Learning Reduced-Dimension Models of Human Actions

Christopher Lee

CMU-RI-TR-00-17

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Robotics

The Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Yangsheng Xu, Chair
Pradeep Khosla
Karun Shimoga
Rajeev Sharma

May 1, 2000

Abstract

A great deal of current robotics research studies the modeling of human reaction skills: learning control mappings to represent a person's task-performance strategies. The important related problem of modeling human *action* skills has received less attention. Action learning is the characterization of the state space or action space explored during typical human performances of a given task. Action models learned from human performances typically represent some form of prototypical performance, and also characterize how the human's performances vary stochastically or due to external influences. They are used for gesture recognition, for realistic computer animations of human motion, for study of an expert performer's motion (e.g., Tiger Woods' golf swing), for generating feed-forward or reference robot control signals, and for evaluating the naturalness of the performances generated in real and simulated systems by reaction-skill models.

This thesis formulates the process of building action models from human performance data as a dimension reduction problem. Characterizing action skills involves determining the lower-dimensional manifolds, within the very high-dimensional space of possible actions, upon which human performances actually tend to lie. This manifold or constraint surface is determined by building two mappings: one from a high-dimensional "raw-data space" to a lower-dimensional "feature space," and another from the feature space back to the raw-data space.

A best-fit trajectory is arguably the best one-parameter model for a typical human action. A new method is developed to find such a trajectory by fitting a curve to sampled position and velocity performance data. A new spline smoother is derived which can be used within a specially-adapted version of the principal curves algorithm to find a best-fit curve through phase space.

The methods investigated in this thesis are evaluated by using them to model a human grasping motion, to recognize hand gestures in a letter-signing application, and to analyze data collected in a robot teleoperation experiment. These experiments show the effectiveness of local nonparametric methods over global parametric methods, and also show that using both position and velocity information results in better models of action trajectories than using position information alone.

Acknowledgements

I am especially grateful for the guidance and patience of Professor Yangsheng Xu throughout my graduate studies, and for encouraging me on to completion from afar. Garth Zeglin, Henry Schneiderman, Michael Nechyba, and Marcel Bergerman all provided excellent help and suggestions over the years. Anne Murray provided the valuable experimental data and photographs for Chapter 6, and her patience and determination to get these results despite having to fix seemingly every piece of equipment in the Advanced Mechatronics Laboratory was inspirational. I would like to thank Matt Mason and the MLab for inviting me to their thought-provoking weekly meetings, and the excellent support staff and administration of the Robotics Institute for their assistance.

Working with Professor Xu and the members of the CMU Space Robotics Laboratory on $(DM)^2$, $(SM)^2$, underactuated manipulators, and teleoperation was a rewarding and highly educational experience. This work forms the basis for my practical understanding of the robotics field. Ben Brown, of course, is the keystone who designed and built the robots and support hardware, and who keeps everything working.

I would like to thank my family for their many years of support, and especially my grandparents for the TI-99/4A which started me on the happy road to computer science and engineering. Finally, I would like to thank Abigail for her suggestions and encouragement, and for quietly doing far more than her share of so many things while I systematically neglected everything except this thesis.

This work was supported in part by a Pennsylvania Space Grant Fellowship, and by a Department of Energy Integrated Manufacturing Predoctoral Fellowship.

Contents

List of Figures

Chapter 1

Introduction

1.1 Learning action models from human demonstrations

This thesis is a study of methods for learning action skills from human demonstrations. The goal of action learning is to characterize the state space or action space explored during typical human performances of a given task. The action models that are learned from studying human performances typically represent some form of prototypical performance, and also characterize the ways that the human's performance of the action tend to vary over multiple performances due to external influences or stochastic variation. Action models can be used for gesture recognition, for creating realistic computer animations of human motion, for detailed study of an expert performer's motion, for building feed-forward signals for complex control systems around which custom feed-back controllers can be designed, and for evaluating the naturalness of the performances generated in real and simulated systems by reaction-skill models.

Although a great deal of work in robotics has gone into the study of methods for modeling human *reaction* skills, much less work has so far been done on the important related problem of modeling human *action* skills. Action learning is the characterization of open-loop control signals into a system, or the characterization of the output of either an open-loop or closed-loop system. While reaction learning generates a mapping from an input space or state space to a separate action/output space, action learning characterizes the state space or action space explored during a performance. Given

data collected from multiple demonstrations of task performance by a human teacher, where the state of the performer or the task-state is sampled over time during each performance, the goal is to extract from the recorded data succinct models of those aspects of the recorded performances most responsible for the successful completion of the task. Reaction learning focuses on the muscle control a dancer uses to move his or her body, for example, while action learning studies the resulting dance. The methods presented here for action learning are based on techniques for reducing the dimensionality of data sets while preserving as much useful information as possible.

The original motivation for this work is for applications involving skill transfer from humans to robots by human demonstration. Instead of explicitly programming a robot to perform a given task, the object is to learn from example human performances a model of the human action skill which a robot may use to perform like its teacher. Consider a remote teleoperation scenario where a human expert on Earth is guiding the operation of a distant space robot performing a complex manipulation task. Instead of attempting closed-loop control over a communication link with a latency of several seconds, it would be more useful to use pre-built models of the human's low-level action skills. The models could be used locally at the teleoperation station to recognize and parameterize the operator's individual low-level actions as he or she controls a virtual-reality robot model in performance of the task. Concisely parameterized symbolic descriptions of the operator's actions can be sent to the remote robot as they are recognized. Provided that we have already transfered the skill models to the robot, it can then execute under local real-time supervision the actions desired by the operator.

Outside such a scenario, the methods presented in this thesis have a more general set of applications. Models of human action can be used for animation in video games, in movies, or for human-like agents for human-computer interaction (HCI); for analysis of what makes an expert's performances of a given task more successful than others; and for prescriptive analysis of how a novice might make their performances more like an expert's (e.g., how can I make my backhand more like that of Pete Sampras). Because this thesis focuses on techniques for extracting human skill models and has not yet been extended to robotic performance of the tasks characterized by the models, the results here are more immediately useful for gesture recognition and these more analytical applications. The connection between the learned skills and robotic performance is the next logical step for study in this research effort.

What does action learning have to do with dimension reduction? Think

about the process of reaching across a table to pick up a coffee mug. When we perform this task, it seems no more complicated than its description: reach toward the mug, grasp the handle, and then lift. We can accomplish this with hardly a conscious thought.[1] To transfer this skill to a robot by the typical process of human to robot skill transfer, however, we need to record our motions in detail. When we instrument our hand and arms to record our motions as we perform the task, the resulting raw data are very different in quantity and quality from our simple description. To record the motion of the palm of our hand, for example, we need to sample three dimensions of position data and three dimensions of orientation information. Recording the motions of our finger joints and wrist typically requires about twenty more channels of information, and several additional channels could be collected for the other joints in our arm. To fully capture the motion of our arm and hand, we would normally sample all these variables many times a second, and write these data to a computer file. The result of recording the motion is a large quantity of high-dimensional data, where the overall structure of the task and the relative purpose and importance of each motion is obscured.

Once we have collected these data, we are left with the question of how to extract from them a useful model of the underlying human action skills. In this thesis we assume a two step approach: (a) high-level analysis of the overall structure of the task to break it into simpler component motions, and (b) learning a typical motion for each low-level component motion, as well as the most significant ways each such motion is most likely to vary from the prototypical motion.

The analysis step isolates those portions of performances which are similar in purpose and execution, and from which it should be easiest to build good models of typical motions and their variations. If we can build good models, we should be able to relate their basic features to their purpose within the structure of the overall task. We may also be able to learn from the low-level models something about the complexity of the skills underlying their performance.

A particular performance can be compared to a given model for gesture recognition, resulting in some measure of how likely it is that the performance belongs to the corresponding class of motions. If the performance does belong to that class, a good action model will let us concisely describe

[1] I know this because I am not capable of conscious thought in the morning before I actually drink the coffee (or tea).

the most important ways that the performance differs from the prototypical motion. This concise description is a reduced-dimension representation of the performance.

The high-level task analysis, the first step of the action-learning process described above, is a simple process in most cases. Because we generally have a solid high-level understanding of how we perform tasks, we can use this knowledge to focus on the low-level action skills which we really want to model using the collected data. In our example, we know that our strategy for picking-up the mug is roughly reach, then grasp, then lift. Instead of building a single model the entire task, we can instead write a high-level program which calls the low-level tasks 'reach,' 'grasp,' and 'lift' as subroutines, and then focus on learning these low-level skills from the collected performance data.

Because the analysis process is not difficult for the skills we look at in this thesis it will not be the topic of study here, although some research related to this problem is discussed in Section 1.3. Writing the high-level strategy for use by a robot and design of a high-level architecture for implementing such a strategy are interesting problems which I discuss in Appendix A.

Once we have performed the high-level analysis, we can focus on learning models for individual low-level action skills. Each low-level action will have an associated set of training data which is a subset of the overall performance data. The goal is to build a description of the action skill which is as simple and straightforward as the action itself, in a form which can adequately explain the associated high-dimensional training data. For example, although our raw data from the grasping of the mug handle has on the order of 30 data channels indicating the state over time of the many joints in the hand and arm, the hand configurations actually observed during the performances can be represented using only a few task-specific parameters. This is because some parts of the motion will be highly consistent for each grasping motion: we normally grasp the handle with our palm vertical, our thumb near the top of our hand, our wrist firm, and we usually curl all the joints in our fingers in unison as we grasp the handle. These consistencies in the motion of the hand can be modeled as linear or non-linear constraints between the different dimensions of the raw-performance data, and they effectively restrict the intrinsic dimensionality of our grasping motion. Once we have accounted for and represented these constraints, we can accurately describe the position of the hand with a simple parameterization such as the position of our fingertips as they approach the handle, how spread-apart they are and how closed

around the handle. Such a description is similar to our own understandings of the motion.

Given a library of low-level task descriptions of this type, parameterized representations of the typical configurations for each given motion, we can specify a performance of an action by supplying the name of a skill model and a set of parameters. There should be few parameters for adjusting a simple skill, and more parameters for adjusting a skill that is more complex. For the grasp, we might want to specify "use the grasp skill, and keep the fingers close together." In building a model of a low-level skill, then, we need to be able to do several things: (a) we need to be able to isolate the *intrinsic dimensionality* of the skill, meaning the number of parameters we need to adequately describe a given performance with respect to a particular skill model, (b) we need to be able to map a high-dimensional raw state description of a given performance to a description with this intrinsic dimensionality, and (c) we need to be able to map a lower-dimensional description back to a corresponding state description in the space of the raw data. The next section discusses how these tasks are formulated in this thesis. In this work, we address (b) and (c) for parametric and nonparametric models, and do some work on addressing (a) for parametric models. Although some potential approaches for addressing the determination of intrinsic dimensionality in nonparametric models will be discussed, the full investigation of this problem is still an issue for future research.

1.2 Formulation of the dimension reduction problem

This thesis deals with the problem of finding a low-dimensional representation for high-dimensional data sets collected from multiple human performances of a given task. Such performances consist both of some motions which are essential for successful task completion and some which are inessential. Because all the data correspond to successful performances of the task, the essential motions will always be present, and will probably be more consistent and less stochastic in nature than the inessential motions. In our example from the previous section, when we reach across a table to grasp the coffee mug, the essential aspects of the performance are the end point of our hand's trajectory toward the handle of the mug, the orientation and configuration

of our fingers when we grasp the handle, and the force closure of our final grasp. The inessential aspects are the particular path by which our hand moves to the handle, the orientation and configuration of our fingers in the earlier parts of our reach, the overall speed of the motion, and the small jitters in our hands and fingers over the course of the the motion.

To assist in the extraction of the essential aspects of the performance, we will first need to isolate the low-level skills most suitable for learning, and we need to supply training data which we believe most directly shows the important aspects of the performance. In this case, the low-level skills may be a particular kind of reach and a specific kind of grasp, and the learning algorithm may be told to look at the motion of the hand relative the mug rather than to a global frame of reference.

We formulate the extraction of the essential aspects of the performance as a dimension-reduction problem. Multiple human performances of a given task are recorded by sampling the performance state over time. Each resulting sample is an m-dimensional vector, where m may be a fairly high number (e.g., a Cyberglove records roughly 20 channels of data about the configuration of the finger joints in the hand, and tracking position and orientation of the hand requires six more channels). Given $\mathbf{X}_{(m\times n)}$, the training data matrix containing some or all of the samples from the example performances, the dimension reduction process generates a mapping $\mathbf{g} : \mathbb{R}^m \to \mathbb{R}^p$ $(p < m)$ to a p-dimensional space which preserves as much information as possible about each point, and a second mapping $\mathbf{h} : \mathbb{R}^p \to \mathbb{R}^m$ which maps feature points back to the original. We will call the input space $S \subseteq \mathbb{R}^m$ *raw data space*, and space $S^* \subseteq \mathbb{R}^p$ *feature space*. These spaces and the mappings between them are summarized in Figure 1.1. We optimize $\mathbf{g}$ and $\mathbf{h}$ such that mapping training data $\mathbf{X}$ into and back from the p-dimensional feature space results in a faithful reconstruction: $\|\mathbf{X} - \mathbf{h}(\mathbf{g}(\mathbf{X}))\|$ is minimized over $\mathbf{g}$ and $\mathbf{h}$.[2]

After this optimization,

- $\mathbf{g}$ encodes information about the similarities between the points in training data $\mathbf{X}$,

- $\mathbf{g}(\mathbf{X})$ encodes the information about how training vectors $\mathbf{x}_i \in \mathbf{X}$ differ significantly from one another,

[2]When $\mathbf{g}$ and $\mathbf{h}$ are written with matrix arguments, they operate on each individual column-vector of the matrix.

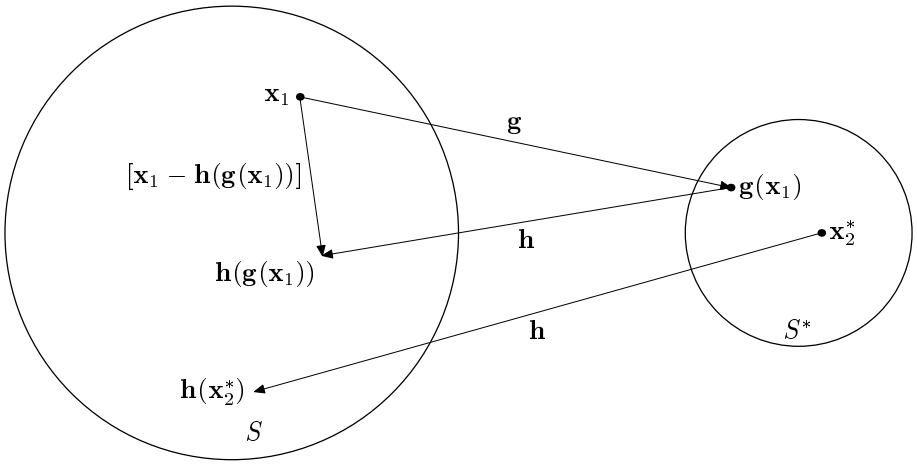


Figure 1.1: Mappings between raw data space S and feature space S^*

- the image $\mathbf{h}(S^*) \subset S$ is the p-dimensional manifold in the m-dimensional raw data space within which or close to which the training vectors lie, and
- $[\mathbf{X} - \mathbf{h}(\mathbf{g}(\mathbf{X}))]$ is the set of residual vectors by which the low-level representation fails to model the high-level data.

If the norms of the residual (error) vectors are generally small, then these vectors likely encode inessential random motions in the task demonstrations. If the norm of the error vector for a given point is much higher than that of most, then it is not typical of the training data set. Moreover, if the norms of most error vectors are high, then either $\mathbf{g}$ and $\mathbf{h}$ are significantly sub-optimal, or the specified dimensionality for the feature space is lower than the intrinsic dimensionality (p) of the training set.

In addition to providing us with a great deal of information about the nature of the sub-task being analyzed, the method of dimension reduction provides a low-dimensional feature space in which to plan robotic performances of the sub-task. If we are able to generate mappings of sufficient quality, then performances planned within this feature space may closely resemble human performances.

An important special case of dimension reduction is modeling using a one-dimensional parameterization. We will discuss in Section 3.2 that the best one-dimensional parameterization of an action skill is a variable describing a

temporal ordering of points. The process of building such a parameterization is *trajectory fitting*. If $\mathbf{g}$ and $\mathbf{h}$ are smooth mappings to and from a one-dimensional feature space, then the result of smoothly varying that single parameter over time from a starting value to a final value, and projecting that value into the raw performance space using mapping $\mathbf{h}$, can represent a "best-fit" model of the action. Another way to describe the fitted trajectory is "the one-dimensional model most likely to have generated training data $\mathbf{X}$."

The remainder of this chapter will review research related to the thesis. Chapter 2 will investigate the use of general purpose parametric modeling methods to create mappings $\mathbf{g}$ and $\mathbf{h}$ that minimize modeling error $\|\mathbf{X} - \mathbf{h}(\mathbf{g}(\mathbf{X}))\|$. Local modeling based on general purpose non-parametric methods is the focus of Chapter 3, which discusses the advantages that local models (i.e., as opposed to global parametric models) hold for human performance modeling. Chapters 4 and 5 present adaptations to these non-parametric methods for the specific purpose of modeling human performance. I derive a spline smoother which can build best-fit trajectories of human performance data through phase space in Chapter 4, and I adapt the principal curves algorithm to use this smoother in Chapter 5. Chapter 6 demonstrates the use of these methods, particularly the method from Chapter 5, for analyzing data from a telerobotics experiment. A very different kind of action model based on hidden Markov models is presented in Chapter 7, which is specialized for the purpose of gesture recognition. Chapter 8 summarizes the conclusions and contributions of the thesis.

1.3 Related research

This section relates the work presented in this thesis to several areas of the robotics and machine learning literature. First, we describe the relationship between the high-level analysis described in Section 1.1 and current work on task-level recognition of human actions for learning assembly plans and for building symbolic descriptions of human motions. Next we survey some the current work on skill-based robot programming, and describe how the conceptions of skills or primitives in this work relate to the low-level action models used in this thesis. Because action modeling is formulated as a dimension reduction problem in this thesis, we review current work on dimension reduction, particularly those methods based on local data models. Finally,

we describe how potential uses of action models relate to current applications of reaction models, and note a few other applications which could potentially benefit through the use of the use of action models.

High-level analysis Section 1.1 divides the process of action learning into two steps: high-level analysis, and learning individual low-level actions. A general task-level strategy or plan is inferred from watching one or more human performances, and the set of low-level actions from which this strategy may be constructed is identified. This is an inverse and potentially complimentary process to the traditional task planning problem, where a strategy or plan of low-level actions is deduced from a task description in a "task-level language" such as described by Lozano-Pérez [41].

The high-level analysis process is relatively straightforward for the tasks examined in this thesis. Thus the problem of decomposing tasks into low-level actions is not a focus of this research. During the process of performing experiments, however, subtasks within full task performances were recognized, labeled, and graded by hand using computer animation interfaces to represent the collected performance data (e.g., Chapter 6). Research that could be useful in automating this labor intensive process to some degree for experiments in the manipulation domain includes the work of Ikeuchi et al. [32, 34] and Hasagawa et al. [27] on using vision systems to extract assembly plans from human demonstrations. Tung and Kak [78] achieve a similar goal of recognizing and parameterizing low-level actions using a DataGlove.

The process of task recognition for extracting assembly plans from human performances is focused on the effects of human actions. Ikeuchi et al., for example, compare "before" and "after" images from human assembly actions. When we are modeling human actions, however, it is often more important to identify the actual human *motions* rather than the *effects* of these motions. For recognizing whole-body human motions, Davis et al. [15] use Motion Histograms, while Wilson et al. [82] use hidden Markov models (HMMs) of visual data. Azoz et al. [6] use a method based upon the extended Kalman filter [6]. Hand motions may be identified using techniques for sign-language recognition, such as the work of Starner [71], which uses HMMs with vision information. This thesis describes two methods for recognizing sign-language with data from an instrumented glove: one using HMMs (e.g., Chapter 7), and the other using a most-likely trajectory model (Section 3.5). Methods for real-time visual interpretation of hand gestures are reviewed in [57].

Primitives and skills The robotics literature often uses the term "skill" or "primitive" to describe a unit of functionality by which robots achieve an individual task-level objective. A "skill" is often defined as an ability to use knowledge effectively and readily, and the effectiveness of a skill for attaining a given desired result is its primary attribute [74]. The development of high-level strategies as combinations of these units is often referred to as "skill-based programming." Examples of this approach are often found in the area of dextrous manipulation, such as the work of Michelman and Allen [44], and Nagatani and Yuta [52]. Skills or primitives are also useful for encapsulating expert knowledge and for the software engineering purpose of making this expertise simple to interface into working systems by task-level programmers, as demonstrated by Morrow et al [46, 47, 48] and Archibald and Petriu [2]. In the robotics literature as a whole, they are used as symbolic units of description for computer representations of task-level plans, and for mapping to human-language descriptions (e.g., "grasping," "placing," "move-to-contact") for human understanding, communication, and reasoning. In the manipulation domain, these primitives are often further defined as sensorimotor units for eliminating motion error due to modeling uncertainty in the task environment.

In contrast to this work in the manipulation domain, this thesis will focus on skills or primitives (the terms "low-level actions" or "component motions" will also be used) as descriptions of actions instead of as a mapping from sensed task state to actions. Thus, we will be characterizing classes of motions or subspaces of action space which correspond to hand movement when reaching, finger movement when grasping, or body movement when walking. This is more closely related to work like that of Yang et al. [88], which uses a hidden Markov model representation to build a model of a "most-likely performance" from multiple example performances.

Dimension reduction and local learning As discussed in Section 1.2, our approach to action modeling is to build mappings to and from reduced-dimension representations of human action data. Most work on skill learning deals with reaction learning (i.e., control) rather than action learning. One notable exception is the work of Bregler and Omohundro [9]. They present a technique for representing constraint surfaces within high-dimensional spaces. Points are sampled from the constraint surface, which is a low-dimensional subspace of the possible space of points. K-means clustering is used to gen-

erate a representative set of cluster centers for these sampled points. In the region containing each cluster, local principal component analysis (PCA) is used to linearly approximate the constraint surface and determine its local dimensionality, and the constraint-surface approximation at a given point is a blended approximation formed from the local PCA models of the nearest cluster centers. Expectation-maximization is used to refine the global model defined by the combination of all the local models. Bregler and Omohundro use this method in a speech recognition application to model a speaker's lip motions. A "snake" representation is used to model the contour of the speaker's lips. This representation consists of a chain of 40 points which track the edges of the speaker's lips within a video image, and the image coordinates of these 40 points can be represented as a single point in an 80-dimensional space. The model of the lip's motion for a given word is the surface within this huge space within which the snake model is constrained while the word is spoken. Although this method for building a reduced-dimension representation of high-dimensional data from a set of distinct local models seems fundamentally different from the methods used later in this thesis such as principal curves, similarities between the blended PCA models and the formulation of the nonparametric smoothers used within the principal curves algorithm actually form a strong relationship between them. Tibshirani's formulation of the principal curve [76] and Tipping and Bishop's mixtures of probabilistic principal component analysis [77] provide an interesting connection between such mixture models and the theory presented in Chapter 3.

Zhang and Knoll [89] present an interesting reaction-learning method based on a global parametric model of a high-dimensional input signal. They present a visual servoing application whereby an eigenspace representation is used to reduce the dimensionality of the image used to control a robot. This reduced-dimension representation is input to a fuzzy rule-based system based on B-splines, which is trained to control the robot for performing simple manipulation tasks.

Chapter 2 of this thesis presents global parametric methods for action learning, including principal component analysis [33] and Kramer's nonlinear principal component analysis [36], while Chapters 3, 4, and 5 focus on local, nonparametric methods. In the comparison of the two, it will be demonstrated that the nonparametric methods have many practical and theoretical advantages. Schaal [63], in the context of learning nonlinear transformations for control, comes to similar conclusions. He is concerned with finding learn-

ing methods which can enable autonomous systems, particularly humanoid robots [62], to develop complex motor skills in a manner similar to humans and other animals. He concludes that nonparametric models are more appropriate than parametric models when the functional form of the mapping to be learned is not known *a priori*, though the nonparametric methods can be more computationally expensive on non-parallel hardware.

Atkeson et al. [4] survey the literature in locally weighted learning, and in the process present a good summary of the available methods for creating local models, including local regression and scatter-plot smoothers. In [5] they survey the use of these locally weighted learning techniques for control: mapping task-state information to an appropriate action. Schaal et al. [64] compare several of these local modeling technique using Monte Carlo simulations, and find that locally weighted partial least squares regression [20, 85] gives the best average results.

One concern mentioned in the locally-weighted learning surveys [4, 5] is the problem of high-dimensional input spaces. It is simply infeasible to collect enough training data to fully characterize all regions of high-dimensional space, and it is difficult to blend different local models together because all points begin to look equidistant in these spaces [67]. Nevertheless, Vijayakumar and Schaal [79] present a method called Locally Adaptive Subspace Regression (LASS) for learning mappings in these high-dimensional spaces. Their approach is based on empirical observations that "despite being globally high dimensional and sparse, data distributions from physical movement systems are locally low-dimensional and dense." LASS learns functional mappings in high-dimensional spaces by using locally-weighted principal-component analysis to reduce the dimension of the local model before attempting to fit the model. Its input space is a combination of local PCA patches which looks similar in some ways to the constraint surfaces of Bregler and Omohundro [9].

An important problem for building reduced-dimensional representations of data sets is choosing the dimensionality of the model. Minka [45] gives one approach to doing this for models based upon principal component analysis (PCA). His work uses the interpretation, from Tipping and Bishop [77], of PCA as a maximum-likelihood density estimation.

Uses for action models In addition to presenting methods for building models of human actions, this thesis presents example applications including

the representation and analysis of a human grasping motion, analysis of a telerobotics experiment, and two gesture recognition experiments. These are only a small subset of the potential applications of this research, however. Most of the potential applications for action models are similar to those for reaction or control models.

Although reaction learning is a popular strategy for abstracting primitives from human demonstration data for use in dextrous manipulation, for example with neural networks [35, 40] and hidden Markov models [30], action learning also can build models useful for robot execution [88]. When combined with a system for recognizing a human operator's actions in a virtual environment such as that described by Ogata and Takahashi [56], such models can be used for task-level teleoperation of a remote robot (e.g., [29]). Some motions such as the brush stroke of a painter or the pen stroke of a calligrapher may be very subtle and appropriate for learning by demonstration, but may however be primarily feed-forward in nature (possibly with applied force as one dimension of the action state) and best modeled using action learning rather than reaction learning. For these motions, we probably want to learn not only the typical stroke of the painter or calligrapher, but also the ways which the stroke may vary yet remain typical of the artist. This could allow for appropriate variation in the execution of the skill to make the result look less "robotic."

Action models are also useful for applications in which skills are transferred from a human expert to a student. Nechyba and Xu [53] demonstrate a system whereby an expert's reaction skill, modeled by a cascade neural network, is used to guide a student's learning of a difficult inverted-pendulum stabilization task. They also present a stochastic similarity measure [54] for quantifying the similarity between reaction-task performances, which is useful for evaluating skill transfer experiments. The methods developed in this thesis are particularly applicable to action skill transfer due to their ability to *analyze* the motions of human experts, and their ability to *describe* an expert's nominal performance and likely modes of variation for comparison with student performances. For human motions such as swinging a golf club or throwing a baseball, a student's action can be compared to an expert's motion trajectory in space to form prescriptive suggestions, so an action model built around a best-fit trajectory will have important advantages over a black-box input/output model such as a neural network.

Computer animation is one of the most promising applications for action models. Instead of simply recording the motion of a human actor for

later playback with an animated character, action learning can be used to characterize the full range of motions which are typical of that person's performances. These models could be used to create animations of the corresponding motions (e.g., walking) which are stochastically varied in a "human" manner to give them a more natural look. For research in physics-based animation of human models, action models could be used for evaluating different control methods. For example, Matarić et al. [43] evaluate various methods for controlling the movements of a humanoid torso simulation in performance of the macarena. Their evaluations are based on qualitative and quantitative measures of "naturalness of motion," where their quantitative analysis is based primarily on a measure of end-effector jerk. A distance metric from an appropriate action model learned from human performances could be used to build an improved criterion for the naturalness of these simulated motions.

Chapter 2

Global parametric methods for dimension reduction

2.1 Introduction

In this chapter we look at some methods for dimension-reduction which are not specific to characterizing human performance data sets, but which may however be useful for this purpose. These methods do not look at the human performance data in terms of multiple example trajectories in time, nor do they look directly at the local relationships between example points from similar parts of a given performance or the state space. They rather look at all the points from all the training examples as a single set of vectors in raw data space $\mathbf{X}_{(m \times n)} = [\mathbf{x}_0 | \mathbf{x}_1 | \ldots \mathbf{x}_{n-1}]$ and simply try to map them to a lower-dimensional feature space and back again while preserving as much useful information as possible. Since these methods do not look at the local structure of task performances, we call them *global* methods for dimension reduction.

In Section 2.2 we will review the general topic of parametric methods for global modeling, and in the rest of the chapter we will discuss global parametric modeling of human performance data. We will present three global methods for this purpose: principal component analysis (PCA), non-linear principal component analysis (NLPCA), and a variation on NLPCA called sequential non-linear principal component analysis (SNLPCA). Given training-data $\mathbf{X}$, these methods develop mappings $\mathbf{g} : \mathbb{R}^m \rightarrow \mathbb{R}^p$ to feature space and $\mathbf{h} : \mathbb{R}^p \rightarrow \mathbb{R}^m$ back to raw-data space. PCA generates linear map-

pings. The forms of the mappings resulting from both NLPCA and SNLPCA are non-linear, but they are found in slightly different ways. To demonstrate and compare these methods, we use an example human performance data set described in Section 2.3.

2.2 Parametric methods for global modeling

Reducing the dimensionality of a data set can be thought of as the process of finding correlations between its different variables or dimensions. Strong correlations between dimensions of the data may be used as constraint equations to explicitly reduce the dimensionality of their representation.

The tool boxes of statisticians are full of methods for uncovering these relationships between two or more variables in a given set of data. These methods generally assume that there is some underlying structural relationship between the variables, and also some random error or variation in one or more of the variables. If we have a data set of n observations of two variables x and y, we might first make a scatter plot of the points (x, y) to get a rough idea of how one relates to the other. Figure 2.1(a) on page 18 shows an example scatter plot. Such plots are more difficult to use for higher-dimensional data sets, but we can still learn a great deal about high-dimensional data by plotting two- and sometimes three-dimensional projections.

The most familiar methods for modeling the relationship between the different parts of a data set are parametric methods. These result in parametric models—equations relating one set of variables to another, with a set of parameters which can be adjusted to fit a given dataset. Adjusting one of the parameters of these models tends to change the model over the entire domain, and the effect is generally measured against a single scalar value indicating the goodness-of-fit over the entire set of training data. In this section we review several well-known global parametric methods, then in the rest of the chapter we will describe how similar methods may be used for modeling human performance data.

2.2.1 Polynomial regression

The class of polynomial functions is a commonly-used set of parametric models. They are most useful when we can separate the dimensions of the data we are modeling into a single *explanation* variable x and a set of *response*

variables $\mathbf{y}$. We will assume a single response variable y in this discussion. All the error is assumed to appear in the response variable, while the explanation variable is assumed to be error-free.

We generally assume that the data comes from a process with an underlying systematic relation between the variables, and a separate source of random error. We model the systematic relation using a p-th order polynomial, and the random error with variable ϵ:

$$y = \sum_{j=0}^{p-1} c_j x^j + \epsilon. \tag{2.1}$$

If we assume that error ϵ comes from a Gaussian distribution with an expected value of zero, we can estimate the coefficients $\mathbf{c}^*$ of the polynomial by minimizing the sum of squared error

$$S_p = \sum_{i=0}^{n-1} \left(y_i - \sum_{j=0}^{p-1} c_j (x_i)^j \right)^2. \tag{2.2}$$

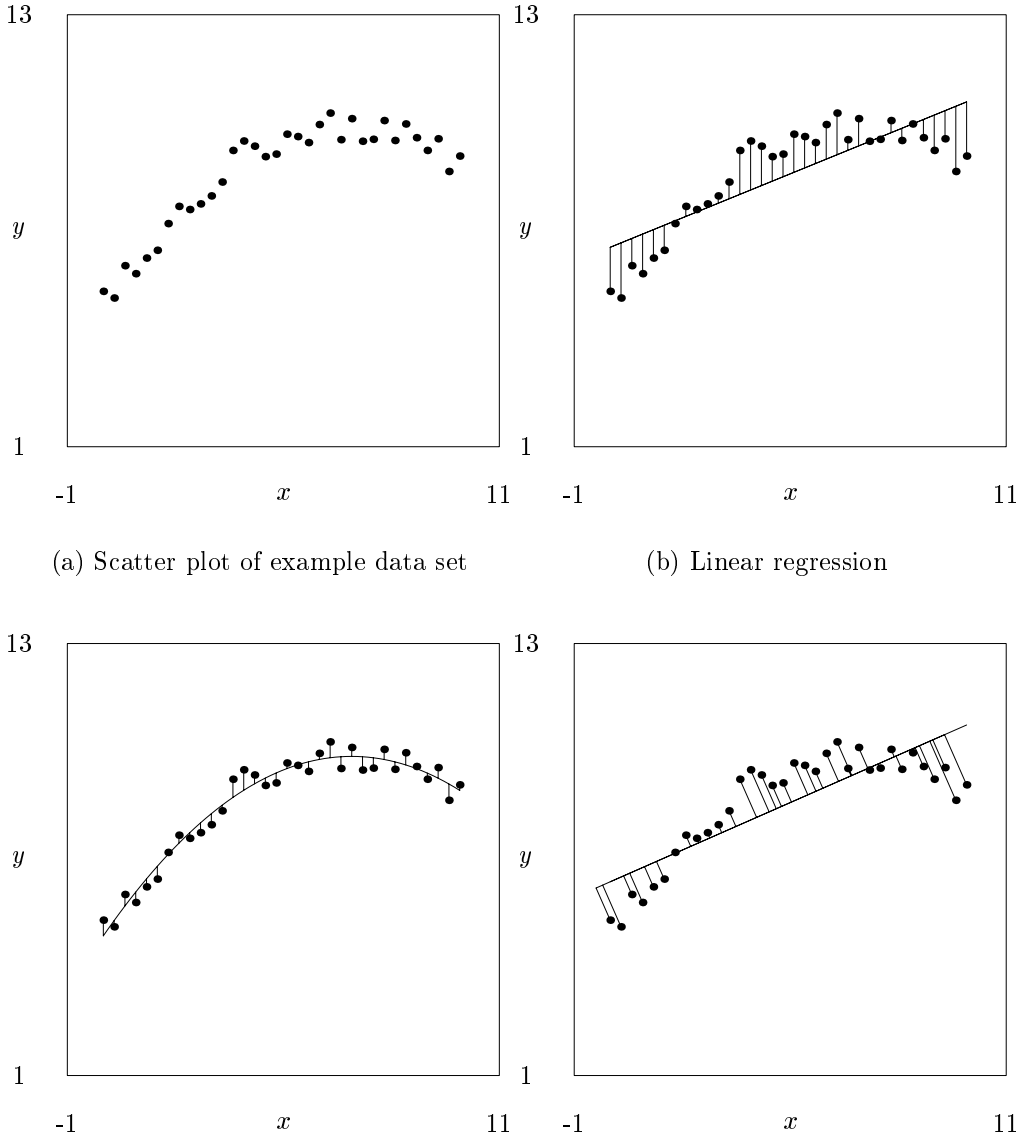
The estimated coefficients of the polynomial $\mathbf{c}$ can be found by solving the linear system

$$\mathbf{M}_{(p\times p)}\mathbf{c}_{(p)} = \mathbf{m}_{(p)}, \tag{2.3}$$

where $\mathbf{M}$ is a symmetric matrix with terms $M_{ij} = \sum_{k=0}^{n-1} (x_k)^{i+j+1}$, and $\mathbf{m}$ is the vector such that $m_i = \sum_{k=0}^{n-1} y_k (x_k)^i$ (although this is not the most efficient solution).

The polynomial most often used for fitting data is a straight line. Figure 2.1(b) shows the result of performing linear regression (i.e., polynomial regression with $p = 2$) on the dataset in Figure 2.1(a). Because we assume that all the error is in the response variable, the error vectors, which connect each data point to its projection on the model line, are vertical. Figure 2.1(c) shows the result of a polynomial curve fit with $p = 3$, a parabolic curve fit. Again, the error vectors are vertical.

Since we have an explicit parameterization (or explanation) variable, polynomial regression of a dataset with a multi-dimensional space of response variables is equivalent to performing regression separately on each individual response variable.



(a) Scatter plot of example data set

(b) Linear regression

(c) Parabolic curve fit

(d) First principal component

Figure 2.1: Various parametric models of an example data-set. Each model minimizes the sum of squared errors, where the errors are length of the vector between data-point and its projection onto the model.

2.2.2 First principal component

Sometimes we want to find the best linear model for a dataset without first choosing an explanation variable, or we might want to do a linear regression assuming that the variance of the error in the explanation variable is the same as that for the other variables. The first principal component is the proper model for these cases. If we assume for multi-dimensional dataset $\mathbf{Y} = \{\mathbf{y}_i\}$ the model

$$\mathbf{y}_i = \mathbf{u}_0 + \mathbf{a}\lambda_i + \epsilon_i, \tag{2.4}$$

and that covarience$(\epsilon_i) = \sigma^2\mathbf{I}$, then the least-squares estimate of slope vector $\mathbf{a}$ is the first principal component. Figure 2.1(d) shows the first principal component for the example dataset. Note that the error vectors are orthogonal to the model line. This is due to the fact that error is being minimized in both dimensions simultaneously. The error vectors are parallel to the second principal component.

The parameterization variable λ for the principal component is not part of the original problem formulation—it is newly introduced by the modeling equation (2.4). It can be considered a "discovered" (or "latent") explanation variable. If we rotate Figure 2.1(d) slightly clockwise to make the first principal component line horizontal, then the error bars will be vertical like in the regression plots. The regression against explanation variable λ in the rotated plot results in the horizontal-line model, a rather boring result because the principal component already explained-away the correlation.

Principal component analysis (PCA), also known as the Karhunen-Loève transform [33], is a well-understood and commonly used method which can be performed on data sets of arbitrary dimension. Because it does not need an *a priori* explanation variable to generate its models, because it is computationally inexpensive, and because it generates a linear model which is easy to understand and interpret, it is a useful first modeling technique to apply to human performance data. We will describe its use for this application in Section 2.4.

2.3 An experimental data set

In Chapter 1 we described how the process of human to robot skill transfer typically involves recording several individual human task performances

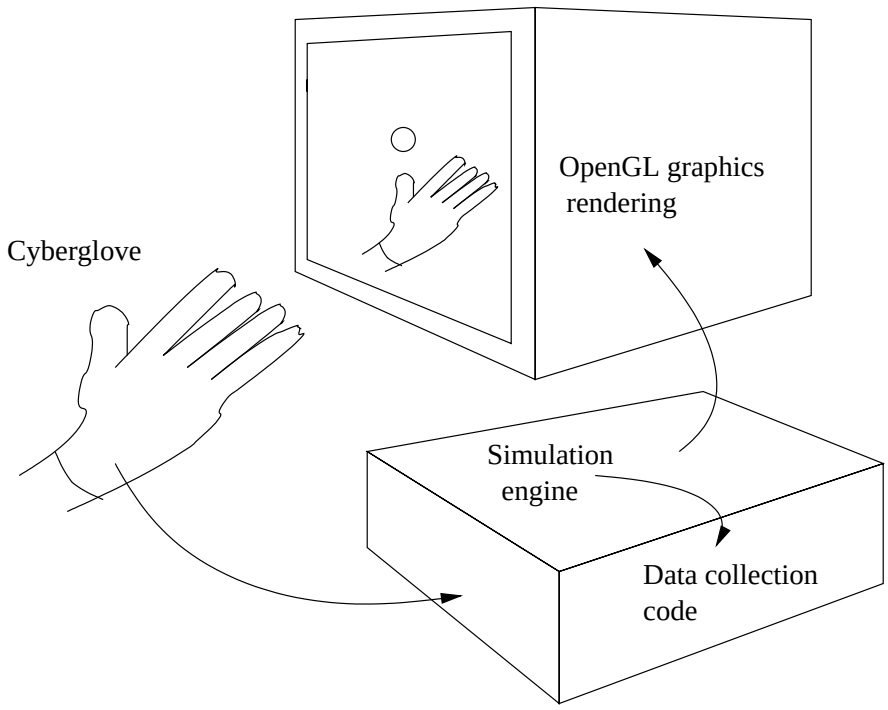


Figure 2.2: "Virtual reality" skill demonstration system

by sampling performance state over time using signals from various input devices. Some example input devices include a Cyberglove or other instrumented glove, a Polhemus or other 6-DOF tracker, a joystick, a mouse, a haptic interface, or other measuring devices such as visual trackers, instrumented cockpits, etcetera. Although there may be a large number of dimensions in the resulting representations of recorded performances, the actual intrinsic dimensionality of the underlying human skill is often much lower. In this section, we present an example set of recorded human task performances whose raw-data space has a large number of dimensions, but which should be represented more appropriately using only a few independent parameters. This data is collected from the grasping phase of a ball-catching task which was demonstrated in a simple virtual environment.

A human subject performed a number of catches of a virtual ball using the system outlined in Figure 2.2. The input device was a Virtual Technologies Cyberglove which measures 18 joint angles in the fingers and wrist of its wearer, with a Polhemus sensor which returns the overall position and

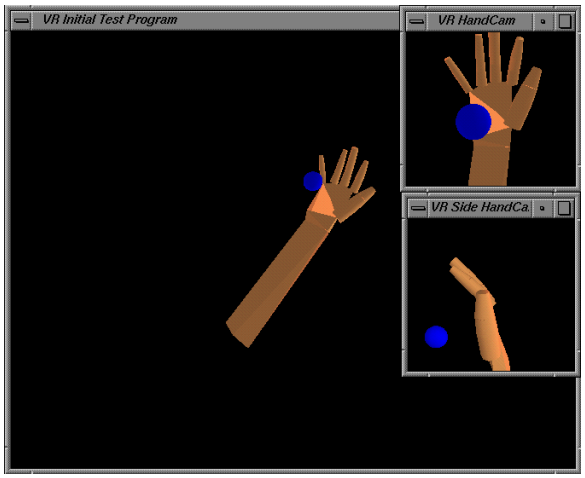


Figure 2.3: Graphical feedback

orientation of the wearer's hand. The feedback to the user was a rendering of the hand and ball from several perspectives, on the graphical display monitor of an SGI workstation (as shown in Figure 2.3). The recorded data was interpolated and resampled evenly at 10Hz, and stored in a performance database for later retrieval and analysis.

We manually segmented the performances into approach and grasp phases, and subjectively graded each recorded grasp on a scale from 0-9. We selected the vectors representing the joint angles in the fingers and wrist during the grasp phase of all catches for which the grasp satisfied a minimum grade. This generated a data set of 461 vectors, each representing a hand configuration of 18 joint-angles during grasping motions. These vectors were randomly allocated into a training set of 155 points, a cross-validation set of 153 points, and a test set of 153 points.

2.4 Principal component analysis for modeling human performance data

As mentioned in Section 2.2.2, principal component analysis (PCA) [33] is a well-understood and useful method for modeling data sets. When applied to a set of multidimensional vectors, it finds a linear mapping between them and each lower-dimensional space such that when the vectors are mapped

to a lower-dimensional space and then mapped back to the original space, the sum-of-squared error of the reconstructed vectors is minimized. In the compressed representation of a vector, we can consider each dimension a separate *feature*, and the value of that dimension of the feature-vector a *feature-score*.

To perform principal component analysis of a set of n m-dimensional zero-normed vectors $\mathbf{X}_{(m\times n)} = [\mathbf{x}_0|\mathbf{x}_2|\ldots|\mathbf{x}_{n-1}]$, we find the eigenvalues λ_i and eigenvectors $\mathbf{v}_i$ of the symmetric matrix $\mathbf{X}\mathbf{X}^T$. To generate a p-dimensional feature-vector representation of a m-dimensional vector[1] $\mathbf{y}$ (where $p < m$), we form a $m \times p$ matrix $\mathbf{V}_p = [\mathbf{v}_0|\ldots|\mathbf{v}_{(p-1)}]$ where $\mathbf{v}_0 \ldots \mathbf{v}_{(p-1)}$ are the eigenvectors corresponding to the p largest eigenvalues. Then the *feature-vector* $\mathbf{y}^*$ is

$$\mathbf{y}^* = \mathbf{g}(\mathbf{y}) = \mathbf{V}_p^T\mathbf{y}, \tag{2.5}$$

and the reconstructed approximation $\tilde{\mathbf{y}}$ of the original vector $\mathbf{y}$ from feature-vector $\mathbf{y}^*$ is

$$\tilde{\mathbf{y}} = \mathbf{h}(\mathbf{y}^*) = \mathbf{V}_p\mathbf{y}^* = \mathbf{V}_p\mathbf{V}_p^T\mathbf{y}. \tag{2.6}$$

The value of the i-th element of the feature vector $\mathbf{y}^*$ is the magnitude of the projection of $\mathbf{y}$ upon the corresponding eigenvector $\mathbf{v}_i$. Because eigenvalue λ_i is the variance of training data in the direction given by the eigenvector $\mathbf{v}_i$ [7], the eigenvalue indicates relative significance of feature score y_i^* for representing vectors from a distribution similar to the training data.

Performing this analysis on the data set from Section 2.3 gives us a set of eigenvectors which attempt to explain the data set in terms of linear dependencies between its dimensions. The ability of the first 5 eigenvalues to represent the data set is summarized in Figure 2.4. The values shown are the relative magnitudes of the residuals,

$$\%\text{-Err}_p = \frac{\|\mathbf{Y} - \mathbf{V}_p\mathbf{V}_p^T\mathbf{Y}\|}{\|\mathbf{Y}\|}, \tag{2.7}$$

where $\mathbf{Y}$ is the set of testing vectors from the experiment (independent from the set $\mathbf{X}$ used to generate the eigenvectors), and $\|\cdot\|$ is the Frobenius (L_2) norm. Figure 2.4 clearly shows that the first two linear features are nearly equal in importance for explaining the configuration in the test data, while the remaining features are much less significant.

[1] In this chapter, we will use $\mathbf{x}$ to refer to a vector which is in the training-set, and $\mathbf{y}$ to refer to a vector which is not in the training-set.

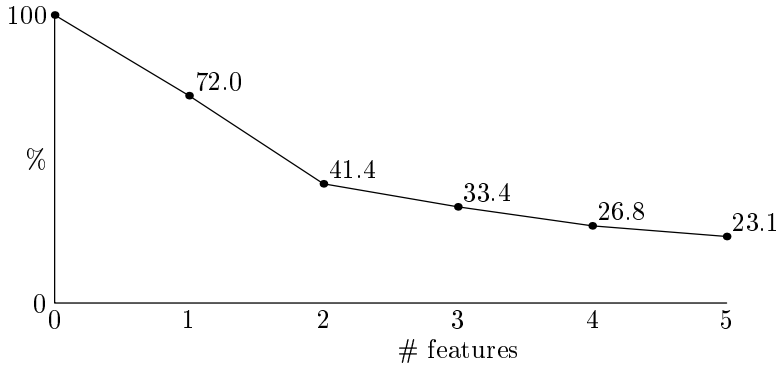


Figure 2.4: Error unexplained by p features of PCA.

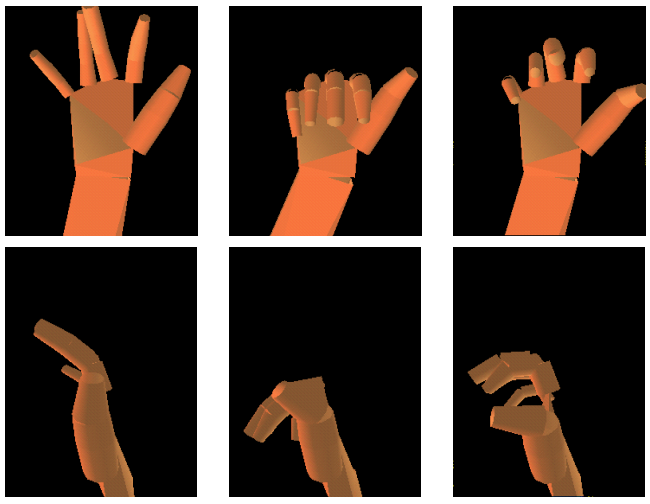


Figure 2.5: PCA analysis of grasp gesture. Initial position, and first and second principal "eigenhands," front and side views.

Figure 2.5 provides a graphical illustration of the effects of these first and second linear principal components, which we might call "eigenhands."[2] These hand configurations are generated by varying the components y_0^* and y_1^* of the feature vector, and then mapping these features to $\tilde{\mathbf{y}}$ using an interface like the one shown in Figure 2.10 on page 31. We are able to use the slider bars in this interface to look at the separate effects of the two features y_0^* and y_1^* because PCA learns a linear mapping, and thus the effects of the features are independent and additive:

$$\tilde{\mathbf{y}} = \mathbf{v}_0 y_0^* + \mathbf{v}_1 y_1^*. \tag{2.8}$$

While this linearity makes the resulting model easy to analyze and interpret, it also limits the generality of the model.

We can see that the main effect of the first principal component is to bring the fingers closer together and to bend the fingers at the knuckles (mcp), while the effect of the second principal component is to curl the fingers at the second (pip) and third (dip) joints. Moreover, we see that although the grasp positions of the hand generated from the first and second eigenvectors look plausible, the best attempt to create an adequate initial (open) position for the grasp using only the first principal component looks less plausible. Closer inspection of this configuration reveals that the pinky and index fingers overlap in space, and that it is thus physically impossible. The problem is the linear nature of the mapping. The first principal component reduces the average sum-of-squares error for all joint-angles during the grasps by fitting a straight line in the space of training configurations, but if the general trend of the performances in state space is not linear, then this principal component will fit the trend of the performances poorly at some stages. Nevertheless, we see from Figure 2.4 that PCA allows much of the 18-dimensional data set to be explained by only a few linear feature values.

2.5 NLPCA

Nonlinear principal component analysis (NLPCA) also attempts to find mappings between a multidimensional data set and a lower-dimensional feature-space while minimizing reconstruction error, but allows the mappings to be

[2]The thumb positions in these diagrams are slightly erroneous due to a sensor which was not working during the experiments.

nonlinear. In contrast to linear mappings (2.5) and (2.6), the nonlinear mappings are of the general form

$$\mathbf{y}^* = \mathbf{g}(\mathbf{y}) \tag{2.9}$$

$$\tilde{\mathbf{y}} = \mathbf{h}(\mathbf{y}^*) = \mathbf{h}(\mathbf{g}(\mathbf{y})) = \mathbf{n}(\mathbf{y}). \tag{2.10}$$

If the lower-intrinsic dimensionality of a data set arises from a nonlinear relationship between the different dimensions of the data set, a nonlinear principal component analysis is capable of better representing the original data set with a reduced-dimension representation than would a linear principal component analysis. Several methods proposed for performing NLPCA include the use of autoassociative neural networks, as described by Kramer [36]; principal curves analysis, as described by Hastie and Stuetzle [28], and Dong and McAvoy [17]; adaptive principal surfaces, as described by LeBlanc and Tibshirani [37]; and optimizing neural network inputs, as presented by Tan and Mavrovouniotis [75]. In this section we will focus on Kramer's method for NLPCA, and in Section 2.6 we look at a modification of that method called SNLPCA. These are global parametric methods because for a given neural-network architecture, there is a corresponding parametric equation whose parameters are adjusted to optimize a global measure of goodness-of-fit, and there is no specific relationship between parameters of the equation and local regions of the mapping space.

Kramer's method for NLPCA involves training a neural network with three hidden layers, such as the one shown in Figure 2.6. These neural networks are autoassociative, meaning they are trained to map a set of input vectors $\mathbf{X}$ to an identical set of output vectors. If the second hidden layer, or "bottleneck" layer, has a lower dimension than the input and output layers, then training the network creates a lower-dimensional representation $\mathbf{X}^*$ of the vectors presented to the networks inputs in the form of the activations of the units in the bottleneck layer. The mapping from the input vectors to these activations in the bottleneck layer is the "compression" transform $\mathbf{g}$, and the mapping from the bottleneck activations to the activations of the output units is the "decompression" transform $\mathbf{h}$. Using sigmoidal units of the form

$$\sigma(x) = (1 + e^{-x})^{-1} - \tfrac{1}{2} \tag{2.11}$$

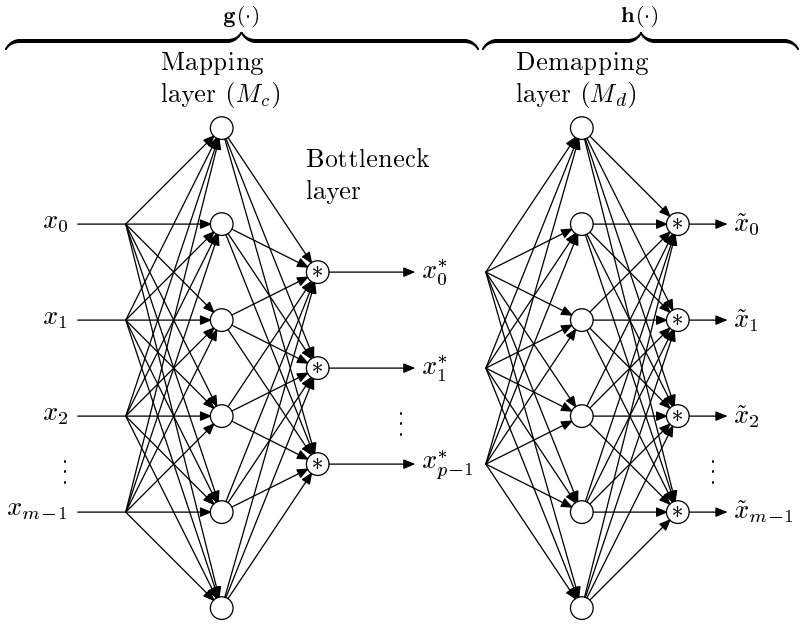


Figure 2.6: Neural network architecture for NLPCA. ◯ indicates a sigmoidal unit, and ⊛ indicates a unit which may be either linear or sigmoidal.

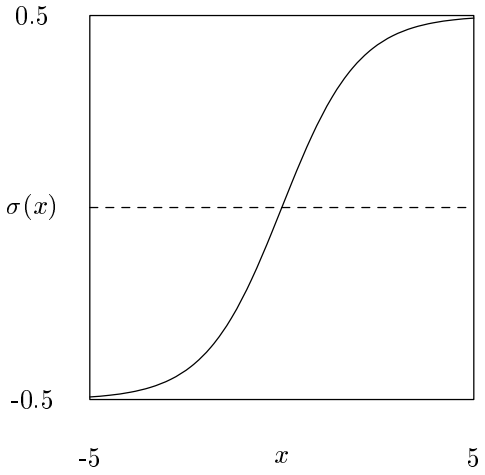


Figure 2.7: Sigmoid function

in the first and third hidden layers (the sigmoid function is shown in Figure 2.7) allows the mapping function $\mathbf{g}$ and de-mapping function $\mathbf{h}$ to take the forms

$$\mathbf{g} : g_k(\mathbf{x}) = \tilde{\sigma}\left(\sum_{j=0}^{M_c} w_{kj}^{(2)} \sigma\left(\sum_{i=0}^{m} w_{ij}^{(1)} x_i\right)\right) k \in 0 \ldots p-1 \tag{2.12}$$

$$\mathbf{h} : h_k(\mathbf{x}^*) = \tilde{\sigma}\left(\sum_{j=0}^{M_d} w_{kj}^{(4)} \sigma\left(\sum_{i=0}^{p} w_{ij}^{(3)} x_i^*\right)\right) k \in 0 \ldots m-1, \tag{2.13}$$

where $\tilde{\sigma}(\cdot)$ may either be the sigmoidal function (2.11) or the identity function $\tilde{\sigma}(x) = x$ depending on whether sigmoidal or linear units are used in the bottleneck and output layers. Given enough mapping units, these functional forms may approximate any bounded, continuous multidimensional nonlinear function $v = f(u)$ with arbitrary precision [14]. Just as PCA defines a linear mapping to and from a reduced-dimension representation which minimizes the sum of squared reconstruction error $\|\mathbf{X} - \mathbf{V}^T\mathbf{V}\mathbf{X}\|^2$ for a given set of vectors, training the weights $\mathbf{W}$ of the autoassociative neural network to minimize the sum of squared error

$$e_p^2(\mathbf{X}) = \|\mathbf{X} - \mathbf{n}(\mathbf{W}, \mathbf{X})\|^2 \tag{2.14}$$

of mapping vectors $\mathbf{x}_i$ to themselves through the bottleneck layer effectively performs a nonlinear principal component analysis of the vectors.

The principal advantage of NLPCA over PCA is its ability to represent and learn more general transformations, which is necessary in cases when one wishes to eliminate correlations between dimensions in a set of data which cannot be adequately approximated by a linear dependency. However, NLPCA also has important disadvantages compared to PCA.

The trade-off for the extra-representational power of the nonlinear mapping functions **g** and **h** is that they cannot be as easily interpreted as the eigenvector-based mappings returned by PCA. In addition, NLPCA tends to require several orders more computation time than linear PCA, and because training the neural network is a high-dimensional nonlinear optimization problem over the weights of the neural network, we can guarantee only a locally optimal solution, unlike the globally optimal solution returned by PCA. In the version of NLPCA presented to this point the relative importance of each output dimension of the compression mapping cannot be determined by the training process, and there is no guarantee that any one of the output dimensions corresponds to a primary nonlinear factor of the training data. However, if an explicitly prioritized factorization is desired, Kramer's sequential NLPCA algorithm (SNLPCA), discussed in Section 2.6, may be used.

We used Kramer's NLPCA method to analyze the data set from Section 2.3. The neural networks were trained using the L-BFGS-B implementation of Byrd et. al [10]. We used a network architecture with linear units for the bottleneck and output layers, and without direct interconnections between the input and bottleneck layers, nor between the bottleneck and output layers. Choosing linear output units for the output layer allowed the network to be trained on data which was not rescaled to fit within the output range of the sigmoidal function (although the data was zero-normed). This gave better results than rescaling the data and using sigmoidal units.

The parameter M, the number of mapping units in the first and third hidden layers of each network, was chosen by a heuristic search over the range $p \ldots \min(\frac{n}{4}, p_M - 1)$ where n is the number of training vectors, and p_M is the smallest possible value for M for which the number of weights in the network N_w will exceed the available number of values in the training matrix, (mn). M must be at least as great as p if there are no interconnections across the mapping layers; otherwise, the mapping layers would be the real bottlenecks. The selection criteria for the value of M was not cross-validation error (although it is used for early-stopping of the weight-optimization algorithm),

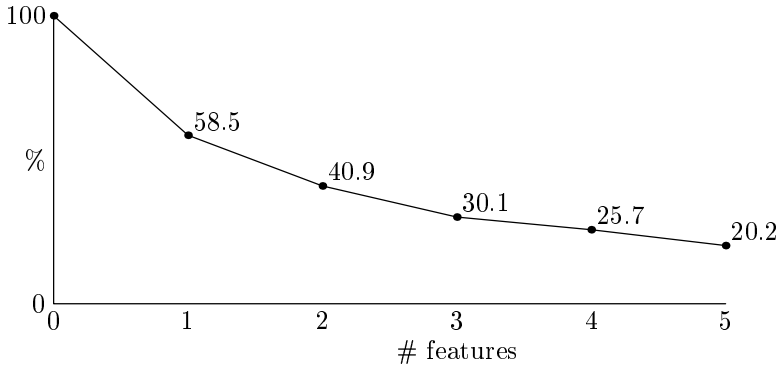


Figure 2.8: Error unexplained by p features of NLPCA.

but rather the information theoretic criterion described in [36]:

$$\text{AIC} = \ln(e/(2m)) + 2N_w/N_d, \tag{2.15}$$

where

$$N_w = m + p + 2M(m + p + 1) \tag{2.16}$$

is the number of weights in the network, $N_d = (nm)$ is the number of training vectors times the dimension of the training vectors, and $e/(2m)$ is the average sum of squares error. This criterion penalizes network complexity, and thus tends to reduce over-fitting of the training data. At least two units are used in the mapping layers, even when $p = 1$, because a network architecture with only a single unit in each of the three hidden layers is degenerate and unable to learn significantly nonlinear principal components.

In Figure 2.8 we show the results of this NLPCA analysis. The effect of NLPCA on residual error for each number of principal nonlinear factors is calculated as

$$\%\text{-Err}_p = \frac{\|\mathbf{Y} - \mathbf{n}_p(\mathbf{Y})\|}{\|\mathbf{Y}\|}. \tag{2.17}$$

The most interesting aspect of the analysis is that although NLPCA explains significantly more of the data set than PCA when $p = 1$, the two methods perform similarly when $p > 1$. This is due to the fact that the grasping motions analyzed are relatively simple and open-loop in nature, due to a lack of

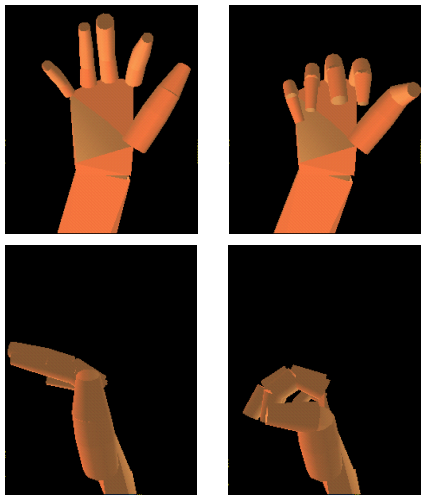


Figure 2.9: NLPCA analysis of grasp gesture. Initial position, and effect of first nonlinear principal component, front and side views

haptic feedback in the virtual environment in which they were demonstrated. We can thus think of each performance as following a nominal grasping trajectory in configuration-space from an open hand to a closed hand, with some stochastic variation. The first nonlinear principal component follows the nominal grasping-trajectory, and since this trajectory is curved, it explains the data in the testing data set better than the first linear principal component. However, since the basic non-stochastic structure of the grasp is adequately explained by the first nonlinear principal component, it is plausible that when a higher-dimensional feature-space is used NLPCA simply optimizes the mutual orthogonality of the resulting features to most efficiently explain the more stochastic nature of residual performance data, and the result is thus similar to PCA.

Figure 2.9 illustrates the first nonlinear principal component from our analysis. These configurations are formed by varying a scalar value y_0^* and mapping it to a hand configuration using function $\mathbf{h}_0 : \mathbb{R}^1 \rightarrow \mathbb{R}^m$. This can be accomplished using an interface with a slider-bar controlling y_0^* as in Figure 2.10. The grasping configuration of the hand looks similar to that generated by the first linear principal component in Figure 2.5, but the initial position generated from the nonlinear principal component is more plausible.

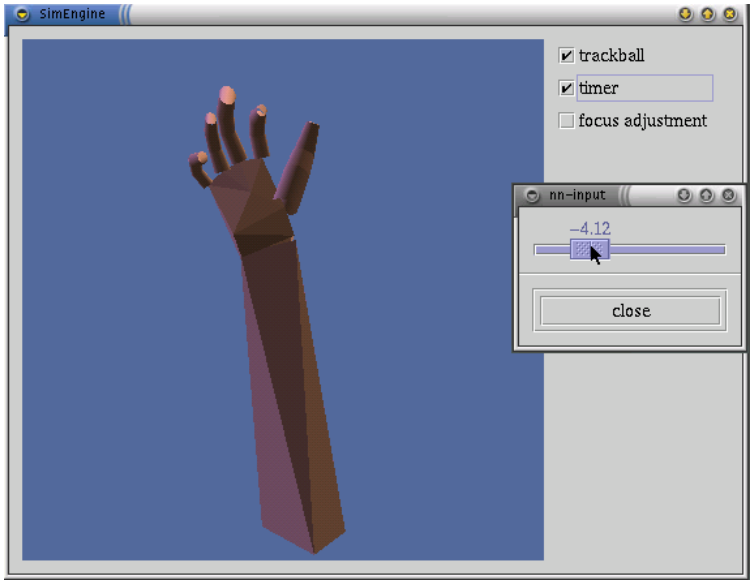


Figure 2.10: Interface for controlling hand configuration using y_0^*

Moreover, the grasping motion resulting from smoothly varying the feature value looks more natural than the motion generated by the principal linear feature.

It should be noted that in this example we started by building a representation for individual hand configurations, but this resulted in a mapping which we could use to animate a nominal grasping performance by smoothly and monotonically varying the nonlinear principal component y_0^* from an initial to a final value. This was possible because the grasping motion is a smooth directed path in configuration space, and because the forms of the mapping functions (2.12) and (2.13) are well suited to learning such smooth mappings. However, we are making a substantial leap here. We are treating a nonlinear regression fit as if it were a best-fit trajectory estimate of the grasping motion—a directed path through configuration space which is typical of the example training performances. Although we have created a one-dimensional parameterization of the grasping motion, this was done using only global information, and our learning techniques learned nothing directly about the local relationships between the points in the data-set and how the hand typically moves between them in configuration space. In Section 2.8, we will discuss the difference further and motivate the methods

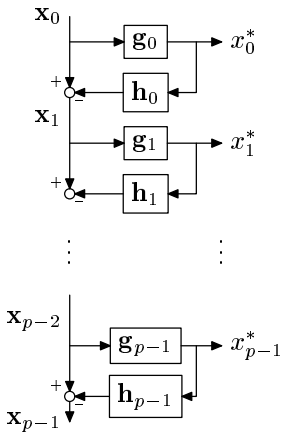


Figure 2.11: SNLPCA computation

presented in later chapters which make use of localized motion information in the training data.

2.6 SNLPCA

Kramer's sequential NLPCA algorithm (SNLPCA) [36], is a modification to the NLPCA method which produces a nonlinear factorization, and where the training process prioritizes each resulting feature as to its relative power in explaining the variations of the training set. SNLPCA performs a series of NLPCA operations, each training a neural network with a bottleneck layer consisting of a single unit. Training such a neural network on the raw performance data set explains as much of the set's variation as can be represented by a single variable, and thus trains the compression part of the network to perform the primary nonlinear factorization of the data set. The next nonlinear factorization should explain the variation in the data set which is not accounted for by the first factorization, and thus a second network is trained on the residuals formed by subtracting each vector of the original data set by its estimate as calculated by the first network. This process, summarized in Figure 2.11, continues until the desired number of iterations has been reached or until enough of the original data set's variation has been explained.

The algorithm is:

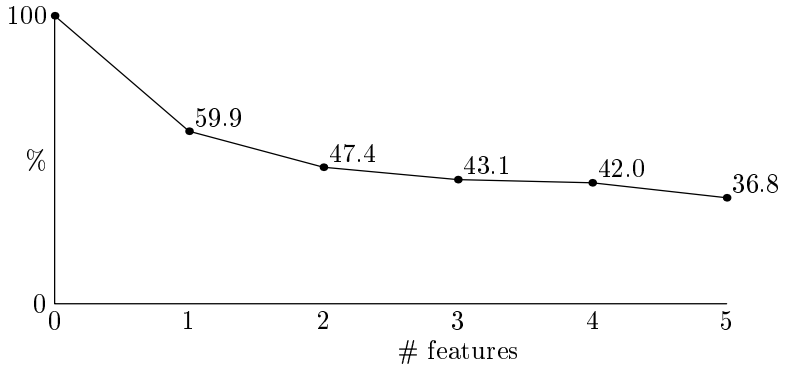


Figure 2.12: Error unexplained by p features of SNLPCA

1. $\mathbf{X}_0 \leftarrow \mathbf{X}, p \leftarrow 0$
2. Loop for p in $0 \ldots (F-1)$ or until $\|\mathbf{X}_p\| < \epsilon$
   (a) Train $\mathbf{W}_{p+1}$ to minimize $\|\mathbf{X}_p - \mathbf{n}_{p+1}(\mathbf{X}_p)\|^2$, where $\mathbf{n}_{p+1}(\cdot) \equiv \mathbf{n}(\mathbf{W}_{p+1}, \cdot)$
   (b) $\mathbf{X}_{p+1} \leftarrow (\mathbf{X}_p - \mathbf{n}_{p+1}(\mathbf{X}_p))$
   (c) Split $\mathbf{n}_{p+1} : \mathbb{R}^m \rightarrow \mathbb{R}^m$ into $\mathbf{g}_{p+1} : \mathbb{R}^m \rightarrow \mathbb{R}^1$ and $\mathbf{h}_{p+1} : \mathbb{R}^1 \rightarrow \mathbb{R}^m$ such that $\mathbf{h}_{p+1}(\mathbf{g}_{p+1}(\cdot)) \equiv \mathbf{n}_{p+1}(\cdot)$
   (d) $\mathbf{X}^*_{(p,\cdot)} \leftarrow \mathbf{g}_{p+1}(\mathbf{X}_p)$ (i.e., sets row p of $\mathbf{X}^*$).

This algorithm generates a reduced-dimension representation $\mathbf{X}^*$ of $\mathbf{X}$, but does not automatically generate the mapping and de-mapping functions $\mathbf{g}$ and $\mathbf{h}$. These may be formed in two ways. The first is to cascade the networks $\mathbf{g}_p$, which were trained in the SNLPCA algorithm, to form a large single network for computing $\mathbf{g}$ in a manner corresponding to the computation shown in Figure 2.11. In a similar fashion, networks $\mathbf{h}_p$ can be cascaded to form a network for computing $\mathbf{h}$. This method requires no additional training and will perform exactly the mappings calculated by the SNLPCA algorithm. The second method is to train separate neural networks to perform the mapping from $\mathbf{X}$ to $\mathbf{X}^*$ and $\mathbf{X}^*$ to $\mathbf{X}$. This method results in smaller, more computationally efficient networks, but may not necessarily converge to acceptable approximations of the desired mappings.

Figure 2.12 shows the results of SNLPCA on the data set from Section 2.3. For this data set, we see that its performance is roughly equivalent to NLPCA

for $p = 1$, and worse than both NLPCA and PCA when $p > 1$ (the comparison is summarized in Table 2.1). This is due to the fact that the first iteration has eliminated almost all of the underlying structure of the grasping skill, thus leaving the residual vectors $\mathbf{X}_1$ dominated by stochastic variations in the individual human performances. Since the nonlinear factors are trained sequentially, it is difficult for the SNLPCA algorithm not to over-fit the remaining noisy residuals at each iteration, rather than learn to represent this noise by building some equivalent of an orthonormal basis of linear features in the manner of PCA. In addition, as the number of features used increases, the compression networks have increasing difficulty learning the generated compression mappings.

An additional explanation for the degradation in performance after the first iteration of the algorithm compared to NLPCA is that the residuals in $\mathbf{X}_p$ (where $p \geq 1$) are correlated with the features generated from previous iterations. This makes some intuitive sense: it seems likely that the particular ways that the sampled grasp configurations vary from the model depends largely on to which part of the grasping motion they correspond. Hand configurations at the beginning of the grasp differ from one another in different ways than configurations near the end of the grasping motion. If we consider the feature-score from the first iteration to correspond roughly with a temporal ordering of some prototypical grasp, then the variations modeled by the second iteration of the algorithm are highly dependent upon the feature-score learned by the first iteration. Since the information represented by the first feature score is not directly available to the learning process in later iterations, the modeling ability of these later iterations is handicapped. For this reason, I suspect that the results from SNLPCA may be improved by adding adding the output value of the preceding iterations to the dimensions of the residual data used for training in later iterations. This hypothesis is not investigated in this thesis, however, as we will turn our attention to local methods and away from those based upon neural-networks.

2.7 Comparison

Table 2.1 compares the results from PCA, NLPCA, and SNLPCA. Based on its ability to represent the nominal trajectory of the grasping skill in configuration space, we conclude that the one-dimensional NLPCA mapping (equivalent to the one-dimensional SNLPCA mapping) is the best representation

p	PCA	NLPCA		SNLPCA		
	%-Err	%-Err	M	%-Err	M_c	M_d
1	72.0	58.5	12	59.9	14	3
2	41.4	40.9	3	47.4	14	6
3	33.4	30.1	8	43.1	12	5
4	26.8	25.7	5	42.0	15	9
5	23.1	20.2	10	36.8	19	8

Table 2.1: Experimental results. Percent of test data set $\mathbf{Y}$ unexplained by p factors for each method, calculated by (2.7) for PCA and by (2.17) for NLPCA and SNLPCA. M is the size of the mapping layers in NLPCA, and M_c and M_d are the sizes of the mapping layers in the compression and decompression networks trained by SNLPCA.

of the grasping skill of the models we generated. For faithful reconstruction of a given hand configuration from a reduced-dimension representation, PCA analysis is the simplest and most effective method. Higher-dimensional NLPCA and SNLPCA models might potentially be more appropriate for more complex skills.

Although NLPCA and SNLPCA both have the advantage that they can generate nonlinear models, these models have a black-box nature. It is difficult to understand exactly what they do and how, and this difficulty increases dramatically with the dimensionality of the input space and feature space. When NLPCA is used to model grasping motion using two feature components, and then we control the two activation values of the bottleneck layer using two slider-bars like the one shown in Figure 2.10, it is difficult to get a sense for what the two values do independently of one another. Moreover, given the functional form of the decompression network and how the entire bottleneck network is trained, there is little likelihood of any clear independent relationship.

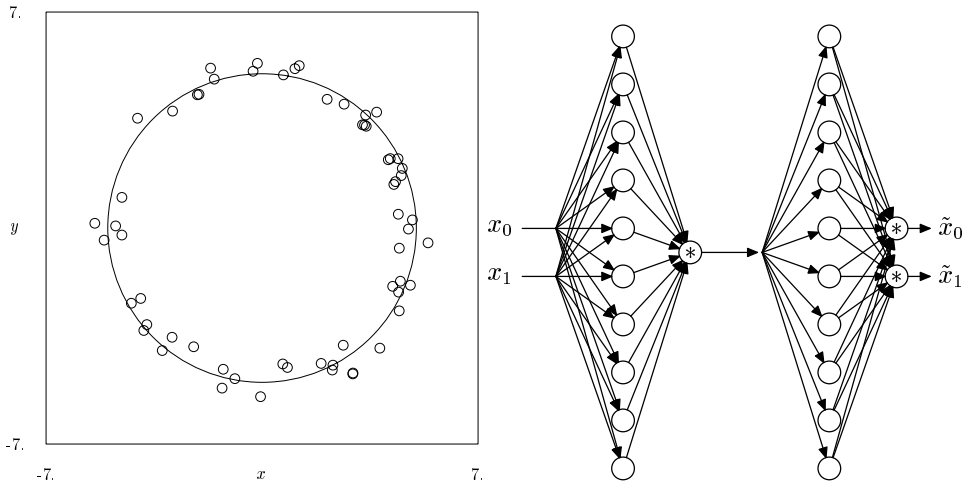
2.8 Characterizing NLPCA mappings

A paper by Malthouse [42] discusses Kramer's NLPCA method and indicates that it has several important limitations, including an inability to model curves and surfaces that intersect themselves, and an inability to parame-

terize curves with discontinuities. These limitations are due to the fact that the mapping and de-mapping projections (2.12) and (2.13) are continuous functions. For our example data set, and for many typical human skills, continuous mappings to and from the feature representation are not particularly restrictive because the skills can be smoothly parameterized. Global models in general will have a difficult time learning about data-sets which are highly discontinuous. Later in this thesis, we will show how some curves which intersect themselves in position space can be successfully modeled by other methods in phase space.

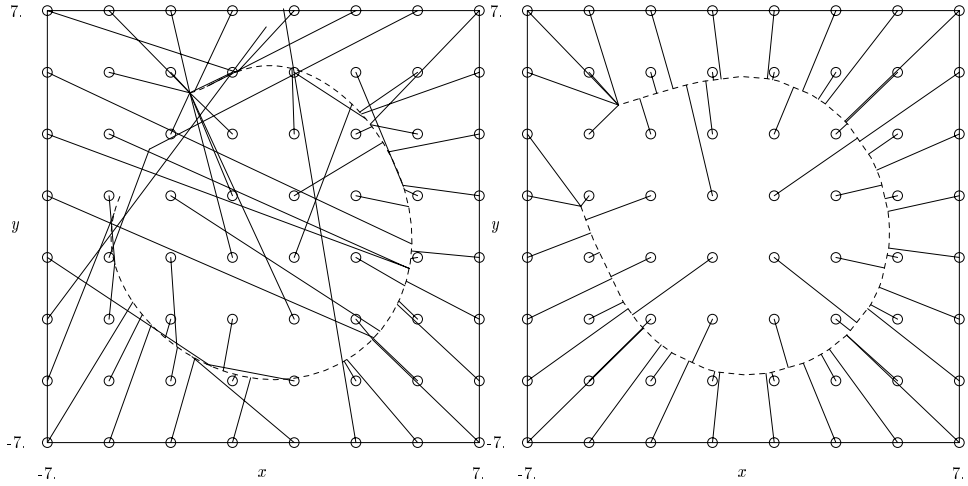
A criticism in Malthouse's paper which is more relevant to the goals of this thesis is that Kramer's NLPCA method tends to result in suboptimal projections, and that methods based on principal curves [17, 28, 37] tend to result in better parameterizations. This problem is demonstrated in Figure 2.13. In 2.13(a), we first randomly sample a set of points from a circular distribution, and then add uniform random noise to the x and y dimensions of the sample points. A NLPCA network, shown in 2.13(b) is used to learn to map these points to themselves through a single bottleneck node, using 10 neurons in each mapping layer. Figure 2.14(a) shows how the training points map to the learned model, and Figure 2.13(c) shows how this network projects points in the immediate vicinity of the training points onto the learned model. The circles are the input points, and the line from the center of each circle shows to where the network maps that input. The output of the decompression part of the network $\mathbf{h}_0$ over the full range of activations on the bottleneck neuron, from the minimum value generated by the training data to the maximum value, is drawn as a dashed curve. We see that although some of the mappings are plausible, particularly those in the lower-right hand side of the figure (e.g., around the $(5, -5)$ coordinate), the projections of other points are far less sensible. The closer we look toward the upper-left corner of the figure, the more distorted the mappings appear. Instead of mapping to the closest point on the dashed model-line, many inputs project to points on the far side of the model. The input at $(3, -7)$, which is fairly close to one part of the model curve and near to inputs points whose projections are very plausible, even maps past the far side of the model curve to the top of the plot.

These strange projection results should not be surprising given the nature of back-propagation neural networks. The power of these functional forms is that they are sufficiently nonlinear and have enough degrees of freedom that they can fit complex mappings with ease, even when trained using straight-



(a) Training data: Points sampled from circle with added noise.

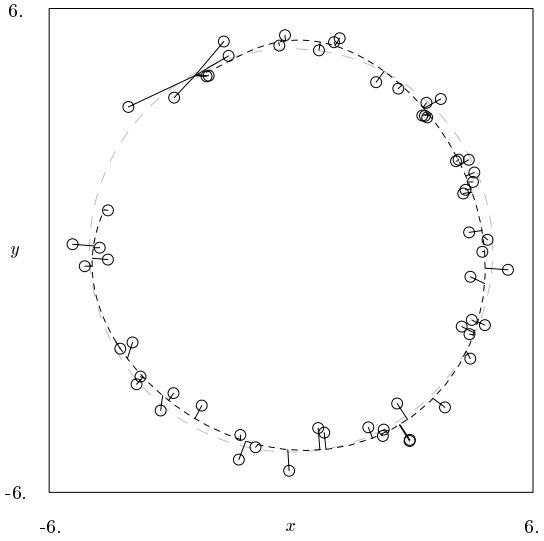
(b) NLPCA network used for learning circle mapping.



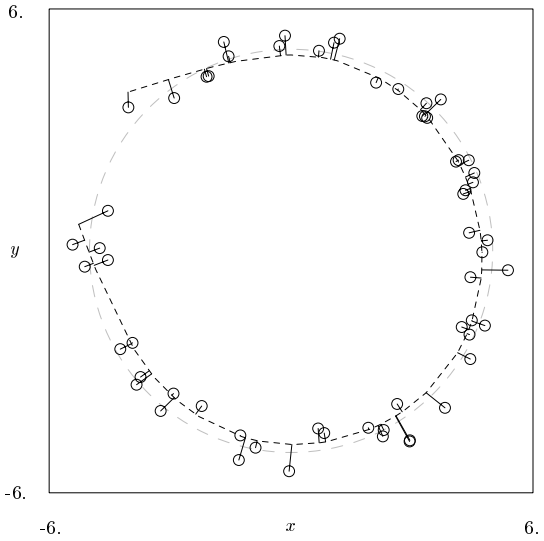
(c) Mapping generated by NLPCA.

(d) Mapping generated by principal curve.

Figure 2.13: Mappings to circular figure of surrounding space, generated by NLPCA and principal curves.



(a) NLPCA



(b) Principal curve

Figure 2.14: Mapping of training points to models learned by neural networks and principal curves.

forward hill-climbing algorithms. Starting with a random mapping in this case, the network adjusts its projection of the output points until they nearly match the input points. Along the way, the learning function can easily warp the projection for any region as necessary to reduce the error measure. This ease of warping the output space, and the fact that what happens outside the training set has no effect on training error, gives us reason to believe that the resulting projection for points outside the training set may look somewhat bizarre.

There are several reasons why we are looking at plots of projections in a two-dimensional space. The first of course is that these plots are much easier to understand than higher-dimensional projections, and it is easier to make the connection between fitting the training points and modeling trajectories which might pass through them. Since building a good one-dimensional model for trajectory fitting is one of the more important uses of dimension reduction, we want our methods to perform well in this case. The complexity of the neural network mapping increases with its dimensionality, and we would expect the mappings in higher dimensions to become more rather than less strange as we look at less simple cases, so these results for two dimensions should give use pause. We are thus motivated to look for modeling techniques where the resulting projections are more understandable, and which are explicitly based on principals appropriate for fitting trajectory information.

What kind of projection function would be better than that shown in Figure 2.13(c)? The least-squares projection method of the PCA model is an appealing answer. If we associate a metric function with the raw-data space, and consider the model to be some lower-dimensional manifold in this space, then the projection of each point in the raw-data space onto the model is the nearest point in the model. Figure 2.13(d) shows a model built from the training points of 2.13(a) where the definition of projection is based on a least-squares function in a similar manner to PCA, but using a nonlinear model. The method used to generate this model is the principal curves algorithm of Hastie and Stuetzle [28]. Figure 2.14(b) shows how the training points map to the principal curve learned from them. Unlike the methods introduced in this chapter, the principal curves algorithm examines the relationship between points which are close in proximity, and thus it is not a global method.

It should be noted here, that at least one global parametric model can generate a mapping which looks more like that of the principal curve model of

Figure 2.13(d). The input-training neural network of Tan and Mavrovouniotis [75] generates a much better mapping than does Kramers's NLPCA method. For modeling human performance data, their method should thus outperform NLPCA. Their method still has the problem that it is based on a neural network mapping, however, and is thus more difficult to analyze than the methods based on local models which we present later in this thesis.

In the following chapters we will demonstrate how methods such as principal curves, which make use of local information, can be used to build models of human performance data. We will also exploit this local information to address another problem with the global methods: the fact that while we really would like to build parameterizations which can express typical performance trajectories, global methods can only fit individual performance data points, and thus information about how typical human performances progress from one sampled point to another is completely lost from the training data. Figure 2.13 demonstrates that while neural-network based NLPCA does a good job of fitting the training data, we should be hesitant to use the resulting projections of points between the training points, and thus we should be wary of using trajectories in feature-space to model realistic trajectories and motions in the raw-configuration space. This problem can be addressed by methods for fitting trajectories which use local information in the human performance data set.

Chapter 3

Local methods for dimension reduction

3.1 Introduction

This chapter introduces the use of local, non-parametric methods for dimension reduction of human performance data. The previous chapter demonstrated the use several global methods for this purpose, methods which look at all the data points from all the training examples as a single set of vectors $\mathbf{X}_{(m\times n)} = [\mathbf{x}_0|\mathbf{x}_1|\ldots\mathbf{x}_{n-1}]$ and simply try to map them to a lower-dimensional space and back again so as to preserve maximum information. Global parametric models have a number of difficulties when it comes to modeling complex data sets such as those from human performances, however. When the models are simple enough for easy analysis, they can be too simple to adequately model the data. PCA is analytically beautiful, but the first principal component adequately describes configurations along a best-fit performance trajectory only when that trajectory is linear in configuration space. On the other hand, models like NLPCA and SNLPCA, which are flexible enough to fit a wide variety of data sets, can be very difficult to analyze and use. Methods for dimension reduction such as these, which are based upon neural networks, also tend to result in inadequate projections to and from the feature space.

Focusing on non-parametric methods will help us to construct suitable projections to and from feature space via nonlinear models, and to generate trajectories typical of human performance from multiple examples. These

methods are data-driven rather than based *a priori* upon a global parametric form, and generally fit data locally rather then globally. The output value at a given domain point is typically found by a simple average or weighted regression of the values of nearby sample points. Because these local models are simple, it easy to understand their outputs. The entire model is still flexible enough to adequately represent an almost arbitrary set of functions or mappings, however, since it is constructed from a large number of these local models.

In this chapter, we will focus on trajectory-fitting. As we discuss in the next section, a best-fit trajectory is the most basic kind of reduced-dimension action model. Two general-purpose non-parametric methods will be reviewed for this purpose: scatter plot smoothing and principal curves. We will discuss how these work, and how their use of local information helps build good trajectory models from human performance data. The next two chapters will present adaptations of these methods I have developed for the specific purpose of modeling human performance data.

3.2 Local, non-parametric methods for trajectory fitting

Non-parametric methods can allow us to explicitly design the structure of the local model for the specific purpose of modeling human performance. The simplest logical model for human action data is the "best-fit" or "most-likely" performance trajectory.

In Section 1.2, we formulated dimension reduction for a training set of action data as a feature-extraction problem. Given a set of performance data $\mathbf{X}$ in space S, we would like to convert these data to a lower-dimensional representation $\mathbf{X}^*$ in space S^* where $\mathbf{X}^* = \mathbf{g}(\mathbf{X})$. These lower-dimensional data map to a corresponding representation in the original space $\tilde{\mathbf{X}} = \mathbf{h}(\mathbf{X}^*) = \mathbf{h}(\mathbf{g}(\mathbf{X}))$. One important feature of such a mapping is that smooth paths in S should map to smooth paths in S^*, and smooth paths in S^* should map to smooth paths in S. This allows directional-derivatives in S^* to be meaningful, and it simplifies the problem of representing a trajectory in S^* which maps a plausible trajectory in S. This lets us create an animation of action-performance in a simulation environment by moving a few slider-bars representing feature-values in S^*, and it simplifies writing successful

controllers that use input or output vectors from space S^*.

We have seen in Chapter 2 that it may be plausible in some cases to model an extrinsically high-dimensional data-set using just a single parameter. If we use a scalar parameter $s \in S^*$ to model a dataset from multiple examples of some action, then we want the image $\mathbf{h}(s)$ to contain representative points for all parts of a "most-likely" or "best-fit" version of the action. If a smooth non-intersecting best-fit trajectory path in S maps to a smooth path in S^* (an interval $\subset \mathbb{R}^1$), then s is a parameterization variable which can be transformed via a monotonic function to a time-parameterization of the best-fit trajectory. Parameter s represents a *temporal-ordering* of points along this best-fit trajectory. If a one-dimensional feature parameter s does not specify a temporal ordering of points along a continuous trajectory in S, then it will be much more difficult to use it to represent motion, and derivatives with respect to that parameter will not always have a meaningful interpretation. In a simulation interface such as the one shown in Figure 2.10 on 31, moving a slider bar to vary s from one value to another at the appropriate speed should create a plausible animation representing some portion of the action, and moving the slider from an s-value which is typical of a starting configuration to an s-value which is typical of an ending configuration should create a plausible animation of the entire action.

A best-fit trajectory is very useful model of a human action. Examples include the motion for a typical power grasp, Mark McGwire's typical home run swing, a typical walking or swimming motion, or a typical motion of the lips when saying a given word. Such a trajectory can be used for creating an animation for a video game or movie, for comparing a novice's actions to an expert's (e.g., "how does my golf-swing compare to Tiger Woods' "), or for gesture recognition.

We saw in the last chapter that while it is possible for global methods such as NLPCA to generate something akin to a best-fit trajectory, there is no explicit constraint requiring them to generate a satisfactory model. Global methods fit individual training points, but may not necessarily interpolate between them in a manner plausible for an actual performance. Non-parametric methods, on the other hand, can make explicit tradeoffs on the local level such as balancing quality of fit verses the smoothness of the model, as depicted in Figure 3.1. Spline smoothers, discussed in the next section, are based on precisely this trade-off. Moreover, the projections of points onto the local models can be done in a principled manner by projecting to the nearest model point, as demonstrated by the principal curves in

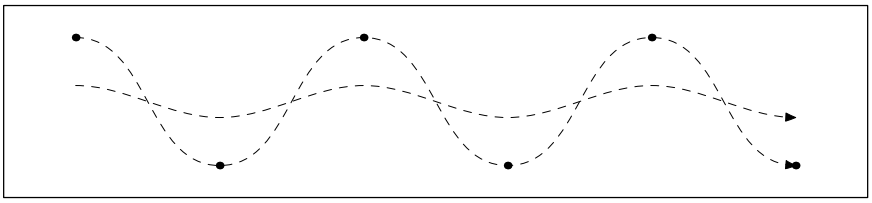


Figure 3.1: Explicit trade-off between fitting error and smoothness of the model.

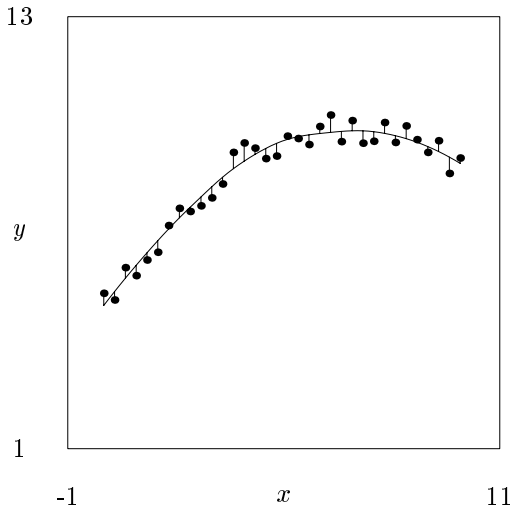


Figure 3.2: Modeling a data-set with a scatter plot smoother.

Figure 2.13(d) on page 37 and Figure 3.5 on page 53.

3.3 Scatter plot smoothing

Scatter plot smoothers allow us to balance local smoothness of the estimated model against modeling error, or to smoothly combine localized fitting into a global model, without assuming a parametric form for the model. Figure 3.2 shows the result of a smoother based on a robust locally-weighted regression [11, 12]. Note that the error vectors are vertical. As in linear regression and polynomial fitting (Section 2.2.1), scatter plot smoothers minimize error

in the response variable only.

There are several kinds of scatter plot smoothers. Kernel smoothers fit each point in the dataset using a locally weighted average, while locally-weighted regression fits local models to the data [4].

Spline smoothers [16, 61, 66, 81] generate a model by minimizing a cost function weighing modeling error against the integral of the d-th derivative of the model curve. The cost function is

$$S = p \sum_{i=0}^{n-1} \left(\frac{y_i - f(x_i)}{\delta y_i} \right)^2 + (1-p) \int_{x_0}^{x_{n-1}} (f^{(d)}(x))^2 dx, \tag{3.1}$$

where p is the smoothing parameter weighing approximation error against smoothness of the curve, and the δy_i weigh individual data points. Weights δy_i are often local estimates of standard deviation. Minimizing S for a given set of values p and δy_i results in a model curve f which is a polynomial spline of order $k \equiv 2d$ with simple knots at $x_0, \ldots, x_{n-1}$ ($x_i < x_{i+1}$), and natural end conditions:

$$f^{(j)}(x_0) = f^{(j)}(x_{n-1}) \qquad \text{for } j \in \{d-1, \ldots, k-3\}. \tag{3.2}$$

Optimization is often performed using an acceleration penalty ($d = 2$), which results in a cubic spline solution with the form

$$P_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 \tag{3.3}$$

between each knot.

For a given value of the smoothing parameter p, we can solve for parameters $\mathbf{c} = \{c_i\}$ using the tridiagonal system [16]

$$(6(1-p)\mathbf{Q}^T\mathbf{D}^2\mathbf{Q} + p\mathbf{R})\mathbf{c} = 3p\mathbf{Q}^T\mathbf{y}, \tag{3.4}$$

where $\mathbf{D}$ is the diagonal matrix of weights $\lceil \delta y_0, \ldots, \delta y_{n-1} \rfloor$, $\mathbf{R}$ by (4.19) on page 72, and $\mathbf{Q}^T$ by (4.20). Band matrices $\mathbf{R}$ and $\mathbf{Q}^T$ are described in Chapter 4 as I derive a spline smoother which also penalizes error in velocity information. Once $\mathbf{c}$ has been computed in this way, we can easily determine the other terms of the smoothing spine parameterization:

$$\begin{aligned} \mathbf{a} &= \mathbf{y} - 2\left(\frac{1-p}{p}\right)\mathbf{D}^2\mathbf{Q}\mathbf{c} \\ d_i &= \frac{c_{i+1} - c_i}{3\Delta x_i} \\ b_i &= \frac{a_{i+1} - a_i}{\Delta x_i} - c_i \Delta x_i - d_i(\Delta x_i)^2, \end{aligned} \tag{3.5}$$

where $\Delta x_i = (x_{i+1} - x_i)$.

From (3.1), it might at first appear that a spline smoother is a global parametric model rather than a local model. The model is indeed defined as the minimum of a global error function, and the resulting curve may be parameterized as a spline function. However, the number of parameters defining the spline is actually greater than the number of values in the training data, and the error term of the cost function is balanced against a smoothness measure based upon derivatives, which are by definition local. In fact, Silverman [68] shows that a spline smoother is equivalent to a kernel smoother with a variable-sized kernel.

When using spline smoothers, it is important to choose a suitable value for the smoothing parameter p. Cross validation [73] is a good method to use for selecting this value for a given data set. While leave-one-out cross-validation for a data set using the simple spline solution presented in this section is computationally expensive, Craven and Wahba [13] present a closed-form method for computing the optimal solution for a given data set, and Hutchinson and de Hoog [31] show how to compute this closed-form solution in linear time.

3.4 Action recognition using smoothing splines

The error term of spline-smoother cost function (3.1) is based upon the assumption of a Gaussian error distribution at each parameterization x_i. We can use this assumption to formulate a probability that a given performance belongs to the class of actions corresponding to a given model, and to compare this probability across several models for the purpose of recognition.

Smoothing each dimension of a data set $\mathbf{Y}$ against parameterization vector $\mathbf{x}$ results in a model $(\mathbf{x}, \mathbf{f}, \{\mathbf{D}_i\})$, where $\mathbf{f} : \mathbb{R}^1 \rightarrow \mathbb{R}^m$ is the multi-dimensional spline model of the best-fit trajectory, and the diagonal elements of each matrix $\mathbf{D}_i$ are the estimated standard-deviation for each dimension of the training data at parameterization x_i. Given this model, and the assumption of a Gaussian distribution at each parameterization value, we can estimate the probability of a given example point $(x_\mathrm{e}, \mathbf{y}_\mathrm{e})$ given the model

as [7]

$$p(\,(x_{\mathrm{e}}, \mathbf{y}_{\mathrm{e}}) \mid (\mathbf{f}, \{\mathbf{D}_i\})\,) = \frac{1}{(2\pi)^{m/2}|\hat{\mathbf{D}}^2|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{y}_{\mathrm{e}} - \mathbf{f}(x_{\mathrm{e}}))^T \hat{\mathbf{D}}^2 (\mathbf{y}_{\mathrm{e}} - \mathbf{f}(x_{\mathrm{e}}))\right). \quad (3.6)$$

For this evaluation, $\mathbf{f}(x_{\mathrm{e}})$ may either be computed exactly from the spline equation, or approximated by linear interpolation between the nearest points $\mathbf{y}_k, \mathbf{y}_{k+1}$ on the model such that $x_k < x_{\mathrm{e}} < x_{k+1}$, and $\hat{\mathbf{D}}$ can be estimated by linear interpolation between $\mathbf{D}_k, \mathbf{D}_{k+1}$.

While the cost function for the spline smoother relates the points in its model by the roughness penalty term, this term balances model smoothness against an error cost which treats each sample as independent of the others. When we look at data from an example performance $\mathbf{Y}_{\mathrm{e}}$, where n_{e} samples of the performance have been taken over time, we thus treat each point as independent, uncorrelated with the other samples. This gives us

$$p(\,(\mathbf{x}_{\mathrm{e}}, \mathbf{Y}_{\mathrm{e}})|\,(\mathbf{f}, \{\mathbf{D}_i\})\,) = \prod_{i=0}^{n_{\mathrm{e}}-1} p(\,(x_{\mathrm{e}i}, \mathbf{y}_{\mathrm{e}i}) \mid (\mathbf{f}, \hat{\mathbf{D}}(x_{\mathrm{e}i}))\,)$$
$$= \prod_i \frac{1}{(2\pi)^{m/2}|(\hat{\mathbf{D}}(x_{\mathrm{e}i}))^2|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{y}_{\mathrm{e}i} - \mathbf{f}(x_{\mathrm{e}i}))^T (\hat{\mathbf{D}}(x_{\mathrm{e}i}))^2 (\mathbf{y}_i - \mathbf{f}(x_{\mathrm{e}i}))\right). \quad (3.7)$$

For an example with many sample points, this probability will be very small, so it is useful to consider its logarithm instead:

$$\log p(\,(\mathbf{x}_{\mathrm{e}}, \mathbf{Y}_{\mathrm{e}}) \mid (\mathbf{f}, \{\mathbf{D}_i\})\,) = \sum_i \log\left(\frac{1}{(2\pi)^{m/2}|(\hat{\mathbf{D}}(x_{\mathrm{e}i}))^2|^{1/2}}\right)$$
$$+ \sum_i \left(-\frac{1}{2}(\mathbf{y}_{\mathrm{e}i} - \mathbf{f}(x_{\mathrm{e}i}))^T (\hat{\mathbf{D}}(x_{\mathrm{e}i}))^2 (\mathbf{y}_{\mathrm{e}i} - \mathbf{f}(x_{\mathrm{e}i}))\right), \quad (3.8)$$

which simplifies to

$$\log p(\,(\mathbf{x}_{\mathrm{e}}, \mathbf{Y}_{\mathrm{e}}) \mid (\mathbf{f}, \{\mathbf{D}_i\})\,)$$
$$= -\frac{n_{\mathrm{e}} m}{2}\log(2\pi) - \sum_{i=0}^{n_{\mathrm{e}}-1}\sum_{j=0}^{m-1} \log(\delta\hat{y}_j(x_{\mathrm{e}i})) - \frac{1}{2}\sum_{i=0}^{n_{\mathrm{e}}-1}\sum_{j=0}^{m-1}\left(\frac{y_{ji} - f_j(x_{\mathrm{e}i})}{\delta\hat{y}_j(x_{\mathrm{e}i})}\right)^2. \quad (3.9)$$

Note that right-most terms of (3.8) and (3.9) are expressions of the Mahalanobis distance measure [7].

Bayes' theorem can then be used to express the probability that the model $(\mathbf{f}, \{\mathbf{D}_i\})$ was the cause of the data set:

$$\log p(\,(\mathbf{f}, \{\mathbf{D}_i\}) \mid (\mathbf{x}_e, \mathbf{Y}_e)\,) = \log p(\,(\mathbf{x}_e, \mathbf{Y}_e) \mid (\mathbf{f}, \{\mathbf{D}_i\})\,) + \log p(\,(\mathbf{f}, \{\mathbf{D}_i\})\,) - \log p(\,(\mathbf{x}_e, \mathbf{Y}_e)\,). \quad (3.10)$$

The probability of a real data set is 1 (because it actually happened), so we may drop the last term of (3.10).

Since the absolute magnitude of these probabilities is so low, we typically compare the ratio of probabilities between pairs of models, which corresponds to the difference between their log probabilities. If we assume that the *a priori* probabilities for all the candidate models are equal, then for recognition purposes, we need only compare the first term of (3.10), which is the probability of the data given each model.

3.5 A gesture-recognition experiment using spline smoothing

Because human performance data sets do not usually contain an appropriate explanation variable against which to smooth, the main role of spline smoothers in this thesis will be as the preferred smoothers for trajectory fitting using the principal curves algorithm. In some cases, however, we are able to find a suitable explanation variable. If the speed of motion is very consistent between example performances, for example, we may be able to smooth against time.

Data from finger motions in letter-signing is one potential application for building models by smoothing against time. The data set used for testing sign-language recognition using hidden Markov models in Chapter 7 has some characteristics which make it appropriate enough for modeling with smoothing splines. This data set was recorded while a user signed 14 different letters: A, B, C, D, E, F, G, I, K, L, M, U, W, and Y. The user signed each letter thirty times, but in a randomized order as prompted by a computer program. The program also ensured that the user was not prompted to sign the same letter twice in a row. The signer wore a Virtual Technologies

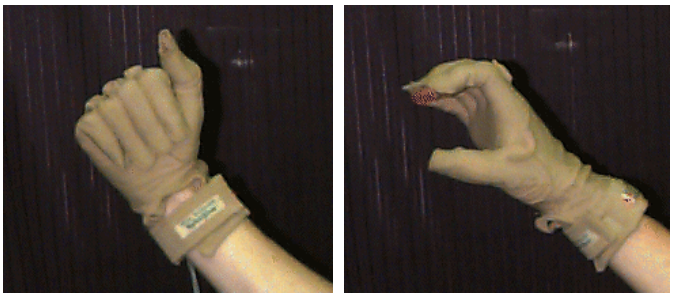


Figure 3.3: Final hand positions for letter-signs 'A' and 'C.'

'Cyberglove' which collected 18 channels of information about the configuration of the joints in the fingers, and the letters chosen for signing were a set of motions which could be unambiguously recognized using only finger motions (i.e., without data about the position, orientation, or motion of the hand). Figure 3.3 shows the final hand positions for the letter signs A and C. Because the example performances for each letter start from the final configuration of the random previous letter, the endings of these examples are much more consistent than their beginnings. As the signs are so varied near the beginning of each example, precise alignment of the examples in time is not important until the end of the gesture. Scaling the time values $\mathbf{t}$ for each gesture to fall between 0 and 1,

$$\mathbf{t}^* = \mathbf{t} / \max t_i, \tag{3.11}$$

will cause points near the end of the gestures to appear better aligned in time, and thus to correspond fairly well there.

We built a model for each letter by smoothing against t^*. For each letter, we combined the data from a set of training examples into a single matrix whose column-vectors are sorted by their corresponding values t_i^*, and smoothed each row j of this matrix against parameterization variable t^*.

The smoothing parameter p was chosen by experimentation, and the numerical stability of the smoothing solution was increased using the method for combining points with similar parameterizations documented (in a form extended to include velocity information) in Section 4.5. Initially, the variance matrices $\mathbf{D}_i^2 = \lceil \delta y_{0,i}^2, \ldots, \delta y_{m-1,i}^2 \rfloor$ were set to the identity matrix ($\delta y_{ji} = 1$), and then we used Equation (4.38) on page 77 to compute equivalent values of δy_{ji} for points which need to be merged due to similar parameterizations. Equation (4.39) was used to merge the corresponding values y_{ji}.

After smoothing each dimension j of the data, the local variance for each dimension as a function of time was estimated using the method suggested by Silverman in [69] and documented (in an extended form) in Section 4.7. This variance estimate was combined via point-wise multiplication with the result from (4.38), and this new variance estimate was used to smooth the data again.

The result is a best-fit trajectory $\mathbf{f}$ for each letter-sign, and a set of variance estimates $(\delta y_{ji})^2$ for each dimension j of the motion at each rescaled time value t_i^*. We store the model for a given letter sign as $(\mathbf{t}^*, \mathbf{Y}, \{\mathbf{D}_i^2\})$, where $\mathbf{t}^*$ is the vector of rescaled time values, $\mathbf{Y}$ is composed of the column vectors $\mathbf{y}_i = \mathbf{f}(t_i^*)$, and the diagonal elements of $\mathbf{D}_i^2$ contain the corresponding variance estimates $(\delta y_{ji})^2$. Figure 3.4 plots four dimensions from the model of the sign for A. As expected, the example points from the start of each plot are widely scattered, and points near the end of the plot are clustered more tightly. This is true for all eighteen dimensions of the data. The estimated standard deviations are thus large at the beginning and small at the end for each dimension. The roughness of the standard deviation plots is due to the square window used for their estimation in (4.38).

These models can be used for classification of unknown performance data using the method presented in Section 3.4. This classification method was tested on 294 examples outside the training set: 21 examples for each of the 14 letter-signs. Table 3.1 summarizes the results from using a minimum of 3 and a maximum of 9 training examples per model. After training on 6 examples per model, we see that the classifier has better than 97% reliability. After training on 9 examples per model there are no misclassifications, and the minimum ratio of the probability of the correct model to the next most likely is nearly 30 to 1.

3.6 Principal curves

Smoothing, like regression, requires an explanation variable against which to model the response variables. The principal curve, introduced by Hastie and Stuetzle [28], is a kind of smoother which can build its own parameterization against which to smooth the points in the data set. A principal curve is thus a non-linear analogue of the first principal component discussed in Section 2.2.2. If we assume that $\mathbf{y}_i = \mathbf{f}(\lambda_i) + \boldsymbol{\epsilon}_i$ where $\mathbf{f}$ is a smooth curve, and that covarience$(\boldsymbol{\epsilon}_i) = \sigma^2\mathbf{I}$, then optimizing $\mathbf{f}$ to minimize the modeling

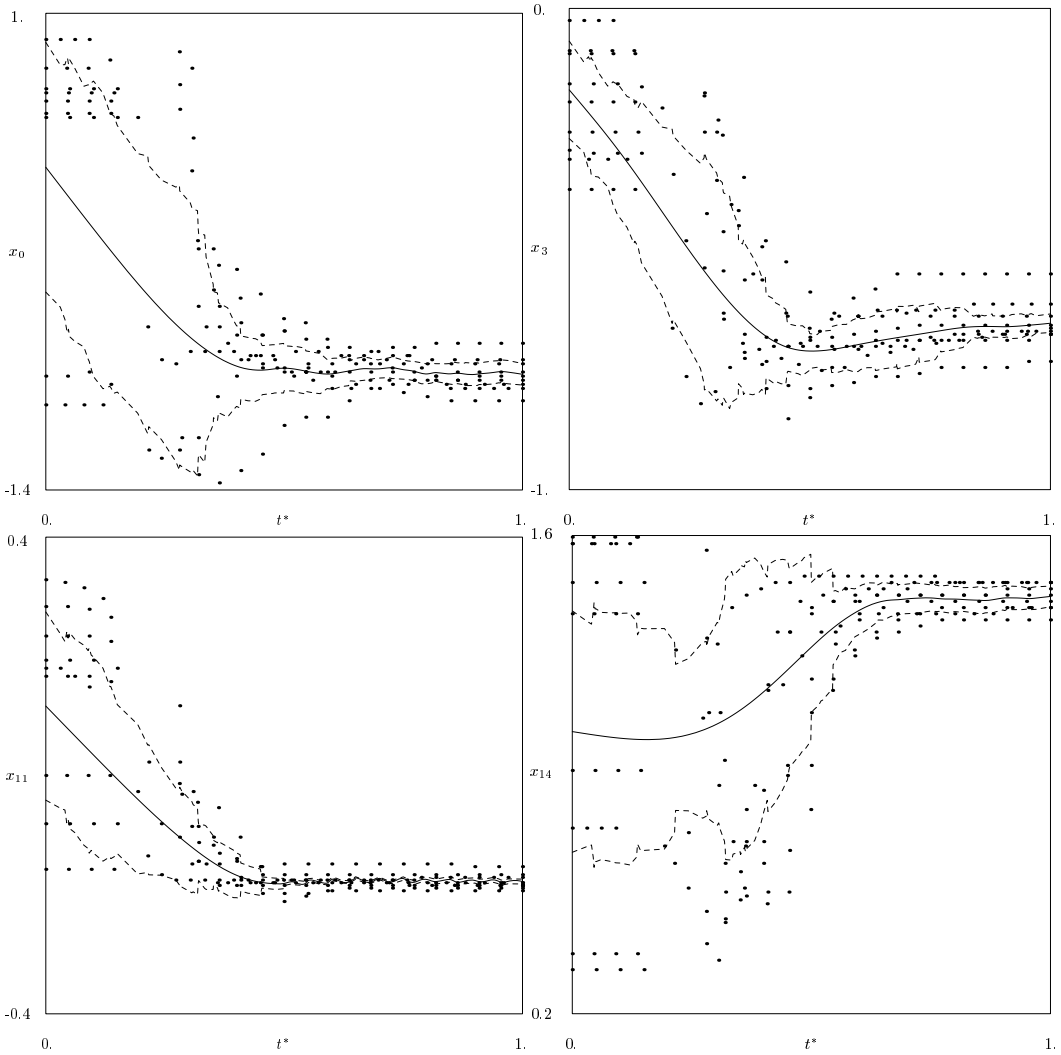


Figure 3.4: Spline smoother fit of four representative dimensions from the eighteen measured by a Cyberglove during signing of the letter A. Dots are sample points, the continuous line is the smoothed fit, and the dashed lines show the estimated region within one standard deviation at each time value. Time values of each example performance are rescaled to fall between 0 and 1.

Training examples	Misclassifications number	percent	$\min(\frac{P(M_{\text{cor}})}{P(M_{\text{inc}})})$
3	46	16	-
4	69	23	-
5	24	8.2	-
6	7	2.4	-
7	8	2.7	-
8	6	2.0	-
9	0	0.0	29.9

Table 3.1: Letter sign classification results. For the models trained using 9 training examples there were no errors in 294 classifications, and the smallest ratio of the probability of the correct model $P(M_{\text{cor}})$ to the next most likely model $P(M_{\text{inc}})$ over all the classification trials is nearly 30 to 1.

error transforms it into an estimated principal curve of the data set.

Figure 3.5 shows a principal curve model of the example dataset from Figure 2.1(a). Figure 2.14(b) on page 38 also shows a principal curve, this time resulting from a noisy data-set sampled from a circular two-dimensional distribution. In these plots, we see that the error vectors are orthogonal to the model curve. This is because the principal curves algorithm runs a smoother on each dimension of the data (the *conditional expectation* step, which will be described later), and then each point is modeled by the nearest point on the resulting curve (the *projection* step).

Because every dimension of the data is smoothed, we need an additional parameterization against which to smooth them. The parameterization variable λ arises here in a manner similar to the way it does in the principal component model of Section 2.2.2. For the principal component model, λ was a discovered explanation variable against which variables in the orthogonal direction could be modeled. This turned a problem of finding a linear model with two response variables into a (trivial) regression of one response variable against one explanation variable. In a similar manner, the projection step of the principal curves algorithm gives us a parameterization variable λ against which to use a scatter plot smoother for each dimension of the data.

Because the principal curves algorithm can use a smoother which balances local quality of fit against the smoothness of the model curve, it is intuitively

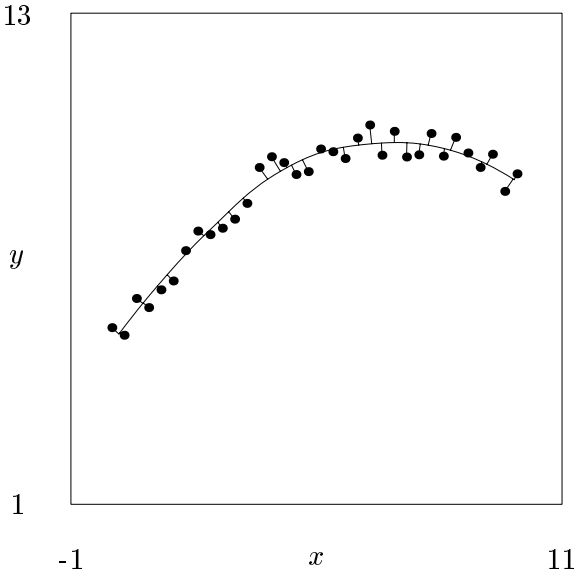


Figure 3.5: Principal curve

appealing as a method for smoothing trajectories. Figures 3.5 and 2.14(b) each suggest a plausible trajectory which fits a set of configuration points. In fact, Hastie and Stuetzle give an example of where principal curves were used to smooth the path of particle beams through the Stanford Linear Collider at the Stanford Linear Accelerator Center. Chapter 5 will demonstrate how local velocity data from example performances can be used within the principal curves algorithm to find best-fit or most-likely trajectories in phase space from human performance data.

3.6.1 Definition of principal curves

Principal curves are those smooth curves that are *self consistent* for a distribution or dataset. This means that if we pick any point on the curve, collect all the data in the distribution that project onto that point, and average them, this average coincides with the point on the curve.

To be more precise, the principal curve is defined as follows [28]. Let X be a random vector in $\mathbb{R}^m$ with density h and finite second moments. Assume, without loss of generality, that $E(X) = 0$. Let $\mathbf{f}$ be a smooth C^∞ unit speed curve in $\mathbb{R}^m$, parameterized over $\Lambda \subseteq \mathbb{R}^1$, a closed (possibly infinite) interval,

that does not intersect itself, and has finite length inside any ball in $\mathbb{R}^m$.

The *Projection index* $\lambda_{\mathbf{f}}$: $\mathbb{R}^m \to \mathbb{R}^1$ is

$$\lambda_{\mathbf{f}}(\mathbf{x}) = \sup_\lambda \left[\lambda : \|\mathbf{x} - \mathbf{f}(\lambda)\| = \inf_\mu \|\mathbf{x} - \mathbf{f}(\mu)\|\right]. \tag{3.12}$$

The curve $\mathbf{f}$ is called *self-consistent* or a *principal curve* of h if

$$f(\lambda) = E(X \mid \lambda_{\mathbf{f}}(X) = \lambda) \tag{3.13}$$

for a.e. λ.

In general, we do not know for what kinds of distributions principal curves exist, nor the properties of the curves for these distributions. For some cases, however, we can give answers to these questions. For ellipsoidal distributions, the first principal component is a principal curve. For spherically-symmetric distributions, any straight line through the center of the distribution is a principal curve.

3.6.2 Distance property

Principal curves are critical points of the distance from observations. Let $\mathfrak{G}$ be a class of curves parameterized over Λ. For $\mathbf{g} \in \mathfrak{G}$ define $\mathbf{f}_t \equiv \mathbf{f} + t\mathbf{g}$. Then, curve $\mathbf{f}$ is called a critical point of the distance function D for variations in the class $\mathfrak{G}$ iff:

$$\left.\frac{dD^2(h, \mathbf{f}_t)}{dt}\right|_{t=0} = 0 \quad \forall g \in \mathfrak{G}. \tag{3.14}$$

A nice property of the principal curve is that this distance property is similar to the minimization of the cost function in a spline smoother.

3.6.3 Principal curves algorithm for distributions

Principal curves are actually defined over continuous distributions rather than discrete data sets. Thus before we discuss how to approximate the principal curve from a given set of data, we present the algorithm for distributions. We can roughly state the algorithm for finding a principal curve of a given distribution as:

1. Starting with any smooth curve (usually the largest principal component), check whether the curve is self-consistent by projecting and averaging.

2. If it is not, repeat the procedure using the new curve obtained by averaging as a starting guess.

3. Iterate until the estimate (hopefully) converges.

More precisely, the algorithm is

1. Initialization:
   Set $\mathbf{f}^{(0)}(\lambda) = \bar{\mathbf{x}} + \mathbf{a}\lambda$ where $\mathbf{a}$ is the first linear principal component of h. Set $\lambda^{(0)}(\mathbf{x}) = \lambda_{\mathbf{f}^{(0)}}(\mathbf{x})$.

2. Repeat, over iteration counter j:

   (a) Conditional expectation:
   Set $\mathbf{f}^{(j)}(\cdot) = E(X \mid \lambda_{\mathbf{f}^{(j-1)}}(X) = \cdot)$.

   (b) Projection:
   Define $\lambda^{(j)}(\mathbf{x}) = \lambda_{\mathbf{f}^{(j)}}(\mathbf{x})\ \forall \mathbf{x} \in h$; transform $\lambda^{(j)}$ so that $\mathbf{f}^{(j)}$ is unit speed.

   (c) Error evaluation:
   Calculate $D^2(h, \mathbf{f}^{(j)}(\mathbf{x})) = E_{\lambda^{(j)}} E[\|X - \mathbf{f}(\lambda^{(j)}(X))\|^2 \mid \lambda^{(j)}(X)]$

   Until the change in $D^2(h, \mathbf{f}^{(j)}(\mathbf{x}))$ is below some threshold.

Unfortunately, there is no guarantee that the curves produced by the conditional expectation step are differentiable (especially at the ends). Therefore, convergence cannot be proved.

However, the algorithm converges well in practice. Some justifications for why it generally works include that by definition, principal curves are fixed-points of the algorithm. Also, assuming that each iteration is well defined and differentiable, the distance does converge. Finally, if we use straight lines for conditional expectation, the algorithm converges to the first principal component.

3.6.4 Principal curves algorithm for data sets: projection step

Although principal curves are defined in terms of distributions, in practice we are generally concerned with finding principal curves of datasets containing a finite number of points sampled from a distribution. The algorithm for

finding a principal curve through a dataset is roughly analogous to that for finding the curve through a distribution, with a projection step and a conditional expectation step. The conditional expectation step, however, necessarily involves smoothing to estimate the effects of the distribution from which the data was sampled.

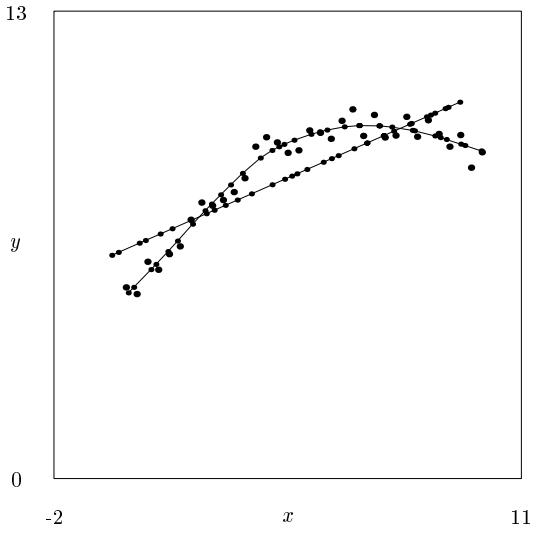
Figure 3.6 shows the result of the first two iterations of the principal curves algorithm on the example data set from Figure 2.1(a). We see in Figure 3.6(b) that the effect of the second iteration is small, indicating that it is converging to a self-consistent curve. The initial estimate, the straight-line in Figure 3.6(a), is the first principal component.

In the projection step, we parameterize each data-point in terms of a distance from some starting point along the estimated principal curve from the last step. The assumption here is that the probability that a given data-point corresponds to a given point on a model-trajectory is a monotonically decreasing function of the distance between these points. Thus, for fixed $\mathbf{f}^{(j)}(\cdot)$ we find for each $\mathbf{x}_i$ in the sample the value

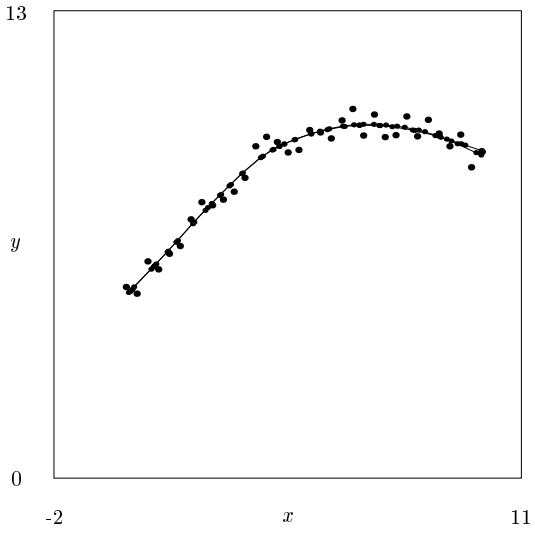
$$\lambda_i = \lambda_{\mathbf{f}^{(j)}}(\mathbf{x}_i). \tag{3.15}$$

The most obvious way to do this is to find the λ-parameterization of the nearest point $\tilde{\mathbf{x}}_i$ of the set of $(n-1)$ points nearest to $\mathbf{x}_i$ on the line segments comprising the current estimate of the principal curve. Let $\boldsymbol{\lambda}^{(j)+}$ be the vector of $\lambda_i^{(j)}$ values sorted into order of increasing magnitude. The current estimate of the principal curve is then the set of line segments with endpoints $\mathbf{f}^{(j)}(\lambda_k^{(j)+})$ and $\mathbf{f}^{(j)}(\lambda_{k+1}^{(j)+})$. If λ_{ik}^{+} is the parameterization of the closest point to $\mathbf{x}_i$ on the line segment between $\mathbf{f}^{(j)}(\lambda_k^{(j)+})$ and $\mathbf{f}^{(j)}(\lambda_{k+1}^{(j)+})$, and d_{ik} is the distance $\|\mathbf{x}_i - \mathbf{f}^{(j)}(\lambda_{ik}^{(j)+})\|$, then $\tilde{\mathbf{x}}_i$ is the point $\mathbf{f}^{(j)}(\lambda_{ik}^{(j)+})$ for which index $k \in (0, \ldots, n-2)$ corresponds to smallest value of d_{ik} (see Figure 3.7). The λ-parameterization for $\mathbf{x}_i$ can then be computed as $\|\mathbf{f}^{(j)}(\lambda_k^{(j)+}) - \tilde{\mathbf{x}}_i\| + \lambda_k^{(j)+}$. Note that it is an approximation to consider the principal curve estimate a sequence of line-segments. The actual form of the principal curve estimate between consecutive points depends on the kind of smoother used in the conditional expectation step. For a spline smoother which penalizes acceleration, the functional form of the curve-estimate is actually a cubic spline. The line segment approximation tends to work well enough in practice, however.

Unfortunately, finding each $\tilde{\mathbf{x}}_i$ in this way involves computing the nearest point to each of $(n-1)$ line segments, so the entire projection step is an $O(n^2)$ operation. If we are willing to assume that the line segment containing $\tilde{\mathbf{x}}_i$



(a) Iteration 1: Starting from the first principal component.



(b) Iteration 2: Principal curve estimate is close to converging.

Figure 3.6: Result of first two iterations of the principal curves algorithm, starting from a projection onto the first principal component. The data-points, the larger dots, are from a sine curve with some added noise. The smaller dots are the projections of the data points onto the principal curve estimates.

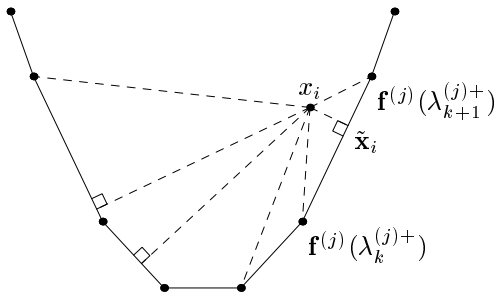


Figure 3.7: Projection step

has at least one endpoint in the set of l points $\mathbf{f}^{(j)}(\lambda_k^{(j)+})$ that are closest to $\mathbf{x}_i$, we can dramatically reduce the expense of the search. We can use a method such as kd-trees to find the l nearest neighbors in $\mathbf{f}^{(j)}(\lambda_k^{(j)+})$, then assemble all segments with at least one of these points as endpoints (there will not be less than $l-2$ and not more than $2l$ such segments), and select $\tilde{\mathbf{x}}_i$ as the closest of the points on those segments closest to $\mathbf{x}_i$. I chose to use this approximation when I coded my implementation of the principal curves algorithm for data sets, generally using $l = 3$.

The efficiency of this method is determined by the cost of building the kd-tree once and then doing n searches for the l closest points in the tree. Building the kd-tree is an $O(n \log n)$ operation [22]. The cost of the l-nearest-neighbor search depends on the intrinsic dimensionality of the set of points in the kd-tree. Generally, if the intrinsic dimensionality is approximately 8 or greater, the naïve $O(n^2)$ search is more efficient, and if the intrinsic dimension is much lower than 8, then building and searching the kd-tree to do the projection step is close to a cost of $O(n \log n)$. Since the points in the kd-tree are from a representative subset of a smooth one-dimensional curve, we can expect the kd-tree method to be much faster than the naïve search.

3.6.5 Principal curves algorithm for datasets: conditional expectation step

In the conditional expectation step, we make a new estimate of the principal curve. The goal is to estimate

$$\mathbf{f}^{(j+1)}(\lambda) = E(\mathbf{X} \mid \lambda_{\mathbf{f}^{(j)}} = \lambda), \tag{3.16}$$

for the values $\lambda \in \{\lambda_0 \ldots \lambda_{n-1}\}$ from the projection step. Because we have a data set rather than a distribution, the process for each data point $\mathbf{x}_i$ is actually to find the point which is the expected value at $\lambda_i^{(j)} \equiv \lambda^{(j)}(\mathbf{x}_i)$ of the distribution most likely to have generated the data in this vicinity of λ-space. That is, given the points which are near $\mathbf{x}_i$ in terms of their λ-values, we estimate a local distribution over λ then determine what the value of that distribution is at $\lambda_i^{(j)}$. Fortunately, as described in Section 3.3, this is exactly what a scatter plot smoother does. The projection step of the principal curves algorithm thus generates a parameterization against which a smoother can be run for each dimension of the data-set to generate the next estimate of the principal curve.

The fact that the conditional-expectation step of the principal curves algorithm uses a scatter plot smoother is what makes the principal curve of a data set a local model. Section 3.2 noted that local models should be able to use local information in the example data to build good trajectory models while balancing trajectory smoothness against modeling error. By choosing an appropriate smoother, we can leverage this information to use the principal curves algorithm to find most-likely or best-fit trajectories from multiple example human performances.

3.7 Expanding the one-dimensional representation

There are some significant problems with dimension reduction by trajectory fitting as it has been presented so far. The most obvious, discussed in this section, is that a best-fit trajectory is only a one-dimensional parameterization. The second, discussed in the next section, is the problem of "branching" in the distribution of example trajectories in the training data. The problem of principal curves over-fitting the data set is discussed in Section 3.9.

Trajectory fitting generates a one-dimensional action model, with a parameterization representing the temporal ordering of points in the best-fit trajectory. For sufficiently complex action skills, this is an inadequate parameterization. If the motion of my fingers while I grasp the handles of different mugs varies consistently in particular ways depending on the shape of the particular handle or just due to stochastic variation, then there are other important variables besides temporal ordering necessary for modeling

the grasping motion.

The solution to this problem is to construct additional parameterizations locally around the temporal-ordering. A simple first step is to estimate the local variance of the training points which map to each given point on the model trajectory. For a principal curve $\mathbf{f}$, and a distribution of training points X, this is

$$\sigma^2(\lambda) = E(\|X - \mathbf{f}(\lambda)\|^2 \mid X = \lambda), \tag{3.17}$$

where σ is the standard deviation. Such an estimate identifies those parts of the model trajectory corresponding to portions of the example trajectories which are most consistent, and also those parts of the model corresponding to portions of the example trajectories which are highly variable.

This variance estimate is also helpful for gesture recognition applications. When comparing an unknown gesture to the model, the variance estimate tells you how to weigh the distance between each sample point and the closest point on the gesture model. The use of variance here is similar to its use for gesture recognition using smoothing splines in Section 3.5.

Though useful, variance estimates do not actually increase the dimensionality of the model—the number of feature values assigned to a given data point to show where it projects onto the model. Using localized principal component analysis along the model trajectory can increase the dimensionality of the model, however. A small enough portion of a smooth model curve looks like a straight line segment, which should approximate the first principal component of the data that is nearby with respect to parameterization λ. Principal component analysis of the residuals from the nearby data points $[\mathbf{x}_i - \mathbf{f}(\lambda(\mathbf{x}_i))]$, weighted by their distance in λ-space, should give the directions of greatest variation which are orthogonal to the local model curve. If each point along the model curve is associated with one or more local principal component vectors, then the dimension of the model increases by this number of local principal components. To project a data point $\mathbf{x}$ onto this augmented model, one first projects it onto the model trajectory to get parameterization value λ, then computes the parameter $a_{\lambda,j}$ for each local principal component vector $\mathbf{v}_{\lambda,j}$ by projecting the residual vector onto that principal component vector

$$a_{\lambda,j} = [\mathbf{x} - \mathbf{f}(\lambda)] \cdot \mathbf{v}_{\lambda,j}. \tag{3.18}$$

This approach is related to other work on local dimensionality reduction, particularly that of Bregler and Omohundro [9] who blend local PCA mod-

els from neighboring patches in a high-dimensional space. Tibshirani [76] presents an alternative definition for principal curves which is very related to such a blend of locally linear models. The work on mixtures of probabilistic principal component analyzers of Tipping and Bishop [77] is also very similar.

Augmenting the dimensionality of a one-dimensional trajectory model by using local PCA is like adding hyper-ellipsoidal "flesh" to a one-dimensional "skeleton" model. Use of one local principal component turns the one-dimensional curve model into a ribbon-shaped manifold in the raw-data space, and adding two local principal components turns the model into a snake-like shape. The associated singular values of the local principal components describe the length of each diameter of the local hyper-ellipsoid, serving an analogous function to the local variance value discussed earlier, but weighing an associated direction in "residual space."

As opposed to the multi-dimensional feature-models generated by NLPCA and SNLPCA, this type of multi-dimensional model is much easier to interpret. The first dimension of the model corresponds to the temporal ordering of points from the model (i.e., what part of the motion), and the remaining dimensions represent the projections of the point in the local directions of greatest variance from the best-fit trajectory. The model describes a great deal about the nature of the performances it was trained from, and it is easy to understand the meaning of the feature-scores for a particular raw data-point projected onto the model.

The problem with this approach of growing a higher-dimensional data representation from the "skeleton" of a best-fit trajectory is the issue of whether the meaning of the second or third feature score is similar in different regions of the model. For instance, imagine a nearly cylindrical distribution, where the first feature score s_0 is roughly a length along the axis of the cylinder, and the second feature score s_1 is the projection of the residual vector onto the most significant principal component of a nearly 2-dimensional circular distribution of residual vectors $[\mathbf{x} - \mathbf{h}_0(s_0)]$. Since this distribution of residual vectors is nearly cylindrical, the orientation of the first principal component of the residuals may be a highly erratic function of the first feature score. Thus, there is no consistent interpretation of the second feature score in this case. This is an issue for future research.

3.8 Branching

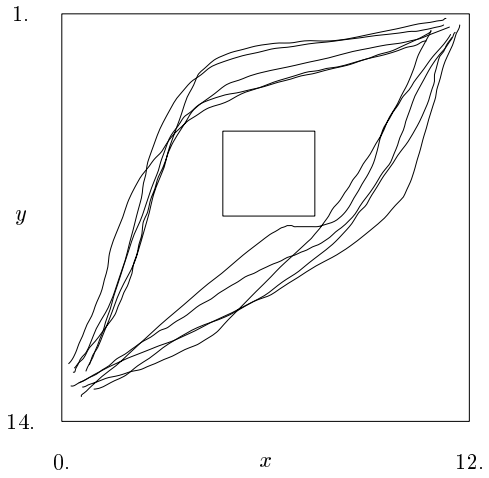
The second major problem with dimension-reduction by trajectory-fitting is branching. What if there is no single best-fit trajectory in the set of example performances, but instead two or more distinct prototypical trajectories? Imagine that the action I am modeling is my path as I walk from one side of a room to another, and that there is a table in the center of the room. Figure 3.8(a) shows a simple illustration of my paths across the room. This data was actually entered using the interface shown in Figure 5.1 on page 88, with the obstacle drawn on-screen. Sometimes I may walk to the left of the table, and sometimes to the right, but of course never through.

When I try to model a best-fit trajectory of my room-crossing action using a principal curve, one of two things can happen. The first is that the model assigns equivalent parameterization values for points in both branches of my paths as they pass the table. In this case, demonstrated in Figure 3.8(b), the trajectory smoother will draw the model trajectory through the table as it takes the expected value of the points with similar parameterizations. One method for diagnosing this problem is to examine the distribution of residual vectors from training points in each region of the trajectory. If the distribution over some region of the λ-space has two or more distinct clusters rather than a roughly Gaussian shape, this is an indication that the principal curve model has averaged two branches.

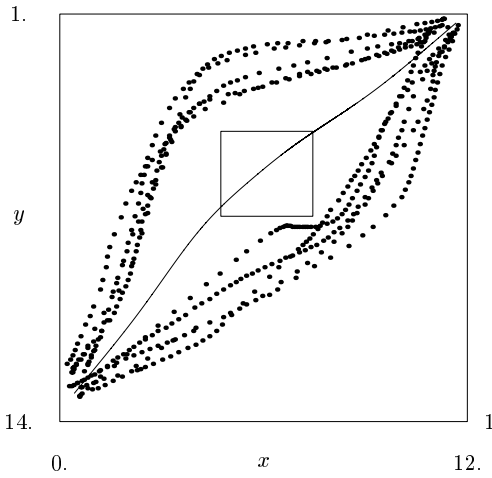
Another possibility is that the branching of paths will completely confuse the parameterization step of the model-building procedure. This will result in a failure to build the model—the principal curves algorithm will not converge.

To address the problem of branching, we need to treat each branch as an action to be modeled separately, as demonstrated in Figure 3.8(c). This solution, however, may result in two further difficulties. The first is that we need sufficient example points in each branch to build a quality model, but are subdividing a fixed pool of training data. The second difficulty is the number of models we may need to deal with. If we are modeling letter-signing, for instance, each signing gesture is a path in hand-configuration space from the final configuration of one letter-sign to another. Thus, instead of learning 26 models we (theoretically) need to learn 26^2 models to show each possible transition, which is an infeasible undertaking. This is the problem we avoided in Section 3.5 by eliminating the parameterization step of the principal curves algorithm and instead smoothing against time.

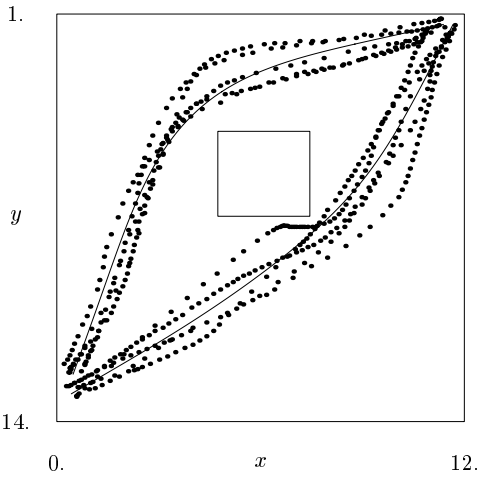
The multiple starting points were handled by the high estimated variance



(a) Paths crossing a room, going around a table.



(b) A single room-crossing skill.



(c) Two separate skills.

Figure 3.8: Room crossing as an illustration of the branching problem for local modeling. A table in the center of the room is an off-limits part of the state space for the room-crossing skill. Using a single model for the skill gives a result by which the path is averaged into the state-space obstacle. Using separate models for each path results in acceptable models.

values at the beginning of the gestures. This approach is more useful for building models for recognition rather than for performance.

3.9 Over-fitting

I fit the grasping data from the previous chapter with a principal curve, using a spline smoother for conditional expectation. The difficulty in comparing the result of this model with the one-dimensional models from PCA and NLPCA, summarized in Table 2.1 on page 35, is that principal curves can easily over-fit the data. Using a fairly crude method for cross-validation of the smoothing spline (compared to generalized cross-validation), I was easily able to reduce the error of the testing data to 43.1% while still reducing cross-validation error. Hastie and Stuetzle note that when finding principal curves, using cross-validation to weigh the smoothness penalty verses approximation error tends to over-fit the data.

The problem with using cross validation to determine the smoothing parameter when using the principal curves algorithm is that over-fitting the data tends to lengthen the principal curve as it wiggles through space to come near to each data point. The fact that there is a greater length of curve means that it is more likely that there is a point on that curve which happens to be close to a given cross-validation point. Although Hastie and Stuetzle recommend selecting the smoothing weight manually or using early stopping of the principal curves algorithm, it is difficult to do this for fitting data in a very high-dimensional space, which is by nature difficult to visualize.

It is important to note that while an over-fit principal curve can approximate the individual points in a human performance data set very accurately, such a curve will not look like any of the example performances. In particular, the local direction of the curve at its approximation of a given example point will likely be very different than the direction of the trajectory in configuration space from which the point was sampled. The greater the over-fitting in configuration space, the greater the error we would see if we plotted the velocity data from the training data against the local derivative of the principal curve. Fortunately, the methods presented in the next two chapters will allow us to fit both the position and velocity components of the training data simultaneously, and this will greatly reduce the problem of over-fitting.

Chapter 4

A spline smoother in phase space for trajectory fitting

4.1 Trajectory smoothing with velocity information

Many important problems in robotics and related fields can be addressed by fitting smooth trajectories to datasets containing several example trajectories. Chapter 3 describes some methods based upon local, non-parametric methods which can be used to model such trajectories. These models can be used for robot programming by demonstration, gesture recognition, animation, and comparison of a novice's motions to that of an expert.

Although using non-parametric models allows us to improve the quality of best-fit trajectory estimates by accounting for effects such as the trade-off between local fitting-error and the smoothness of the model curve, in the chapters to this point we have been using a collection of static positions learn to model trajectories of dynamic systems. We have not used any explicit information about what happens *between* these points. By considering position information only, we have also been limited to using only part of the state-space information of most dynamical systems. The state space for physical systems typically includes both position and velocity variables.

Using velocity information in addition to configuration information enables us to build better trajectory models. When multiple example performances have been combined into a single data set without explicit information indicating which points correspond to which example performance, for

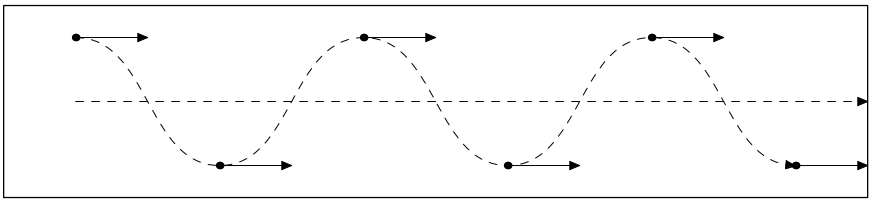


Figure 4.1: Velocity information can help a smoother determine that these are two parallel straight paths which should be averaged rather than interpolated between.

instance, we can sometimes exploit the close structural relationship between local position and velocity information to reconstruct the individual performances. Figure 4.1 depicts how local velocity information can aid in finding a best-fit trajectory from two example trajectories. A smoother operating in phase space rather than configuration space, used alone or with the principal curves algorithm, can thus potentially build better trajectory models.

Conventional smoothers such as kernel smoothers, locally weighted regression [11], and spline smoothers [16, 61, 81] are not suitable for building smoothed curves in phase space. These smoothers are designed for performing nonlinear regression—fitting response variables to explanation variables. They are not designed to operate in multidimensional spaces where dimensions are tightly coupled on a local scale, as are position and velocity variables within phase space vectors. Elementary calculus gives us this relationship for smooth curves: $\frac{1}{\epsilon}(\mathbf{x}_{t+\epsilon} - \mathbf{x}_t) \approx \dot{\mathbf{x}}_t$.

Because they cannot deal directly with such coupling, we could try two different methods for modeling phase space data with conventional smoothers: (a) smooth only in position space, then estimate the velocity values from the smoothed model (this is easiest with spline smoothers), or (b) smooth the full phase space data as if all its dimensions were independent. Method (a) has the disadvantage that all potentially useful state information in the velocity data is lost. Method (b) has the problem that although we are using the velocity information, we are not considering the close structural relationship between position and local velocity. Thus the result of method (b) will be an inconsistent trajectory through phase space where the smoothed paths in position space and velocity space are not compatible. That is, if we integrate the path in the velocity subspace, there is no reason why we should exactly

get the corresponding path in the position subspace, and if we differentiate the position-path, there is no reason why that should exactly match the velocity-path.

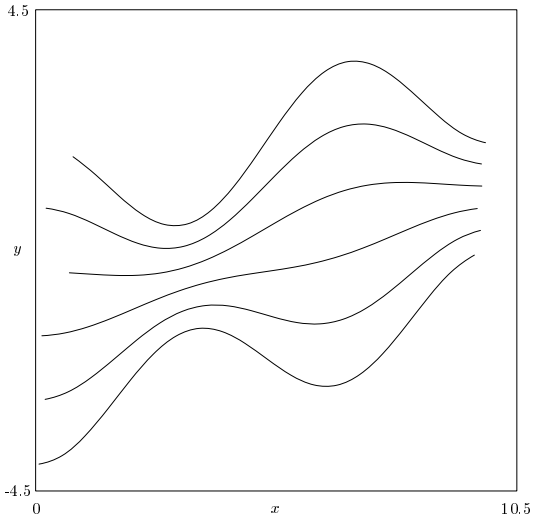
For this reason, I have derived a spline smoother which can smooth data in phase space by correctly modeling the relationship between velocity and position variables. This derivation is presented in this chapter. As discussed in Section 3.5, where we used a conventional spline smoother to model letter-signing motions, smoothers are only directly useful for modeling human performance data when we have a suitable explanation variable to smooth against. The principal curves algorithm can often generate such a suitable parameterization if one cannot be specified *a priori*. In Chapter 5, I show how my smoother can be used as part of the principal curves algorithm to find principal curves in phase space for trajectory modeling.

The formulation is based upon a smoothing spline because the polynomial form of the spline sections expresses the relationship between position and velocity in a straightforward manner. A spline function is also an explicit representation of a smooth trajectory, which is our desired output. It is a convenient form for use in interpolation, and thus well suited for tasks such as animation, or for measuring the distance between the smoothed trajectory and points sampled from other trajectories.

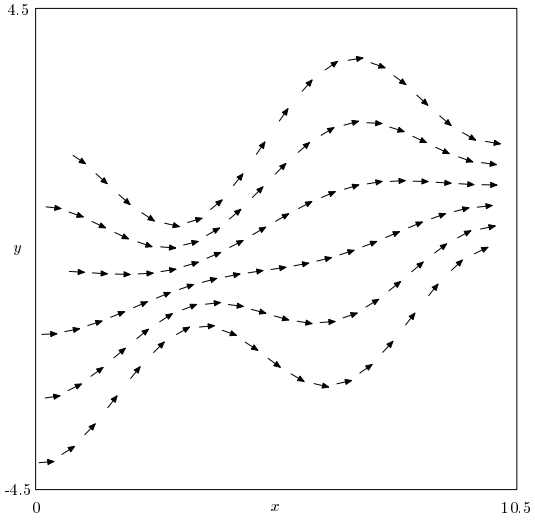
4.2 Problem formulation

Assume we are attempting to find a "best-fit" or "characteristic" trajectory f^*, over a given domain, of a given process $\frac{dy}{dx} = p_{\boldsymbol{\eta}}(x, y)$. The state equation of the process is unknown, and it is controlled by a set of unknown parameters $\boldsymbol{\eta}$ which vary stochastically between trials, so that we cannot expect two trials of the process over the given domain to result in the same trajectory. Suppose the process's trajectory to be smooth over each trial, and traversing through roughly the same region of the phase space within each region of the domain, but with some stochastic variation in position y and velocity $v \equiv \frac{dy}{dx}$ (e.g. Figure 4.2 (a)).

During a number of trials we sample y and v over a representative set of the domain x, and call the samples $(\mathbf{x}_{ts}, \mathbf{y}_{ts}, \mathbf{v}_{ts})$ where t indicates the trial index and s indicates the index of the sample within a given trial. Instead of explicitly preserving the information about which samples correspond to which trial, we sort the data points from the multiple trials into a single list



(a) Example trajectories



(b) Sampled vector field

Figure 4.2: Artificial trajectories generated over domain x using equation $y = 2\beta - \cos\left(\frac{\pi}{10}x\right) + \beta\cos\left(\frac{3\pi}{10}x\right)$ for $\beta \in \{1.0, 0.6, 0.2, -0.2, -0.6, -1.0\}$.

(x_i, y_i, v_i) such that $x_0 < x_1 < \cdots < x_{n-1}$. The data now resembles a set of samples in a vector field (e.g., Figure 4.2 (b)). We want to find the path through the field which best balances smoothness against the quality of fit. We achieve this by minimizing the cost function

$$S = p_1 \sum_{i=0}^{n-1} \left(\frac{y_i - f(x_i)}{\delta y_i} \right)^2 + p_2 \sum_{i=0}^{n-1} \left(\frac{v_i - f'(x_i)}{\delta v_i} \right)^2 + (1 - p_1 - p_2) \int_{x_0}^{x_{n-1}} (f^{(d)}(x))^2 dx, \quad (4.1)$$

which weighs accuracy in position and velocity against smoothness of the curve, where smoothness is defined as the integral of the square of the d-th derivative over the trajectory. Thus minimizing S for $p_1 = 1, p_2 = 0$ interpolates the data (i.e., strong over-fitting), while any curve f for which $f^{(d)}(x) = 0$ over $x_0 \leq x < x_{n-1}$ (e.g., a straight line for $d = 2$) will minimize S when $p_1 = p_2 = 0$. If we assume that $(y_i - f(x_i))$ and $(v_i - f'(x_i))$ are drawn from roughly Gaussian distributions at each point x_i, then δy_i and δv_i should be the corresponding estimated standard deviations at x_i. Equation (4.1) is a simple extension of the cost function for a conventional spline smoother (3.1).

The function $f = f^*$ which minimizes S is a spline[1] of order $k \equiv 2d$ with simple knots at $x_0, \ldots, x_{n-1}$ $(x_i < x_{i+1})$, and natural end conditions:

$$f^{(j)}(x_0) = f^{(j)}(x_{n-1}) = 0 \qquad \text{for } j \in \{d, \ldots, k-2\}. \quad (4.2)$$

Although there may be some arguments for minimizing jerk ($d = 3$) for certain human performance datasets, we will focus on penalizing acceleration ($d = 2$). Spline smoothing is similar to smoothing with a variable kernel, and it has been demonstrated [68] (in the conventional smoother case) that the equivalent kernel for $d = 3$ is very similar to that for $d = 2$, so we don't expect to see a dramatic difference in results between the two. The result of using $d = 2$ is that the optimal curve f^* is a cubic spline with free end conditions:

$$f''(x_0) = f''(x_{n-1}) = 0. \quad (4.3)$$

The following derivation of this smoothing spline starts in a similar manner to that presented by De Boor [16] for the conventional case which does not consider velocity information [16, 61].

[1] Note that this is proved for the conventional smoothing case, but I have yet to prove it for this case.

4.3 Solution

Given a set of n data points $(x_i, g(x_i))$ for which $x_0 < x_1 < \cdots < x_{n-1}$, the cubic spline that interpolates these points is

$$f(x) = P_i(x) \qquad \text{for } x_i \le x < x_{i+1}, \tag{4.4}$$

where each $P_i(x)$ is a fourth-order polynomial function, and $f(x)$ is continuous in position, velocity, and acceleration. This gives rise to the following smoothness constraints:

$$P_{i-1}(x_i) = P_i(x_i) = g(x_i) \tag{4.5}$$
$$P'_{i-1}(x_i) = P'_i(x_i) \tag{4.6}$$
$$P''_{i-1}(x_i) = P''_i(x_i). \tag{4.7}$$

Polynomials P_i may be expressed in Newton form

$$P_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3, \tag{4.8}$$

where the coefficient terms may be expressed in terms of divided differences[2] with the knot sequence (x_i, x_i, x_i, x_{i+1}):

$$P_i(x) = P_i(x_i) + (x - x_i)[x_i, x_i]P_i + (x - x_i)^2[x_i, x_i, x_i]P_i + (x - x_i)^3[x_i, x_i, x_i, x_{i+1}]P_i. \tag{4.9}$$

The divided differences in (4.9) are determined by this divided divided

[2]Following [16], we define the *k-th divided difference of a function g at the points* $x_i, \ldots, x_{i+k}$ (written $[x_i, \ldots, x_{i+k}]g$) to be the leading coefficient (i.e. the coefficient of x^k) of the polynomial of order $k+1$ which agrees with g at the points $x_i, \ldots, x_{i+k}$. We say that *function p agrees with function g at points* $\tau \in \{\tau_0, \ldots, \tau_n\}$ if, for each point τ which occurs m times in the sequence $\{\tau_0, \ldots, \tau_n\}$, p and g agree m-fold at τ, i.e.

$$p^{(i)} = g^{(i)} \qquad \text{for } i \in \{0, \ldots, m-1\}.$$

differences table.

	$[\ \ ]P_i$	$[\ \ ,\ \ ]P_i$	$[\ \ ,\ \ ,\ \ ]P_i$	$[\ \ ,\ \ ,\ \ ,\ \ ]P_i$
x_i	$g(x_i)$			
		b_i		
x_i	$g(x_i)$		c_i	
		b_i		$\frac{([x_i,x_{i+1}]g-b_i-c_i\triangle x_i)}{(\triangle x_i)^2}$
x_i	$g(x_i)$		$\frac{([x_i,x_{i+1}]g-b_i)}{\triangle x_i}$	
		$[x_i,x_{i+1}]g$		
x_{i+1}	$g(x_{i+1})$			

(4.10)

This gives us an expression for d_i in terms of b_i and c_i

$$d_i = ([x_i, x_{i+1}]g - b_i - c_i\triangle x_i)/(\triangle x_i)^2 \tag{4.11}$$

which we can solve for b_i:

$$b_i = [x_i, x_{i+1}]g - c_i\triangle x_i - d_i(\triangle x_i)^2. \tag{4.12}$$

Applying smoothness constraint (4.7) to (4.8) gives us another expression for d_i:

$$\begin{aligned} c_i + 3d_i\triangle x_i &= c_{i+1} \\ d_i &= \frac{1}{3\triangle x_i}(c_{i+1} - c_i). \end{aligned} \tag{4.13}$$

Using (4.12) and (4.13), we can express b_i as

$$\begin{aligned} b_i &= [x_i, x_{i+1}]g - \tfrac{2}{3}\triangle x_i c_i - \tfrac{1}{3}\triangle x_i c_{i+1} \\ &= \frac{\triangle a_i}{\triangle x_i} - \tfrac{2}{3}\triangle x_i c_i - \tfrac{1}{3}\triangle x_i c_{i+1} \qquad \text{for } i \in \{0, \ldots, n-2\}. \end{aligned} \tag{4.14}$$

Applying smoothness constraint (4.6) to (4.8) gives us the equation

$$b_{i-1} + 2\triangle x_{i-1}c_{i-1} + 3(\triangle x_{i-1})^2 d_{i-1} = b_i. \tag{4.15}$$

Using (4.14) and (4.13), we can write this constraint in terms of c_i for $i \in \{1, \ldots, n-2\}$:

$$\begin{aligned} &([x_{i-1}, x_i]g - \tfrac{2}{3}\triangle x_{i-1}c_{i-1} - \tfrac{1}{3}\triangle x_{i-1}c_i) \\ &\qquad + 2\triangle x_{i-1}c_{i-1} + 3(\triangle x_{i-1})^2\left(\frac{1}{3\triangle x_{i-1}}(c_i - c_{i-1})\right) \\ &\qquad = [x_i, x_{i+1}]g - \tfrac{2}{3}\triangle x_i c_i - \tfrac{1}{3}\triangle x_i c_{i+1}, \end{aligned}$$

which simplifies to

$$\triangle x_{i-1}c_{i-1} + 2(\triangle x_{i-1} + \triangle x_i)c_{i-1} + \triangle x_i c_{i+1} = 3\left(\frac{\triangle a_i}{\triangle x_i} - \frac{\triangle a_{i-1}}{\triangle x_{i-1}}\right). \quad (4.16)$$

Since (4.3) tells us that

$$c_0 = c_{n-1} = 0, \quad (4.17)$$

we can write the relationship between $\mathbf{a} \equiv (a_i)_0^{n-1}$ and $\mathbf{c} \equiv (c_i)_1^{n-2}$ in matrix form

$$\mathbf{Rc} = 3\mathbf{Q}^T\mathbf{a} \quad (4.18)$$

where $\mathbf{R}_{(n-2\times n-2)}$ is the symmetric tridiagonal matrix having general row

$$[\triangle x_{i-1}, \quad 2(\triangle x_{i-1} + \triangle x_i), \quad \triangle x_i]$$

$$\mathbf{R} = \begin{bmatrix} 2(\triangle x_0 + \triangle x_1) & \triangle x_1 & \dots & \mathbf{0} \\ \triangle x_1 & 2(\triangle x_1 + \triangle x_2) & \triangle x_2 & \vdots \\ & \triangle x_2 & \ddots & \\ \vdots & \ddots & \ddots & \triangle x_{n-3} \\ \mathbf{0} & \dots & \triangle x_{n-3} & 2(\triangle x_{n-3} + \triangle x_{n-2}) \end{bmatrix} \quad (4.19)$$

and $\mathbf{Q}^T_{(n-2\times n)}$ the tridiagonal matrix with general row

$$[1/\triangle x_{i-1}, \quad -1/\triangle x_{i-1} - 1/\triangle x_i, \quad 1/\triangle x_i]$$

$$\mathbf{Q}^T = \begin{bmatrix} \frac{1}{\triangle x_0} & -\frac{1}{\triangle x_0} - \frac{1}{\triangle x_1} & \frac{1}{\triangle x_1} & \dots & \mathbf{0} \\ & & & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \\ \mathbf{0} & \dots & \frac{1}{\triangle x_{n-3}} & -\frac{1}{\triangle x_{n-3}} - \frac{1}{\triangle x_{n-2}} & \frac{1}{\triangle x_{n-2}} \end{bmatrix}. \quad (4.20)$$

We can determine b_{n-1} using the derivative of (4.8), with (4.14) and (4.13) and the fact that $c_{n-1} = 0$:

$$\begin{aligned} b_{n-1} &= b_{n-2} + 2\triangle x_{n-2}c_{n-2} + 3(\triangle x_{n-2})^2 d_{n-2} \\ &= ([x_{n-2}, x_{n-1}]g - \tfrac{2}{3}\triangle x_{n-2}c_{n-2}) + 2\triangle x_{n-2}c_{n-2} + 3(\triangle x_{n-2})^2 \frac{1}{3\triangle x_{n-2}}(-c_{n-2}) \\ &= \frac{-1}{\triangle x_{n-2}}a_{n-2} + \frac{1}{\triangle x_{n-2}}a_{n-1} + \tfrac{1}{3}\triangle x_{n-2}c_{n-2}. \end{aligned} \quad (4.21)$$

Combining (4.21) and (4.14), and writing in matrix form, we have

$$\mathbf{b} = \mathbf{W}\mathbf{a} - \mathbf{Z}\mathbf{c} \tag{4.22}$$

where

$$\mathbf{W}_{n\times n} = \begin{bmatrix} \frac{-1}{\triangle x_0} & \frac{1}{\triangle x_0} & & \cdots & \mathbf{0} \\ & \frac{-1}{\triangle x_1} & \frac{1}{\triangle x_1} & & \vdots \\ \vdots & & \ddots & \ddots & \\ & & & \frac{-1}{\triangle x_{n-2}} & \frac{1}{\triangle x_{n-2}} \\ \mathbf{0} & \cdots & & \frac{-1}{\triangle x_{n-2}} & \frac{1}{\triangle x_{n-2}} \end{bmatrix} \tag{4.23}$$

and

$$\mathbf{Z}_{n\times n-2} = \frac{1}{3}\begin{bmatrix} \triangle x_0 & & \cdots & \mathbf{0} \\ 2\triangle x_1 & \triangle x_1 & & \vdots \\ & \ddots & \ddots & \\ \vdots & & 2\triangle x_{n-3} & \triangle x_{n-3} \\ & & & 2\triangle x_{n-2} \\ \mathbf{0} & \cdots & & -\triangle x_{n-2} \end{bmatrix}. \tag{4.24}$$

Using (4.18), we can write this in terms of $\mathbf{a}$ only

$$\mathbf{b} = (\mathbf{W} - 3\mathbf{Z}\mathbf{R}^{-1}\mathbf{Q}^T)\mathbf{a} = \mathbf{F}\mathbf{a} \tag{4.25}$$

where $\mathbf{F} \equiv (\mathbf{W} - 3\mathbf{Z}\mathbf{R}^{-1}\mathbf{Q}^T)$.

Now, we simplify the form of the integral term in (4.1). Over each interval (x_i, x_{i+1}), the smoothness term in (4.1) is the integral of expression $(2c_i + 6d_i(x - x_i))^2$, which is the square of the area under the straight line segment with endpoints $(2x_i, 2c_i)$ and $(2x_{i+1}, 2c_{i+1})$. Since for any straight line l

$$\int_0^h l^2(x)dx = (h/3)(l^2(0) + l(0)l(h) + l^2(h), \tag{4.26}$$

we can write the integral term as

$$(1 - p_1 - p_2)\int_{x_0}^{x_{n-1}} (f''(x))^2 dx = \tfrac{4}{3}(1 - p_1 - p_2)\sum_{i=0}^{n-2} \triangle x_i (c_i^2 + c_i c_{i+1} + c_{i+1}^2). \tag{4.27}$$

Using (4.25) and (4.27) we can express cost function (4.1) in vector form, in terms of $\mathbf{a}$ and $\mathbf{c}$

$$S = p_1(\mathbf{y}-\mathbf{a})^T\mathbf{D}_p^{-2}(\mathbf{y}-\mathbf{a}) + p_2(\mathbf{v}-\mathbf{F}\mathbf{a})^T\mathbf{D}_v^{-2}(\mathbf{v}-\mathbf{F}\mathbf{a}) + \tfrac{2}{3}(1-p_1-p_2)\mathbf{c}^T\mathbf{R}\mathbf{c}, \quad (4.28)$$

where $\mathbf{D}_p$ is the diagonal matrix $\lceil \delta y_0, \ldots, \delta y_{n-1} \rfloor$ and $\mathbf{D}_v$ is $\lceil \delta v_0, \ldots, \delta v_{n-1} \rfloor$. By (4.18) we can rewrite (4.28) as a function of $\mathbf{a}$ only:

$$\begin{aligned} S(\mathbf{a}) =& p_1(\mathbf{y}-\mathbf{a})^T\mathbf{D}_p^{-2}(\mathbf{y}-\mathbf{a}) + p_2(\mathbf{v}-\mathbf{F}\mathbf{a})^T\mathbf{D}_v^{-2}(\mathbf{v}-\mathbf{F}\mathbf{a}) \\ &+ 6(1-p_1-p_2)(\mathbf{R}^{-1}\mathbf{Q}^T\mathbf{a})^T\mathbf{R}(\mathbf{R}^{-1}\mathbf{Q}^T\mathbf{a}). \end{aligned} \quad (4.29)$$

Because $\mathbf{D}_p^{-2}$, $\mathbf{D}_v^{-2}$, and $(\mathbf{R}^{-1}\mathbf{Q}^T\mathbf{a})^T\mathbf{R}(\mathbf{R}^{-1}\mathbf{Q}^T\mathbf{a})$ are positive definite, the cost function is minimized when $\mathbf{a}$ satisfies

$$12(1-p_1-p_2)(\mathbf{R}^{-1}\mathbf{Q}^T)^T\mathbf{R}(\mathbf{R}^{-1}\mathbf{Q}^T\mathbf{a}) - 2p_1\mathbf{D}_p^{-2}(\mathbf{y}-\mathbf{a}) - 2p_2\mathbf{F}^T\mathbf{D}_v^{-2}(\mathbf{v}-\mathbf{F}\mathbf{a}) = 0. \quad (4.30)$$

Solving for $\mathbf{a}$,

$$\left[p_1\mathbf{D}_p^{-2} + p_2\mathbf{F}^T\mathbf{D}_v^{-2}\mathbf{F} + 6(1-p_1-p_2)(\mathbf{R}^{-1}\mathbf{Q}^T)^T\mathbf{R}(\mathbf{R}^{-1}\mathbf{Q}^T)\right]\mathbf{a} = [p_1\mathbf{D}_p^{-2}\mathbf{y} + p_2\mathbf{F}^T\mathbf{D}_v^{-2}\mathbf{v}], \quad (4.31)$$

and simplifying using $(\mathbf{R}^{-1})^T = \mathbf{R}^{-1}$ (because $\mathbf{R}$ is symmetric), we can solve the following linear system for $\mathbf{a}$:

$$\begin{aligned} \left[p_1\mathbf{D}_p^{-2} + p_2\mathbf{F}^T\mathbf{D}_v^{-2}\mathbf{F} + 6(1-p_1-p_2)\mathbf{Q}\mathbf{R}^{-1}\mathbf{Q}^T\right]\mathbf{a} &= [p_1\mathbf{D}_p^{-2}\mathbf{y} + p_2\mathbf{F}^T\mathbf{D}_v^{-2}\mathbf{v}] \\ \mathbf{M}\mathbf{a} &= \mathbf{z}. \end{aligned} \quad (4.32)$$

Once we know $\mathbf{a}$, we can use (4.18) to solve for $\mathbf{c}$, and by (4.25) we can compute $\mathbf{b} = \mathbf{F}\mathbf{a}$. Finally, we can use (4.13) to determine $\mathbf{d} \equiv (d_i)_{i=0}^{n-2}$. Knowing $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, and $\mathbf{d}$, we now can use (4.4) and (4.8) to compute the trajectory $f^*(x)$ over the domain $x_0 \le x \le x_{n-1}$. Figure 4.3 shows the result of this computation for the dataset plotted in Figure 4.2 (b), using the method of Section 4.7 for estimating variances $\mathbf{D}_p$ and $\mathbf{D}_v$.

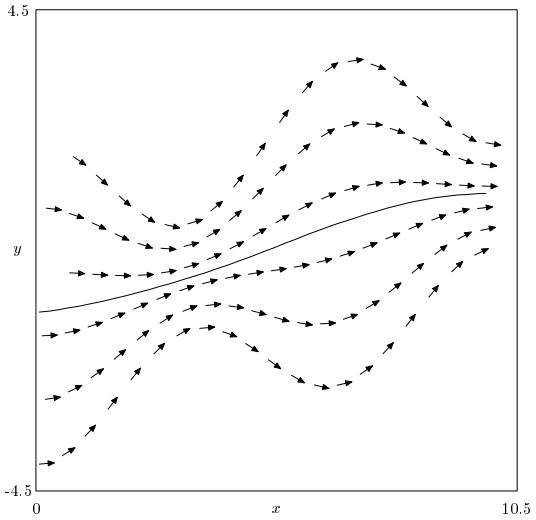


Figure 4.3: Computed interpolation using $p_1 = p_2 = 0.1$, with variance estimation.

4.4 Notes on computation and complexity

One major problem with the computation in (4.32) is the matrix $\mathbf{R}^{-1}$, which is always used in conjunction with $\mathbf{Q}^T$. Letting $\mathbf{H} = \mathbf{R}^{-1}\mathbf{Q}^T$, we can solve the following linear system for $\mathbf{H}$

$$\mathbf{R}\mathbf{H} = \mathbf{Q}^T, \tag{4.33}$$

such that we can use $\mathbf{H}$ to eliminate explicit computation of $\mathbf{R}^{-1}$ in (4.32) and (4.25). Because $\mathbf{R}$ is a tridiagonal symmetric matrix composed of $\triangle x_i$ elements (see (4.19)), it turns-out that although $\mathbf{H}$ is not a band matrix, it is highly band-dominated.

If we can adequately approximate $\mathbf{H}$ by a band matrix $\tilde{\mathbf{H}}$, then this bandedness will propagate through all terms composing $\mathbf{M}$ in (4.32) to give us a band matrix approximation $\tilde{\mathbf{M}}$. In this case, inspection of equation (4.32) shows us that matrix $\tilde{\mathbf{M}}$ not only can be computed in linear time, but also is a positive definite symmetric band matrix. Cholesky factorization of $\tilde{\mathbf{M}}$ can be performed in linear time and space [25], and then approximations to $(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d})$ can be also found in linear time and space.

If we do not approximate $\mathbf{H}$ with a band matrix, then $\mathbf{M}$ will be a non-banded positive-definite symmetric matrix, and Cholesky decomposition will

be an $O(n^3)$ operation [25], dominating the cost of the smoothing solution. Since $\mathbf{M}$ is an $(n \times n)$ matrix, storage cost of the computation will be $O(n^2)$.

The main difference in computational expense between the smoothing algorithm in [16] and our smoother stems from the fact that without velocity information the linear system may be computed in terms of $\mathbf{c}$ rather than $\mathbf{a}$ using (3.4), which is introduced on in Chapter 3 but repeated here for convenience:

$$(6(1-p)\mathbf{Q}^T\mathbf{D}^2\mathbf{Q} + p\mathbf{R})\mathbf{c} = 3p\mathbf{Q}^T\mathbf{y}.$$

This reduces the amount of computation necessary for computing the smoothed function through the data compared to the trajectory-smoother presented here. Equation (3.4) is a tridiagonal linear system which can be solved in linear time and space.

4.5 Combining points with similar parameterizations

Matrices $\mathbf{Q}^T$ and $\mathbf{W}$ are built from terms of the form $1/\triangle x_i$. This is problematic when multiple points in the data-set have the same or nearly the same parameterization value x. The solution is to replace each set of points having similar parameterization values with a single point such that the function $f = f^*$ which minimizes cost function (4.1) remains unchanged. Thus, for each set of points for which

$$|x_k - x_*| < \epsilon \qquad \text{for } k \in (a, a+1, \ldots, b) \tag{4.34}$$

we compute parameters $(y_*, \delta y_*, v_*, \delta v_*)$, of a replacement point such that

$$\frac{\partial}{\partial f(x_*)}\left(\frac{y_* - f(x_*)}{\delta y_*}\right)^2 = \frac{\partial}{\partial f(x_*)}\sum_{k=a}^{b}\left(\frac{y_k - f(x_*)}{\delta y_k}\right)^2 \tag{4.35}$$

and

$$\frac{\partial}{\partial f'(x_*)}\left(\frac{v_* - f'(x_*)}{\delta v_*}\right)^2 = \frac{\partial}{\partial f'(x_*)}\sum_{k=a}^{b}\left(\frac{v_k - f'(x_*)}{\delta v_k}\right)^2. \tag{4.36}$$

Simplifying each side of (4.35) gives us

$$\frac{-2(y_* - f(x_*))}{(\delta y_*)^2} = -2\left(\sum_{k=a}^{b} \frac{y_k}{(\delta y_k)^2} - f(x_*)\sum_{k=a}^{b} \frac{1}{(\delta y_k)^2}\right). \tag{4.37}$$

Setting

$$\frac{1}{(\delta y_*)^2} = \sum_{k=a}^{b} \frac{1}{(\delta y_k)^2}, \tag{4.38}$$

$$y_* = (\delta y_*)^2 \sum_{k=a}^{b} \frac{y_k}{(\delta y_k)^2} \tag{4.39}$$

ensures the equality in (4.37). A similar argument from (4.36) gives us

$$\frac{1}{(\delta v_*)^2} = \sum_{k=a}^{b} \frac{1}{(\delta v_k)^2}, \tag{4.40}$$

$$v_* = (\delta v_*)^2 \sum_{k=a}^{b} \frac{v_k}{(\delta v_k)^2}. \tag{4.41}$$

Thus to smooth data $(\mathbf{x}, \mathbf{y}, \mathbf{v})$ given $(\delta\mathbf{y}, \delta\mathbf{v})$, we map these vectors via (4.38)-(4.41) to $(\mathbf{x}_*, \mathbf{y}_*, \mathbf{v}_*)$ and $(\delta\mathbf{y}_*, \delta\mathbf{v}_*)$ such that $x_{*i} + \epsilon < x_{*i+1}$, then smooth the mapped points. Many practical uses of the smoother will require mapping the smoothed-results back to the order of the original data, which may require mapping individual smoothed points to more than one location in the original ordering.

In most cases $\delta v_* = \delta v_k$ for $k \in (a, \ldots, b)$, because δv is an estimated variance which is a function of x. Then, (4.41) and (4.40) reduce to

$$\frac{1}{(\delta v_*)^2} = (b - a + 1)(\delta v_a)^2, \tag{4.42}$$

$$v_* = \frac{1}{b - a + 1} \sum_{k=a}^{b} v_k, \tag{4.43}$$

and the same happens for the position values. This happens for a posteriori computations of variance, as in Section 4.7.

4.6 Multi-dimensional smoothing

When smoothing data of more than one dimension, the cost function becomes

$$S = p_1 \sum_{i=0}^{n-1} \|(\mathbf{y}_i - \mathbf{f}(x_i))\mathbf{D}_{\mathrm{p}i}^{-1}\|^2 + p_2 \sum_{i=0}^{n-1} \|(\mathbf{v}_i - \mathbf{f}'(x_i))\mathbf{D}_{\mathrm{v}i}^{-1}\|^2 + (1 - p_1 - p_2) \int_{x_0}^{x_{n-1}} \|\mathbf{f}^{(d)}(x)\|^2 dx \quad (4.44)$$

where $\mathbf{D}_{\mathrm{p}i}$ and $\mathbf{D}_{\mathrm{v}i}$ are diagonal weighting matrices. The process of minimizing this expression can be split into a set of separate minimizations of the form (4.1), one for each dimension of the data. Thus, there is no significant difference in method of solution. The method for combining points with similar parameterizations (Section 4.5) can also be performed separately for each dimension.

Note that if we use the same weights for each dimension, $\mathbf{D}_{\mathrm{p}i} = (\delta y_i)\mathbf{I}$ and $\mathbf{D}_{\mathrm{v}i} = (\delta v_i)\mathbf{I}$, then we need only compute matrix $\mathbf{M}$ from (4.32) once for the entire multi-dimensional smoothing problem. Moreover, we can compute its Cholesky decomposition once, and use this decomposition to smooth each dimension of the data. Thus for each separate dimension of the smoothing problem, we need only multiply a vector by a matrix (a band matrix, if we are using the band matrix approximation $\tilde{\mathbf{H}}$), and sum the pairwise product of two vectors to form $\mathbf{z}$, then use forward and backward substitution to solve for $\mathbf{a}_i$.

4.7 Estimation of variances

In the problem formulation (Section 4.2), we said that if we assume that $(y_i - f(x_i))$ and $(v_i - f'(x_i))$ are drawn from roughly Gaussian distributions at each point x_i, then δy_i and δv_i should be the corresponding standard deviations at x_i. The variances are thus $(\delta y_i)^2$ and $(\delta v_i)^2$. Because we rarely know these variances *a priori*, we need some way to estimate them from the training data. If we assume that the variances are smooth functions of domain x, then we can estimate them by locally weighing error residuals from an unweighted iteration of the smoothing procedure [69]. In this case, we first smooth data $(\mathbf{y}, \mathbf{v})$ using unit weights $\delta y_i = \delta v_i = 1$ to obtain $(\mathbf{a}_{\mathrm{u}}, \mathbf{b}_{\mathrm{u}})$, and calculate the unweighted residuals $\mathbf{r}_{\mathrm{pu}} = (\mathbf{y} - \mathbf{a}_{\mathrm{u}})$, $\mathbf{r}_{\mathrm{du}} = (\mathbf{v} - \mathbf{b}_{\mathrm{u}})$. Then, the

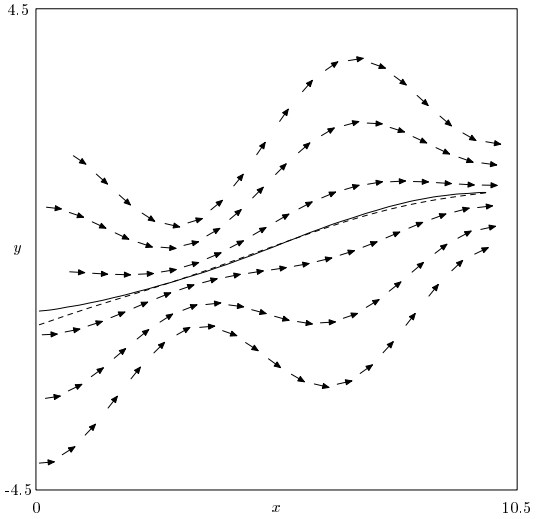


Figure 4.4: Effect of variance estimation with $p_1 = p_2 = 0.1$, $k = 5$. Dashed line unweighted, solid line weighted.

variances are estimated with a local moving average of squared residuals

$$\begin{aligned}(\delta y_{1i})^2 &= (n_i - m_i + 1)^{-1} \sum_{k=m_i}^{n_i} r_{\mathrm{pu}k}^2 \\ (\delta v_{1i})^2 &= (n_i - m_i + 1)^{-1} \sum_{k=m_i}^{n_i} r_{\mathrm{du}k}^2,\end{aligned} \tag{4.45}$$

where

$$m_i = \max(0, i - k) \qquad \text{and} \qquad n_i = \min(n - 1, i + k). \tag{4.46}$$

Silverman [69], in the context of conventional spline smoothing, explains that this method is generally reliable because highly accurate estimates of the variances are generally not necessary. He suggests that $k = 5$ has produced good results for data sets of moderate size, and this has produced reasonable results for our spline smoother as well. Figure 4.4 shows the effect of this kind of variance estimation on the example dataset, with $k = 5$.

If desired, another iteration of variance re-estimation can also be performed by smoothing with our previous estimates $(\delta y_{1i}, \delta v_{1i})$ to obtain $(\mathbf{a}_1, \mathbf{b}_1)$

and residuals $\mathbf{r}_{\mathrm{p}1} = (\mathbf{y} - \mathbf{a}_1)$, $\mathbf{r}_{\mathrm{d}1} = (\mathbf{v} - \mathbf{b}_1)$:

$$\begin{aligned}(\delta y_{2i})^2 &= (n_i - m_i + 1)^{-1}(\delta y_{1i})^2 \sum_{k=m_i}^{n_i} r_{\mathrm{p}1k}^2 \\ (\delta v_{2i})^2 &= (n_i - m_i + 1)^{-1}(\delta v_{1i})^2 \sum_{k=m_i}^{n_i} r_{\mathrm{d}1k}^2.\end{aligned} \tag{4.47}$$

If we are performing multidimensional smoothing and want to use a single variance estimate for all dimensions of each point, we perform the unweighted smoothing on data $(\mathbf{Y}, \mathbf{V})$ using weights $\mathbf{D}_{\mathrm{pu}} = \mathbf{D}_{\mathrm{vu}} = \mathbf{I}$ to obtain $(\mathbf{A}_{\mathrm{u}}, \mathbf{B}_{\mathrm{u}})$, and calculate the unweighted residuals $\mathbf{r}_{\mathrm{pu}i} = (\mathbf{y}_i - \mathbf{a}_{\mathrm{u}i})$, $\mathbf{r}_{\mathrm{du}i} = (\mathbf{v}_i - \mathbf{b}_{\mathrm{u}i})$. The variances are then estimated as

$$\begin{aligned}(\delta y_{1i})^2 &= (n_i - m_i + 1)^{-1} \sum_{k=m_i}^{n_i} \|\mathbf{r}_{\mathrm{pu}k}\|^2 \\ (\delta v_{1i})^2 &= (n_i - m_i + 1)^{-1} \sum_{k=m_i}^{n_i} \|\mathbf{r}_{\mathrm{du}k}\|^2 \\ \mathbf{D}_{\mathrm{p}1i} &= (\delta y_{1i})\mathbf{I} \\ \mathbf{D}_{\mathrm{v}1i} &= (\delta v_{1i})\mathbf{I},\end{aligned} \tag{4.48}$$

A second iteration can be performed by smoothing with $(\mathbf{D}_{\mathrm{p}1}, \mathbf{D}_{\mathrm{v}1})$ to obtain $(\mathbf{A}_1, \mathbf{B}_1)$ and residuals $\mathbf{r}_{\mathrm{p}1i} = (\mathbf{y}_i - \mathbf{a}_{1i})$, $\mathbf{r}_{\mathrm{d}1i} = (\mathbf{v} - \mathbf{b}_{1i})$:

$$\begin{aligned}(\delta y_{2i})^2 &= (n_i - m_i + 1)^{-1}(\delta y_{1i})^2 \sum_{k=m_i}^{n_i} \|\mathbf{r}_{\mathrm{p}1k}\|^2 \\ (\delta v_{2i})^2 &= (n_i - m_i + 1)^{-1}(\delta v_{1i})^2 \sum_{k=m_i}^{n_i} \|\mathbf{r}_{\mathrm{d}1k}\|^2 \\ \mathbf{D}_{\mathrm{p}2i} &= (\delta y_{2i})\mathbf{I} \\ \mathbf{D}_{\mathrm{v}2i} &= (\delta v_{2i})\mathbf{I}.\end{aligned} \tag{4.49}$$

4.8 Windowing variance estimates

Because the recorded trajectories may not all begin and end at the exact same locations in the domain, the data at the beginning and end of the smoothing

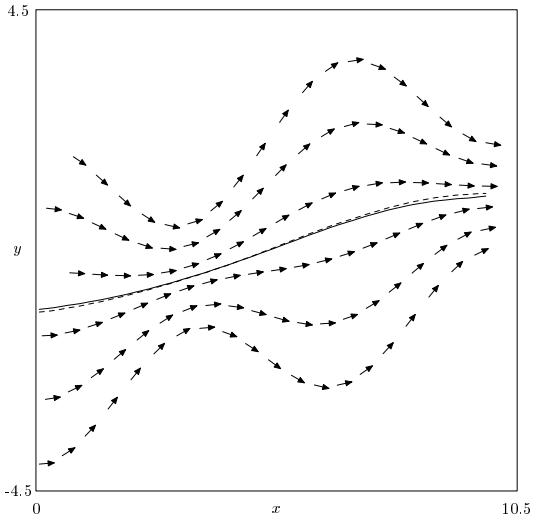


Figure 4.5: Effect of windowing the variance estimates. Dashed line is without windowing, solid with windowing.

domain will often be sparser and less representative of the underlying process than data in the middle of the domain. Therefore, we may want to adjust the weights of the points at the extremes of the domain to make them count less in the smoothing process. This can be easily accomplished by applying a windowing function, such as the Hamming function, to the first and last k points in the dataset:

$$\begin{aligned} \delta y_i &\leftarrow (0.54 + 0.46\cos(\pi(k-j)/k))\delta y_i \\ \delta v_i &\leftarrow (0.54 + 0.46\cos(\pi(k-j)/k))\delta v_i, \end{aligned} \tag{4.50}$$

where

$$j = \begin{cases} i & \text{for } i \in \{0, \ldots, k-1\} \\ n-i-1 & \text{for } i \in \{n-1-k, \ldots, n-1\}. \end{cases}$$

The effects of windowing the example data-set are shown in Figure 4.5. In this case, the effect is minimal because the variance estimation procedure has solved most of the problem on its own.

4.9 The effect of velocity information

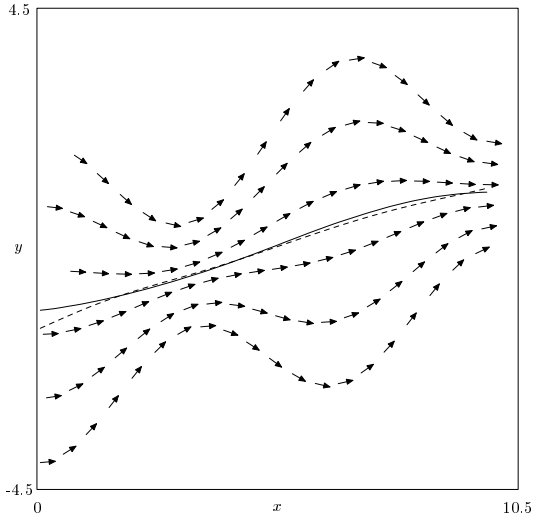
The effects of smoothing with and without velocity information are demonstrated in Figures 4.6 and 4.7. In 4.6(a), the effects of the velocity information are most noticeable at the ends of the plot. Since the trajectories starting at the lowest y values tend to start slightly before those with higher initial y values, we see that the trajectory fitted using only position information starts low, with a positive first-derivative which does not match those of the sampled points around it. The end of the trajectory has a similar problem, but it is less pronounced since the local variance of the sample points is smaller. The fitted trajectory which weighs velocity information has a more plausible beginning and ending, as we would expect. Figure 4.6(b), shows an example where a random half the of data-points are removed. The trajectory fitted using velocity information more closely tracks a single example trajectory. Figure 4.7 shows the effect of velocity information for an experimental data-set that is presented in Chapter 5.

4.10 Cross-validation

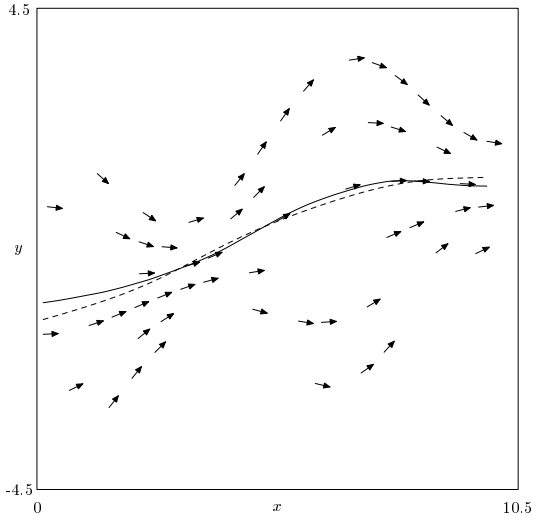
Recall from Section 3.3 that cross-validation [73] is typically used to automatically select the smoothing parameter p for a conventional spline smoother, and that the work of Craven and Wahba [13] and Hutchinson and de Hoog [31] show how to compute this parameter in closed-form and in linear time as part of the smoothing process.

I have not attempted to derive a similar generalized cross-validation method for the smoother presented in this chapter, and suspect that it may not be a simple extension of the generalized cross-validation formulation for conventional spline smoothing. For deriving generalized cross-validation, conventional spline smoothers are generally formulated using the mathematics of reproducing kernel Hilbert spaces [3, 81]. I do not currently know whether the spline smoother in phase space can be formulated in this manner.

The following chapter uses this smoother within the principal curves algorithm. Cross-validation is not used for this purpose since, as described in Section 3.9, it tends to result in over-fitting when used within the principal curves algorithm in configuration space. Whether cross-validation would work better in phase space is a question for further research.

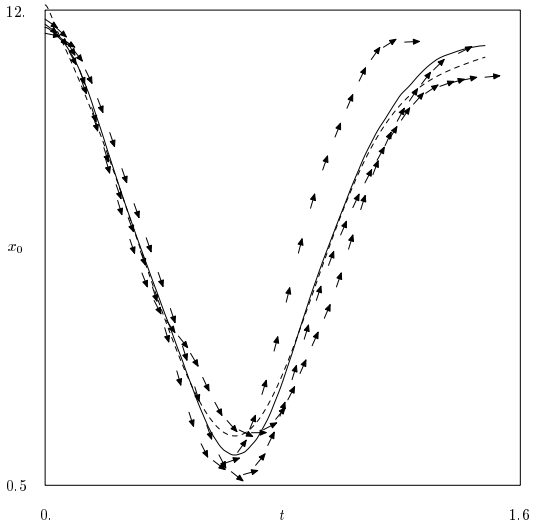


(a) Solid: $p_1 = p_2 = 0.05$, dashed: $p_1 = 0.1$, $p_2 = 0$.

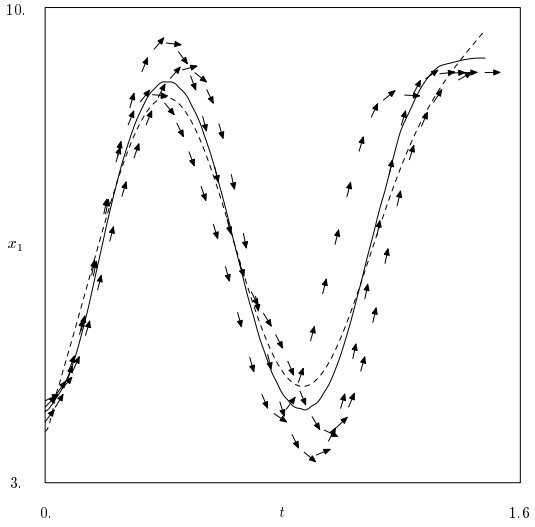


(b) Solid: $p_1 = 0.01$, $p_2 = 0.09$, dashed: $p_1 = 0.1, p_2 = 0$.

Figure 4.6: Effect of derivative information. Plots use local variance estimation ($k = 5$).



(a) Smoothing of horizontal coordinate



(b) Smoothing of vertical coordinate

Figure 4.7: Smoothing data from multiple examples of a drawing motion. Solid line: $p_1 = 0.645$, $p_2 = 0.35$. Dashed: $p_1 = 0.995$, $p_2 = 0.0$. No variance estimation.

Chapter 5

Principal curves in phase space for trajectory fitting

5.1 Model formulation

In the last chapter we derived a spline smoother which, when given a parameterization to smooth against, can generate a best-fit trajectory through phase space. However, it is rarely possible to provide a good parameterization variable *a priori*. For instance, although time is often the most convenient variable against which to smooth human performance data, it is generally only satisfactory when constant-speed motion is an integral part of the action being learned.

In this chapter, we demonstrate how the principal curves algorithm can be used in combination with the smoother from Chapter 4 to fit trajectories through phase space without an *a priori* parameterization. Chapter 3 showed how the principal curves algorithm can construct such a parameterization for data sets in configuration space. In phase space, this process leads to excellent best-fit trajectory models for smooth motions. The assumptions necessary for the smoother and principal curves algorithm to work together can be combined into the following list:

1. The training data consists of samples from several runs of a process consisting of a nominal trajectory (structural component), with some stochastic deviation.

2. The stochastic deviation from the nominal trajectory is a function of the relative "completedness" of the trajectory (i.e., roughly a function

of time over the trajectory, but taking into account that some example trajectories will be executed more quickly than others), and that at any given state of completedness, the probability density of the stochastic deviation over many trials can be adequately approximated by a zero-mean Gaussian function with some unknown standard-deviation.

3. Points with similar positions and velocities in the training-set can be assumed to correspond to the same part of the nominal trajectory. Thus, we assume that the nominal trajectory does not intersect itself in phase space, or come very close to intersecting itself.

4. The position and velocity of the nominal trajectory are smooth functions in time, and the position and velocity of the example trajectories are expected to be smooth functions in time.

Given these assumptions, we can find the best-fit trajectory by computing the trajectory model most likely to have generated the given training data. Moreover, given estimates of standard deviations at each point along the best-fit trajectory, we should be able to estimate the probability that a given data set was generated by that model.

5.2 Input data

We are interested in finding a best-fit or most-likely trajectory from a set of example trajectories. Typically, the k-th such example trajectory is stored in the form $(\mathbf{t}_k(n_k), \mathbf{X}_k(m \times n_k))$, where the i-th column vector of $\mathbf{X}_k$ is the m-dimensional configuration-space vector of the example-performance sampled at time t_{ki}.

Our trajectory-smoothing algorithm expects as input a data set of phase space samples from all the example trajectories, where to which example each data-point corresponds is not (explicitly) important. These data points can be stored in the matrix[1]

$$\mathbf{Y}^{(\mathrm{t})} = \begin{bmatrix} \mathbf{X} \\ \mathbf{V}^{(\mathrm{t})} \end{bmatrix} \tag{5.1}$$

[1] In this chapter, $\mathbf{y}$ refers to phase space data: the combination of position and velocity information. This is in contrast to Chapter 2 where $\mathbf{y}$ is used for points that are not in the training set, and in contrast to Chapter 3 where it use used for response variables (e.g., as opposed to explanation variables).

such that $\mathbf{X} = [\mathbf{X}_0|\mathbf{X}_1|\dots]$, and $\mathbf{V}^{(\mathrm{t})}$ contains the measured or estimated time-derivative of each element of $\mathbf{X}$. Later we will map the velocity values into a projection space and a conditional-expectation space, and give these mapped vectors different superscripts to avoid confusion. Vector $\mathbf{x}_i$, the i-th column-vector of $\mathbf{X}$, represents the configuration of the system at time t_i of some training example, and x_{ij} represents the value of the j-th element of this vector. Therefore, $\mathbf{v}_i = (\frac{d}{dt}\mathbf{x}_i)|_{t_i}$, and $v_{ij} = (\frac{d}{dt}x_{ij})|_{t_i}$.

Rarely is explicitly-measured velocity information available to us. We usually need to estimate velocities from position data. There are numerous methods for estimating these time-derivatives from sampled data, but since numerical differentiation of sensor measurement tends to generate noisy results, it is best to smooth the data before taking the numerical derivative. A spline smoother, as described in Section 3.3, is a useful tool here because it can simultaneously smooth and take the time-derivative of a signal, it does not require samples to be evenly spaced in time, and it allows an explicit tradeoff between error in approximation of the data-set and smoothness of the sampled inputs.

In this chapter, we will be using an example of finding a best-fit trajectory of an α-drawing skill based on examples from a user drawing onto a graphical computer interface using a mouse as a pointing device, as shown in Figure 5.1. This interface samples two-dimensional motion of the mouse (x_0, x_1, t) within the drawing window to record multiple examples of a given drawing motion. Two reasons we use this data are that a 2-dimensional position space is easier to visualize and understand than a higher-dimensional one, especially since even this results in a 4-dimensional phase space; and that a 2-dimensional position space is constrained enough that self-intersecting paths are likely and easy to demonstrate. Figure 5.2 shows the result of using a spline smoother to reduce the noise and jitter to and estimate velocity of data-points from an example performance recorded within this interface.

5.3 Probability, distance metrics, and cost function

Our approach to finding a best-fit trajectory will be to find the principal curve of the training data through phase space. Recall from Section 3.6.1 that a principal curve is self-consistent with respect to a given distribution.

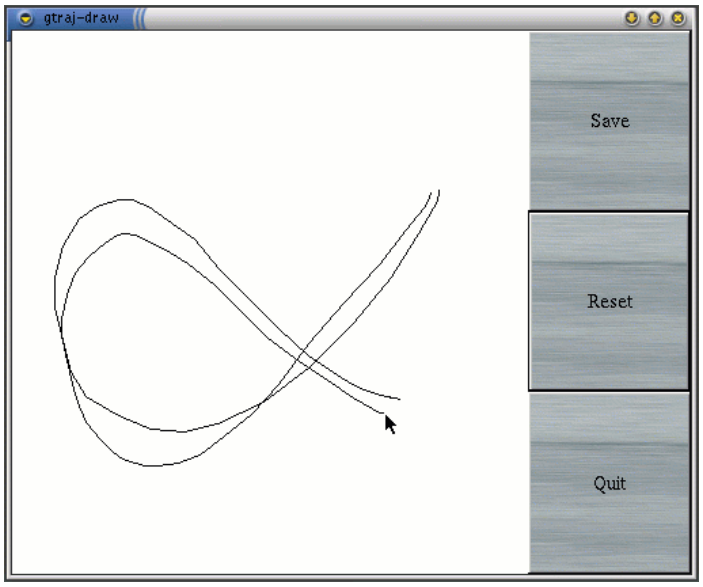


Figure 5.1: Figure-drawing interface

This means that curve $\mathbf{f}$ is a principal curve of random variable X if

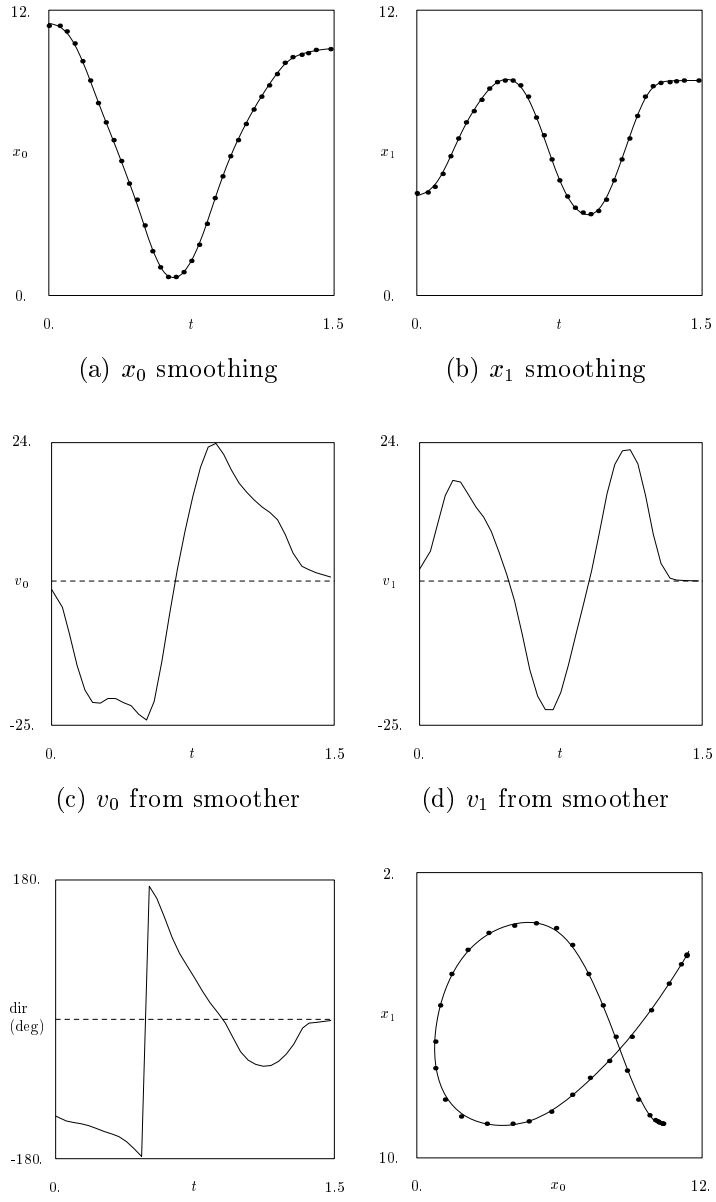
$$\mathbf{f}(\lambda) = E(X \mid \lambda_{\mathbf{f}}(X) = \lambda),$$

where $\lambda_{\mathbf{f}}$: $\mathbb{R}^p \rightarrow \mathbb{R}^1$ is the projection index which specifies onto which point of $\mathbf{f}$ each point X maps (this is Equation (3.13) from page 54). Equation (3.12) (shown here for convenience)

$$\lambda_{\mathbf{f}}(\mathbf{x}) = \sup_{\lambda} [\lambda : \|\mathbf{x} - \mathbf{f}(\lambda)\| = \inf_{\mu} \|\mathbf{x} - \mathbf{f}(\mu)\|]$$

specifies that the projection $\mathbf{f}(\lambda(\mathbf{x}))$ of a point $\mathbf{x} \in X$ onto principal curve $\mathbf{f}$ is the point on that curve which is closest to $\mathbf{x}$. When determining the principal curve of a set of discrete data-points rather than of a continuous distribution, we must use a smoother to estimate the conditional expectation in (3.13).

The trajectory smoother based on principal curves finds a best-fit trajectory by iteratively improving a trajectory-estimate. It first uses the current estimated trajectory to build a parameterization against which to smooth the data-set (projection), and then it smoothes each dimension of the training data against that parameterization to get an improved trajectory estimate



(a) x_0 smoothing

(b) x_1 smoothing

(c) v_0 from smoother

(d) v_1 from smoother

(e) Motion direction in smoothed plot

(f) Smoothed 2D plot in position space

Figure 5.2: Preprocessing of a 2D alpha-drawing by spline smoothing

(conditional expectation). If the trajectory converges to a stable estimate, then it is self-consistent. An example of this convergence for the conventional principal curve is shown in Figure 3.6 on page 57. The initial estimate, the straight-line in Figure 3.6(a), is the first principal component. This is the typical starting-point for the principal curves algorithm when it is used for nonlinear regression. In contrast to the regression example, the iterative process for trajectory smoothing is more complicated because *we cannot use the same space for both projection and conditional expectation.* Moreover, we will see later in this chapter that the first principal component is not necessarily a good initial estimate of a best-fit trajectory.

The projection phase of the principal curves algorithm maps each point in the training data to the closest point on the current estimate of the principal curve. The assumption here is that the probability that a given data-point corresponds to a given point on a model-trajectory is a monotonically decreasing function of the distance between these points. Thus, finding a principal curve of a data-set is entirely dependent upon having a meaningful distance metric in the space used for projection, or *projection space.*

In the conditional-expectation step we will be using the smoother described in Chapter 4, which minimizes a cost function, one portion of which penalizes a measure of distance between the data-points and the projection of these points onto the current estimate of the principal curve (the other portion of the cost function will be a measure of the smoothness of the estimated principal curve). This distance-cost is computed in the *conditional-expectation space*, which we will define in such a way as to make velocity values consistent with the position space values. For the position-space trajectory to be consistent with the velocity-space trajectory, the derivative of the configuration-space variables with respect to the parameterization variable should be equal to the velocity-space variables, scaled by some constant (i.e., speed). In other words, the derivative of the configuration-space variables with respect to the parameterization variable specifies a line upon which the velocity values must lie.

To build a self-consistent curve using projection and conditional-expectation in phase space, not only does the curve need to be a self-consistent path in phase space, but the distance-cost in conditional-expectation space must be equal (or at least proportional) to the corresponding distance-metric in projection space. When this is true, then a curve which is stable after an iteration of the principal curves algorithm is one where the expected value of all data-points which map to a point on the principal curve is the value

of that point. The next few sections will demonstrate how this result can be achieved.

5.4 Projection space

The projection step of the principal curves algorithm generates a parameterization against which a smoother may be run. As described in Section 3.6.4, this step assigns to each point in the training data a parameterization value which corresponds to the "closest" point on the current principal curve estimate. Thus, we need to define a metric function to compare distances in phase space. In this section, we tailor such a metric specifically for the purpose of estimating best-fit trajectories from human performance data.

There are multiple challenges in designing this metric. The position and velocity dimensions are obviously qualitatively different from one another, and even the various position dimensions may represent measurements with different units and relative magnitudes (e.g., angles in radians, distances in centimeters, forces in Newtons, etcetera). Nevertheless, our ability to choose how to weigh each variable allows us to make our assumptions explicit, and to thus construct a metric which is meaningful rather than arbitrary. Figure 5.3 demonstrates why velocity information, in addition to position information, is important in determining to which part of a the model a given point projects. Our metric will need to balance distance in position dimensions against distance in velocity dimensions, and also speed against direction within the velocity dimensions.

To be compatible with the error terms in the smoothing cost function (5.12) (described in Section 5.5), we base our distance metric $\|\cdot\|_{(\mathrm{prj})}$ on a sum of squared errors formulation:

$$\begin{aligned} \|\Delta\mathbf{y}\|_{(\mathrm{prj})} = \|\Delta\mathbf{y}^{(\mathrm{prj})}\| &= \sqrt{\sum_{j<2m} (\Delta y_j^{(\mathrm{prj})})^2} \\ &= \sqrt{\sum_{j<m} (\Delta x_j^{(\mathrm{prj})})^2 + \sum_{j<m} (\Delta v_j^{(\mathrm{prj})})^2}. \end{aligned} \tag{5.2}$$

Since this metric is used only for *comparing* distances to find nearest neighbor points rather than for measuring *absolute* distance values, in practice we can instead use any monotonic function of the metric. We thus actually use the square of distance (5.2) when we compare distances. This not only saves us

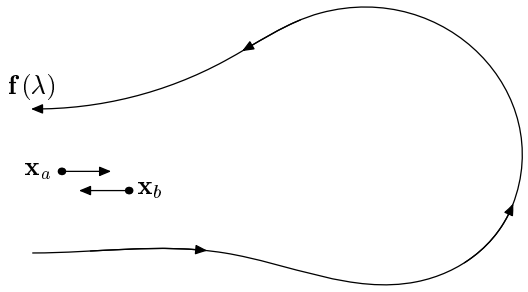


Figure 5.3: When determining to which part of a model curve a point maps, its direction and speed are relevant as well as its position. In this case, we probably want points $\mathbf{x}_a$ and $\mathbf{x}_b$ each to map to the closest point of the model curve *that is compatible in direction*, rather than the absolute closest point in position only.

the computational expense of the square-root, a transcendental operation, but also allows us to compute the phase space metric as a sum of separate position and velocity terms:

$$\|\Delta\mathbf{y}\|^2_{(\mathrm{prj})} = \|\Delta\mathbf{x}\|^2_{(\mathrm{pprj})} + \|\Delta\mathbf{v}\|^2_{(\mathrm{vprj})}. \tag{5.3}$$

Moreover, instead of just building a metric function $\|\cdot\|_{(\mathrm{prj})}$ in the input space, we build a *projection space*

$$\mathbf{Y}^{(\mathrm{prj})} = \begin{bmatrix}\mathbf{X}^{(\mathrm{prj})} \\ \mathbf{V}^{(\mathrm{prj})}\end{bmatrix}, \tag{5.4}$$

within which the L_2 norm is equivalent to the projection metric. This allows us to map all the data points into projection space at the beginning of the projection step, and then use sum of squared errors to compare distances. This greatly simplifies implementation issues, particularly when using the projection method based on kd-trees discussed in Section 3.6.4.

The position dimensions should be scaled such that errors of equal scalar magnitude in each dimension should be considered equally "unlikely" or "bad":

$$E(x_{ia} - x_{ib} \mid a \neq b) = E(x_{jc} - x_{jd} \mid c \neq d) \qquad \forall i, j. \tag{5.5}$$

Likewise, each velocity dimension should be scaled such that a difference of of a given magnitude in one dimension is equivalently unlikely or bad when

compared to the same difference in another:

$$E(v_{ia} - v_{ib} \mid a \neq b) = E(v_{jc} - v_{jd} \mid c \neq d) \qquad \forall i, j. \tag{5.6}$$

In the following discussion, we will assume that such scaling to comparable expected error magnitudes in each position and velocity dimension has already been done.

Once we have accounted for scaling the relative distance weights within the position and velocity dimensions, we need to weigh the sum of the contributions from the position dimensions against those from the velocity dimensions. We will do this in a manner similar to the smoother cost function (4.1): the ratio p_2/p_1 is used to weigh the relative contribution of squared velocity error compared to squared position error. Section 5.7 will discuss how to select this ratio. We write the distance metric for position space as

$$\|\Delta\mathbf{x}\|^2_{(\text{pprj})} = p_1\|\Delta\mathbf{x}\|^2, \tag{5.7}$$

so to map the position dimensions into projection space we use

$$\mathbf{x}_i^{(\text{prj})} = \sqrt{p_1}\mathbf{x}_i. \tag{5.8}$$

Weighting parameter p_2 will scale the velocity component of the distance metric once we determine its form.

As we determine the form of the velocity component of the projection metric, we should note that velocity dimensions $\mathbf{v}$ represent both a direction of motion in space $\hat{\mathbf{v}} = \frac{\mathbf{v}}{\|\mathbf{v}\|}$ and the speed of motion in that direction $\|\mathbf{v}\|$. Even when the training examples for a given action are performed at very different speeds, we want corresponding states in each example performance to map to the same parameterization value. It is thus useful to focus the velocity metric more on direction of motion rather than speed when speed is "great enough." When speed is small, especially when it is very close to zero, then direction should matter less. To focus the distance metric on direction only when speed is high, we can scale the velocity vectors using the tanh function

$$\tanh(a) \equiv \frac{e^a - e^{-a}}{e^a + e^{-a}}, \tag{5.9}$$

(shown in Figure 5.4) or a similar envelope function which asymptotically approaches 1.0 when the speed is high and which approaches 0.0 linearly

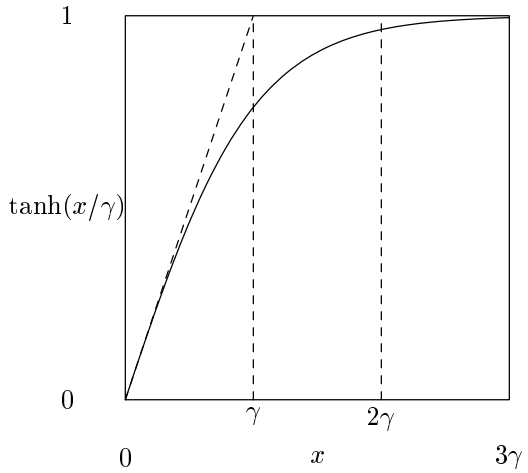


Figure 5.4: $\tanh(x/\gamma)$

when the speed is small. Parameter γ is a scalar which controls the shape of the $\tanh(\cdot)$ envelope. The velocity component of the distance metric is thus

$$\begin{aligned} \|\Delta\mathbf{v}\|^2_{(\text{vprj})} &= \|\mathbf{v}_a - \mathbf{v}_b\|^2_{(\text{vprj})} \\ &= p_2 \left\| \frac{\mathbf{v}_a}{\|\mathbf{v}_a\|} \tanh(\|\mathbf{v}_a\|/\gamma) + \frac{\mathbf{v}_b}{\|\mathbf{v}_b\|} \tanh(\|\mathbf{v}_b\|/\gamma) \right\|^2 . \end{aligned} \tag{5.10}$$

The equivalent mapping into the velocity subspace of the projection space is

$$\mathbf{v}^{(\text{prj})} = \mathbf{v} \left(\frac{\sqrt{p_2}\, \tanh(\|\mathbf{v}\|/\gamma)}{\|\mathbf{v}\|} \right) = \sqrt{p_2} \tanh(\|\mathbf{v}\|/\gamma)\hat{\mathbf{v}}, \tag{5.11}$$

where $\hat{\mathbf{v}}$ is the unit-vector $\frac{\mathbf{v}}{\|\mathbf{v}\|}$.

Let us look at the effect of our projection metric on three examples of drawing α-figures using the interface in Figure 5.1. These example performances were preprocessed with a conventional spline smoother, as demonstrated in Figure 5.2. The resulting data is plotted as a vector field in Figure 5.5. The most difficult part of these figures for the projection step to parameterize correctly is the region where the examples intersect themselves. Within this region, we want points to be parameterized to similar values only if their direction of motion is roughly the same. Thus, we need the velocity metric to outweigh the position metric for pairs of points within this region.

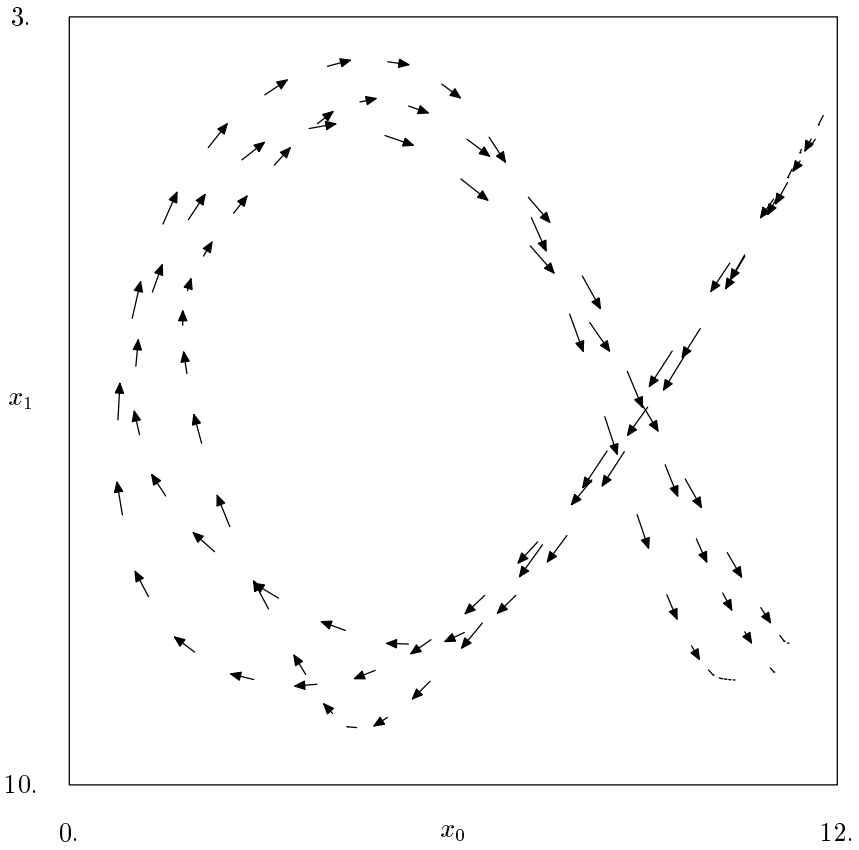


Figure 5.5: Samples from three examples of α-drawing. The arrows give position and velocity of the sampled points, with speed indicated by the length of the arrows. The x_1 axis is reversed to match the orientation of the coordinate system of the interface window.

At the end of the figure, the direction of motion tends to vary slightly as the user stops his motion and releases the mouse button to end the drawing stroke. We don't want the local variations in direction to count for much here. They contribute little to the resulting figure because the motion is slow. This is precisely what our projection metric is designed to to, and the results from running the principal curves algorithm on this data later in this chapter will show that our metric was able to generate suitable parameterizations for this data.

5.5 Conditional-expectation space

The projection step results in a parameterization value for each point $\mathbf{y}_i$ in the data-set. This parameterization $s(\mathbf{y})$ is a measure of distance along the estimated principal-curve. It is the time at which the projection of each point onto the principal-curve estimate will be reached by moving along the curve at unit speed, starting from one of the end-points at time $s = 0$. The conditional expectation step of the principal curves algorithm, discussed in Section 3.6.5, smoothes each dimension of the data-set against this parameterization. We will see that the projection-space used for computing this parameterization is incompatible with our smoother, and thus we will map the data to a separate space before performing conditional-expectation. Furthermore, we will actually need to smooth against a parameter which is a linear function of s rather than directly against s.

We will be using the smoother from Chapter 4, which balances the quality of fit in position space and velocity space against a measure of the total smoothness of the curve. The smoother minimizes this cost function:

$$S = p_1 \sum_{i=0}^{n-1} \|(\mathbf{x}_i - f(\lambda_i))\mathbf{D}_{\mathrm{p}i}^{-1}\|^2 + p_2 \sum_{i=0}^{n-1} \|(\mathbf{v}_i{}^{(\mathrm{ce})} - f'(\lambda_i))\mathbf{D}_{\mathrm{v}i}^{-1}\|^2 + (1 - p_1 - p_2) \int_{\min(\lambda_i)=0}^{\max(\lambda_i)=1} \|\mathbf{f}''(\lambda)\|^2 d\lambda \quad (5.12)$$

where $\mathbf{D}_{\mathrm{p}i}$ and $\mathbf{D}_{\mathrm{v}i}$ are diagonal weighting matrices with elements δx_{ji} and δv_{ji} respectively. Here we again see the weighting scalars p_1 and p_2 from (5.11) and (5.8). Minimizing cost function (5.12) will lead us to a cubic spline solution. Hastie and Stuetzle [28] recommend that if spline smoothers are used for finding principal curves, the parameterization over the data-set

should be scaled to fit the interval from 0 to 1 rather than using a unit-speed parameterization. This re-parameterization is necessary because unit-speed paths defined over an arbitrarily large interval can satisfy any smoothness criterion while visiting every point in the data-set. For this reason, points in conditional-expectation space will be parameterized by variable λ, where

$$\lambda = \frac{s}{s_{\max}}. \tag{5.13}$$

Value $s_{\max}$ is the maximum parameterization value of the data-points in projection-space

$$s_{\max} = \max_i(s(\mathbf{y}_i)). \tag{5.14}$$

We need to provide the smoothing algorithm with input points whose velocity dimensions are compatible with derivatives of the position with respect the smoothing parameter λ. This means that the velocity dimensions will be uniformly scaled versions of the directional derivative. First, we compute the derivative with respect to unit-speed parameterization s:

$$\begin{aligned} \mathbf{v}^{(\mathrm{s})}(s) \equiv \frac{d}{ds}\mathbf{x}^{(\mathrm{s})} &= \frac{dt}{ds}\frac{d}{dt}\mathbf{x}^{(\mathrm{s})}(t(s)) \\ &= \frac{1}{\|\mathbf{v}(t(s))\|}\frac{d}{dt}\mathbf{x}^{(\mathrm{t})}(t(s)), \end{aligned} \tag{5.15}$$

because $\frac{ds}{dt}$ is speed. Next, we re-parameterize with respect to λ, using the fact (5.13) that $\frac{ds}{d\lambda} = s_{\max}$:

$$\begin{aligned} \mathbf{v}^{(\lambda)}(\lambda) &= \frac{ds}{d\lambda}\frac{d}{ds}\mathbf{x}^{(\lambda)}(\lambda(s)) \\ &= \frac{ds}{d\lambda}\frac{dt}{ds}\frac{d}{dt}\mathbf{x}^{(\mathrm{t})}(t(\lambda)) \\ &= \frac{s_{\max}}{\|\mathbf{v}(t(\lambda))\|}\frac{d}{dt}\mathbf{x}^{(\mathrm{t})}(t(\lambda)), \end{aligned} \tag{5.16}$$

so

$$\mathbf{v}_i^{(\lambda)} = \mathbf{v}_i^{(\mathrm{t})}\frac{s_{\max}}{\|\mathbf{v}_i^{(\mathrm{t})}\|} = s_{\max}\hat{\mathbf{v}}_i. \tag{5.17}$$

Thus, we convert the input data to *conditional-expectation space*

$$\mathbf{Y}^{(\mathrm{ce})} = \begin{bmatrix} \mathbf{X} \\ \mathbf{V}^{(\lambda)} \end{bmatrix} \tag{5.18}$$

before we call the smoother. Value $s_{\max}$ is the length of the current principal curve estimate, and changes with each iteration of the principal curves algorithm, but it is the only component of $\mathbf{v}_i^{(\lambda)}$ in (5.17) that is not known upon first reading the training data. Thus, we can normalize each velocity vector at the beginning of the principal-curves algorithm, and then simply scale this normalized velocity by $s_{\max}$ (provided by the projection step) before each call to the smoother.

It might at first seem that we could avoid re-scaling the velocity values in conditional-expectation space between each iteration of the principal curves algorithm by instead adjusting the ratio of the parameters p_1 and p_2 in the smoother's cost function. Unfortunately, this will not work. The local form of curve $\mathbf{f}$ at any point is a cubic polynomial $\mathbf{P}_i$, which may be written in Newton form as

$$\mathbf{P}_i(s) = \mathbf{c}_{0i} + \mathbf{c}_{1i}(s - s_i) + \mathbf{c}_{2i}(s - s_i)^2 + \mathbf{c}_{3i}(s - s_i)^3. \tag{5.19}$$

Thus, the squared magnitude of the local smoothness penalty is

$$\|\mathbf{P}_i''(s)\|^2 = \sum_{j=0}^{m-1} (2c_{2ij} + 6c_{3ij}(s - s_i))^2. \tag{5.20}$$

Because this equation is not linear in s and has a non-zero value when $s = s_i$, we cannot account for the effects of smoothing against $\lambda = s/s_{\max}$ by adjusting the ratios between weighting parameters p_1, p_2, and $(1 - p_1 - p_2)$. Rescaling the parameterization does not affect the smoothness penalty proportionally.

5.6 Consistency between projection metric and smoothing penalty

A principal curve is a self-consistent curve through a given distribution. In the case of trajectory smoothing in phase space, this means that the projection metric should be equivalent to the sum of position and velocity error terms in (5.12). We can assure this consistency during the smoothing stage by determining the proper weighting matrices $\mathbf{D}_{\mathrm{p}i}$ and $\mathbf{D}_{\mathrm{v}i}$. A quick comparison between the position cost term in (5.12) and the position component of

the distance metric (5.7) shows that they are already equivalent. Therefore, we set

$$\delta x_{ji} = 1 \qquad \forall j, i \tag{5.21}$$

so that $\mathbf{D}_{\mathrm{p}i} \equiv \mathbf{D}_{\mathrm{p}} = \mathbf{I}_{(m\times m)}$ is the identity matrix of rank m.

For the velocity dimensions, however, we need to determine the weight δv_{ji} by which to scale each velocity error in conditional-expectation space. We have already required in (5.6) that error magnitudes in each velocity dimension should be considered equivalent in importance, so we weigh them equally:

$$\begin{aligned} \delta v_{ji} &= \delta v_i, \\ \mathbf{D}_{\mathrm{v}i} &= (\delta v_i)\mathbf{I}_{(m\times m)}. \end{aligned} \tag{5.22}$$

To ensure that the principal curve is self consistent, we set the velocity component of the distance metric equal to the corresponding cost in conditional-expectation space: $\|\Delta\mathbf{v}\|^2_{(\mathrm{vprj})} = S_{\Delta\mathbf{v}_i^{(\mathrm{ce})}}$. This expands to

$$\begin{aligned} p_2 \left\| \frac{\mathbf{v}_i}{\|\mathbf{v}_i\|} \tanh(\|\mathbf{v}_i\|/\gamma) - \frac{\frac{d}{dt}\mathbf{f}(\lambda_i)}{\|\frac{d}{dt}\mathbf{f}(\lambda_i)\|} \tanh(\|\tfrac{d}{dt}\mathbf{f}(\lambda_i)\|/\gamma) \right\|^2 \\ = p_2 \sum_{j=0}^{m-1} \left(\frac{{v_{ji}}^{(\lambda)} - \mathbf{f}'(\lambda_i)}{\delta v_i} \right)^2. \end{aligned} \tag{5.23}$$

We haven't considered the time-derivative of the model function $\mathbf{f}$ yet, since we have been fitting the curve against the λ parameterization rather than time. This value, $\frac{d}{dt}\mathbf{f}$, is the velocity of the best-fit trajectory, and $\|\frac{d}{dt}\mathbf{f}\|$ is the speed. Section 5.12 will deal with the estimation of this value, but for the purposes of computing δv_i, we will make the approximation that the speed of the model at λ_i is the same as that of the example point: $\|\frac{d}{dt}\mathbf{f}(\lambda_i)\| \approx \|\mathbf{v}_i\|$. Fortunately, since we we do not need to know δv_i to any great precision for smoothing, this approximation is good enough. This gives us:

$$\begin{aligned} p_2 \left(\frac{\tanh(\|\mathbf{v}_i\|/\gamma)}{\|\mathbf{v}_i\|} \right)^2 \sum_{j=0}^{m-1} (v_{ji} - \tfrac{d}{dt} f_j(\lambda_i))^2 \approx \frac{p_2}{(\delta v_i)^2} \sum_{j=0}^{m-1} \left({v_{ji}}^{(\lambda)} - \tfrac{d}{d\lambda} f_j(\lambda_i) \right)^2 \\ \approx \frac{p_2}{(\delta v_i)^2} \sum_{j=0}^{m-1} \left(\frac{ds}{d\lambda}\frac{dt}{ds} v_{ji} - \frac{ds}{d\lambda}\frac{dt}{ds}\frac{d}{dt} f_j(\lambda_i) \right)^2, \end{aligned} \tag{5.24}$$

and then

$$p_2 \left(\frac{\tanh(\|\mathbf{v}_i\|/\gamma)}{\|\mathbf{v}_i\|} \right)^2 \sum_{j=0}^{m-1} (v_{ji} - \tfrac{d}{dt} f_j(\lambda_i))^2$$

$$\approx \frac{p_2}{(\delta v_i)^2} \left(\frac{s_{\max}}{\|\mathbf{v}_i\|} \right)^2 \sum_{j=0}^{m-1} \left(v_{ji} - \tfrac{d}{dt} f_j(\lambda_i) \right)^2, \quad (5.25)$$

which can be easily solved for the weight value

$$\frac{1}{\delta v_i} \approx \frac{\tanh(\|\mathbf{v}_i\|/\gamma)}{s_{\max}}. \quad (5.26)$$

Note that the numerator of (5.26) can be computed during the initialization step of the principal curves algorithm, but the denominator changes between each call to the smoother.

5.7 Selection of scalar weights p_1, p_2, and γ

Parameter γ Parameter γ controls the envelope by which we limit the effect of speed on the projection-space distance metric (5.11). From Figure 5.4 it is evident that we should select γ to be a threshold beyond which we don't care *much* about the actual speed of the action, and beyond 2γ we shouldn't care about additional speed at all. Direction and speed should both be important below $\gamma/2$. Of course, a different function may be chosen to envelope the effect of speed if these rules of thumb do not make sense for characterizing a given action.

Parameters p_1 and p_2 If we are using the $\tanh(\cdot)$ envelope for velocity, then we can select the ratio of p_1 to p_2 by selecting a value for the difference in direction (in radians) between two velocities $\mathbf{v}_1$ and $\mathbf{v}_2$, both at full-speed (i.e., $\|\mathbf{v}_1\| \gg 2\gamma$, $\|\mathbf{v}_2\| \gg 2\gamma$), such that projection metric for their difference is equivalent to that for a pair of points whose positions are one unit apart:

$$\|\Delta \mathbf{x}_{\text{unit}}\|^2_{(\text{pprj})} = \|\mathbf{v}_1 - \mathbf{v}_2\|^2_{(\text{vprj})}$$

$$p_1 = \left\| \frac{\sqrt{p_2}\, \tanh(\|\mathbf{v}_1\|/\gamma)}{\|\mathbf{v}_1\|} \mathbf{v}_1 - \frac{\sqrt{p_2}\, \tanh(\|\mathbf{v}_2\|/\gamma)}{\|\mathbf{v}_2\|} \mathbf{v}_2 \right\|^2. \quad (5.27)$$

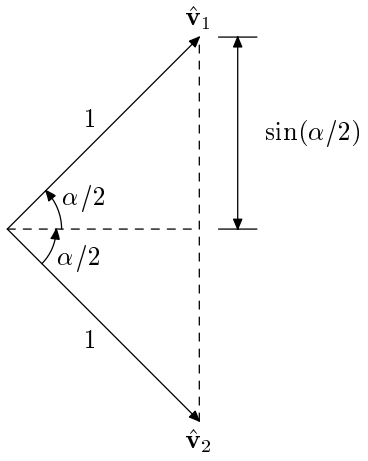


Figure 5.6: Distance between unit vectors $\hat{\mathbf{v}}_1 = \frac{\mathbf{v}_1}{\|\mathbf{v}_1\|}$ and $\hat{\mathbf{v}}_2 = \frac{\mathbf{v}_2}{\|\mathbf{v}_2\|}$ is $2\sin(\alpha/2)$

Because the speed of both points is high the $\tanh(\cdot)$ expressions asymptotically approach 1, which gives us

$$p_1 \approx p_2 \left\| \frac{\mathbf{v}_1}{\|\mathbf{v}_1\|} - \frac{\mathbf{v}_2}{\|\mathbf{v}_2\|} \right\|^2 = \|\hat{\mathbf{v}}_1 - \hat{\mathbf{v}}_2\|^2. \tag{5.28}$$

The right-hand-side of (5.28) contains a difference of unit-vectors. By simple geometry, shown in Figure 5.6, we can express the magnitude of this vector-difference in terms of the angle α between the two vectors, allowing us to rewrite (5.28) as

$$p_1 \approx p_2 (2\sin(\alpha/2))^2, \tag{5.29}$$

which immediately gives us an expression for the ratio of p_1 and p_2

$$\frac{p_1}{p_2} \approx 4\sin^2(\alpha/2). \tag{5.30}$$

Thus, given difference in velocity-direction α (at high speed) whose distance metric in projection space we consider equal to a unit position-distance, the necessary ratio of parameters p_1 and p_2 is given by (5.30). Once we know this ratio, then choosing the actual values of p_1 and p_2 is an exercise in balancing smoothness of the model-trajectory against approximation-error

of the trajectory in the cost function (5.12) for conditional-expectation. This might be accomplished using cross-validation, as discussed in Section 4.10.

We can also solve for the ratio p_1/p_2 for which a given angle of motion at high speed is equivalent to a given difference in position. From the argument above, we can write

$$\begin{aligned} p_1\|\Delta\mathbf{x}\|^2 &\approx p_2(2\sin(\alpha/2))^2 \\ \frac{p_1}{p_2} &\approx \left(\frac{2\sin(\alpha/2)}{\|\Delta\mathbf{x}\|}\right)^2 \end{aligned} \tag{5.31}$$

Solving for $\alpha = \frac{\pi}{2}$, we get the ratio of weights such that a right angle difference is equivalent to a given $\|\Delta\mathbf{x}\|^2$:

$$\frac{p_1}{p_2} \approx \frac{2}{\|\Delta\mathbf{x}\|^2} \qquad (\text{for } \alpha = \tfrac{\pi}{2}). \tag{5.32}$$

Solving for $\alpha = \pi$, we get

$$\frac{p_1}{p_2} \approx \frac{4}{\|\Delta\mathbf{x}\|^2} \qquad (\text{for } \alpha = \pi). \tag{5.33}$$

To make sure points in the region of intersection of Figure 5.5 are correctly parameterized, we might choose a difference in position of 2 or 3 units to have an equivalent metric value to a right angle difference in velocity direction. By (5.32), this implies that we want a ratio in the range $\frac{2}{9} \le p_1/p_2 \le \frac{1}{2}$. The results in Figures 5.8 and 5.9 were made using almost equal values for p_1 and p_2, which corresponds to a more daring right-angle equivalent difference of $\|\Delta\mathbf{x}\| = \sqrt{2} \approx 1.4$. Note that this did result in misclassifications in the first iteration, but also that later iterations correct them.

5.8 Initial principal curve estimate

In applications of the principal curves algorithm for the purpose of non-linear regression, the principal curve estimate is often initialized using the first principal component of the data-set. Figure 3.6(a) on page 57 is an example. This is appropriate because it is computationally inexpensive, and more importantly, because the first linear principal component is the linear analogue of the principal curve—if we restrict the principal curves algorithm to use straight lines, the result must converge to the first principal component [28].

When using principal curves for trajectory smoothing rather than for regression, however, the first principal component is not necessarily a good initial guess. Instead of a mere regression fit, our principal-curve estimate for a trajectory is a directed path through phase space. Not only does this path have a direction, a beginning and an ending point, but there is a close local relationship between the position and velocity dimensions of the points along the curve. In valid trajectory estimates in the conditional-expectation space, the velocity dimensions are the directional derivatives of the position variables, scaled by $1/s_{\max}$ as indicated in equation (5.17). If we initialize the principal-curve estimate using the first principal component of the training data, we must either decide to find the principal component in position space (choosing one of the two possible directional orientations), and then choose a compatible path in velocity space, or we must determine a principal component in both position and velocity space simultaneously, in which case the position and velocity dimensions of the resulting line may not be compatible with one-another.

Figure 5.7 shows how data points from the α-drawing examples and also some β-drawings project onto the principal component of the position data. If we consider the principal component to be a directed path progressing either leftward or rightward in time, then roughly half the data points will project onto it with incorrect orientations.

In our case, fortunately, we start with a number of example trajectories. If these examples are similar enough that our principal curves-based method should work, then each of these individual examples is probably a fairly good choice as an initial best-fit trajectory estimate. The next section describes the procedure for initializing the principal-curve algorithm with a given training example.

5.9 Summary of trajectory-smoothing algorithm

In this section, we present the full algorithm for trajectory-smoothing using principal curves in phase space. The skeletal form of the algorithm is that of the traditional principal curves algorithm for datasets discussed in Sections 3.6.4-3.6.5, but this algorithm has been adjusted in the manner discussed in the previous sections so that (a) distances are weighted properly

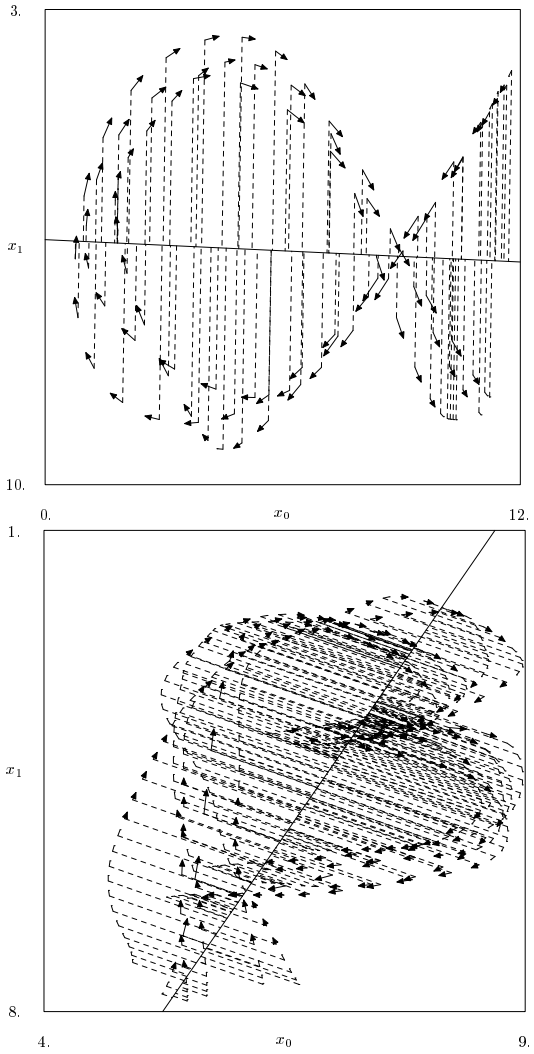


Figure 5.7: Samples from three examples of α-drawing and five examples of β-drawing, where the data has been projected onto the first principal component of the position sub-space. Note that in each case, if the principal component is considered to be a directed path, then roughly half the points will project onto it in with the wrong orientation.

in projection space (Section 5.4), (b) the velocity dimensions of the data are computed appropriately for use with the smoother in conditional-expectation space (Section 5.5), and (c) the weighting matrices for the smoother cause the penalty for errors in relative position and velocity of the smoothed curve to match the error-distance metric as measured in projection space (Section 5.6).

1. INITIALIZATION: $(\mathbf{Y}, \mathbf{Y}_\mathrm{e}) \rightarrow (\mathbf{Y}_0^{(\mathrm{prj})} \equiv \mathbf{Y}^{(\mathrm{prj})}, \boldsymbol{\lambda}_0, s_{(0)\max})$

   (a) Let $\mathbf{Y}_\mathrm{e}$ be a set of data-points from a given training example, where the column-vectors are the individual points stored in order of their sampling-time. Map these data-points into projection-space by equation (5.4) to get $\mathbf{Y}_\mathrm{e}^{(\mathrm{prj})}$, using (5.11) for the velocity dimensions and (5.8) for the position dimensions.

   (b) Compute the projection-space version of the training data, $\mathbf{Y}_0^{(\mathrm{prj})} \equiv \mathbf{Y}^{(\mathrm{prj})}$, by applying (5.11), (5.8), and (5.4) to $\mathbf{Y}$.

   (c) Specify path $\boldsymbol{\rho}_{(0)}$ consisting of line-segments connecting consecutive points which are the column-vectors of $\mathbf{Y}_\mathrm{e}^{(\mathrm{prj})}$ (i.e., the first line-segment in the path has end-points $(\mathbf{y}_{\mathrm{e}0}^{(\mathrm{prj})}, \mathbf{y}_{\mathrm{e}1}^{(\mathrm{prj})})$, the second has end-points $(\mathbf{y}_{\mathrm{e}1}^{(\mathrm{prj})}, \mathbf{y}_{\mathrm{e}2}^{(\mathrm{prj})})$, etcetera). Project each point from the projection-space version of the training data-set $\mathbf{Y}^{(\mathrm{prj})}$ to the nearest point on $\boldsymbol{\rho}_{(0)}$. Assign to each point $\mathbf{y}_i^{(\mathrm{prj})}$ in the training data-set a value $s_{(0)i} = d(\boldsymbol{\rho}_{(0)}, \mathbf{y}_i^{(\mathrm{prj})})$ that is the distance of its projection onto path $\boldsymbol{\rho}_{(0)}$ along that path starting from end-point $\mathbf{y}_{\mathrm{e}0}^{(\mathrm{prj})}$. Compute vector $\boldsymbol{\lambda}_0$ with $\lambda_{(0)i} = s_{(0)i}/s_{(0)\max}$ for each point, where $s_{(0)\max} = \max_i(s_{(0)i})$.

2. Repeat, over iteration counter $j \in (1, 2, \ldots)$:

   (a) CONDITIONAL EXPECTATION: $(s_{(j-1)\max}, \mathbf{Y}, \mathbf{Y}^{(\mathrm{prj})}, \boldsymbol{\lambda}_{(j-1)}) \rightarrow (\boldsymbol{\rho}_{(j)})$ Use the value of $s_{(j-1)\max}$ in (5.17) and (5.18) to map the training-data to conditional-expectation space points $\mathbf{Y}_{(j-1)}^{(\mathrm{ce})}$. Compute $\mathbf{D}_\mathrm{v}$ from (5.26) using $s_{(j-1)\max}$, then use the smoother from Chapter 4 to compute $\mathbf{f}_{(j)}^{(\mathrm{ce})}(\lambda_{(j-1),i})$, which is the computed expected value of each point $\mathbf{y}_{(j-1),i}^{(\mathrm{ce})}$ given parameterization value $\lambda_{(j-1),i}$. By multiplying the velocity dimensions of $\mathbf{f}_{(j)}^{(\mathrm{ce})}(\lambda_{(j-1),i})$ by $\|\mathbf{v}_i\|/s_{(j-1)\max}$,

we compute $\mathbf{Y}_{(j)} : \mathbf{y}_{(j),i} \leftarrow \mathbf{f}_{(j)}(\lambda_{(j-1),i})$. Let $\text{px}_{(j)}$ be a permutation which sorts $\boldsymbol{\lambda}_{(j-1)}$ into order of ascending magnitudes. Path $\boldsymbol{\rho}_{(j)}$, the j-th estimate of the best-fit trajectory, is the set of line segments that connect consecutive column-vectors of $\mathbf{y}_{(j),\text{px}_{(j)i}}$, which is $\mathbf{Y}_{(j)}$ whose column vectors are sorted into order $\text{px}_{(j)}$.

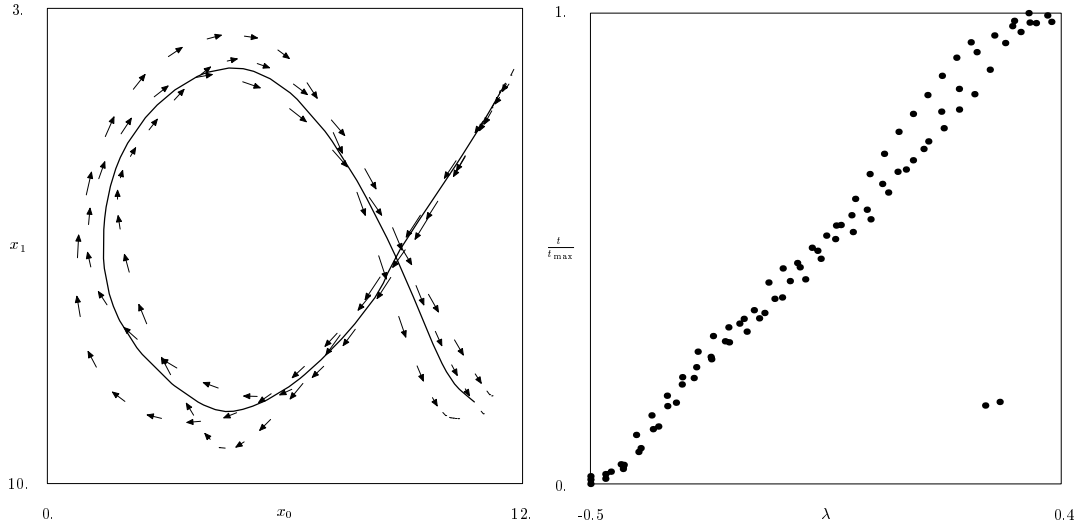
(b) PROJECTION: $(\mathbf{Y}^{(\text{prj})}, \boldsymbol{\rho}_{(j)}) \rightarrow (\boldsymbol{\lambda}_{(j)})$
Compute $\boldsymbol{\rho}_{(j)}^{(\text{prj})}$ by transforming the end-points of the line segments in path $\boldsymbol{\rho}_{(j)}$ using (5.11), (5.8), and (5.4). Project each point from the projection-space version of the training data-set $\mathbf{Y}^{(\text{prj})}$ to the nearest point on $\boldsymbol{\rho}_{(j)}^{(\text{prj})}$. Assign to each point $\mathbf{y}_i^{(\text{prj})}$ in the training data-set a value $s_{(j)i} = d(\boldsymbol{\rho}_{(j)}^{(\text{prj})}, \mathbf{y}_i^{(\text{prj})})$ that is the distance of its projection onto path $\boldsymbol{\rho}_{(j)}^{(\text{prj})}$ along that path starting from end-point $\mathbf{y}_{(j),0}^{(\text{prj})}$. Compute parameterization vector $\boldsymbol{\lambda}_{(j)}$ with $\lambda_{(j),i} = s_{(j),i}/s_{(j)\text{max}}$ for each point, where $s_{(j)\text{max}} = \max_i(s_{(j),i})$.

(c) ERROR EVALUATION: $(\mathbf{Y}^{(\text{prj})}, \mathbf{Y}_{(j)}) \rightarrow (d_j)$
Calculate $d_j = \|\mathbf{Y}^{(\text{prj})} - \mathbf{Y}_{(j)}^{(\text{prj})}\|$.

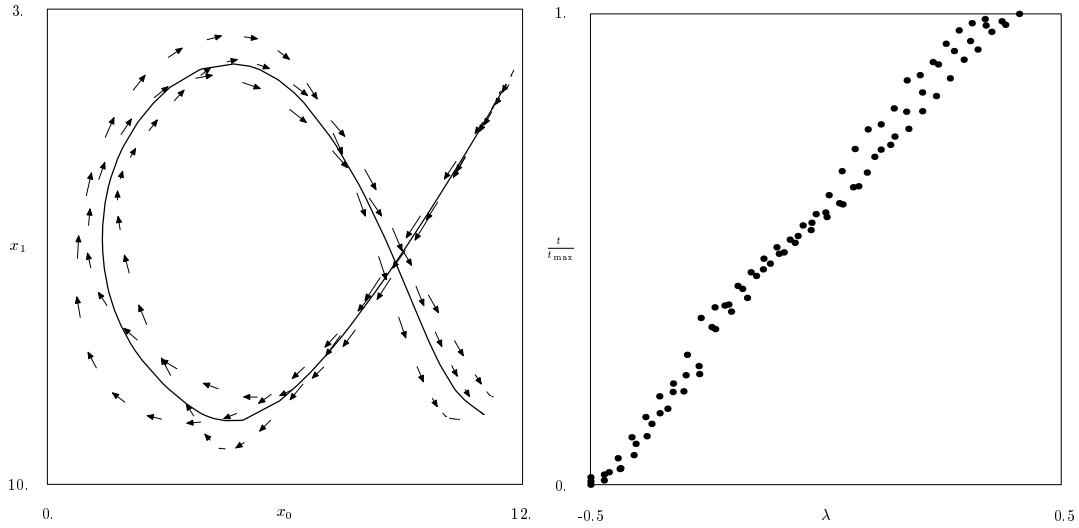
Until the change $(d_{j-1} - d_j)$ is below some threshold.

5.10 Results from α-example

The results from running the algorithm presented in the previous section are shown in Figures 5.8 and 5.9. Using principal curves in phase space, we are actually able to learn a best-fit trajectory which intersects itself in position space. In the left-hand plot of Figure 5.8(a) where the path crosses itself, we actually see that the two points from the beginning of the figure are misclassified as belonging to the later part of the figure. The plot relating the percentage of completion time for each point to its λ value, in the right-hand plot, clearly shows these two misclassified points. Figure 5.8(b) shows the result from the next iteration of the algorithm. The first iteration of the algorithm has improved the principal curve estimate enough that all the points are correctly parameterized by this run of the projection step. The estimated best-fit trajectory is fairly good. Figure 5.9 shows the result from the third iteration, which we can see has converged to a stable, self-consistent curve.



(a) Result of iteration 1: 2D plot and time-percent vs. λ.



(b) Result of iteration 2: 2D plot and time-percent vs. λ.

Figure 5.8: Results from first two iterations of principal curves algorithm on α-drawing skill.

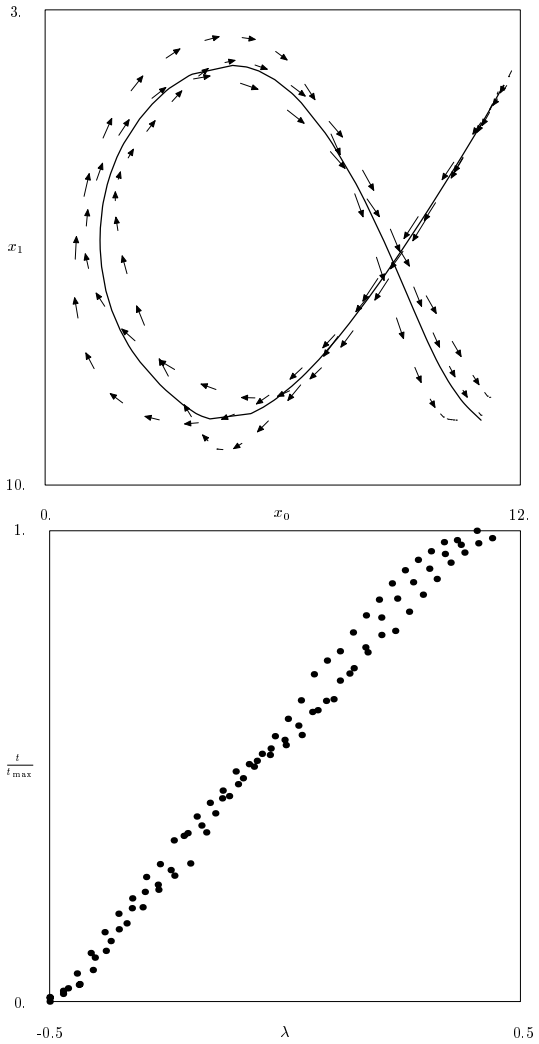
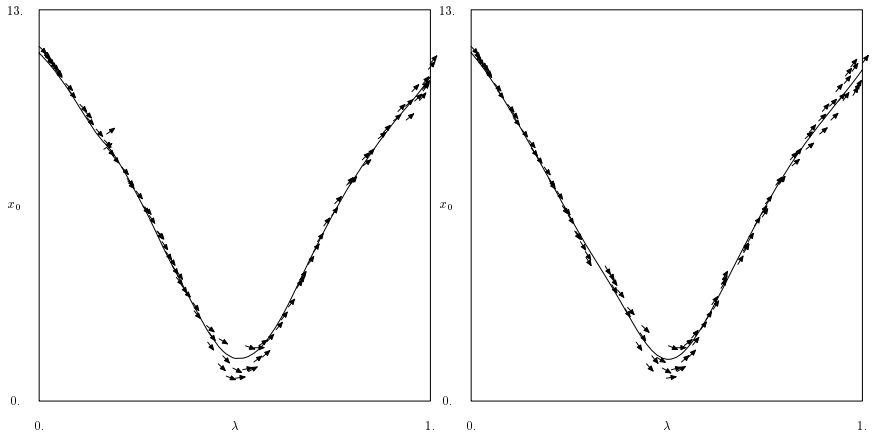
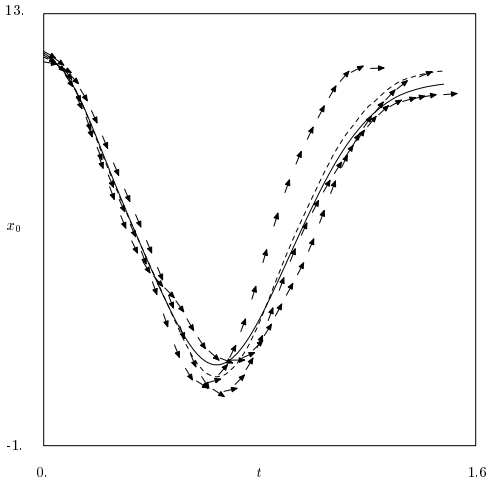


Figure 5.9: Results from third iteration of principal curves algorithm on α-drawing skill.

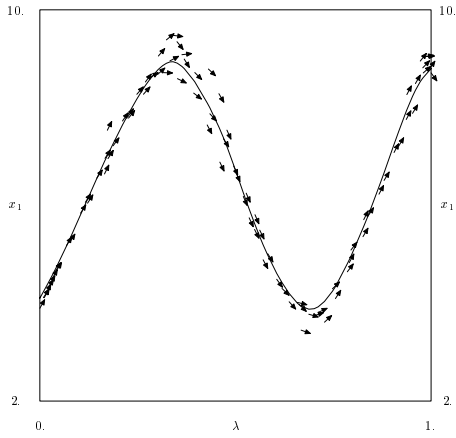


(a) Smoothing PC iteration 0 (b) Smoothing PC iteration 2

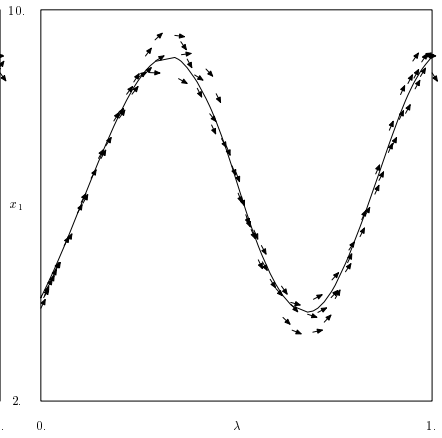


(c) Smoothing verses time parameterization. Solid curve uses variance estimation, dashed doesn't.

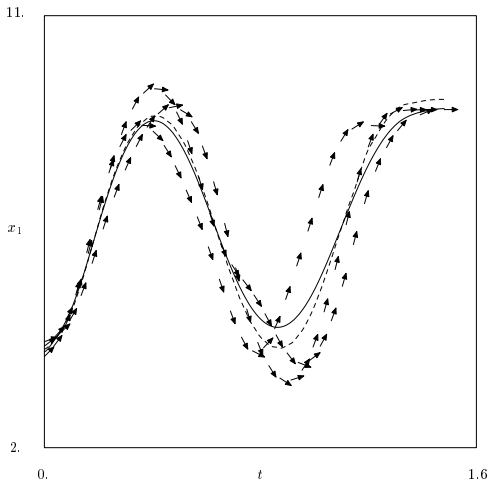
Figure 5.10: Comparison of smoothing x_0 variable of α-drawing data against λ in two iterations of principal curves algorithm verses smoothing against a time parameterization.



(a) Smoothing PC iteration 0



(b) Smoothing PC iteration 2



(c) Smoothing verses time parameterization. Solid curve uses variance estimation, dashed doesn't.

Figure 5.11: Comparison of smoothing x_1 variable of α-drawing data against λ in two iterations of principal curves algorithm verses smoothing against a time parameterization.

Figures 5.10 and 5.11 show the smoothing operations of the two position dimensions of the data against the parameterization generated by iterations 0 and 2 of the principal curves algorithm. In addition, plots 5.10(c) and 5.11(c) show smoothing against a time parameterization. It is readily apparent from these figures that the λ-parameterization is a much better parameterization than time, as the λ-parameterized plots have significant error only when the slope of the fit in a given dimension is changing sign. Also, note the difference between the λ plots and time plots at their ends: it is obvious that the motion is slow at the ends of the time plots, whereas this is not obvious in the λ plots because λ is a constant-speed parameterization.

These plots clearly show that the parameterization step of the principal curves algorithm has had the intended result in this case: the λ-parameterization gives a much more consistent set of data to smooth, and thus makes the job of the smoother relatively easy. One minor point to note is that in the left-most quarter of plot 5.10(a) we see the two misclassified points pointing upward rather than downward. The smoothing operation is not especially effected by them, and the improved curve estimate allows the points to be properly parameterized in later iterations.

Another data set for which principal curves in phase space can generate a high-quality best-fit trajectory where conventional principal curves cannot work well is shown in Figure 5.12. In addition to the basic drawing motion intersecting itself at the end of the figure, the reversal of direction between the two humps of the β-figure is extremely sharp in position space, and is bounded by motions which have opposing velocities but roughly coincide in position space. The four iterations it took the principal curves algorithm to converge to a stable solution are shown in Figure 5.13. The final result in Figure 5.13(d) is much a better figure than are any of the individual examples, although we might like the beginning of the trajectory to be better aligned with the rest of the initial vertical stroke.

The reversal of direction between the humps of the β is the most interesting part of the figure to watch develop over the iterations of the algorithm. It starts as a shallow indentation, and then gets drawn in towards the sharpest data in the examples. The reason that the figure created by a smoother can have such a sharp feature is that the feature is actually far less sharp in phase space; the figure shown is merely the projection of the 4-dimensional phase space onto the 2-dimensional position space. Although the position space projection looks like a sharp point, the points on either side of the points are more distant in velocity space, and thus their λ values are relatively distinct.

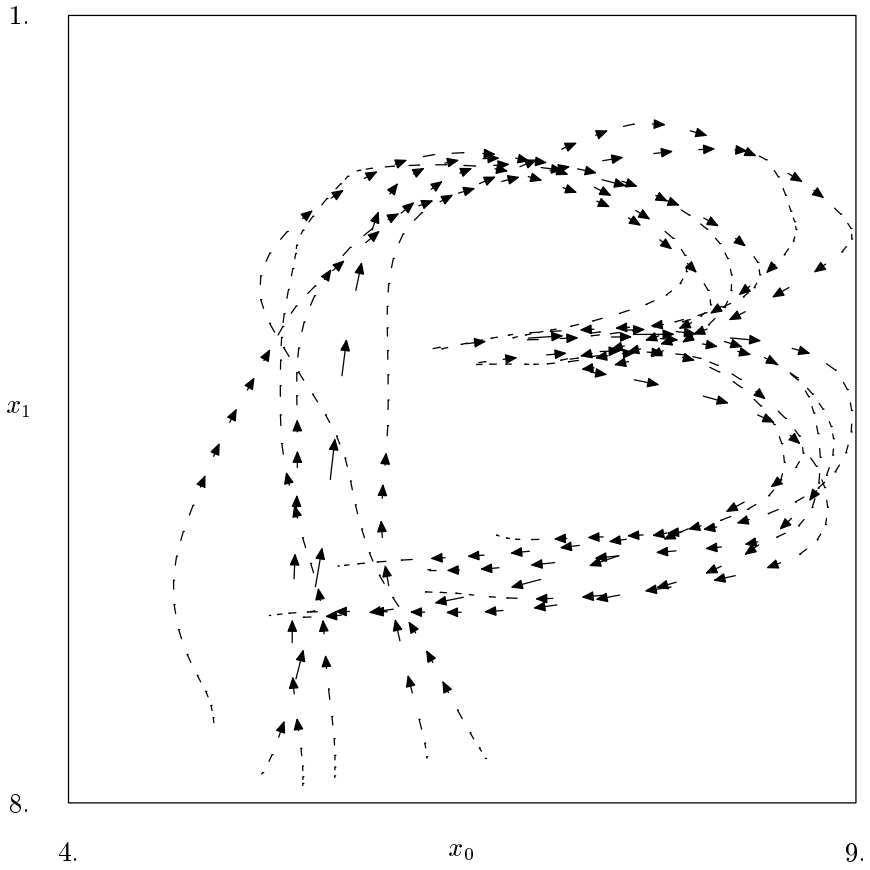
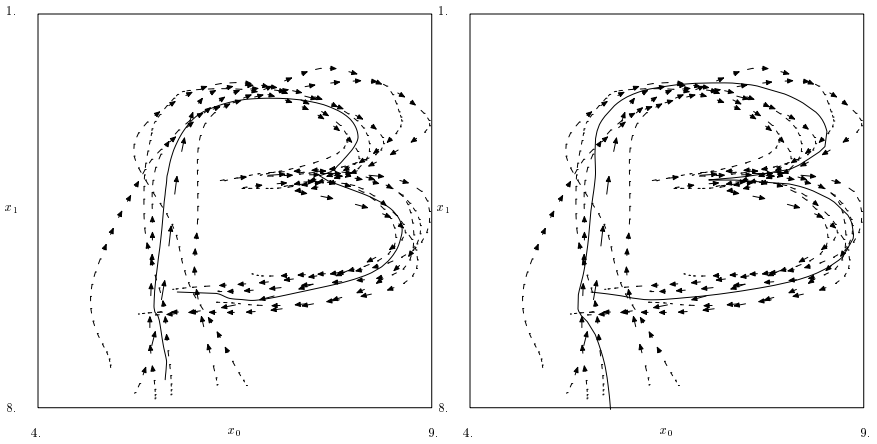


Figure 5.12: Smoothed input data from β-drawing examples



(a) Iteration 0.

(b) Iteration 1.

(c) Iteration 2.

(d) Iteration 3.

Figure 5.13: Principal curves iteration results for β example

5.11 Diagnostics

In the analysis of the α-drawing example, plots of $t/t_{\max}$ in Figures 5.8 and 5.9 were used to indicate the success of the projection step for parameterizing the data. A related way to judge the quality of the parameterization step is to plot the λ value of each point in a given training example verses its sampling time. For good parameterizations, the results should generally be a monotonic relationship. Figure 6.8 on page 132 gives examples of this kind of plot for data from a telerobotics experiment.

It is often more useful to have a numerical measure of the quality of the parameterization rather than a large number of graphs to study. One possible measure is

$$M_{\mathrm{p}} = \frac{\sum_{k=0}^{n_e-1}\sum_{i=0}^{n_k-2}\mathrm{sgn}(\Delta\lambda_{ki}(n_k-1))}{(n-n_e)}, \qquad \mathrm{sgn}(x) = \begin{cases} 1 & (x > \epsilon) \\ 0 & (|x| < \epsilon) \\ -1 & (x < -\epsilon) \end{cases} \tag{5.34}$$

where n_e is the number of training examples, n_k is the number of points in the k-th training example, and $\Delta\lambda_{ki} = (\lambda_{k,i+1} - \lambda_{k,i})$, where $\lambda_{k,i}$ is the parameterization of the i-th point in the k-th example. The value of ϵ should be a value indicating close to "no change" in λ, possibly $1/(10(n_e-1))$. An alternative measure, which is differentiable, is

$$M_{\mathrm{p1}} = \frac{\sum_{k=0}^{n_e-1}\sum_{i=0}^{n_k-2}\tanh(3\Delta\lambda_{ki}(n_k-1))}{(n-n_e)}. \tag{5.35}$$

Both M_{p} and M_{p1} approach 1 for good parameterizations, approach -1 with parameterizations which are exactly backwards, and tend toward 0 for parameterizations which are far from monotonic.

Principal curve models can be used to perform gesture-recognition in a manner similar to that for the smoother in Section 3.4. However, not only does the probability of each point need to be evaluated given its fitted parameterization value, but the overall parameterization of the example to be classified must be evaluated as well. If, after parameterizing an example against a given principal curve model, the resulting parameterization has a low value of M_{p} or M_{p1}, then the example should be identified as not belonging to the class of actions corresponding to the model even before equation (3.9) is applied. For example, if we do not evaluate the quality of

Figure		M_{p}	M_{p1}
α-drawing	(5.9, pg. 108)	0.96	0.96
β-drawing	(5.13(d), pg. 113)	0.68	0.71
Pre-grasp	(6.6, pg. 130)	0.76	0.79
Part transfer	(6.9, pg. 134)	0.98	0.93
Loading spring box	(6.11, pg. 137)	0.75	0.72

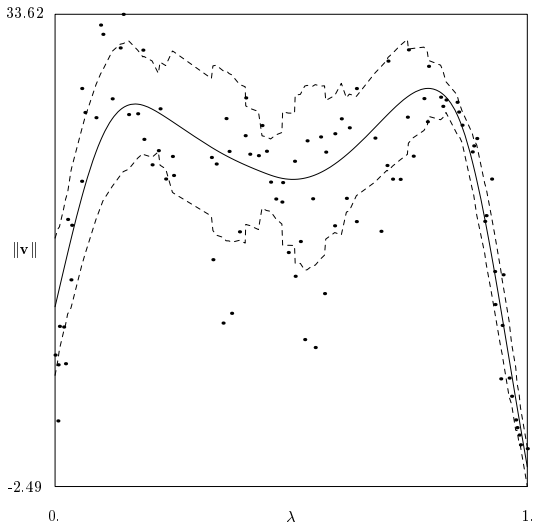
Table 5.1: Summary of parameterization diagnostics. Computation of M_{p} uses $\epsilon = 0.1$.

the parameterization, then a set of 10 points all lying exactly on the model of the α-drawing would measure as an example of an α-drawing motion with probability 1, even if these points all lie on only the first tenth of the model trajectory ($\lambda_i < 0.1$).

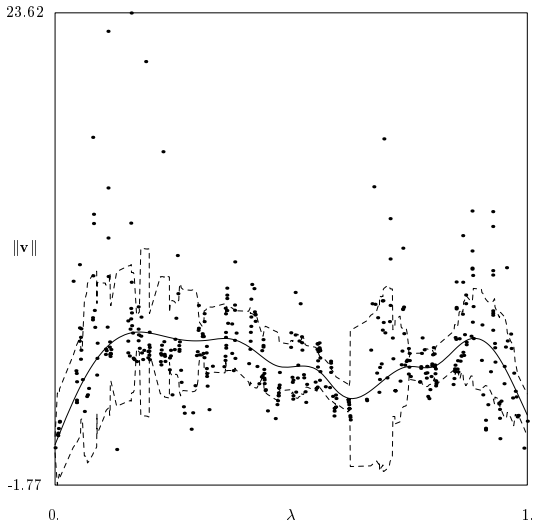
Table 5.1 summarizes the diagnostic values for the parameterizations of the best-fit trajectories from the α-drawing and β-drawing data in this chapter, as well as the final fits of the experimental data presented in Chapter 6. We see that the diagnostic values M_{p} and M_{p1} for the computed fits are fairly consistent, but they not interchangeable. Although the measures for the fits of the α-drawing and the part transfer data sets are significantly better than those of the other three, the difference is due more to how closely the trials in each data set resemble one another rather than the relative validity of the fits. All the resulting measures should simply be considered high enough to indicate a successful fit. The diagnostic measures presented in this section are a useful tool for evaluating the quality of the fits of the data, but a more principled derivation of such a measure is an important issue for further research.

5.12 Speed of performances

The conditional-expectation step uses a constant-speed representation because the velocity dimensions are a scaled unit-vector (5.17). Thus, the speed of the best-fit trajectory as a function of λ cannot be directly determined from the principal curve. However, we can easily estimate values for the speed of the best-fit trajectory by smoothing the speeds of the training data against the λ-parameterization. Figure 5.14 shows the resulting speed



(a) α-drawing



(b) β-drawing

Figure 5.14: Smoothed plots of speed verses λ for α-drawing and β-drawing figures.

parameterizations from the α-drawing and β-drawing examples, as well as the standard-deviation of the estimates. Since the spline smoother minimizes the second derivative of the speed estimates, the resulting trajectory motions minimize jerk.

5.13 Problems with the principal curve model

There are two main problems with the use of principal curves as an action model as presented in this chapter and Chapter 3. The first is that the theory for extending the representation to more than one dimension is not yet fully developed. There is no reason why this cannot be done however. A general approach to extending the dimensionality is described in Section 3.7 and will be further discussed in the future research section at the end of the thesis.

The other main problem is the size of the model. As derived, the number of knots in the spline composing the trajectory model is the same as the number of training points. This is computationally awkward, and clearly excessive given the smoothness of the resulting curves. Future work needs to be done to develop methods for eliminating knots in the model which do not significantly inflect the model path. Conventional spline smoothers can be defined which fit data to models with reduced numbers of knots [16], and it should be possible to derive such a smoother for phase space as well.

5.14 Discussion

This chapter presents a method for fitting a trajectory to a given set of phase space data. Such a best-fit trajectory is arguably the best one-dimensional model of a given action, as the fits of the α-drawing and β-drawing skills suggest, and as will be demonstrated for various actions skills in the robot teleoperation experiment analyzed in the next chapter. Some actions are more amenable to this kind of modeling than others, however. The suitability of an action for modeling with a best-fit trajectory is determined by how well the training data satisfies the assumptions listed at the beginning of this chapter. In addition to characterizing which actions can be modeled in this manner, it is also important to answer the questions of whether the results from this method can be considered optimal in some sense, and how the

results from this method might be compared to those from other methods, possibly yet to be developed.

Suitability of an action for modeling with a trajectory in phase space The assumptions listed at the beginning of the chapter specify that the action consists of a single smooth, consistent motion where the state of the action is adequately described by the phase space vector, and where the stochastic variability between different performances can be adequately described by a Gaussian distribution at any given point along the basic motion. Roughly, this means that although the example performances may vary, the overall structure of the motion is consistent. If a data point from one example motion has nearly the same position and velocity of as a point from another example, they should correspond to roughly the same part of the basic action, and the average of these points should be considered a good estimate of the expected value of the state during that part of the action. What the principal curve method learns, then, is basically an averaged trajectory.

While the α-drawing and β-drawing data sets work well with this method, they would thus not work if the different example figures were drawn with significantly different sizes, or especially if the example trajectories were rotated more than a few degrees with respect to one another. In the same way, while the best-fit trajectory is suitable for modeling a grasping motion, a single walking stride, a tennis swing, or a consistent pick-and-place motion, it would not be a good method for modeling a sanding motion where the velocity of the sanding-block during the many back-and-forth strokes does not correspond closely with specific locations on the surface being sanded. Erratic scribbling motions, and any other actions which involve randomized jiggling (or other motion components where the current action state is not adequately described by position and velocity vectors) will fail to be modeled by this method.

Optimality of the phase space principal curve model While it might be nice to find a more universal method for modeling actions, the specificity of the principal curve's applicability is actually a strength of the method. The assumptions upon which the method is based not only give a fairly clear indication of which actions it is best suited to model; they also provide a good starting point from which to define optimality criteria for comparing different models. We consider the sampled data points from all examples si-

multaneously without explicitly considering which which example each point was sampled, and assume that points which are similar in position and velocity project to the same part of the action, and can be averaged together. The metric for the projection step can be designed explicitly, instead of arising as a side-effect of a black-box method such as one based on neural networks. Once the projection is accomplished, we average together the data points which project to similar parts of the action to get a best-fit model of the entire action. Again, instead of using an opaque learning process, we design the averaging process to explicitly optimize a chosen set of criteria. In this case, we balance the smoothness of the curve against its overall accuracy in approximating the data.

A principal curve found by the method presented in this chapter will be a local minimum of these optimization criteria. For data sets which satisfy the assumptions made at the beginning of the chapter, the resulting curve should be at least close to the globally optimal solution.

Although this method successfully optimizes the model curve given an explicit weighting of fitting error to smoothness, it does not tell us how to choose this weighting. There are three weighting parameters which need to be specified before performing the optimization: γ, p_1, and p_2. Although Section 5.7 tells us how to pick γ and how to select the ratio p_1/p_2, the absolute magnitude of p_1 (or alternatively p_2) is still a free parameter.

There are several approaches which may be used to determine a good balance between smoothness and fitting error. The first is to simply make a qualitative adjustment: the balance can be adjusted interactively using an computer interface which shows a graphical animation of the fitted motion at various levels of smoothing, or by plotting two-dimensional projections of the model line against the sampled data. This is the method used in this thesis, and it works fairly well. In the next chapter, for instance, Figures 6.4 and 6.5 on page 128 demonstrate the effect of changing this weighting by a factor of about 50. The smoother motion might be better for use by a robot, to minimize energy or simplify control, while the less smooth motion might look more "human" within a computer animation.

A second method for balancing smoothness against the quality of the fit is to use an external criterion. For instance, the smoothness weight could be adjusted to minimize the energy used during execution of the action while making sure that the result of the action is still acceptable.

A third approach for weighing smoothness against fitting error is to cross-validate the fit against the training data. Section 3.9 discusses why cross-

Fit	$\|\Delta\mathbf{x}\|^2_{(\text{pprj})}$	$\|\Delta\mathbf{v}\|^2_{(\text{vprj})}$	$\|\Delta\mathbf{y}\|^2_{(\text{prj})}$
α-drawing (5.9, pg. 108)	5.82	3.18	9.00
β-drawing (5.13(d), pg. 113)	21.90	6.95	28.85

Table 5.2: Fitting errors for α-drawing and β-drawing.

validation leads to over-fitting for conventional principal curves, but there is reason to believe that it could work for principal curves in phase space. When conventional principal curves over-fit data in position space, the model curve tends to come closer to points in the data set simply because it winds a greater length into the volume containing the sampled data. The longer curve fits into this volume by changing direction often. The fact that the directional derivative of the phase space principal curve is constrained to approximate the velocity of the nearby data points should counteract this tendency. Cross-validation of the principal curve in phase space is discussed in the future research section (Section 8.2) at the end of the thesis.

Comparing models from different methods For actions which are suitable for modeling using the method described in this chapter, it will eventually be desirable to compare the resulting model against those generated by newer techniques. This comparison should be made in a number of ways. The first is by comparing the absolute magnitudes of the numerical criteria being optimized. For instance, the fitting error can be compared for the position and velocity from the sampled performance data (e.g., Table 5.2), and the smoothness measure for the curve should also be computed. Other modeling methods will likely optimize other criteria, so the values of these additional features should be computed for the models as well.

Another way to compare models is through the use of measures which can give a value for the similarity between the model and the recorded performances from which it was learned. Michael Nechyba's stochastic similarity measure [54] is one promising candidate.

The best way to compare action models, however, is to measure suitability for their intended use. This is an application-specific judgment. For artistic purposes such as modeling actions for use in computer animation, this comparison might be of a qualitative, or even subjective, nature.

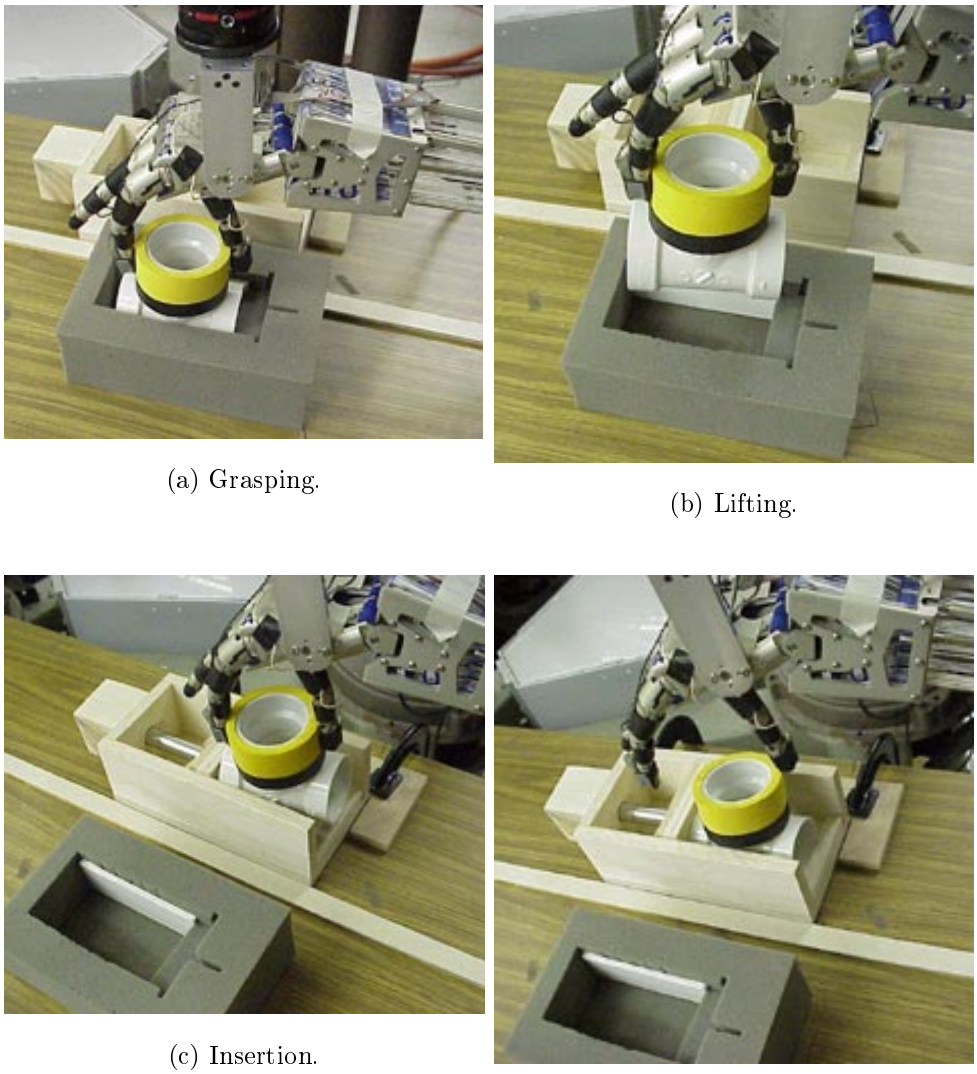
Chapter 6

Action learning in a robot teleoperation experiment

6.1 Introduction: Analysis of a telerobotic experiment

The previous chapters have introduced various methods for building reduced-dimension models of human performance data. In this chapter we use some of these methods, particularly principal curves through phase space, to study data from a telerobotics experiment. We will see that the trajectory fit using velocity information is a valuable tool for analyzing real motions. The data we use is from Anne Murray's spring-box experiment [49]. This experiment was designed to test the effectiveness of tactile feedback on the ability of subjects to perform a challenging manipulation task. The object of the experiment is to insert a part into a spring-loaded box. The task is similar to loading a battery into a radio or other small electronic device, where the battery is held in place partly by spring force.

Figure 6.1 shows the basic steps of the operation. The teleoperator directs a Utah-MIT hand mounted on a Puma robot in grasping a T-section of polyvinyl chloride (PVC) pipe, as shown in Figure 6.1(a). The operator then uses the hand to lift the part (Figure 6.1(b)), then to carry it to the spring-box where the part is then used to depress a spring-loaded plunger (Figure 6.1(c)). The part is then pushed down into the box so that it is held in place by the spring on one side and a lip at the edge of the box on the other. Once the part is secure, the operator causes the robot hand to release



(a) Grasping.

(b) Lifting.

(c) Insertion.

(d) Release.

Figure 6.1: Photographs of the spring-box insertion experiment. The teleoperator directs the motion of the Utah-MIT hand to the PVC part, then uses a Cyberglove and a tactile feedback mechanism to grasp the part with two fingers, pick up the part, and load it into the spring-loaded box. Pictures are courtesy of Anne Murray.

its grasp (Figure 6.1(d)).

As shown in the figure, only the index finger and thumb of the hand are used; the others are positioned so as not to interfere with the task. The operator's hand is instrumented with a Cyberglove to read the joint angles in his fingers, while the location of the operator's hand is read by a Polhemus 6DOF position sensor. The fingertips of the Utah-MIT hand are enhanced with force sensors, and the readings of these sensors are displayed to the user in the form of vibrations from voice coil actuators mounted on the fingertips the operator's index finger and thumb [49, 50, 51]. The operator thus controls the motion of the robot hand by moving his own hand, and controls its grasping motion by making a grasping motion with his own index finger and thumb. The orientation of the Utah-MIT hand in space is fixed, so the operator can control its Cartesian position, but not its roll, pitch, nor yaw. Besides the tactile feedback in the fingers, the operator also has visual feedback from a video monitor, but the operator is isolated from direct view of the robotic testbed and from the sound of the machinery by white noise fed into a pair of earphones. The experiment thus simulates a remote teleoperation scenario.

The original experiment involved two skilled operators, each performing a session of 50 trials each day for three days. In each session, half of the trials were performed with tactile feedback, and half without (i.e., using only visual feedback). In this chapter, we look at data from one session of one of the two operators. The data recorded in the experiment were the reference and measured values of the six joints in the robot hand, the reference position for Cartesian control of the robot hand, and the measured force values from the fingertip sensors. The reference position of the finger angles for the robot were computed from the angles in the index and thumb of the operator as measured by the Cyberglove, while the reference position for Cartesian-space control of the robot hand is computed from low-pass filtered measurements of the operators Cartesian hand position as measured by the Polhemus sensor.

6.2 Simulation and analysis

In Section 1.1 we discussed our general procedure for analyzing data from human performances. The approach is to first perform a high-level analysis of the overall structure of the task, breaking it into simpler component motions, and to then learn a typical motion for each low-level component motion. To

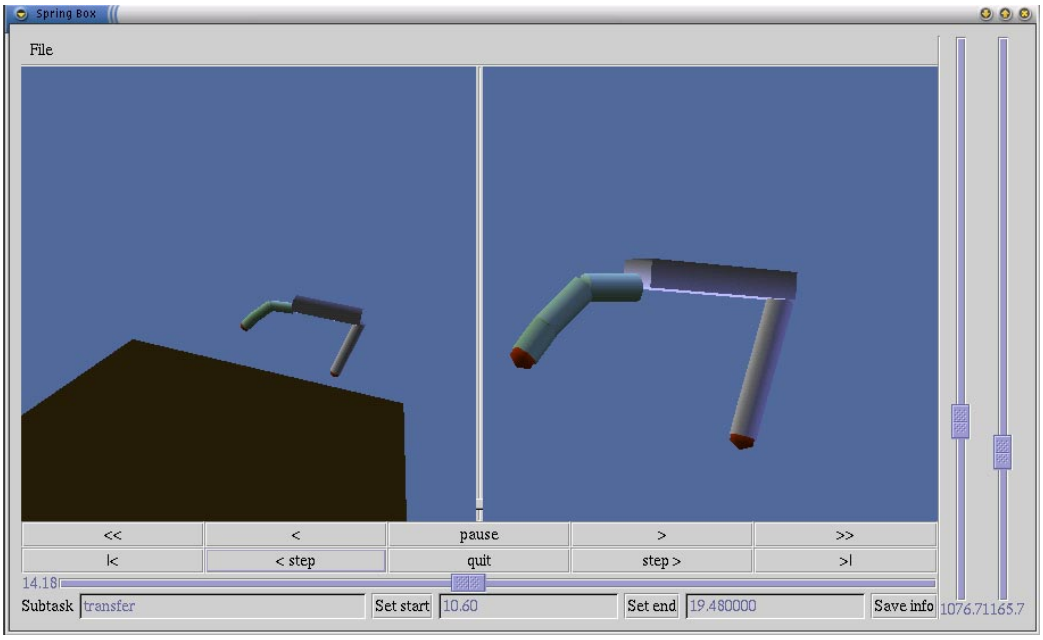


Figure 6.2: Playback interface for studying spring-box data, so that performances can be analyzed, labelled, and rated.

perform this analysis, I wrote a user interface which displays an animated representation of the performance state from the recorded performance data. This interface is shown in Figure 6.2. The interface can load the performance data from a specific trial from a binary (Matlab format) file, and then it can be used like a VCR to view an animation of that trial. The two views shown in the window are OpenGL renderings of the index finger and thumb of the Utah-MIT hand. The view on the left shows the motion with respect to the workspace. The view on the right is a "hand-cam" which maintains a constant position with respect to the simulated robot hand. Both views can be panned and tilted by dragging with the mouse across the simulation image. Forces sensed at the index finger and thumb are shown by the scales on the right of the interface window. The forces are also shown by the color of the fingertips. These are varied from black, for zero pressure, to red for the maximum pressure sensed in a given trial.

The two rows of buttons and the scale below the views control the time index of the data like a VCR control panel. The bottom of the interface

allows the user to label the start and end times for different parts of the trial, and to write this labelling information to a file for later use.

Using this interface, we were able to analyze the structure of the performances. We broke the task down into five component motions:

1. *Pre-grasp.* The teleoperator starts with his hand resting on a platform in front of him, so that the robot hand is above and in front of the PVC part. He lifts his hand off the platform, then moves his hand to position the robotic hand for grasping the part.

2. *Grasp.* The teleoperator closes the robotic fingers until the part is securely grasped.

3. *Part transfer.* Once the part is securely grasped, the teleoperator uses the robotic hand to lift it and transport it to the spring-box.

4. *Loading.* The teleoperator causes the robot hand to push the part forward against the plunger in the spring-box until the end of the part is beyond the lip at the edge of the box. Then, the hand is moved back until the spring secures the part against the lip, and the operator causes the fingers to release the part.

5. *Finish.* The operator carefully moves the robot hand away from the part so as not to knock it out of place, then moves his hand back to the resting platform.

After segmenting the task into these components, we used the interface to label each component in each of the 50 trials, and roughly graded the quality of each grasping motion and loading motion. Once the component motions were graded and labelled, we were able to construct data sets for each of these components. For each component motion, trials were selected either by the grades of that motion (for grasping and loading), or the grades of the component motions bounding the selected component (for pre-grasp, part transfer, and finish), and data was extracted over the proper time interval of each selected trial. This data was preprocessed by a spline smoother to compute the velocity of the motion while avoiding noise problems, and then the resulting data points were sub-sampled to reduce the number of points to a few hundred to simplify the learning computations. The resulting data sets are analyzed in the following sections.

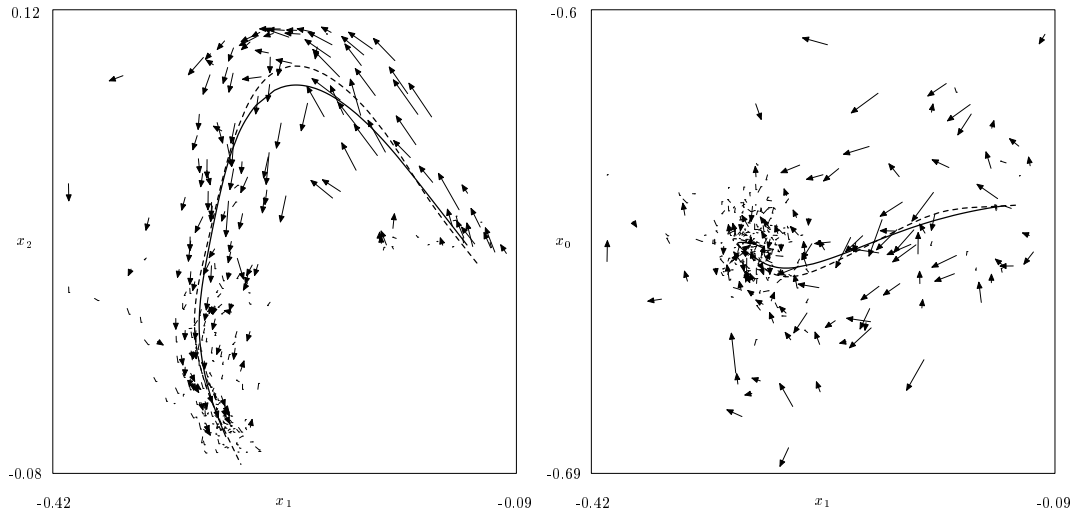
6.3 Pre-grasp: approaching the part

The first motion in the task is where the operator lifts his hand from the resting platform, then directs the robot's hand to a position from which it can grasp the part. We selected data from trials where the operator accurately moved the robot hand to the part with a single motion and was able to lift the part on the first attempt (29 of the 50 trials). The end of the pre-grasp phase is the point of each trial just before the force sensors register contact with the part.

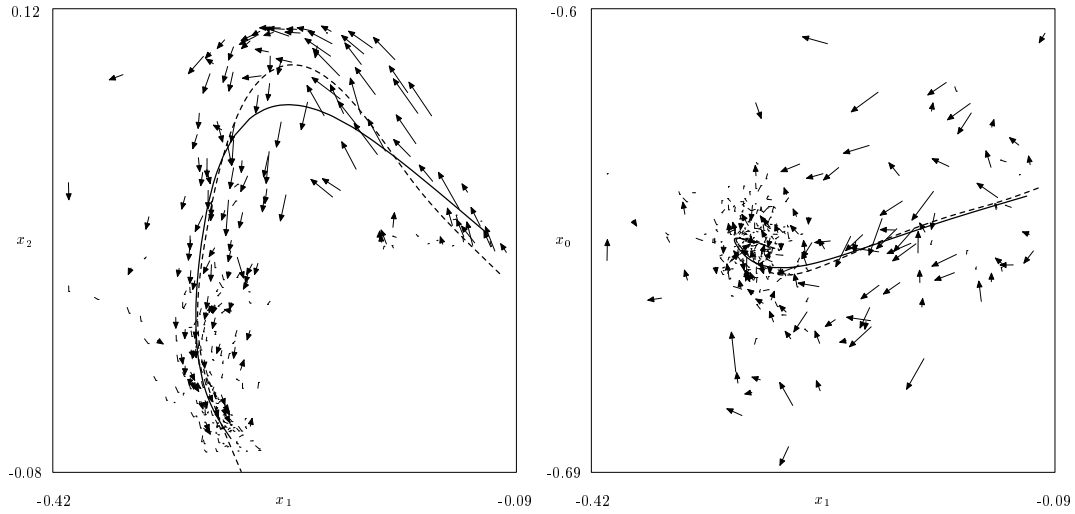
Because this data corresponds to an unconstrained motion in space, it makes sense to model it with a nominal trajectory in phase space using the method developed in the previous chapter. We thus fit a principal curve in phase space to the Cartesian position data for the motion of the robot hand. To investigate the effect of velocity information on the fit we compared the results for example fits both with and without a weight on velocity error, and we investigated the effect of smoothing by varying the ratio of error weighting to the weight of the smoothness penalty term.

Figure 6.3 shows the results of the first two iterations of the principal curves algorithm with and without a velocity penalty, using heavy smoothing. Figure 6.4 shows the third iteration. The left-most plots in these figures present a side view of the course of the motion, with x_2 giving the height of the hand, and x_1 indicating backward-forward motion. Note that the axes of these plots are not scaled proportionally. The height of the starting configurations is consistent because the hand is sitting on a platform, but their x_1 positions vary because the hand may be sitting relatively more forward or backward on the platform. The operator's motion is to start with his hand far out in from of his body, and then to move his hand up and back toward himself, then down to the grasp point. The right-most plots give a top view of the motion, and show that the motion is relatively straight back, with little sideways movement. At the end of the motion, we see that the operator moves the robot hand more slowly and slightly forward as he guides it to the part. When the smoothness penalty is high, as in these plots, we see that the fitted motions are similar whether or not we are weighing velocity error, and that the model of the best-fit trajectory appears to consist of a single continuous motion.

Figure 6.5 shows the final result from fitting principal curves with less of a smoothness penalty, with one fit weighing velocity error and one not. The ratio of error penalty to smoothness penalty is increased by a factor



(a) Iteration 0.



(b) Iteration 1.

Figure 6.3: Results from the first two iterations of the principal curves algorithm for fitting the pre-grasp motion in the spring-box experiment. The solid curve uses weights $p_1 = 0.35$, $p_2 = 0.64$, while the dashed curve uses $p_1 = 0.99$, $p_2 = 0.0$.

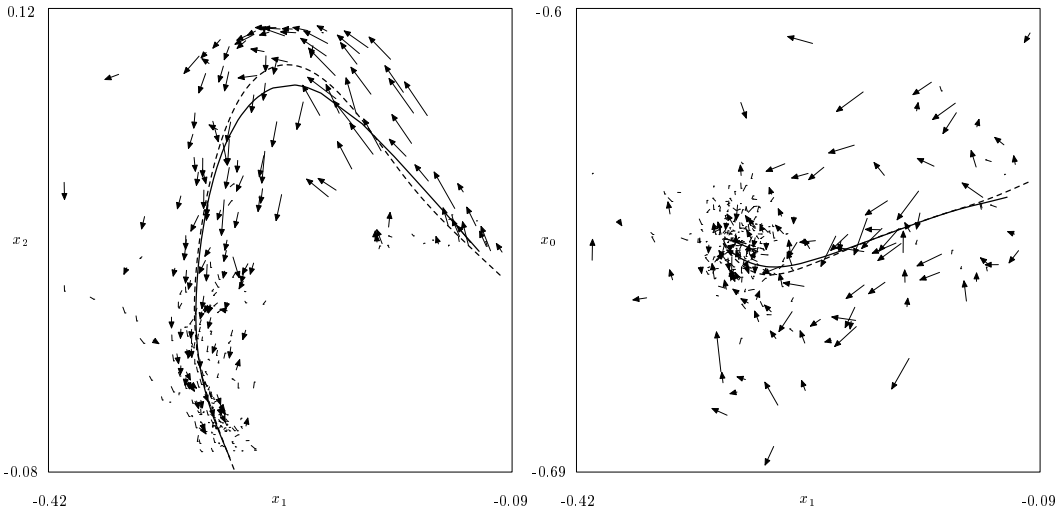


Figure 6.4: Results from iteration 3 of the principal curves algorithm for fitting the pre-grasp motion in the spring-box experiment. The solid curve uses weights $p_1 = 0.35$, $p_2 = 0.64$, while the dashed curve uses $p_1 = 0.99$, $p_2 = 0.0$.

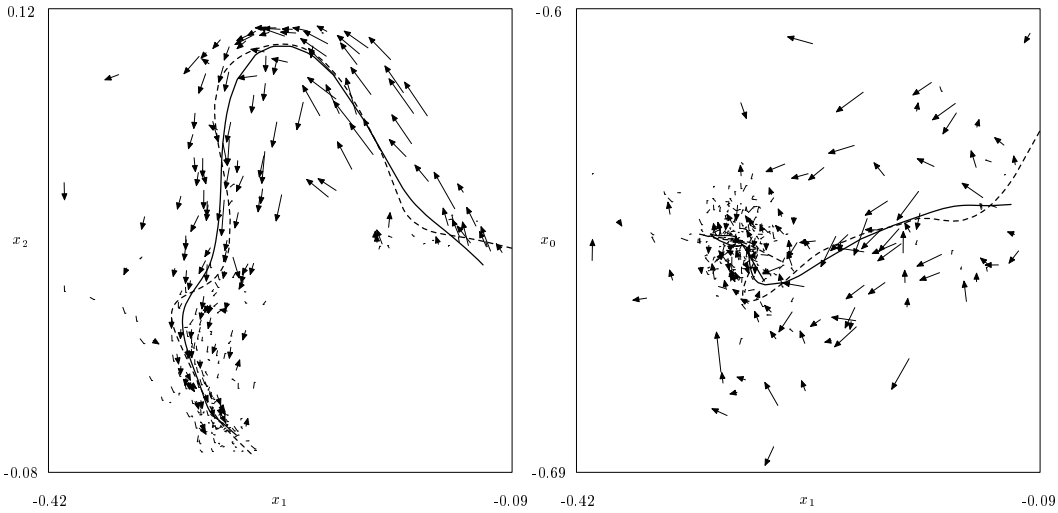


Figure 6.5: Final results from the principal curves algorithm for fitting the pre-grasp motion in the spring-box experiment. The solid curve uses weights $p_1 = 0.35$, $p_2 = 0.6495$, while the dashed curve uses $p_1 = 0.9995$, $p_2 = 0.0$.

of 50 from that of the model shown in Figure 6.4. We see that the best-fit trajectory now appears to be constructed of several distinct motions: a lift, a fast general approach to a point above the part, then a slower approach from slightly behind the part to the eventually grasping position. One effect of the velocity error weight is to improve the fit at the beginning of the motion; the fit without the velocity weight slides along the platform to get closer to more of the starting configuration points in a manner inconsistent with the example trajectories. The other effect of the velocity error weight is to reduce the amount of overshoot as the hand transitions from the fast initial motion to the fine-positioning motion at the end of the trajectory. These effects, and the actual angle of final approach to the grasp position, are better demonstrated in Figure 6.6 which shows example points and best-fit trajectories with proportionally-scaled axes. The results in the remainder of this section refer to the best-fit trajectory estimate from this figure shown with the solid line: the fit with less smoothing and use of velocity information ($p_1 = 0.35$, $p_2 = 0.6495$).

To show the general trend of the speed of motion, we plot velocity against the parameterization variable λ. Figure 6.7 shows the velocity values of the points in the data set, as well as a smoothing-spline model of this data and estimated variance about this model. The expected trend is clearly evident. At the beginning of the motion, while the operator lifts his hand from the platform and moves roughly near the grasp point, the motion is generally fast although the actual speed of the example data points is highly varied. As the hand nears the grasp point the speed of the motion slows, and the variance of the example data is much lower. This is indicative of the care necessary for accurate positioning of the hand in preparation for a successful grasp.

To demonstrate the quality of the fit, we can plot the trend of λ-values verses time for each trial. Four representative plots from the 29 trials we used are shown in Figure 6.8. Most of the trials (16 of the 29) are clearly monotonic like the plot of Trial 10 shown in Figure 6.8(b). A few trials (7 of 29) are generally monotonic, but with minor flat sections or minor decreases in λ in a pair of consecutive points, in a manner similar to Trial 9. Trial 12 shows a single point with a λ-value that is significantly less than that of the point before it. There are 5 trials with this characteristic. Trial 13, whose final 5 points have nearly the same λ-value, is unique. Using the methods described in Section 5.11, we can compute the diagnostic values $M_{\mathrm{p}} = 0.76$ and $M_{\mathrm{p1}} = 0.79$ (see Table 5.1 on page 115).

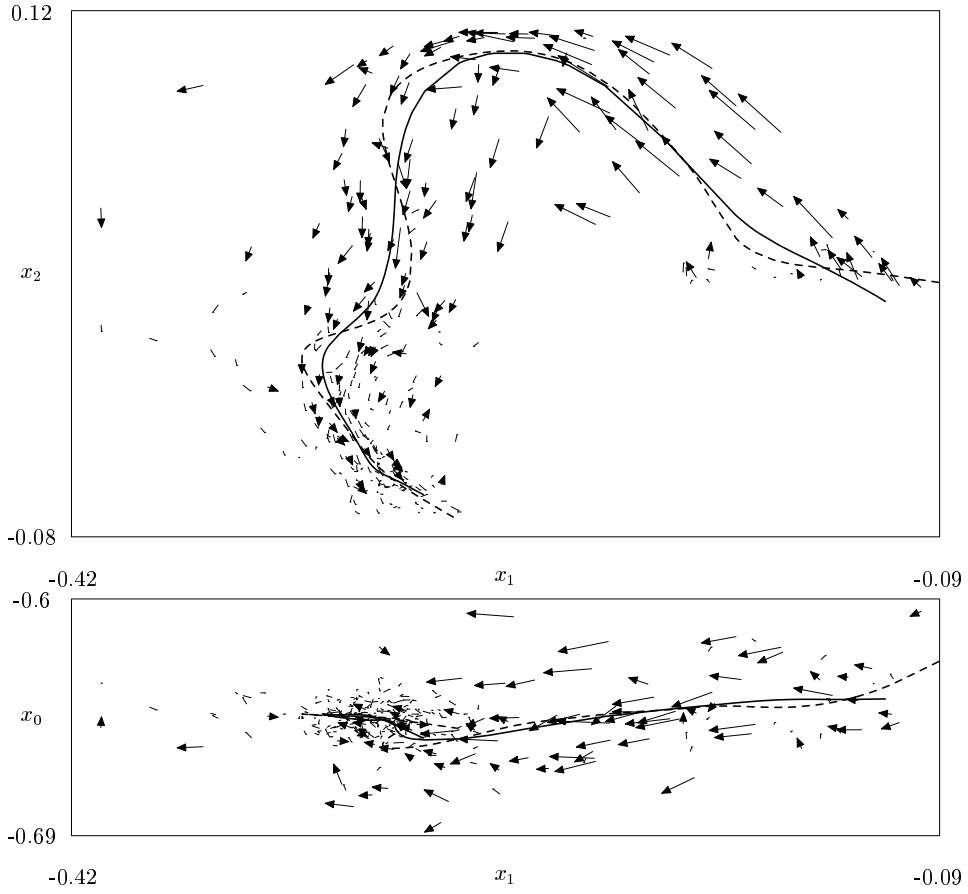


Figure 6.6: Wide version of final results principal curves algorithm for fitting the pre-grasp motion in the spring-box experiment. The solid curve uses weights $p_1 = 0.35$, $p_2 = 0.6495$, while the dashed curve uses $p_1 = 0.9995$, $p_2 = 0.0$.

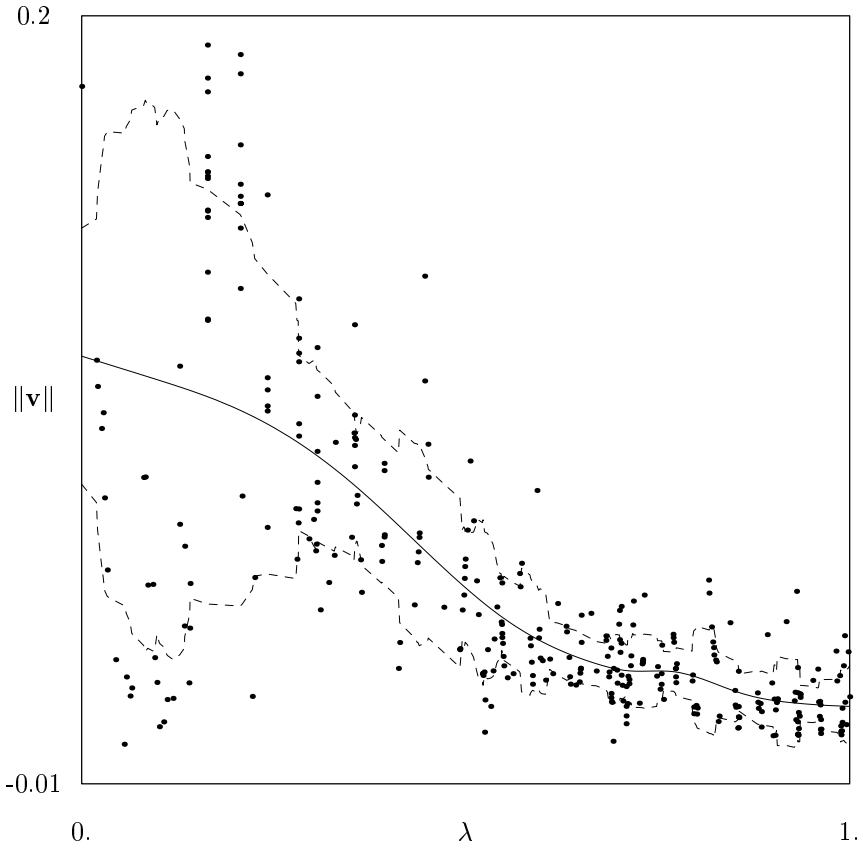
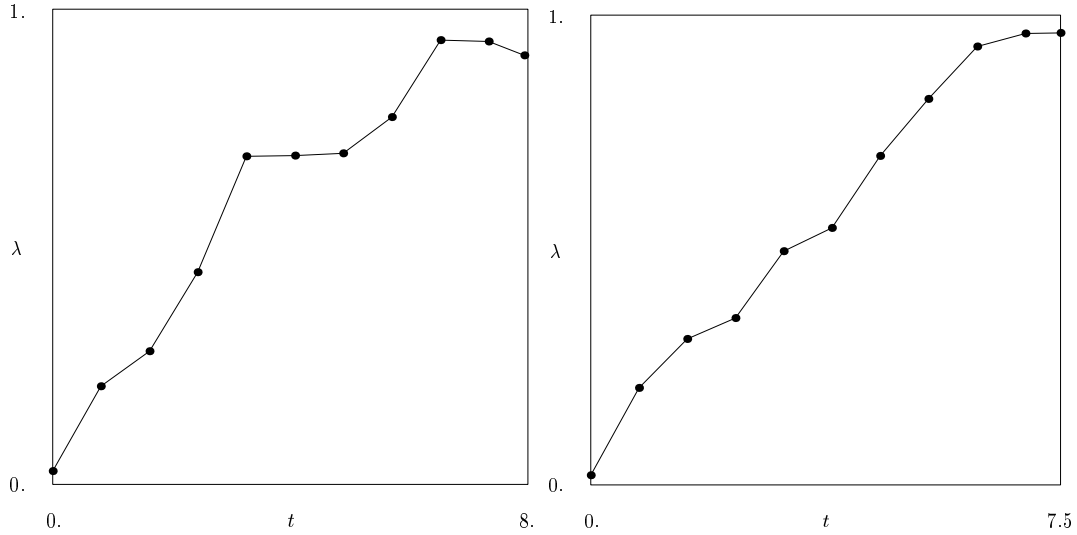
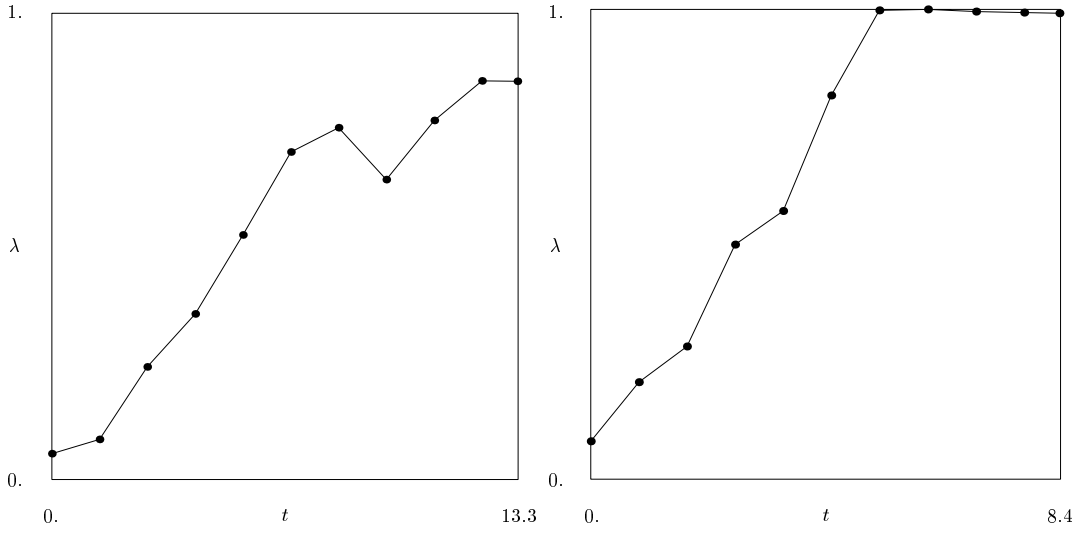


Figure 6.7: Smoothing-spline fit of speed vs. λ with estimated variance.



(a) Trial 9

(b) Trial 10



(c) Trial 12

(d) Trial 13

Figure 6.8: Plots of λ values verses time for four trials from spring-box experiment.

6.4 Grasp and Finish

Although grasping the part was one of the more difficult tasks for the operator, it is really a simple force-regulation task and thus trivial to model. The object of the experiment was to grasp with as little force as possible while still maintaining control of the part. The difficulty of this skill was mainly the interpretation of the vibration feedback from the voice coils to control the force of the grasp. Murray's thesis [49] gives a full analysis of this aspect of the experiment.

The analysis of the finishing motion, after the part is loaded in to the spring-box, is similar enough to the analysis of the pre-grasp and part transfer skills that we will also not include it in this thesis.

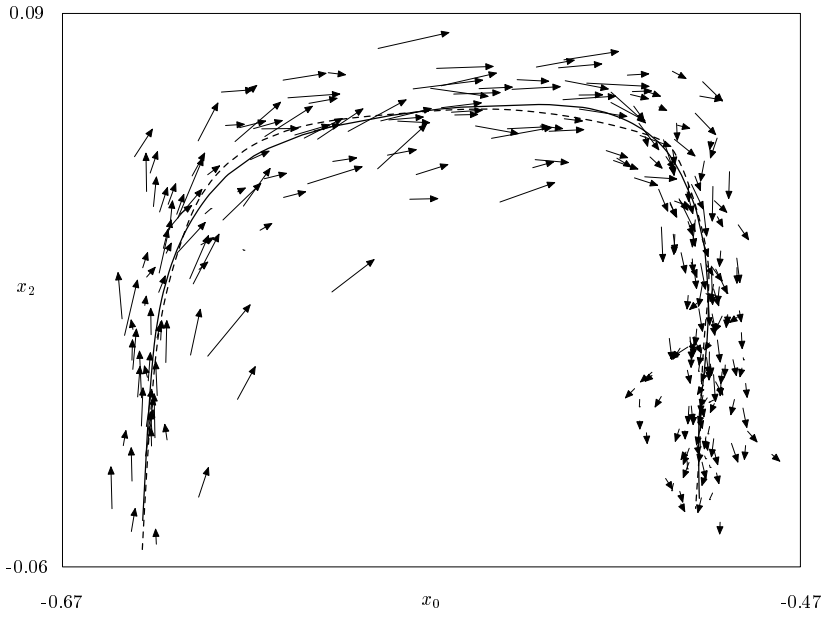
6.5 Part transfer

Because the part transfer from the grasp point to the spring-box is just a motion in space, we can use the same method as that used for the pre-grasp phase. The results of the principal curve fit in phase space are shown in Figures 6.9 and 6.10. We see that velocity information greatly improves the fitted trajectory, and that the motion has roughly three parts. The first part of the motion is a vertical lift of the part from the grasp point out of the part holder. The second part is the horizontal motion into the plane of the spring box, and the third motion is down and forward toward the face of the plunger in the spring-box.

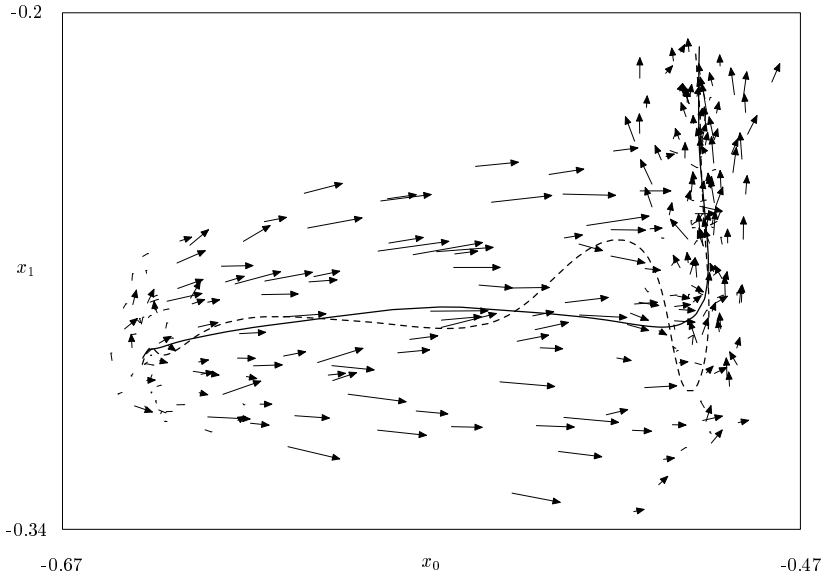
Figure 6.10(b) shows a fit of the speed of motion plotted against the parameterization variable. We see, as expected, that the hand accelerates during the lift and through the horizontal motion, and then slows as it approaches contact with the plunger in the spring-box to allow for careful alignment. Plots of λ verses time for the fit of the transfer motion are all nicely monotonically increasing, resembling Figures 6.8(b) and indicating a very good fit of the examples. The diagnostic values for the parameterization are $M_{\mathrm{p}} = 0.98$ and $M_{\mathrm{p1}} = 0.93$ (Table 5.1 on page 115).

6.6 Loading the spring-box

We fit the action of loading the spring-box using a principal curve in phase space of the Cartesian space motion. Figure 6.11(a) shows the fitted motion

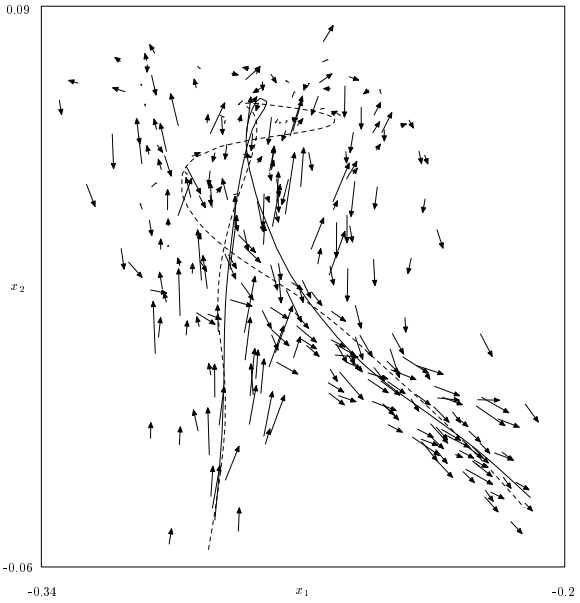


(a) x_0 vs. x_2 plot (operator view)

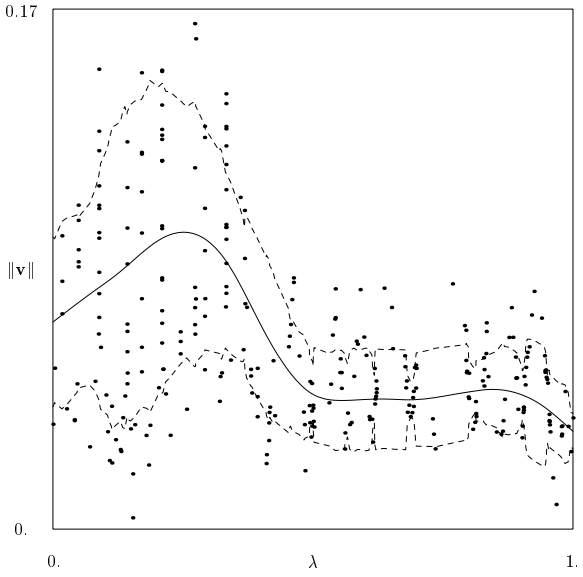


(b) x_0 vs. x_1 plot (top view)

Figure 6.9: PVC transfer motion to spring-box: operator and top views. The solid curve uses weights $p_1 = 0.35$, $p_2 = 0.6495$, while the dashed curve uses $p_1 = 0.9995$, $p_2 = 0.0$.



(a) x_1 vs. x_2 plot (side view)



(b) Speed verses λ

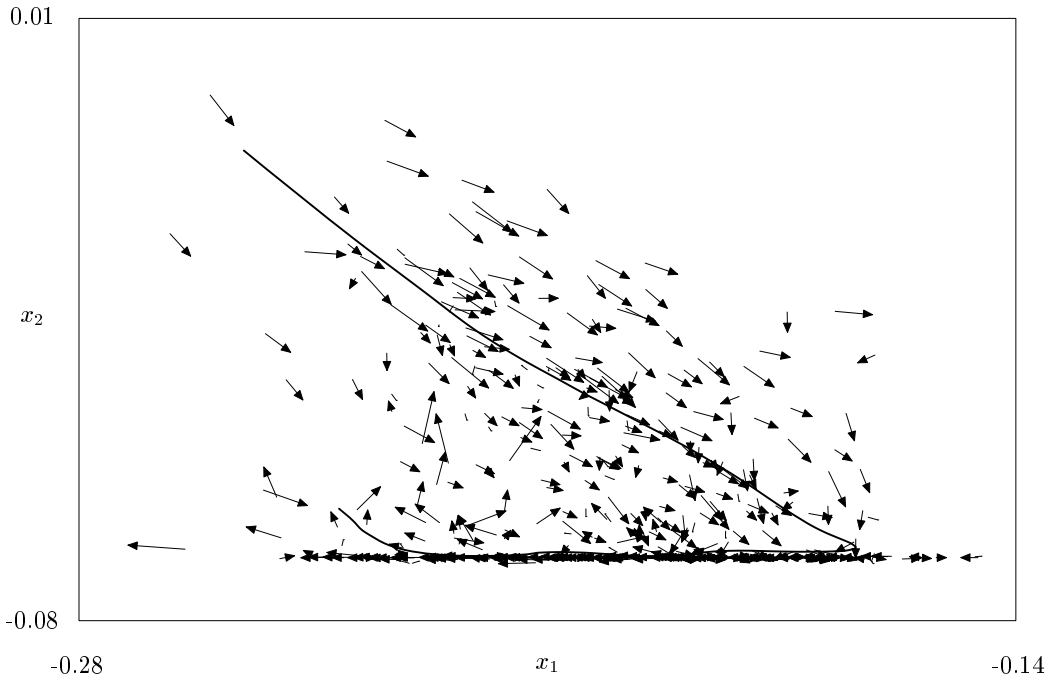
Figure 6.10: PVC transfer motion to spring-box: side view and speed vs. λ. The solid curve uses weights $p_1 = 0.35$, $p_2 = 0.6495$, while the dashed curve uses $p_1 = 0.9995$, $p_2 = 0.0$.

(fit using velocity information). We see very clearly that this skill has two component motions: the part is first moved forward and downward as it is pushed against the plunger and into the box, and then the hand is pulled straight back, leaving the part locked between the plunger and the lip at the edge of the spring-box. Figure 6.11(b) shows that the speed of this motion stays generally low. The λ verses time plots for the fit are generally good ($M_{\mathrm{p}} = 0.75$ and $M_{\mathrm{p1}} = 0.72$).

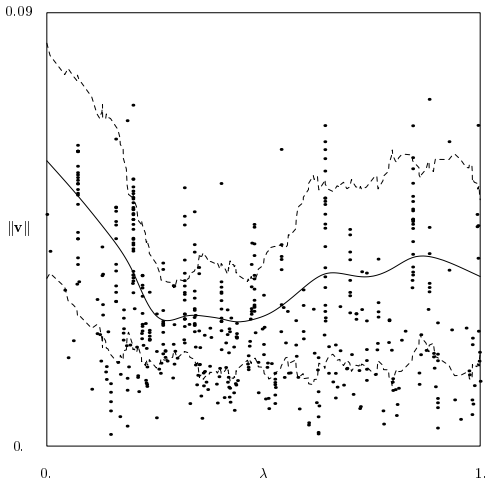
The fundamental aspect of the part-loading skill is this motion in space. Across all the example trials, similar portions of the motion correspond to the same logical part of the loading skill. The resulting λ-parameterization is thus a good measure against which to compare the other aspects of the performances: finger motion and force feedback. We can see how the λ-values relate to the motion in Figure 6.12. The top plot shows the horizontal motion of the hand, moving first forward and then backward. The bottom plot shows the downward vertical motion as the part is pushed into the box, then shows that the hand stays low while it is pulled back. In the Cartesian space projection of the same data in Figure 6.11(a) these λ-values are only implicit in the ordering of the fitted trajectory points.

Figure 6.13 is plotted below Figure 6.12 to show how the force readings correlate to the motion. The most obvious trend is that as the part is pushed down into the box, the force readings approach zero, and then remain zero as the hand is pulled back. This might be due to the fact that the fingers slowly release their grip during the insertion. However, note the many zero readings for both sensors near the beginning of the motion. These are probably due to the fact that the fingertips are too low on the part during these trials, and the grasping pressure thus is being applied above the location of the force sensors on the robotic fingertips. These sorts of effects make interpretation of the force sensor information somewhat uncertain for this data set.

Figure 6.14 plots the motion of three finger joints verses λ, with the plot of forward-backward motion at the bottom for comparison. The top plot, for q_1, is representative of the thumb-joint data: there is no apparent trend in the data. This is probably because the thumb is used as a simple pushing instrument, and its finger-joint motions are just not very important. The middle plots, of q_4 q_5, are from the index finger. We see that the operator tends to release (straighten) this finger during the motion, but not at any specific part of the motion.



(a) x_1 vs. x_2 plot (side view)



(b) speed verses λ

Figure 6.11: Spring-box loading motion

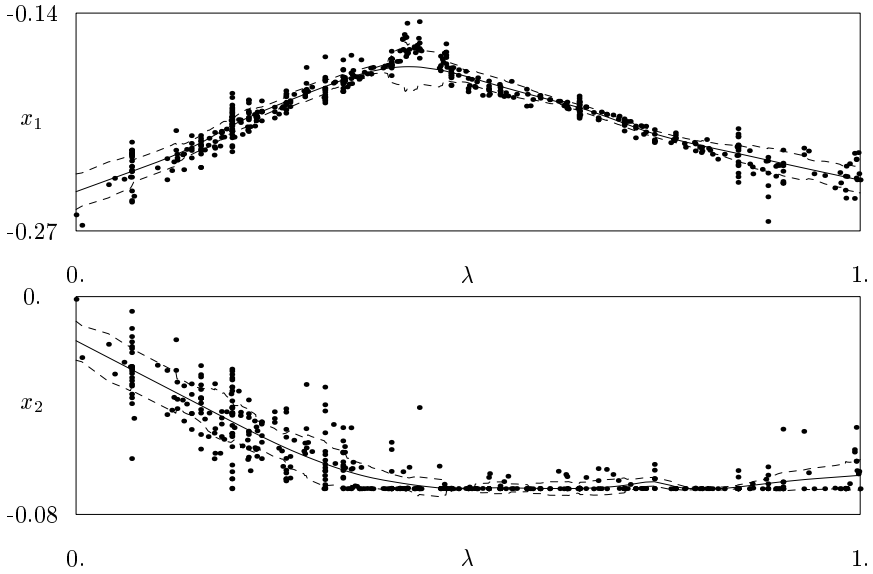


Figure 6.12: Spring-box loading motion: position verses λ

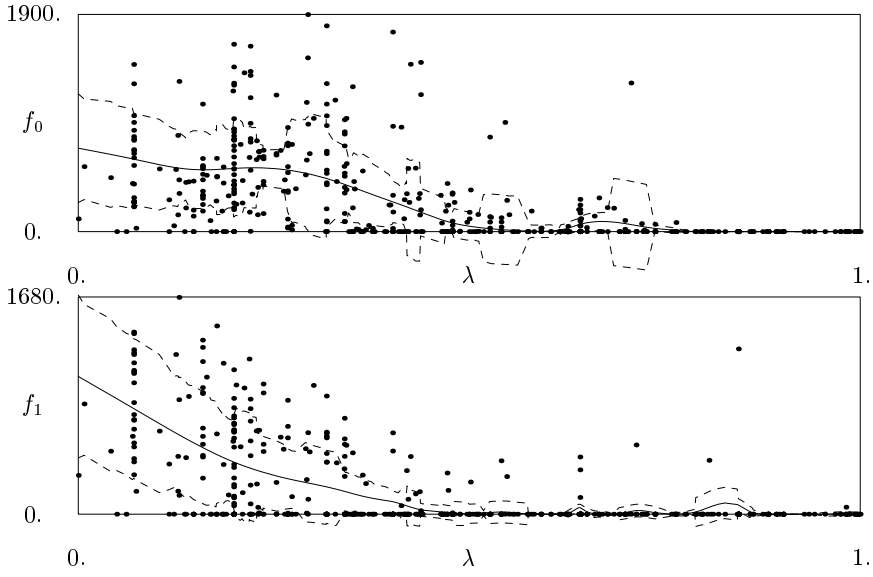


Figure 6.13: Spring-box loading: measured force verses λ. f_0 is force at thumb, f_1 at index finger.

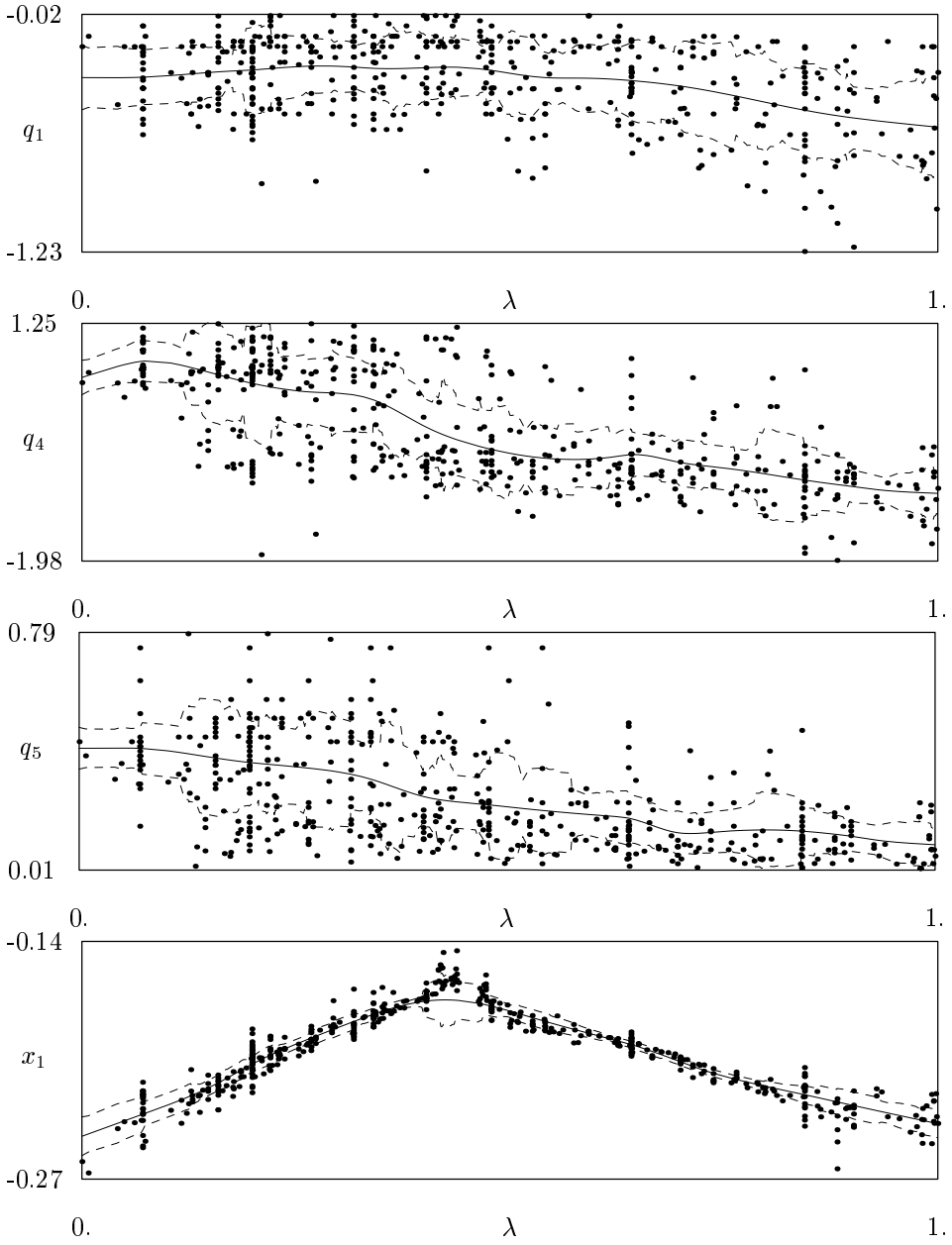


Figure 6.14: Spring-box loading: finger motion verses λ. q_1 is from thumb, q_2 and q_3 from index finger. x_1 is hand motion (for comparison).

6.7 Discussion

In this chapter, we demonstrated the effectiveness of the methods developed in the previous chapter for modeling components of a robotic teleoperation experiment. The trajectory-fit in phase space is a valuable tool for characterizing the motions which the teleoperator used to guide the robot in performing a difficult dextrous manipulation task.

Of course these models do not represent the actual problem the human operator solved: using visual and tactile information to determine which actions are necessary to successfully complete the task. A reaction model is necessary for abstracting this sort of ability, and there is a great deal of research being performed on methods for performing this very challenging learning task. However, the action models presented here are very good at answering the question of what the teleoperator actually did to successfully complete the task. The action model abstracted a representation of what a "typical" performance of the task looked like, for instance. These models were also useful for identifying the critical points in the motion (such as the location of the grasp) for which the operator's performance was slowest, most deliberate, and most consistent between performances. The best-fit trajectory also provided a useful parameterization against which to measure other action and sensory variables.

Although models of reaction skill are the most direct solution for many intelligent control applications, action models can also be used in practice for guiding robotic performance. Many industrial robots are programmed to trace trajectories in space for the purposes of spray painting and welding, and the actual process of supplying the proper via points (e.g., by teach pendant) and building paths between them tends to be difficult, primitive, iterative, and frequently give sub-optimal results. The methods demonstrated here for trajectory fitting and trajectory smoothing could greatly improve the ease of programming and the quality of results for these kinds of applications. Finally, even for applications which need sensory feedback to eliminate modeling errors, action learning can provide a prototypical feed-forward motion around which the feedback control can be developed.

Chapter 7

Modeling actions for gesture recognition

7.1 Introduction

Gesture recognition is one of the most important applications of action learning. When we build models for recognition only, the form of the models does not necessarily need to be useful for generating representations of typical or best-fit trajectories, for animation, nor for qualitative motion comparisons. All we need is to be able to estimate a probability that a given example performance is associated with the class of actions represented by a given model. In this chapter, we describe a kind of model for action learning which is designed solely for use in gesture recognition, and demonstrate its use in an sign-language recognition application which can learn to recognize new gestures interactively.

One example of using action models for gesture recognition has already been presented in Section 3.5. In that approach, the statistical assumptions underlying the fitted model are used to estimate a probability that a given set of data would be generated by the model. Before training the action models, the time variable of each training example was linearly scaled to span the interval 0 to 1. A spline smoother was used to find the best-fit motion from the training examples for each motion, using the scaled time values as the smoothing parameterization. Although the resulting models proved effective for gesture recognition, they are based on assumptions which are much more constraining than necessary for the purpose of gesture recognition. The spline

smoother assumes that the training data has a locally Gaussian distribution about a single best-fit value for each parameterization value, and that the best-fit motion is relatively smooth. In addition, the smoother model will have a difficult time recognizing motions which can involve branching or looping trajectories.

While the assumptions made for the development of smoother-based action models are useful for building best-fit trajectories, they are unnecessary when the models are purely for gesture recognition. The models used in this chapter are based on discrete hidden Markov models (HMMs), and are thus very different kind of model than those used elsewhere in this thesis. Because HMMs are based on Markov models, where state-transition probabilities are tabulated in a matrix, there is no need for any *a priori* assumption of probability distribution like that used for the spline smoother. Thus, unlike models based upon smoothing splines or principal curves, there is no assumption of Gaussian distributions nor any problem with probabilistic branching (Section 3.8). For the purposes of these models, an example performance is represented as a single-dimensional or multi-dimensional sequence of symbols. Each HMM model consists of an initial state distribution vector $\boldsymbol{\pi}$, a state-transition probability matrix $\mathbf{A}$, and an output-probability matrix $\mathbf{B}$. Instead of recording a "snapshot" of the state of the system at a given time, each symbol or symbol-vector in the sequence represents the state of the system within a particular time interval, and the time intervals corresponding to the symbols in the sequence overlap. The symbols will be represented as integers, but their ordinal value has no particular meaning: symbol 1 is no "closer" to symbol 2 than it is to symbol 3. This is in contrast to the continuous variables of the reduced-dimensional representations from NLPCA or principal curves.

The method demonstrated here for training the models facilitates online, interactive learning of new gestures and automatic refinement of previously-learned models in addition to on-line recognition. The motivation is to allow us to build systems for intuitive human-computer interfaces, particularly for constructing interactive gesture-based robot programming environments. As people interact with machines which are increasingly complex and autonomous, they should be allowed to focus their attention on the content of their interaction rather than the mechanisms and protocol through which the interaction occurs. This is best accomplished by making the style of interaction more closely resemble that to which they are most accustomed: interaction with other people.

This style of interaction should be particularly useful in applications where robots are programmed by example, or for gesture-based programming. Examples of such applications include the research of Tung and Kak [78], who demonstrate automatic learning of robot tasks through a DataGlove interface. Kang and Ikeuchi [34] developed a system for simple task learning by human demonstration using a vision system. Voyles [80] developed a system for gesture-based programming via a multi-agent model.

One capability which is currently lacking in systems such as these is a mechanism for online teaching of gestures with symbolic meanings. Most gesture recognition systems either require some explicit programming, or in the case of neural-nets, require off-line training of model parameters. Such systems are thus unsuitable for interactive applications where gestures must be taught and then recognized without waiting for an off-line training or programming phase. A teach-by-demonstration system should also be able to learn a new gesture or skill in an online manner and with a very small number of examples. This simplifies the teaching process because it more closely resembles the manner of instruction which we commonly use to train people.

The gesture recognition system presented in this chapter can interactively recognize gestures and learn new gestures online with as few as one or two examples. It is also able to update its model of a gesture iteratively with each example it recognizes.

7.2 Hidden Markov models for recognition

Our goal is to make a system which can not only interact with a user by accurately recognizing gestures, but which can learn new gestures and update its understanding of gestures it already knows in an online, interactive manner. Our approach is automated generation and iterative training of a set of hidden Markov models which represent human gestures. Using this approach, we have built and tested a system which recognizes letters from the sign language alphabet using a Virtual Technologies 'Cyberglove'.

The most basic assumption we make about the nature of human gestures are that they are "doubly stochastic" processes. These are Markov processes whose internal states are not directly observable. For each state transition in such a process, the system generates an observable output signal whose value depends on a probability distribution which is fixed for each internal

state. A model of such a system is called a hidden Markov model (HMM). Generation and use of HMMs for gesture-learning and gesture-recognition is discussed in Section 7.4. Modeling human gestures as HMMs allows us to deal with the highly stochastic nature of human gesture performance.

For computational simplicity, we assume that the HMMs are 'discrete' HMMs. These are HMMs whose observable outputs are members of a finite set of symbols. Before we can use discrete HMMs to model gestures, we must therefore preprocess the raw data from the gesture input device into a sequence of discrete symbols. The algorithm we use for this process is discussed in Section 7.5.

In the rest of this thesis, the form of the feature-space representation has been a reduced-dimension vector of real values (Section 1.2). This is very different from the representation used in this chapter. We use a one-dimensional sequence of symbols, but we could also use a multi-dimensional sequence by using multiple output-probability matrices. The design choice to use a discrete symbol representation is partly due the simplified HMM computations, but is also due to the fact that much of the additional information in a real number state representation is not necessary for recognition of actions. A real-number vector representation is useful for action models reconstructing motion in space, but even for this purpose a discrete state-based representation is sometimes adequate [88].

Gesture recognition is also, to a certain extent, a process of data compression, where we input a large amount of raw data and output a signal value that represents the classification of the gesture. For the method outlined in this chapter, this compression process is composed of two stages. The greatest data-reduction occurs when the preprocessor converts the large, multichannel stream of data from the gesture input device to the sequence of discrete observable symbols. We reduce about 0.5-1.0 Kbytes of data from the Cyberglove to a sequence of 5-10 observable symbols from a set of 32 and 256 symbols, which is a reduction of about 100:1. Since the online portion of our gesture learning process is limited to the modification of the parameters of HMMs, this data-reduction greatly increases the speed and simplicity of the online learning process by focusing the HMM on modeling specific features of the signal. In Section 7.7, we discuss the results of our determination of the proper amount of data-reduction necessary for allowing the system to effectively learn gestures through an interactive training process.

7.3 Interactive training

In this chapter, we describe a system which allows for for interactive, online training. In this system, each kind of gesture is represented by an HMM, a list of example observation sequences, and an optional action to be performed upon recognition of the gesture. Our concept of interactive training is currently based on the following general procedure:

1. The user makes a series of gestures.
2. The system segments the stream of data from the input device into separate gestures, and in real time, tries to classify each gesture.
   (a) If the system is certain about its classification of a gesture, it immediately performs an action associated with that gesture (if one has been specified). Such an action could be passing the result of the classification to a higher-level HMM, or sending a command to a robot.
   (b) If the system is in any way unsure about its classification of a gesture, it queries the user for confirmation of its classification. The user either:
      - confirms the system's classification, or
      - corrects the classification, or
      - adds a new gesture class to the system's bank of gesture models.
3. The system adds the symbols of the encoded gesture to the list of example sequences of the proper gesture model, then updates the parameters of that model by retraining the HMM on the accumulated example sequences.

In our implementation, recognition of the gesture and automatic update of the HMM through the Baum-Welch algorithm is fast enough not to be noticeable during normal use of the system. This provides the system with a truly interactive character. For example, a user controlling a robot through the gesture system could perform a gesture which the system has not seen before, and the system would immediately respond by asking what kind of gesture it is. The user could respond that it is a "halt" gesture, and that the robot should stop its current motion when that gesture is made. The

next time the user performs that gesture, the system should recognize it and immediately halt the motion of the robot.

7.4 Learning and recognition

Hidden Markov Models [58, 59] are commonly used for speech recognition, but have also been used for characterizing task information and human skills for transfer to robots in telerobotic applications [26, 87, 88].

A hidden Markov model is a representation of a Markov process which cannot be directly observed (a "doubly stochastic" system). The discrete form of the HMM is represented by three matrices, $(\mathbf{A}, \mathbf{B}, \boldsymbol{\pi})$. The matrix $\mathbf{A} = \{a_{ij}\}$ specifies the probability that the internal state will change from i to j. $\mathbf{B} = \{b_{jk}\}$ represents the probability that the system will generate the observable output symbol k on transition to state j. The third component of a hidden Markov model is a vector $\boldsymbol{\pi}$ indicating the distribution of probability that any given state is the initial state of the hidden Markov process.

There are three problems commonly associated with hidden Markov models [59]:

1. determining the probability that a given sequence of observable symbols would be generated by a given HMM,

2. determining the most likely sequence of internal states in a given HMM which would have given rise to a given sequence of observable symbols, and

3. generating an HMM that best 'explains' a sequence or set of sequences of observables.

We are directly concerned with the first problem for gesture recognition, and the third problem for generating the HMMs used in gesture recognition.

Gesture recognition. The problem of recognizing a gesture from a given set of input data is an example of problem 1. First, raw input data from the input device is preprocessed into a sequence of discrete observation symbols $O = O_1 O_2 O_3 \ldots$. Then it is determined which of a set of HMMs, each modeling a different gesture, is most likely to have generated that sequence: $L = \text{argmax}_l[P(O|(\mathbf{A}_l, \mathbf{B}_l, \boldsymbol{\pi}_l))]$. In addition, the system determines if there

is an ambiguity between two or more gestures (the probabilities of the most likely gestures are too close to one another) or if no known gesture is similar to the observed data (the probability of the most likely gesture is too small).

Learning gesture models Developing the HMM which will be associated with a gesture is an example of problem 3. The algorithm which is commonly used for this purpose is the Baum-Welch (BW) algorithm. Baum-Welch uses an iterative expectation/maximization process to find an HMM which is a local maximum in its likelihood to have generated a set of 'training' observation sequences. Although the space of possible HMMs for a given HMM structure is generally full of many of these local maxima, it has been observed that most work similarly well for practical use in modeling doubly stochastic systems.

While training of HMMs is normally a batch process, where many examples of a given gesture are used simultaneously as inputs to the Baum-Welch algorithm, we use a different approach. We begin with one or some small number of examples, run BW until it converges, then iteratively add more examples, updating the model with BW after each one. This allows for an online, interactive style of gesture training. Section 7.7 compares the results of this kind of incremental training to batch-style processing.

7.5 Signal preprocessing

Since we are using discrete HMMs, we need to represent gestures as sequences of discrete symbols. We must therefore preprocess the raw gesture data, which in our case are values of 20 joint-angles in the hand, estimated from 18 sensors in the Cyberglove at about 10 Hz.

The first choice we must make in performing this preprocessing is whether we want to generate a one-dimensional sequence of symbols or a multi-dimensional sequence. The multi-dimensional sequence may be used as input to a multi-dimensional HMM. If we assume that the dimensions of the observable sequence are dependent only on the internal state and not on one another, then the multi-dimensional HMM will have a single $\mathbf{A}$ matrix and multiple $\mathbf{B}$ matrices, one for each dimension of the output symbols. Although it makes some intuitive sense to train a multi-dimensional HMM which takes as inputs, say, 5 dimensions of symbols (one for each finger), we chose for simplicity to look at the hand as a single entity, and generated a

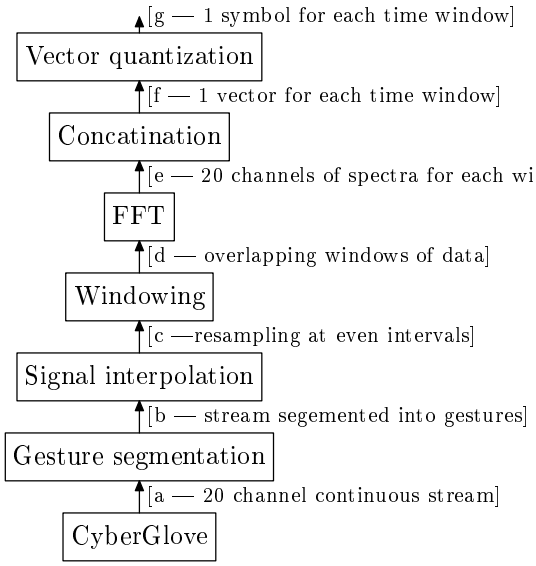


Figure 7.1: Data flow in the gesture-data preprocessor

single-dimensional sequence of symbols which represent features with respect to the entire hand. The specific preprocessing algorithm we chose for this purpose is inspired by work done with HMMs in the speech community—it is vector-quantization (VQ) of a series of short-time fast Fourier transforms (STFFTs) of the signal from the input device.

The preprocessor is designed to encode gestures as sequences of symbols, where each symbol encodes information about the spectral content of the gesture within a given interval of time. Figure 7.1 shows the flow of data through the preprocessor. 20 channels of joint-angle data from the hand are read from the Cyberglove [a]. These data are segmented into separate gestures [b], and resampled by cubic interpolation [c]. Each channel is then broken into a series of overlapping time-windows of 4-8 readings, and each time-window is smoothed with a Hamming function before being passed to the FFT routine [d]. The power spectra of the 20 channels are then concatenated [e] to form a large vector [f]. This vector represents the position and dynamic characteristics of the gesture during the time-window. Each of these large vectors is turned into an observable symbol by a vector-quantizer [g].

The vector quantizer encodes a vector a by returning the index K of a vector $\mathbf{c}_K$ in a set of vectors called the 'codebook' ($\mathbf{c}_K \in \mathbf{C}$) which is closest

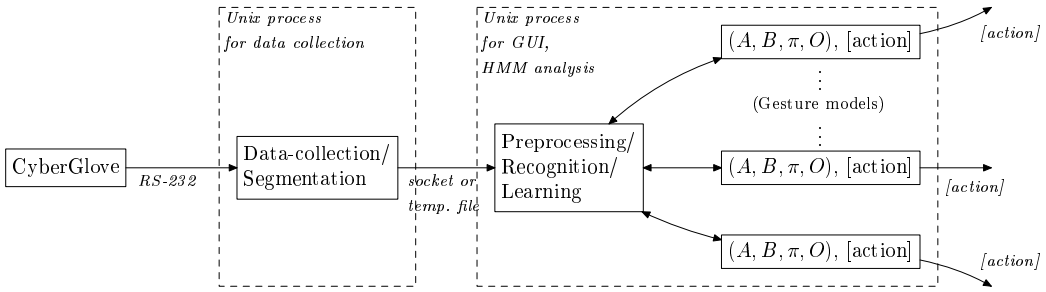


Figure 7.2: Data flow for online learning system

to $\mathbf{a}$ in the L_2-norm sense (i.e., $K = \text{argmin}_k(\mathbf{c}_k \cdot \mathbf{a})$). The codebook is a set of vectors which is believed to be representative of the domain of vectors to be encoded. We generate this codebook using the LBG algorithm [23] on a representative sample of gesture data. Codebook generation is an off-line process.

Because the preprocessor is coded as a data filter, a symbol is sent to the gesture-recognition system as soon as enough data has been read from the glove to generate it. This allows the system to eventually be reconfigured to use HMMs for online segmentation of continuous gestures rather than performing segmentation as a separate step.

This preprocessor is not task specific. It assumes only that all dimensions of the data are related, that the gesture itself is a process which evolves fairly smoothly over time, that the interesting features of the gesture are amenable to clustering in the spectral domain and are consistent in nature over time. It may thus be used without modification for recognizing gestures such as handwriting, facial expressions, or dance motions as well as hand gestures. Note that specially developed, task-specific signal-preprocessors could be expected to perform better for recognition of specific kinds of gestures by leveraging expert *a priori* knowledge of the structure of the gestures, and would be necessary for signals which do not satisfy the assumptions stated above.

7.6 Implementation

Although HMMs are able to automatically segment gestures from continuous streams of data, we perform segmentation before data is sent to the HMM.

This makes the process of learning new kinds of gestures tractable, because the HMMs are unable to segment gestures they have not seen before. Our gesture segmentation procedure is very simple, relying on the hand of the operator to be still for a short time between gestures. We generate a smoothed measure of the overall velocity of points on the hand or of joint angles. When this measure exceeds an 'on' threshold value, we begin recording data as a gesture, and when the measure falls beneath an 'off' threshold, we indicate that the gesture is completed. Another possible tool for segmentation is an acceleration threshold. This is useful for segmentation when the hand does not stop between gestures, and can be combined with the velocity-based segmentation.

Figure 7.2 shows the organization of our system. A Cyberglove is interfaced to a Sun Sparcstation via an RS-232 serial connection, and sends readings of joint angles in the user's hand to a data-collection program at about 10 Hz. Each reading that is part of a gesture is marked with a timestamp, and sent to the gesture-recognition program via either a TCP-based UNIX socket or a temporary file. The socket connection is used for normal operation, and the temporary file is used for creating databases of gesture data for off-line generation of vector-codebooks, and for automated testing and tuning of system parameters.

After the data for a gesture is preprocessed by the algorithm specified in Section 7.5, it is evaluated by all HMMs for the recognition process, and then used to update the parameters of the proper HMM. For recognition of hand gestures, we used 5-state Bakis HMMs. A Bakis HMM is one where the system is restricted to only move from a given state to the same state or one of the next 2 states. A 5-state Bakis HMM may move from state 1 to states 1, 2, or 3, and from state 4 to state 4 or 5 (there is no state 6). This restriction encodes the assumption that the gestures we are classifying are in general a simple sequence of motions, and non-cyclical in nature. Using this HMM structure, we performed tests to determine the ability of the system to accurately recognize gestures and to optimize the parameters of the preprocessor.

The data-collection program for the Cyberglove was written in C, and the interactive HMM recognition/training/GUI program was written in a combination of C, Tk, and Scheme. Although matrix operations are coded in C for speed, the outer loops of Baum-Welch, gesture-recognition, and all other code is currently executed in interpreted Scheme. Even with the majority of the code written in Scheme, gesture-recognition and update of

the proper HMM occurs in a fraction of a second on a Sparcstation.

7.7 Confidence measure

We defined a simple evaluation function to examine the classification power of our algorithm. The function indicates misclassifications and their severity, as well as the system's confidence in its correct classifications.

Our evaluation function is

$$V = \log_{10}\left(\sum_i P_{E_i}/P_C\right), \tag{7.1}$$

where P_C is the probability that the observation sequence would be created by the correct gesture model, and P_{E_i} is the probability that the gesture symbols would be created by the ith incorrect model. Therefore, if $V < 0$, we have correct classification, and if $V > 0$, the gesture is incorrectly classified. If $-1 < V < 1$, the classifier should indicate that its classification is suspect. If $V < -2$, the system has made the correct classification and is very confident of the classification.

We plotted the performance of the system using the following procedure:

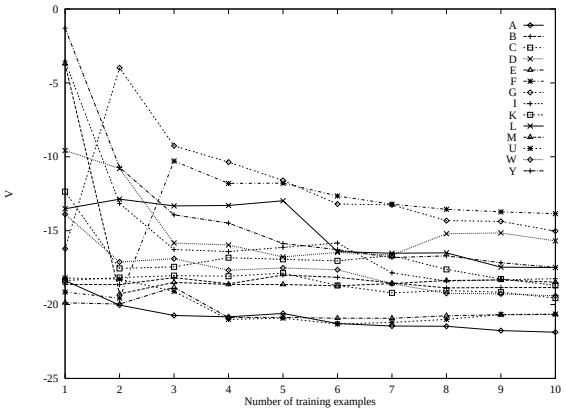
1. Train each gesture model with one example.

2. Test the classification of 20 test examples of each kind of gesture (the models are never trained on these examples).

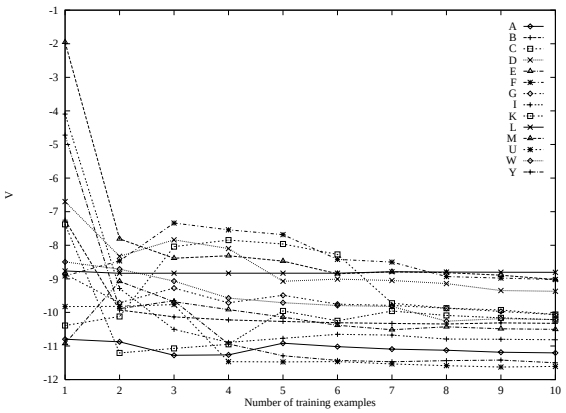
3. Plot the average value of the V verses the number of examples of gestures each model has seen.

This procedure allows us to judge the performance of the algorithm with respect to learning rate and overall confidence. It also allows us to test the effect of varying parameters such as the number of vectors in the codebook (the number of observable symbols), the size of the STFFT windows, and the resampling time step for the signal interpolator.

For our test, we picked 14 letters from the sign language alphabet which were amenable to VQ clustering and unambiguous with respect to general hand orientation, since we did not using the Polhemus 6D position/orientation sensor for the hand. The letters we used are A, B, C, D, E, F, G, I, K, L,



(a) 128 symbols, 10 Hz resampling, 4 point data windows



(b) 64 symbols, 20 Hz resampling, 8 point data windows

Figure 7.3: Evaluation of gesture classification process

M, U, W, and Y. The final positions for two of these gestures are shown in Figure 3.3 on page 49.

Two plots from our tests are shown in Figure 7.3. They demonstrate that the system is very reliable for both these sets of preprocessing parameters. The average value of V in all cases is less than zero, indicating reliable classification. Measuring percentage of classification errors, the trial of plot (b) had 1% error after 2 examples and less than 0.1% error after 4 examples. The trial of plot (a) had 2.4% error after two training examples, and made no classification errors after seeing 6 examples. These results are comparable to the results from Section 3.5, but both methods work so well on the data that their recognition abilities cannot be adequately compared using this data set.

The iterative training of the HMMs usually results in models which are close to the quality of batch-trained HMMs when we compare the likelihood that the training data would be generated by the models. In a few cases, however, batch training did much better. This is probably the result of early training examples biasing an HMM toward a poor local maximum. Fortunately, batch training from a random HMM is a rapid process, and can be easily used to try to improve HMM models during a couple seconds while the system is not doing real time recognition. Our results show that this is not generally necessary, however, as iteratively trained HMMs work well enough in classifying gestures.

7.8 Discussion

We have demonstrated an efficient and reliable system for online learning and recognition of gestures. Online learning of gestures in an interactive system might be used to make cooperation between robots and humans easier in applications such as teleoperation and programming by demonstration.

The number of gestures we can accurately classify is currently limited by the number of observable symbols the preprocessor generates. Thus systems which perform off-line training can recognize a larger gesture vocabulary. Fels and Hinton [19], for example, use neural networks trained off-line to recognize 66 root words, each with up to 6 endings. A future direction in which the work of this chapter could be taken forward is to increasing the potential size of the gesture vocabulary of our online system by having the preprocessor generate 5 dimensions of symbols (one for each finger) as input to a multidimensional HMM.

Due to the expense of a Cyberglove and the awkwardness of its use in real situations, recognition by visual tracking of a person's hand [57, 71] may be more practical for many applications. Since this work was originally published [39], HMMs have recently been used for online adaptive recognition of arm motions using a vision system [82].

It is interesting to compare the results from this chapter to the results from the spline smoother model in Section 3.5. Since both methods did so well on this set of letter-signing data, it would be helpful to have a more difficult data set to better characterize their relative strengths for recognition. Nevertheless, we can say that the online learning capability of the HMM-based approach is a large advantage. It is possible that similar capabilities might be developed for the spline smoother model, however, and the spline smoother model would then have the advantage that it requires no off-line process for codebook generation, and has no need of sophisticated preprocessing of the performance data.

The most interesting difference between the smoothing spline and hidden Markov model methods is the nature of the resulting models. Because the spline smoother model is an estimate of a best-fit trajectory with associated variance values, it can be used for high-quality smooth animations of the learned motion. The HMM model, on the other hand, is far too opaque for effective use in animation. The fact that the HMM model is not tied to a Gaussian distribution makes it better suited for recognizing motions which vary in ways including probabilistic branching and looping in configuration space.

Chapter 8

Conclusion

8.1 Contributions

The central idea of this thesis is the formulation of action learning as a dimension reduction problem. The most important theoretical contribution, however, is a new nonparametric method for fitting trajectories to phase space data. In addition to these, the thesis contributes a comparison of previously existing methods which may be applied to dimension reduction for action learning. Finally, a large amount of new software was written during this thesis, a subset which is being distributed freely on the Internet.

8.1.1 Formulation of action learning as a dimension reduction problem

In this thesis, action learning is formulated as the characterization of the lower-dimensional manifold or constraint surface, within the much higher-dimensional state space of space of possible actions, upon which human actions states tend to lie during performance of a given task. While it is useful to characterize the regions of configuration space visited during typical example performances, the thesis shows that it is even more useful to characterize the regions of phase space visited, as this comprises a more complete description of the performance's state space.

Once such a description has been learned, it is a valuable tool for recognizing and classifying particular observed performances, for performing motion analysis, for skill transfer applications, and for creating computer animations.

8.1.2 Nonparametric methods for trajectory fitting in phase space

This thesis argues that the "best-fit trajectory" is the best one-dimensional model for a set of action data. To build such a model from a set of position and velocity data sampled from multiple examples of task-execution, this thesis develops two new mathematical tools:

- a spline smoother is derived for fitting phase space data sets, and
- the principal curves algorithm is adapted to use this new smoother for fitting trajectories in phase space without an *a priori* parameterization.

Together, these provide a practical solution for an important general problem: *What is the best-fit path through a sampled vector field?*

The spline smoother fits a curve which balances a measure of its smoothness against the errors in its approximation of a given set of parameterized position and velocity data. The optimal curve is found by solving a linear system. When used with this smoother in phase space, the principal curves algorithm is transformed from a method for nonlinear regression into a valuable tool for modeling the motion of a dynamic system.

8.1.3 Examination of existing parametric and nonparametric methods for action learning

The use of parametric and nonparametric methods for mapping raw performance data to and from a lower-dimensional feature space is demonstrated on an example data set, and the nonparametric methods are shown to present greater promise.

Linear parametric models such as PCA, while easy to interpret, are limited in the range and complexity of actions they can represent efficiently. Nonlinear parametric models such as NLPCA are capable of efficiently representing a more general range of actions, but result in opaque and often suboptimal mappings.

Nonparametric methods combine a set of local models which are individually easy to analyze and interpret into a global model which is capable of representing complex actions. The form of the local models can be designed explicitly for the purpose of modeling action data, and as we have seen, can be designed to work simultaneously with both position and velocity data.

8.1.4 Software

Several new software packages were developed in the process of this research, some of which are being freely distributed at
`http://www.cs.cmu.edu/~chrislee/Software/`
The most significant are listed here.

CMATLIB The CMATLIB package is a library for linear algebra. It has functions for building and managing memory for matrices, and has an easy-to-use interface to the same high-performance matrix libraries BLAS and LAPACK which Matlab uses to perform its computations. The BLAS and LAPACK interfaces are generated by a Perl program I wrote called Fortranwrap, which automatically generates glue-code and macros for calling Fortran code from C. Another package I wrote, G-Wrap, is used to create an interface between CMATLIB and the RScheme Scheme interpreter written by Donovan Kolbly. This allows RScheme to be used as a matrix computation environment like Matlab, but with a much more powerful programming language.

G-Wrap I wrote G-Wrap to automatically generate glue-code for calling C functions from a Scheme interpreter. Currently, both the Guile and RScheme interpreters are supported. G-Wrap is also currently being used by Rob Browning at the University of Texas to add a Guile-based scripting interface to GNUCash, a free personal finance program for Unix-compatible operating systems. I recently turned over maintenance of G-Wrap to Rob Browning, and the program has been packaged for an upcoming version of the Debian GNU/Linux operating system.

RSK The Robot Scheme Kernel is a virtual machine which is capable of executing Scheme code in real-time systems. I wrote it as part of the high-level control system for the $(DM)^2$ and $(SM)^2$ robots in the Space Robotics Laboratory, and it will eventually be suitable for directing the high-level operation of a robot which uses low-level actions skills learned by methods such as those presented in this thesis. This system, described more fully in Appendix A, is able to represent high-level robot operations (or simulations) with a special version of the Scheme programming language which has been extended to represent both temporal as well as logical relationships between individual operations. RSK is not yet being distributed.

8.2 Future research

The research presented in this thesis leaves open many avenues for future work.

8.2.1 An investigation of cross-validation for principal curves in phase space

The spline smoother presented in Chapter 4 uses two parameters, p_1 and p_2, which weigh the relative penalty for approximation error in position and velocity respectively against the smoothness of the model curve. Chapter 5 demonstrates how to select a ratio of p_1/p_2 which satisfies a given criterion for disambiguating the projection of sampled data points with similar positions but different velocities. This still leaves us with one free weighting parameter for building our principal curve fit, however.

This parameter, weighing overall approximation error against the smoothness of the model curve, is useful for manually tuning the approximating curve to satisfy other task-dependent criteria. However, we might prefer to use cross-validation to chose the parameter automatically from the training data. Although cross-validation is problematic for the conventional principal curves algorithm, sometimes resulting in severe over-fitting [28], it may not be a problem for principal curves in phase space.

Longer model curves can fit a given set of data better in position space while not significantly increasing the smoothness penalty and not lowering the position-space cross-validation error. This tends to result in a model which does not resemble the original motion trajectories, however. These long model curves will not match the directional velocities of the original trajectories at the sample points, so they are not a likely result of cross-validation in phase space.

To study the effectiveness of cross-validation for principal curves in phase space, the relationship between cross-validation error in position space and velocity space should be related theoretically to the effect of the scaling of the parameterization used for the conditional-expectation step of the principal curves algorithm (Section 5.5). Without a derivation for efficient generalized cross-validation for spline smoothing in phase space, cross-validation for the principal curve will be significantly slower than determination of the principal curve without cross-validation.

8.2.2 Generalized cross-validation for smoothing splines in phase space

Conventional spline smoothers can automatically determine the best smoothing weight based on generalized cross-validation, and this determination can be performed in linear space and time. It may be possible to extend this result to the spline smoother in phase space presented in Chapter 4, but this may require a more sophisticated re-derivation of the smoother.

Such a result would make cross-validation for the principal curve algorithm, discussed above, highly efficient.

8.2.3 Building multidimensional action skill models around the "skeleton" of the best-fit trajectory

Although it is possible to build such a multidimensional model using the method described in Section 3.7, the difficulty is building a model where the second or third parameterizing variable has a consistent, understandable meaning over the entire course of the best-fit trajectory. The work by Tibshirani [76] on redefining the principal curve based on mixture models is a useful starting point, as well as the work of LeBlanc and Tibshirani on adaptive principal surfaces [37], and the work of Tipping and Bishop on mixtures of probabilistic principal component analysis [77].

8.2.4 Reducing the number of knots in the spline models of best-fit trajectories

Spline smoothers can be defined which use less knots than training points [16]. It should be possible to adapt the smoother derived in Chapter 4 to work on this principle, and to modify the principal curves algorithm described in Chapter 5 to adaptively reduce the number of knots in its representation of the principal curve. This should make the models produced by this method more compact and easier to work with, and reduce the computational resources required for using them.

8.2.5 Reducing the computational complexity of the phase space smoother to linear space and time

Section 4.4 outlines how the spline smoother derived in this thesis could be implemented to run with time and memory requirements which are linear in number of points being smoothed. This has not been implemented yet, however, and the nature of the approximations necessary for such a solution need to be researched further.

8.2.6 Characterizing the complexity of low-level action skills

The relationship between the complexity of low-level action skills and their implicit, underlying dimensionality is an interesting direction for further study. The methods demonstrated in this thesis should be very effective for this research.

8.2.7 Building usable robot skills from action models

Action modeling should be beneficial for building usable representations of robot skills for tasks which are difficult but largely feed-forward in nature. This work could be done in the context of the high-level control framework presented in Appendix A.

8.3 Publications from thesis work

Some of the material in this thesis has been published previously in the peer-reviewed literature, in the following papers:

- Christopher Lee, Yangsheng Xu. Trajectory fitting with smoothing splines using velocity information. *IEEE International Conference on Robotics and Automation*, San Francisco, CA. April 2000. pp. 2796–2801.

- Christopher Lee, Yangsheng Xu. Message-based evaluation for high-level robot control. *Journal of Intelligent and Robotic Systems*, Vol. 25, June 1999. pp. 109–119.

- Christopher Lee, Yangsheng Xu. Reduced-dimension representations of human performance data for human-to-robot skill transfer. *Proceedings of the 1998 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Victoria, B.C. vol. 3, pp. 1956–1961.

- Christopher Lee, Yangsheng Xu. Online, interactive learning of gestures for human/robot interfaces. *1996 IEEE International Conference on Robotics and Automation*, Minneapolis, MN. vol. 4, pp. 2982–2987.

Appendix A

High-level robot control for action execution

In Section 1.1, we discussed how action-skills are decomposed into high-level strategy and low-level components. In this appendix, we present an architecture for implementing high-level strategies which could be useful for executing these action skills. We also present a kind of interpreted scripting language based upon the general-purpose Scheme programming language, but which is evaluated using a novel method of code execution called "message-based evaluation" (MBE). This scripting language allows high-level task strategy to be described in terms of both logical and temporal relationships of individual low-level actions. The original motivation for this work is to build a high-level control system which can safely meet the needs of dynamically reconfigurable real-time software system (e.g., for control of manipulator robots running the Chimera operating system), but the general approach is also applicable for animation, simulation, and for building interactive video game engines.

A.1 Dynamically reconfigurable real-time software

A major goal of real-time operating systems like Chimera is to enable sensor-based control applications to be built from libraries of reusable software modules. For this purpose, they provide standard interface specifications for implementing reusable real-time software modules, and a library of functions for building and using configurations of these modules [72]. A well-written

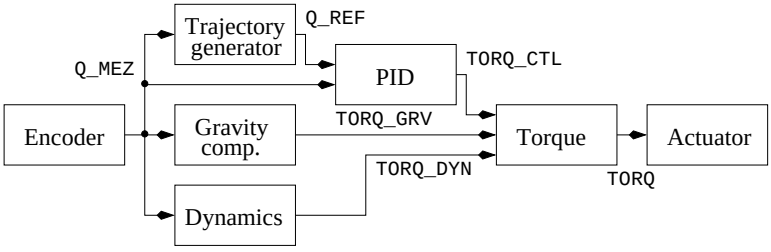


Figure A.1: Example configuration of real-time modules

and debugged library of real-time modules thus facilitates rapid development of reliable sensor-based control systems. In Chimera, these modules or "port-based objects," typically cycle at some fixed frequency and communicate their inputs and outputs through a global state-variable table. A typical configuration of real-time modules for controlling a robot manipulator arm is shown in Figure A.1.

A real-time software module is reusable only if it is sufficiently independent of the specific details of the different applications for which it is used. Therefore, an essential focus of developing reconfigurable software is keeping task-level details out of the reusable modules. For example, a PID control module should not care whether it is controlling a joint-angle in a robot-arm, a Cartesian tool-coordinate, or a feature-coordinate in a visual-servoing process. As a result, reusable software modules are most useful for the lowest-level tasks within a robot software architecture—those which do not require explicit knowledge of the task-level details of the robot's operation.

In robotic applications, this specialization results in a need for a higher-level layer of the software architecture which can direct the use of the reusable modules for the purpose of satisfying the robot's task-level requirements. This layer typically initializes all the reusable modules when the robot is booted, sends messages to modules telling them to modify their working parameters (e.g., adjusting controller gains, or sending via-points to a trajectory-generator module), and receives messages from modules to learn of significant events in the operation of the robot (e.g., significant qualitative changes in the readings of robot sensors). Most importantly, when the qualitative nature of the robot's task changes significantly, the high-level layer of the architecture must change the configuration of reusable-modules to match the needs of the task. For instance, when a manipulator arm is moving in a

Cartesian control mode and contact is sensed at the end-effector, the robot should switch to a force-control or impedance control configuration. Such a "dynamic reconfiguration" typically involves turning off some modules and turning others on. This must be done on the fly, changing the active control law without disturbing the timing or effectiveness of the overall system. In cases such as the switch from Cartesian to compliant control, this must be performed without delay to avoid unacceptable forces at the end-effector. It is thus essential for the safety of the robot and its surroundings that the high-level controller react to important task-level events in hard real-time.

A.2 General purpose scripting languages for high-level control

Several strategies have previously been used for managing such dynamically reconfigurable subsystems, including on-line state machines, and separate high-level programs running on host workstations. In several Chimera-based robot architectures [18, 46], the high-level process reconfigures the real-time subsystem based on an on-line state machine interpreter responding to messages sent from modules in the reconfigurable subsystem. ControlShell [65] for the VxWorks operating system also uses a state machine for managing dynamically reconfigurable real-time subsystems. Implementing interpreters for state machines is fairly straightforward, and state-machines are well understood and amenable to design through graphical user interfaces. Synchronous languages such as Esterel [8] which is used in the ORCCAD [70] robot application development system, may also be useful for this purpose.

Another approach for managing reconfigurable subsystems of real-time control modules is represented by Onika, a visual programming environment for designing control systems as configurations of modules, and for controlling the reconfiguration of these control systems during execution of Chimera applications [24]. Onika's visual programming language is limited in terms of the algorithms it can represent, however, and because it manages dynamic reconfiguration of the low-level Chimera modules from a non real-time workstation, it is inappropriate for managing reconfigurations which must occur in hard real-time.

In this appendix, we present the approach of using an embedded interpreter for a general-purpose programming language for high-level control of

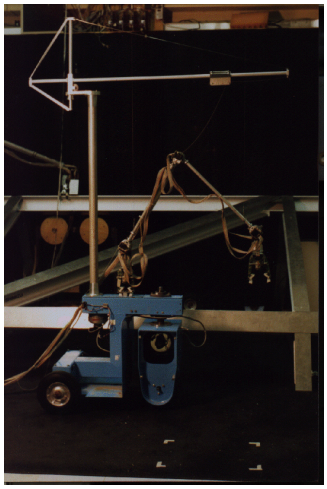


Figure A.2: $(DM)^2$

reconfigurable subsystems. This approach has a number of advantages:

- sufficient expressive capability for most high-level task specifications can be guaranteed by using a suitably powerful interpreted language,
- a general-purpose programming language can specify robot-tasks using traditional structured-programming or object-oriented methods,
- hard real-time response times to events can be achieved through careful implementation of the embedded interpreter, and
- an interpreted language (in source-code form or compiled to virtual-machine code) is a convenient way for remote operators to send general-purpose commands to a robot while it is running (e.g., for remote teleoperation).

Development of a robot architecture for the Dual-use Mobile Detachable Manipulator, $(DM)^2$, originally motivated our adoption of this strategy. $(DM)^2$, shown in Figure A.2, is a mobile robot consisting of a mobile base and a detachable manipulator arm [86]. The manipulator is a symmetric 5-DOF arm with a gripper at each end, and may either grasp the mobile base with one gripper to become a mobile manipulator system, or detach from the base and walk hand-over-hand by grasping special handles with its grippers.

The software for this robot is built upon the Chimera 3.2 operating system. It uses configurations of real-time modules for controlling the motion of the mobile base and manipulator arm, and requires the ability to dynamically change these configurations as the robot changes hardware configurations (i.e., from mobile manipulator to walking arm) or performs different tasks (e.g., switching from walking to grasping and then lifting an object).

$(DM)^2$ requires high-level software which can not only perform the necessary reconfigurations of its low-level software in hard real-time, but which is intelligent enough to manage the overall operation of a mobile robot. Some examples of what the high-level software for $(DM)^2$ must do include: using an internal map of its environment to keep track of the angle of inclination of the surface the arm is walking on (to adjust the gravity vector for calculating gravity-compensation torques in the joints); allowing multiple attempts at grasping handles or the mobile base before admitting failure (possibly perturbing the set-point slightly each time); switching between different controllers during different subtasks (i.e., using an adaptive controller when picking-up an object of unknown mass); following procedural descriptions of arm motions for walking and mobile base movements from on-line or off-line path-planners; and accepting commands from a remote operator. In all these cases, we need to specify alternative actions to be taken if any individual operation fails.

In developing a software architecture for $(DM)^2$, we initially built an interpreter for a simple, custom-designed scripting language to manage the dynamic reconfigurations of the low-level real-time subsystem [38]. After some experience programming this system, however, we decided that a more powerful, general-purpose language would be better suited to our needs and chose Scheme. Scheme is a Lisp dialect with a concise specification for which small, efficient interpreters can be written. It is also a powerful language commonly used for writing artificial-intelligence algorithms and for programming in a functional style [1]. It is simple to use for writing descriptions of the operations necessary for high-level control of our robot, and we felt it easier to write more complex approaches to such task-level needs with a general-purpose programming language than with a state-machine description. Scheme, in particular, has continuations as first-class objects, and these play an important role in our method of executing high-level robot code (as discussed in Section A.3). We thus developed the Robot Scheme Kernel (RSK), which can respond to events in real-time, and which works cooperatively with real-time code written in a system programming language

```
; Move arm in direction dir with speed speed until contact is detected at
; the end-effector, but stop if the motion lasts longer than 5 seconds.
(define (move-to-contact dir speed)
 (race
  (lambda () (move-arm dir speed))
  (lambda () (detect-contact) 'contact)
  (lambda () (pause 5.0) 'no-contact)))

; If moving the arm achieves contact, switch to a configuration
; for compliant control
(case (move-to-contact <down> <slow>)
 ((contact) (start-compliant-control))
 (else (GUI:error "Contact was not detected")))
```

Figure A.3: Robot code for a guarded move

(such as C) within an existing multi-threaded, multiprocessor robot architecture. RSK satisfies these requirements through real-time memory management strategies and a novel execution model which is designed specifically for controlling robots.

A.3 Message-based evaluation

Task-based management for supervision and dynamic reconfiguration of the low-level subsystem requires a very different style of coding than that for which traditional system programming languages are designed. Two of the main challenges in writing such high-level robot code are that

1. there may be a high degree of functional parallelism in the normal operation of the robot's hardware, and

2. its operations involve physical processes that occur much more slowly than the elementary software operations which are used to manage them.

General-purpose programming languages (especially system programming languages such as C), excel at data manipulation and logic-based control of execution flow. However, they are less appropriate for specifying temporal

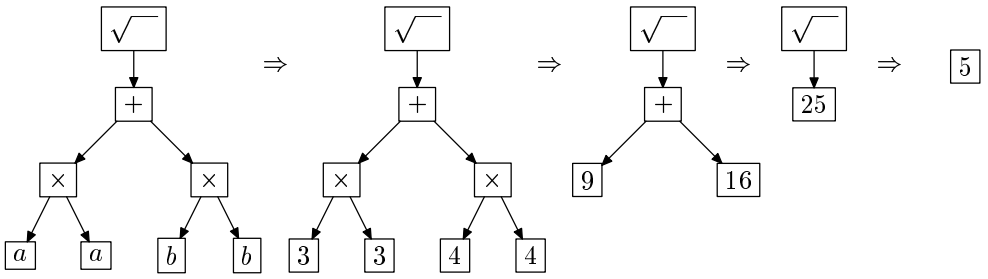


Figure A.4: Evaluation by graph reduction

relationships between subexpressions such as those demonstrated by the code in Figure A.3 (the details of which will be discussed later in this section). RSK executes code like this by employing a method we call *message-based evaluation* (MBE), which is designed to allow the structure of high-level robot control code to reflect the structure of the tasks whose execution it supervises.

In functional programming languages, the evaluation of an expression is often modeled as a process of "graph-reduction." An expression is an acyclic graph, and evaluation is a process whereby the graph is simplified in a step-by-step fashion to a single node representing the value of the expression. For instance, the evaluation of the expression $\sqrt{a^2 + b^2}$, coded in Scheme as `(sqrt (+ (* a a) (* b b)))`, could be represented (for $a = 3, b = 4$) as the graph simplification shown in Figure A.4.

This evaluation could be accomplished by a conventional stack-based computation such as (PUSH a, PUSH a, APPLY '$*$', PUSH b, PUSH b, APPLY '$*$', APPLY '+', APPLY 'sqrt'). Such a method is efficient for conventional computers and does not require a literal graph-based representation of the expression to work. A very different evaluation method could also be used—one based on message-passing between nodes of an explicit graph representation of the expression. In such a method, each node of the graph is represented by an object which may receive messages from and send messages to its parent and child nodes, and which knows how to compute its own value when given the value of each of its child nodes. The evaluation process is triggered by sending a message to the root node commanding it to evaluate the graph. The evaluation occurs through each node implementing the following procedure:

1. For each child node (if you have any), send a message to that child

telling it to evaluate itself and to reply with a message containing the result of this evaluation.

2. Once all child nodes have replied, evaluate yourself and return the result.

In the case of the expression graphed in Figure A.4, the "variable nodes" a and b immediately look-up their values and send them to the "multiplication nodes" which in turn calculate their products and send these to the "addition node", which sends the sum of these products to the "square-root node", which returns the final result (5).

Although this method is obviously inefficient for the example computation, it has some interesting characteristics:

- For each node in the graph which has more than one child node, the order of evaluation of the child nodes is unspecified, and the child nodes could even compute their results in parallel.

- If the underlying messaging system were to support the necessary communication (see Section A.4), each node could be on any CPU of a multiprocessor system or even on a separate computer. The evaluation process would be exactly the same in these cases.

If we extend the evaluation process so that each node controls when and if each of its child nodes is evaluated, we can build expressions which explicitly represent temporal (as well as logical) relationships between the execution of their subexpressions. We can, for instance, implement "async nodes" which evaluate their child nodes sequentially (equivalent to the stack-based evaluation strategy); "conditional nodes" which evaluate some child nodes depending on the results returned by others (e.g., a node implementing an "if-then" operation); "sync nodes" which evaluate all their child nodes in parallel; and even "race nodes" which tell all their child nodes to evaluate themselves and return the result of the first child node to finish (aborting the evaluation of the other child nodes). Figure A.3 shows an example of the use of a race node. Writing an equivalent expression in a system programming language would be much more difficult, typically requiring the use of explicit polling mechanisms, or a combination of a state-machine description and a state-machine implementation.

MBE combines a standard expression evaluation technique, similar to the stack-based method, with an implementation of the message-passing method.

This results in an interpreter with both the efficiency of the standard method and the ability of the message-passing method for executing code representing explicit temporal relationships between subexpressions. MBE extends the model of the message-passing evaluation architecture by specifying that a node must either return its result to its parent node "immediately" or tell its parent node that it is "not done yet." The interpreter can thus use the standard evaluation method to evaluate an expression until a call to a function within the expression raises a "not-done" exception. When this exception is raised, the interpreter creates a child-node object representing the incomplete function call, and a parent-node object representing the remainder of the (as yet unfinished) computation. At the appropriate time, the child-node can cause the interpreter to resume the evaluation by sending its value in a message to its parent node. Such an event is typically triggered by a message sent from another branch of the evaluation tree or from a low-level module indicating that a gripper has been closed, a manipulator motion completed, or an obstacle detected.

When MBE switches from its standard evaluation strategy to its message-passing strategy, it need only create a single object to represent the remainder of the incompletely reduced graph rather than an explicit graph representation of the entire unfinished computation. This is because the object representing the graph above the child node contains a *continuation*. A continuation is a representation of the entire default course of a given computation, and as such is a full representation of the incompletely reduced graph of an expression. Continuations are typically used to implement coroutines, threading, and throw-catch style exception handling. In Scheme, continuations are first class objects. If a Scheme implementation is based on a "continuation chain," or a chain of "incomplete continuation" objects, rather than on a C-style stack, then creating such a continuation object is roughly equivalent in speed and memory cost to a function call. This allows the switching between the two styles of evaluation to be very efficient. Thus, MBE works quickly and cheaply for interpreting the Scheme language.

Note that while this model of evaluation allows parallel operations to be represented by multiple "not-done" child-nodes below a parent node, this does not mean that MBE itself is performing any kind of multi-threaded operation. The "not-done" nodes represent processes occurring outside the main thread of the interpreter, typically in the reconfigurable modules. These processes may include things such as grippers opening and closing, and manipulator arms executing motion commands. A "not-done" node may also

be waiting for command-messages from the teleoperation console, or for a panic-message from anywhere in the robot architecture indicating that the robot needs an immediate shutdown for safety reasons. Thus, RSK is usually waiting for messages rather than running Scheme code, and its job is to react to these messages without delay. The state of the current evaluation tree indicates what actions the high-level system should take when it receives messages from the reusable modules or the host workstation.

A.4 Messaging infrastructure

In discussing the process of code evaluation by message-passing, we noted that if the underlying messaging system were to support the necessary communication, each node in the graph of the expression could be evaluated on a different CPU or computer. This motivated us to design a system for message-passing that is optimized for speed in the local delivery of messages, but which is also able to use whatever operating-system communication mechanisms are available for passing messages to and from remote domains.

Each RSK message contains a "To" address for directing message delivery and a "From" address so that it may be easily replied to. Each address has a local component and a domain name. Two nodes are in the same "domain" if they are able to use valid memory pointers to one another for their local addresses. Message passing within a domain is thus an inexpensive operation, consisting of adding a message to the priority queue "in-box" of its recipient, and adding the recipient node to a prioritized list of nodes in the domain which have pending messages. A function written in C may also register a local address with the messaging system, allowing it to receive messages via a call-back mechanism. Among other things, this allows high-level Scheme code to send parameters such as gains and via-points to low-level control modules in a reconfigurable subsystem. If a message is sent to a node in a different domain, the message is converted from its local representation (a Scheme object) to a binary representation which may be sent via operating-system communication mechanisms to a process in the appropriate destination. This remote process converts the message back to its original form and performs the local delivery. Since Chimera is a multiprocessor operating system, and because it is hosted by a UNIX workstation, RSK's messaging mechanism enables it to deliver commands and information between CPUs of the real-

time computer, and to any computer on the host-workstation's network (e.g., the Internet).

The messaging infrastructure thus allows RSK interpreters running in each CPU of the real-time computer to cooperate with one another, and provides a mechanism for cooperation between the high-level control process of the real-time computer and off-line resources such as remote teleoperation consoles and planners. An additional benefit is that this communication mechanism allows RSK to offload some of the work of Scheme interpretation to the host workstation. The host workstation can parse Scheme code and compile it to a virtual machine-code representation (the compiler is actually a Scheme program running on a UNIX implementation of RSK), and then send a message containing the resulting virtual-machine code to the real-time computer for execution. This allows the high-level process on the real-time computer to focus its resources on managing the operation of the robot rather than on parsing and compiling Scheme code.

A.5 Memory management

In Lisp-like languages, explicit management of dynamically allocated memory is infeasible. These languages rely on "garbage collection", which is a mechanism for automatically determining what memory a program is no longer using and recycling it for other use. Because this determination involves a global analysis of the interpreter's memory pool, most commonly used garbage collection algorithms require that the interpreter be stopped during the collection in ways which are incompatible with real-time operation. Rees and Donald [60] use an embedded Scheme interpreter for control of small mobile robots. This interpreter is appropriate for their work, but garbage collection pauses make it inappropriate for use in real-time applications. If a robot were unable to react quickly to end-effector contact during a guarded move because the high-level was paused for garbage collection, damage to the robot and its environment could result.

In the initial version of RSK, we addressed this problem by simply using reference-counting for garbage collection. This is a local strategy rather than a global strategy for analysis of memory usage, and can be done in small increments which preserve the overall responsiveness of the interpreter. Although this strategy can be made to work for managing the interpreter's own use of data structures [21], and though this strategy has been successful

for controlling our robot without memory leaks, reference-counting cannot reclaim data-structures which point to themselves even when they are not referenced by any data structures in use by the interpreter. Fortunately, there are now methods which allow garbage collectors to run efficiently on stock hardware in hard real-time [83], in addition to those which run on specialized hardware [55]. To allow programmers to use RSK for code which may use cyclic data structures, and to simplify the interpreter, we are implementing a real-time garbage collector based on the "write-barrier" strategy used in Wilson and Johnstone's real-time collector [84].

A.6 Conclusion

We have described dynamically reconfigurable subsystems for sensor-based control of robot systems, and presented the Robot Scheme Kernel (RSK), an embedded Scheme interpreter designed for high-level management of these subsystems. To allow RSK to evaluate Scheme expressions which represent temporal relationships between their subexpressions, we have developed "message-based evaluation" (MBE). MBE allows the structure of high-level robot control code to reflect the structure of the robot's intended task performance. The messaging infrastructure underlying MBE helps to simplify operation in multiprocessor environments, providing a mechanism for the robot to interact cooperatively with remote processes such as teleoperation consoles and off-line planners. Real-time garbage collection strategies allow RSK to respond in hard real-time to important events during the course of robot operation. Future work will focus on a more formal characterization of this method and its use in robot systems, and a more complete comparison between it and other currently available methods for high-level robot control.

Bibliography

[1] Harold Abelson et al. Revised^4 report on the algorithmic language Scheme. *ACM Lisp Pointers IV*, 4(3), July-September 1991.

[2] Colin Archibald and Emil Petriu. Skills-oriented robot programming. In F.C.A. Groen, S. Hirose, and C.E. Thorpe, editors, *Proceedings of the International Conference on Intelligent Autonomous Systems IAS-3*, pages 104–15, Pittsburgh, PA, 1993. IOS Press.

[3] Nachman Aronszajn. Theory of reproducing kernels. *Transactions of the American Mathematical Society*, 68:337–404, 1950.

[4] Christopher G. Atkeson, Andrew W. Moore, and Stefan A. Schaal. Locally weighted learning. *Artificial Intelligence Review*, 11(1-5):11–73, 1997.

[5] Christopher G. Atkeson, Andrew W. Moore, and Stefan A. Schaal. Locally weighted learning for control. *Artificial Intelligence Review*, 11(1-5):75–113, 1997.

[6] Yusuf Azoz, Lalitha Devi, and Rajeev Sharma. Reliable tracking of human arm dynamics by multiple cue integration and constraint fusion. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition (CVPR)*, pages 905–910, Santa Barbara, CA, June 1998.

[7] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press Inc., New York, 1995.

[8] Frédéric Boussinot and Robert De Simone. The Esterel language. *Proceedings of the IEEE*, 79:1293–1304, 1991.

[9] Christopher Bregler and Stephen M. Omohundro. Surface learning with applications to lipreading. In J. D. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems*, volume 6, pages 43–50. Morgan Kaufmann, San Mateo, CA, 1994.

[10] Richard H. Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal of Scientific Computing*, 16(5):1190–1208, 1995.

[11] William S. Cleveland. Robust locally weighted regression and smoothing scatterplots. *Journal of the American Statistical Association*, 74(368):829–36, December 1979.

[12] William S. Cleveland and Clive Loader. Smoothing by local regression: Principles and methods. Technical Report 95.3, AT&T Bell Laboratories, Statistics Department, Murray Hill, NJ, 1994.

[13] Peter Craven and Grace Wahba. Smoothing noisy data with spline functions—estimating the correct degree of smoothing by the method of generalized cross-validation. *Numerische Mathematik*, 31(4):377–403, 1979.

[14] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2:303–314, 1989.

[15] James W. Davis. Recognizing movement using motion histograms. Perceptual Computing Section 487, MIT Media Laboratory, 1999.

[16] Carl De Boor. *A Practical Guide to Splines*. Springer-Verlag, New York, 1978.

[17] D. Dong and Thomas J. McAvoy. Nonlinear principal component analysis-based on principal curves and neural networks. *Computers & Chemical Engineering*, 20(1):65–78, January 1996.

[18] Alexander Douglas and Yangsheng Xu. Real-time shared control system for space telerobotics. *Journal of Intelligent and Robotic Systems: Theory and Applications*, 13(3):247–62, July 1995.

[19] S. Sidney Fels and Geoffrey E. Hinton. Glove-talk: A neural network interface between a data-glove and a speech synthesizer. *IEEE Transations on Neural Networks*, 4(1):2–8, January 1994.

[20] Ildiko E. Frank and Jerome H. Friedman. A statistical view of some chemometrics regression tools. *Technometrics*, 35(2):109–135, May 1993.

[21] Daniel Friedman and David Wise. Reference counting can manage the circular invironments of mutual recursion. *Information Processing Letters*, 8(1):41–45, Janary 1979.

[22] Jerome Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–26, September 1977.

[23] Allen Gersho. On the structure of vector quantizers. *IEEE Transactions on Information Theory*, IT-28(2):157–166, 1982.

[24] Matthew Gertz, David Stewart, and Pradeep Khosla. A software architecture-based human-machine interface for reconfigurable sensor-based control systems. In *Proceedings of 8th IEEE International Symposium on Intelligent Control*, pages 75–80, Chicago, IL, August 1993. IEEE.

[25] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences. The Johns Hopkins University Press, Baltimore, MD., third edition edition, 1996.

[26] Blake Hannaford and Paul Lee. Hidden Markov model analysis of force/torque information in telemanipulation. *The International Journal of Robotics Research*, 10(5):528–539, 1991.

[27] Tsutomu Hasegawa, Takashi Suehiro, and Kunikatsu Takase. A model-based manipulation system with skill-based execution. *IEEE Transactions on Robotics and Automation*, 8(5):535–44, October 1992.

[28] Trevor Hastie and Werner Stuetzle. Principal curves. *Journal of the American Statistical Society*, 84(406):502–16, June 1989.

[29] Gerd Hirzinger, Bernhard Brunner, Johannes Dietrich, and Johan Heindl. Sensor-based space robotics–ROTEX and its telerobotic features. *IEEE Transactions on Robotics and Automation*, 9(5):649–63, October 1993.

[30] Geir Hovland, Pavan Sikka, and Brenan J. McCarragher. Skill aquisition from human demonstration using a hidden Markov model. In *Proceedings of the 1996 IEEE International Conference on Robotics and Automation*, volume 3, pages 2706–11, 1996.

[31] Michael F. Hutchinson and Frank R. de Hoog. Smoothing noisy data with spline functions. *Numerische Mathematik*, 47:99–106, 1985.

[32] Katsushi Ikeuchi and Takashi Suehiro. Toward an assembly plan from observation; part I: Task recognition with polyhedral objects. *IEEE Transactions on Robotics and Automation*, 10(3), June 1994.

[33] Ian T. Jolliffe. *Principal component analysis*. Springer-Verlag, New York, 1986.

[34] Sing Bing Kang and Katsushi Ikeuchi. Robot task programming by human demonstration. In *Proceedings of the Image Understanding Workshop*, 1994.

[35] Ralf Koeppe and Gerd Hirzinger. Learning compliant motions by task-demonstration in virtual environments. In *Fourth International Symposium on Experimental Robotics, ISER'95*, Lecture notes in control and information sciences, Stanford, CA, June-July 1995. Springer-Verlag.

[36] Mark A. Kramer. Nonlinear principal component analysis using autoassociative neural networks. *AIChe Journal*, 37(2):233–43, February 1991.

[37] Michael LeBlanc and Robert Tibshirani. Adaptive principal surfaces. *Journal of the American Statistical Society*, 89(425):53–64, March 1994.

[38] Christopher Lee and Yangsheng Xu. $(DM)^2$: A modular solution for robotic lunar missions. *International Journal of Space Technology*, 16(1):49–58, 1996.

[39] Christopher Lee and Yangsheng Xu. Online, interactive learning of gestures for human/robot interfaces. In *1996 IEEE International Conference on Robotics and Automation*, volume 4, pages 2982–7, Minneapolis, MN., 1996.

[40] Sheng Liu and Haruhiko Asada. Teaching and learning of deburring robots using neural networks. In *Proceedings of 1993 IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 3, pages 339–45. IEEE, July 1993.

[41] Tomás Lozano-Pérez. Task planning. In Michael Brady, editor, *Robot Motion: Planning and Control*. MIT Press, Cambridge, MA, 1982.

[42] Edward C. Malthouse. Limitations of nonlinear PCA as performed with generic neural neworks. *IEEE Transactions on Neural Networks*, 9(1):165–73, January 1998.

[43] Maja J. Matarić, Victor B. Zordan, and Matthew M. Williamson. Making complex articulated agents dance: An analysis of control methods drawn from robotics, animation and biology. *Autonomous Agents and Multi-Agent Systems*, 2(1):23–44, March 1999.

[44] Paul Michelman and Peter Allen. Forming complex dextrous manipulations from task primitives. In *Proceedings 1994 IEEE International Conference on Robotics and Automation*, volume 4, pages 3383–8, San Diego, CA, May 1994. IEEE Computer Society Press.

[45] Thomas P. Minka. Automatic choice of dimensionality for PCA. Technical Report 514, MIT Media Laboratory, Perceptual Computing Section, Cambridge, MA, December 1999.

[46] J. Daniel Morrow. *Sensorimotor primitives for programming robotic assembly skills*. PhD thesis, Robotics Institute, Carnegie Mellon University, April 1997.

[47] J. Daniel Morrow and Pradeep K. Khosla. Sensorimotor primitives for robotic assembly skills. In *Proceedings of the 1995 IEEE International Conference on Robotics and Automation*, volume 2, pages 1894–9. IEEE, May 1995.

[48] J. Daniel Morrow, Brad J. Nelson, and Pradeep K Khosla. Vision and force driven sensorimotor primitives for robotic assembly skills. In *Proceedings of the 1995 IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 3, pages 234–40. IEEE Computer Society Press, August 1995.

[49] Anne Murray. *Engineering Design and Psychophysical Evaluation of a Wearable Vibrotactile Display*. PhD thesis, Electrical and Computer Engineering Department, Carnegie Mellon University, Pittsburgh, PA, 1999.

[50] Anne Murray, Roberta Klatzky, and Pradeep Khosla. Enhancing subjective sensitivity to vibrotactile stimuli. In *ASME Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems*, Anaheim, CA, 1998.

[51] Anne Murray, Roberta Klatzky, and Pradeep Khosla. Summation of multi-finger vibrotactile stimuli. In *ASME Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems*, Nashville, TN, 1999.

[52] Keiji Nagatani and Shin'ichi Yuta. Designing strategy and implementation of mobile manipulator control system for opening door. In *Proceedings. 1996 IEEE International Conference on Robotics and Automation*, volume 3, pages 2828–34, Minneapolis, MN, 1996. IEEE.

[53] Michael C. Nechyba and Yangsheng Xu. Human skill transfer: neural networks as learners and teachers. In *Proceedings of the 1995 IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 3, pages 314–19. IEEE Computer Society Press, 1995.

[54] Michael C. Nechyba and Yangsheng Xu. Stochastic similarity for validating human control strategy models. In *Proceedings of the 1997 IEEE International Conference on Robotics and Automation*, volume 1, pages 278–83, Albuquerque, NM, April 1997. IEEE.

[55] Kevin Nilsen. Reliable real-time garbage collection in C++. *Computing Systems*, 7(4):467–504, 1994.

[56] Hiroyuki Ogata and Tomoichi Takahashi. Robotic assembly operation teaching in a virtual environment. *IEEE Transactions on Robotics and Automation*, 10(3):391–9, June 1994.

[57] Vladimir I. Pavlovic, Rajeev Sharma, and Thomas S. Huang. Visual interpretation of hand gestures for human-computer interaction: A review. *IEEE Transaction on Pattern Analysis and Machine Intelligence*, 19(7), 1997.

[58] Lawrence R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–285, February 1989.

[59] Lawrence R. Rabiner and B. H. Juang. An introduction to hidden Markov models. *IEEE ASSP Magazine*, pages 4–16, January 1996.

[60] Jonathon Rees and Bruce Donald. Program mobile robots in Scheme. In *Proceedings of 1992 IEEE International Conference on Robotics and Automation*, pages 2681–8, Nice, France, May 1992. IEEE.

[61] Christian H. Reinsch. Smoothing by spline functions. *Numerische Mathematik*, 10:177–83, 1967.

[62] Stefan Schaal. Is imitation learning the route to humanoid robots? *Trends in Cognitive Sciences*, 3(6):233–242, 1999.

[63] Stefan Schaal. Nonparametric regression for learning nonlinear transformations. In H. Ritter, O. Holland, and B. Möhl, editors, *Prerational Intelligence in Strategies, High-Level Processes and Collective Behavior*, volume 2, pages 595–621. Kluwer Academic Press, 1999.

[64] Stefan Schaal, Sethu Vijayakumar, and Christopher G. Atkeson. Local dimensionality reduction. In M. I. Jordan, M. J. Kearns, and S. A. Solla, editors, *Advances in Neural Information Processing Systems*, volume 10. MIT Press, Cambridge, MA, 1998.

[65] Stan Schneider, Vincent Chen, Jay Steele, and Gerardo Pardo-Castellote. The ControlShell component-based real-time programming system, and its application to the Marsokhod Martian rover. In *ACM SIGPLAN 1995 Workshop on Languages, Compilers, and Tools for Real-Time Systems*, volume 30 of *SIGPLAN Notices*, pages 146–55, June 1995.

[66] Isaac J. Schoenberg. Spline functions and the problem of graduation. *Proceedings of the National Academy of Science, U.S.A.*, 52:947–50, 1964.

[67] David W. Scott. *Multivariate Density Estimation : Theory, Practice, and Visualization.* John Wiley & Sons, 1992.

[68] Bernard W. Silverman. Spline smoothing: the equivalent variable kernel method. *The Annals of Statistics*, 12(3):896–916, 1984.

[69] Bernard W. Silverman. Some aspects of the spline smoothing approach to non-parametric regression curve fitting. *Journal of the Royal Statistical Society, Series B*, 47(1):1–52, 1985.

[70] Daniel Simon, Bernard Espiau, Konstantinos Kapellos, and Roger Pissard-Gibollet. ORCCAD: software engineering for real-time robotics; a technical insight. *Robotica*, 15(1):111–5, 1997.

[71] Thad Starner, Joshua Weaver, and Alex Pentland. Real-time American Sign Language recognition using desktop and wearable computer based video. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(12), December 1998. also appears as MIT Media Lab Perceptual Computing Section Technical Report No. 466.

[72] David Stewart, Richard Volpe, and Pradeep Khosla. Integration of real-time software modules for reconfigurable sensor-based control systems. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 325–332, 1992.

[73] M. Stone. Cross validatory choice and assesment of statistical preditions (with discusson). *Journal of the Royal Statistical Society, Series B*, 36:111–47, 1974.

[74] Gerald Sussman. *A Computer Model of Skill Aquisition*. Number 1 in Artificial intelligence series. American Elsevier Publishing Company, New York, 1975.

[75] Shufeng Tan and Michael L. Mavrovouniotis. Reducing data dimensionality through optimizing neural network inputs. *AIChe Journal*, 41(6):1471–1480, June 1995.

[76] Robert Tibshirani. Principal curves revisited. *Statistics and Computing*, 2(4):183–90, December 1992.

[77] Michael E. Tipping and Christopher M. Bishop. Mixtures of probabilistic principal component analysis. Technical Report NCRG/97/003, Neural Computing Research Group, Department of Computer Science and Applied Mathematics, Aston University, June 1997.

[78] Cheo-Ping Tung and Avi C. Kak. Automatic learning of assembly tasks using a dataglove system. In *Proceedings of the IEEE/RSJ Conference on Intelligent Robots and Systems*, pages 1–8, 1995.

[79] Sethu Vijayakumar and Stefan Schaal. Robust local learning in high dimensional spaces. *Neural Processing Letters*, 7:139–149, 1998.

[80] Richard Voyles. Tactile gestures for human/robot interaction. In *Proceedings of the IEEE/RSJ Conference on Intelligent Robots and Systems*, 1995.

[81] Grace Wahba. *Spline Models for Observational Data.* SIAM, Philadelphia, Pa, 1990.

[82] Andrew D. Wilson and Aaron F. Bobick. Realtime online adaptive gesture recognition. In *Proceedings of the International Workshop on Recognition, Analysis and Tracking of Faces and Gestures in Real-Time Systems*, Corfu, Greece, September 1999. also appears as MIT Media Lab Perceptual Computing Section Technical Report No. 505.

[83] Paul Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management*, number 637 in Springer-Verlag Lecture Notes in Computer Science, St. Malo, France, September 1992.

[84] Paul Wilson and Mark Johnstone. Real-time non-copying garbage collection. In *ACM OOPSLA Workshop on Memory Management and Garbage Collection*, Washington D.C., September 1993. ACM.

[85] Herman Wold. Soft modelling by latent variables: The Non-Linear Iterative Partial Least Squares (NIPALS) approach. In J. Gani, editor, *Perspectives in Probability and Statistics*, pages 117–142. Applied Probability Trust, 1975.

[86] Yangsheng Xu, Christopher Lee, and H. Benjamin Brown, Jr. A separable combination of wheeled rover and arm mechanism: $(DM)^2$. In *Proceedings of the 1996 IEEE International Conference on Robotics and Automation*, volume 3, pages 2383–8, 1996.

[87] Yangsheng Xu and Jie Yang. Towards human-robot coordination: skill modeling and transferring via hidden Markov model. In *Proceedings of*

the IEEE International Conference on Robotics and Automation, volume 2, pages 1906–1911, 1995.

[88] Jie Yang, Yangsheng Xu, and C.S. Chen. Hidden Markov model approach to skill learning and its application to telerobotics. *IEEE Transactions on Robotics and Automation*, 10(5):621–31, October 1994.

[89] Jianwei Zhang and Alois Knoll. Learning manipulation skills by combining PCA technique and adaptive interpolation. In *Sensor Fusion and Decentralized Control in Robotic Systems*, volume 3523 of *Proceedings of the SPIE – The International Society for Optical Engineering*, pages 211–22, Boston, MA, 1998.