Prediction in Projection: Computer Performance Forecasting, a Dynamical Systems Approach

Joshua T. Garland
University of Colorado at Boulder, joshua.garland@colorado.edu

Follow this and additional works at: https://scholar.colorado.edu/appm_gradetds
Part of the Numerical Analysis and Computation Commons

Recommended Citation
https://scholar.colorado.edu/appm_gradetds/13

This Thesis is brought to you for free and open access by Applied Mathematics at CU Scholar. It has been accepted for inclusion in Applied Mathematics Graduate Theses & Dissertations by an authorized administrator of CU Scholar. For more information, please contact cuscholaradmin@colorado.edu.
PREDICTION IN PROJECTION: COMPUTER PERFORMANCE FORECASTING, A DYNAMICAL SYSTEMS APPROACH

by

J. T. Garland

B.S., Mesa State College, 2009

A thesis submitted to the
Faculty of the Graduate School of the
University of Colorado in partial fulfillment
of the requirements for the degree of
Master of Science
Department of Applied Mathematics
2011
This thesis entitled:
PREDICTION IN PROJECTION: COMPUTER PERFORMANCE FORECASTING, A DYNAMICAL
SYSTEMS APPROACH
written by J. T. Garland
has been approved for the Department of Applied Mathematics

____________________
Elizabeth Bradley

____________________
Dr. James Meiss

____________________
Dr. Bob Easton

Date ___________________

The final copy of this thesis has been examined by the signatories, and we find that both the content and
the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.
Recent work in the field of dynamical systems provides evidence that computer systems are nonlinear-deterministic dynamical systems. This implies the existence of a deterministic update rule, which, in turn, implies the existence of a deterministic forecasting rule for the state variables of a running computer. Even a short-term prediction of these quantities, if accurate, could be effective in tailoring system resources on-the-fly to the dynamics of a computing application. For example, a good prediction of processor load could allow a computer to increase its energy efficiency by dynamically turning off unused CPUs, and then turning them back on based on the programs predicted needs.

To explore this, I use a custom measurement infrastructure, delay-coordinate embedding and nonlinear time-series analysis to forecast processor load and cache performance of a set of simple C programs running on an Intel Core2® Duo. This proved to be quite effective. However, the use of traditional embedding techniques ‘on the fly’ is impractical due to the time required to correctly perform the processing and post-processing of the data. My alternative to this is to use arbitrary low-dimensional projections. While this is not consistent with the requirements in the current literature, recent work by Mischaikow suggests that this alternative might work. I verified this conjecture, showing that forecasts based on two-dimensional projections are largely as effective as strategies that use the full embedded dynamics. This is in contrast to the current view in the nonlinear dynamics community that a one-to-one delay map is sufficient for successful prediction using delay coordinate embedding. My results suggest that this may not be a necessary condition. The success of the projection-based forecasting schemes brings into questions the need for full topological conjugacy in forecasting schema. The results presented here suggest ways of improving computer design at a systems level; they also provide evidence to support the use of semi-conjugacies in forecasting schemes.
Dedication

This thesis is dedicated to my Mom, whose love and support has picked me up and kept me going even in times that I was ready to throw in the towel. Thank you, Mom, for turning around even from Idaho Springs, calling at just the right time, and always saying just what I needed to hear.
Acknowledgements

I would like to thank my advisor, Elizabeth Bradley. She was the first person to expose me to the amazing and vast world of Chaos and has nurtured every step of my journey thus far. Her advice, constructive criticism and patience have made her an incredible thesis advisor.

Grammy and Poppy, thank you so very much for my home away from home. Without your loving reprieve, french toast and flank steak, I would not have survived. Thank you for being the best grandparents I could ever ask for.

My siblings. Jessica: “Both within the family and without, our sisters hold up our mirrors: our images of who we are and of who we can dare to be.” In this chaotic world, you always showed me an image of myself that was far more vast than anything I could have imagined. Benjie and Chunks: “Even a fortress cannot stand in the way of united brothers.” No matter what I face in life I know you both will always have my back and will have an adventure ready when life gets too chaotic.

Kevin, thank you for being an incredible older brother and role model. I always have and always will look up to you. Dayna, without you being willing to give up Christmas I would not even be in graduate school. Thank you for all your patience and diligence in both editing of application essays and graphic design.

I would like to thank Zach Alexander, who spent countless hours patiently answering all my questions and helping me to find my place in this research group. There is no way I could have delved into such a vast subject without such a patient mentor, colleague and friend.

Dr. Bob Easton and Dr. James Meiss for being willing to serve on my committee, as well as for your feedback throughout the process of developing my thesis.
# Contents

## Chapter

1 Introduction ................................................. 1

2 Theoretical Background: Embedology, and Nonlinear Forecast Models 3

2.1 Delay Coordinate Embedding .................................. 3

2.1.1 Motivation for the Use of State Space Embedding............ 3

2.1.2 Theoretical Treatment of Embedding Parameters ............... 7

2.1.3 Numerical Methods for Choosing the Time Delay Parameter ..... 9

2.1.4 Numerical Methods for Choosing the Embedding Dimension $m$ ... 12

2.2 Prediction Methods .............................................. 14

2.2.1 Adaptation of Lorenz Method of Analogues ................... 15

2.2.2 $k$-Ball Adaptation of SLMA .................................. 17

2.3 Discussion of Prediction In Projection .......................... 19

3 Forecasting of Computer Performance Dynamics .................. 21

3.1 The Dynamical System: A Computer ............................ 21

3.1.1 The State Space ............................................... 21

3.1.2 The Map ...................................................... 22

3.1.3 The Specific $\vec{F}_{code}$ I am Studying .................... 24

3.2 Methods: Data Collection, Embedding Techniques and Prediction . 26

3.2.1 Time Series Collection: The Observable ...................... 26
3.2.2 Performance Metrics ......................................................... 28
3.2.3 The Time Series in the Time Domain ............................... 29
3.2.4 Delay Coordinate Embedding ............................................ 31

3.3 Prediction of Computer Performance Dynamics ...................... 32
3.3.1 Forecasting Using Estimated Embedding Parameters ............. 32
3.3.2 Forecasting With Two-Dimensional Projections .................... 39

3.4 Future Work ................................................................. 45

4 Conclusions ................................................................. 48

Bibliography ........................................................................ 49

Appendix

A Thesis Source Code .......................................................... 51

A.1 Matlab Code ................................................................. 51
A.1.1 SLMA Code .............................................................. 51
A.1.2 k-ball SLMA code ..................................................... 52
A.1.3 Preliminary Corrective k-ball SLMA Source Code .......... 54
A.1.4 Matlab Supporting Programs for Both SLMA and k-ball SLMA 56
A.1.5 Data Processing Code ................................................ 58

A.2 C code ............................................................................. 58

A.3 gnuplot & TISEAN scripts ................................................. 58
A.3.1 Mutual Script ............................................................ 58
A.3.2 False-Nearest Script .................................................. 59
Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Collective RMSPE for all Predictions with Full Embeddings</td>
<td>37</td>
</tr>
<tr>
<td>3.2</td>
<td>Collective RMSPE for all Column-Major Prediction Methods</td>
<td>44</td>
</tr>
</tbody>
</table>
Figures

Figure

2.1 Delay coordinate embedding illustration using the Rössler attractor . . . . . . . . . . . . . . . 6
2.2 Increasing $\tau$ to illustrate the effects of the time delay on the reconstruction . . . . . . . . 11
2.3 The mutual information of the synthetic Rössler time series . . . . . . . . . . . . . . . . . . . 12
2.4 An example [7] of a “false-nearest” neighbor in the Hénon Attractor . . . . . . . . . . . . . 14

3.1 A visualization of row-major and column-major matrix traversal . . . . . . . . . . . . . . . . 24
3.2 Row-major and column-major matrix traversals as they occur in memory in C . . . . . . . . 25
3.3 Full time series in the time domain . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
3.4 Detailed view of column-major cache-miss rate and row-major IPC . . . . . . . . . . . . . . . 30
3.5 Mutual and false nearest plot for the four performance traces . . . . . . . . . . . . . . . . . . 31
3.6 Legend for figures that present forecasting results . . . . . . . . . . . . . . . . . . . . . . . . . 32
3.7 Prediction of Column-Major Cache-Miss Rate Using Full Embedding . . . . . . . . . . . . . . 34
3.8 Prediction of Column-Major IPC Using Full Embedding . . . . . . . . . . . . . . . . . . . . . 35
3.9 Row-major cache prediction using full embedding . . . . . . . . . . . . . . . . . . . . . . . . . 37
3.10 Row-major IPC prediction using full embedding . . . . . . . . . . . . . . . . . . . . . . . . . 38
3.11 Standard projection cache miss prediction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
3.12 Best projected cache miss prediction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
3.13 Standard projected cache prediction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
3.14 Best projected cache prediction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
Chapter 1

Introduction

Building on recent work [13] that establishes that computer systems can be effectively analyzed using a dynamical systems approach, I use time series methods such as delay-coordinate embedding and the sampled Lorenz method of analogues to forecast processor and memory usage patterns. Even a short-term prediction of these quantities, if accurate, could be effective in tailoring system resources on-the-fly to the dynamics of a computing application. While both memory and processors are growing in power, the demand on them is also growing. As such, efficient management of memory and processor usage is vital in the design and manufacture of tomorrow’s computer systems. This unique use of nonlinear time series analysis to forecast memory management and processor usage will provide innovation in computer system design as well as new applications in the field of dynamical systems.

I monitor several performance metrics during the execution of a simple loop. One such loop initializes an array in row-major order and then column-major order. Several precautions have been taken to ensure that the data collected reflects the observable dynamics and not other underlying attractors. The metrics I most commonly use are cache misses and the number of instructions committed per cycle.

Once the time series has been collected I use delay coordinate embedding to embed the time series, the embedding enables the use of the signals natural temporal flow in the prediction process [17]. The TISEAN (TIme SEries ANalysis) software package [4] provides several routines to embed and analyze time series data. To embed the data it is essential to choose embedding parameters that accurately portray the underlying dynamics. By using standard algorithms I was able to determine estimates of these parameters, which I verified by analyzing dynamical invariants such as correlation sum, entropy and dimension, as well
as checking consistency in maximal Lyapunov exponents and the relation between the correlation entropy and dimension.

My prediction method needs to be simple enough to perform on-the-fly while accurate enough to provide a true picture of the future dynamics. I am currently using two forecast schemes. The first method is an adaptation of the Lorenz method of analogues [5]. When a prediction of the signal is needed the nearest neighbor of the final point is found. The prediction is then taken to be the image of the nearest neighbor. The second method I use is a variant of the first that uses information from the final point’s $k$ nearest neighbors. To make a prediction, the image of the $k$ nearest neighbors are averaged to synthesize the next point in the time series.

Several interesting problems have arisen in these preliminary studies. The most important is the validity of using low-dimensional embeddings to predict high-dimensional dynamics. Using delay coordinate embedding requires a great deal of human interpretation. However, if an arbitrary low-dimensional embedding can provide a glimpse of the dynamics, then a low dimensional unfolding can be used with minor impact on the forecast error. I have tested this hypothesis on the column-major dynamics and received “accurate” predictions of both the processor load and cache misses.

We will begin with a theoretical treatment of delay coordinate embedding and implementation methods, as well as a discussion of the prediction techniques I will utilize. This discussion will be followed by experimental methodology and prediction results. I will conclude with a discussion of current and future research.
Chapter 2

Theoretical Background: Embedology, and Nonlinear Forecast Models

2.1 Delay Coordinate Embedding

2.1.1 Motivation for the Use of State Space Embedding

The term nonlinearm deterministic dynamical system describes a set $X$ combined with a deterministic nonlinear evolution or update rule $\Phi$. The set $X$ could be as simple as $\mathbb{R}^n$ or a similar geometric manifold, or as abstract as a set of symbols [9]. Elements of the set are referred to as states of the system and the set itself is generally referred to as the state space. The update or evolution rule is a fixed deterministic mapping which gives a unique image to any particular element of the set. It is important to realize that this update rule is deterministic and fixed, meaning that, given a particular state, the next state of the system is completely determined.

The theory of dynamical systems is both vast and rich and contains plethoras of mathematical tools for the analysis of systems which arise both in theory and in practice. For the purposes of this thesis, dynamical systems fall into one of two categories, those which are discrete in time and those that are continuous in time. The former is referred to as a map and denoted by:

$$\vec{x}_{n+1} = \vec{F}(\vec{x}_n), n \in \mathbb{N}$$  \hspace{1cm} (2.1)

the latter is referred to as a flow and is represented by a system of first order ordinary differential equations:

$$\frac{d}{dt} \vec{x}(t) = \vec{f}(\vec{x}(t)), t \in \mathbb{R}^+$$  \hspace{1cm} (2.2)

One fundamental difference that arises in studying dynamical systems is that some of them have
obvious and readily available update rules and state spaces, while others have completely unknown update rules and state spaces. For example, consider the forced, damped simple harmonic oscillator, its motion is completely described by \( m\ddot{x} + b\dot{x} + kx = A\cos(\alpha t) \) and the state space is known to be \( \mathbb{R}^3 \). With this knowledge, given any set of initial conditions, namely a position and a velocity, one can completely determine the future trajectory, i.e., all the future states of this initial condition. In contrast to this simple scenario, consider a computer system, which exhibits all the normal behavior of a dynamical systems[13], but no obvious update rule or state space is readily available to an analyst. So how can I use the methods of dynamical systems to analyze a system when neither \( X \) or \( \Phi \) are readily available?

Since I do not have \((X, \Phi)\), it makes sense to inventory what I do have and see what information can be extracted from it. For example, in studying the dynamical system presented in [13]: a computer, what information is available to me? Using profiling tools, described in Chapter 3, I can observe several aspects of the dynamical system, such as L2 cache misses or how many instructions are committed in a single cycle. These observable quantities give a glimpse of the dynamics but do not tell the full picture. How can I use these glimpses of the dynamics to reconstruct the full picture?

Returning to the example of the force, damped simple harmonic oscillator, i.e.. \( m\ddot{x} + b\dot{x} + kx = A\cos(\alpha t) \). Recall that I can rewrite this second order differential equation as a system of first order differential equations:

\[
\dot{x} = \omega \\
m\dot{\omega} = -b\omega - kx + A\cos(\alpha t)
\]

The reason to recall this is that it illustrates that many, even trivial, dynamical systems are coupled on some fundamental level. By coupled, I mean that the position of the pendulum \( x \) and the angular velocity \( \omega \) are completely dependent on one another. When the update rule is applied to a position in state space, \( x \) directly impacts \( \omega \) and vice versa. With this in mind it might be possible to rebuild a dynamical system from a single observation function as long as the observations smoothly depended on \((x, \omega)\). For example, given a pendulum’s position at regular time intervals but not the angular velocity, would it be possible to estimate the trajectory of the pendulum in state space? My intuition tells me that due to the coupling of
the variables, this reconstruction should be possible, and that intuition is correct.

For the remainder of this section I will need some terminology which I will borrow from [18]. The single measurable quantity discussed earlier will now be referred to as an observation function which is denoted by $h$. Think of the function $h$ as a tool to smoothly sample states of the dynamical system, or at least influences of the dynamical system on the state space. I want to use this observation function $h$ to measure the dynamical system at regular time intervals. The record made by these regular samplings of the observation function is denoted by $x_n$ or sometimes $x_n(t)$ to denote that the measurement is time dependent.

The collection of all $(x_n)_{n=1}^N$ is known as the time series or trace. To be somewhat more formal, think of each $x_n$ in the time series as the result of evaluating the observation function $h$ at the current state $x(t)$ that is $x_n = h(x(t))$ [18]. For some portions of this paper it is important to make distinctions between the direct measurement of state variables and the measurement of the dynamics’ influence on the state space. When this distinction is necessary I may refer to time series reflecting the latter as $(s_n)_{n=1}^N$. This emphasizes that the time series is a simultaneous sampling of several state space variables and not necessarily a direct sampling of an isolated state space variable.

The technique I will use to reconstruct the dynamical systems from these observations is a type of embedding known as delay coordinate embedding. A delay coordinate map attempts to reconstruct a dynamical system from a single observation function, and for this reason it is ideal for reconstructing computer performance dynamics. Takens [20] formally proves that if the original system and the observed quantity are smooth and generic, then the delay coordinate map from an $n$-dimensional smooth compact manifold $M$ to $\mathbb{R}^{2n+1}$ is a diffeomorphism on $M$ [18]. To understand what Takens was talking about I must define what is meant by the delay coordinate map, diffeomorphic, and generic\(^1\).

**Definition 1** (Delay Coordinate Map). If $\Phi$ is a flow on a manifold $M$, $\tau$ is a positive number (called the delay), and $h : M \rightarrow R$ is a smooth function, define the delay coordinate map $F(h, \Phi, \tau, m)(x) = ([x(t) \ x(t + \tau) \ldots \ x(t + (m)\tau)])$ [18].

**Definition 2** (Diffeomorphism, Diffeomorphic). A function $f : M \rightarrow N$ is said to be a diffeomorphism\(^1\) For a complete discussion of genericity see [6].
if it is a $C^1$ bijective correspondence whose inverse is also $C^1$. Two manifolds $M$ and $N$ are said to be 

diffeomorphic if there exists a diffeomorphism $F$ that maps $M$ onto $N$.

**Definition 3 (Generic).** A function is generic in $D$ if there exists a set of parameters in $D$ that is a countable 

intersection of open dense subsets of $D$. A countable intersection of open sets is called a $G_δ$ set. A set is 

called generic if it contains a dense $G_δ$[6].

The Takens' theorem [20] is important because it proves the existence of a topologically conjugate 

dynamics, accessed through the delay coordinate map. That is, if the trajectories of the original dynamical 

system lie on a manifold $M$ then the manifold’s image under the delay coordinate map $F(M)$ has fundamen-

tally the same structure as the unknown system. Moreover, I can analyze the reconstructed system to draw 

mathematically valid conclusions about the original unknown system. This makes the technique extremely 

powerful and appealing to dynamicists.

For my purposes, Takens’ theorem means that given a computer, which is a highly complex nonlinear 

dynamical system with no obvious $(X, \Phi)$, I can completely reconstruct the underlying dynamics up to 

diffeomorphism. For example, say I measure the instructions committed per cycle (IPC), i.e. processor 

efficiency, as long as certain requirements are met on the observation function, namely generic and smooth 

[20], then I can reconstruct the unknown state space up to diffeomorphism. Moreover, this conjugacy yields 

the remarkable fact that what holds true for this embedded state space also holds true for the original space
in a topological sense.

While this is incredible there are still a few caveats I must consider before proceeding. Takens’ theorem [20] introduces two parameters for the delay coordinate map, namely $\tau$ and $m$. The concern with choosing $m$ is that I must have foreknowledge of the systems dimension, $n$. Working under the assumption that the dimension is unavailable to me a priori, which is most likely the case, how may I choose $m > 2n$? Thus, the map $F(h, \Phi, \tau, m)(x) = (\{x(t) \ x(t+\tau) \ \ldots \ x(t+(m)\tau)\})$ seems impossible to construct. The second parameter I must choose is the time delay. In theory, the time delay has no impact on the reconstructed dynamics, but this is only true in the ideal case, i.e., the time series is infinitely long, every entry in the time series is perfect (no error in measurement), and I use infinite precision arithmetic [20]. This ideal scenario is never the case in experimentation, and due to this it turns out a good choice of the time lag is essential to the usefulness of the reconstructed dynamics. The following section discusses these two parameters and methods for estimating them from the time series, in the absence of full knowledge of the original system.

2.1.2 Theoretical Treatment of Embedding Parameters

A common view of a time series is a projection of a dynamical system onto a one-dimensional sub-manifold. To reconstruct these dynamics, it is imperative to “pull apart” this one-dimensional projection, allowing for the system to unfold or inflate back to the manifold’s original shape. The time delay can be thought of as the amount each coordinate is stretched apart during reconstruction. Recall the delay coordinate map from [20] is $\phi(t) = \{x(t) \ x(t+\tau) \ \ldots \ x(t+(m)\tau)\}$, where each coordinate $x(t+i\tau)$ is an entry in the time series i.e., an observation of the dynamical system, and the consecutive coordinates are observations spread apart $\tau$ in time. Hence, $\tau$ is the amount I stretch the one dimensional projection in each direction of reconstruction space. Generally, $\tau$ will be chosen to spread apart each coordinate direction until a desired level of dependence is reached.

The second embedding parameter I must determine is the embedding dimension $m$. According to [20] $m$ must be greater than $2n$, where $n$ is the dimension of the original dynamical system. If this condition is satisfied, along with the above mentioned restrictions on the observation function $h$, the delay coordinate map will describe a dynamical system that is topologically conjugate to the unknown system. The restriction
that \( m > 2n \) was relaxed in [18] to \( m > 2d_{\text{cap}} \), where \( d_{\text{cap}} \) is the box counting or capacity dimension of the system’s attractor. For completeness I will now provide a definition of capacity dimension from [20].

**Definition 4** (Capacity Dimension 1). Let \((M, \rho)\) be a compact metric space. For \( \epsilon > 0 \) define the following: 
\( s(M, \epsilon) \) is the maximal cardinality of a subset of \( M \) such that no two points have distance less than \( \epsilon \); such a set is called a maximal \( \epsilon \)-separated set; \( r(M, \epsilon) \) is the minimal cardinality of a subset of \( M \) such that \( M \) is the union of all the \( \epsilon \)-neighborhoods of its points; such a set is also called a minimal \( \epsilon \)-spanning set. Next define the capacity dimension \( d_{\text{cap}} \) of \( M \) as

\[
d_{\text{cap}} = \liminf_{\epsilon \to 0} \frac{\ln r(M, \epsilon)}{-\ln(\epsilon)} = \liminf_{\epsilon \to 0} \frac{\ln s(M, \epsilon)}{-\ln(\epsilon)} \tag{2.5}
\]

**Remark.** While this definition is analytically very appealing in that you need not consider if the limit exists, in practice however the following definition is more useful.

**Definition 5** (Capacity Dimension 2). The capacity dimension of a compact metric space \( X \) is a real number \( d_{\text{cap}} \) such that if \( n(\epsilon) \) denotes the minimum number of open sets of diameter less than or equal to \( \epsilon \), then \( n(\epsilon) \) is proportional to \( \epsilon^{-D} \) as \( \epsilon \to 0 \), explicitly,

\[
d_{\text{cap}} = - \lim_{\epsilon \to 0^+} \frac{\ln N}{\ln \epsilon} \tag{2.6}
\]

(if this limit exists), where \( N \) is the number of elements forming a finite cover of the relevant metric space \( X \) and \( \epsilon \) is a bound on the diameter of the open covering sets [18].

Takens’ theorem is beautiful from a theoretical standpoint: it guarantees existence of a diffeomorphic embedding relating the unknown state space and a constructible space. On the other hand, it does not give insight into the numerical implementation of the embedding. I know from [20] that the embedding is precisely defined by the delay coordinate map. Regardless, as discussed above, I must choose \( m > 2d_{\text{cap}} \).

Without having an understanding of the dynamics the likelihood of knowing the capacity dimension \textit{a priori} is highly unlikely. Thus, I must numerically estimate both time lag and embedding dimension, since their rigorous derivation is most likely unavailable to me. In light of this discussion, to begin the reconstruction of the time series I must discuss methods for choosing both embedding dimension and time delay such that I recover the original dynamics. Fortunately, there is a rich amount of literature [3, 7, 14, 18] for this.
2.1.3 Numerical Methods for Choosing the Time Delay Parameter

As stated above, the time delay parameter describes the amount that each dimension is stretched apart in the reconstructed state space. In theory the choice of the time delay $\tau$ has no impact on the validity of the reconstructed trajectory. However this assumes that you possess an infinite quantity of noise-free data and use infinite precision arithmetic [18, 20]. In this idealized case the choice of time delay $\tau$ is completely arbitrary as long as $\tau > 0$. In practice, however, experimentalists have shown that the time delay has a strong impact on the usefulness of the embedding [3]. In particular, the amount of information which can be recovered about the original dynamics strongly depends on the choice of $\tau$ [3]. The reason for this is that the finite precision of both measurements and arithmetic can inject linear dependence into coordinates of the embedding space which are not in fact dependent.

Several methods for choosing the most effective time delay have been proposed [3, 18, 20], two of which have become standard practice. The first, proposed in [14], selects the first root of the time series auto-correlation function, the second suggested by Shaw, and which is experimentally and theoretically validated in [3], instead chooses the first minimum of the signal’s mutual information. The foundation of both these methods focus on the components of the delay vector $\vec{X}(t)$, namely $x(t + k\tau)$, with $k \in \{0, \ldots, (m - 1)\}$ being independent. As the auto-correlation function is a measure of the linear dependence of each coordinate, choosing $\tau$ as its first root I am choosing the configuration where the components have zero correlation. That is choosing $\tau$ in this way results in the components of $\vec{X}(t)$ being linearly independent [3, 14]. In this thesis, I use TISEAN’s `corr` function to calculate the auto-correlation function [4].

As proposed in [3], a more refined method for choosing $\tau$ is by selecting the first minimum of the time-delayed mutual information. According to [3] the advantage of this is that mutual information is a measure of the general or mutual dependence of two variables rather than simply the linear dependence, as is the case with the auto-correlation. I can view the mutual dependence of two variables $x$ and $y$ as the information about $x$ gained while observing $y$ or the difference between the uncertainty of $x$ and the remaining uncertainty of $x$ after observing $y$, i.e. the reduction in uncertainty of $x$ gained by observing $y$. In the context of delay-coordinate embedding, this is the information one already possess about the value
of \( x(t + \tau) \) if I know \( x(t) \) \([5]\). Besides, mutual independence of two variables is much more general than linear independence. Mutual information also takes into account nonlinear correlations \([5]\), to which the auto correlation function is blind.

**Definition 6 (Mutual Information).** Define \( H(Q) \) to be the uncertainty of \( q \) in isolation, and \( H(Q \mid S) \) to be the uncertainty of \( q \) given a measurement of \( s \). The we define the mutual information \( I \) as follows:

\[
I(Q,S) = H(Q) - H(Q \mid S)
\]

\[
= H(Q) + H(S) - H(S \mid Q) = I(S,Q)
\]

By choosing a minimum of the mutual information—the approach advocated by \([3]\)—one is attempting to remove as much mutual dependence as possible from each coordinate in the delay vectors \([3]\). The choice of the first minimum as opposed to an arbitrary minima is due to spreading and folding inherent to chaotic and strange attractors. By choosing the first minimum of the delay mutual information I minimize unnecessary stretching of the attractor \([3]\). An \( N \log N \) algorithm to calculate the time delayed mutual information is presented in \([3]\). A variation of this algorithm, \texttt{mutual}, is implemented in the \texttt{TISEAN} \([4]\) package.

**Remark.** While much research has gone into selecting the “optimal delay/lag,” and many new methods have been presented since the prior mentioned literature, e.g. \([3, 14, 18]\), it has been suggested \([5]\) that selection of the lag \( \tau \) is very application specific. For example, for predictions with long horizons a larger lag is optimal, whereas for noise reduction algorithms small delays are more revealing \([5]\).

For my choice of \( \tau \) I primarily use the mutual information approach, choosing the first minima of the time delayed mutual information which I obtained with \texttt{mutual}.\(^2\) As suggested by \([5]\) I begin with this estimate of \( \tau \) but with the final goal of prediction in mind I may increase \( \tau \) in order to expand the prediction horizon.

\(^2\) We also calculated the delay by using the auto-correlation method but received almost identical results.
Example 1. As an example consider the Rössler system defined by:

\[
\begin{align*}
\dot{x} &= -y - z \quad (2.9) \\
\dot{y} &= x + ay \quad (2.10) \\
\dot{z} &= b + z(x - c) \quad (2.11)
\end{align*}
\]

I will construct a time series in the following way. First define the projection operator \( P_i[\vec{X}] = x_i \).

Now using a fourth-order Runge-Kutta integrator, integrate the Rössler system choosing standard parameters, \( a = 0.2, b = 0.2, \) and \( c = 5.7 \), initial condition \([0.0001 \ 0.0001 \ 0.0001]^T\) and a time-step of 0.001. I will call the resulting orbit \( \phi_R \); in the limit as \( t \to \infty \), this orbit will trace out the chaotic Rössler attractor.

Now define the time series \((x_j)_{j=1}^N = P_1[\vec{X}_j]\)—that is, project the orbit down to the \( x \)-axis. I then treat this collection of \( x_j \) values as a synthetic time series and calculate the proper time lag to reconstruct the Rössler attractor using the time series’ mutual information. The results are shown in Figure 2.3.

![Figure 2.3: Increasing \( \tau \) to illustrate the effects of the time delay on the reconstruction](image)

Choosing the first minimum of this plot I can conclude that the time delay should be chosen to be 150. That means that each axis in embedded space should be separated by 1.5 seconds, since during the creation of the time series the time step was 0.01 seconds. Figure 2.2 illustrates the effect on the reconstructed space as I increase the time delay while fixing the embedding dimension. This illustrates that increasing \( \tau \) does in fact stretch the attractor out, allowing it to unfold. Also observe that choosing the time delay to be 150, like the mutual information suggests, I obtain the reconstruction that resembles the original attractor the closest\(^3\).

\(^3\) Note that a \( \tau \) larger than 150 may also give me similar results but choosing the first \( \tau \) for which this occurs is optimal [3].
2.1.4 Numerical Methods for Choosing the Embedding Dimension $m$

As has been stated above, the Takens theorem requires an embedding dimension $m$ which is at least twice as large as the dimension of the original state space. But what is the logic behind this? The time series $x_n$ is fundamentally a projection of the original state space onto a one-dimensional sub-manifold. The purpose of the delay coordinate map is to unfold this one-dimensional projection into a multidimensional state space that represents the original dynamics. The basis of the embedding dimension requirement is to ensure that I allow the one-dimensional projection to unfold “enough”. That is, I need to allow for enough dimensions that the original attractor can be pulled apart to the point where there are no longer self-intersections. Recall, according to [18], that this can be accomplished by choosing $m > 2d_{cap}$.

In practice, even if I knew $d_{cap}$ I do not necessarily want to choose $m$ to be $2d_{cap} + 1$. This is simply a lower bound that theoretically guarantees the embedding is a diffeomorphism. That is, the theorem is
simply a sufficient condition, it could be the case that the attractor completely unfolds prior to embedding dimension $2d_{cap} + 1$. This is the case with the Lorenz attractor that has $d_{cap} = 2.06 \pm 0.01$, Takens theorem [18] would suggest using $m = 7$ ($m > 2d_{cap}$), but in fact this system can be embedded properly by using $m = 3$ [7].

Naively, it may seem that I would just want to choose an “extremely large” $m$ so that the trajectory can unfold. While this problem is not of mathematical interest, it is actually of practical interest. Many of the algorithms for deducing information about dynamical systems scale with $m$, and thus choosing the minimal $m$, is highly sought after. Numerically choosing a minimal embedding dimension is still an open problem, but several algorithms have been presented in the literature [7, 20] for accurately determining an embedding dimension sufficiently large.

One common method for choosing $m$ is to calculate a set of dynamical invariants for increasing embedding dimensions until the invariants stop changing. In [20] it is shown that several quantities, such as capacity dimension, are independent of $m$ after the attractor has unfolded. That is, in theory, once the attractor has unfolded, $m$ no longer affects the particular invariant. However, in practice, noise in the data has larger and larger impact on these calculations as $m$ increases. The underlying idea here is that once the attractor has been unfolded completely, the reconstruction remains diffeomorphic for higher $m$, and thus topological properties preserved by conjugacy will no longer change. So if I find a range of embedding dimension, for which the dynamical invariants stay fairly consistent, I can more safely assume that I have successfully unfolded the attractor. This method is computationally expensive and requires a great deal of post processing and human intuition however. In light of this, if topological conjugacy is vital for the particular application, it is standard to find $m$ with an alternative algorithm and then post facto corroborate this selection by verifying stability of dynamical invariants.

One standard algorithm for choosing $m$, known as the “false nearest neighbor” approach, is presented in detail in [7]. This method calculates the fraction of true nearest neighbors to false nearest neighbors. A false nearest neighbor is a neighbor in a low dimensional projection but not in the next higher dimension. Figure 2.4, taken from [7], illustrates this concept. Notice that for the 1D projection of the Hénon attractor, points $A, B$ and $C$ are all nearest neighbors. In a 2D embedding however, $A$ and $C$ remain neighbors, but $B$
is no longer a neighbor. In the language of this algorithm, that would mean that $B$ is a false nearest neighbor of $A$ and $C$ while $A$ and $C$ are true neighbors. The standard practice is to calculate the false nearest ratio for several successive embedding dimensions and choose the first dimension for which the ratio is less than 10%. This is a rule of thumb, however; depending on noise in the data the arbitrary threshold may not be correct so this is somewhat flexible. In this thesis, I will use the TISEAN implementation of this algorithm (false_nearest) for false nearest analysis.

### 2.2 Prediction Methods

As discussed in [5] the existence of an underlying deterministic evolution rule also implies the existence of a deterministic forecast function. Before I discuss particular prediction methods, I first discuss how I will evaluate the results. In order to evaluate the effectiveness of a given prediction algorithm I need to compare
the predicted signal with a segment of the true signal. In this capacity, I truncate and save the last 10% of the time series, this is referred to as the comparison signal and denoted by $c_n$. The remaining 90% will be used to “train” the prediction algorithm, this is referred to as the training signal. Note, the comparison signal can never be used during the prediction process. The sole purpose of the comparison signal is to validate the effectiveness of the prediction. Using the training signal combined with a prediction algorithm I can predict the 10% of the time series I truncated, this is the prediction signal and denoted by $\hat{p}_n$. According to Kantz et al., the most common error measure used in prediction is the root mean squared prediction error (RMSPE) [5]. RMSPE is defined as

$$
\epsilon = \sqrt{\langle (\hat{p}_n - c_n)^2 \rangle}
$$

where $\langle \cdot \rangle$ denotes average over all $n$. The units of RMSPE are the same as the quantity being measured. A RMSPE of zero means that the prediction method forecasted the signal with perfect accuracy. It is important to remember that RMSPE values may only be used for comparing two prediction methods on the same signal. That is you cannot conclude based on RMSPE that a prediction method predicted one signal better than another, it must only be used as a metric to compare prediction methods on a single signal. The prediction method with the lowest RMSPE will be considered the most accurate forecast scheme for a particular time series.

### 2.2.1 Adaptation of Lorenz Method of Analogues

In 1969 Lorenz proposed a noise reduction and forecasting algorithm [8] now known as the “Lorenz method of analogues” (LMA). LMA assumes a deterministic dynamical system which is discrete in time, it also assumes knowledge of the evolution function $F$, which is not realistic in experimental practice. To describe this method, I will follow [5]. Given a trajectory from a discrete time dynamical system, $(\vec{x}_n)_{n=1}^N$ where $\vec{x}_i = \vec{F}(\vec{x}_{i-1})$ for $i \in \{1, \ldots, N\}$ and where $F$ is the evolution equation of the dynamical system. To predict the next element of the signal, LMA applies the following procedure. Given the final state space observation $\vec{x}_N$, find its nearest neighbor in state space. Call it $\vec{x}_{n_0}$ and assume $n_0 < N$. By continuity of the evolution function, since (by construction) $\vec{x}_{n_0}$ is close in state space to $\vec{x}_N$, it will also be the case that $\vec{x}_{N+1}$ will be close to $\vec{x}_{n_0+1}$. On this basis, LMA makes the prediction $\vec{x}_{N+1} = \vec{x}_{n_0+1}$ which is available
Based on the assumption that \( n_0 < N \).

**Algorithm 2.1** Sampled Lorenz Method of Analogues

**Input:** A set of delay vectors: \( \Phi \)

**Output:** The prediction of the last 10\% of \( \Phi \): Prediction

1. numToTruncate ← numVectors(\( \Phi \)) * 0.1
2. numToExtend ← numToTruncate
3. trainingSignal ← \( \Phi \) \( \Phi \) (numVectors(\( \Phi \))−numToTruncate:end)
4. finalEntry ← trainingSignal(end)
5. nearestNeighbor(finalEntry, trainingSignal) ⇒ neighbor \{ nearestNeighbor returns the nearest neighbor of the finalEntry in a Euclidian sense \}
6. if \( \text{dist}(\text{neighbor}, \text{finalEntry}) = 0 \) then
7. print WARNING: The nearest neighbor was 0 away. Projection error has occurred.
8. end if
9. Prediction(1) ← finalEntry
10. j = indexof(neighbor) \{ j is the index in the training-signal of the nearest neighbor \}
11. for \( i = 2 \) to numToExtend do
12. Prediction(i) ← \( \Phi(j + 1) \) j ← \( \Phi(j + 1) \)
13. end for
14. return Prediction

Unfortunately I can not guarantee that the image of the measurement function is a pure sampling of a state space variable or even a state of the dynamical system, and most likely it is not. In fact the time series is most likely a sampling of many different state space variables aggregated into a single observation. So for generality assume that I have a time series that is a sampling of one or more state space variables. I will call the time series \( (s_n)_{n=1}^N \) in contrast to \( (\vec{x}_n)_{n=1}^N \) to emphasize both that the entries in the time series are scalars and that they are samplings of (possibly several, but not necessarily all) state variables as opposed to actual points in state space. That is I assume that \( s_n = h(\vec{x}_n) \) where \( h \) is the smooth measurement function. Now recall I can reconstruct a manifold representing the trajectory \( (\vec{x}_n)_{n=1}^N \) with \( (s_n)_{n=1}^N \) by means of the delay coordinate map.

[5] shows that I can use the delay coordinate map to predict in the short term by implementing a simple modification of LMA. Taking \( (s_n)_{n=1}^N \) construct delay vectors \( (\vec{S}_n)_{n=1}^M \) where \( \vec{S}_i = [s_i, s_{i+\tau}, \ldots s_{i+(m-1)\tau}, s_{i+m\tau}] \) for all \( i \in \{0, \ldots, M\} \). Consider the final vector \( \vec{S}_M \). In the spirit of the algorithm above, choose the nearest neighbor of \( \vec{S}_M \) call it \( \vec{S}_{n_0} \). Now by the continuity of the measurement function, since \( \vec{S}_M \) is close to \( \vec{S}_{n_0} \) then \( \vec{S}_{M+1} \) is close to \( \vec{S}_{n_0+1} \) and this is the prediction of \( \vec{S}_{M+1} \). Then, from the definition of the delay coordinate mapping I can use \( \vec{S}_{M+1} \) to predict the next entry in the time series by projecting the vector back onto the first coordinate, i.e. \( s_{N+1} = P_1(\vec{S}_{M+1}) \), where \( P_1([x_1, \ldots, x_n]) = x_1 \). The presentation of this
method follows from [5], however, this method seems to originate in [15] and is similar to the method for noise reduction and forecasting in [19]. For the remainder of this text I will refer to this adaptation of LMA as (Sampled LMA) SLMA. My implementation of this algorithm can be found in Appendix A.1.1.

2.2.2 k-Ball Adaptation of SLMA

Given the final vector in the set of delay vectors \( \vec{S}_N \) instead of simply considering this element’s nearest neighbor, choose this element’s \( k \)-nearest neighbors. In this capacity, I construct a \( k \)-(nearest-neighbor)-ball around the final point in state space, call this ball \( B_k(\vec{S}_N) \). When system measurements are noisy, it is equally likely that each point in \( B_k(\vec{S}_N) \) is the true nearest neighbor of \( \vec{S}_N \), so instead of arbitrarily choosing a single neighbor—as is done with the SLMA—I use all points in the \( k \)-ball.

Enumerate \( B_k(\vec{S}_N) \), referring to each element as \( \hat{n}_j \) with \( j = \{1, \ldots, |B_k(\vec{S}_N)|\} \). Note that this enumeration is arbitrary and has no effect on the outcome of the prediction; also, note that \( \hat{n}_j \) is a vector and not a scalar. Now I define the delay vector \( \hat{s}_j \) to be the forward image of \( \hat{n}_j \) under the delay coordinate map. I now define each component \( s_{N+1}^{i(i)} \) of \( \vec{S}_{N+1} \) according to:

\[
\hat{s}_{N+1}^{i(i)} = \frac{1}{|B_k(\vec{S}_N)|} \sum_{j=1}^{|B_k(\vec{S}_N)|} \hat{s}_j^{i(i)} \quad i \in \{1, \ldots, m\} \tag{2.13}
\]

This process allows me to fabricate an image of \( \vec{S}_N \) that utilizes the aggregate behavior of the final points \( k \)-nearest neighbors—as opposed to a single nearest neighbor, as is the case with SLMA. Observe that each point in \( B_k(\vec{S}_N) \) has a forward image and that the cardinality of \( B_k(\vec{S}_N) \) is not zero; so this method is well defined. A forward trajectory of \( \vec{S}_N \) can be constructed by continuing this process. For the rest of this thesis this algorithm will be referred to as \( k \)-Ball SLMA. My implementation of this algorithm can be found in Appendix A.1.2.

Remark. During trajectory construction I do not need to construct \( B_k(\vec{S}_{N+1}) \) after we have synthesized \( \vec{S}_{N+1} \), as \( B_k(\vec{S}_{N+1}) \) is simply the collection of \( \hat{s}_j \). Hence, to construct the trajectory I simply map the \( \hat{s}_j \) forward using the delay coordinate map (assuming each \( \hat{s}_j \) has a forward image, and removing points that do not) and then collapse this new ball using equation 2.2.2 until the prediction horizon is met.
A variant of this algorithm is presented in [5], where it is called zeroth as it is a zeroth order prediction method (as opposed to linear, quadratic, etc.) This method is different from \( k \)-Ball SLMA in that it utilizes fixed-diameter epsilon-balls around the final point, whereas \( k \)-Ball SLMA uses \( B_k(S_N) \) as described above.

**Algorithm 2.2** \( k \)-Ball SLMA

**Input:** A set of delay vectors, and number of neighbors in \( k \)-ball: \( \Phi, k \)

**Output:** The prediction of the last 10% of \( \Phi \): Prediction

1. \( \text{numToTruncate} \leftarrow \text{numVectors}(\Phi) \times 0.1 \)
2. \( \text{numToExtend} \leftarrow \text{numToTruncate} \)
3. \( \text{trainingSignal} \leftarrow \Phi \setminus \Phi(\text{numVectors}(\Phi)-\text{numToTruncate}:\text{end}) \)
4. \( \text{finalEntry} \leftarrow \text{trainingSignal(}\text{end}) \)
5. \( \text{epsilonBall} \leftarrow \text{kNearestNeighbor(finalEntry,LearnedSignal,k)} \) \{kNearestNeighbor returns the \( k \) nearest neighbor, of the finalEntry in the trainingSignal using the Euclidian norm\}
6. \( \text{for all } \vec{x} \in \text{EpsilonBall do} \)
7. \( \quad \text{if } \text{dist}(\vec{x},\text{finalEntry})=0 \text{ then} \)
8. \( \quad \quad \text{print WARNING: The nearest neighbor } \vec{x} \text{ was 0 away. Projection Error has occurred.} \)
9. \( \quad \text{end if} \)
10. \( \text{end for} \)
11. \( \text{Prediction(1) } \leftarrow \text{finalEntry} \)
12. \( \text{while notDone do} \)
13. \( \quad \text{indexToRemove } \leftarrow \emptyset; \)
14. \( \quad \text{for all } j \in \text{epsilonBallIndices do} \)
15. \( \quad \quad \text{if } j < \text{numVectors(}\text{trainingSignal}) \text{ then} \)
16. \( \quad \quad \quad \text{epsilonBallIndices}(i) \leftarrow \text{epsilonBallIndices}(i)+1 \)
17. \( \quad \quad \text{else} \)
18. \( \quad \quad \quad \text{indexToRemove}(\text{end}+1)=j \)
19. \( \quad \text{end if} \)
20. \( \quad \text{REMOVE Elements of EpsilonBall recorded in indexToRemove} \) \{Clean up epsilon ball\}
21. \( \quad j \leftarrow j+1 \) \{Get next prediction by collapsing the epsilonBall to a Point\}
22. \( \quad \text{Prediction}(j) \leftarrow \text{collapseBallToPoint(epsilonBall)}; \)
23. \( \quad \text{numToExtend } \leftarrow \text{numToExtend}-1 \)
24. \( \text{end for} \)
25. \( \text{if numToExtend}\leq0 \text{ or epsilonBall}=\emptyset \text{ then} \)
26. \( \quad \text{notDone } \leftarrow \text{FALSE} \)
27. \( \text{end if} \)
28. \( \text{end while} \)
29. \( \text{return Prediction} \)

**Remark.** While \( k \)-ball SLMA uses more information than SLMA in each prediction, it is not the case that this method is vastly more computationally expensive. In fact the difference is negligible. Observe in both methods I must find the nearest neighbor of \( S_N \). There is a vast amount of literature for implementing nearest neighbor searches in \( \log n \) time. These algorithms can easily be adapted to find the \( k \)-nearest neighbor in \( \log n \) time as well. So the expensive part of each algorithm—the nearest neighbor search—is the same in both case.
2.3 Discussion of Prediction In Projection

Let me conclude the theoretical discussion with a reality check recalling the reason that I wish to do delay coordinate embedding on time series data. My application is ‘on-the-fly’ prediction of computer performance dynamics. As is clear from the discussion of delay coordinate embedding, the choice of embedding parameters and the construction of the delay coordinate map requires a great deal of human intuition and interpretation. Moreover, the verification and corroboration of these parameters involve calculating dynamical invariants such as, Lyapunov exponent, correlation sum, dimension, etc. Algorithms to calculate these invariants require even further human intuition to interpret. So I must consider what the reason behind getting a “true” embedding is, and whether construction of a “true” embedding is even possible.

In theory, I desire a true embedding because then the reconstructed dynamics and the actual dynamics are related by a diffeomorphic conjugacy. That is, the reconstructed space is identical topologically to the original dynamics. According to [18] “If the objective is to use $F(A)$ to predict the future behavior of trajectories, then it is sufficient to have the map $F$ be one-to-one, in which case $n > 2\text{dim}_{\text{cap}}(A)$ is needed. Knowing the current state in $F(A)$ is sufficient to predict the future of the trajectory (at least in the short run).” The core of this thesis is an exploration of whether this is really necessary for prediction of computer performance dynamics. While the restriction that $n > 2\text{dim}_{\text{cap}}(A)$ is a sufficient condition, it may not a necessary condition. I hypothesize that conjugacy is not necessary for “adequate” prediction of computer performance dynamics, and the results in Chapter 3 bear this out.

Aside from it being impractical to verify the delay coordinate map “on-the-fly”, there are several fundamental reasons I do not believe that striving for topological conjugacy is necessary. The most prevalent mathematical reason is that a major assumption of the embedding theorems [18, 20] never holds true in experimentation. That is, every experimental data set has error and real world limitations of a computer do not allow for either infinite length time series or infinite precision arithmetic. Thus, it is impossible to satisfy all the assumptions of the embedology theorems in practice. The point being, even if we were to successfully choose $m > 2d_{\text{cap}}$ we still cannot guarantee that the reconstructed system is diffeomorphic to the original structure because we cannot, in practice, satisfy the aforementioned assumption. As was described in [10]
conjugacy is really a luxury that is unnecessary for my application. I seek to exploit the overall landscape of the dynamical system but the minor details are superfluous. Moreover, the minor details are just as likely to be present due to noise as to actual dynamical features.

In addition to this reasoning, work is currently being completed [11] by Mischaikow et al. that questions the mathematical stringency of embedology. [11] suggests that much of the geometry of an attractor, in particular connectedness, can be reconstructed without choosing $m > 2d_{cap}$ and only assuming that the observation function is continuous, instead of smooth. This is an example of the traction which can be received by relaxing some of the assumptions placed on embedding theory in experimentation.

\footnote{The time series is an infinite, smooth ($C^1$) sampling of the original dynamics which is completely noise free. Moreover, arithmetic used in processing must be of infinite precision.}
Chapter 3

Forecasting of Computer Performance Dynamics

3.1 The Dynamical System: A Computer

As reported in [13], modern computer systems are complex nonlinear dynamical systems. The following discussion of a computer as a dynamical system will closely follow the presentation in [13]. To reframe and study a computer as a dynamical system, I need to understand both the update rule \( \Phi \) and the state space \( X \). A computer system is a complex physical collection of silicon, metal, transistors and circuit boards which are designed to implement deterministic sets of instructions. Abstractly, a computer can be thought of as a system which accepts input, manipulates it according to a predefined set of deterministic rules and produces output. From a dynamical systems perspective, I am tempted to simply interpret the input and output as a “state” in the dynamical system, this is not the case however. The program’s input and output are not directly a “state” of the dynamics the state of the dynamical system is defined by the contents of the addressable memory at a fixed position in time. Thus, the state space is some collection of all possible contents of the addressable memory. (This definition will be made more concrete in the next section.) I could then define the update rule as a map which takes the content of the addressable memory and assigns to it a unique “next” content. This map is dictated by instructions, input, output, physical configuration of the computer and many other factors, some measurable, some not.

3.1.1 The State Space

Let me consider in more detail what the state space of the computer system will look like. For the purpose of this thesis, let me consider a “stored-program computer”, i.e. a standard von Neumann
architecture. In a stored-program computer, the current state, both instructions and data, are stored in some form of addressable memory, generally random-access memory or RAM. With x86 machines for example, each computer state has associated to it a 32-byte address in memory which is a low level organization of all current instruction set variables. These $2^{32}$ different states will correspond to all possible states of the computer system. Thus, as in [13] I treat the addressable memory, a $2^{32}$ long vector of bytes, (4 gigabyte long) $\vec{m}$ as the state space. As stated in [13] this does not however describe the full state space, “As different processors instantiate these 32-byte data in different ways: using different configurations of transistors on the chip, for instance, and different strategies for organization and use, and the processor is only part of the architecture: External memory, video cards and the like also affect performance.” [13] goes on to say that these implementation decisions not only affect the dynamics of the computer system but introduce new, unknown state space variables, these are referred to as implementation variables and denoted by $\vec{u}$. Thus I also define the state space $X$, as follows:

$$X = \{ \vec{x} \mid \vec{x} = [\vec{m}, \vec{u}] \}$$

(3.1)

### 3.1.2 The Map

As suggested by equation 3.1, computer performance dynamics $\vec{F}_{perf}$ is defined by the composition of two independent maps. One map acts, on the addressable memory $\vec{m}$ directly, based on the program instructions; this will be referred to here as program dynamics. The second map describes how the hardware and instruction implementations deterministically affect the next state of the program; this map will be referred to here as implementation dynamics.

To gain a further understanding consider the program dynamics, which are defined by the actual execution of the program, i.e. the change of the program counter and the sequence of steps that the computer follows according to the source code. On the surface it would seem that the update rule is readily available to us, in the form of the source code. The full picture of the program dynamics is slightly more complicated. To run software, computer systems do not use the source code directly. Instead, programs written in a high level language, C for instance, are translated by a compiler into a set of opcodes or machine language instructions which the computer then executes. The translation strategy from high level
language into opcodes is dictated by the instruction set architecture (ISA) for that computer, an abstract specification of how each program instruction is actually implemented on the architecture [13].

A computer program is a set of instructions which affect $\vec{m}_i$ in specific ways. The content of the addressable memory evolves through it’s state space based on a deterministic update rule (the instructions) in discrete time (one instruction per time step)[13]. Borrowing from the notation in [13] I will refer to the update rule for the program dynamics as $\vec{F}_{\text{code}}$, from which I may define the iterated map,

$$\vec{m}_{n+1} = \vec{F}_{\text{code}}(\vec{m}_n)$$

(3.2)

where $\vec{m}_n$ is the state of the ISA variables at time $n$.

This still does not describe the full dynamics, however, as one more level of abstraction is present in the implementation of the source code: the concept of micro-architecture. Micro-architecture is the proprietary set of processor-specific design techniques used to implement hardware that runs the instruction set. The micro-architecture is generally proprietary and specific to the processor. Even if a computer adheres to a particular ISA –such as x86 as is the case with the Intel Core2® Duo, and the Intel Pentium 4®—the way that the opcodes specified by the ISA are actually implemented can vary among processors. These implementation dynamics also plays a role in $F$. Both the Intel Core2® Duo, and the Intel Pentium 4®, for example, comply with the x86 ISA but have radically different dynamics [13].

In addition to micro-architecture, I must also consider physical architecture: placement of transistors, and the temperature of the CPU during runtime, peripheral interaction such as GPU usage and remote memory access, and other effects. These unmeasurable and unknown variables are included in the portion of state space denoted by $\vec{u}$ in equation 3.1. These effects are evident in any computer system and they play key roles in the performance dynamics.

Thus the software being run and the hardware the software is being run on both play a crucial role in the evolution of a computer system. Thus define the iterated map $\vec{F}_{\text{impl}}$ as the hardware specific abstraction which describes the implementation decisions made by the manufacturer at both the physical and micro-architecture levels. I am now able to define the iterated map for computer performance dynamics
as follows:

\[ x_{n+1} = F_{perf}(x_n) = F_{impl} \circ F_{code}(x_n). \]  \hspace{1cm} (3.3)

### 3.1.3 The Specific $F_{code}$ I am Studying

![A visualization of row-major and column-major matrix traversal](image.png)

Figure 3.1: A visualization of row-major and column-major matrix traversal

To explore the dynamical performance of a computer I want to study an $F_{code}$ that is simple enough to understand, but complex enough to exhibit interesting dynamics. I study the dynamics of a segment of code which initializes a 2048 x 2048 matrix in row-major and column-major order. Row-major initialization refers to matrix traversal across each row, instead of down each column, the latter is referred to as column-major, to see what is meant by row-major and column-major traversal see Figure 3.1. The following C code segments perform row-major and column-major matrix initialization.

#### Column-Major Matrix Initialization

```c
for(i=0;i<SIZE;i++)
    for(j=0;j<SIZE;j++)
        data[j][i]=0;
```

#### Row-Major Matrix Initialization

```c
for(i=0;i<SIZE;i++)
    for(j=0;j<SIZE;j++)
        data[i][j]=0;
```

To the untrained eye these two sets of instructions may seem to be identical. They both receive a block of data, namely static int data[SIZE][SIZE], and the result is a block of memory SIZE\*SIZE*sizeof(int)
long with every entry set to 0. However the fact that one is done in row-major order and one is done in column-major order makes the dynamics fundamentally different. Many computer systems attempt to anticipate resources that future instructions will need and have them available in the fastest level of memory, the cache, a process generally referred to as caching or prefetching. As one can see from the way that a matrix is mapped into memory (Figure 3.2), it makes the most sense when traversing a data structure which is stored in row-major order, to do the traversal in row-major order. During execution of code the CPU makes assumptions about program memory access patterns based on the physical arrangement of memory. Since cache access is so much faster than main memory access in anticipation of its usage this chunk of data, as well as the associated instruction, will be prefetched and loaded into the cache. Here I am modifying $\vec{F}_{code}$ intentionally to stimulate the way that $\vec{F}_{impl}$ reacts.
3.2 Methods: Data Collection, Embedding Techniques and Prediction

3.2.1 Time Series Collection: The Observable

While the program dynamics are at least partially dictated by code; the implementation dynamics are for the most part unobservable and unknown. The goal here is to understand the composition of both functions in equation 3.3, and the first step in doing that is to observe $s_n$. As was discussed in Section 3.1.1, one portion of the state space is the addressable memory of the computer. It is tempting then, to simply dump main memory on a regular interval, for instance every 100,000 cycles. This would create a terabyte of data every 40 ms [13]. The process of creating and storing this quantity of data would completely dominate the underlying dynamics. Moreover, this much data would overwhelm even the fastest nonlinear time series analysis methods. Thus, in order to study the dynamics I will make observations of the system and then reconstruct the dynamics using delay coordinate embedding.

As is common practice in computer performance analysis, I utilize the hardware performance monitors (or HPMS) on the processor chip to record specific performance metrics while a program runs. HPMS are dedicated hardware registers that can count events, such as instructions committed per cycle or number of times the L2 cache was missed. A respected and well supported tool for collecting data from the HPMS is the Performance Application Programming Interface (PAPI) [1]. According to the ICL group, the role of PAPI is: “... to provide the tool designer and application engineer with a consistent interface and methodology for use of the performance counter hardware found in most major microprocessors. PAPI enables software engineers to see, in near real time, the relation between software performance and processor events.” [1]

To collect time series traces I utilize profile-me, a script developed by T. Mytkowicz [12] which interfaces with PAPI in order to record values in the HPMS, such as total number of cycles, L2 cache misses or total number of instructions committed. Profile-me does this by setting PAPI flags such as PAPI_TOT_CYC, PAPI_L2_DCM, or PAPI_TOT_INS. After every 100,000 instructions (this interrupt rate is justified below), profile-me puts the program on hold and collects the data that PAPI is recording. These observations are then recorded in an events file and the program continues execution. This process is continued until the program terminates.
To ensure a smooth observation function, I must pay special attention to three key issues. The first being that I should not interfere with the dynamics by observing too aggressively, but I must measure frequently enough to paint a true picture of the underlying dynamics. Much attention was put into this in the development of [13]; after much analysis, it was determined that interrupting the program every 100,000 instructions was the most effective. By interrupting the program for observation more frequently, the measurement infrastructure began to play a role in the dynamics, less frequently seemed to ill sample the system resulting in error [13]. For this reason, I choose to interrupt the software for observation every 100,000 instructions.

The next concern to address is that multiple programs are often running simultaneously on a computer. If the assumption, about the dynamics of the computer, are correct, that is:

\[ x_{n+1} = F_{perf}(x_n) = F_{impl} \circ F_{code}(x_n) \]  

Then each program that is running has a specific \( F_{impl} \circ F_{code}(x_n) \) and hence the time series may sample multiple dynamical systems either in an inter-leaved or a (worse) composed fashion. Thus to it is critical that I understand and effectively isolate the processes running. If multiple processes are sampled by the time series, delay coordinate embedding no longer makes sense. To obtain a comparatively untainted trace of the performance dynamics, I compile and execute the code using only local disks and running as few other processes as possible—that is, running Linux INIT Level 1 to reduce the number of background process. I further make certain that no other user-defined programs are running during experimentation. This guarantees that, to the best of my ability, \( x_{n+1} = F_{perf}(x_n) = F_{impl} \circ F_{code}(x_n) \) is the only dynamical system being observed.

Finally, it is important to remember the measurement facility is actually a part of the physical system. That is, the observation utility is a program running on the computer and thus may be a part of the dynamical system. The current view in the systems community is that, there is a lack of PAPI effects on the program being observed; for frequent observations, however, drastic bifurcations were observed in the underlying dynamics [13]. It was experimentally determined in [13] that by choosing an interrupt rate of 100,000 instructions PAPI had little affect on the dynamics being observed. E. Bradley et al. collected
performance traces at various interrupt rates and analyzed the frequency spectra of this data. Similarity in spectra between interrupting at 50,000 and 100,000 instructions suggested that PAPI had minimal effect on the observed dynamics if interrupting was done every 100,000 instructions.

3.2.2 Performance Metrics

My study focuses on four performance traces obtained by monitoring two performance metrics (L2 cache misses and instructions per cycle[IPC]) for two different programs (row and column-major matrix initialization). In section 3.2.1 I described how I monitor specific metrics; in this section I provide a brief description of each metric and why it is of interest from computational and forecasting perspectives.

3.2.2.1 Cache Misses

The L2 cache, is the second level of cache in the on-board memory hierarchy in a CPU. The cache is a small and extremely fast memory that stores copies of data that the program uses the most frequently or that the CPU anticipates the program will need in the near future. The performance of a program is directly bound by it’s effectiveness in utilizing the cache—in particular, its ability to properly “prefetch” the data it will require. When data is needed by a processor, it first checks the cache. If the data is available it is loaded and used; this is referred to as a cache hit. Otherwise, the data must be fetched from main memory, which is referred to as a cache miss.

Cache misses are interesting performance metrics because they allow for observation of the memory usage of a particular application. One possible advantage to predicting cache misses is the ability to preemptively prefetch data needed by a micro-kernel.

3.2.2.2 Instructions Per Computational Cycle

Instructions per computational cycle (IPC) is a measure of processor performance: how efficiently it can execute the instructions of a particular program in a single clock cycle. Great fluctuation in IPC can occur during something as simple as an array initialization. This is due to the fact that many aspects of the computer hardware and the software—computer memory hierarchy, I/O of software, cache misses and
bus contention—affect IPC. This coupling makes this particular metric an ideal observable of the dynamics of the computer. It is also interesting from a computational perspective because it illustrates how busy a computer stays each cycle. The ability to forecast this signal has direct applications in load balancing and also energy efficiency as CPUs can be dynamically turned on and off as they are needed.

### 3.2.3 The Time Series in the Time Domain

Figures 3.3 and 3.4 are views of the time series we aim to predict each plotted in the time domain. Figure 3.3 shows the full time series generated while running the code presented in Appendix A.2. This code alternates between initializing a matrix in row-major and column-major order, a fixed number of times. Switching between these two initialization paradigms is the reason behind the large jumps in the time series. Visually there is a clear delineation between row-major and column-major dynamics, column-major for example, has high cache miss and low IPC while row-major has the exact opposite behavior. Closer views of the separated signals can be seen in Figure 3.4.

**Figure 3.3: Full time series in the time domain**
Figure 3.4: Detailed view of column-major cache-miss rate and row-major IPC
3.2.4 Delay Coordinate Embedding

3.2.4.1 Estimating Embedding Parameters

First step is to numerically estimate embedding parameter values for each time series, as discussed in Section 2.1.2. Figure 3.5(a) shows the mutual information of each of the four time series. From this, following [3], I choose the time delay $\tau = 100,000$ instructions. Notice that some of these signal’s mutual information reach their first minimum at 200,000 instructions but [3] is a rough heuristic, so either 100,000 or 200,000 are acceptable choices.

By examining 3.5(b) I estimate $m$ by using the standard method presented in Section 2.1.4. That is, following [7] I choose $m$ to be the dimension for which the fraction of false nearest neighbors drops below a 10% threshold. Based on this heuristic I conclude that $10 \leq m \leq 25$. This range of $m$ is then narrowed down by analyzing dynamical invariants, as described in Section 2.1.4. This analysis implies that $m$ should be 12 for each of the signals [13]. Notice this choice of $m$ corresponds to no more than a 20% fraction of false nearest neighbors for any of the signals.

![Figure 3.5: Mutual and false nearest plot for the four performance traces](image-url)

(a) Mutual information for each of the four time series  
(b) Fraction of false nearest neighbors for each of the four time series
3.3 Prediction of Computer Performance Dynamics

Recall that each of the following methods truncates the final 10% of the time series for comparison and that data is not used in the “training-signal”, this data is known as the comparison signal. The training-signal is the portion of the time series we use to learn the signal’s behavior. In our case, the training-signal is the first 90% of the original time series.

3.3.1 Forecasting Using Estimated Embedding Parameters

For each of the following prediction diagrams I will use the legend in Figure 3.6. The prediction results in this thesis are presented in the time domain, and in each figure I plot the comparison signal as well as the predicted signal. Error bars are provided to aid in visually comparing points which correspond to each other in the time domain. Notationally, the embedding which results from the estimated embedding parameters will be referred to as the full embedding to distinguish it from the projected embeddings which will be discussed later.

![Legend for figures that present forecasting results](image)

Figure 3.6: Legend for figures that present forecasting results

3.3.1.1 Prediction Results for Column-Major Dynamics with Full Embedding

To begin this discussion I will consider the column-major dynamics. As this implementation of matrix initialization is counterintuitive to system design, I conjecture that the column-major time series will exhibit more complicated dynamics. Recall that the column-major time series is designed to exhibit large cache misses and thus ill computer performance. As this code is designed to stimulate interesting cache miss behavior, I think this signal should prove challenging to predict.

As can be seen in figures 3.7 and 3.8, the dynamics of column-major matrix initialization were predicted quite accurately. While I anticipated this to be challenging to predict, as it goes against computer design,
my methods were able to predict them. As shown in Table 3.1, by comparing RMSPE I can conclude that the $k$-ball adaptation to SLMA was superior in predicting both IPC and cache misses. Recall though, that RMSPE of two algorithms can only be compared when applied to the same signal. That is, even though $8.1577e - 04 < < 15.2381$ this does not imply that the prediction of IPC was more successful than the prediction of cache misses, as the scale of these signals are different.
Figure 3.7: Prediction of column-major cache-miss rate: RMSPE for (a) is 15.2381 and (b) is 11.5660
Figure 3.8: Prediction of column-major IPC: RMSPE for (a) is 8.1577e-04 and (b) is 5.0020e-04
3.3.1.2 Prediction Results for Row-Major Dynamics with Full Embedding

Viewed from a computer performance standpoint, row-major dynamics should not be interesting as the computer memory layout, caching, and many other factors would say that these dynamics should be efficient and thus perhaps simple. As such, I do not expect that the row-major dynamics will be difficult to predict.

Table 3.1 shows that, once again, the $k$-ball algorithm predicted both signals far more accurately; in particular, the cache miss dynamics prediction was improved by orders of magnitude over the SLMA algorithm. This is also apparent from the images of the two signals. The SLMA cache prediction is so inaccurate that I can hardly distinguish any of the points due to the error bars which cloud the picture. However, the $k$-ball SLMA algorithm greatly reduces the error over SLMA and with the reduction of error bars present the signals become far more visible.

These results are somewhat bothersome. I conjectured that the row-major time series would be a far more regular signal than either of the column-major signals. My conjecture was based on the fact that the computer design and memory layout lends itself particularly well to row-major traversal of an array; I thought that these design principles would correlate to simpler dynamics but this did not turn out to be the case with row-major dynamics.

I hypothesize that the reason I see such poor prediction results with both row-major signals is that IPC is far higher with row-major over column-major. I believe the heightened IPC causes a *quickness* in the dynamics which the current measurement infrastructure has difficulty observing accurately. Column-major is largely memory bound and must constantly fetch data from main memory. Each cache miss results in an interaction with main memory, causing a bottleneck in the program. I hypothesize that this bottlenecking slows down the dynamics enough that I obtain a good picture of them by interrupting every 100,000 instructions. This suggests to me that to reconstruct the row-major signal and obtain more-adequate prediction results, it is necessary to interrupt more frequently. Unfortunately due to the nature of the custom measurement infrastructure I cannot observe the dynamics any more frequently without affecting the dynamics [12].
Table 3.1: Collective RMSPE for all Predictions with Full Embeddings

<table>
<thead>
<tr>
<th></th>
<th>SLMA RMSPE</th>
<th>$k$-Ball SLMA RMSPE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cache-Miss Rate</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Column-Major</td>
<td>15.2381</td>
<td>11.5660</td>
</tr>
<tr>
<td>Row-Major</td>
<td>206.7005</td>
<td>59.8668</td>
</tr>
<tr>
<td><strong>IPC</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Column-Major</td>
<td>8.1577e-04</td>
<td>5.0020e-04</td>
</tr>
<tr>
<td>Row-Major</td>
<td>0.0327</td>
<td>0.0208</td>
</tr>
</tbody>
</table>

Figure 3.9: Prediction of row-major cache-miss rate: RMSPE for (a) is 206.7005 and (b) is 59.8668
Figure 3.10: Prediction of row-major IPC: RMSPE for (a) is 0.0327 and (b) is 0.0208
3.3.2 Forecasting With Two-Dimensional Projections

Due to the issues with measuring the row-major dynamics, I will only study two-dimensional projections in the context of column-major dynamics. Some notation is necessary for the following section. In all of the projection results, I begin by projecting the full state space down onto two dimensions. There are \( \binom{n}{2} \) possibilities here.

Let \( i, j \in \{1, \ldots, m\} \), then define the 2D projection map of a vector \( \vec{X} \) with \( m \) components in the obvious way, that is: \( P_{(i,j)}(\vec{X}) = [x_i \ x_j]^T \). I choose the “standard” projection as \( P_{(1,2)} \), as this would be equivalent to “arbitrarily” using delay-coordinate embedding with \( m = 2 \), this makes the most sense for on-the-fly embedding. An important question to consider is the following: Does it matter what projection I choose? That is, do I obtain equivalent results, from a forecasting perspective, if I choose any \( P_{(i,j)} \)? I established that this is not the case. In fact, almost every unique combination of \( i, j \) result in a slightly different prediction.

One could also choose the “best” projection by testing all possible \( P_{(i,j)} \) and then choosing the one that produces the smallest RMSPE. I take that approach here, defining the best projection as the one for which the resulting forecast (using SLMA) has the smallest RMSPE. Notice that a different projection may result in a lower RMSPE for \( k \)-ball SLMA; I have chosen to use the simpler algorithm as the baseline. Figures 3.11 - 3.14 show the associated prediction results.
Figure 3.11: Standard projection of the column-major cache-miss rate: RMSPE for (a) is 521.4182 and (b) is 300.7126
Figure 3.12: Best Projection of the column-major cache-miss rate: RMSPE for (a) is 24.3713 and (b) is 16.0508
Figure 3.13: Standard Projection of the column-major IPC: RMSPE for (a) is 8.1577e-04 and (b) is 5.5163e-04
Figure 3.14: Best Projection of the column-major IPC: RMSPE for (a) is $6.9028e-04$ and (b) is $5.2564e-04$
3.3.2.1 Discussion of Column-Major Projection Results

Table 3.2: Collective RMSPE for all Column-Major Prediction Methods

<table>
<thead>
<tr>
<th>Cache-Miss Rate</th>
<th>SLMA RMSPE</th>
<th>k-Ball SLMA RMSPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full Embedding</td>
<td>15.2381</td>
<td>11.5660</td>
</tr>
<tr>
<td>Standard Projection</td>
<td>521.4182</td>
<td>300.7126</td>
</tr>
<tr>
<td>Best Projection</td>
<td>24.3713</td>
<td>16.0508</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IPC</th>
<th>SLMA RMSPE</th>
<th>k-Ball SLMA RMSPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full Embedding</td>
<td>8.1577e-04</td>
<td>5.0020e-04</td>
</tr>
<tr>
<td>Standard Projection</td>
<td>8.1577e-04</td>
<td>5.5163e-04</td>
</tr>
<tr>
<td>Best Projection</td>
<td>6.9028e-04</td>
<td>5.2564e-04</td>
</tr>
</tbody>
</table>

These projection results, which are summarized in Table 3.2, suggest that it is viable to use low dimensional projections to forecast computer performance dynamics. From the cache dynamics I can conclude that doing full embedding is the best choice since it produces the smallest RMSPE for both SLMA and k-SLMA. However, the “best projection” generally rivals the full embedding results very convincingly. Even more convincingly, consider RMSPE for IPC, the “best” projection actually produced a lower RMSPE than with the full embedding. This means that the projected dynamics produced more accurate results than the full embedding! When using the k-ball SLMA the full embedding is superior to the projected dynamics but by a small margin. The most probable conclusion, is that column-major IPC is a noisy signal. This becomes clear if one considers the construction of the delay coordinate vector. Assume one point is noisy in the time series, then with a twelve dimensional embedding this data point is touched 12 times. If I instead consider a two dimensional projection, this noise point is only touched twice. Thus, this two dimensional projection provides a more accurate prediction because it is less affected by the noise in the data.

According to the literature [18, 20], it could very well be the case that the two-dimensional projected dynamics used in my forecasting method are not topologically conjugate to the original dynamics; even topologically semi-conjugacy is questionable. Regardless, my results show I can quite successfully predict the future behavior of memory and processor dynamics of an Intel Core2® Duo through arbitrary two dimensional predictions, at least for this simple micro-kernel. Even the standard projection, which appears far worse than the other prediction results I presented, gives a glimpse of future dynamics and might still be
helpful in the load balancing on the processor. Important to note that the prediction of computer performance need not be perfect. Even if I can provide a glimpse of the future processor or memory load, this may be extremely helpful. The mathematical requirements (conjugacy) may well be excessively stringent for these purposes. These results strongly suggest that projections, which ignore topologically conjugacy, are viable for the prediction of computer performance dynamics.

**Remark.** The “best projection” results in Table 3.2 illustrate that instead of simply using a naïve projection it is advantageous to determine the best projection. As can be seen in Table 3.2 this type of preprocessing can produce results which rival the full embedding results. Furthermore, while the best projection results are similar to the full embedding results, in comparison, this preprocessing is far less time consuming than analyzing dynamical invariants false_nearest, and mutual, which are all computationally expensive and processing is human intensive. Moreover, this analysis must be done post facto, which is not an option in forecasting schema.

### 3.4 Future Work

Several interesting problems have arisen during the development of this thesis. Each of these problems focus around improving the results presented here by making the algorithms not only more robust but also more versatile. In addition to this, particular micro-kernels are being examined as candidates for performance forecasting, that are also useful to the systems community.

The corrective aspect of the improved $k$-ball SLMA algorithm is concerned with false crossings caused by low dimensional projections of the time series. This algorithm aims to correct $k$-Ball SLMA by exploiting the fact that the underlying dynamical system is continuous in space. That is, fix $\epsilon > 0$ and define $f$ to be the map or flow defining the underlying dynamics. By the spacial continuity of $f$, there exists a $\delta$, for this $\epsilon$, such that every ball of radius $\delta$ ($B_\delta$) has a forward image whose radius is less than $\epsilon$, that is radius($f(B_\delta)$) < $\epsilon$. This fact is precisely the property of $f$ I aim to exploit to correct the $k$-ball SLMA algorithm.

Consider a low dimensional projection of a dynamical system which contains false crossings. If I then construct $B_\delta$ around a point and map $B_\delta$ forward, the forward image may not have radius less than $\epsilon$. For
example, consider a false crossing going in two directions: one element may get mapped one direction and one gets mapped the opposite. However, this contradicts the assumption that the unknown dynamical system is continuous in space. Thus I can simply throw away points that are outside of the $\epsilon$-ball. The challenge here is knowing which points should be a part of the $\epsilon$-ball and which points had pre-images that were false nearest neighbors of the point I am predicting.

With this in mind, I propose the following improvement upon algorithms described in this thesis. The first step is similar to $k$-Ball SLMA, I construct $B_k(\mathbf{S}_N)$ and map this ball forward call the mapped ball $F(B_k)$. ($F$ denotes the delay coordinate map, not the dynamical system map.) Now I check to make sure that each point in the image ball is within some tolerance $\epsilon$ of each other. In the spirit of the $k$-ball implementation, rather than set a fixed $\epsilon$, I define $\hat{k}$ as the ceiling of 10% of the cardinality of $F(B_k)$, e.g. if 99 points are in $F(B_k)$ then $\hat{k} = 10$. My algorithm then takes each point in $F(B_k)$ and confirms that at least 30% of its $\hat{k}$ nearest neighbors intersect $F(B_k)$. This assumes that every element in the image of the $k$-ball have nearest neighbors that are in the image of the $k$-ball, ensuring the spacial continuity of $f$ is preserved. If a point does not have enough nearest neighbors in the image, my algorithm removes it from $F(B_k)$. After all points in $F(B_k)$ have gone through this analysis, average the remaining points to synthesize the “next point” in the time series, using equation 2.2.2, then repeat this until the specified prediction horizon is reached. This algorithm will mitigate the influence of false crossings in the low dimensional projections. While in theory it should be superior to $k$-Ball SLMA in that it checks for false crossings and adjusts based on this, it is also far more expensive. Currently I am working on balancing this tradeoff. The preliminary code for this project can be found in Appendix A.1.3.

The next step in evaluating these prediction methods is to apply them to a “real world” computer program. The examples in this thesis, while potentially chaotic, were very controlled and the problem of initializing a matrix is not very interesting and useful. Successful prediction of a real program’s performance would be far more interesting.

For this to be a useful tool it will also be necessary to develop methods that can sense the presence of multiple attractors, adaptively re-learn the signal to take into account the interchange between dynamical attractors, and handle noise. These are realities for scientists in any field. The standard approach to noise
reduction and signal separation is to use the Fourier transform to identify predominant frequencies in the signal and then separate components and/or remove extraneous frequencies. When the signal is nonlinear, and in particular chaotic, a continuum of frequencies are present and Fourier based techniques tend to destroy information carrying content [21].

A number of methods have been proposed for removing noise from chaotic data. One class of techniques uses the fact that trajectories of continuous dynamical systems are confined to smooth manifolds in state space. This idea is exploited in [2] by approximating the local geometry of the manifold and in [16] by exploiting the resulting perfectness of the attractor. These techniques specifically utilize properties of a continuous time dynamical system and cannot be applied when the dynamics are discrete. The method currently being developed by Z. Alexander, E. Bradley, J. Meiss and myself extends the latter idea to the case of discrete nonlinear maps.
Chapter 4

Conclusions

Using a deterministic dynamical systems approach to computer performance forecasting simply does not appear anywhere in the literature. In fact, the success of this type of forecasting is in stark contrast to current computer design principles, which assert that today’s computer systems are a highly complex coupling of several different unobservable stochastic systems. The success of forecasting results based in the theory and methods of deterministic dynamics provides further evidence to support [13], in particular, that computer systems are deterministic nonlinear dynamical systems. The use of the robust language of dynamical systems is a potential asset to the computer systems community, as it provides a new and unique metric for analyzing and comparing computer performance. The results presented here suggest potential methods for improving computer design at a systems level as well as evidence to support the use of semi-conjugacies in forecasting schemes.
Bibliography


Appendix A

Thesis Source Code

A.1 Matlab Code

A.1.1 SLMA Code

**SLMA Driver**

```matlab
function Prediction = Predictor(DCEofTimeSeries)
%This produces a prediction of the final 10% of DCEofTimeSeries using SLMA.
%Import the requested portion of the time series data.
A = DCEofTimeSeries;
%Truncate last 10% of time series.
N = size(A,1);
numToTruncate = N*.1;
numToExtend = numToTruncate;
%Determine m in the embedding and create a matrix to begin prediction.
Prediction = zeros(numToExtend,size(A,2));
A(size(A,1) - numToTruncate:size(A,1),:) = [];
%Get Last Element.
N = size(A,1);
xCurrent = A(N,:);
%Attempt to rebuild this time series.
    Prediction(1,:) = xCurrent;
    distToEnd = 0;
    while(distToEnd == 0)
        [neighborRow neighDist] = nearestNeighbor(A,xCurrent,numToExtend);
        if (neighDist == 0)
            disp('Warning: Projected Distance was 0. False crossing has occurred');
        end
        max = size(A,1);
        distToEnd = max - neighborRow;
    end
    Prediction(i1,:) = A(neighborRow+i1,:);
end
plotError(DCEofTimeSeries,Prediction)
end
```

**SLMA Supporting Programs**

```matlab
function [neighborTime neighborDistance] = nearestNeighbor(NeighborMatrix,xToNeighbor,theiler)
```
% Remove the final 10% of the known signal this ensures that the neighbor
% I determine has enough forward images to make the full prediction
% This also means that I will only use 80% of the original signal to
% learn the signal which is a downfall of this compared with $k$-ball
% SLMA.
NeighborMatrix(size(NeighborMatrix,1) - theiler:size(NeighborMatrix,1,:),:) = [];
% Dynamically determine the size of the delay vectors and the number of
% vectors I have.
m = size(NeighborMatrix,1);
n = size(NeighborMatrix,2);
% Construct a "matrix" which has xToNeighbor as every row.
ToNeighbor = ones(m,n);
for i = 1:m
    ToNeighbor(i,:) = xToNeighbor;
end
% ToNormalize is a vector of every matrix in NeighborMatrix-xToNeighbor,
% this will allow me to directly apply the euclidean normal to a signal set
% of vectors.
ToNormalize = NeighborMatrix - ToNeighbor;
MatrixNorm = zeros(m,1);
for i = 1:m
    % Record the Euclidian norm of each element in the matrix.
    MatrixNorm(i,1) = norm(ToNormalize(i,:)',2);
end
% Choose the neighbor which has the smallest Euclidian distance with our
% xToNeighbor (the final point in the time series. Return both its
% distance and its placement in the set of delay vectors.
[neighborDistance neighborTime] = min(MatrixNorm);

A.1.2 $k$-ball SLMA code

function Prediction = Predictor(DCEofTimeSeries,numNeighbors)
% This function predicts the last 10% of DCEofTimeSeries using $k$-ball SLMA
% using $k$ = numNeighbors.
% Import the requested portion of the time series data.
A = DCEofTimeSeries;
% Truncate last 10% of time series.
N = size(A,1);
numToTruncate = N*.1;
numToExtend = numToTruncate;
% Determine $m$ in the embedding and create a matrix to begin prediction
Prediction = zeros(numToExtend,size(A,2));
A(size(A,1) - numToTruncate:size(A,1,:),:) = [];
% Get Last Element
N = size(A,1);
xCurrent = A(N,:);
% Attempt to rebuild this time series.
Prediction(1,:) = xCurrent;
    max = ones(numNeighbors,1);
    max = max.*size(A,1);
% Build an epsilon ball around the final image point of the time series.
[EpsilonBallIndexes BallRadi] = ConstructEpsilonBall(A,xCurrent,numNeighbors);
EpsilonBallIndexes = EpsilonBallIndexes';

j = 1;

% This is the local distance to the end of the learning signal.
LocaldistToEnd = max - EpsilonBallIndexes;
notdone = 0;
while (notdone == 0)
    indexToRemove = [];
    for i = 1:size(EpsilonBallIndexes,1)
        if (LocaldistToEnd(i,1) >= 1)
            % Move that element in the ball forward, and update EpsilonBall
            EpsilonBallIndexes(i,1) = EpsilonBallIndexes(i,1) + 1;
        else
            % Record elements of epsilonBall which did not have a forward
            % image so that they are deleted from the ball before the
            % prediction is made.
            if isempty(indexToRemove)
                indexToRemove(1,1) = i;
            else
                indexToRemove(end + 1) = i;
            end
        end
    end

    % Clean up Epsilon Ball and supporting data.
    max(indexToRemove) = [];
    LocaldistToEnd(indexToRemove) = [];
    EpsilonBallIndexes(indexToRemove) = [];
    LocaldistToEnd = max - EpsilonBallIndexes;
    j = j + 1;

% Get next prediction by collapsing the Ball To a Point.
Prediction(j,:) = collapseBallToPoint(EpsilonBallIndexes, A);
numToExtend = numToExtend - 1;
% Check that I have not reached the prediction horizon and that the
% epsilon ball is not empty.
if (numToExtend <= 0 || (size(EpsilonBallIndexes,1)==0))
    notdone = 1;
end
end
% Plot the result.
plotError(DCEofTimeSeries, Prediction)
end

SLMA k-ball Supporting Programs

function [neighborIds neighborDistances] = ConstructEpsilonBall(dataMatrix, queryMatrix, k)
[neighborIds neighborDistances] = kNearestNeighbors(dataMatrix, queryMatrix, k);
end

function [newPoint] = collapseBallToPoint(ballIndexes, TimeSeries)
% This crushes a $k$-ball to a single point in space. The first entry of
% this new point is the prediction for the time series.
dataBall = zeros(size(ballIndexes,1), size(TimeSeries,2));
% Construct the dataBall from the index ball
for indj = 1:size(ballIndexes,1)
    % reduce the $k$-ball to a point.
dataBall(indj,:) = TimeSeries(ballIndexes(indj,1,:));
end
newPoint = mean(dataBall);
end

Remark. The following function kNearestNeighbors is an unlicensed source file which can be downloaded for free at http://www.mathworks.com/matlabcentral/fileexchange/15562-k-nearest-neighbors it is merely included here for completeness but was not developed by me.

function [neighborIds neighborDistances] = kNearestNeighbors(dataMatrix, queryMatrix, k)
neighborIds = zeros(size(queryMatrix,1),k);
neighborDistances = neighborIds;

numDataVectors = size(dataMatrix,1);
umQueryVectors = size(queryMatrix,1);
for i=1:numQueryVectors,
dist = sum((repmat(queryMatrix(i,:),numDataVectors,1)-dataMatrix).^2,2);
[sortval sortpos] = sort(dist,'ascend');
neighborIds(i,:) = sortpos(1:k);
neighborDistances(i,:) = sqrt(sortval(1:k));
end

A.1.3 Preliminary Corrective $k$-ball SLMA Source Code

Corrective SLMA $k$-ball Driver

function Prediction = Predictor(DCEofTimeSeries,numNeighbors)
%This is the preliminary code to adaptively correct the $k$-ball SLMA algorithm.

%Import the requested portion of the time series data.
A = DCEofTimeSeries;
%Truncate last 10% of time series.
N = size(A,1);
numToTruncate = N*.1;
umToExtend = numToTruncate;
%Determine m in the embedding and create a matrix to begin prediction
Prediction = zeros(numToExtend,size(A,2));
A(size(A,1) - numToTruncate:size(A,1),:) = [];

%Get Last Element
N = size(A,1);
xCurrent = A(N,:);

%Attempt to rebuild this time series.
Prediction(1,:) = xCurrent;
max = ones(numNeighbors,1);
max = max.*size(A,1);

[EpsilonBallIndexes BallRadi] = ConstructEpsilonBall(A,xCurrent,numNeighbors);
%This could be helpful in removing the vectors zero away
%myVector(myVector == 0) = [];

EpsilonBallIndexes = EpsilonBallIndexes';
%if (neighDist == 0)
%disp('Projected Distance was 0.');

j =1;

%This is the distance to the end of the learning signal.
LocaldistToEnd = max - EpsilonBallIndexes;
PRediction(j,:) = collapseBallToPoint(EpsilonBallIndexes,A)
notdone =0;

while(notdone ==0)
    indexToRemove = [];
    for i=1:size(EpsilonBallIndexes,1)
        if(LocaldistToEnd(i,1)>=1)
            %Move that element in the ball forward, and update EpsilonBall
            EpsilonBallIndexes(i,1) = EpsilonBallIndexes(i,1)+1;
        else
            %Record elements of epsilonBall which did not have a forward
            %Image so that they are deleted from the ball before the
            %prediction is made
            if isempty(indexToRemove)
                indexToRemove(1,1) =i;
            else
                indexToRemove(end+1)=i;
            end
        end
    end
    %Clean up Epsilon Ball
    OutlierIndexes = DetectOutliers( EpsilonBallIndexes,A ,l0);
    indexToRemove = [indexToRemove OutlierIndexes];
    max(indexToRemove) = [];
    LocaldistToEnd(indexToRemove) = [];
    EpsilonBallIndexes(indexToRemove) = [];
    LocaldistToEnd = max - EpsilonBallIndexes;

    ballSize = size(EpsilonBallIndexes,1)
    j = j+1;
    %Get next prediction by collapsing the Ball To a Point
    Prediction(j,:) = collapseBallToPoint(EpsilonBallIndexes,A);
    numToExtend = numToExtend-1
    if(numToExtend<=0 || (size(EpsilonBallIndexes,1)==0))
        notdone =1;
    end
end

plotError(DCEofTimeSeries,Prediction)
end

Corrective SLMA \( k \)-ball supporting programs

function [ OutlierIndexs] = DetectOutliers( EpsilonBallIndexes,TimeSeries,littleBallNum )
%Detects elements that have left the big epsilon ball and records the
%elements have ejected themselves out of the ball.
littleBallNum = ceil(size(EpsilonBallIndexes,1)*.1);
littleBallNum = ceil(littleBallNum)
tol = ceil(littleBallNum*.3)
dataBall = zeros(size(EpsilonBallIndexes,1),size(TimeSeries,2));
%Construct the dataBall
for indj = 1:size(EpsilonBallIndexes,1)
    if EpsilonBallIndexes(indj,1) > size(TimeSeries,1)
        disp('something is broken')
    end
    dataBall(indj,:) = TimeSeries(EpsilonBallIndexes(indj,1),:);
end
OutlierIndices = [];
NumOutliers = 0;
for indj = 1:size(EpsilonBallIndexes,1)
    % Get the elements (littleBallNum) nearest neighbors
    [neighborIds neighborDistances] = kNearestNeighbors(TimeSeries, dataBall(indj,:), littleBallNum);
    neighborShare = [];
    neighborShare = intersect(neighborIds, EpsilonBallIndexes);
    if size(neighborShare,2) < tol
        disp('outlier detected')
        OutlierIndices(end+1) = indj;
        NumOutliers = NumOutliers + 1;
    end
end

A.1.4 Matlab Supporting Programs for Both SLMA and k-ball SLMA

function plotError(A,Prediction)
    scrsz = get(0,'ScreenSize');
    % Position and size screen for consistency.
    figure('Position', [1 scrsz(4)/2 scrsz(3)*2/3 scrsz(4)/2])
    hold on
    N = size(A,1);
    numToTruncate = N*.1;
    startIndex = size(A,1) - numToTruncate;
    CorrectSignal = A(startIndex-1:size(A,1),:);
    x = linspace(1,1000,1000);
    % Calculate the RMSPE for the prediction and output it.
    RMSPE = MeanSquaredError(Prediction,CorrectSignal)
    % Plot the first 1000 points of the prediction and correct signal
    % for visual comparison.
    plot(x,CorrectSignal(1:1000,1),'bx');
    plot(x,Prediction(1:1000,1),'ro');
    % Plot Error bars, connecting each point in the prediction to the correct
    % signal.
    for i = 1:1000
        X = [i i];
        Y = [CorrectSignal(i) Prediction(i)];
        plot(X,Y,'k')
    end
    ylabel('Cache Misses')
    xlabel('time (instructions x 100,000)')
function [ RMSPE ] = MeanSquaredError( Prediction,CorrectSignal )
%Calculate and return the RMSPE in the prediction.
u = Prediction(:,1)-CorrectSignal(1:size(Prediction,1),1);
RMSPE = sqrt(sum(u.*conj(u))/size(u,1));
end

function crossSections(A12)
for ii = 1:12
    for kk = 1:12
        subplot(1,12,kk)
        A = [A12(:,kk) A12(:,ii)];
        Prediction = Predictor(A);
        kk
        ii
        plotError(A,Prediction);
    end
end
end

function signalPredictionMovie(Prediction,A)
%Creates a movie of the prediction of the time series.
figure
hold on
N = size(A,1);
umToTruncate = N*.1;
startIndex = size(A,1) - numToTruncate-1;
CorrectSignal = A(startIndex:size(A,1),:);
x = linspace(1,100,100);
for k = 1:899
    subplot(2,1,1);
    hold on
    axis([1 100 .86 .91])
    plot(x(1:50),CorrectSignal(k:k+49),'kx','MarkerSize',10);
    plot(x(1:75),Prediction(k:k+74,1),'bo');
    for i = 0:50
        X = [i+1 i+1];
        Y = [CorrectSignal(k+i) Prediction(k+i)];
        plot(X,Y,'r')
    end
    hold off
    subplot(2,1,2);
    hold on
    axis([1 100 .86 .91])
    plot(x(1:50),CorrectSignal(k:k+49),'-ko','LineWidth',2);
    plot(x(1:75),Prediction(k:k+74,1),b');
    myMovie(k) = getframe;
cif
    hold off
end
end
A.1.5 Data Processing Code

function DCEtser = delayCordEmbedTS(filename, m, tau)
%Construct delay vectors from the time series filename using m and tau, this
%code was developed purely for understanding of the process. The vectors I used
%was generated with TISEAN delay.
tser = load(filename);
reconState = zeros(size(tser,1),m);
time = tser(:,2);
i = 1;
step = time(2)-time(1);
TG = uint32(tau/step)
comblen = TG*(m-1)
maxR = 1+comblen;
while maxR <= size(tser,1);
    for j = 1:m
        reconState(i,j) = tser(i+(j-1)*TG,1);
    end
    i = i+1;
    maxR = 1+comblen;
end
reconState = reconState(1:i-1,:);
DCEtser = reconState;

A.2 C code

//The code to generate 100k.dat
#define SIZE (2048)
int main () {
    int i, j, n;
    static int data[SIZE][SIZE];
    for(n = 0; n < 20; n++) {
        for(i = 0; i < SIZE; i++)
            for(j = 0; j < SIZE; j++)
                data[i][j] = 0;
    }
    return (0);
}

A.3 gnuplot & TISEAN scripts

A.3.1 Mutual Script

set term postscript enhanced color
set output "TimeSeriesMutual.eps"
set xlabel "{/Symbol t} (x 100,000 instructions)"
set ylabel "I (bits)"
set xtics (0,1,2,3,4,5,10,15,20)
plot '<mutual -c 3 row3.dat' w lp title 'Row-major cache misses', '<mutual row3IPC.dat'
w lp title 'Row-major IPC','<mutual -c 3 col3.dat' w lp title 'Column-major cache misses',
A.3.2 False-Nearest Script

set term postscript enhanced color
set output "TimeSeriesFalseNearest.eps"
set xlabel "embedding dimension m"
set ylabel "fraction of false nearest neighbors"
set xtics (1,5,6,7,8,9,10,11,12,13,14,15,.16.17,18,19,20)
plot '<false_nearest row3IPC.dat -M1,25' with linespoints title 'Row-major IPC',
    '<false_nearest -c 3 row3.dat -M1,25' with linespoints title 'Row-major cache misses',
    '<false_nearest col3IPC.dat -M1,25' with linespoints title 'Column-major IPC',
    '<false_nearest -c 3 col3.dat -M1,25' with linespoints title 'Column-major cache misses'