which this form was identified.

When the measurements consist of $\{X_3\}$, $\{X_1, X_3\}$, or $\{X_2, X_3\}$, the expressions for the rate of change of all three variables are constrained in the ODE model for the system. Figure 4.11 summarizes the results of 50 independent trials using the time series data $\{X_3\}$, $\{X_1, X_3\}$, or $\{X_2, X_3\}$ as the basis for model identification. The target form of the ODE models for this problem in general is:

$$\begin{cases} \frac{dX_1}{dt} = \alpha X_1 \\ \frac{dX_2}{dt} = \beta X_1 - \chi X_2 \\ \frac{dX_3}{dt} = \delta X_2 \end{cases} \quad (4.23)$$

with initial values $y_1(0) = \varepsilon, y_2(0) = \varphi, y_3(0) = 0.0$, where $\alpha, \beta, \chi, \delta, \varepsilon, \varphi$ are floating

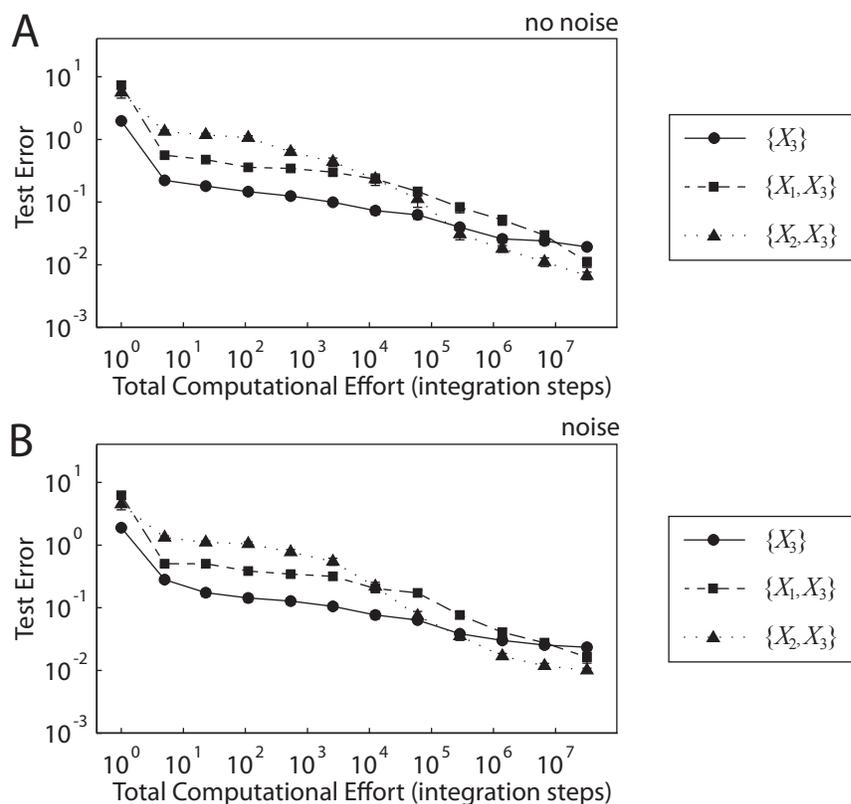


Figure 4.11. Summary of evolutionary runs for the Chemical reaction network section with measured variables $\{X_3\}$, $\{X_1, X_3\}$, $\{X_2, X_3\}$. All runs used the 128 All evaluation method. Panel A shows results for noise-free data and panel B shows results for data with noise.

point constants. (In the case when X_1 and X_3 were measured, ε was set to the observed initial value. When X_2 and X_3 were measured, φ was set to the observed initial value.) Table 4.4 shows the frequency with which this form was identified and the variability of the six constants across trials in which this form was identified.

The results show that identifying the target form and parameters of the model is difficult in all cases except when $\{X_1\}$ or $\{X_1, X_2\}$ are measured. As much of this difficulty could be due to becoming trapped in local optima in model structure and parameter space, the initial value scanning approach was used again. It was successful in many cases. For example, with $\{X_2\}$ measured, the following partial model was

recovered from noisy data using the 128 All method:

$$\begin{cases} \frac{dX_1}{dt} = -1.4020X_1 \\ \frac{dX_2}{dt} = X_1 - 4.1926X_2 \\ \frac{dX_3}{dt} = \sim \end{cases} \quad (4.24)$$

with initial values $y_1(0) = 0.1398, y_2(0) = 0.0, y_3(0) = \sim$, which closely matches Equation 4.22. As another example, with $\{X_2, X_3\}$ measured, the following model was recovered from noisy data using the 128 All method:

$$\begin{cases} \frac{dX_1}{dt} = -1.3906X_1 \\ \frac{dX_2}{dt} = X_1 - 4.2451X_2 \\ \frac{dX_3}{dt} = 4.1547X_2 + X_2X_3 \end{cases} \quad (4.25)$$

with initial values $y_1(0) = 0.1407, y_2(0) = 0.0000, y_3(0) = 0.0002$, which closely matches Equation 4.20 aside from the X_2X_3 term in the third equation. The behavior of Equation 4.25 is plotted in Figure 4.9 and compared with the behavior of the target model given in Equation 4.20.

Physical 5-azacytidine Degradation Kinetics

5-azacytidine is an anti-cancer drug that rapidly degrades in the body when administered by intravenous injection (Beisler 1978, Kissinger and Stemm 1986). The initial product formed as a result of this degradation is *N*-(formylamidino)-*N'*- β -D-

ribofuranosylurea (RGU-CHO). Due to the therapeutic benefits of 5-azacytidine, it is of interest to characterize the kinetics of this degradation reaction and many studies have done so (Argemí and Saurina 2007, Beisler 1978, Kissinger and Stemm 1986). Here, empirical data from the physical 5-azacytidine degradation reaction obtained by Argemí and Saurina are used (Argemí and Saurina 2007). These experimental measurements, shown in Figure 4.12, were the input to the GP algorithm that attempted to infer a model for the data.

It was first assumed that all the data were available, that is, that both the concentrations of 5-azacytidine (X_1) and RGU-CHO (X_2) were measured as in the original experiment (Argemí and Saurina 2007). Using the 128 All evaluation method, models of the following form were readily obtained:

$$\begin{cases} \frac{dX_1}{dt} = \alpha X_1 \\ \frac{dX_2}{dt} = \beta X_1 - \chi X_2 \end{cases} \quad (4.26)$$

with initial values $y_1(0) = 1.0000, y_2(0) = 0.0578$, where α, β, χ are floating point constants. In all cases, α was approximately -0.02, β was approximately 0.02, and χ was approximately 0.001. This explanatory model is consistent with the known reaction mechanism (Beisler 1978) and provided a baseline for comparison with subsequent experiments described next.

It was then assumed that only the concentration of RGU-CHO (X_2) was measured and that the concentration of 5-azacytidine (X_1) was a hidden variable. Figure 4.13 summarizes the results of 50 independent trials using the empirical X_2 data

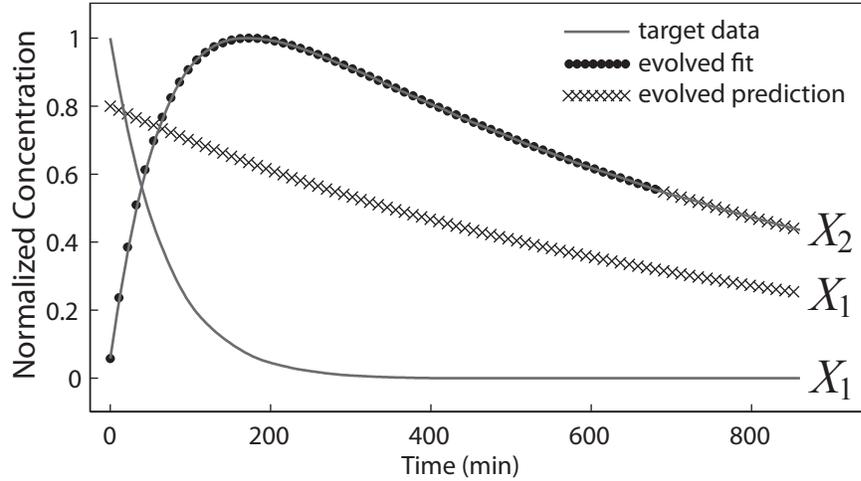


Figure 4.12. Behavior of the target system and an evolved model from the Physical 5-azacytidine Degradation Kinetics Section. The target data shown in the solid gray line were obtained from physical experiments performed by Argemí and Saurina (Argemí and Saurina 2007).

as the basis for model identification. A model structure resembling that of Equation 4.26 was frequently identified:

$$\begin{cases} \frac{dX_1}{dt} = \alpha X_1 \\ \frac{dX_2}{dt} = \beta X_1 - \chi X_2 + \delta \end{cases} \quad (4.27)$$

with initial values $y_1(0) = \varepsilon, y_2(0) = 0.0578$, where $\alpha, \beta, \chi, \delta, \varepsilon$ are floating point constants. Table 4.5 shows the frequency with which the form in Equation 4.27 was identified and the variability of the floating point constants across trials in which this form was identified. Regardless of whether the form in Equation 4.27 was identified or how low their test error was, the evolved models showed high variability in the values of the floating point constants across several trials.

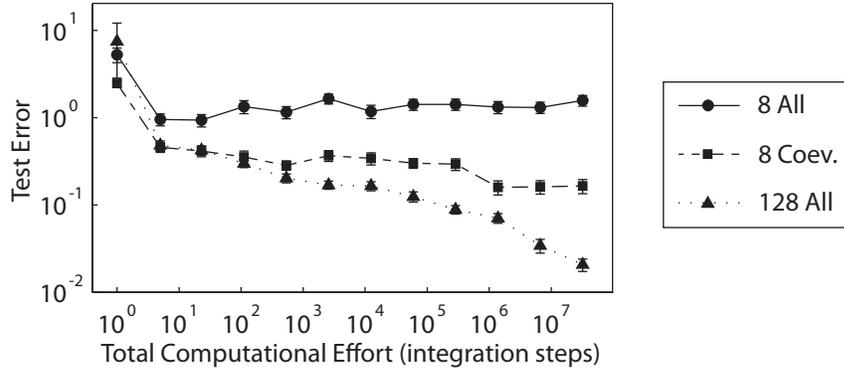


Figure 4.13. Summary of evolutionary runs for the Physical 5-azacytidine Degradation Kinetics section.

The initial value scanning approach was then used in an attempt to identify the model more precisely and reliably with X_1 as a hidden variable. The scan in this case was not successful at identifying models matching the experimental data. Even when models of the form shown in Equation 4.26 or Equation 4.27 were found, the nature of the coupled system of ODEs made identifying the correct parameters difficult. For example, with the 128 All method, this model was found:

$$\begin{cases} \frac{dX_1}{dt} = -0.0013X_1 \\ \frac{dX_2}{dt} = 0.0236X_1 - 0.0150X_2 \end{cases} \quad (4.28)$$

with initial values $y_1(0) = 0.8000, y_2(0) = 0.0578$. Figure 4.12 compares the response of this evolved model to the empirical data, showing that the behavior of the hidden variable X_1 was not predicted accurately even though the correct model form was recovered and the behavior of the measured variable X_2 was predicted accurately.

Table 4.5. Evolved models for the Physical 5-azacytidine Degradation Kinetics section. A successful trial indicates that the form given in Equation 4.27 was evolved in the trial.

Evaluation Method	Num. Successful Trials Out of 50	α	β	χ	δ	ε
128 All	15	-0.0032 ± 0.0032	0.2120 ± 0.5560	0.0127± 0.0032	0.0039 ± 0.0017	0.8600 ± 0.3445
8 Coev.	4	-0.0054 ± 0.0059	0.2683 ± 0.4881	0.2626 ± 0.5063	0.1074 ± 0.2087	0.8453 ± 0.3095
8 All	0	NA	NA	NA	NA	NA

Physical Pendulum

In ODE form, a simple pendulum is modeled with two dynamic variables, one for the angular displacement of the bob (θ) and one for its angular velocity (ω). However, the practice of referring to the variables as X_1 and X_2 will be continued, with X_1 representing angular displacement and X_2 representing angular velocity. Here, empirical data obtained from a physical pendulum by Bongard and Lipson were used (Bongard and Lipson 2007). These experimental measurements, shown in Figure 4.14, were the input to the GP algorithm that attempted to infer a model for the data.

As with the 5-azacytidine physical system, it was first assumed that all the data were available, that is, that both the angular displacement (X_1) and angular velocity (X_2) of the pendulum were measured as in the original experiment (Bongard and Lipson 2007). Using the 128 All evaluation method, models of the following form were readily obtained:

$$\begin{cases} \frac{dX_1}{dt} = X_2 \\ \frac{dX_2}{dt} = \alpha \sin(X_1) - \beta X_2 \end{cases} \quad (4.29)$$

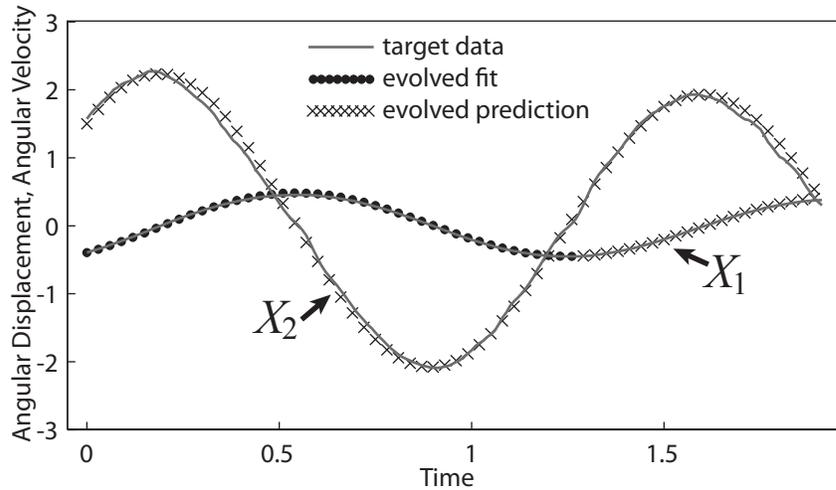


Figure 4.14. Behavior of the target system and an evolved model from the Physical Pendulum section. The target data shown in the solid gray line were obtained from physical experiments performed by Bongard and Lipson (Bongard and Lipson 2007).

with initial values $y_1(0) = -0.3933, y_2(0) = 1.5701$, where α, β are floating point constants. In all cases, α was approximately -20.0 and β was approximately 0.2 as in Bongard and Lipson 2007. This explanatory model matches the well-known model derivable from first principles with Newtonian physics and provided a baseline for comparison with subsequent experiments described next.

It was then assumed that only the angular displacement (X_1) was measured and that the angular velocity (X_2) was a hidden variable. Figure 4.15 summarizes the results of 50 independent trials using the empirical X_1 data as the basis for model identification. Despite many models achieving low test error, no models with the structure of Equation 4.29 were identified, and this hidden variable modeling problem appeared to be the most difficult example considered so far. However, the initial value scanning approach generated models of the following form with both the 128 All and 8 Coev. evaluation techniques:

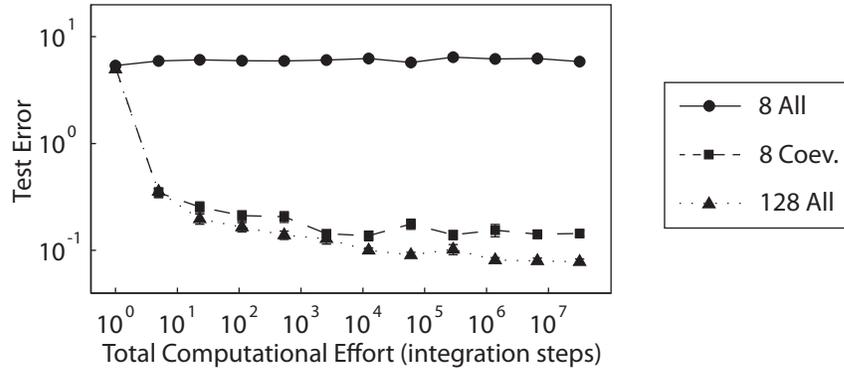


Figure 4.15. Summary of evolutionary runs for the Physical Pendulum section.

$$\begin{cases} \frac{dX_1}{dt} = X_2 \\ \frac{dX_2}{dt} = \alpha X_1 - \beta X_2 \end{cases} \quad (4.30)$$

with initial values $y_1(0) = -0.3933, y_2(0) = \chi$, where α, β, χ are floating point constants. In all cases, α was approximately -20.0 and β was approximately 0.2 . An example of one of these models is:

$$\begin{cases} \frac{dX_1}{dt} = X_2 \\ \frac{dX_2}{dt} = -19.8412X_1 - 0.2070X_2 \end{cases} \quad (4.31)$$

with initial values $y_1(0) = -0.3933, y_2(0) = 1.5000$. Figure 4.14 compares the behavior of this model to the empirical data, showing that the model accurately predicts both the measured and hidden variables despite being more parsimonious than the first principles model given in Equation 4.29.

Random Systems of ODEs

In order to study how the GP algorithm scales with increasing problem difficulty, performance as a function of target system complexity was studied for random systems of ODEs. These random systems were generated by building a random binary tree of operations and operands for each variable in the system. The expression corresponding to each tree was then simplified using MATLAB's Symbolic Math Toolbox⁷ to obtain an accurate measure of complexity for the system. Random systems with two variables and with three variables were generated. An example of a random ODE system with two variables and a complexity of 8 is:

$$\begin{cases} \frac{dX_1}{dt} = X_1 X_2 \\ \frac{dX_2}{dt} = X_1 - 4.4362 X_2 \end{cases} \quad (4.32)$$

with initial values $y_1(0) = 3.5121, y_2(0) = -8.3313$. An example of a random ODE system with three variables and a complexity of 17 is:

$$\begin{cases} \frac{dX_1}{dt} = \frac{4.0801}{X_2 - 7.1723} + 9.4551 \\ \frac{dX_2}{dt} = X_1^2 X_3^2 \\ \frac{dX_3}{dt} = 1.1222 + X_1 \end{cases} \quad (4.33)$$

with initial values $y_1(0) = -1.6493, y_2(0) = -0.8373, y_3(0) = 7.4122$.

⁷ <http://www.mathworks.com/products/matlab>

Figure 4.16 summarizes the results for random systems with two variables and shows performance in terms of test error for the best model found at the end of each run, as a function of model complexity. 50 independent trials with the 128 All fitness evaluation method were performed for each of 12 different model complexities ranging from 2 to 24. For each trial within each complexity class, a different random target was generated. In the experiments summarized in Figure 4.16a, both variables in the target system were measured. Test error was uniformly low and increased smoothly as a function of target complexity, both without measurement noise and with noise added. In the experiments summarized in Figure 4.16b, one of the variables in the system was hidden. The relatively flat test errors across different target complexities for both noisy and noise-free conditions indicate that the GP algorithm had difficulty identifying the more complex target models.

Figure 4.17 summarizes the results for random systems with three variables and 12 different model complexities ranging from 3 to 25. In the experiments summarized in Figure 4.17a, all three variables were measured. In Figure 4.17b, one variable was hidden and in Figure 4.17c, two variables were hidden. Again, the results collectively indicate that the more complex target models could not be reliably modeled when hidden variables were present, with little or no effect of added measurement noise.

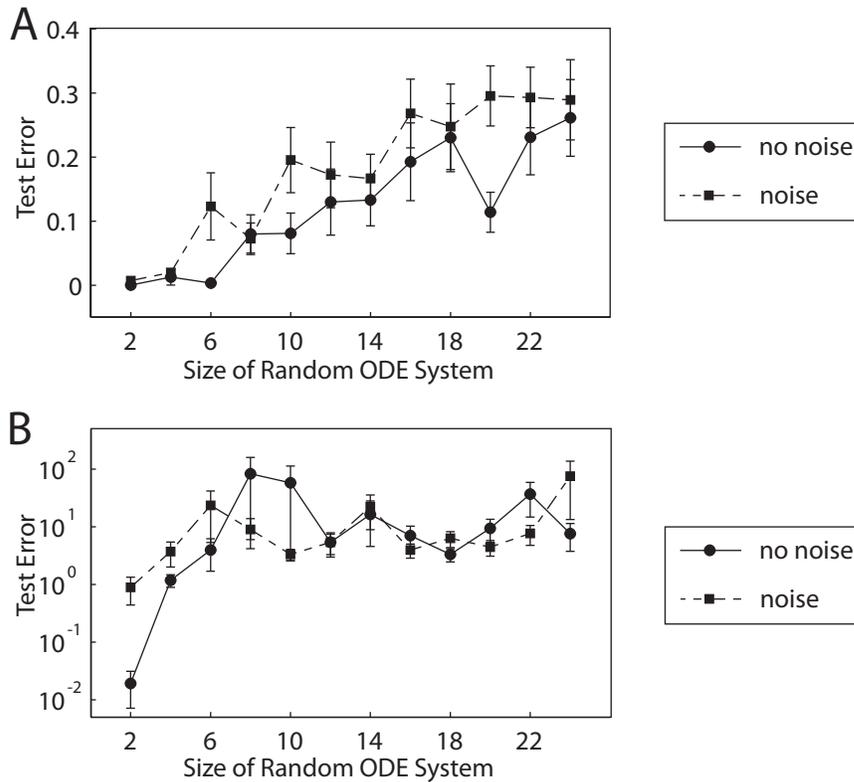


Figure 4.16. Summary of experiments with random systems of ODEs of two variables. The error of the best model on test data not seen during training was calculated at the end of each run. The mean of this test error is plotted for trials with and without added measurement noise using symbols as shown in the legend. Error bars represent \pm the standard error of the mean. Panel **A** shows results for trials in which both variables in the system were measured and panel **B** shows results for trials with one hidden variable.

Discussion

This chapter represents the first work on inferring systems of ODEs with hidden variables using GP. The goal was to reverse-engineer the exact form and parameters of the ground truth model in the case of synthetic data, or a parsimonious explanatory model in the case of physical data. Many of the results were promising, but three main challenges to achieving this goal were encountered.

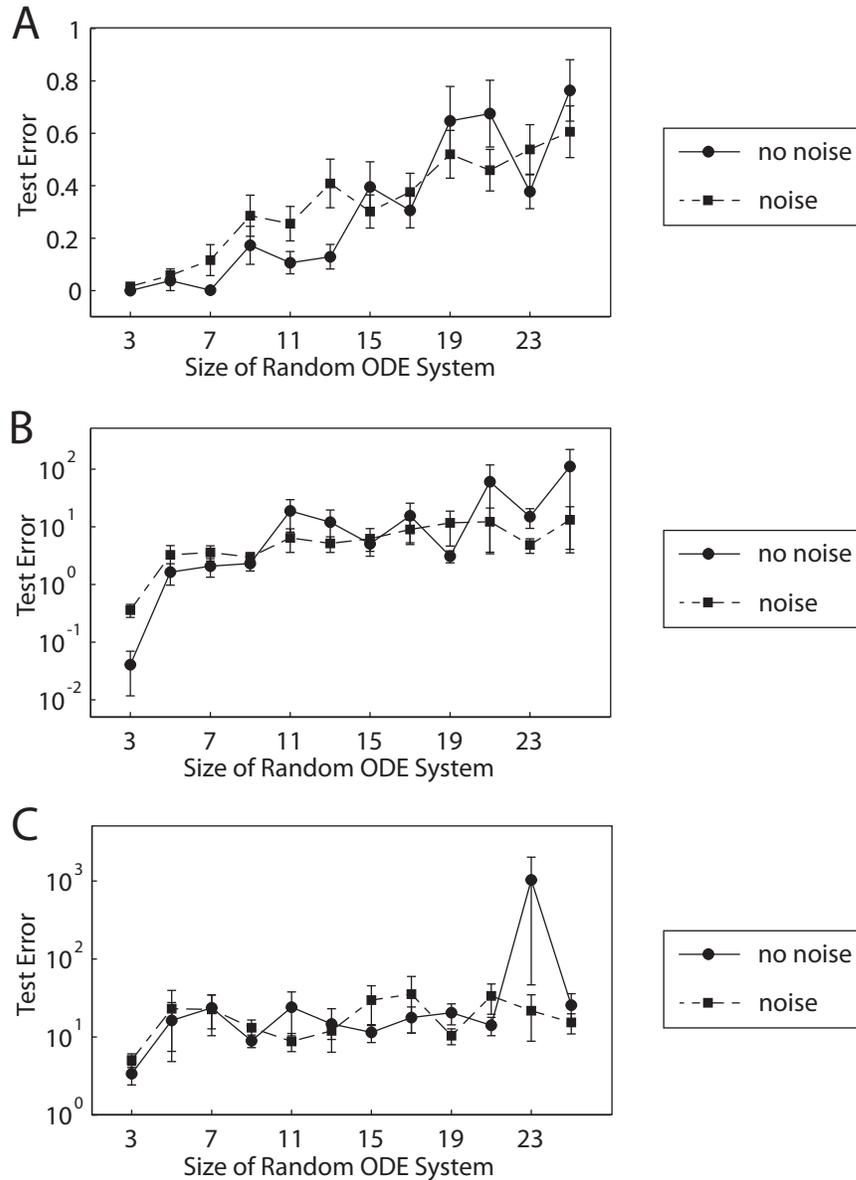


Figure 4.17. Summary of experiments with random systems of ODEs of three variables. Panel **A** shows results for trials in which all variables in the system were measured, panel **B** shows results for trials with one hidden variable, and panel **C** shows results for trials with two hidden variables.

As mentioned previously, the primary challenge is that ODE models with alternative forms and parameter values often appear as attractive local optima in the space of possible models (Figure 4.18). Disambiguating between these on the basis of error or parsimony criteria can be difficult in practice or even theoretically impossible,

true model, error = 0, size = 6

$$\frac{dx}{dt} = xy$$

$$\frac{dy}{dt} = \frac{y}{x}$$

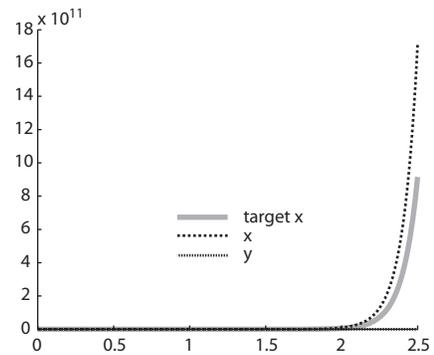
$$x(0) = 10, y(0) = 10$$

near-perfect candidate model, error = 0.0251, size = 6

$$\frac{dx}{dt} = xy$$

$$\frac{dy}{dt} = \frac{y}{x}$$

$$x(0) = 10, y(0) = 10.1$$



poor candidate model, error = 0.000006, size = 6

$$\frac{dx}{dt} = 10.0977x - 0.6482$$

$$\frac{dy}{dt} = -49.3525$$

$$x(0) = 10, y(0) = 0.1419$$

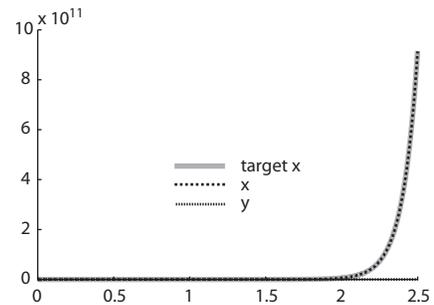


Figure 4.18. Ambiguity in nonlinear systems of ODEs. In this example, x is an observed variable and y is a hidden variable. The ground truth target model shown in the top row is almost identical to the evolved candidate model in the second row and only the evolved initial value for the y variable is slightly different. However, the error value that would be assigned to this apparently near-perfect candidate model is very high due to the large disparity between the target and observed x behavior. The other evolved candidate model in the third row appears to be a much worse candidate and bears almost no resemblance to the algebraic form of the target model. However, the error value that would be assigned to this apparently poor candidate model is very low due to the very close agreement between the target and observed x behavior.

which depends on the structure of model space and the nature of measurement noise that are unique to each modeling problem. This is essentially a problem of model non-identifiability (Figure 4.19). Many studies have addressed this issue for ODEs in an optimization context where the model structure is fixed but parameters are free to vary (Hengl et al. 2007, Raue et al. 2010). General statements can be made about the identifiability of systems of linear ODEs, although it is much more difficult to draw conclusions about the identifiability of arbitrary systems of nonlinear ODEs, especially when both the form and parameters of the model are free to vary (Chis et al. 2011, Grewal and Glover 1976, Vajda and Rabitz 1994, Walter and Pronzato 1997).

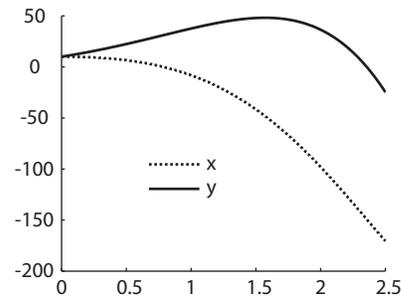
The simple examples considered here suggest that non-identifiability of both model structures and of parameter values play a role in shaping the fitness landscape. The initial value scanning approach was proposed as a means of systematically traversing one dimension of the landscape in an attempt to isolate regions in which the target model resides. In practice, this enabled recovery of the desired models much more often than would otherwise be possible. The primary downside of this approach, besides the computational effort involved, is that each trial in the scan produces a Pareto front of candidate models representing the best tradeoffs between model error and complexity found within that run. Independent trials of the scan often produced models of equivalent error and complexity, and distinguishing between these models would necessarily involve human effort and possibly domain knowledge. However, this type of post hoc human involvement in the modeling process is arguably very different than the a priori involvement that characterizes knowledge-based modeling. The assessment of high-quality candidate models generated by GP would conceivably

system 1

$$\begin{aligned}\frac{dx}{dt} &= x - y \\ \frac{dy}{dt} &= x + y \\ x(0) &= 10, y(0) = 10\end{aligned}$$

exact solution

$$\begin{aligned}x(t) &= 10e^t (\cos(t) - \sin(t)) \\ y(t) &= 10e^t (\cos(t) + \sin(t))\end{aligned}$$

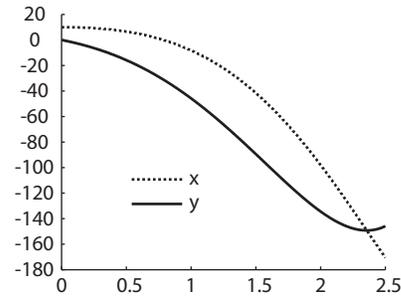


system 2

$$\begin{aligned}\frac{dx}{dt} &= y \\ \frac{dy}{dt} &= -2x + 2y \\ x(0) &= 10, y(0) = 0\end{aligned}$$

exact solution

$$\begin{aligned}x(t) &= 10e^t (\cos(t) - \sin(t)) \\ y(t) &= 20e^t (\sin(t))\end{aligned}$$



system 3

$$\begin{aligned}\frac{dx}{dt} &= 2x + y \\ \frac{dy}{dt} &= -2x \\ x(0) &= 10, y(0) = -20\end{aligned}$$

exact solution

$$\begin{aligned}x(t) &= 10e^{2t} (\cos(t) - \sin(t)) \\ y(t) &= -10e^{2t} (\cos(t))\end{aligned}$$

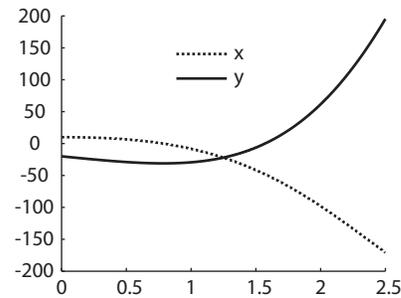


Figure 4.19. Non-identifiability in linear systems of ODEs. In each system, the observed variable is x and the hidden variable is y . Despite having completely different algebraic forms and different behavior of the y variable, the behavior of x is identical in every system, which can be seen in the exact solutions for $x(t)$ and in the plotted x data on the right. If only the observed x data from any one of the systems were available, there would be no way to determine which of the three systems generated the observed data.

involve less human effort than the manual creation of a knowledge database that would then be provided to a knowledge-based modeling algorithm.

The conservative attitude taken in this chapter was to consider trials unsuccessful unless the evolved models precisely matched particular model forms and

parameter values. However, in many cases trials that fail due to non-identifiability can still generate meaningful results. For example, consider the ideal rocket equations as described above in **Ideal Rocket Equations**. The target model for these experiments was given in Equation 4.16. The initial value scanning approach recovered the target model but also generated models such as:

$$\begin{cases} \frac{dX_1}{dt} = X_2 \\ \frac{dX_2}{dt} = \frac{2.0373}{X_3} \\ \frac{dX_3}{dt} = -1.0183 \end{cases} \quad (4.34)$$

with initial values $y_1(0) = 2.0, y_2(0) = 1.0, y_3(0) = 5.0929$. Equation 4.34 has parameter values that differ greatly from the target in Equation 4.16 and the predicted behavior of the hidden variable X_3 matches the behavior of X_3 in the target system very poorly. However, the parameter values in Equation 4.34 are very close to 10 times the values of the parameters in Equation 4.16. If target Equation 4.16 uses units of meters for position (X_1), meters/second for velocity (X_2) and kg for mass (X_3), then Equation 4.34 has nearly identical behavior in all variables if its units are simply reinterpreted as meters for X_1 , meters/second for X_2 , and $\text{kg} \cdot 10$ for X_3 . Although this is a relatively simple example, it shows that the essential scientific knowledge captured by a model can still be conveyed despite non-identifiability.

A second challenge to GP-based modeling with hidden variables is that the number and meaning of hidden variables would generally not be known in advance. Here, this issue was partially sidestepped by assuming that the number of relevant

variables in the system is known. This is not without justification as it would be possible to simply repeat any modeling trial with different numbers of hidden variables until the “correct” number of hidden variables was found. The correct number of hidden variables could be recognized as the number that resulted in the most accurate and parsimonious models. Alternatively, techniques from nonlinear dynamics theory such as correlation dimension analysis could be applied to estimate the relevant number of dynamic variables (Adachi et al. 2006, Gouesbet 1991, Kennel et al. 1992).

A third challenge unique to modeling with hidden variables is that the search space of possible models is much larger than other ODE modeling tasks. This is because the problem cannot be “partitioned” into simpler symbolic regression tasks (Bongard and Lipson 2007). For example, consider the problem of modeling a system of ODEs consisting of two variables X_1 and X_2 . If both X_1 and X_2 are measured, dX_1/dt and dX_2/dt can be estimated. A model for $dX_1/dt = f_1(t, X_1, X_2)$ can then be found separately from a model for $dX_2/dt = f_2(t, X_1, X_2)$. The observed data for X_1 and X_2 are substituted as necessary on the right hand side of the equation to evaluate candidate models. This is not possible when hidden variables are present and the entire system must be integrated every time a candidate model is to be evaluated. As a result, the size of the search space scales linearly with the number of variables in the system when all variables are measured, but scales exponentially with the number of variables when one or more hidden variables are present. This could account for the difficulty encountered in inferring the more complex randomly generated systems in the **Random Systems of ODEs** section above. It was hoped that some of this difficulty

could be offset by simply performing fewer integration steps and more candidate model evaluations with the 8 All evaluation method. However, the results indicate universally poorer performance of this method vs. the 128 All and 8 Coev. methods.

Despite these challenges, in many cases it was possible to reverse-engineer the form and parameters of target models. Success with the modest examples presented here could be extended in several ways. First, sophisticated parameter estimation and noise handling techniques could be incorporated directly into the GP algorithm (Cao et al. 2000, Qian et al. 2008). Although this could greatly increase the computational effort required to evaluate individual candidate models, the tradeoff might be desirable for many applications. Alternatively, additional parameter optimization could be applied as a post-processing, fine-tuning step for the evolved models. Techniques designed for multiresponse data might be particularly useful in this regard (Routray and Deo 2005, Stewart et al. 1992). Second, the traditional tree-based GP used here has been outperformed in many problem domains by linear GP (Brameier and Banzhaf 2007), grammar-based GP (Ryan et al. 1998, Whigham 1995), and other evolutionary algorithms. Studying the performance of these alternative approaches when evolving ODE models in the presence of hidden variables is a promising area for future research. Finally, instead of drawing a sharp distinction between free-form and knowledge-based approaches to automated modeling as in this chapter, the two might be fruitfully combined. Such a hybrid approach could be one way of circumventing the issues with non-identifiability encountered in a pure free-form algorithm while improving the flexibility of a pure knowledge-based method.

Most of the dynamical systems considered in this chapter have qualitative

properties that make them relatively amenable to modeling with the proposed algorithm. However, dynamical systems encountered in the real world often exhibit behavior that could make modeling with a GP-based approach such as the one described here considerably more difficult. For example, physical systems and their corresponding ODE models are often studied in the neighborhood of singularities that could make it difficult to obtain the data measurements needed to evaluate the fitness of a candidate model (Pnevmatikos 1985). Many systems exhibit a high sensitivity to one or more parameters in a region of interest, such as the laminar to turbulent flow transition boundary in a fluid (Sharp and Adrian 2004). Due to the large changes in observed behavior that result from small changes in parameter values, systems studied in the region of such bifurcation points could be particularly difficult to model with any approach that relies on a fitness gradient, including the one used in this chapter. As the algorithm described in the present work also relies on optimization of the initial values of ODEs in some cases, chaotic systems with high sensitivity to the choice of those initial values, such as those modeling weather, could further increase the complexity of the fitness landscape (Lorenz 1963). Finally, the degree of coupling between observed and hidden variables in a system may influence the ease with which the hidden variables can be modeled (Boccaletti 2008). Complete decoupling would make modeling of hidden variables impossible, as shown in the **Chemical Reaction Network** section above, whereas weak coupling could greatly increase the amount of observed data required to infer the algebraic forms describing any hidden variables. It will be important for future work in GP-based automated modeling of dynamical systems to address these complicating factors.

CHAPTER 5

CONCLUSION

Data-driven modeling is of growing importance as the pace at which scientific data is gathered continues to increase. The goal of this work was to address three challenges that present major barriers to the practical use of data-driven, free-form modeling algorithms for inferring explanatory models of natural phenomena. These challenges are three features commonly observed in interesting, naturally-occurring dynamical systems: complexity and nonlinearity, dynamics with multiple time scales, and dynamics that require hidden variables for interpretable and parsimonious representation as mathematical models.

The contributions of this work consist of three primary lines of research, each of which focuses on one of the aforementioned three challenges. In the first line of research, techniques for the automated design of analog electrical circuits were adapted and extended for the modeling of the types of complex, nonlinear systems often encountered in neurophysiology. In the second line of research, a new symbolic regression algorithm was proposed to facilitate the modeling of dynamical systems with multiple time scales. Finally, in the third line of research, the first steps were taken toward a practical genetic programming algorithm for the modeling of dynamical systems with hidden variables.

The work of science is never complete, and the research presented here could be extended in many ways. In the future, it is likely that the data-driven approach

followed in this work will be only one small part of sophisticated robotic and software-based artificial intelligence systems that scientists in all fields will routinely use for the automatic generation of complex, explanatory models.

APPENDIX 1

EVOLUTION OF SORTING ALGORITHMS

Introduction

Devising algorithms to sort the elements of a list is a classic computational problem. The simplicity of the problem statement and the importance of its efficient solution have made it one of the best studied problems in all of computer science (Cormen et al. 2009). In fact, sorting algorithms are perhaps the most common vehicle for introducing students to the basics of algorithm analysis (Kordaki et al. 2008). Several ingenious sorting algorithms have been proposed through the years, and most of the best solutions can be expressed with relative simplicity (Knuth 1973). Surprisingly though, sorting is not a completely “solved” problem in the sense that new sorting algorithms are still being invented to this day. See for example the recent work in Bender et al. 2006. In many ways then, sorting is an ideal test bed for artificial intelligence techniques that attempt to emulate human creativity and automatically devise solutions to computational problems.

One particularly promising such technique is evolutionary computation. In the broadest sense, evolutionary computation refers to the branch of artificial intelligence that is inspired by biology, and in particular, by the naturally-occurring process of evolution by natural selection (Fogel et al. 1966, Fogel 1995, De Jong 2006). Perhaps the best known evolutionary algorithm is the genetic algorithm (Holland 1975,

Mitchell 1996). In the classic genetic algorithm, solutions to a computational problem of interest are encoded into strings called chromosomes by analogy with the DNA-containing structure in biological cells. Several chromosomes form a population of candidate solutions to the problem. The best solutions to the problem are selected to survive into a subsequent generation, but before chromosomes are transferred into this new generation, variation is introduced into the population. The two classic variation operations are mutation, in which portions of a chromosome are altered, and recombination, in which portions of two different chromosomes are exchanged with each other. The cycle of mutation, recombination, and selection continues for several generations until a solution of high quality is obtained.

A landmark study by Hillis (1990) used a genetic algorithm to automatically construct sorting networks that rivaled the best known hand-designed networks. A sorting network is a special type of sorting algorithm for fixed-size input lists (Batcher 1968). The sequence of exchanges between elements of the list that is necessary to produce a sorted output list can be represented graphically, which is convenient for the purposes of efficient hardware implementations of the algorithm. Hillis represented the sequence of exchanges in a network as a sequence of ordered pairs in a chromosome. The chromosomes were then evolved using a genetic algorithm.

Of particular note in the work by Hillis is an excellent example of drawing inspiration from biology to improve the design of artificial intelligence techniques: the coevolution of sorting network topology with test input lists. Hillis found that when a fixed selection of input lists was used to test the sorting capabilities of candidate evolved networks, the ability of the evolved networks to sort arbitrary input lists

quickly reached a plateau. Hillis hypothesized that the fixed examples of input lists were causing evolved networks to exploit the unique features of the lists and achieve only a local optimum in the search space of all possible sorting networks. To address this issue, Hillis used coevolution to maintain selection pressure on the sorting networks. This involved a selection of input lists that could itself evolve and antagonistically attempt to counter-exploit any tendencies of the evolving networks to sort only particular lists and not all possible lists. Only with this coevolution of “hosts” (networks) and “parasites” (input lists) did Hillis obtain highly successful sorting networks for arbitrary input lists.

Although Hillis applied a genetic algorithm with success for the evolution of sorting networks, sorting networks are severely restricted versions of sorting algorithms in that a given network will only sort an input list of a particular size. Even assuming that a reasonable network can be found for a given input list size, the requirement that sorting be accomplished with successive exchanges of list elements means that a sorting network does not necessarily achieve the average case or even worst case complexity of the more familiar general sorting algorithms such as Bubble Sort, Merge Sort, or Quick Sort (Knuth 1973).

For evolving more general algorithms, a genetic algorithm would not suffice without considerable augmentation. One such augmentation, which is typically considered a distinct evolutionary algorithm in its own right, is genetic programming (GP). GP has built up an impressive track record as a general problem solving tool in the last 20 years (Koza 1992, Koza 1994, Koza et al. 1999, Koza et al. 2003). Traditional “tree-based” GP works by starting with a primordial soup of program

components, or building blocks (Figure 1.1a). These components are randomly combined into computer programs represented as expression trees (Figure 1.1b). For example the tree in the upper right portion of Figure 1.1b represents the simple arithmetic program $\exp(\pi * v)$. The set of these trees forms a population of candidate solutions to the computational problem of interest. The candidate programs that solve the problem best are selected to survive and transfer to another population of solutions. As with chromosomes in the genetic algorithm, variation operations are applied to individuals before they move into this new generation of solutions (Figure 1.1c). Although many different techniques for introducing variation into GP populations have been proposed, mutation of nodes and recombination of subtrees are the most common (Kouchakpour et al. 2009). Again as in a genetic algorithm, several cycles of selection and variation are performed until one or more of the surviving individuals are acceptable solutions to the computational problem of interest.

GP has been used in several prior studies to evolve general sorting algorithms. Among the first and best known of these is the work by Kinnear (Kinnear 1993a, Kinnear 1993b) that expands on similar work in Koza 1992. Kinnear used tree-based GP in the form of Lisp S-expressions. He restricted the possible building blocks to only seven Lisp functions hand-crafted for the sorting application. Notably, these included a function `dob1` that performs an operation for each element of a list. Also noteworthy is that with these building blocks, it was not possible for recursion or explicit subroutines to evolve. Even when he restricted the space of possible programs so radically, the results were modest. After spending what he anecdotally describes as “an unfortunate amount of time” experimenting with different techniques, Kinnear

was able to evolve Lisp trees that sorted all test input lists provided to it. All his reported evolved Lisp trees rely on a similar technique involving a `do1` loop within another `do1` loop. Here is one example that was hand simplified by Kinnear (1993a):

```
(do1 0
  (e1- *len*)
  (do1 0
    (e1- *len*)
    (swap (wismaller (e1+ index) index)
           index))))
```

Such algorithms appear to be equivalent to a variant of Bubble Sort with guaranteed $O(n^2)$ performance.

More recent studies have used non tree-based GP techniques to evolve sorting algorithms. Spector and colleagues used GP and a custom designed, stack-based language called Push to evolve programs reminiscent of those written in an assembly language (Spector et al. 2005). Their technique is closely related to linear GP, in which programs are represented as lists of assembly language commands (Brameier and Banzhaf 2001, Squillero 2005, Brameier and Banzhaf 2007). They were able to evolve a compact sorting algorithm equivalent to a variation on Bubble Sort when using building blocks very similar to those used by Kinnear. In addition, their system allowed for recursive programs to evolve, but it is not clear how difficult it would be for high quality recursive sorting algorithms such as Quick Sort to evolve in their system and with their choice of building blocks. Another study using GP to evolve sorting algorithms is that of Shirakawa and Nagao, who used a graph-based variant of GP called GRAPE (Shirakawa and Nagao 2007). In GRAPE, an operation on a piece of list data is represented as a node in a directed graph. Several such nodes, with

many possible execution flows, define a program. Evolutionary operations can mutate and recombine the nodes and their connections. The authors state that a graph-based approach results in a more natural and flexible representation of loops and other control structures than tree-based GP. However, they too obtain results similar to that of Kinnear above, and like Kinnear, they do not allow for recursion or explicit subroutines.

Given the shortcomings of previous studies, alternative approaches to evolving sorting algorithms with GP are worthy of study. In the present work, one essential requirement was that the evolved algorithm be compact, which should greatly simplify the process of evolution. This was arguably achieved in previous studies. However, here the goal is also to obtain readable programs. None of the previous attempts used particularly readable program representations, although of course they could be hand-translated into a more readable pseudocode or C-like procedural representation. In fact, directly evolving pseudocode is an attractive way of achieving compactness and readability in the domain of sorting algorithms, but this would require a custom interpreter for the pseudocode language to be built from scratch and was considered infeasible for present purposes.

Another major requirement for the present work was that the program representation be capable of handling recursion naturally. This was achieved in the representation used by Spector et al. (2005) with a linear GP-like technique, and so the possibility of using linear GP to evolve readable programs in the MIPS assembly language was initially considered here. However, during the course of this work, it eventually became clear that compact MIPS programs implementing sophisticated

algorithms like Quick Sort would have to be based on a library of custom MIPS functions and that such programs would likely never be as easy to read and understand as pseudocode or C-like procedural programs. Finally, another requirement not met by any prior work was a representation that allowed for subroutines to evolve. Explicit subroutines seem to be almost essential for readable implementations of many sorting algorithms including Quick Sort, which requires a partitioning subroutine, and Merge Sort, which requires a merging subroutine (Knuth 1973).

Taking all the above considerations into account, a representation in the C language appeared most appropriate. No other representation is as universally readable and easy-to-understand while still handling recursion and subroutines naturally. In addition, with appropriate choice of building blocks, it seemed likely that no other nontrivial representation could represent sophisticated algorithms in as compact a manner as C. To perform the evolutionary search through the space of C programs, this study relies on a recently-developed technique called grammar-based GP (O'Neil and Ryan 2003). In grammar-based GP, the representation of a program with which evolution operates is completely separate from the representation of the program that is actually evaluated for fitness. The mapping between the two representations relies on a BNF grammar that can generate all possible programs.

In this work, a grammar-based GP system was used to evolve sorting algorithms in C with a representation that encouraged the evolution of compactness, readability, recursion, and the use of subroutines. The details of the GP system designed with these requirements in mind are discussed next in *Methods*, followed by a description of experiments performed with the system in *Experiments*. Finally, this

appendix ends with some closing remarks and a discussion of potential avenues for future work.

Methods

Overview

There are three essential components in a system implementing grammar-based GP (O’Neil and Ryan 2003). These components will be referred to as the “evolver,” the “mapper,” and the “evaluator” (Figure A1.1). The evolver is typically a genetic algorithm that operates on strings of numbers, or chromosomes. As in any genetic algorithm, these chromosomes are mutated, recombined, selected, and otherwise operated on over the course of several generations with the goal of maximizing their fitness. However, in grammar-based GP, fitness calculations require additional steps. When it is time for the fitness of a particular chromosome to be evaluated, the information in that chromosome must first be translated into an executable program. This translation is performed by the mapper, which relies on a BNF grammar capable of generating all possible programs that can potentially be evolved. The grammar is what defines the space of programs through which evolution searches, and so the selection of the appropriate program building blocks described in the grammar is a very important consideration when using grammar-based GP. As a result, a significant amount of effort in this work was devoted to investigating the appropriate building blocks (see **Building Blocks**, below). Finally, the evaluator takes an executable program generated by the mapper and runs it to determine its fitness. The

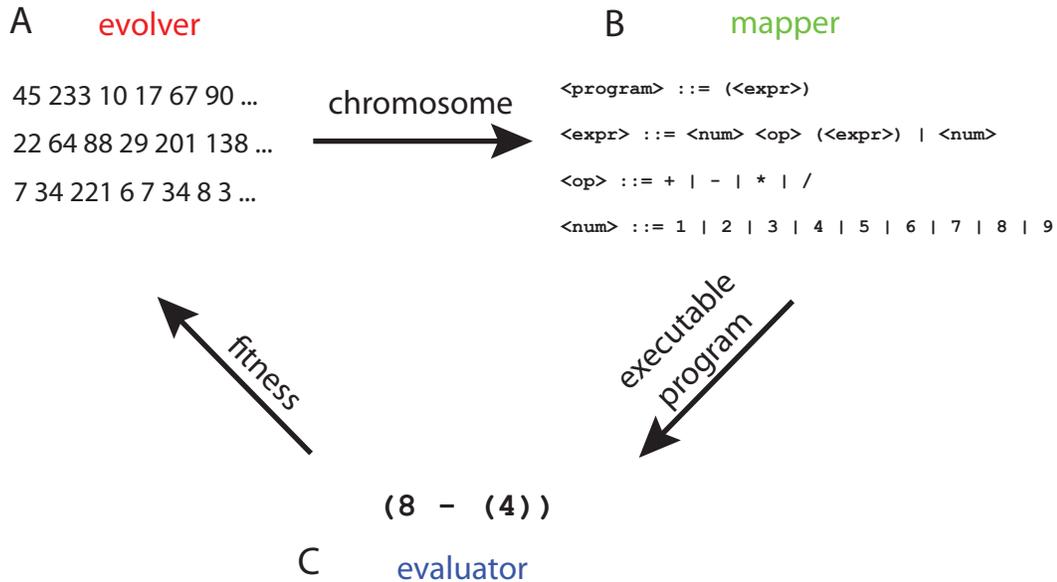


Figure A1.1. Grammar-based genetic programming. **A)** A genetic algorithm is used by the evolver to search for chromosomes comprised of strings of integers. **B)** To evaluate the fitness of a given chromosome as part of the standard genetic algorithm loop, the evolver passes it to the mapper. The mapper uses a BNF grammar to translate the chromosome into a program. In this case, the grammar is for simple arithmetic expressions. The mapper works by using each successive integer in the chromosome to choose an expression from the right hand side of the derivation rules in the grammar. For example, if the chromosome [22 64 88 29 201 138] is to be translated, the mapper begins with the first derivation rule and calculates $22 \bmod 1 = 0$, where 1 is the number of expressions to choose from on the right hand side of the first derivation rule. The 0th expression (`<expr>`) is chosen to replace the non-terminal `<program>`. The leftmost non-terminal in the resulting expression is `<expr>` and so the mapper next shifts focus to the derivation rule for `<expr>`. Because $64 \bmod 2 = 0$, the 0th rule on the right hand side of the derivation rule for `<expr>` is chosen to replace `<expr>`. This process continues until no non-terminals are present in the string. If the end of the chromosome is reached before then, the mapper wraps around to the beginning of the chromosome. The resulting program in this case is `(8 - (4))`. **C)** The mapper passes the program to the evaluator, which wraps the program in invariant header and footer code and then executes the program to calculate its fitness. Finally, the evaluator reports the fitness of the program back to the evolver and the cycle continues.

fitness is of course dependent on the exact nature of the problem, and this is discussed in more detail in **Fitness Calculations** below.

The Evolver and Mapper

C programs were represented in the system by strings of integers in the range 0 - 255. A standard genetic algorithm implemented using the GALib 2.47 C++ library⁸ was used to evolve these strings. Although a large number of parameters in GALib can be used to control the evolutionary process, preliminary results suggested that default values generally gave results similar to custom values. The main parameters used are experiment-specific and are discussed below in *Experiments*.

The mapping between chromosomes used in the genetic algorithm and executable C programs was performed by the LibGE 0.26 C++ library⁹. LibGE takes strings of integers as input and maps them to executable programs with a BNF grammar as shown in Figure A1.1. This straightforward process has relatively few modifiable parameters. One of these, the “wrap factor,” sets a limit on the number of passes through the integer string that are made by the mapper as it attempts to map all non-terminal symbols to terminal symbols. In theory, the grammars used in this project can generate infinitely long programs and require a wrapping limit, but in practice no valid programs ever approached the wrapping cutoff of 100 that was used. The CPU time required by both the evolver and the mapper is negligible compared to the CPU time required by the evaluator, and so their performance is not discussed further.

⁸ <http://lancet.mit.edu/ga>

⁹ <http://bds.ul.ie/libGE>

Building Blocks

Before attempting to write the BNF grammar that would be used by the mapper, a thorough study of different representations for general sorting algorithms was undertaken. The goal was to find the appropriate C language building blocks that would best meet the requirements discussed in *Background* above. One possible set of C building blocks that was considered was simply every construct in the entire C language. BNF grammars for ISO C99 are available (Harbison and Steele 2002) and they would certainly allow any conceivable sorting algorithm to be expressed. However, the use of such a large set of building blocks would radically increase the size of the search space, making evolution more difficult. Also, even if efficient search algorithms could be successfully evolved, the resulting programs would be much less compact than desired.

A more reasonable approach was suggested by the presentation of linked list sorting algorithms that is used in many textbooks. Code and pseudocode for sorting algorithms often take a particularly readable and compact form when linked lists are used. This is especially the case when many different sorting algorithms are to be described in a consistent and easy-to-understand framework. For example, the `do_b1` function in Kinnear's work facilitates the evolution of Bubble Sort-like iterative algorithms but would be of little use in compactly expressing the more elegant forms of Quick Sort. In contrast, both Bubble Sort and Quick Sort have reasonably compact implementations in terms of linked list operations.

Knowing that the building blocks would be expressed as linked list operations, focus was then shifted to studying the tradeoffs inherent in different levels of building

block abstraction. For example, if “concrete” is the opposite of “abstract,” then the most basic variant of Bubble Sort can be described in a relatively concrete form using linked lists:

```
void concrete_bubble_sort(list input, int a, int b)
{
    int i;
    int j;
    int k;

    i = a;
    while(i < b)
    {
        j = a;
        while(j < b)
        {
            if(value(element(input, j)) > ...
               value(element(input, j + 1)))
            {
                k = value(element(input, j));
                assign(element(input, j), ...
                       value(element(input, j + 1)));
                assign(element(input, j + 1), k);
            }
            j = j + 1;
        }
        i = i + 1;
    }
}
```

where `input` is a list of integers to be sorted, `a` is the starting index of the list, and `b` is the ending index of the list. `element(x, y)` returns the `y`th element of list `x` and `assign(x, y)` assigns the integer value `y` to list element `x`. (In this code listing and all others below, “...” indicates that the statement continues uninterrupted on the next line.) Alternatively, the most basic variant of bubble sort can be described in a somewhat more compact and abstract form that also uses linked lists:

```
void abstract_bubble_sort(list input, int a, int b)
{
    int i;
    int j;

    i = a;
    while(i < b)
    {
        j = a;
```

```

        while(j < b)
        {
            if(value(element(input, j)) > value(element ...
                (input, j + 1))
            {
                swap(input, j, j + 1);
            }
            j = j + 1;
        }
        i = i + 1;
    }
}

```

Assuming that the `swap()` operation has an $O(1)$ linked list implementation, then there does not seem to be much reason to include the lower level `assign()` operation in the building blocks. On the other hand, `assign()` could potentially be used to implement higher level operations other than `swap()`, and so it still might have value as one of the building blocks.

As another example of tradeoffs in using different levels of building block abstraction, consider a relatively concrete version of Selection Sort:

```

void concrete_selection_sort(list input, int a, int b)
{
    int i;
    int j;
    int k;
    int l;

    i = a;
    while(i <= b)
    {
        j = i + 1;
        k = value(element(input, i));
        l = i;
        while(j <= b)
        {
            if(value(element(input, j)) < k)
            {
                k = value(element(input, j));
                l = j;
            }
            j = j + 1;
        }
        if(l != i)
        {
            swap(input, i, l);
        }
        i = i + 1;
    }
}

```

and a more abstract version:

```
void abstract_selection_sort(list input, list result)
{
    list p;

    make(&p);
    while(not_empty(input))
    {
        p = delete_min(input);
        insert_last(result, p);
    }
}
```

Unlike with the `swap()` operation above, it is not reasonable to assume that the `delete_min()` operation, which finds the smallest element of the input list and deletes it, would have an $O(1)$ implementation. This is a problem for present purposes because `abstract_selection_sort` appears to be worst case $O(n)$, whereas `concrete_selection_sort` appears to be worst case $O(n^2)$. In reality though, the use of the more abstract `delete_min()` operation hides the true worst case $O(n^2)$ nature of `abstract_selection_sort`. If the fitness of a candidate algorithm is to be based on complexity, then clearly there is a tradeoff between accuracy of the fitness value and compactness of representation.

To keep the number of building blocks as small as possible while maximizing their expressiveness and maximizing the flexibility in choosing the appropriate level of abstraction for a given application, a sorting API in C that manipulates doubly linked lists of integers was written. The full API from the user's (or mapper's) point of view is given in Appendix 2. Notice that it contains all the concrete and abstract building blocks used in the four sorting algorithms shown above in this section. It also contains building blocks necessary for the compact implementation of many other algorithms spanning all the major classes of sorting algorithms. These include

concrete and abstract implementations of Insertion Sort, Heap Sort, Merge Sort, and Quick Sort. An implementation of 14 example sorting algorithms using the API is given in Appendix 3.

A BNF grammar that generates sorting algorithms using the building blocks of the sorting API is given in Appendix 4. This full version of the grammar was used as the basis for all algorithm evolution experiments, and it allows for recursive calls and the generation of explicit subroutines. However, for some experiments as discussed below, subsets of the full grammar were used in order to adjust the nature of the search space of possible programs. In all algorithm examples using the grammar, including those in Appendix 3, the algorithm is passed the argument list

```
(list entries, list result, int a, int b)
```

where **entries** is the original list of values to be sorted, **result** is an empty list, **a** is the starting index of **entries**, and **b** is the final index of **entries**. Unless otherwise noted, an algorithm is assumed to store the final sorted list in **entries**. Due to the precise nature of some algorithms however, the final sorted list may instead be stored in **result** or passed as the return value from the function. These alternative methods are required to achieve compactness and readability for some of the algorithms implemented in Appendix 3. To conserve space, some of the elements generated by the grammar that have no effect on program execution, such as variables that are declared but not used, are not shown in code examples below. In addition, all functions are formatted with the Allman indentation style for readability, although strictly speaking, this formatting information is not generated by the grammar.

Fitness Calculations

In the grammar-based GP system used in this work, the evaluator takes the evolved algorithm generated by the mapper, adds some header and footer code, and then pipes the resulting program directly to the gcc 4.3 compiler¹⁰ for rapid generation of machine code. The last action of the evaluator is to execute the program and collect the fitness value of the algorithm, which is actually called by footer code containing `main()`. The fitness value is then returned to the evolver, which uses the information to continue the next generation of evolution.

Robust evaluation of a sorting algorithm's fitness would require that it be tested with numerous different input lists as in Kinnear 1993a-b). Coevolution of input lists and algorithms as in Hillis (1990) could also be used. However, in the current version of the system used here, an evolved algorithm is only called between one and ten times with an unsorted list of 30 integers. As discussed below, this simple means of evaluation was more than adequate for the present purposes.

Even when an algorithm is only called one or a few times, the fitness value could still be calculated in many different ways. Most would agree that an ideal sorting algorithm both sorts the input list and does so with the smallest number of operations possible. However, other dimensions of sorting algorithm quality are also conceivable, such as memory usage or number of disk reads/writes required. Here, the focus is on the degree to which the input list is sorted and the number of operations required to do so. The degree to which the input list is sorted can be quantified simply by the number of inversions in the list. An inversion is defined as a pair of elements in

¹⁰ <http://gcc.gnu.org>

a list that is out of order (Knuth 1973). The number of operations is somewhat more difficult to define exactly. However, a measure that approximates the information provided by big O notation is to simply count the number of `while()` loop iterations in an evolved algorithm. To do so, a statement `++global_count;` was inserted at the end of every `while()` loop in order to increment a global counter variable that is always initialized to 0 before the algorithm is called. The final values of `global_count` for 14 different sorting algorithms implemented using the API are listed in Table A1.1, which shows the surprisingly close correspondence between final `global_count` value and the qualitative notion of “algorithm quality.”

Experiments

Initial Experiments

As an initial experiment, it was of interest to confirm that valid programs could be generated with the methods described above. Another important goal of this initial experiment was to confirm that at least some variation in fitness levels is present in randomly generated programs containing the `global_count` variable. Fitness in this experiment was defined simply as the final `global_count` value if the program ran normally, 0 if the program did not run, or 0 if the program ran longer than 3 seconds. `global_count` was always initialized to 0 and fitness was being maximized. Note that no attempt was made in this initial experiment to evaluate fitness in terms of sorting ability, and the number of operations performed on lists was to be maximized instead of minimized as a way of obtaining programs that performed non-trivial

Table A1.1. Time complexity of different sorting algorithms. Here, time complexity is estimated with the `global_count` method described in the text. Each algorithm was run 1000 times with a randomly generated (unsorted) input list of integers. The mean and standard deviation of the loop iterations counted with `global_count` are shown. The counted value for “abstract” algorithms is generally not informative because each fundamental operation has complexity greater than $O(1)$. However, among the “concrete” methods, the counted value is easily able to distinguish between higher quality algorithms (such as Quick Sort, Merge Sort, and Heap Sort) and lower quality algorithms (such as Bubble Sort, Selection Sort, and Insertion Sort). This counting method even distinguishes between slightly different variants of the same algorithm, as in Bubble Sort and Quick Sort.

Algorithm	Mean <code>global_count</code> $\pm \sigma$
Abstract Bubble Sort	870.00 \pm 0.00
Concrete Bubble Sort	870.00 \pm 0.00
Abstract Bubble Sort (variant 2)	515.82 \pm 39.94
Concrete Selection Sort	465.00 \pm 0.00
Abstract Bubble Sort (variant 1)	464.00 \pm 0.00
Abstract Bubble Sort (variant 3)	376.98 \pm 17.47
Concrete Insertion Sort	209.67 \pm 12.70
Abstract Quick Sort	163.00 \pm 13.20
Concrete Quick Sort	126.74 \pm 12.59
Concrete Merge Sort	104.70 \pm 2.59
Concrete Quick Sort (variant 1)	58.29 \pm 3.37
Concrete Heap Sort	44.00 \pm 0.00
Abstract Selection Sort	30.00 \pm 0.00
Abstract Insertion Sort	30.00 \pm 0.00

computation. To make things even simpler, a version of the grammar in Appendix 4 was used that did not permit subroutines or recursion. Aside from those restrictions, all building blocks provided by the sorting API were present and the grammar was the same as in Appendix 4. One generation and a population size of 10,000 were used. This corresponds essentially to a random search in the space of possible programs with 10,000 iterations.

This random search experiment required approximately 45 minutes to run on a single CPU core. A high proportion (84.1%) of the 10,000 randomly generated

programs ran to completion, or were “valid.” In other words, despite the fact that nothing prevented infinite loops, floating-point exceptions, memory leaks, or other problems that might lead to a Linux segmentation fault, the grammar-based approach usually generated valid candidate programs that did not suffer from those problems. An example of three valid programs is shown in Figure A1.2., and an example of three invalid programs is shown in Figure A1.3. Unfortunately, all valid programs had the same fitness value of 0. An investigation into this revealed that apparently all programs containing `while()` loops either did not run at all and were considered invalid, looped infinitely and were considered invalid, or were valid but looped 0 times due to the logical condition in the `while()` statement not being met.

for() Loops

`for()` loops are C statements equivalent to more general versions of `while()` loops. In `for()` loops, initialization and update statements with side-effects are automatically included along with a logical conditional statement that determines whether the loop iterates again. A `while()` loop on the other hand, has only the logical conditional statement. Based on the results from the first experiment, it was hypothesized that the lack of appropriate initialization and update statements prevented many randomly generated programs containing `while()` loops from being valid or otherwise from iterating in a sensible fashion, and so `while()` statements:

```
while(<logic_expression>
{
    <statement>
    ++global_count;
}
<statement>
```

```

void evolved_sort(list entries, list result, int a, int b)
{
    return;
}

void evolved_sort(list entries, list result, int a, int b)
{
    int j = 0;
    boolean y = FALSE;
    list p;
    list q;
    list r;
    make(&p);
    make(&q);
    make(&r);

    if(0 < 2)
    {
    }
    else
    {
        insert_inorder(copy_of_element(q, value(delete_any(r))), r);
        put(p, delete_min(entries), j);
    }
    y = TRUE;
    return;
}

void evolved_sort(list entries, list result, int a, int b)
{
    int l = 0;
    list q;
    make(&q);

    while(1 < (value(q) * l))
    {
        ++global_count
    }
    return;
}

```

Figure A1.2. Three examples of randomly generated, valid programs obtained in the first experiment. In all three cases, the programs run to completion but `global_count` is not incremented.

were replaced with `for ()` statements:

```

for(<sideeffect_expression>; <logic_expression>; ...
sideeffect_expression)
{
    <statement>
    ++global_count;
}
<statement>

```

but the grammar was otherwise the same as in the first experiment. One generation and a population size of 10,000 were used. The experimental conditions were

```

void evolved_sort(list entries, list result, int a, int b)
{
    int l = 0;
    list p;
    list r;
    list s;
    make(&p);
    make(&r);
    make(&s);

    while((value(delete_first(p)) - 2) > 0 && not_empty(element(s, ...
value(copy_of_element(r, ((value(delete_last(r)) + value(delete_any(r)) - ...
value(delete_min(p))) * value(copy_of_element(r, ((value(delete_any(s)) - ...
(value(r) * l)) - ((l * 2) / (value(r) + (value(r) + (l + a))))))))))
    {
        ++global_count;
    }
    return;
}

void evolved_sort(list entries, list result, int a, int b)
{
    list r;
    list s;
    make(&r);
    make(&s);
    swap(r, (b / value(delete_first(s))), 1);
    return;
}

void evolved_sort(list entries, list result, int a, int b)
{
    while(2 != value(entries))
    {
        ++global_count;
    }
    return;
}

```

Figure A1.3. Three examples of randomly generated, invalid programs obtained in the first experiment. The first and second programs give segmentation fault errors by attempting to access empty lists and the third leads to an infinite loop that must be externally killed. In all code listings here and below, “...” indicates that the statement continues on the next line. Also, variable initialization statements at the beginning of the function and extraneous semicolons are not shown to conserve space.

otherwise the same as above. Again, this random search experiment required approximately 45 minutes to run. 84.4% of the 10,000 programs were valid, virtually the same as in the previous experiment, and again all valid programs had a fitness of 0. It appears that valid programs using `for()` loops failed to increment the `global_count` variable for the same reasons that valid programs using `while()` loops failed in the experiment above. Figure A1.4 shows three examples of valid programs

using `for()` loops that nonetheless do not increment `global_count`.

The above results show that programs generated with the grammar are not only syntactically correct, they are mostly valid and so in a sense are semantically correct as well. It is clear though that even in this restricted space of all possible programs, a given arbitrary behavior -even one as simple as incrementing the `global_count` variable a few times- is extremely unlikely to be stumbled upon randomly. This “needle in a haystack” problem can plague any machine learning method (Turner et al. 2009). Couched in the terms of evolutionary computation, the algorithm needs at least some variation in fitness levels if individuals are to be evolved on the basis of fitness. With no such fitness gradient present even for the simple experiments above, it seems extremely unlikely that any attempts to evolve more complicated behaviors such as sorting will ever succeed.

Seeding Experiments

One way to address the needle in a haystack problem is to assist the evolutionary optimization process by “seeding” the initial population, whereby it is initialized with nonrandom individuals. With that motivation, a series of seeding experiments were performed next. The first such experiment was similar to the first experiment above except that the entire initial population was seeded with these individuals:

Figure A1.4. Three examples of valid programs randomly generated using the grammar from the `for()` Loops experiment (next page). For these experiments, `while()` loops have been replaced with `for()` loops. In all three cases, the programs run to completion but `global_count` is not incremented.

```

void evolved_sort(list entries, list result, int a, int b)
{
    int j = 0;
    list p;
    list s;
    make(&p);
    make(&s);

    b = (value(s) * value(element(p, value(s))));
    for(b = 0; j <= value(s); assign(entries, value(s)))
    {
        ++global_count;
    }
    return;
}

void evolved_sort(list entries, list result, int a, int b)
{
    int k = 0;
    boolean y = FALSE;
    list p;
    list q;
    list r;
    list s;
    make(&p);
    make(&q);
    make(&r);
    make(&s);

    if(not_empty(entries))
    {
    }
    else
    {
        for(a = value(delete_min(p)); ((value(p) - (2 + ((b / (value(r) / ...
value(r))) * (2 - 0)) - ((a * (a + b) / (k + (1 - value(s)))) ...
* (1 + value(q)))))) / value(p)) > value(p); b = value(entries))
        {
            y = FALSE;
            ++global_count;
        }
        y = TRUE;
    }
    y = TRUE;
    return;
}

void evolved_sort(list entries, list result, int a, int b)
{
    int i = 0;
    int j = 0;
    int l = 0;
    boolean y = FALSE;
    list q;
    make(&q);

    for(insert_inorder(entries, entries); not_empty(q); y = FALSE)
    {
        ++global_count;
    }
    if((((i + 0) + ((0 / value(p)) / 1)) + j) > 1)
    {
    }
    else
    {
    }
    return;
}

```

```

void evolved_sort(list entries, list result, int a, int b)
{
    int i = 0;

    while(i < 2)
    {
        i = (i + 1);
        ++global_count;
    }
    return;
}

```

It was hypothesized that it would be easier to evolve programs with the desired behavior if seed programs of this form were used; a sensible framework for incrementing the `global_count` variable is already in place and evolution only needs to alter the framework in small ways to achieve large values of `global_count`. For example, simply setting the conditional statement to `i < (2 * 2)` would result in a larger `global_count` value than any attained by programs in the previous experiments above.

Again, the desired behavior was not sorting but much simpler: “increment the `global_count` variable as much as possible without running over 3 seconds.” For this experiment however, the full evolutionary algorithm was employed with a population size of 40 and 10,000 generations, which took about 9.5 hours to complete on a single CPU core. The mutation rate was set at 0.01 per base per generation, meaning that each integer in a chromosome was randomly switched to another integer with a probability of 0.01 each generation. The recombination rate was set at 0.9, meaning that each chromosome recombined with another randomly chosen chromosome in the population with a probability of 0.9 each generation. Simple one-point crossover was used for this purpose. The best evolved program is shown below:

```

void evolved_sort(list entries, list result, int a, int b)
{
    int j = 0;
    int k = 0;
    list p;
    list r;
    make(&p);
    make(&r);

    while((((2 - value(element(r, (b - value(p)))))) + ...
(j * 2)) + k) > a || not_empty(p))
    {
        insert_last(r, entries);
        ++global_count;
    }
    return;
}

```

and the dynamics of program evolution during the run are shown in Figure A1.5.

Seeding appears to have facilitated a type of evolution known as punctuated equilibrium, which is evolution with occasional large increases in fitness instead of numerous smaller increases. This is not a surprise given that the search space likely suffers badly from the needle in a haystack problem as compared with other search spaces that have successfully been tackled by genetic programming, such as symbolic regression (Augusto and Barbosa 2000). As suggested by Figure A1.5, the final program evolved from the seed program in two main epochs that occurred in generations 3577 - 3602 and then in generations 4373 - 4411.

Closer examination of the population changes that occurred during these epochs showed that only one intermediate program form was present in between the seed individual and the final best evolved program:

```

void evolved_sort(list entries, list result, int a, int b)
{
    int k = 0;
    int l = 0;
    boolean y = FALSE;
    list q;
    make(&q);

```

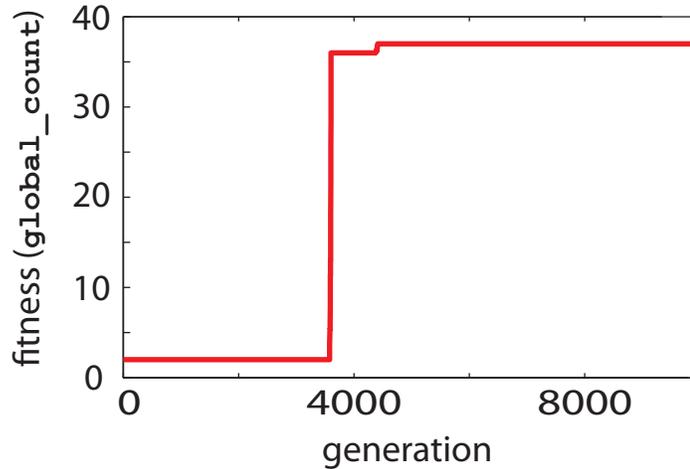


Figure A1.5. Mean fitness of the population in the seed experiment as a function of generation.

```

while(value(entries) >= 1 && ((k - value(q)) + 0) / ...
(value(delete_min(entries)) * value(q)) == k)
{
    while(not_empty(q))
    {
        y = TRUE;
        ++global_count;
    }
    ++global_count;
}
return;
}

```

At generation 3577, this individual was discovered and over the next 25 generations, it came to dominate until it was the only remaining version of the program in the population. A similar population shift occurred at generation 4373 when the best evolved program was discovered.

An important control experiment in any study of optimization methods is a random search that evaluates the same number of individuals as the evolutionary search. If this random search performs as well or better than the evolutionary search, then using GP to evolve these programs would be pointless as it would be easier to just randomly make programs until an acceptable one is found by chance. 10,000

generations of 40 individuals means that 400,000 programs were evaluated with evolutionary search in the seed experiment above, and so the corresponding random search control would be the evaluation of 400,000 randomly mutated seed individuals. The results of doing so showed that 99.85% of these 400,000 randomly mutated individuals had fitness values equal to or worse than the seed individual and 0.15% had better fitness values, although no valid programs had fitness equal to or better than the best evolved individual in the evolutionary search. Interestingly, four programs had a fitness better than the best evolved individual in the evolutionary search. These all turned out to be invalid programs for which errors were not caught at compile or run time. An example of one of these is this program that would loop infinitely were it not for the undefined overflow behavior of integer variables:

```
void evolved_sort(list entries, list result, int a, int b)
{
    int i = 0;
    int j = 0;
    boolean y = FALSE;

    i = 1;
    while(i < ((j + (i + 1)) * i))
    {
        y = TRUE;
        i = (i + 1);
        ++global_count;
    }
    return;
}
```

Evolution of Sorting Algorithms

As a final experiment, an attempt was made to evolve any algorithm that could sort at least one input sequence, without regard to efficiency or whether it could sort arbitrary sequences. This approach was inspired by Kinnear's work, which is by far the best documented success in sorting algorithm evolution that has been reported in the literature (Kinnear 1993a, Kinnear 1993b). He was able to evolve algorithms that

sorted at least some test input sequences fairly easily by using very large population sizes. Aside from his representation, there is no obvious reason why he would be able to succeed with tree-based GP where grammar-based GP would fail. To match the experimental conditions as closely as possible to that used by Kinnear, the grammar was reduced to only the building blocks necessary to implement an “Abstract Bubble Sort” as shown in **Building Blocks** above. In addition, a population size of 1000 was evolved for 1000 generations and the number of output list inversions was used as the fitness measure. Each candidate program was tested with ten randomly generated lists instead of just one. This experiment took about 20 hours to complete on one CPU core.

Unfortunately, evolution found a clever loophole. This was the final best program, which was obtained in the very first generation:

```
void evolved_sort(list entries, list result, int a, int b)
(
    int i = 0;
    int j = 0;

    swap(element(entries, a), value(element(entries, a)), j);
    while(value(element(entries, 1)) >= (i + (b + a)))
    {
        ++global_count;
    }
    return;
}
```

The level of inversions is low in the resulting output lists from this program not because they are properly sorted, but because `swap()` is taking only individual list elements (lists of size 1) as the first argument instead of the entire input list. This disrupts the chain of pointers in the input list and causes smaller output lists than input lists, which on average have fewer inversions. This problem is caused by the fact that individual list elements have the same type as full lists, which was intentional in the

API as many algorithms (such as Merge Sort) must operate on multiple-element sublists. It is not clear how to prevent problems like this while still allowing for maximum representational flexibility.

Conclusion

The goal of this work was to use grammar-based genetic programming to automatically create efficient sorting algorithms that are readable, compact, and that could take advantage of recursion and explicit subroutines as a natural part of the representation. Two major unexpected problems arose. The first relates to the lack of a fitness gradient in the search space. Optimization algorithms typically work by finding a partial solution to the problem and then incrementally improving it. In the case of sorting algorithms, it is very difficult to find a partially valid program that partially sorts the data; rather it is more of an all-or-nothing, needle in a haystack problem.

The second major problem is likely due to the use of C as the representation. Although a C-based representation is desirable for the easy-to-read results that it could in theory produce (in contrast with say, a graph-based representation), it appears that issues specific to the language make evolution more difficult than it would otherwise be. The diversity of function argument and return types in particular seems to have caused problems, as in the last experiment above. Tree-based GP deals with this by simply making every function argument and return type identical (Koza 1992), but it is not clear how to address this in the C-based approach here, which was designed to

facilitate evolution of a wide variety of different sorting algorithms.

Besides data type issues, another interesting avenue for future research is to more fully explore the possibilities of evolutionary search. Although standard mutation and recombination were used here, more sophisticated variation operations have been proposed and implemented for GP (Kouchakpour et al. 2009). Only one machine with a single-core processor was available for the current work, and so larger experiments and parameter searches were not possible here. However, if more powerful machines were used and if parallelization of fitness evaluation was implemented, such experiments could be feasible.

APPENDIX 2

C API FOR SORTING PROGRAMS

This Appendix lists the complete sorting API used as the building blocks for the sorting algorithm evolution experiments described in Appendix 1. The “behind-the-scenes” code of the API uses some data structures provided in GLib 2.24¹¹.

```
/* Basic list manipulation */

/* An abstract list a. See element() and value() for accessing
 * members of the list. */
list a;

/* An abstract boolean variable x. Only possible values are
 * TRUE and FALSE. */
boolean x;

/* Allocate memory for empty list a. */
void make(list *a);

/* Return the bth element of list a as a list of length 1.
 * Lists are accessed starting from 0. */
list element(list a, int b);

/* Return a copy of the bth element of list a as a list of
 * length 1. */
list copy_of_element(list a, int b);

/* Return the value of the 0th element of list a as an int. */
int value(list a);

/* Return TRUE if list a has length >= 1. */
boolean not_empty(list a);

/* Exchange elements b and c of list a. */
void swap(list a, int b, int c);

/* Set value of the 0th element of list a to b. */
void assign(list a, int b);

/* Functions for deleting list elements */

/* Randomly delete one element from the input list a and return
 * the deleted element as a list of length 1. */
list delete_any(list a);

/* Delete the element with largest value from the input list
 * a and return the deleted element as a list of length 1. */
list delete_max(list a);
```

¹¹ <http://library.gnome.org/devel/glib>

```

/* Delete the element with smallest value from the input list
 * a and return the deleted element as a list of length 1. */
list delete_min(list a);

/* Delete the 0th element from the input list a and return
 * the deleted element as a list of length 1. */
list delete_first(list a);

/* Delete the last element from the input list a and return
 * the deleted element as a list of length 1. */
list delete_last(list a);

/* Functions for inserting list elements */

/* Insert list b of length 1 into list a such that list a is
 * in correctly sorted order. */
void insert_inorder(list a, list b);

/* Insert list b of length 1 into list a at index c. */
void put(list a, list b, int c);

/* Prepend list b of length 1 before list a. */
void insert_first(list a, list b);

/* Append list b of length 1 after list a. */
void insert_last(list a, list b);

/* Append list b (of any length) after list a. */
void append_list(list a, list b);

```

APPENDIX 3

IMPLEMENTATION OF 14 SORTING ALGORITHMS

This Appendix demonstrates that the sorting API given in Appendix 2 and the BNF grammar given in Appendix 4 can be used to implement a very broad range of simple to sophisticated sorting algorithms in an efficient and readable way. Comments before each function describe the algorithm using the same names shown in Table A1.1. The BNF grammar given in Appendix 4 can generate all of the functions below, with four caveats: 1) To conserve space, some of the elements generated by the grammar that would have no effect on the execution of the program, such as variables that are declared but not used and extraneous semicolons are not shown. 2) The generic function name `evolved_sort` specified in the grammar is changed to a more descriptive name for each sorting function. 3) All functions are formatted with the Allman indentation style for readability, although strictly speaking, this formatting information is not generated by the grammar. 4) As in other code listings throughout this work, “...” indicates that the statement continues without interruption on the next line.

```
/* abstract bubble sort. The sorted list is stored in the variable "entries." */
void abstract_bubble_sort1(list entries, list result, int a, int b)
{
    int i = 0;
    int j = 0;

    i = a;
    while(i < b)
    {
        j = a;
        while(j < b)
        {
```

```

        if(value(element(entries, j)) > value(element(entries, j + 1)))
        {
            swap(entries, j, j + 1);
        }
        j = j + 1;
        ++global_count;
    }
    i = i + 1;
    ++global_count;
}

/* concrete bubble sort. The sorted list is stored in the variable "entries." */
void concrete_bubble_sort1(list entries, list result, int a, int b)
{
    int i = 0;
    int j = 0;
    int k = 0;

    i = a;
    while(i < b)
    {
        j = a;
        while(j < b)
        {
            if(value(element(entries, j)) > value(element(entries, j + 1)))
            {
                k = value(element(entries, j));
                assign(element(entries, j), value(element(entries, ...
                    j + 1)));
                assign(element(entries, j + 1), k);
            }
            j = j + 1;
            ++global_count;
        }
        i = i + 1;
        ++global_count;
    }
}

/* abstract bubble sort (variant 2). The sorted list is stored in the variable
* "entries." */
void abstract_bubble_sort3(list entries, list result, int a, int b)
{
    int i = 0;
    boolean x = TRUE;

    while(x)
    {
        x = FALSE;
        i = a;
        while(i < b)
        {
            if(value(element(entries, i)) > value(element(entries, i + 1)))
            {
                swap(entries, i, i + 1);
                x = TRUE;
            }
            i = i + 1;
            ++global_count;
        }
        ++global_count;
    }
}

/* concrete selection sort. The sorted list is stored in the variable "entries." */
void straight_selection_sort(list entries, list result, int a, int b)
{
    int i = 0;

```

```

int j = 0;
int k = 0;
int l = 0;

i = a;
while(i <= b)
{
    j = i + 1;
    k = value(element(entries, i));
    l = i;
    while(j <= b)
    {
        if(value(element(entries, j)) < k)
        {
            k = value(element(entries, j));
            l = j;
        }
        j = j + 1;
        ++global_count;
    }
    if(l != i)
    {
        swap(entries, i, l);
    }
    i = i + 1;
    ++global_count;
}
}

/*abstract bubble sort (variant 1). The sorted list is stored in the variable
* "entries." */
void abstract_bubble_sort2(list entries, list result, int a, int b)
{
    int i = 0;
    int j = 0;

    i = a + 1;
    while(i <= b)
    {
        j = a;
        while(j <= (b - i))
        {
            if(value(element(entries, j)) > value(element(entries, j + 1)))
            {
                swap(entries, j, j + 1);
            }
            j = j + 1;
            ++global_count;
        }
        i = i + 1;
        ++global_count;
    }
}

/* abstract bubble sort (variant 3). The sorted list is stored in the variable
* "entries." */
void abstract_bubble_sort4(list entries, list result, int a, int b)
{
    int i = 0;
    boolean x = TRUE;

    b = b + 1;
    while(x)
    {
        x = FALSE;
        b = b - 1;
        i = a;
        while(i < b)
        {

```

```

        if(value(element(entries, i)) > value(element(entries, i + 1)))
        {
            swap(entries, i, i + 1);
            x = TRUE;
        }
        i = i + 1;
        ++global_count;
    }
    ++global_count;
}
}

```

/* concrete insertion sort. The sorted list is stored in the variable "entries." */
void straight_insertion_sort(list entries, list result, int a, int b)

```

{
    int i = 0;
    int j = 0;
    list p;
    make(&p);

    i = a;
    while(i <= b)
    {
        p = element(entries, i);
        j = i;
        while(value(element(entries, j - 1)) > value(p))
        {
            put(entries, element(entries, j - 1), j);
            j = j - 1;
            ++global_count;
        }
        put(entries, p, j);
        i = i + 1;
        ++global_count;
    }
}

```

/* abstract quick sort. The sorted list is stored in the variable "entries." */
void abstract_quick_sort(list entries, list result, int a, int b)

```

{
    list p;
    list q;
    list r;
    list s;
    make(&p);
    make(&q);
    make(&r);
    make(&s);

    if(not_empty(entries))
    {
        r = delete_first(entries);
        while(not_empty(entries))
        {
            s = delete_first(entries);
            if(value(s) < value(r))
            {
                insert_last(p, s);
            }
            else
            {
                insert_last(q, s);
            }
            ++global_count;
        }
        abstract_quick_sort(p, result, 0, 0);
        abstract_quick_sort(q, result, 0, 0);
        append_list(entries, p);
        insert_last(entries, r);
    }
}

```

```

        append_list(entries, q);
    }
}

/* concrete quick sort. The sorted list is stored in the variable "entries." Note
 * that this function calls subroutine1, which follows. */
void array_quick_sort(list entries, list result, int a, int b)
{
    int i = 0;

    if((b - a + 1) > 0)
    {
        i = subroutine1(entries, result, a, b, 0);
        array_quick_sort(entries, result, a, i - 1);
        array_quick_sort(entries, result, i + 1, b);
    }
}

/* subroutine1 function called by concrete quick sort */
int subroutine1(list entries, list aux, int a, int b, int c)
{
    int i = 0;
    int j = 0;

    i = a + 1;
    while(value(element(entries, i)) < value(element(entries, a)) && i <= b)
    {
        i = i + 1;
        ++global_count;
    }
    j = b;
    while(value(element(entries, j)) > value(element(entries, a)))
    {
        j = j - 1;
        ++global_count;
    }
    while(i < j)
    {
        swap(entries, i, j);
        i = i + 1;
        while(value(element(entries, i)) < value(element(entries, a)) ...
        && i <= b)
        {
            i = i + 1;
            ++global_count;
        }
        j = j - 1;
        while(value(element(entries, j)) > value(element(entries, a)))
        {
            j = j - 1;
            ++global_count;
        }
        ++global_count;
    }
    if(a < j)
    {
        swap(entries, a, j);
    }
    return j;
}

/* concrete merge sort. The sorted list is stored in the return value. Note
 * that this function calls subroutine1, which follows. */
list concrete_merge_sort(list entries, list result, int a, int b)
{
    int i = 0;
    list p;
    list q;
    make(&p);

```

```

make(&q);

if(a == b)
{
    return copy_of_element(entries, a);
}
else
{
    i = (a + b) / 2;
    p = concrete_merge_sort(entries, result, a, i);
    q = concrete_merge_sort(entries, result, i + 1, b);
    return subroutinel(p, q, 0, 0, 0);
}
}

/* subroutinel function called by concrete merge sort */
list subroutinel(list entries, list aux, int a, int b, int c)
{
    list p;
    make(&p);

    while(not_empty(entries) && not_empty(aux))
    {
        if(value(element(entries, 0)) < value(element(aux, 0)))
        {
            insert_last(p, delete_first(entries));
        }
        else
        {
            insert_last(p, delete_first(aux));
        }
        ++global_count;
    }
    append_list(p, entries);
    append_list(p, aux);
    return p;
}

/* concrete quick sort (variant 1). The sorted list is stored in the variable
* "entries." Note that this function calls subroutinel and subroutine2, which
* follow. */
void median_quick_sort(list entries, list result, int a, int b)
{
    int i = 0;
    int j = 0;

    if((b - a + 1) <= 1)
    {
        ;
    }
    else
    {
        if((b - a + 1) == 2)
        {
            if(value(element(entries, a)) > value(element(entries, b)))
            {
                swap(entries, a, b);
            }
        }
        else
        {
            j = (a + b) / 2;
            subroutinel(entries, result, a, j, b);
            if((a + 1) != j)
            {
                swap(entries, a + 1, j);
            }
            i = subroutine2(entries, result, a + 1, b - 1, 0);
            median_quick_sort(entries, result, a, i - 1);
        }
    }
}

```

```

        median_quick_sort(entries, result, i + 1, b);
    }
}

/* subroutine1 function called by concrete quick sort (variant 1) */
void subroutine1(list entries, list aux, int a, int b, int c)
{
    if(value(element(entries, a)) > value(element(entries, b)))
    {
        swap(entries, a, b);
    }
    if(value(element(entries, a)) > value(element(entries, c)))
    {
        swap(entries, a, c);
    }
    if(value(element(entries, b)) > value(element(entries, c)))
    {
        swap(entries, b, c);
    }
}

/* subroutine2 function called by concrete quick sort (variant 1) */
int subroutine2(list entries, list aux, int a, int b, int c)
{
    int i = 0;
    int j = 0;

    i = a + 1;
    while(value(element(entries, i)) < value(element(entries, a)) && i <= b)
    {
        i = i + 1;
        ++global_count;
    }
    j = b;
    while(value(element(entries, j)) > value(element(entries, a)))
    {
        j = j - 1;
        ++global_count;
    }
    while(i < j)
    {
        swap(entries, i, j);
        i = i + 1;
        while(value(element(entries, i)) < value(element(entries, a)) ...
            && i <= b)
        {
            i = i + 1;
            ++global_count;
        }
        j = j - 1;
        while(value(element(entries, j)) > value(element(entries, a)))
        {
            j = j - 1;
            ++global_count;
        }
        ++global_count;
    }
    if(a < j)
    {
        swap(entries, a, j);
    }
    return j;
}

```

```

/* concrete heap sort. The sorted list is stored in the variable "entries." Note
* that this function calls subroutine1 and subroutine2, which follow. */
void concrete_heap_sort(list entries, list result, int a, int b)
{
    int i = 0;

    subroutine1(entries, result, a, 0, 0);
    i = a;
    while(i > 0)
    {
        swap(entries, 0, i);
        subroutine2(entries, result, 0, i - 1, 0);
        i = i - 1;
        ++global_count;
    }
}

/* subroutine1 function called by concrete heap sort. Note that this function calls
* subroutine2, which follows. */
void subroutine1(list entries, list result, int a, int b, int c)
{
    int i = 0;

    i = a / 2;
    while(i >= 0)
    {
        subroutine2(entries, result, i, a, 0);
        i = i - 1;
        ++global_count;
    }
}

/* subroutine2 function called by concrete heap sort and subroutine1
void subroutine2(list entries, list result, int a, int b, int c)
{
    int i = 0;

    i = 2 * a;
    if(i <= b)
    {
        if(i < b && value(element(entries, i)) < value(element(entries, ...
i + 1)))
        {
            i = i + 1;
        }
        if(value(element(entries, a)) < value(element(entries, i)))
        {
            swap(entries, a, i);
            subroutine2(entries, result, i, b, 0);
        }
    }
}

/* abstract selection sort. The sorted list is stored in the variable "result." */
void abstract_selection_sort(list entries, list result, int a, int b)
{
    list p;
    make(&p);

    while(not_empty(entries))
    {
        p = delete_min(entries);
        insert_last(result, p);
        ++global_count;
    }
}

```

```
/* abstract insertion sort. The sorted list is stored in the variable "result." */
void abstract_insertion_sort(list entries, list result, int a, int b)
{
    list p;
    make(&p);

    while(not_empty(entries))
    {
        p = delete_any(entries);
        insert_inorder(result, p);
        ++global_count;
    }
}
```

APPENDIX 4

GRAMMAR FOR SORTING PROGRAMS

This Appendix lists a BNF grammar that generates sorting programs using the building blocks from the sorting API given in Appendix 2. The number of subroutines (`subroutine1` and `subroutine2`) is limited to two, based on the observation that no well-known general purpose sorting algorithm requires more than two subroutines to express in a readable fashion with C. Also, the size of the grammar was held down by allowing some types of invalid programs to potentially be generated. For example, recursive function calls made through the `<generic_function>` non-terminal do not take into account the return type with which the function was declared. It seems difficult to account for this in a pure BNF grammar without greatly increasing the grammar's size, using pointers of type void, or another undesirable technique. As in other code listings throughout this work, “...” indicates that the statement continues without interruption on the next line.

```
<program> ::= <aux1> <aux2> <pri>

<aux1> ::= ; | <type> subroutine1(list entries, list aux, int a, int b, int c) ...
        {int i = 0; int j = 0; int k = 0; int l = 0; ...
        boolean x = TRUE; boolean y = FALSE; list p; list q; ...
        list r; list s; make(&p); make(&q); make(&r); make(&s); ...
        <statement> return;}

<aux2> ::= ; | <type> subroutine2(list entries, list aux, int a, int b, int c) ...
        {int i = 0; int j = 0; int k = 0; int l = 0; ...
        boolean x = TRUE; boolean y = FALSE; list p; list q; ...
        list r; list s; make(&p); make(&q); make(&r); make(&s); ...
        <statement> return;}

<pri> ::= <type> evolved_sort(list entries, list result, int a, int b){int i = 0; ...
        int j = 0; int k = 0; int l = 0; boolean x = TRUE; ...
        boolean y = FALSE; list p; list q; list r; list s; make(&p); ...
        make(&q); make(&r); make(&s); <statement> return;}
```

```

<statement> ::= <sideeffect_expression>; <statement>
             | while(<logic_expression>){<statement> ++global_count;} <statement>
             | if(<logic_expression>){<statement>}else{<statement>} <statement>
             | ;

<logic_expression> ::= <logic_ident>
                     | <generic_function>
                     | <logic_function>
                     | <numeric_expression> <logic_op> <numeric_expression>
                     | <logic_expression> <logic_conn> <logic_expression>

<numeric_expression> ::= <numeric_function>
                       | <generic_function>
                       | (<numeric_expression> <numeric_op> <numeric_expression>)
                       | <numeric_ident>

<list_expression> ::= <list_function>
                    | <generic_function>
                    | <list_ident>

<sideeffect_expression> ::= <sideeffect_function>
                          | <generic_function>
                          | <numeric_variable> = <numeric_expression>
                          | <logic_variable> = <logic_constant>

<numeric_op> ::= + | - | * | /

<logic_op> ::= \< | \> | \<= | \>= | == | !=

<logic_conn> ::= && | \|\|

<numeric_ident> ::= <numeric_variable> | <numeric_constant>

<numeric_variable> ::= i | j | k | l | a | b | c

<numeric_constant> ::= 0 | 1 | 2

<list_ident> ::= p | q | r | s | entries | aux | result

<logic_ident> ::= <logic_variable> | <logic_constant>

<logic_variable> ::= x | y

<logic_constant> ::= TRUE | FALSE

<type> ::= void | int | list | boolean

<logic_function> ::= not_empty(<list_expression>)

<numeric_function> ::= value(<list_expression>)

<list_function> ::= delete_any(<list_ident>)
                 | delete_max(<list_ident>)
                 | delete_min(<list_ident>)
                 | delete_first(<list_ident>)
                 | delete_last(<list_ident>)
                 | element(<list_ident>, <numeric_expression>)
                 | copy_of_element(<list_ident>, <numeric_expression>)

<sideeffect_function> ::= insert_inorder(<list_expression>, <list_expression>)
                      | insert_first(<list_expression>, <list_expression>)
                      | insert_last(<list_expression>, <list_expression>)
                      | append_list(<list_expression>, <list_expression>)
                      | put(<list_expression>, <list_expression>, ...
                          <numeric_expression>)
                      | swap(<list_expression>, <numeric_expression>, ...
                          <numeric_expression>)
                      | assign(<list_expression>, <numeric_expression>)

```

```
<generic_function> ::= evolved_sort(<list_expression>, <numeric_expression>, ...  
    <numeric_expression>)  
    | subroutine1(<list_expression>, <numeric_expression>, ...  
    <numeric_expression>)  
    | subroutine2(<list_expression>, <numeric_expression>, ...  
    <numeric_expression>)
```

APPENDIX 5

MACHINE LEARNING OF OPTIONS TRADING STRATEGIES

Introduction

Several studies reported in the last 30 years have attempted to predict changes in stock prices using machine learning methods (Tay and Shen 2002, Cao and Tay 2003, Enke and Thawornwong 2005, Hassan 2007). However, most of the studies published to date focus on relatively low risk/low reward investments and on the development of long term, “buy and hold” trading strategies. Although many investors have successfully used such methods, there are a variety of other investment strategies that carry much higher potential for short-term rewards, albeit with correspondingly higher risks.

One such strategy is to invest in financial derivatives known as stock options. Stock options are attractive primarily because of the leverage they provide; they allow the purchaser to indirectly control large numbers of stock shares with a relatively small up-front investment. For a comprehensive discussion of options see McMillan 2001. Briefly, the purchase of one call option contract for a company XYZ gives the buyer of the contract the right to buy 100 shares of XYZ’s stock, a right which can be exercised any time up until the expiration date of the contract. A put option contract is similar except that it gives the buyer of the contract the right to sell 100 shares instead of buy them. There is one trading day each month during which all options contracts

that expire during that month cease trading. For example, in this study, options data were obtained for two such expiration cycles: 9/20/10 - 10/15/10 and 10/18/10 - 11/19/10. In the first trading cycle, options contracts expiring in October, 2010 ceased trading on 10/15/10 and in the second trading cycle, options contracts expiring in November, 2010 ceased trading on 11/19/10.

Like shares of a stock, options contracts are traded on the open market and prices are negotiated between buyers and sellers. The value of a contract depends on several factors, foremost of which is the strike price of the contract. Continuing the example above, if the call contract had a strike price of \$50, then the contract holder has the right to buy 100 shares of company XYZ's stock for \$50 per share. If one share of XYZ stock is currently valued at \$55, then the value of the contract will be (at least) $\$55 - \$50 = \$5$. (The value of the contract in this case would actually be \$500 dollars because 100 underlying shares are controlled with a single contract, but by convention, value is always expressed on a per share basis.) Other factors, such as time until expiration, also affect the contract's value. The most important implication of option contract valuation as described above is that typical option prices are far more variable than typical stock prices over short term periods of days, hours, and even minutes.

In this study, options price data were collected for several different underlying stocks to investigate whether these rapid changes in price provide opportunities for short-term profit that have been overlooked by previous machine learning studies of automated investing. The basic strategy was to gather and process financial data from publicly available data sources, use machine learning techniques to train models that

classify an option contract price time series as a “good buy” or “poor buy,” and finally to test the performance of the classifier on completely unseen option contract price time series data. The details involved in each of these steps are discussed below.

Methods

Data Collection

The acquisition and processing of raw data for this study was a computationally-intensive process that involved writing a “web crawler” in Perl from scratch to download and analyze a large quantity of dynamic html data from financial websites. For this project, the options data were collected from Yahoo Finance¹². This was done in accordance with Yahoo’s Terms of Service, which at the time data were collected, did not explicitly prohibit “bots” and other means of automatically accessing web content. The web crawler was provided with a list of 55 underlying stock symbols and was used to automatically obtain a snapshot of the current prices of all options contracts sold on the Chicago Board Options Exchange (CBOE) related to the underlying stock symbol at approximately 100 - 200 second intervals. These 55 stocks were generally chosen to provide a good sample of stocks with historically high trading volume (liquidity) and high variability, two features generally assumed to be important for any successful options trading strategy (McMillan 2001). The data acquisition process was carried out every trading day from 9:30 AM - 4:00 PM, starting on 9/20/10. The relevant options time series were extracted from the raw data

¹² <http://finance.yahoo.com>

by linearly interpolating all time series and aligning them with respect to a master “trading timeline.” This trading timeline ignored nontrading days and parts of trading days in which the market was closed.

Only data for the most actively traded options contracts were used. This was extracted using a two step data-pruning process. First, all data collected from the 9/20/10 - 10/15/10 trading cycle were thrown out unless it was for a contract that expired in October, 2010. Similarly, all data collected from the 10/18/10 - 11/19/10 trading cycle were thrown out unless it was for a contract that expired in November, 2010. The justification for this step was that options are most heavily traded in the month during which they expire. Secondly, all time series were thrown out unless there were at least 100 total changes in price during the time series, a step that further increased the average liquidity of the useful data. Altogether, these two data-pruning steps excluded about 80% of the total data collected, although the remaining data were assumed to be the most useful 20%.

The end result of the above data collection process was two large data structures that each represented one month of data: one 518 x 1580 matrix for data collected from the 9/20/10 - 10/15/10 trading cycle and one 590 x 1975 matrix for data collected from the 10/18/10 - 11/19/10 trading cycle. Here, 518 and 590 are the total number of options time series in their respective months, and 1580 and 1975 are the total number of time points in the time series in their respective months. The total number of time points was defined by setting the resolution of the trading timeline to 300 seconds, somewhat longer than the average time resolution of the raw data as collected. Figure A5.1 below shows some time series from the data set in the form

described.

Data Processing with Sliding Windows

Although the raw time series data could be directly manipulated by machine learning methods, a simpler approach was used that describes each time series by a much lower dimensional set of features in order to generate the data for a supervised learning process. This involved calculating features and class membership on a sliding window basis, whereby each time series was divided into several windows of arbitrary length I and O (Dietterich 2002). Input features x were calculated based on the I data points in an “input window,” and a single output result y was calculated based on the O data points in the “output window.” For the present purposes, the output of a trading strategy consisted of a decision to either purchase or not purchase an options contract, and so options contracts were labeled as either a “good buy” or a “poor buy.” Accordingly, the possible classes of y are either 1 = “good buy” or 0 = “poor buy.” The class value y was calculated for a given output window by examining the values in the output window and comparing them to the final value v in the input window. “Gain” of an output window was defined as $gain = (g/v - 1)$, where g is the highest price in the output window. Another important parameter is referred to below as the “gain threshold.” Specifically, a gain value of more than *gain threshold* for an output window resulted in a y value of 1 corresponding to a “good buy” recommendation and otherwise, the y value was set to 0. Data for learning, validation, and test purposes were obtained by positioning the start of the input window at time t and the start of the output window at time $t + I$, performing feature calculation and class calculation, and

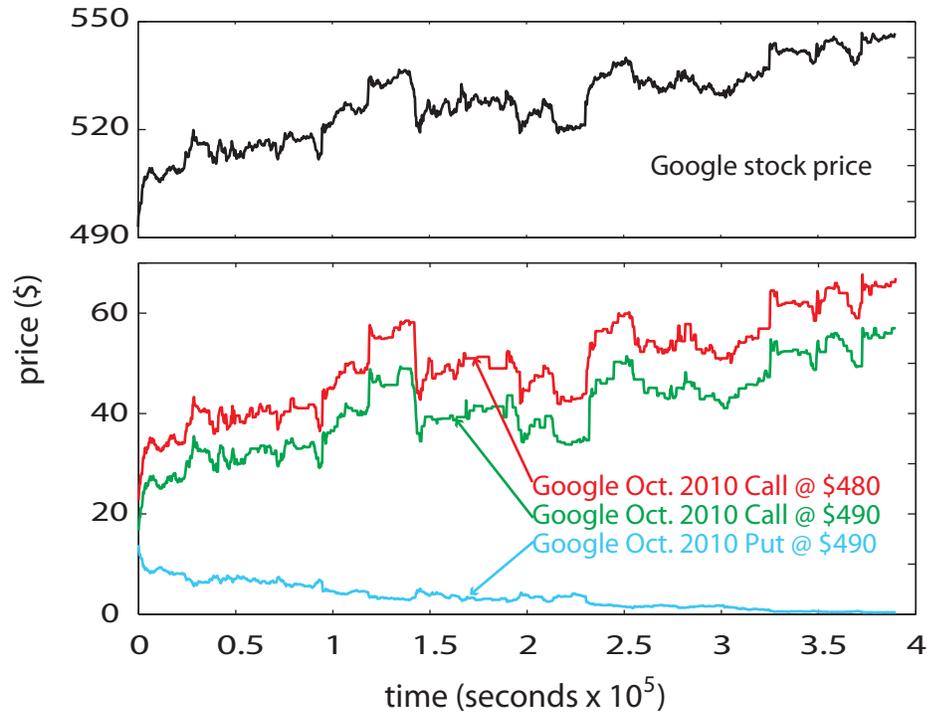


Figure A5.1. One month of raw data. The lower panel shows the changes in price of three different options contracts for Google that expired at the end of the trading period shown. The top line is the price of the call options contract with a strike price of \$480, the middle line is the price of the call options contract with a strike price of \$490, and the bottom line is the price of the put options contract, which had a strike price of \$490. The smaller panel at the top shows the price of Google stock plotted at the same time resolution and on the same time axis. For the month plotted, the stock price increases from about \$490 to about \$550, an increase of about 12%. Over the same period, the \$490 call option shown increases in price from about \$18 to about \$58, an increase of 222%. The put option loses essentially 100% of its value.

then repeating the process after shifting the input and output windows one time step forward to $t + 1$ and $t + I + 1$, respectively.

Clearly, the values chosen for I , O , and *gain threshold* will have a significant impact on the algorithm. It was assumed that the size of the output window O should be as large as possible within reason, because the longer any purchase is held, the greater opportunity there is for profit. For all experiments described below, O was set to 3×10^4 seconds = 8.3 hours, which is somewhat longer than one trading day and a

reasonably long period to hold an options contract in the extremely fast-paced options trading market. However, the input window size I could conceivably be larger to encompass a greater amount of historical information, smaller to represent information about more recent events only, or some type of average. Similarly, the ideal value of *gain threshold* was unclear. The value of *gain threshold* could be high to emphasize only the trades with the greatest profits, but these trades might also have the highest risk of large losses as well. In the absence of any a priori reason for choosing specific values for I and *gain threshold*, extensive cross validation experiments were used to choose good values (Figure A5.2).

Feature Construction

As mentioned above, the raw time series data from input windows were not used directly. 24 features were calculated for each input window. These features can be divided into two main categories. The first category of features is comprised of relatively simple, traditional econometric descriptors hand-selected to capture the two most relevant aspects of options time series data: variability and directional tendencies in price. There were nine such features, all designed to capture one or both of those properties: 1) standard deviation, 2) moving average, 3) coefficient of deviation, 4) disparity, 5) normalized momentum (using the midpoint and end of the window), 6) price oscillator value (at the end of the window), 7) A/D oscillator value (at the end of the window), 8) relative strength index, and 9) slope of a first-order polynomial fit. See McMillan 2001 and Kim 2003 for details on these features.

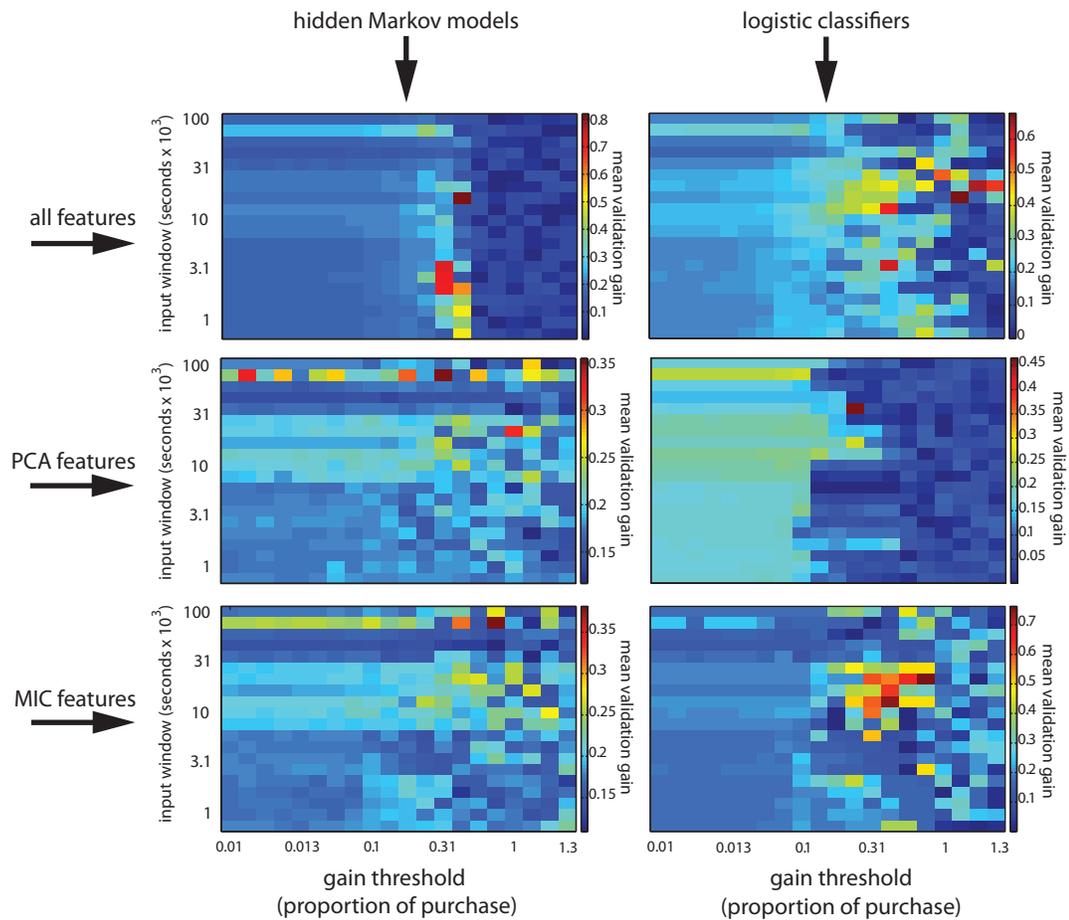


Figure A5.2. Cross validation experiments. Two important parameters for which no clear good values existed a priori were input window size and *gain threshold*, as described in the text. Six different classifiers were trained and applied to validation data for 400 different combinations of input window size and *gain threshold*. The mean gain generated by applying the trained classifier to the validation set and executing every “good buy” is represented with the color gradation. The color scale is unique for each of the six panels and is shown on the panel’s right side.

Features in the second category were calculated based on wavelet descriptions of the time series data. A relatively recent development in signal processing theory involves time series described using a small number of component terms called wavelets, much like the coefficients in a Fourier series (Mallat 1989). Unlike Fourier

analysis however, wavelet analysis takes into account local variations in the time series. This makes wavelet coefficients particularly useful for the analysis of nonstationary time series such as financial time series (Ramsey 2002). Although there are many ways to use wavelet coefficients as part of a machine learning model, here the information in wavelet coefficients was summarized by taking the sum of the squared coefficient values separately for each level of the wavelet transform. This N -valued vector is called the wavelet power spectrum, where N is the number of wavelet decomposition levels (Subramani et al. 2006). 15 features were obtained in this way. These correspond to the first five values of the wavelet power spectrum that resulted from a continuous wavelet transform of the input window for each of three different wavelet types. The three wavelet types were Harr wavelets, Morlet wavelets, and Daubechies wavelets, all of which have been described as particularly appropriate for financial time series analysis in the literature (Ramsey 2002).

The 24 total features described above are likely to contain a large amount of redundant information, and this can harm the performance of a classifier by encouraging overfitting and limiting its ability to generalize to new data. Here, this issue was addressed by using principal components analysis (PCA) as a dimensionality reduction technique and the mutual information criterion (MIC) as a feature selection technique. In the cross validation results shown in Figure A5.2, experiments were performed for: 1) classifiers trained using all 24 features, 2) classifiers trained using the first two PCA features, and 3) classifiers trained using the 2 highest scoring features by the mutual information criterion. In the case of PCA features, the first two constructed features always explained more than 90% of the

variance in the data, so choosing only two seemed justified. In the case of MIC features, which are not transformed in any way as with PCA-constructed features, the aggregate score for a feature is a fairly complicated calculation (Peng et al. 2005). However, selecting the two highest scoring features from the original 24 features seems reasonable as the two highest scoring features always explained more than 50% of the variance in the data and usually explained more than 95%. Interestingly, the two highest scoring MIC features were always among the traditional economic descriptors, suggesting that wavelet analysis may be of limited value for this particular machine learning application.

Classifier Models

A high degree of correlation exists between many of the time series in the data set. For example, two different time series for two different call option contracts for the same stock are expected to have virtually the same dynamics over the same time interval, aside from a scaling factor (Figure A5.1). To address this issue, it was necessary to carefully partition the data set into separate training, validation, and test data sets in order to prevent the inferred classifiers from having an unfair advantage. The approach here was simply to use the first month's data set for training data, the first two weeks of the second month's data set for validation and parameter tuning, and the remaining portion of the second month's data set for testing.

The features constructed as described above from the training data were used as the inputs to train two types of models: logistic classifiers and hidden Markov models (HMMs). The logistic classifiers were simply the result of performing logistic

regression using input features as the x -value and output window class as the binary y -value. The HMMs were the result of treating the set of input features as the observation for one input window and treating the output window class as the corresponding state. The common simplifying assumption was made that each input feature was generated independently, and so the overall HMM was actually several independent HMMs in parallel, one HMM for each feature (Dietterich 2002). The overall probability of observing the features conditional on the output window class was then simply the product of observing each feature conditional on the output window class. Because both states and observations were known, HMMs were estimated directly from the training data. For predictions over validation and test data, the most likely output window class was found using the Viterbi algorithm (Rabiner 1989) and compared with the true output window class. Along with the other cross validation experiments, results comparing logistic classifiers and HMMs are shown in Figure A5.2.

Results

The cross validation results shown in Figure A5.2 reflect the great importance of specifying the training and model parameters correctly. The results here are expressed in terms of mean validation gain, where gain is defined as above. The gain values were obtained from applying the trained classifiers to every input/output window pair in the validation set and then averaging. A gain value from one particular input/output window pair in the validation data set was considered as part of

the average only when the classifier returned a value of 1 = “good buy” for that input/output window pair. This was a relatively simple method of finding good values for *gain threshold* and input window size. For example, it did not take into account false positive rates, different penalties and rewards for different types of decisions, etc. Even still, the results are difficult to interpret. It appears that higher values of *gain threshold* and larger input window sizes are better, but many exceptions to this general trend are apparent.

The two model types and three feature selection methods define six distinct classifiers: all features with HMM, all features with logistic regression, PCA features with HMM, PCA features with logistic regression, MIC features with HMM, and MIC features with logistic regression. As a baseline control, a “naive” classifier was also trained that used only the slope econometric feature described above with both HMM and logistic regression. Based on the above results giving a rough indication of good values for input window size and *gain threshold*, all eight classifiers were trained with *gain threshold* and input window sizes as shown in Table A5.1.

The final results of applying these eight trained classifiers to the never-before-seen test data are shown in Table A5.2. Both HMM and logistic classifiers using all features significantly outperformed all the other methods and were not significantly different from each other when characterized by the distribution of gains resulting from all combined trades. Interestingly, all features seemed to be necessary to capture the differences between good buys and poor buys. Feature selection and dimensionality reduction appeared to harm classifier performance.

Table A5.1. Parameter values used to train the final classifiers.

Classifier type	<i>gain threshold</i> (proportion of purchase)	Input window size (seconds x 10³)
all features, HMM	0.3	2.5
all features, logistic	0.3	20
PCA, HMM	0.3	20
PCA, logistic	0.15	20
MIC, HMM	0.4	20
MIC, logistic	0.4	20
slope, HMM	0.4	20
slope, logistic	0.4	20

Table A5.2. Performance of classifiers on test data measured using gain. The two classifiers using all features significantly outperformed the other six classifiers at the $p < 0.05$ level as measured by unpaired t -tests. In addition, the two classifiers using all features did not significantly differ from each other in performance. The Bonferroni correction for multiple tests was used to adjust p -values. Trades were only made when a “good buy” signal was generated, but the data for “poor buys” are included for the sake of completeness.

Classifier type	Mean/standard deviation of gain for “good buys”	Number of “good buy” signals	Mean/standard deviation of gain for “poor buys”	Number of “poor buy” signals
all features, HMM	0.6153/0.9467	5680	0.3558/0.9433	420920
all features, logistic	0.7289/1.6591	13880	0.3478/0.7299	412720
PCA, HMM	0.3628/0.8058	96360	0.3728/0.8311	330240
PCA, logistic	0.3145/0.8405	487	0.3499/0.8830	426113
MIC, HMM	0.3849/1.2527	48560	0.3628/0.7397	378040
MIC, logistic	0.3123/0.6786	522	0.3085/1.0115	426078
slope, HMM	0.3729/0.7123	1218	0.3661/0.8597	425382
slope, logistic	0.3728/0.7087	10733	0.3676/0.8394	415867

If the trained classifiers are taken to represent options trading strategies in the sense that they could be used to guide real-life purchases, then mean gain by itself is likely to be an unreliable indicator of strategy quality. For example, it does not take into account the frequency of trades, variability in trade results, different distributions

for profitable trades vs. losing trades, and many other factors that are likely to make or break a real-life trading strategy. Although accurately modeling a trading system with variables such as market liquidity, commissions, and missing or time-delayed data is beyond the scope of this work, a simple all-in-one measure that captures many of these real-life considerations is the expectancy score (ES). ES can be defined as:

$$ES = \frac{(ag)(pg) + (ap)(pp)}{ap} \quad (A5.1)$$

where ag is the average gain of “good” trades, pg is the proportion of trades that are “good,” ap is the average gain of “poor” trades, and pp is the proportion of trades that are “poor.” For present purposes, “good” trades were defined as those made when a “good buy” signal was generated and when the gain was greater than 0.3, while “poor” trades were defined as those made when a “good buy” signal was generated and the gain was less than or equal to 0.3. Note that trades were never made when a “poor buy” signal was generated. Table A5.3 shows the ES values obtained by all eight classifiers, indicating that classifiers using all features outperform the other six classifiers in terms of the ES metric as well as the simple gain.

Discussion

This study represents a first attempt at applying standard machine learning methods to the automation of options trading. The main conclusion was that HMM and logistic classifiers can perform significantly better than chance. However, there

Table A5.3. Performance of classifiers on test data measured using the expectancy score. Although there is no significance test for the expectancy score per se, the results shown are consistent with the results in Table A5.2 indicating that the classifiers using all features outperformed the other six classifiers.

Classifier type	Expectancy score
all features, HMM	8.9243
all features, logistic	10.9900
PCA, HMM	5.0721
PCA, logistic	2.5873
MIC, HMM	5.6525
MIC, logistic	4.3367
slope, HMM	5.2354
slope, logistic	6.1921

are many caveats. Even if these preliminary results hold up after additional data are collected, the implications of such a classifier are not clear for the development of realistic trading strategies. It is possible that only very large numbers of purchases can result in significant net profits given the high variability of profits on a per-purchase basis. This would require a large and perhaps prohibitively large up-front investment. Although in this initial work, the space of trading strategies was restricted to simple purchases of individual put and call options contracts, classifiers could also be constructed to predict the results of more complex types of options trades, such as simultaneous purchases of identical calls and puts. This is an interesting avenue of research for future experiments.

A few other issues might also limit the practical use of any classifiers developed in this work. One of these is the problem of liquidity. It is possible for the purchase of an options contract to occur successfully even if the contract cannot be sold later because a buyer willing to pay the desired price cannot be found. This would be a symptom of poor liquidity. In the above results, it was assumed that a

contract could be purchased and sold at any time. To make this assumption as realistic as possible, only options for large, high-profile companies with stock traded on major exchanges such as NYSE and NASDAQ were used. Another practical issue relates to trade commissions. It was assumed here that the per-trade fee of most self-service brokers is likely to be negligible compared to the price of even one inexpensive options contract. However, the validity of this assumption needs to be investigated.

Finally, only relatively simple machine learning techniques were used. It is likely that more sophisticated machine learning algorithms can fruitfully be brought to bear in this application. In particular, advanced probabilistic graphical models such as conditional random fields are promising.

APPENDIX 6

RANDOM FUNCTIONS WITH MULTIPLE TIME SCALES

Table A6.1. Ten example randomly generated target functions with multiple time scales for each of six different complexities. Complexity is defined as the number of operators and operands in a function.

<p>Complexity 9</p> <p>$f(t) = \sin(0.6555*t) + \sin(222.8254/t)$</p> <p>$f(t) = \sin(t) + \sin(\cos(\exp(34.6981*t)))$</p> <p>$f(t) = \sin(\cos(\log(t))) + \sin(146.6631*t)$</p> <p>$f(t) = \sin(t) + \sin(-0.0709/(t-0.0573))$</p> <p>$f(t) = \sin(t-0.1663) + \sin(100.0000*t)$</p> <p>$f(t) = \sin(t) + \sin(28769.4797*t*t)$</p> <p>$f(t) = \sin(\sin(t)) + \sin(\cos(100.0000*t))$</p> <p>$f(t) = \sin(\exp(\cos(t))) + \sin(100.0000*t)$</p> <p>$f(t) = \sin(1.7112*t) + \sin(240.7999*t)$</p> <p>$f(t) = \sin(t) + \sin(422.2429*t-2.5138)$</p>
<p>Complexity 11</p> <p>$f(t) = \sin(\sin(\cos(t-1.8002))) + \sin(100.0000*t)$</p> <p>$f(t) = \sin(t*t) + \sin(100.0000*t-0.1862)$</p> <p>$f(t) = \sin(0.0612-0.7935*t) + \sin(100.0000*t)$</p> <p>$f(t) = \sin(\cos(2.9292*\sin(t))) + \sin(100.0000*t)$</p> <p>$f(t) = \sin(28.0159-3.7821*t) + \sin(632.5783/t)$</p> <p>$f(t) = \sin(\exp(\sin(t+0.0680))) + \sin(200.0000*t)$</p> <p>$f(t) = \sin(\cos(\sin(\sin(\cos(t)))))) + \sin(100.0000*t)$</p> <p>$f(t) = \sin(\sin(t)) + \sin(0.2533*\exp(100.0000*t))$</p> <p>$f(t) = \sin(\exp(\cos(t*t))) + \sin(100.0000*t)$</p> <p>$f(t) = \sin(t) + \sin(-8.1382*t-63.6163/t)$</p>
<p>Complexity 13</p> <p>$f(t) = \sin(\log(\exp(t+\exp(t)))) + \sin(\exp(100.0000*t))$</p> <p>$f(t) = \sin(t/(\log(t)-4.3383)) + \sin(\exp(100.0000*t))$</p> <p>$f(t) = \sin(t) + \sin(\sin(100.0000*t+\cos(288.8896*t)))$</p> <p>$f(t) = \sin(-0.2259*t-0.7042) + \sin(100.0000*t-8.1376)$</p> <p>$f(t) = \sin(-0.9912*t-6.2775) + \sin(-5172.9455*t*t)$</p> <p>$f(t) = \sin(\cos(t*t)) + \sin(2625.4851/\exp(875.6974*t))$</p> <p>$f(t) = \sin(-2.7653*t*(t-4.4928)) + \sin(236.0264/t)$</p> <p>$f(t) = \sin(0.0003/(t-0.0037)-0.8427) + \sin(100.0000*t)$</p> <p>$f(t) = \sin(t+\exp(t)) + \sin(\cos(100.0000*t-3.5390))$</p> <p>$f(t) = \sin(\sin(t)) + \sin((100.0000*t)/\sin(100.0000*t))$</p>

Complexity 15

$f(t) = \sin(\sin(t+\log(\sin(t))-3.4258))+\sin(\cos(200.0000*t))$
 $f(t) = \sin(\exp(1.0000/\exp(3.8702*t)))+\sin(100.0000*t+1.5436)$
 $f(t) = \sin(t)+\sin((0.0066*\cos(\cos(100.0000*t)))/(t-0.0137))$
 $f(t) = \sin(\sin(t)-1.0003*\sin(8.1179*t))+\sin(200.0000*t)$
 $f(t) = \sin(\cos(1.3205*t))+\sin(200.0000*t*\exp(100.0000*t))$
 $f(t) = \sin(\log((3.3864*\sin(t))/t))+\sin(100.0000*t-1.7107)$
 $f(t) = \sin(0.0438-1.5301*t*t)+\sin(-100.0000*t-5.4138)$
 $f(t) = \sin(5.5732*t-0.5952)+\sin(\exp(\exp(\sin(\sin(100.0000*t))))))$
 $f(t) = \sin(-2.0474*\exp(t))+\sin((100.0000*\sin(100.0000*t))/t)$
 $f(t) = \sin(1.0000/\exp(0.1050/t))+\sin(\exp(100.0000*t)-6.9114)$

Complexity 17

$f(t) = \sin(\cos(\cos(\log(t))*(2.0000*t-0.9487)))+\sin(\cos(100.0000*t))$
 $f(t) = \sin(1.0000/\exp(1.6299*t)+0.0658)+\sin(\sin(100.0000*t-2.2036))$
 $f(t) = \sin(t*t*(\log(t)-1.5784))+\sin(\cos(774.9838*t)+0.3339)$
 $f(t) = \sin(-0.4758*\sin(t+2.9917))+\sin(\log(\exp(\cos(100.0000*t))))+0.1970$
 $f(t) = \sin(0.0809*t)+\sin(7161.0448*t*t-100.0000*t-4.8359)$
 $f(t) = \sin(\sin(-4.7230*\cos(t)-4.7230*t*t))+\sin(100.0000*t)$
 $f(t) = \sin(\cos(\exp(t-1.8587))*(t-0.5404))+\sin(100.0000*t+9.2103)$
 $f(t) = \sin((0.2498*t+1.8442)*(t-0.0878)+0.5172)+\sin(-788.0345*t)$
 $f(t) = \sin(\exp(\exp(0.4202-0.3128*t))*(t+0.1041))+\sin(100.0000*t)$
 $f(t) = \sin(\exp(t))+\sin(-4.2440*\exp(200.0000*t)*(100.0000*t-0.1345))$

Complexity 19

$f(t) = \sin(6.7516*t-2.6332*\sin(t))+\sin(\exp(100.0000*t)-114.2168/t)$
 $f(t) = \sin(1.0000/\exp(6.7453/(t*t)))+\sin(100.0000*t+\exp(100.0000*t))$
 $f(t) = \sin(0.9739*t)+\sin(\sin(-100.0000*t-3.0485)-3.0100*\sin(502.9871/t))$
 $f(t) = \sin(\exp(t)-0.3334*t-0.9977*\sin(t-2.9782))+\sin(80.9528/t)$
 $f(t) = \sin(\cos(0.8718*t*t)-0.0458)+\sin(200.0000*t*\cos(100.0000*t))$
 $f(t) = \sin(\cos(t)+\exp(t))+\sin(-10000.0000*t*t+100.0000*t-0.3559)$
 $f(t) = \sin(\sin(t))+\sin(1.0000/\exp(2.9279*\sin(100.0000*t))+\sin(100.0000*t))$
 $f(t) = \sin(t*(\log(t)+1.1001))+\sin(\cos(100.0000*t)/(100.0000*t-1.6153))$
 $f(t) = \sin(0.1601*t+0.8024)+\sin(-26432.9000*t*t+514.5355*t+1.0321)$
 $f(t) = \sin(\cos(\sin(t))-1.0025*t-1.0025*\cos(t))+\sin(\cos(100.0000*t))$

APPENDIX 7

COMPARISON OF EVOLUTIONARY ALGORITHMS FOR OPTIMIZATION AND SYMBOLIC REGRESSION

Introduction

Evolutionary algorithms are frequently used to solve optimization and machine learning problems. However, reports in the literature rarely perform rigorous, objective comparisons between different methods. As a result, it can be difficult to draw conclusions about the relative performance of different approaches, especially when constraints demand that performance per unit of computational effort be maximized. In this appendix, several different evolutionary algorithms are compared for two different benchmark problems: the traveling salesman optimization problem (Press et al. 2007), and the symbolic regression machine learning problem (Koza 1992). For these experiments, particular attention was paid to evaluating numerous different types of algorithms and comparing them in as objective a manner as possible in order to gain insight into the strengths and weaknesses of the different approaches.

Genetic Algorithms for the Traveling Salesman Problem

The traveling salesman problem is a classic benchmark in the field of numerical optimization. The statement of the problem is simple:

Given N points in a Euclidean space, what is the shortest path that visits each point exactly once?

Despite its simplicity, this problem is of great importance as many real world problems can be reduced to what is essentially a traveling salesman problem or a closely related problem. No efficient (polynomial time) algorithm for the general traveling salesman problem is known, and so many general-purpose optimization algorithms have been applied to the problem with varying degrees of success (Press et al. 2007). The genetic algorithm (Holland 1975, Mitchell 1996) is a general-purpose optimization algorithm that has met with success in numerous problem domains. Here several variants of the genetic algorithm are applied to different variations on the traveling salesman problem with the goal of objectively evaluating different approaches.

The traveling salesman test problems for these experiments were of four types applied to each of 3 different sets of cities, for a total of 12 distinct problems. In problem type 1 (“shortest path”), the shortest path that visits each city once was the search objective. In problem type 2 (“longest path”), the longest path that visits each city once was the search objective. In problem type 3, (“path with least horizontal travel”), the path that involved the smallest amount of horizontal travel while still visiting each city once was the search objective. In problem type 4, (“path closest to home”), the path that visits each city once but has the smallest sum over path segments of the mean distance from home was the search objective. For problem type 4, the mean distance from home of each path segment was calculated as the length of the segment multiplied by the distance of the midpoint of the segment from the home

point (0.25, 0.25). This value was calculated for each path segment and added to obtain the final fitness value. The 3 different sets of cities are referred to as “Test Problem 1,” “Test Problem 2,” and “Test Problem 3” in the results figures below.

The genetic algorithm for solving the traveling salesman problem used here implements a priority list encoding, in which a solution is represented as a fixed list of points in the Euclidean plane. These points below will be referred to as “cities” to be consistent with literature on the traveling salesman problem. Each gene in each chromosome is a real number within the range (0.0, 1.0) specifying the order in which the city is visited in the traveling salesman’s path. The order in which the cities are arranged on the chromosome is determined by starting arbitrarily with the first city given in a list of city locations and then adding the next closest city, then the next closest city, and so on. The cities were ordered in this way as a simple means of potentially improving linkage (Mitchell 1996). Pilot experiments suggested that relatively large population sizes of 1000 would work well and result in adequate diversity for finding good solutions.

Each generation, two parents were selected at random for reproduction. Random selection of parents in this way appeared to prevent premature convergence of the population and allowed adequate exploration of the solution space. Once two parents were selected, two offspring were made by crossover and mutation. Either two-point crossover or uniform crossover was performed. In the case of uniform crossover, each gene had a 25% chance of being inherited from one parent and a 75% chance of being inherited from the other parent. This type of uniform crossover eliminates all location biases while also preventing crossover from becoming a

completely random operation, which would be the case if each gene had a 50% chance of being inherited from one or the other parent. Regardless of the crossover method used, children were then mutated with a probability of 2.5% per gene. When a gene was mutated, a new real number for the gene was found by adding a randomly generated number $\sim N(0, 0.01)$. This ensured that mutation maintained locality and made only small changes to genes.

After creating offspring using the above operations, the fitnesses of the offspring were evaluated. The population in the next generation was then created by culling the combined population of parents and offspring using one of two methods for survival selection. The first method of survival selection was tournament selection with replacement and tournaments of size 2. This was implemented with a scheme whereby two random members of the combined population were selected. The one with the lower fitness was discarded and the one with higher fitness was returned to the pool. This continued until the population size was reduced back to 1000. The second method of survival selection was truncation selection, which was expected to result in much stronger selection pressure than tournament selection. Truncation selection was implemented by simply ranking all members of the combined parent and offspring population and then allowing only the better performing half of the population to survive.

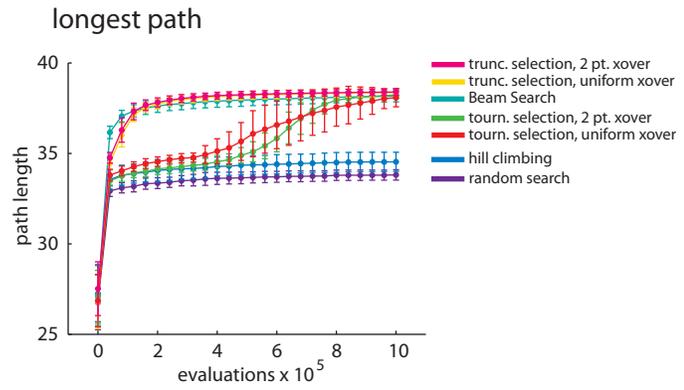
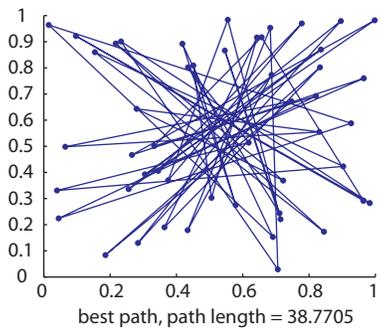
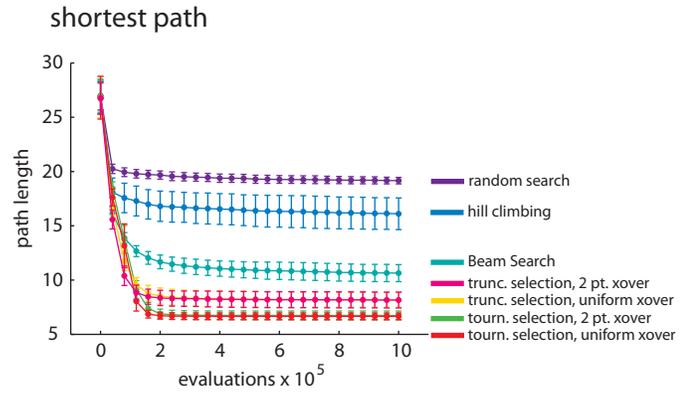
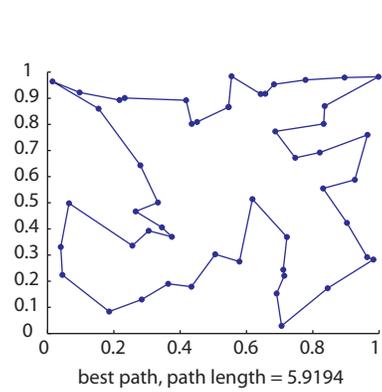
Four variations of the genetic algorithm were studied, which corresponded to each combination of crossover type (two-point and uniform) and selection type (tournament and truncation). These variations of the genetic algorithm were compared with random search, hill climbing, and Beam Search. In general, the results show that

the genetic algorithm with tournament selection performed the best, with little effect of crossover type. However, crossover was clearly important to the genetic algorithm's success as shown by the generally inferior performance of Beam Search, which is essentially equivalent to the genetic algorithm without crossover and with truncation survival selection. The hill climbing algorithm in turn was essentially equivalent to Beam Search without the benefits of a population-based approach. All seven algorithms were run for a total of 1,000,000 fitness evaluations, with 50 independent trials performed for each. Performance curves in Figure A7.1, Figure A7.2, and Figure A7.3 show the mean best solution found vs. number of fitness evaluations, where error bars represent \pm standard deviation of the best solution found.

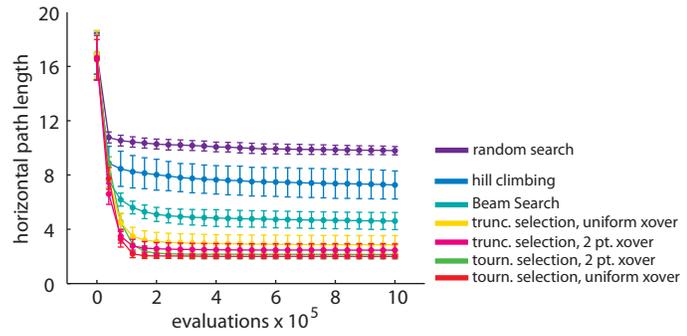
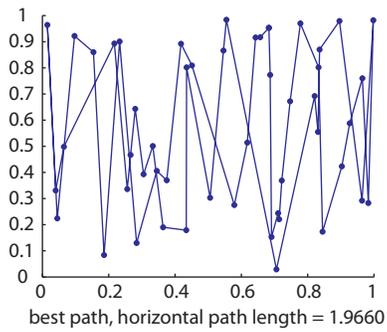
Genetic Programming Algorithms for the Symbolic Regression Problem

Symbolic regression is the task of identifying both the form and parameters of an algebraic model that fits a given data set. This contrasts with ordinary linear and nonlinear regression, in which the form of the algebraic model is fixed in advance. As noted above, many studies have proposed various evolutionary approaches to the symbolic regression problem, yet rarely are rigorous, objective comparisons made between different methods. Here, several different evolutionary and non-evolutionary

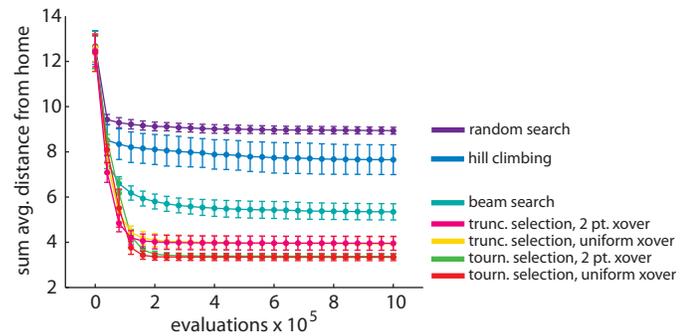
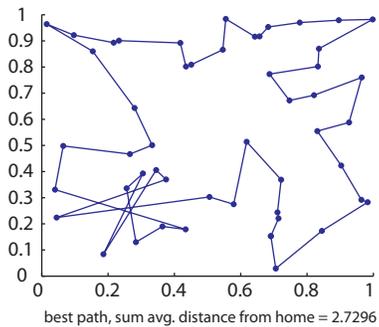
Figure A7.1. Results for traveling salesman Test Problem 1 (next page). The first column shows the locations of the cities in Test Problem 1 overlaid with the best path found for the indicated problem type. The second column shows performance of the different algorithms on the problem as a function of the number of fitness evaluations performed.



path with least horizontal travel



path closest to home



algorithms are compared on a series of symbolic regression test problems with an emphasis on evaluating algorithm performance per unit of computational effort.

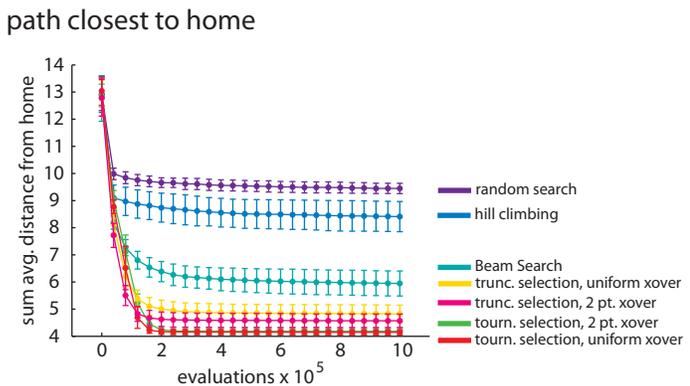
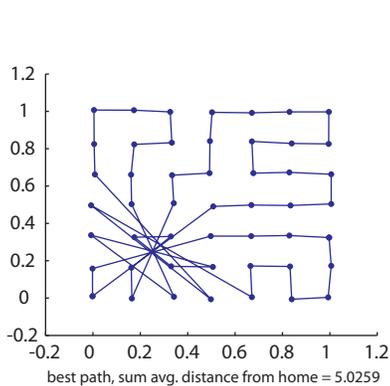
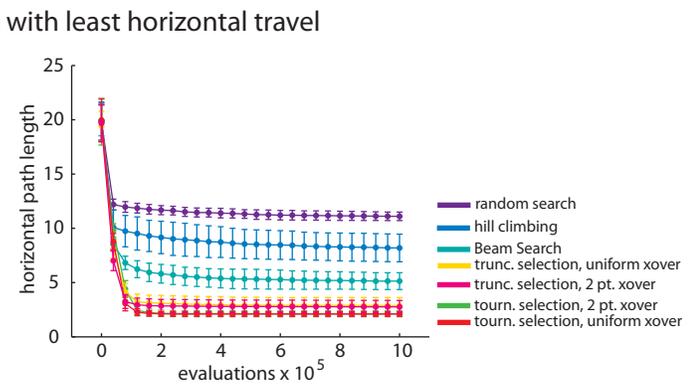
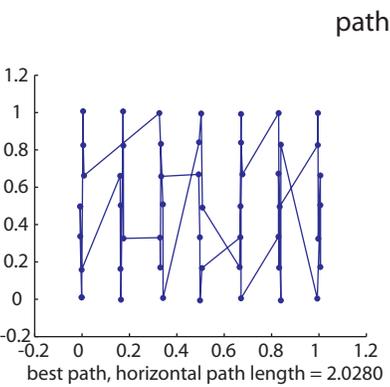
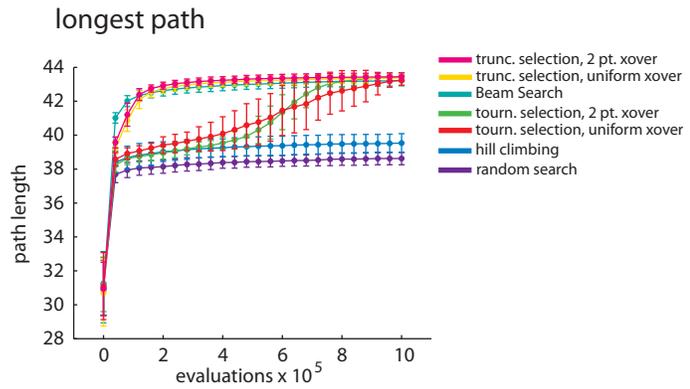
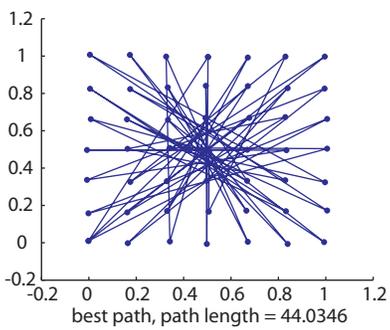
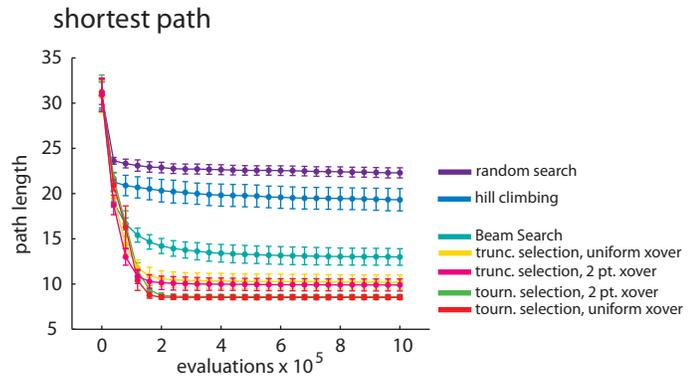
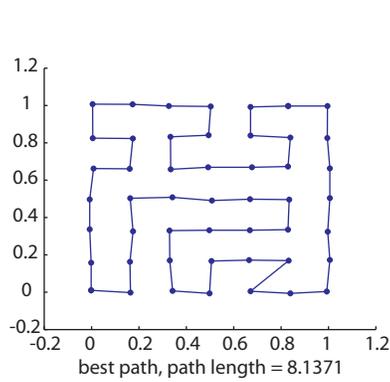
Six symbolic regression test problems were used: a linear data set (Test Problem 4) both with and without noise, a nonlinear data set (Test Problem 5) both with and without noise, and an elliptic data set (Test Problem 6) both with and without noise. The test data for each of these six problems are plotted below in Figure A7.4, Figure A7.5, and Figure A7.6, for Test Problem 4, Test Problem 5, and Test Problem 6, respectively.

All algorithms for solving the symbolic regression problem in these experiments use a linear encoding, in which a solution is represented as a list of operators and operands. Each gene in the list is similar to a machine language instruction, and registers are used to store intermediate values as the list is evaluated from first instruction to last. For example, an individual with a genome of length 5 corresponding to the expression $f(x, y) = x + y - 1.2345$ could be represented as

id	type	value 1	value 2
id 1	variable	x	
id 2	variable	y	
id 3	add	id 1	id 2
id 4	constant	1.2345	
id 5	subtract	id 3	id 4

Such a list of machine language instructions is equivalent to a directed acyclic graph representation, which has been shown to have advantages over the more traditional

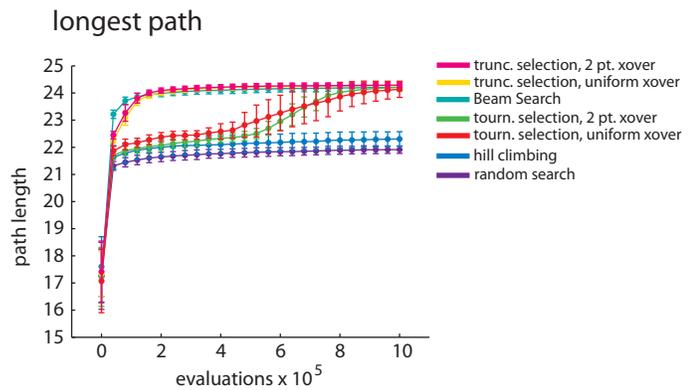
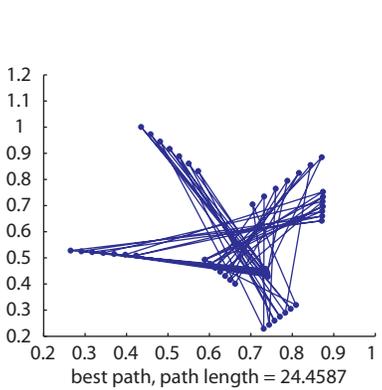
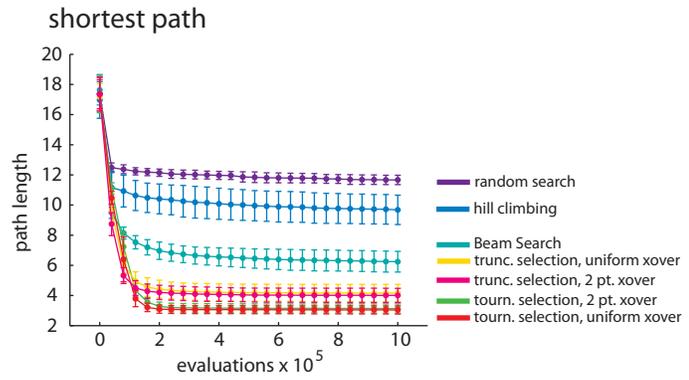
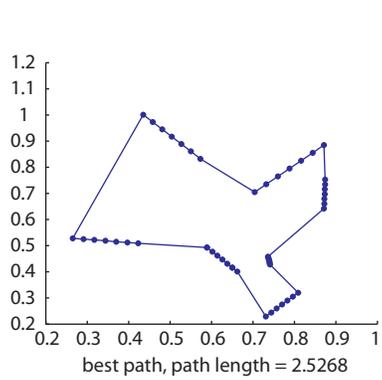
Figure A7.2. Results for traveling salesman Test Problem 2 (next page). Rows and columns are as described in Figure A7.1.



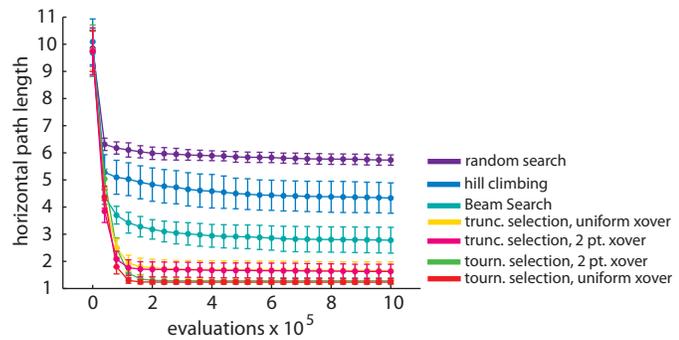
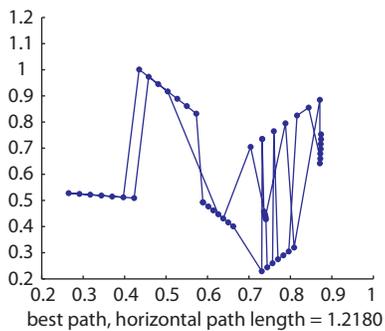
tree-based representation for symbolic regression applications of genetic programming (Schmidt and Lipson 2007). Seven different algorithms using this representation were used to search through the space of algebraic expressions: random search, hill climbing, Beam Search, Non-Memetic Deterministic Crowding GP, Memetic Deterministic Crowding GP, Non-Memetic Age-Fitness GP, and Memetic Age-Fitness GP.

For random search, random solutions were constructed containing between 2 and 20 instructions and the best solution found so far was stored each time. For hill climbing, a single random solution was constructed and this individual was repeatedly mutated. If the mutated individual had better fitness than the “parent,” it replaced the parent. During mutation, each instruction in a solution was replaced with a randomly chosen instruction with a probability of 2.5%. In addition, a new randomly chosen instruction was added at a randomly chosen location in the genome with a probability of 2.5% and a randomly chosen instruction was deleted with a probability of 2.5%. For Beam Search, reproduction selection was performed by selecting each individual from a population of 100 individuals as a parent. A child was then formed by copying the parent and mutating the copy as described for the hill climbing algorithm. This continued until the population size was doubled. Simple truncation selection was then used to create the next generation.

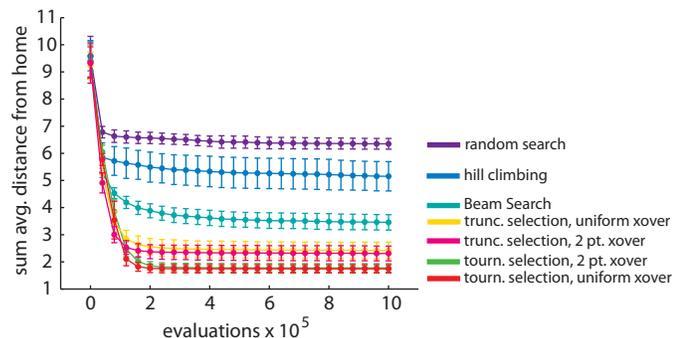
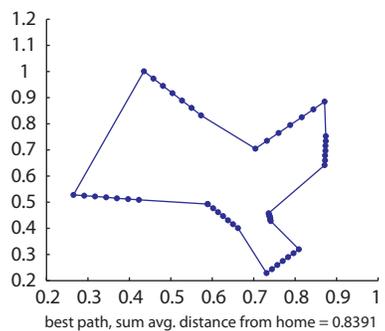
Figure A7.3. Results for traveling salesman Test Problem 3 (next page). Rows and columns are as described in Figure A7.1.



path with least horizontal travel



path closest to home



For Deterministic Crowding GP, reproduction selection was performed by selecting pairs of parents at random, although each member of the population of 100 individuals was selected exactly once. After two parents were selected, two offspring were made by crossover and mutation. Two-point crossover of the parental instruction lists was used. The resulting children were then mutated as described for the hill climbing algorithm above. Each offspring then replaced the more similar of its two parents, but only if it had better fitness. Similarity was quantified by the sum of squared differences between the phenotypes of the two individuals under comparison. In the memetic version of the deterministic crowding algorithm, a single round of hill climbing followed each generation of the basic deterministic crowding algorithm. In this hill climbing step, a copy of each individual was made and each instruction in the copy corresponding to a real-valued constant was changed by adding a randomly generated number $\sim N(0, 0.1)$. If this altered individual had a higher fitness than the “parent,” it replaced the parent in the population.

For Age-Fitness GP (Schmidt and Lipson 2010a-b), reproduction selection was performed by selecting pairs of parents at random. After two parents were selected, two offspring were made by crossover and mutation as described for the Deterministic Crowding GP algorithm. Survival selection was performed by culling the combined population of parents and offspring using tournament selection with replacement and tournaments of size 2. This process was implemented with a Pareto tournament scheme in which two random members of the combined population were selected. If one of the pair had both lower fitness and higher age than the other, it was thrown out. The survivor was then returned to the pool. This continued until the population size

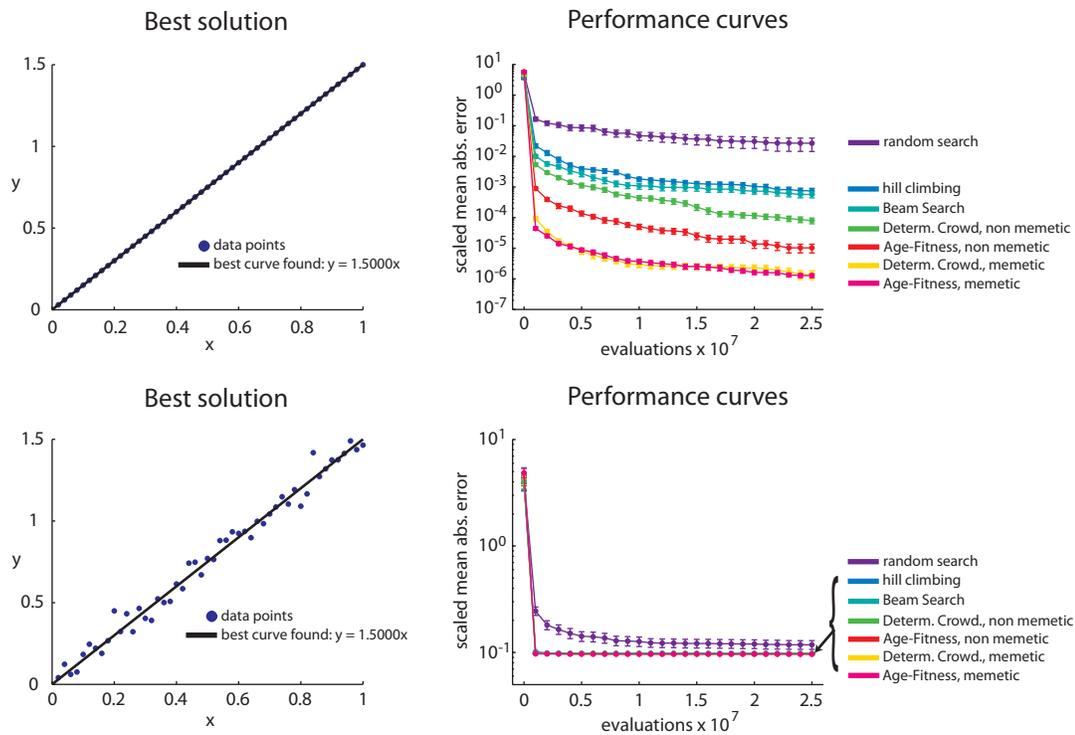


Figure A7.4. Results for symbolic regression Test Problem 4. The first column shows the training data points overlaid with a plot of the best curve found. The second column shows performance of the different algorithms on the problem as a function of the number of fitness evaluations performed. The first row is for the variant of the problem without noise and the second row is for the variant of the problem with noise.

was reduced back to 100. Age was defined as the number of generations in which an individual had been present in the population. Offspring age was set to the parent's age, and offspring inherited the age of the older parent in the case of crossover. One new, randomly generated individual with an age of 0 was added to the population each generation.

For all seven algorithms, fitness was calculated in one of two ways. In both cases, fitness was defined as an error value, which was to be minimized. For Test

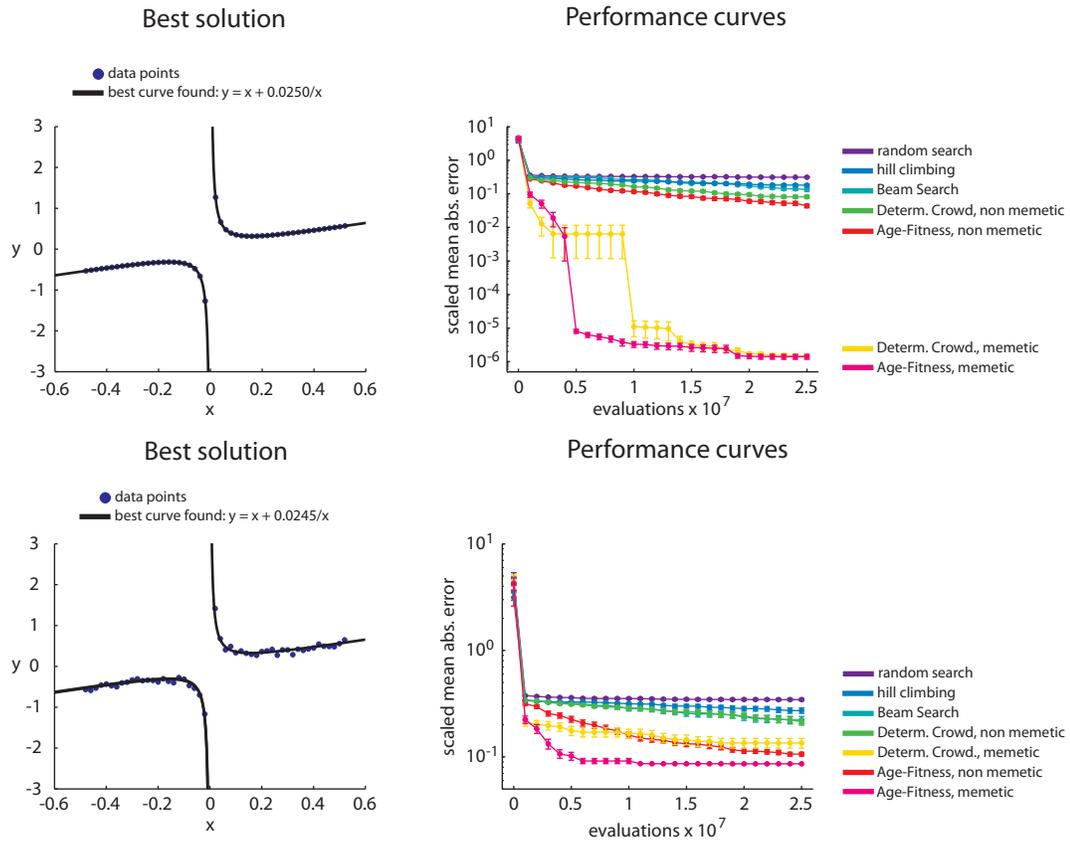


Figure A7.5. Results for symbolic regression Test Problem 5. Rows and columns are as described in Figure A7.4.

Problem 4 and Test Problem 5, the scaled mean absolute error of a candidate expression was used for searching for expressions of the form $y = f(x)$. The scaled mean absolute error was calculated as

$$\text{s.m.a.e.} = \frac{1}{n} \sum_{i=1}^n \frac{|y_i - f(x_i)|}{\sigma} \quad (\text{A7.1})$$

where n is the number of data points, y_i is the i th value of y in the data set, $f(x_i)$ is the i th predicted value of y obtained by evaluating the candidate expression f , and σ is

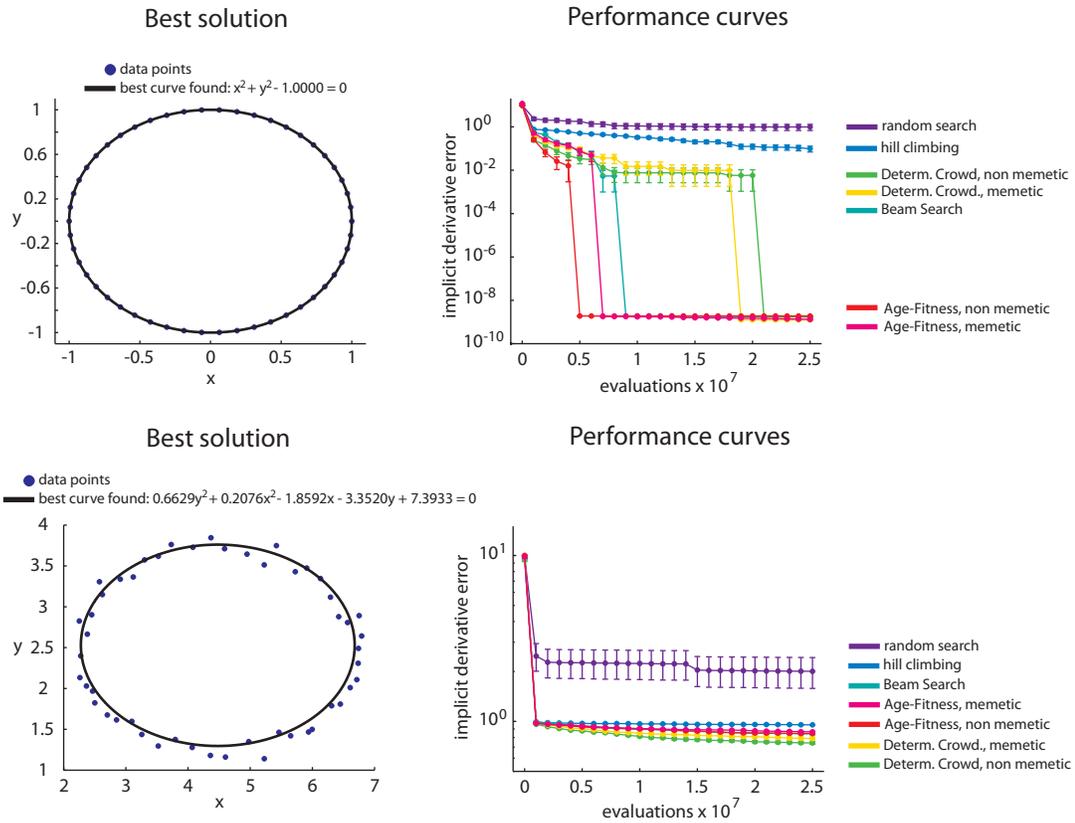


Figure A7.6. Results for symbolic regression Test Problem 6. Rows and columns are as described in Figure A7.4.

the standard deviation of the y values in the data set. For Test Problem 6, the implicit derivative error was used for searching for expressions of the form $f(x, y) = 0$. The implicit derivative error was calculated as

$$\text{i.d.e.} = \frac{1}{n} \sum_{i=1}^n \ln \left(1 + \left| \frac{\Delta x_i}{\Delta y_i} - \frac{\partial x_i}{\partial y_i} \right| \right) \quad (\text{A7.2})$$

where n is the number of data points, $\Delta x_i / \Delta y_i$ is the implicit derivative estimated from the data set at point i , and $\partial x_i / \partial y_i = (\partial f / \partial y_i) / (\partial f / \partial x_i)$ is the implicit derivative

estimated by evaluating the candidate expression f at point i . This implicit derivative error was added to the implicit derivative error for $\Delta y_i / \Delta x_i$, which was calculated in a similar way. For both the noise-free and noisy Test Problem 6 data sets, the data points were reordered with respect to a circular path to make estimating the implicit derivatives easier. In addition, for the noisy Test Problem 6 data set, the data were smoothed with a 5-point moving average before application of any algorithm.

All seven algorithms were run for a total of 25,000,000 fitness evaluations, with 50 independent trials performed for each. Performance curves in Figure A7.4, Figure A7.5, and Figure A7.6 show the error of the best solutions found vs. number of fitness evaluations, where error bars represent \pm standard error of the mean of the best solution found.

APPENDIX 8

GATING OF DENDRODENDRITIC INHIBITION OF MITRAL CELLS BY GRANULE CELLS IN THE MAMMALIAN OLFACTORY SYSTEM

Introduction

The primary goal of the work presented in this appendix is to examine the hypothesis recently proposed in Balu et al. 2007 and Gao et al. 2009 that cortical feedback to the mammalian olfactory bulb (MOB) can serve to “gate” dendrodendritic inhibition (DDI) of mitral (Mi) cells by granule cells (Gr). A model of the reciprocal Mi→Gr→Mi DDI circuit based on the work in Inoue and Strowbridge 2008 was made using the NEURON simulation environment (Hines and Carnevale 2001). Preliminary results show that important features of the circuit -including gating of DDI by cortical inputs- are reproduced by the model. Future work will include using the model to explore the functional implications of this “cortical gating hypothesis” for the production and maintenance of oscillatory activity in larger MOB circuits.

Methods

Following Inoue and Strowbridge 2008, the Mi cell model used here consisted of two compartments, one representing the cell body and another representing dendrites (Figure A8.1). Mi cells contained three types of membrane ion channels

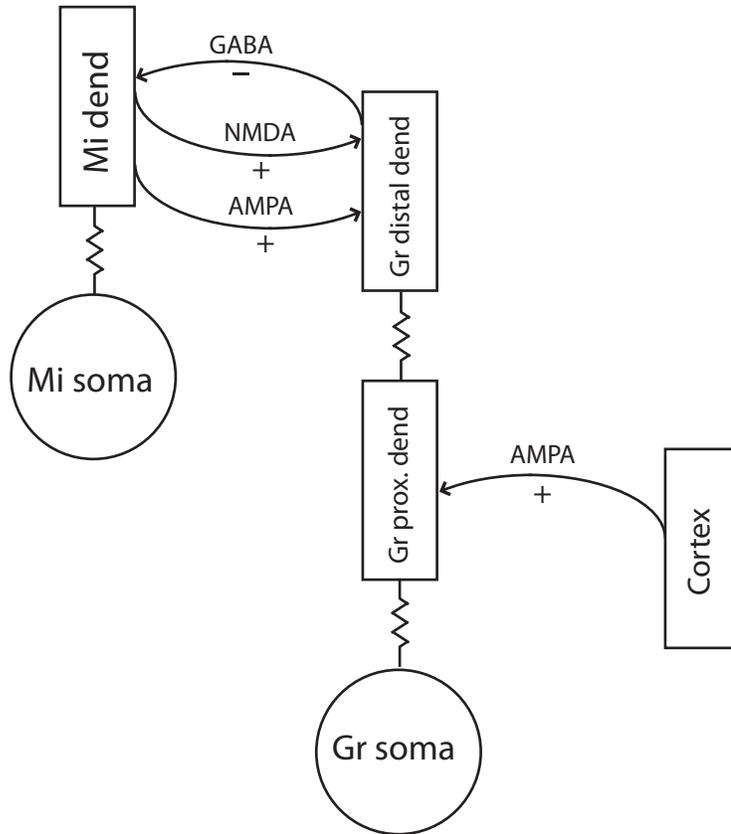


Figure A8.1. The Mi→Gr→Mi model.

evenly distributed across their entire surface area: a generic membrane leak conductance *pas*, a Hodgkin-Huxley fast sodium current *nafast*, and a delayed rectifier potassium current *kdr*. In contrast, the Gr cell model consisted of three compartments: one representing the cell body, one for proximal dendrites, and one for the distal dendritic arbor. In addition to the membrane ion channels used in the Mi cell model, the Gr cell model contained several additional potassium and calcium channels (see Inoue and Strowbridge 2008 for details). Parameters such as axial resistance between compartments as well as parameters describing the dynamics of all the membrane ion channels were taken directly from those published in Inoue and Strowbridge 2008.

The various synapses in the model were implemented with an eye toward reproducing the results in Balu et al. 2007 and Gao et al. 2009. Both AMPA and NMDA excitatory synapses connected the Mi cell dendrite compartment with one associated Gr cell distal dendrite compartment. The model and parameter values of the AMPA synapse were based on the AMPA synapse model in Inoue and Strowbridge 2008, whereas the NMDA synapse was based on the model in Destexhe et al. 1998, with parameter values chosen largely to give agreement with the empirical behavior of the mammalian Mi→Gr→Mi circuit observed in Balu et al. 2007. An inhibitory GABAergic synapse from the Gr cell distal dendrite compartment onto the Mi cell dendrite compartment completed the connections required for the reciprocal Mi→Gr→Mi circuit. As with the AMPA synapse above, the GABA synapse was based on the model in Inoue and Strowbridge 2008. Finally, an AMPA synapse from a dummy compartment onto the Gr cell proximal dendrite compartment was used to model excitatory input arising from feedback projections originating in the anterior piriform cortex (Balu et al. 2007). As with the NMDA synapse above, this proximal AMPA synapse had parameter values chosen largely to give agreement with Balu et al. 2007.

Results and Discussion

First, to verify that translation of the model from Inoue and Strowbridge 2008 into NEURON code was successful, the properties of the model Mi and Gr cells were tested in a manner similar to the tests performed empirically in that paper. For

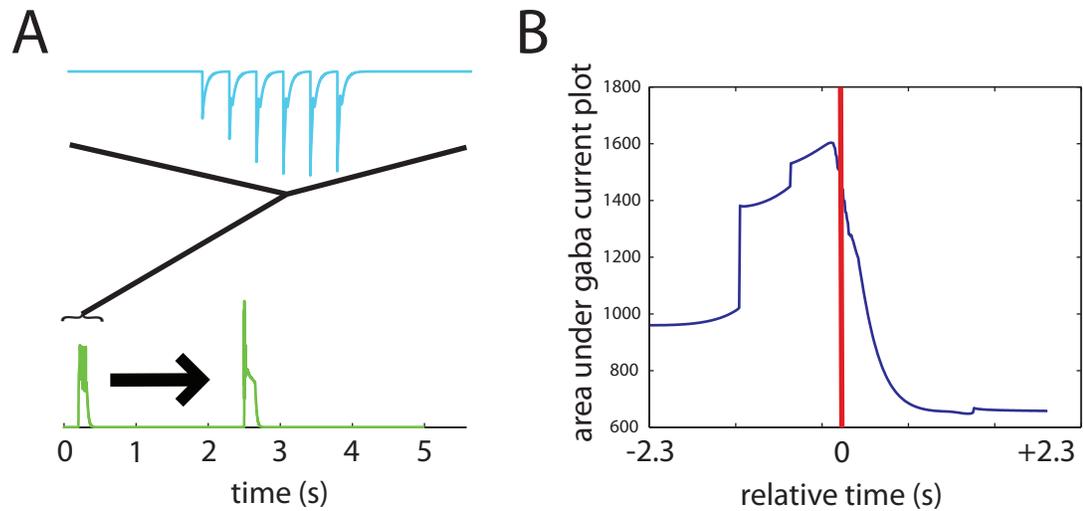


Figure A8.2. Effect of relative timing of proximal (APC) and distal (mitral cell soma) inputs to the granule onto mitral DDI. **A)** Distal inputs to the granule cell were held fixed at time 2.5 s while a proximal input was delivered between 0 and 5 s. The plot shows proximal input at about 0.25 s. **B)** DDI in the mitral cell measured for several different relative timings of the proximal and distal inputs.

example, injecting current into the Gr cell body produced a response very similar to the response observed in Inoue and Strowbridge 2008. Post-inhibitory rebound spikes due to low-threshold Ca currents as well as oscillatory behavior in isolated Gr cells also matched the results from Inoue and Strowbridge 2008.

Phenomena observed in Balu et al. 2007 were also recreated by the model. Slice recordings showed that short-term plasticity at the Gr cell distal and proximal excitatory synapses is modulated in different ways; stimulus input trains to proximal AMPA synapses at first show facilitation but then show depression, whereas stimulus input trains to distal AMPA synapses immediately show depression. In addition, DDI in the Mi→Gr→Mi circuit was observed in Balu et al. 2007 to be gated by proximal input to Gr cells arising from the cortex (APC). The authors hypothesized that APC input achieved this effect by relieving the Mg block on NMDA receptors at the

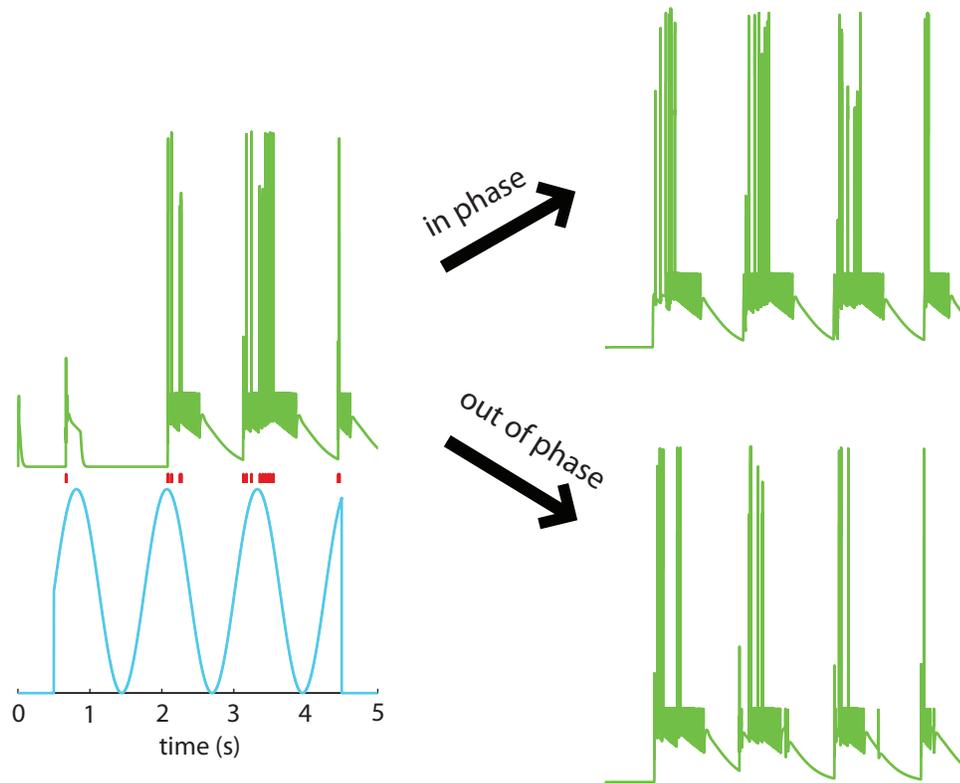


Figure A8.3. Repeated proximal and distal stimuli at the same frequency, Part I. DDI was measured when repeated proximal and distal stimuli were delivered at the same frequency but different phases. When both stimuli are strong, there is little difference between DDI when the stimuli are in phase (top right) and when the stimuli are out of phase (bottom right).

Mi→Gr synapse. Although the in-vitro appearance of DDI activity in their slice preparation was not recreated in detail by the model, the essential aspects of the cortical gating effect were reproduced. A subsequent paper (Gao and Strowbridge 2009) showed that, in addition to short term plasticity, the proximal and distal inputs display different forms of long-term plasticity. LTP could be added to the model in future work.

An intriguing hypothesis is that cortical gating could depend in interesting ways on the relative timing of proximal and distal stimuli. To investigate this idea, a fixed distal input was delivered at different times relative to a sliding proximal input

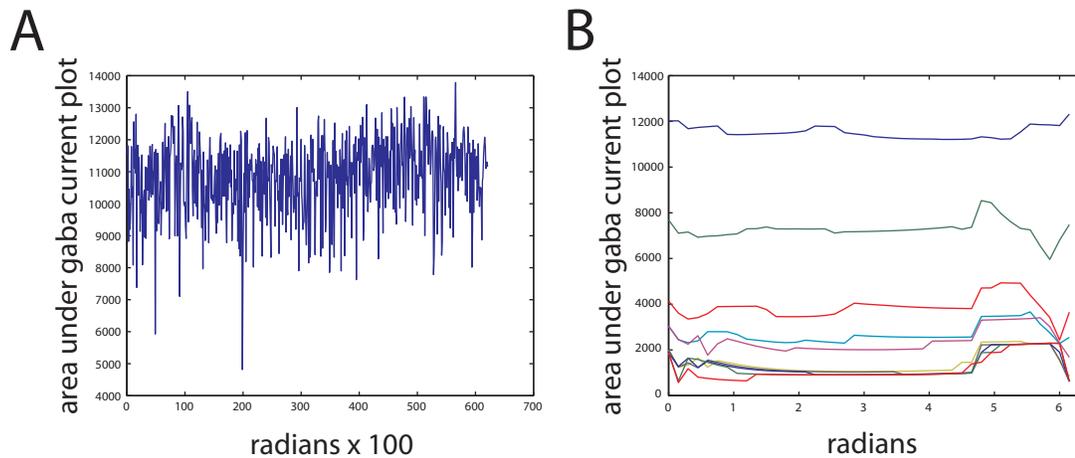


Figure A8.4. Repeated proximal and distal stimuli at the same frequency, Part II. **A)** When both proximal and distal stimuli are strong, there is little difference in DDI level regardless of phase difference between the proximal and distal stimulus trains as measured in radians. **B)** When proximal and distal stimuli are weaker, there are clear differences between DDI levels depending on the phase difference between the proximal and distal stimulus trains. The different lines shown are DDI level as a function of phase for several different frequencies of proximal/distal stimulus trains. The frequency of these trains decreases from 20 Hz (top curve) to 2 Hz (bottom curve).

over several different trials. These results show that the case where proximal input leads the distal input slightly gives rise to the greatest amount of mitral cell DDI (Figure A8.2). In subsequent experiments, distal and proximal inputs that repeated at the same fixed frequency except that the phases of the two stimuli were shifted relative to each other were used (Figure A8.3). Under strong stimulation, there appeared to be little to no effect of phase differences between the stimulus trains (Figure A8.4a), but with weaker stimulation, slightly leading proximal inputs created the largest mitral cell DDI responses (Figure A8.4b).

A functional implication of cortical gating of DDI in the $Mi \rightarrow Gr \rightarrow Mi$ circuit is that it is a potential mechanism by which the cortex can control synchronization of oscillatory activity across the bulb. This possibility should be explored further in

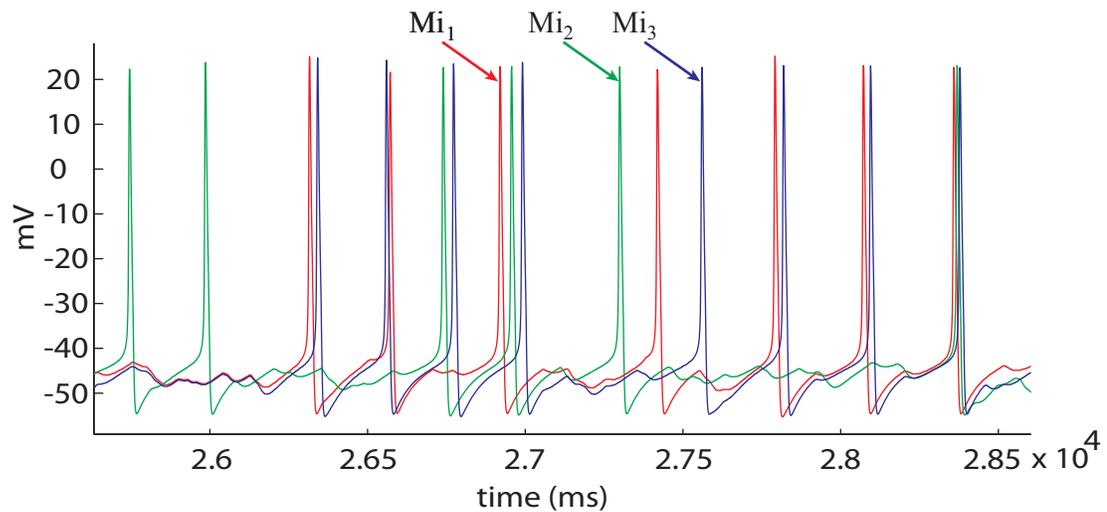


Figure A8.5. Synchronization of Mi cell oscillatory activity. Plot shows a portion of the spike trains from three different isolated model Mi cells: Mi_1 , Mi_2 , and Mi_3 . Poisson inhibitory inputs to the Mi_1/Mi_3 pair had $C_{in} = 0.8$ and inputs to the Mi_1/Mi_2 pair had $C_{in} = 0.0$.

future work, but as a first step, the essential results of Galan et al. 2006 were reproduced in the model in order to confirm that Gr cell input could achieve synchronization of oscillatory activity in Mi cells in the model. Galan et al. 2006 showed that the oscillatory activity of two Mi cells can be synchronized if they share correlated Poisson input, which in their work was applied in the form of current directly injected into the Mi cells. The same effect was observed in the present $Mi \rightarrow Gr \rightarrow Mi$ model circuit with direct injection of current into Mi cells (Figure A8.5). The next step in future work could be to create a larger network of on the order of 10 $Mi \rightarrow Gr \rightarrow Mi$ units with cortical inputs in order to explore the dynamics of synchronization of Mi cell oscillatory activity in this context.

APPENDIX 9

OVERVIEW OF MACHINE LEARNING

This appendix is a brief overview of some of the issues that lie at the heart of supervised machine learning problems, in which the task is to use training data to infer a model f that, it is hoped, will accurately map inputs to outputs for both examples in the training data set and for examples outside of the training data set, which may be part of a test data set. The treatment is based in part on the textbooks Bishop 2006 and Abu-Mostafa et al. 2012.

Introduction

Two fundamentally different approaches to identifying f (an unknown target function) are learning and design. Learning involves using training data to guide a search for a hypothesis that matches f for the training data. Design involves analytically deriving f using information about the problem, without the need to see any data. An example of design is modeling a process with a normal distribution or a physical law derived from first principles, and then constructing the optimal decision rule based on that model. Statistical approaches to learning tend to focus on a restricted class of idealized statistical models that can be rigorously analyzed, whereas machine learning deals with a wider variety of models that are often more difficult to analyze rigorously. Data mining is similar to machine learning except the emphasis

tends to be on analyzing and summarizing large data sets instead of learning predictive models.

Most machine learning problems are supervised learning problems, in which the correct output is known for each example of an input in a fixed data set. Active learning is when training data is acquired through queries made by the learning algorithm, usually in such a way as to maximize the information value of the examples. Online learning is when example data is continuously acquired and used for learning even after the learner is “deployed.” Reinforcement learning is used for problems in which the target output for each input is not known, but some information about the output is known. The easiest way to see how reinforcement learning is different from supervised learning is with the example of learning to play a game. In this case, the correct move (output) from a given board state (input) is not known. Instead, the reward for a particular move can be specified without knowing the rewards for other possible moves. Unsupervised learning involves identifying patterns in input data (such as clusters) without any knowledge of the target outputs.

The error measure used as part of the learning algorithm depends on the way in which the learned function is to be used. For example, it might be much worse to have false positives than false negatives on a classification task, in which case the error value assigned to false positives should be much higher than the error value assigned to false negatives.

Training and Testing

As f is unknown, training data never tells us anything *certain* about the behavior of f outside of the training data. However, if training examples and examples outside the training data set are independently drawn from the same arbitrary probability distribution, and if the target outputs consist of a binary class label such as $\{-1, +1\}$, it can be guaranteed that the classification error rate on examples outside the training data set E_{out} is bounded by the error rate on examples in the training data set E_{in} for any tolerance $\delta > 0$:

$$E_{out}(g) \leq E_{in}(g) + \sqrt{\frac{8}{N} \ln \frac{4m_H(2N)}{\delta}} \quad (\text{A9.1})$$

with probability $\geq 1 - \delta$, where g is the learned approximation to the unknown target function f , N is the number of examples in the training data set, and $m_H(N)$ is the growth function: the maximum number of dichotomies that can be generated by any hypothesis in H on any N data points. Classification error is the fraction of the time that the classification generated by a hypothesis for some input data disagrees with the classification generated by f . H is the space of possible hypotheses considered by the learning algorithm, from which g is chosen. A dichotomy is an assignment of binary class labels to a sample of N data points. (2^N is the largest number of dichotomies possible for N data points.)

Equation A9.1 is one formulation of the VP generalization bound, and is perhaps the most important mathematical result in the theory of learning. Although it

is usually difficult to use the VP generalization bound for practical purposes, a general rule of thumb that has been observed in practice is that N should be at least $10 \cdot d_{VC}$, where d_{VC} is the VC dimension, which is the largest value of N for which $m_H(N) = 2^N$. The VC generalization bound also shows how increasing the complexity of H makes the bound on E_{out} more unfavorable, which is equivalent to poorer generalization. (Good generalization is when $E_{out} \approx E_{in}$.) However, more complex H can achieve lower E_{in} . In general, the level of complexity of H should be chosen to minimize E_{out} , which involves a balance between achieving a low E_{in} (approximating f well on the training data) and achieving $E_{out} \approx E_{in}$ (good generalization on new data). This is the approximation-generalization tradeoff that is a fundamental aspect of machine learning.

The tradeoff involved in choosing more or less complex H can also be quantified by decomposing the squared out-of-sample error E_{out} (not the same as the classification out-of-sample error as above) into bias and variance terms:

$$E_D [E_{out}(g^{(D)})] = E_X [\text{bias}(\mathbf{x}) + \text{var}(\mathbf{x})] \quad (\text{A9.2})$$

where $E_D [E_{out}(g^{(D)})]$ is the expected value of E_{out} , E_X is the expected value with respect to input data \mathbf{x} , $\text{bias}(\mathbf{x}) = (\bar{g}(\mathbf{x}) - f(\mathbf{x}))^2$, $\text{var}(\mathbf{x}) = E_D [(g^{(D)}(\mathbf{x}) - \bar{g}(\mathbf{x}))^2]$, and $\bar{g}(\mathbf{x})$ is the average hypothesis obtained by the learning algorithm from an infinite number of runs on an infinite number of data sets D . Intuitively, bias is the difference

between the average learned hypothesis and the target function, which tends to go down with larger H , and variance is the average difference between the average learned hypothesis and a given learned hypothesis, which tends to go up with larger H . Note that neither bias nor variance are exactly equivalent to either approximation (E_{in}) or generalization ($E_{out} - E_{in}$). As with VC analysis, it can be difficult to directly apply bias-variance analysis in practice. Reducing the variance without significantly increasing the bias can be accomplished through some general techniques such as regularization. Reducing the bias without significantly increasing the variance requires using some prior information regarding the target function to steer the selection of H in the direction of f , and this task is largely application-specific.

Regardless of the complexity of H , the following general principles hold in most practical machine learning applications: 1) a more complex or a noisy f makes low E_{in} more difficult to achieve, 2) increasing N causes E_{in} to increase, 3) increasing N causes E_{out} to decrease, and 4) increasing N causes both generalization error and variance to decrease. E_{test} , the error rate on a test data set, gives a better estimate of E_{out} than E_{in} and is bounded by:

$$\Pr[|E_{test}(g) - E_{out}(g)| > \epsilon] \leq 2e^{-2\epsilon^2 N} \quad (\text{A9.3})$$

We say the hypothesis g_1 overfits the data as compared to g_2 when $E_{out}(g_1) > E_{out}(g_2)$ but $E_{in}(g_1) < E_{in}(g_2)$. Note that overfitting can occur when either g_1 or g_2 are more complicated, less complicated, or equally complicated when

compared to the target f or to each other. If g_1 is more complex than g_2 , then regardless of whether $E_{out}(g_1)$ is greater than, less than, or equal to $E_{out}(g_2)$, a more complex or a noisier f increases $E_{out}(g_1) - E_{out}(g_2)$ and an increasing N decreases $E_{out}(g_1) - E_{out}(g_2)$.

Regularization is the practice of decreasing $E_{out}(h)$ by minimizing $E_{in}(h) + \Omega(h)$ instead of just $E_{in}(h)$ alone, where $\Omega(h)$ is the regularizer: some measure of the complexity of an individual hypothesis h . This can often be done in such a way as to greatly reduce variance without significantly increasing bias. In practice it often works better to use complex H but constrain hypotheses $h \in H$ to be simple rather than to use a simple H . The best type of regularization is largely determined by experiment, but a common approach is weight decay, where $\Omega(h) = \lambda \mathbf{w}^T \mathbf{w}$, \mathbf{w} is the weight vector that specifies h , and λ is the regularization parameter, which controls the amount of regularization. For linear regression, a closed form solution can be found that minimizes $E_{in}(\mathbf{w}) + \lambda \mathbf{w}^T \mathbf{w}$. In practice, different regularizers often perform similarly as long as λ is chosen correctly, which can be done with validation.

Classification error is the fraction of the time that the classification generated by a hypothesis h for some input data disagrees with the classification generated by the unknown target function f . Sometimes it is of interest to estimate the classification error rate for arbitrary input data drawn from a distribution D ($error_D(h)$) by using the classification error rate obtained on a sample of input data drawn from the same fixed

distribution $D(\text{error}_s(h))$. Note that in this setting, h is fixed and independent of the sample of input data, that is, the sample is not training data used to learn h . Also, the samples must be drawn independently of each other. Basic statistical sampling theory shows that with approximately 95% probability, $\text{error}_D(h)$ lies in the interval:

$$\text{error}_s(h) \pm 1.96 \sqrt{\frac{\text{error}_s(h)(1 - \text{error}_s(h))}{n}} \quad (\text{A9.4})$$

This bound approximately holds if the sample size $n \geq 30$ and $\text{error}_s(h)$ is not too close to 0 or 1, or more precisely, when:

$$n \cdot \text{error}_s(h)(1 - \text{error}_s(h)) \geq 5 \quad (\text{A9.5})$$

The bound follows from the fact that any fixed process that randomly generates one of two possible values 1 (with probability p) or 0 (with probability $(1 - p)$) will generate r observations of 1 out of n total observations with probability given by the Binomial distribution:

$$\text{Pr}(r \text{ out of } n) = \frac{n!}{r!(n-r)!} p^r (1-p)^{n-r} \quad (\text{A9.6})$$

The mean of this distribution is np and the standard deviation is $\sqrt{np(1-p)}$. This mean and standard deviation can be estimated using an observed value for r , and a distance around the mean (measured in standard deviations) that contains $N\%$ of the distribution can then be calculated. If, for example, N is chosen to be 95, then 95% of the time, the observed r must fall within this distance around the mean, which is

equivalent to saying that 95% of the time, the mean must fall within this distance around the observed r . The exact endpoints that define the 95% confidence interval are calculated relative to the observed r .

Finding this interval for a given observed r and value of N is difficult to do directly using the Binomial probability density function, but the Binomial distribution is well approximated by the Normal distribution when $np(1-p) \geq 5$. One-sided intervals can also be obtained for which confidence can be raised from, for example, 80% to 90% by bounding the interval on only one side. The argument applies equally well to obtaining confidence intervals on the estimate of $error_D(h)$ as shown above. Note that the value $f(x)$ of a probability density function describing some observation x at a given observed value is not a probability itself; rather, we are interested in the area under the probability density function curve integrating to $N\%$ of the total area under the curve, which always equals 1.

If the distribution governing the measured value (r in this case) is something other than the Binomial distribution, the same approach can be used with another probability density function to obtain confidence intervals. In particular, the Central Limit Theorem says that when the measured value is the sample mean, then as $n \rightarrow \infty$, the distribution of the measured values approaches a Normal distribution with mean equal to the mean of the arbitrary distribution governing the individual samples and with standard deviation equal to $\frac{\sigma}{\sqrt{n}}$, where σ is the standard deviation of the arbitrary distribution governing the individual samples. Note that for any $N\%$ confidence interval, it is not strictly true that there is an $N\%$ chance that the true

parameter value lies within the interval given. This is because the properties of the distribution governing the parameter can only be estimated.

As the distribution of $error_s(h)$ is approximately Normal, the estimated difference in the error of two hypotheses $error_{s_1}(h_1) - error_{s_2}(h_2)$ is also Normal, and this fact can be used to put confidence intervals on the true difference in the error $error_D(h_1) - error_D(h_2)$. If the confidence interval is one-sided, statements can be made such as $error_D(h_1) > error_D(h_2)$ with 95% confidence.

The estimated difference in the error of two hypotheses can be repeatedly calculated for two different learning algorithms in order to estimate which is the better learner “on average.” Each calculation of a difference δ_i involves an independent set of training data and a set of test data. If the size of the test data set is $n \geq 30$, then each δ_i is approximately governed by a Normal distribution as discussed above. As the sample mean of several Normally distributed variables is also Normal, the mean value of the δ_i is $\hat{\delta}$, which follows a Normal distribution. As usual, the standard deviation s of the distribution governing $\hat{\delta}$ is not known, but can be estimated as:

$$s = \sqrt{\frac{1}{k(k-1)} \sum_{i=1}^k (\delta_i - \hat{\delta})^2} \quad (\text{A9.7})$$

where k is the number of samples δ_i . However, this estimate applies not to a Normal distribution but to a (paired) t distribution. As a result, confidence intervals on the true value of δ are based on the area under the appropriate t distribution with mean $\hat{\delta}$. In

practice each δ_i must be calculated by resampling a limited set of data. Different schemes for resampling the data involve different tradeoffs in attempting to reach good confidence intervals that approximate those that would be obtained with the idealized scenario.

Bayesian Perspectives on Learning

The “best” hypothesis given the training data can be defined as the most probable hypothesis h from the space of possible hypotheses H given the training data D . The probabilities of different hypotheses given the training data can be found with Bayes’ Theorem:

$$P(h | D) = \frac{P(D | h)P(h)}{P(D)} \quad (\text{A9.8})$$

where $P(h)$ is the prior probability of h (which is uniform over all possible hypotheses in H if all hypotheses are equally likely), $P(h|D)$ is the posterior probability of h , $P(D)$ is the prior probability of observing the training data, and $P(D|h)$ is the likelihood of observing the training data given h . More precisely, $P(D|h)$ is:

$$\prod_{i=1}^m p(x_i, d_i | h) = \prod_{i=1}^m p(d_i | h, x_i) p(x_i) \quad (\text{A9.9})$$

where m is the number of (mutually independent) training examples, d_i is the i th target value for input value x_i , x (but not d) is independent of h , and p is the probability of

observing that target value if the output generated by $h(x_i)$ is the true target value. If there is measurement noise in the target values (such as Normally distributed noise with 0 mean) and the output of the hypothesis is supposed to match the target output, the probability of observing a particular target value will be nonzero even if it does not exactly equal the output generated by $h(x_i)$, whereas it will be 0 if there is no noise in the target values and the observed target value does not exactly equal the output generated by $h(x_i)$.

The most probable hypothesis h is the maximum a posteriori (MAP) hypothesis:

$$h_{\text{MAP}} = \operatorname{argmax}_{h \in H} \frac{P(D|h)P(h)}{P(D)} = \operatorname{argmax}_{h \in H} P(D|h)P(h) \quad (\text{A9.10})$$

The MAP hypothesis can be determined without $P(D)$ because it is a constant independent of h . If in addition every hypothesis in H is assumed to be equally probable, then the above reduces to:

$$h_{\text{ML}} = \operatorname{argmax}_{h \in H} P(D|h) \quad (\text{A9.11})$$

where h_{ML} is the maximum likelihood (ML) hypothesis.

The most probable classification for a new instance of input data is not necessarily the classification generated by the MAP hypothesis for that input data. Rather, it is the classification given by the Bayes optimal classifier:

$$\operatorname{argmax}_{v_j \in V} P(v_j | D) = \operatorname{argmax}_{v_j \in V} \sum_{h_i \in H} P(v_j | h_i) P(h_i | D) \quad (\text{A9.12})$$

where the classification of the new instance can take on any value v_j from the set of possible values V and h_i is the i th hypothesis from the set of all possible hypotheses H . In practice this classification can be much more difficult to find than the classification generated by the MAP hypothesis. A compromise is to use Gibbs sampling, in which the new instance is classified by a hypothesis h randomly chosen from H according to the posterior distribution $P(h|D)$. The expected error of the Gibbs algorithm is at most twice the expected error of the Bayes optimal classifier.

An alternate form for the Bayes optimal classifier that does not explicitly involve a hypothesis is:

$$\operatorname{argmax}_{v_j \in V} P(v_j | a_1, a_2 \dots a_n) = \operatorname{argmax}_{v_j \in V} P(a_1, a_2 \dots a_n | v_j) P(v_j) \quad (\text{A9.13})$$

where $\langle a_1, a_2 \dots a_n \rangle$ are the (discrete) attribute values that describe the new instance. This form is useful primarily because if the attribute values are independent, the optimal classifier is equivalent to the Naive Bayes classifier:

$$\operatorname{argmax}_{v_j \in V} P(v_j) \prod_i P(a_i | v_j) \quad (\text{A9.14})$$

with probability terms that are much more easily estimated from the training data than the terms in the optimal classifier. To avoid difficulties in estimating the probability terms when the number of observations is small, the m -estimate can be used instead:

$$\frac{n_c + mp}{n + m} \tag{A9.15}$$

where n_c is the number of training examples that take on a certain value, n is the total number of training examples considered for the probability calculation, p is a prior estimate of the probability that is being calculated (which is often assumed to be uniform), and m is a constant called the equivalent sample size, which determines how heavily to weight p relative to the observed data. Even if the assumption of independent attribute values is incorrect, the Naive Bayes classifier can perform surprisingly well in many practical situations.

In general, the joint probability distribution for an n -tuple of random variables $\langle Y_1 \dots Y_n \rangle$ is:

$$P(y_1, \dots, y_n) = \prod_{i=1}^n P(y_i | Parents(Y_i)) \tag{A9.16}$$

where $Parents(Y_i)$ is the value of the variables on which variable Y_i is dependent. Each variable could be dependent on all the other variables, none of the other variables (as in the assumption underlying the Naive Bayes classifier), or on any combination of the other variables. An arbitrary set of dependencies along with a “table” for each variable specifying $P(y_i | Parents(Y_i))$ can be represented with a Bayesian belief network, which is a type of probabilistic graphical model. (The other major type of probabilistic graphical model is the Markov random field.) Inference with a Bayesian belief network involves calculating probability distributions for subsets of the network

variables that are of interest. Numerous exact and approximate methods for performing this inference have been proposed, both for settings in which all variables are measured as well as settings in which only a subset of the variables are measured. Learning of a network from training data is closely related to inference with the network, and learning can be accomplished with methods such as gradient ascent even when some of the variables are unmeasured or when the structure of the network is unknown.

The EM algorithm is an approach to learning a hypothesis h (and estimating values for unobserved variables) that consists of some set of parameters that describe an underlying probability distribution, in the presence of unobserved variables. It is applicable in a wide variety of settings, including the learning of Bayesian belief networks. Call the full data Y where each instance consists of the value of each random variable in an $(m + n)$ -tuple $\langle X_1 \dots X_m, Z_1 \dots Z_n \rangle$. The observed data X consists of the m observed variables for each instance and the unobserved data Z consists of the n unobserved variables for each instance. In general, the EM algorithm repeats the following two steps until (local) convergence:

Step 1: *Estimation (E) step*: Calculate $Q(h' | h)$ using the current hypothesis h and the observed data X to estimate the probability distribution over Y .

$$Q(h' | h) \leftarrow E[\log p(Y | h') | h, X]$$

Step 2: *Maximization (M) step*: Replace hypothesis h by the hypothesis h' that maximizes this Q function.

$$h \leftarrow \operatorname{argmax}_{h'} Q(h' | h)$$

The prototypical example of a problem involving hidden variables is the k -means problem, in which the task is to estimate $h = \langle \mu_1, \dots, \mu_k \rangle$, which are the means of a mixture of k Normal distributions. Each training instance in the problem is generated by one of the k distributions and consists of the observed value plus unobserved variables indicating which distribution generated the instance. This problem can be solved by the k -means algorithm, an instantiation of the generalized EM algorithm.

The Bayesian framework allows one way to characterize the behavior of learning algorithms even when the learning algorithm does not explicitly manipulate probabilities. For example, every consistent learner in a classification task outputs a MAP hypothesis if we assume a uniform prior probability distribution over H and if we assume deterministic, noise free training data, that is, $P(D|h) = 1$ if h and D are consistent, and 0 otherwise. Here, a consistent learner is one that outputs a hypothesis that is consistent with the data D in that it commits no errors on the training data.

Bayesian analysis can also be used to show that for continuous-valued target functions, any learning algorithm that minimizes the sum of the squared errors between the output hypothesis predictions and the observed training data will output a maximum likelihood hypothesis. This holds under the assumption that the observed training values are generated by adding random measurement noise to the true target value, where this random noise is drawn independently for each example from a Normal distribution with 0 mean. If instead the training values are binary $\{-1, +1\}$

and free of measurement noise but are probabilistically generated by an unknown function, then the desired target function outputs a probability of the output being +1 for any given input. Bayesian analysis can be used to show that the maximum likelihood hypothesis in this setting is found by any learning algorithm that minimizes the cross entropy of the output of the hypothesis and the observed training data. A gradient descent algorithm very similar to BACKPROPAGATION can be derived for ANNs that output probabilities and that attempts to minimize the cross entropy.

If efficiency is defined as the number of bits necessary to uniquely identify a message, then the theoretical most efficient way to uniquely identify a randomly chosen message i from among the set of all possible messages is to assign $-\log_2 p_i$ bits to identify the message, where p_i is the probability of choosing the message and the quantity as a whole is called the description length of message i . (The average message description length $\sum_i p_i(-\log_2 p_i)$ over the set of all possible messages is the entropy of the set of all possible messages.) Other assignments define different encodings C that are less efficient than the theoretical optimal encoding. The description length of a message under encoding C is $L_C(i)$. Using the optimal encoding, the description length of a hypothesis h is $-\log_2 P(h)$ and the description length of a set of training data given that hypothesis h is true is $-\log_2 P(D|h)$. The sum of these two values is the theoretical minimum number of bits needed to both identify the hypothesis out of the set of all possible hypotheses and identify the set of training data out of the set of all possible training data sets. The hypothesis that minimizes this sum is the same as the MAP hypothesis:

$$h_{\text{MAP}} = \underset{h \in H}{\operatorname{argmax}} P(D|h)P(h) = \underset{h \in H}{\operatorname{argmin}} -\log_2 P(h) - \log_2 P(D|h) \quad (\text{A9.17})$$

In general, finding the minimum description length (MDL) hypothesis:

$$h_{\text{MDL}} = \underset{h \in H}{\operatorname{argmin}} L_{C_1}(h) + L_{C_2}(D|h) \quad (\text{A9.18})$$

is equivalent to finding the MAP hypothesis when C_1 and C_2 are the optimal encodings. The Minimum Description Length principle can be thought of as a formalization of Occam's Razor that states that the best hypothesis for a given set of data is the one that leads to the best compression of the data, where compression is the sum of hypothesis complexity and accuracy on the training data set. This principle is only justified in the Bayesian sense when the best hypothesis (the MDL hypothesis) is found with respect to the optimal encodings.

Linear Models

Linear models are good learning methods to try first as they are simple and generalize well due to the low complexity of the hypothesis space. A linear model (also called a perceptron) makes predictions based on a linear combination of the inputs:

$$s(\mathbf{x}) = \sum_{i=0}^d w_i x_i = \mathbf{w}^T \mathbf{x} \quad (\text{A9.19})$$

where s is the signal, d is the dimensionality of the data, $\mathbf{x} \in \mathbb{R}^d$ is a vector of inputs,

$\mathbf{w} \in \mathbb{R}^{d+1}$ is the weight vector that represents the learned hypothesis, and $x_0 = 1$.

When the output of the linear model $h(\mathbf{x}) = s(\mathbf{x})$, it is a linear regression model, whereas when the output of the linear model $h(\mathbf{x}) = \text{sign}[s(\mathbf{x})]$, it is a linear classification model (with binary outputs $\{-1, +1\}$). The linear classification model represents a hyperplane decision surface in the n -dimensional space of instances. The weights specify the normal vector to the decision hyperplane (plus the offset of the hyperplane from the origin). The sign function is called a threshold function.

When the output of the linear model $h(\mathbf{x}) = \theta[s(\mathbf{x})]$, where the sigmoid function $\theta(s) = \frac{e^s}{1 + e^s}$, it is a logistic regression model. The threshold function in this case is nonlinear (and differentiable) and takes on values between 0 and 1.

The weights in a linear classification model can be learned from data using the very simple PERCEPTRON LEARNING ALGORITHM, which converges after a finite number of iterations to a hypothesis that perfectly classifies the training data, provided the training examples are linearly separable. The weights in a linear regression model can be “learned” from the data directly with the Normal equations, which specify the optimal weights that minimize the sum of the squared errors between observed and predicted target values. If the training values are binary $\{-1, +1\}$ and free of measurement noise but are probabilistically generated by an unknown function, then the desired target function outputs a probability of the output being +1 for any given input. The output of a logistic regression model can be interpreted as such a probability, and in that case, the optimal weights are those that minimize the cross entropy between the output of the hypothesis and the observed training data. An

analytic solution for the optimal weights is not feasible in this case, so gradient descent can be used. Also in this particular case, gradient descent works well because the function to be minimized is convex and hence has a single local minimum equivalent to the global minimum. The optimal weights learned in a linear regression model (if the target values are $\{-1, +1\}$) and the optimal weights learned in a logistic regression model that outputs probabilities (if the output is mapped: $\{0.5 \text{ or greater} = +1, \text{ less than } 0.5 = -1\}$) are approximations to the optimal weights for the corresponding linear classification model, and it can be much easier to find these approximations than trying to directly find good weights for the linear classification model.

Batch gradient descent and the variant stochastic gradient descent are important general paradigms for learning. Stochastic gradient descent can be much faster because weights are updated upon examining a single randomly chosen training example instead of only after examining all the training examples as in batch gradient descent. Both are strategies for searching through a large or infinite hypothesis space that can be applied whenever the hypothesis space contains continuously parameterized hypotheses (such as the weights in a linear model), and whenever the error can be differentiated with respect to these hypothesis parameters. The key practical difficulties in applying gradient descent are that convergence to a local minimum can sometimes be quite slow, and if there are multiple local minima in the error surface, there is no guarantee that the procedure will find the global minimum.

The input data to a linear model can be transformed with an arbitrary function $\Phi(\mathbf{x}) = \mathbf{z}$ and the resulting \mathbf{z} used in a linear model just as with \mathbf{x} above. The feature

transform Φ maps data in the input space $\mathbf{x} \in X$ to data in the feature space $\mathbf{z} \in Z$, where X and Z may have different dimensionality and not all $\mathbf{x} \in X$ may be mapped to a unique $\mathbf{z} \in Z$. For example, if the linear classification model is used and the feature transform is $\Phi(\mathbf{x}) = (1, x_k)$ for the k th coordinate of \mathbf{x} , the model is called the decision stump model on dimension k . Selecting the feature transform should be done before seeing the data but can be based on an understanding of the problem setting. Error may be much lower in feature space, but the usual penalties apply if the dimensionality (and hence complexity of hypothesis space) of the feature space is higher than that of the input space.

Artificial Neural Networks

Artificial neural networks (ANNs) are for problems in which instances are represented by many attribute-value pairs, the target function may be discrete, real, or a vector of several discrete or real values, the training examples may contain errors, long training times are acceptable, fast evaluation of the learned target function may be required, and in which the ability of humans to understand the target function is not important.

BACKPROPAGATION is a gradient descent method for optimizing the weights of multilayer ANNs with continuous-valued outputs in an attempt to minimize the squared error between the outputs of the network and the target outputs in the training data set. A perceptron with a nonlinear, differentiable threshold function is the fundamental unit of a multilayer ANN that can represent highly nonlinear functions

and can be learned with BACKPROPAGATION. Gradient descent search over the complex error surfaces represented by multilayer ANNs with nonlinear units is prone to being trapped in local optima. However, this problem can be ameliorated with: momentum terms, stochastic gradient descent instead of true gradient descent, training several times and taking the network with best performance on validation data, and committees of networks. Although the knowledge is not necessarily useful in practice, it is known that any boolean function and any bounded continuous function can be approximated with arbitrarily small error by a network with two layers of units, and that any function can be approximated to arbitrary accuracy by a network with three layers of units.

The hypothesis space considered by ANNs and BACKPROPAGATION is the continuous n -dimensional Euclidean space of the n network weights. The fact that it is continuous, together with the fact that the error function is differentiable with respect to the continuous parameters of the hypothesis, results in a well-defined error gradient that provides a very useful structure for organizing the search for the best hypothesis. The ability of multilayer networks to automatically discover useful representations at the hidden layers is a key feature of ANN learning. In contrast to learning methods that are constrained to use only predefined features provided by the human designer, this provides an important degree of flexibility that allows the learner to invent features not explicitly introduced by the human designer. Overfitting is a serious problem with ANNs, and this can be overcome to some extent by weight decay or various cross-validation techniques such as k -fold cross-validation (which is useful when small amounts of data are available) to find the best number of training

iterations to obtain good generalization to unseen data. Early stopping guided by cross-validation can be misleading though as error on validation data as a function of the number of training epochs can exhibit complex behavior.

Alternatives to the simplest error functions include: penalty terms for weight magnitude to reduce the risk of overfitting, adding a term for errors in the derivative of the target function, minimizing cross entropy when the network is to output probability estimates, and weight sharing. In the latter, different network weights are forced to take on identical values, usually to enforce some constraint known in advance to the human designer that can constrain the space of potential hypotheses and thereby reduce the risk of overfitting and improve the chances for accurately generalizing to unseen situations. Alternatives to gradient descent for weight optimization that can be faster than gradient descent (but generally do not improve generalization of the final network) include line search and the conjugate gradient method. Techniques for dynamically modifying network structure have met with mixed success in reliably improving on the generalization accuracy of BACKPROPAGATION. These techniques include adding units dynamically (CASCADE-CORRELATION) and removing units dynamically.

Recurrent networks (those with feedback loops) apply to time series data and can automatically incorporate information from previously seen data from an arbitrary window of time in the past, such as the prediction of stock price dependent on current indicators and on historical values of indicators. Recurrent networks are generally more difficult to train and do not generalize as reliably as those with no feedback loops.

REFERENCES

- Abbott, L.F. 1999. Lamicque's introduction of the integrate-and-fire model neuron. *Brain Research Bulletin* **50**:303-304.
- Abu-Mostafa, Y.S., M. Magdon-Ismail, H.-T. Lin. 2012. *Learning from Data*. AMLBook.
- Adachi, F., T. Washio, H. Motoda. 2006. Scientific discovery of dynamic models based on scale-type constraints. *IPSJ Digital Courier* **2**:607-619.
- Aguirre, L.A., U.S. Freitas, C. Letellier, J. Maquet. 2001. Structure-selection techniques applied to continuous-time nonlinear models. *Physica D* **158**:1-18.
- Aguirre, L.A., C. Letellier. 2009. Modeling nonlinear dynamics and chaos: a review. *Math. Prob. Engr.* doi:10.1155/2009/238960
- Ahalpara, D.P., A. Sen. 2011. A sniffer technique for an efficient deduction of model dynamical equations using genetic programming. In *Proceedings of the EuroGP 2011*, ed. by S. Silva, J.A. Foster, M. Nicolau, P. Machado, M. Giacobini. pp. 1-12.
- Akansu, A.N., P.R. Haddad. 2000. *Multiresolution Signal Decomposition: Transforms, Subbands, and Wavelets*. Academic Press, Waltham, MA.
- Ando, S., E. Sakamoto, H. Iba. 2002. Evolutionary modeling and inference of gene network. *Inf. Sci.* **145**:237-259.
- Andrew, H. 1996. System identification using genetic programming. In *Proceedings of the 2nd International Conference on Adaptive Computing in Engineering Design and Control*, pp. 57-62.
- Angeline, P.J., D.B. Fogel. 1997. An evolutionary program for the identification of dynamical systems. In *Proceedings of the SPIE*, pp. 409-417.

Argemí, A., J. Saurina. 2007. Study of the degradation of 5-azacytidine as a model of unstable drugs using a stopped-flow method and further data analysis with multivariate curve resolution. *Talanta* **74**:176-182.

Åström, K.J., P. Eykhoff. 1971. System identification-A survey. *Automatica* **7**:123-162.

Augusto, D.A., H.J.C. Barbosa. 2000. Symbolic regression via genetic programming. In *Proceedings of the 6th Brazilian Symposium on Neural Networks*, pp. 173.

Baake, E., M. Baake, H. G. Bock, K. M. Briggs. 1992. Fitting ordinary differential equations to chaotic data. *Phys. Rev. A* **45**:5524-5529.

Babovic, V., M. Keijzer. 2000. Evolutionary algorithms approach to induction of differential equations. In *Proceedings of the 4th International Conference on Hydroinformatics*, pp. 251-258.

Bakker, R., J. C. Schouten, C. L. Giles, F. Takens, C.M. van den Bleek. 2000. Learning chaotic attractors. *Neural Computation* **12**: 2355-2383.

Balu, R., R.T. Pressler, B.W. Strowbridge. 2007. Multiple modes of synaptic excitation of olfactory bulb granule cells. *J. Neurosci.* **27**:5621-5632.

Banzhaf, W., P. Nordin, R.E. Keller, F.D. Francone. 1997. *Genetic Programming: An Introduction on the Automatic Evolution of Computer Programs and Its Applications*. Morgan Kaufmann, San Francisco.

Batcher, K.E. 1968. Sorting networks and their applications. In *Proceedings of AFIPS Spring Joint Computer Conference*, pp. 307:314.

Beisler, J.A. 1978. Isolation, characterization, and properties of a labile hydrolysis product of the antitumor nucleoside, 5-azacytidine. *J. Med. Chem.* **21**:204-208.

Bender, M.A., M. Farach-Colton, M. Mosteiro. 2006. Insertion sort is $O(n \log n)$. *Theory of Computing Systems* **39**:391-397.

Bernardino, H.S., H.J.C. Barbosa. 2011. Inferring systems of ordinary differential equations via grammar based immune programming. *LNCS* 6825:198-211.

Bishop, C.M. 2006. *Pattern Recognition and Machine Learning*. Springer, London.

Boccaletti, S. 2008. *The Synchronized Dynamics of Complex Systems*. Elsevier, Amsterdam.

Bongard, J.C. 2011. Innocent until proven guilty: Reducing robot shaping from polynomial to linear time. *IEEE Trans. Evol. Comp.* **15**:571-585.

Bongard, J.C., H. Lipson. 2007. Automated reverse engineering of nonlinear dynamical systems. *PNAS* **104**:9943-9948.

Borcard, D., P. Legendre, C. Avois-Jacquet, H. Tuomisto. 2004. Dissecting the spatial structure of ecological data at multiple scales. *Ecology* **85**:1826-1832.

Boyer, R., J.S. Moore. 1984. A mechanical proof of the Turing completeness of pure Lisp. In *Automated Theorem Proving: After 25 Years*, ed. by W.W. Bledsoe, D.W. Loveland. American Mathematical Society, Providence, RI. pp. 133-167.

Bradley, E., M. Easley, R. Stolle. 2001. Reasoning about nonlinear system identification. *Artificial Intelligence* **133**:139-188.

Brameier, M.F., W. Banzhaf. 2001. A comparison of linear genetic programming and neural networks in medical data mining. *IEEE Trans. Evol. Comp.* **5**:17-26.

Brameier, M.F., W. Banzhaf. 2007. *Linear Genetic Programming*. Springer, New York.

Breeden, J.L., A. Hübler. 1990. Reconstructing equations of motion from experimental data with unobserved variables. *Phys. Rev. A* **42**:5817-5826.

Bridewell, W., P. Langley, L. Todorovski, S. Džeroski. 2008. Inductive process modeling. *Machine Learning* **71**:1-32.

Bridewell, W., J.N. Sanchez, P. Langley, D. Billman. 2006. An interactive environment for the modeling and discovery of scientific knowledge. *International Journal of Human-Computer Studies* **64**:1099-1114.

Butcher, J.C. 2008. *Numerical Methods for Ordinary Differential Equations*. Wiley, New York.

Cao, H., L. Kang, Y. Chen, J. Yu. 2000. Evolutionary modeling of systems of ordinary differential equations with genetic programming. *Genetic Programming and Evolvable Machines* **1**:309-337.

Cao, L.J., F.E.H. Tay. 2003. Support vector machine with adaptive parameters in financial time series forecasting. *IEEE Trans on Neural Networks* **14**:1506-1518.

Chen, W., J. Fish. 2001. A dispersive model for wave propagation in periodic heterogeneous media based on homogenization with multiple spatial and temporal scales. *J. Appl. Mech.* **68**:153-161.

Chen, W.W., M. Niepel, P.K. Sorger. 2010. Classic and contemporary approaches to modeling biochemical reactions. *Genes. Dev.* **24**:1861-1875.

Chis, O.-T., J.R. Banga, E. Balsa-Canto. 2011. Structural identifiability of systems biology models: a critical comparison of methods. *PLOS One* **6**:e27755

Cho, K.-H., S.-Y. Shin, H.-W. Kim, O. Wolkenhauer, B. McFerran, W. Kolch. 2003. Mathematical modeling of the influence of RKIP on the ERK signaling pathway. *LNCS* **2602**:127-141.

Chung, J., D. Jung, H.H. Yoo. 2004. Stability analysis for the flapwise motion of a cantilever beam with rotary oscillation. *J. Sound Vibration* **273**:1047-1062.

- Clery, D., D. Voss. 2005. All for one and one for all. *Science* **308**:809.
- Conca, P., G. Nicosia, G. Stracquadanio. 2008. A clonal selection algorithm for the automatic synthesis of low-pass filters. In *Proceedings of Wivace 2008*, pp. 69-79.
- Cormen, T.H., C.E. Leiserson, R.L. Rivest, C. Stein. 2009. *Introduction to Algorithms, Third Edition*. MIT Press, Cambridge, MA.
- Cremers, J., A. Hübler. 1987. Construction of differential equations from experimental data. *Zeitschrift für Naturforschung A* **42**:797-802.
- Crutchfield, J.P., B. S. McNamara. 1987. Equations of motion from a data series. *Complex Systems* **1**:417-452.
- Das, A., R. Vemuri. 2007. An automated passive analog circuit synthesis framework using genetic algorithms. In *Proceedings of IEEE VLSI '07*, pp. 145-152.
- Das, A., R. Vemuri. 2009. A graph grammar based approach to automated multi-objective analog circuit design. In *Proceedings of Design, Automation, Test Eur. Conf.*, pp. 700-705.
- De Jong, K.A. 2006. *Evolutionary Computation: a Unified Approach*. MIT Press, Cambridge, MA.
- Deb, K., A. Pratap, S. Agarwal, T. Meyarivan. 2000. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evol. Comp.* **6**:182-197.
- Deng, L., X. Trepate, J.P. Butler, E. Millet, K.G. Morgan, D.A. Weitz, J.J. Fredberg. 2006. Fast and slow dynamics of the cytoskeleton. *Nat. Materials* **5**:636-640.
- Destexhe, A., Z.F. Mainen, T.J. Sejnowski. 1998. Kinetic models of synaptic transmission. In *Methods in Neuronal Modeling, 2nd Edition*, ed. by C. Koch, I. Segev. MIT Press, Cambridge, MA. pp. 1-26.
- Dietterich, T.G. 2002. Machine learning for sequential data: A review. *LNCS*

2396:15-30.

Douglas, R., M. Mahowald, C. Mead. 1995. Neuromorphic analog VLSI. *Ann. Rev. Neurosci.* **18**:255-281.

Draper, N.R., H. Smith. 1998. *Applied Regression Analysis*. Wiley, New York.

Džeroski, S., L. Todorovski. 1993. Discovering dynamics. In *Proceedings of the 10th International Conference on Machine Learning*, pp. 97-103.

Džeroski, S., L. Todorovski. 1995. Discovering dynamics: From inductive logic programming to machine discovery. *J. Intelligent Information Systems* **4**:89-108.

El-Turky, F.M., R.A. Nordin. 1986. BLADES: An expert system for analog circuit design. In *Proceedings of IEEE ISCAS*, pp. 552-555.

Ellner, S.P., J. Guckenheimer. 2006. *Dynamic Models in Biology*. Princeton University Press, Princeton, NJ.

Enke, D., S. Thawornwong. 2005. The use of data mining and neural networks for forecasting stock market returns. *Expert Systems with Applications* **29**:927-940.

Evans, J., A. Rzhetsky. 2010. Machine science. *Science* **329**:399-400.

Floreano, D., C. Mattiussi. 2008. *Bio-Inspired Artificial Intelligence: Theories, Methods, and Technologies*. MIT Press, Cambridge, MA.

Fogel, D.B. 1995. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. IEEE Press, Piscataway, NJ.

Fogel, L.J., A.J. Owens, M.J. Walsh. 1966. *Artificial Intelligence through Simulated Evolution*. John Wiley, New York.

- Gabashvili, I.S., B.H.A. Sokolowski, C.C. Morton, A.B.S. Giersch. 2001. Ion channel gene expression in the inner ear. *J. Assoc. Res. Otolaryngol.* **8**:305-328.
- Galan, R.F., N. Fourcaud-Trocme, G.B. Ermentrout, N.N. Urban. 2006. Correlation-induced synchronization of oscillations in olfactory bulb neurons. *J. Neurosci.* **26**:3646-3655.
- Gao, Y., B.W. Strowbridge. 2009. Long-term plasticity of excitatory inputs to granule cells in the rat olfactory bulb. *Nat Neurosci.* **12**:731-733.
- Gouesbet, G. 1991. Reconstruction of the vector fields of continuous dynamical systems from numerical scalar time series. *Phys. Rev. A* **43**:5321-5331.
- Grassberger, P., I. Procaccia. 1983. Measuring the strangeness of strange attractors. *Physica D* **9**:189-208.
- Gray, G.J., D.J. Murray-Smith, Y. Li, K.C. Sharman. 1997. Nonlinear model structure identification using genetic programming. In *Proceedings of the 2nd International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications*, pp. 308-313.
- Gray, G.J., D.J. Murray-Smith, Y. Li, K.C. Sharman, T. Weinbrenner. 1998. Nonlinear model structure identification using genetic programming. *Control Eng. Pract.* **6**:1341-1352.
- Grewal, M.S., K. Glover. 1976. Identifiability of linear and nonlinear dynamical systems. *IEEE Trans. Auto. Ctrl.* **21**:833-837.
- Gurkiewicz, M., A. Korngreen. 2007. A numerical approach to ion channel modeling using whole-cell voltage-clamp recordings and a genetic algorithm. *PLOS Comp. Biol.* **3**:1633-1647.
- Güven, A. 2009. Linear genetic programming for time-series modeling of daily flow rate. *J. Earth Syst. Sci.* **118**:137-146.

Harbison, S.P., G.L. Steele. 2002. *C: A Reference Manual, Fifth Edition*. Prentice Hall, Upper Saddle River, NJ.

Harjani, R., R.A. Rutenbar, L.R. Carley. 2006. OASYS: A framework for analog circuit synthesis. *Transactions on Computer-Aided Design of Integrated Circuit Systems* **8**:1247-1266.

Harris-Warrick, R.M., R.E. Flamm. 1987. Multiple mechanisms of bursting in a conditional bursting neuron. *J. Neurosci.* **7**:2113-2128.

Hassan, M. 2007. A fusion model of HMM, ANN and GA for stock market forecasting. *Expert Systems with Applications* **33**:171-180.

Häusser, M. 2000. The Hodgkin-Huxley theory of the action potential. *Nat. Neurosci.* **3**:1165.

Hawkes, A.G. 2003. Stochastic modeling of single ion channels. In *Computational Neuroscience: a Comprehensive Approach*, ed. by J. Feng. CRC Press, London. pp. 126-152.

Hedrich, L., E. Barke. 1995. A formal approach to nonlinear analog circuit verification. In *Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design*, pp. 123-127.

Hengl, S., C. Kreutz, J. Timmer, T. Maiwald. 2007. Data-based identifiability analysis of non-linear dynamical models. *Bioinformatics* **23**:2612-2618.

Henzler-Wildman, K.A., M. Lei, V. Thai, S.J. Kerns, M. Karplus, D. Kern. 2007. A hierarchy of timescales in protein dynamics is linked to enzyme catalysis. *Nature* **450**:913-916.

Hille, B. 2001. *Ion Channels of Excitable Membranes*. Sinauer Associates, Sunderland, MA.

Hillis, W.D. 1990. Co-evolving parasites improve simulated evolution as an

optimization procedure. *Physica D* **42**:228-234.

Hines, M.L., N.T. Carnevale. 2001. NEURON: A tool for neuroscientists. *The Neuroscientist* **7**:123-135.

Hodgkin, A.L., A.F. Huxley. 1952a. Currents carried by sodium and potassium ions through the membrane of the giant axon of Loligo. *J. Phys.* **116**:449-472.

Hodgkin, A.L., A.F. Huxley. 1952b. The components of membrane conductance in the giant axon of Loligo. *J. Phys.* **116**:473-496.

Hodgkin, A.L., A.F. Huxley. 1952c. The dual effect of membrane potential on sodium conductance in the giant axon of Loligo. *J. Phys.* **116**:497-506.

Hodgkin, A.L., A.F. Huxley. 1952d. A quantitative description of membrane current and its application to conduction and excitation in nerve. *J. Phys.* **117**:500-544.

Hodgkin, A.L., A.F. Huxley, B. Katz. 1952. Measurement of current-voltage relations in the membrane of the giant axon of loligo. *J. Phys.* **116**, 424-448.

Holland, J.H. 1975. *Adaptation in natural and artificial systems*. University of Michigan Press, Ann Arbor, MI.

Hong, X., R.J. Mitchell, S. Chen, C. J. Harris, K. Li, G.W. Irwin. 2008. Model selection approaches for non-linear system identification: a review. *Int. J. Sys. Sci.* **39**:925-946.

Hoppensteadt, F.C., C.S. Peskin. 2002. *Modeling and Simulation in Medicine and the Life Sciences*. Springer-Verlag, New York.

Iba, H. 2008. Inference of differential equation models by genetic programming. *Inf. Sci.* **178**:4453-4468.

Iba, H., H. de Garis, T. Sato. 1994. Genetic programming with local hill-climbing. In

Proceedings of PPSN III, pp. 302-311.

Iniewski, K. (ed.) 2008. *VLSI Circuits for Biomedical Applications*. Artech House, Norwood, MA.

Inoue, T., B.W. Strowbridge. 2008. Transient activity induces a long-lasting increase in the excitability of olfactory bulb interneurons. *J. Neurophys.* **99**:187-199.

Jakobi, N., P. Husbands, I. Harvey. 1995. Noise and the reality gap: The use of simulation in evolutionary robotics. In *Proceedings of 3rd Eur. Conf. Art. Life*, pp. 704-720.

Jara, E.C. 2011. Long memory time series forecasting by using genetic programming. *Genet. Prog. Evol. Mach.* **12**:429-456.

Jones, C.K.R.T. 2001. *Multiple-Time-Scale Dynamical Systems*. Springer-Verlag, New York.

Julier, S.J., J.K. Uhlmann. 1997. A new extension of the Kalman filter to nonlinear systems. In *Proceedings of Int. Symp. Aerospace/Defense Sensing, Simul. and Controls*, pp. 182-193.

Julier, S.J., J.K. Uhlmann, H.F. Durrant-Whyte. 2000. A new method for the nonlinear transformation of means and covariances in filters and estimators. *IEEE Trans. Automat. Contr.* **45**:477-482.

Kaboudan, M.A. 2000. Genetic programming prediction of stock prices. *Comput. Econ.* **16**:207-236.

Kandel, E.R., J.H. Schwartz, T.M. Jessell. 2000. *Principles of Neural Science*. McGraw-Hill, New York.

Kashtan, N., U. Alon. 2005. Spontaneous evolution of modularity and network motifs. *PNAS* **102**:13773-13778.

Kennel, M.B., R. Brown, H.D. Abarbanel. 1992. Determining embedding dimension for phase-space reconstruction using a geometrical construction. *Phys. Rev. A* **45**:3403-3411.

Keymeulen, D., R.S. Zebulum, Y. Jin, A. Stoica. 2000. Fault-tolerant evolvable hardware using field-programmable transistor arrays. *IEEE Trans. Reliability* **49**:305-316.

Kherlopian, A.R., F.A. Ortega, D.J. Christini. 2011. Cardiac myocyte model parameter sensitivity analysis and model transformation using a genetic algorithm. In *Proceedings of GECCO '11*, pp. 755-758.

Kim, K.-J. 2003. Financial time series forecasting using support vector machines. *Neurocomputing* **55**:307-319.

Kim, K.-J., A. Wong, A., H. Lipson. 2010. Automated synthesis of resilient and tamper evident analog circuits without a single point of failure. *Genet. Program. Evolvable Mach.* **11**:35-59.

King, R.D., K.E. Whelan, F.M. Jones, P.G.K. Reiser, C.H. Bryant, S.H. Muggleton, D.B. Kell, S.G. Oliver. 2004. Functional genomic hypothesis generation and experimentation by a robot scientist. *Nature* **427**:247-252.

King R.D., J. Rowland, S.G. Oliver, M. Young, W. Aubrey, E. Byrne E, M. Liakata, M. Markham, P. Pir, L.N. Soldatova, A. Sparkes, K.E. Whelan, A. Clare. 2009. The automation of science. *Science* **324**:85-89.

Kinnear Jr., K.E. 1993a. Evolving a sort: Lessons in genetic programming. In *Proceedings of 1993 Int. Conf. Neur. Net.*, pp. 881-888.

Kinnear Jr., K.E. 1993b. Generality and difficulty in genetic programming: Evolving a sort. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pp. 287-294.

Kissinger, L.D., N.L. Stemm. 1986. Determination of the antileukemia agents

cytarabine and azacitidine and their respective degradation products by high-performance liquid chromatography. *J. Chromatography* **353**:309-318.

Klein, R. 2001. Asymptotic adaptive methods for multi-scale problems in fluid mechanics. *J. Engr. Math* **39**:261-343.

Knuth, D. 1973. *The Art of Computer Programming, Volume 3: Searching and Sorting*. Addison-Wesley, Reading, MA.

Koh, H.Y., C.H. Sequin, P.R. Gray. OPASYN: A compiler for CMOS operational amplifiers. *IEEE Trans. Comput.-Aided Des.* **9**:113-125.

Kohno, T., K. Aihara. 2007. Bottom-up design of Class 2 silicon nerve membrane. *J. Int. Fuzzy Sys.* **18**:465-475.

Kordaki, M., M. Miatidis, G. Kapsampelis. 2008. A computer environment for beginners' learning of sorting algorithms: Design and pilot evaluation. *Computers and Education* **51**:708-723.

Kouchakpour, P., A. Zaknich, T. Braeunl. 2009. A survey and taxonomy of performance improvement of canonical genetic programming. *Knowledge and Information Systems* **21**:1-39.

Koza, J.R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA.

Koza, J.R. 1994. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA.

Koza, J.R., F.H. Bennett III, D. Andre, M.A. Keane. 1996a. Automated WYWIWYG design of both the topology and component values of electrical circuits using genetic programming. In *Proceedings of 1996 IEEE International Conf. Evol. Comp.*, pp. 1-10.

Koza, J.R., F.H. Bennett III, D. Andre, M.A. Keane. 1996b. Four problems for which

a computer program evolved by genetic programming is competitive with human performance. In *Proceedings of the First Annual Conference on Genetic Programming*, pp. 28-31.

Koza, J.R., F.H. Bennett III, D. Andre, M.A. Keane. 1996c. Automated design of both the topology and sizing of analog electrical circuits using genetic programming. In *Proceedings of Artificial Intelligence in Design '96*, pp. 151-170

Koza, J.R., F.H. Bennett III, D. Andre, M.A. Keane, F. Dunlap. 1997. Automated synthesis of analog electrical circuits by means of genetic programming. *IEEE Trans. Evol. Comput.* **1**:109-128.

Koza, J.R., F.H. Bennett III, D. Andre, M.A. Keane. 1999. *Genetic Programming III: Darwinian Invention and Problem Solving*. MIT Press, Cambridge, MA.

Koza, J.R., M.A. Keane, M.J. Streeter, W. Mydlowec, J. Yu, G. Lanza, D. Fletcher. 2003. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, Norwell, MA.

Krogh, A., J. Vedelsby. 1995. Neural network ensembles, cross validation, and active learning. In *Advances in Neural Information Processing Systems*. MIT Press, Cambridge, MA. pp. 231-238.

Kruiskamp, W., D. Leenaerts. 1995. DARWIN: CMOS opamp synthesis by means of a genetic algorithm. In *Proceedings of the 32nd Design Automation Conference*, pp. 433-438.

Langley, P. 1981. Data-driven discovery of physical laws. *Cog. Sci.* **5**:31-54;

Langley, P., J. Sanchez, L. Todorovski, S. Džeroski. 2002. Inducing process models. In *Proceedings of the 19th Int. Conf. Mach. Learn*, pp. 347-354

Langley, P., H.A. Simon, G. Bradshaw, J.M. Zytkow. 1987. *Scientific Discovery: Computational Exploration of the Creative Processes*. MIT Press, Cambridge, MA.

Lapicque, L. 1907. Recherches quantitatives sur l'excitation électrique des nerfs traitée comme une polarisation. *J. Physiol. Pathol. Gen.* **9**:620-635.

Lawlor, F.D. 1973. MFIT: A multiple nonlinear curve fitting technique. *IBM Poughkeepsie Laboratory Technical Report* TR 00.2482.

Le Pourhiet, A., M. Corrège, D. Caruana. 2003. Control of self-oscillating systems. *IEEE Proc. Control Theory Appl.* **150**:599-610.

Lee, C.H., H.G. Othmer. 2009. A multi-time-scale analysis of chemical reaction networks: I. Deterministic systems. *J. Math. Biol.* **60**:387-450.

Lindsay, R.K., B.G Buchanan, E.A. Feigenbaum, J. Lederberg. 1980. *Applications of Artificial Intelligence for Organic Chemistry: The DENDRAL Project*. McGraw-Hill, New York.

Ljung, L. 1999. *System Identification-Theory For the User, 2nd edition*. Prentice Hall, Upper Saddle River, N.J.

Lockery, S.R., G. Wittenberg, W.B. Kristan, G.W. Cottrell. 1989. Function of identified interneurons in the leech elucidated using neural networks trained by back-propagation. *Nature* **340**:468-471.

Lohn, J.D., S. Colombano. 1998. Automated analog circuit synthesis using a linear representation. In *Proceedings of the Second International Conference on Evolvable Systems: From Biology to Hardware*, pp. 125-133.

Lorenz, E.N. 1963. Deterministic non-periodic flow. *J. Atmos. Sci.* **20**:130-141.

Luke, S. 2011. *Essentials of Metaheuristics*.
<http://cs.gmu.edu/~sean/book/metaheuristics/>

Mamani, G., J. Becedas, V.F. Battle, H. Sira-Ramírez. 2008. Algebraic observer to estimate unmeasured state variables of DC motors. *Eng. Lett.* **16**:248-255.

Mallat, S. 1989. A theory for multiresolution signal decomposition: The wavelet representation. *IEEE Trans Pattern Analysis and Machine Intelligence* **7**:674-693.

Mattiussi, C., D. Floreano. 2007. Analog genetic encoding for the evolution of circuits and networks. *IEEE Trans. Evol. Comp.* **11**:596-607.

McMillan, L.G. 2001. *Options as a Strategic Investment*. Penguin Putnam, New York.

Menon, V., N. Spruston, W.L. Kath. 2009. A state-mutating genetic algorithm to design ion-channel models. *PNAS* **106**:16829-16834.

Mitchell, M. 1996. *An introduction to genetic algorithms*. MIT Press, Cambridge, MA.

Montana, D.J. 1995. Strongly typed genetic programming. *Evol. Comp.* **3**:199-230.

Mulloy, B.S., R.L. Riolo, R.S. Savit. 1996. Dynamics of genetic programming and chaotic time series prediction. In *Genetic Programming 1996: Proceedings of the First Annual Conference*, pp. 166-174.

Mydlowec, W., J.R. Koza. 2000. Use of time-domain simulations in automatic synthesis of computational circuits using genetic programming. In *Proceedings GECCO '00*, pp. 187-197.

Neely, C., P. Weller, R. Dittmar. 1997. Is technical analysis in the foreign exchange market profitable? A genetic programming approach. *J. Financial. Quantitative Analysis* **32**:405-426.

Noble, D. 2008. Computational models of the heart and their use in assessing the actions of drugs. *J. Pharm. Sci.* **107**:107-117.

Nocedal, J., S.J. Wright. 2006. *Numerical Optimization*. Springer-Verlag, New York.

O'Neill, M., C. Ryan. 2003. *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Kluwer Academic Publishers, Norwell, MA.

Oakley, H. 1994. Two scientific applications of genetic programming: Stack filters and non-linear equation fitting to chaotic data. In *Advances Genetic Programming*, ed. by K.E. Kinneer Jr. MIT Press, Cambridge, MA. pp. 369-389

Packard, N.H., J.P. Crutchfield, J.D. Farmer, R.S. Shaw. 1980. Geometry from a time series. *Phys. Rev. Ltrrs.* **45**:712-716.

Palsson, B.O. 1987. On the dynamics of the irreversible Michaelis-Menten reaction mechanism. *Chem. Eng. Sci.* **42**:447-458.

Pandolfi, J.M. 2002. Coral community dynamics at multiple scales. *Coral Reefs* **21**:13-23.

Peng, H.C., F. Long, C. Ding. 2005. Feature selection based on mutual information: Criteria of max-dependency, max-relevance, and min-redundancy. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **8**:1226-1238.

Pnevmatikos, S.N. (ed.) 1985. *Singularities & Dynamical Systems*. Elsevier, Amsterdam.

Press, W.H., B.P. Flannery, S.A. Teukolsky, W.T. Vetterling. 2007. *Numerical Recipes: The Art of Scientific Computing, 3rd edition*. Cambridge University Press, Cambridge, UK.

Qian, L., H. Wang, E.R. Dougherty. 2008. Inference of noisy nonlinear differential equation models for gene regulatory networks using genetic programming and Kalman filtering. *IEEE Trans. Signal Process.* **56**:3327-3339.

Rabiner, L.R. 1989. A tutorial on hidden Markov models and selected applications in speech recognition. *Proc IEEE* **77**:257-286.

Ramsey, J.B. 2002. Wavelets in economics and finance: Past and future. *Studies in Nonlinear Dynamics Econometrics* **6**:1-27.

Raue, A., V. Becker, U. Klingmüller, J. Timmer. 2010. Identifiability and observability analysis for experimental design in nonlinear dynamical models. *Chaos* **20**:045105.

Rodriguez-Vázquez, K., P.J. Fleming. 2005. Evolution of mathematical models of chaotic systems based on multiobjective genetic programming. *Knowl. Inf. Syst.* **8**:235-256.

Routray, K., G. Deo. 2005. Kinetic parameter estimation for a multiresponse nonlinear reaction model. *AIChE J.* **51**:1733-1746

Russell, S., P. Norvig. 2009. *Artificial Intelligence: A Modern Approach, 3rd edition*. Prentice Hall, Upper Saddle River, NJ.

Rutenbar, R.A. 1993. Analog design automation: Where are we? Where are we going? In *Proceedings of the 15th IEEE CICC*, pp. 13.1.1-13.1.8.

Rutenbar, R.A., G.G.E. Gielen, B.A. Antao. 2002. *Computer-Aided Design of Analog Integrated Circuits and Systems*. Wiley-IEEE Press, New York.

Ryan, C., J.J. Collins, M. O'Neill. 1998. Grammatical evolution: Evolving programs for an arbitrary language. In *Proceedings of EuroGP 1998*, pp. 83-96.

Sakamoto, E. and H. Iba. 2001. Inferring a system of differential equations for a gene regulatory network by using genetic programming. In *Proceedings of the 2001 Congress on Evolutionary Computation*, pp. 720-726.

Sapargaliyev, Y.A, T.G. Kalganova. 2012. Open-ended evolution to discover analogue circuits for beyond conventional applications. *Genet. Prog. Evol. Mach.* **13**:411-443.

Schaumann, R., M.E. Van Valkenburg. 2001. *Design of Analog Filters*. Oxford University Press, New York.

Schmidt, M.D., H. Lipson. 2007. Comparison of tree and graph encodings as function of problem complexity. In *Proceedings of GECCO '07*, pp. 1674-1679.

Schmidt, M.D., H. Lipson. 2008. Coevolution of fitness predictors. *IEEE Trans. Evol. Comp.* **12**:736-749.

Schmidt, M.D., H. Lipson. 2009. Distilling free-form natural laws from experimental data. *Science* **324**:81-85.

Schmidt, M.D., H. Lipson. 2010a. Age-fitness Pareto optimization. In *Proceedings GECCO '10*, pp. 543-544.

Schmidt, M.D., H. Lipson. 2010b. Age-fitness Pareto optimization. *Genetic Programming Theory and Practice* **8**:129-146.

Schmidt, M.D., H. Lipson. 2010c. Predicting solution rank to improve performance. In *Proceedings GECCO '10*, pp. 949-956.

Schmidt, M.D., R.R. Vallabhajosyula, J.W. Jenkins, J.E. Hood, A.S. Soni, J.P. Wikswo, H. Lipson. 2011. Automated refinement and inference of analytical models for metabolic networks. *Phys. Biol.* **8**:055011.

Schwabacher, M., P. Langley. 2001. Discovering communicable scientific knowledge from spatio-temporal data. In *Proceedings of ICML 2001*, pp. 489-496.

Sharp, K.V., R.J. Adrian. 2004. Transition from laminar to turbulent flow in liquid filled microtubes. *Exp. Fluids* **36**:741-747.

Shirakawa, S., T. Nagao. 2007. Evolution of sorting algorithm [sic] using graph structured program evolution. In *Proceedings of IEEE Int. Conf. Systems, Man and Cybernetics*, pp. 1256-1261.

Sicard, G., G. Bouvier, V. Fristot, A. Lelah. 1999. An adaptive bio-inspired analog silicon retina. In *Proceedings of the 25th European Solid-State Circuit Conference*, pp. 306-309.

Simoni, M.F., G.S. Cymbalyuk, M.E. Sorensen, R.L. Calabrese, S.P. DeWeerth. 2004. A multiconductance silicon neuron with biologically matched dynamics. *IEEE Trans. Biom. Eng.* **51**:342-354.

Sjöberg, J., Q. Zhang, L. Ljung, A. Benveniste, B. Delyon, P.-Y. Glorennec, H. Hjalmarsson, A. Juditsky. 1995. Nonlinear black-box modeling in system identification: A unified overview. *Automatica* **31**:1691-1724.

Smith, L.S. 2010. Neuromorphic systems: Past, present and future. *Adv. Exp. Med. Biol.* **657**:167-82.

Spector, L., J. Klein, M. Keijzer. 2005. The Push3 execution stack and the evolution of control. In *Proceedings GECCO '05*, pp. 1689-1696.

Squillero, G. 2005. MicroGP: An evolutionary assembly program generator. *Genetic Prog. Evol. Mach.* **6**:247.

Stanley, K.O., Miikkulainen, R. 2002. Evolving neural networks through augmenting topologies. *Evol. Comp.* **10**:99-127.

Stewart, W.E., M. Caracotsios, J. P. Sørensen. 1992. Parameter estimation from multiresponse data. *AICHE J.* **38**:641-650.

Strogatz, S.H. 2001. Exploring complex networks. *Nature* **410**:268-276.

Strogatz, S.H. 2007. The end of insight. In *What is Your Dangerous Idea? Today's Leading Thinkers on the Unthinkable*, ed. by J. Brockman. Harper Perennial, New York.

Subramani, P., R. Sahu, S. Verma. 2006. Feature selection using the Haar wavelet

power spectrum. *BMC Bioinformatics* **7**:432.

Szalay, A., J. Gray. 2006. 2020 Computing: Science in an exponential world. *Nature* **440**:413-414.

Takens, F. 1981. Detecting strange attractors in turbulence. In *Dynamical Systems and Turbulence, Lecture Notes in Mathematics*, ed. by D.A. Rand, L.S. Young. Springer-Verlag, Berlin.

Tay, F.E.H., L. Shen. 2002. Economic and financial prediction using rough sets model. *European Journal of Operational Research* **141**:641-659.

Todorovski, L., S. Džeroski. 1997. Declarative bias in equation discovery. In *Proceedings of ICML '97*, pp. 376-384.

Torresen, J. 2002. A scalable approach to evolvable hardware. *Genet. Prog. Evol. Mach.* **3**:259-282.

Turner, S.D., M.D. Ritchie, W.S. Bush. 2009. Conquering the needle-in-a-haystack: How correlated input variables beneficially alter the fitness landscape for neural networks. *LNCS* **5483**:80-91.

Vajda, S., H. Rabitz. 1994. Identifiability and distinguishability of general reaction systems. *J. Phys. Chem.* **98**:5265-5271.

Verhulst, F. 2005. *Methods and Applications of Singular Perturbations: Boundary Layers and Multiple Timescale Dynamics*. Springer-Verlag, New York.

Vittoz, E.A. 1985. The design of high-performance analog circuits on digital CMOS chips. *IEEE J Solid-State Cir* **20**:657-665.

Voss, H.U., J. Timmer, J. Kurths. 2004. Nonlinear dynamical system identification from uncertain and indirect measurements. *Int. J. Bifurcation Chaos* **14**:1905-1933.

Wagner, N., Z. Michalewicz, M. Khouja, R.R. McGregor. 2007. Time series forecasting for dynamic environments: The DyFor genetic program model. *IEEE Trans. Evol. Comp.* **11**:433-452.

Walter, É., L. Pronzato. 1997. *Identification of Parametric Models From Experimental Data*. Springer, New York.

Waltz, D., B.G. Buchanan. 2009. Automating science. *Science* **324**:43-44.

Washio, T., H. Motoda, Y. Niwa. 2000. Enhancing the plausibility of law equation discovery. In *Proceedings of ICML '00*, pp. 1127-1134.

Wei, H.-L., S. A. Billings. 2008. Model structure selection using an integrated forward orthogonal search algorithm assisted by squared correlation and mutual information. *Int. J. Mod. Ident. Ctrl.* **3**:341-356.

Whigham, P.A. 1995. Grammatically-based genetic programming. In *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pp. 33-41

Winkler, S., M. Affenzeller, S. Wagner. 2005. New methods for the identification of nonlinear model structures based upon genetic programming techniques. *J. Syst. Sci.* **31**:5-14.

Yan, R., Y. Liu, R.X. Gao. 2010. Correlation dimension analysis: A non-linear time series analysis for data processing. *IEEE Instrumentation and Measurement Magazine* **Dec 2010**:19-25.

Yoshida, T., L.E. Jones, S.P. Ellner, G.F. Fussmann, N.G. Hairston Jr. 2003. Rapid evolution drives ecological dynamics in a predator-prey system. *Nature* **424**:303-306.

Zebulum, R.S., M.A. Pacheco, M. Vellasco. 1998. Comparison of different evolutionary methodologies applied to electronic filter design. In *Proceedings of IEEE World Congress on Computational Intelligence*, pp. 434-439.