Applications of Machine Learning in Sensorimotor Control

by

Susanne Bradley

B.Sc., Queen's University, 2013

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

August 2015

Abstract

There have been many recent advances in the simulation of biologically realistic systems, but controlling these systems remains a challenge. In this thesis, we focus on methods for learning to control these systems without prior knowledge of the dynamics of the system or its environment.

We present two algorithms. The first, designed for quasistatic systems, combines Gaussian process regression and stochastic gradient descent. By testing on a model of the human mid-face, we show that this combined method gives better control accuracy than either regression or gradient descent alone, and improves the efficiency of the optimization routine.

The second addresses the trajectory-tracking problem for dynamical systems. Our method automatically learns the relationship between muscle activations and resulting movements. We also incorporate passive dynamics compensation and propose a novel gain-scheduling algorithm. Experiments performed on a model of the human index finger demonstrate that each component we add to the control formulation improves performance of fingertip precision tasks.

Preface

The methods described in Chapter 5, with the exception of the gain-scheduling algorithm of Section 5.4, formed part of the following publication:

> Prashant Sachdeva, Shinjiro Sueda, Susanne Bradley, Mikhail Fain, and Dinesh K. Pai. Biomechanical simulation and control of hands and tendinous systems. *ACM Transactions on Graphics*, 34(4):42:1-42:10, July 2015.

For the control algorithm in this paper, we began with the methods described in Mikhail Fain's M.Sc. thesis, to which I added configuration-dependent AVMs (see Section 5.2), activation smoothing (Section 5.3), and passive dynamics compensation (Section 5.5). The index finger simulator described in the paper and used in this thesis for all control algorithms in Chapters 4 and 5 was developed by Prashant Sachdeva, Shinjiro Sueda and Duo Li, under the supervision of Dinesh Pai. The software interface between the controller and the simulator I developed jointly with Prashant Sachdeva.

The model of the skin of the human mid-face described in Chapter 3 was developed by Debanga Raj Neog under the supervision of Dinesh Pai. It is part of his upcoming doctoral thesis and is used here with his permission.

All algorithms, experiments and analysis in this thesis were developed by me, with guidance from my supervisor, Dinesh Pai.

Table of Contents

List of Tables

List of Figures

Glossary

ACT	Anatomically Correct Testbed
CE	Contractile Element
SE	Series Element
PE	Parallel Element
PID	Proportional-Integral-Derivative
PD	Proportional-Derivative
PI	Proportional-Integral
LPV	Linear Parameter-Varying
QP	Quadratic Program
MDP	Markov Decision Process
MSPE	Mean Squared Prediction Error
BLUP	Best Linear Unbiased Predictor
LSH	Locality-Sensitive Hashing
LHD	Latin Hypercube Design
IMSE	Integral Mean-Squared Error
MMSE	Maximum Mean-Squared Error

OOM Orbicularis Oculi Muscle

MLE Maximum Likelihood Estimate

AVM Activation-Velocity Matrix

LPS Levator Palpebrae Superioris

Acknowledgments

First and foremost, I would like to express my gratitude to my supervisor, Dr. Dinesh Pai, for guiding me through my initiation into the realm of academic research. Whenever I was wrestling with a particularly difficult problem, Dinesh proved to be an invaluable source of help. He would be able to identify – usually within about 30 seconds – the next approach to try, and it always turned out to be a great idea. I was deeply impressed by his knowledge and insight, and this thesis would not have been possible without his leadership. I want to thank Dr. Mark Schmidt for lending his expertise on different learning algorithms and serving as the second reader of this thesis. Many thanks also go to my co-author, Dr. Shinjiro Sueda, and my supervisor-to-be, Dr. Chen Greif, for all their help and guidance.

I'm very grateful to all my current and former labmates in the Sensorimotor Systems Lab. Special thanks goes to my SIGGRAPH buddy, Prashant Sachdeva, for all his help with the simulation and control of the index finger model. In particular, Prashant built the software platform I used for developing and testing control algorithms; without his skill and hard work our project would have been impossible. I also want to thank Debanga Raj Neog for developing the mid-face simulator I used for my work in Chapter 3 and allowing me to use it here. Lastly, huge thanks to Mikhail Fain, who gave me lots of great information about control algorithms and whose Master's thesis served as the inspiration for this one.

I want to thank my parents for all they have done to get me to where I am today. For all their love and support (both financial and otherwise) over the years, I am deeply grateful. Finally, very special thanks to my husband, Nick, for his unwavering love and encouragement, and for all the times he helped me with computer issues.

Chapter 1

Introduction

The human musculoskeletal system is a complex, intricate structure – a network of bones, muscles, tendons, and other connective tissues that supports the body and allows us to perform a huge variety of complex movements. Though it has been widely researched, how the brain learns to control these movements is not fully understood. The movement of each joint in our body is constrained by the muscles and tissues surrounding it: as a result, the relationship between muscle activation signals and resulting motor movements is complicated, non-convex and non-linear. From a computational perspective, even a minor task like holding your arm in a particular position is a challenging problem.

The goal of this thesis is to develop learning algorithms for motor control. We focus on simulated, biologically motivated systems, with particular emphasis on tendon-driven systems. In these systems, tendons are modelled as stiff elastic strands connecting bones and muscles. When the muscles are activated, they contract, which causes the tendon to pull on the bones and rotate the joints.

Most work in robotics focuses on torque-driven systems, in which motors manipulate each joint directly. Because their dynamics can be computed analytically, these systems are easier to control [80]. By comparison, several factors complicate the control of tendon-driven (and real biological) systems. Unlike the torque-driven approach, there is no one-to-one mapping between muscle activations and joint movements: a single muscle may span multiple joints and a single joint can be spanned by multiple muscles [22]. These mappings are complicated further by

non-linear interactions between muscles, tendons, bones and joints [17]. Tendon-driven systems also encounter issues with redundancy: there are multiple muscle activation patterns that result in the same movement [39].

These challenges beg the question: why not just work with torque-driven systems, thereby avoiding all the difficulties arising from the tendon-driven paradigm? One reason is that mimicking human biomechanics may enable us to develop robotic systems that approach human levels of dexterity. Another is that, in the process of developing control methods capable of handling the challenges inherent in biological systems, we may be able to gain insight into how the brain controls the body.

A key feature of human motor control is adaptability. The brain needs to be able to provide appropriate motor commands for different contexts we experience [84]. For example, if you're helping a friend move, you need to interpret contextual information and generate the correct motor signals to lift light boxes, lift heavy boxes, and not fling your arm behind your head and topple over backwards when a box you think will be heavy turns out to be light. These kinds of context changes can be brought about by external factors – as with the differently weighted boxes – or by internal ones, such as changes in muscle strength caused by aging or by muscle fatigue from repeatedly lifting heavy boxes.

If we want robots that are similarly adaptable, we need control algorithms that are:

1. Capable of learning complex, non-convex and non-linear mappings between muscle activations and resulting movements

2. Flexible enough to learn and perform motor skills in the presence of different external dynamics and internal constitutive models

To this end, our goal is to develop control algorithms that learn how to perform movements with no previous knowledge of the dynamics of the system or its environment. In the control of tendon-driven systems, much of what has been implemented relies on prior knowledge of internal dynamics (see [72] and the "white box" controller of Sachdeva et al. [60]) and manual tuning of controller parameters (see [18, 46, 88]).

The contributions of this thesis are:

- A novel framework combining supervised learning and local search for the control of an anatomy-based quasistatic system (Chapter 3)
- A data-driven method to learn control of tendon-driven dynamical systems, including passive dynamics compensation (Chapters 4 and 5.5), configuration-dependent mappings between muscle activations and motor movements (Section 5.2), and automatic selection of feedback parameters (Section 5.4)

Chapter 2

Background and Related Work

The goal of this thesis is to use machine learning techniques to improve existing methods for the control of biomechanical systems. We begin by considering the control of a dynamical system (also referred to as a *plant*), which is defined by:

$$x_{t+1} = f(x_t, u_t) \tag{2.1}$$

where f is the function defining the forward dynamics of the system, x_t is the state of the system at time t, and u_t is the control signal. In practice, f is often a complicated non-linear function that is not known in advance.

The problem in control is to select the activations u_t to achieve some desired behaviour of the plant. This thesis focuses mainly on *trajectory-tracking* problems: at each timestep t we have a specified target state x_r, and the goal is to make the plant follow the target points as closely possible. An important special case that we explore in detail is reaching tasks, which can be formulated as a trajectory-tracking problem by keeping x_r constant for all timesteps.

Section 2.1 deals with some properties of biological motor systems, including how motor learning occurs in the brain and how these systems are represented in computer science and robotics. Section 2.2 overviews existing methods for control of dynamical systems, and Section 2.3 covers some concepts in machine learning that can be used for this task.

2.1 Properties of Biological Motor Systems

The control literature is replete with work on motor tasks: examples include locomotion [22, 38, 83, 87], swimming [70], and movements of the head [39], eyes [36] and hands [18, 20, 43, 46, 50]. This section examines some properties of biological learning and discusses how biomechanical systems are represented in the literature.

2.1.1 Components of Biological Learning

Current approaches in biological motor learning focus mainly on information extraction and decision-making [85]. Learning to perform a task requires effective gathering and processing of sensory information relevant to action. Humans are highly skilled in this regard: Najemnik and Geisler [51] showed that human saccades (rapid movements of the eye between fixation points) are near-optimal in the Bayesian sense at extracting task-relevant information. Meanwhile, performing a task consists of decisions and strategies: we decide when to take what action as information is obtained. Research suggests the existence of simultaneous processes in the brain that specify potential motor actions and select between them [11].

Dimensionality reduction The human body has approximately 650 skeletal muscles [55]. One can imagine the difficulty of learning to control a system with 650 degrees of freedom. However, human movement consists of many muscles moving together in coordinated actions, rather than each muscle working individually. This means that the dimensionality of the control problem is reduced.

This concept is referred to as *muscle synergies*: neural control modules that can be flexibly combined to generate a large repertoire of behaviours [84]. The central nervous system controls large numbers of degrees of freedom by joint-level and muscle-level synergies [78]. Support for the synergy hypothesis is given by Berger et al. [7], who investigated reaching tasks in a virtual environment where the normal muscle-to-force mappings could be changed. After identifying muscle synergies, they found that the subjects' adaptation was faster when a full range of movements could be achieved by recombining synergies than when new synergies were required.

From a computational perspective, this means that we can simplify control tasks using standard dimensionality reduction techniques [26]. For example, Ingram et al. [35] found that 80% of the variance of hand movements was accounted for by two principal components.

Challenges of biological control Motor control suffers several inherent challenges. One is the need for adaptability: our brain/controller must be able to provide appropriate motor commands for different contexts likely to be experienced [84]. Error correction is difficult as sensory streams are temporally delayed and corrupted by noise [36, 85]. In addition, the mappings between muscle activations and resulting movements are difficult to learn. Muscles, tendons, bones and joints interact non-linearly [17], and there is an issue of redundancy: for example, one musculoskeletal configuration can be achieved in many ways by co-activating agonist and antagonist muscles [39].

2.1.2 Representations of Biomechanical Systems

Biomechanical systems are represented in control literature in one of two ways: *torque-* or *tendon-driven*. Torque-driven systems are used by Liu [43], Yin et al. [87] and Wang et al. [83], among others.

Tendon-driven systems are simulated using thin strands for tendons and rigid bodies for bones. A simulated tendon-driven system is used by Sueda et al. [72], and a robotic example is the Anatomically Correct Testbed (ACT) hand [17]. Tendon-driven systems have the advantage of matching real anatomy: the fibers of the strand contract, which transmits contractile force directly along the fiber. They also allow for accurate imitation of physiological phenomena – such as joint coupling [60] – which are difficult to model with a torque-driven approach. They do, however, pose difficulties for control: tendon-driven muscles do not provide direct control over degrees of freedom (as is typically assumed with computed torque methods) because a single muscle may span multiple joints, and a single joint may be spanned by multiple muscles [22].

Muscles are commonly simulated using Hill-type muscle models (for example, in [22, 39, 70]). The model consists of a Contractile Element (CE) and two non-

linear spring elements: the Series Element (SE) and Parallel Element (PE). The PE represents the passive force of the connective tissues surrounding the muscle, while the SE represents the tendons and provides an energy-storing mechanism [47].

The force produced by the CE, F_{CE}, is given by:

$$F_{CE} = u\mathrm{F}_{\max} f_L(L_{CE}) f_V(V_{CE}) \tag{2.2}$$

where u is the muscle activation level, $\mathrm{F}_{\max}$ is the muscle's maximum isometric force, f_L describes the relationship between force and muscle length (L_{CE}), and f_V gives the relationship between force and current contraction velocity (V_{CE}).

The passive forces F_{PE} and F_{SE} are modelled as non-linear springs based on their length:

$$F_{SE} = f_{SE}(L - L_{CE}) \tag{2.3}$$

$$F_{PE} = f_{PE}(L_{CE}) \tag{2.4}$$

where f_{SE}, f_{PE} are non-linear force-length relations and L is the total muscle length.

The total muscle force F and the forces in the CE, PE and SE satisfy

$$F = F_{PE} + F_{SE} \tag{2.5}$$

$$F_{CE} = F_{SE} \tag{2.6}$$

For an illustration of the model, see Figure 2.1.

2.2 Control of Dynamical Systems

Now that we have covered some dynamical system representations of biomechanical systems, we discuss methods for controlling dynamical systems. A key issue in control is the choice of parametrization of the desired task. This is given in terms of a *configuration variable*, denoted by q_t. The subtle but important distinction between the configuration variable q_t and the state x_t is that q_t describes the position

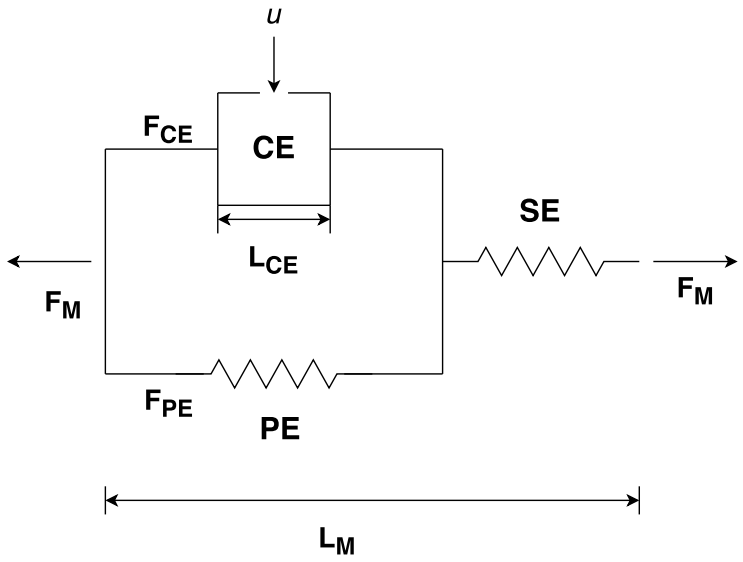


Figure 2.1: Illustration of a Hill-type muscle model.

of the system, while x_t describes both the position and the velocity.

Section 2.2.1 deals with asymptotic reaching control, which is the problem of guiding a plant to a desired configuration. Section 2.2.2 introduces feedback control, and Section 2.2.3 discusses hierarchical control methods. Section 2.2.4 then overviews the control of stochastic systems and Section 2.2.5 explores issues relating specifically to the control of sensorimotor tasks.

2.2.1 Reaching Control

Consider a dynamical system governed by Equation 2.1. The reaching control problem consists of computing the activation u_t at each timestep so that the system state x_t converges to some target x_r.

Equilibrium control A conceptually straightforward approach to this problem is equilibrium control, reviewed in [66]. It relies on the concept of *equilibrium points*: these are state-activation pairs $\{x_{eq}, u_{eq}\}$ such that $x_{eq} = f(x_{eq}, u_{eq})$. That is, the system at state x_{eq} does not move when given the activation u_{eq}. The main idea

behind equilibrium control is that if we wish to drive the plant to the state x_{eq}, it may suffice to apply the controls u_{eq} that keep the system in equilibrium at that point.

The advantage of equilibrium control is its simplicity. The existence of a mapping between activations and states means that the problem can be handled with standard supervised learning methods. However, this simplicity comes at a price. The activation-to-state mapping means the effects of starting configuration and intermediate dynamics are ignored. There is no guarantee that the mappings are bijective: multiple activations could stabilize at a point, or an activation could stabilize the system at multiple points. Convergence tends to be slow, though this can be overcome with the *pulse-step* paradigm, in which a stronger "pulse" activation precedes the equilibrium activation. The pulse-step pattern is seen in the control of saccadic eye movements (see [36, 71]).

Because of its drawbacks, equilibrium control is rarely used by itself in practice, but it is often used as part of a more sophisticated controller scheme. Deshpande et al. [18] and others have used equilibrium control to account for passive muscle forces at a given plant configuration. Liu [43] used it as a preprocessing step for object manipulation from a grasping pose: at an input pose, equilibrium controls were used to maintain the pose against gravity.

Task parametrization Reaching tasks can be parametrized either by spline-based trajectories or with dynamics-based approaches. The spline-based approach is easier to understand, but does not easily generalize to new behavioural situations without complete recomputation of the spline. It also can't easily be co-ordinated with other events in the environment (for example, catching a ball) [52]. The alternative is to use a dynamical system to specify an attractor landscape. This allows for an extension of equilibrium control-type methods beyond reaching for a single point: with this parametrization, the target state can also be a periodic orbit or manifold [33, 70]. Peters and Schaal [52] used parametrized non-linear dynamical systems as motor primitives, where the attractor properties of the systems defined the desired behaviour.

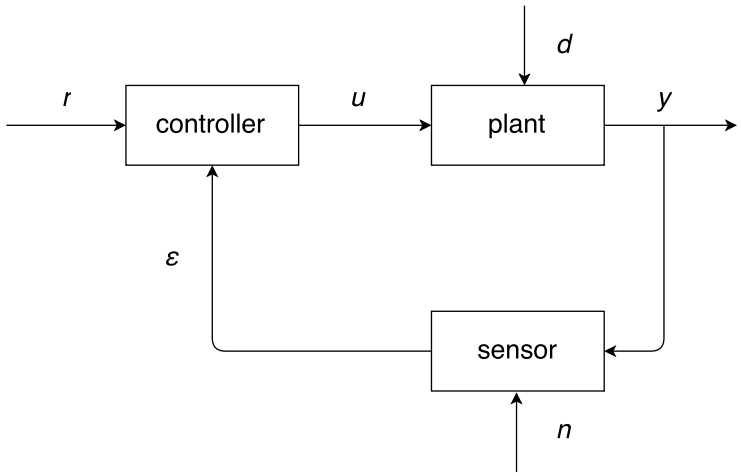


Figure 2.2: A basic feedback system, where: r is the reference command; ε is the sensor output, which gives an estimate of the control error; u is the actuating signal; d is the external disturbance acting on the plant; y is the plant output and measured signal; and n is sensor noise.

2.2.2 Feedback Control

Equilibrium control is an example of *feedforward* control: it is based purely on sensory signals that are selected prior to movement [84]. Purely feedforward methods are not the best approach in practice: errors inevitably occur during movement, so we need a way to correct errors as we perceive them.

Feedback control basics This brings us to feedback control, which is based on the outcome of a movement. A block diagram of a basic feedback system [19] is given in Figure 2.2. It consists of three components – the controller, the plant, and the sensor – each having two inputs and one output.

PID control A common method for control of feedback systems is the Proportional-Integral-Derivative (PID) controller. Let ε_t denote the sensor's error signal at time t. The PID controller gives the control signal:

$$u(t) = K_P\varepsilon_t + K_I\int_0^t \varepsilon_\tau \mathrm{d}\tau + K_D\frac{\mathrm{d}}{\mathrm{d}t}\varepsilon_t \tag{2.7}$$

	Ziegler-Nichols parameters		
Control type	K_P	K_I	K_D
PI	$0.45K_U$	$1.2K_P/T_U$	-
PD	$0.8K_U$	-	$K_PT_U/8$
PID	$0.6K_U$	$2K_P/T_U$	$K_PT_U/8$

Table 2.1: PID tuning with the Ziegler-Nichols method.

where K_P, K_I and K_D are scalar tuning parameters referred to as *gains*. The first term of this equation is the proportional term, which produces an output proportional to the current error. The integral term (second term) contributes an output proportional to the accumulated error over time; this helps eliminate steady-state error. Lastly, the derivative term (third term) produces an output proportional to the slope of the error; this helps improve control by predicting the behaviour of the system [19].

Special cases of PID control include Proportional-Integral (PI) control (same as Equation 2.7 but with the derivative term omitted) and Proportional-Derivative (PD) control (integral term omitted). PID control is used by [18, 46, 88] and PD control by [2, 14, 22, 39, 70, 83, 87]. In practice, the gains parameters are mostly tuned by hand; this is the case in all the papers just mentioned. This tuning can be complicated, as gains may need to be chosen per controller state [88], per joint [18], or both [46].

Loop tuning One method for tuning a PID controller is the *Ziegler-Nichols method* [90]. In this method, the integral and derivative gains K_I and K_D are set to 0. The proportional gain K_P is set to 0 and gradually increased until it reaches the *ultimate gain*, K_U, defined as the point at which the output of the control loop oscillates with a constant oscillation period T_U. The gains values are then set according to the type of controller used – see Table 2.1.

There are other, similar tuning methods – for example, Haugen's [30] – that do not require the loop to get into oscillations during tuning.

Gain scheduling In gain scheduling, the gains parameters K_P, K_I and K_D depend on the current system configuration. The general idea in classical gain scheduling is

to divide the configuration space into small areas that are assumed to be linear, create a PID controller for each section of the space, and use blending or interpolation at other points to ensure that changes in controller parameters are smooth [40]. In other words, we model the dynamical system as a Linear Parameter-Varying (LPV) system (mentioned explicitly in [64, 69] and used implicitly in [18, 88]). This is a class of non-linear systems that can be modelled as parametrized linear systems whose parameters change with their state.

The drawback of classical gain scheduling is that adequate performance and stability are not guaranteed at points other than the design points [68]. In particular, we are restricted to slow variations in the scheduling variable (some functional of which determines the gains at a given configuration), so we may encounter problems if there are fast parameter variations.

For detailed descriptions of specific gain scheduling algorithms, we refer the reader to [28, 69, 77].

2.2.3 Hierarchical Control Methods

Hierarchical control is a divide-and-conquer approach that separates a control task into multiple (usually two) simpler tasks. Typically, one level controls high-level dynamics and a second level computes muscle activations to satisfy the high-level commands.

These methods are biologically plausible – evidence suggests that the central nervous system works hierarchically [78] – and confer several computational advantages. Hou et al. [32] found that a hierarchical control framework greatly improves the efficiency of neural network calculations. By including a formulation for converting high-level commands into something that is solvable by a low-level method, this control framework allows the user to provide commands parametrized at the high level: hence, it becomes easier and more intuitive to specify control goals. For example, the method of Mordatch et al. [50] for object manipulation employs high-level trajectory optimization to synthesize motion and contact events from only a few high-level goals. This allows the user to specify a goal in terms of movement of the object, while the hierarchical control algorithm uses an inverse dynamics Quadratic Program (QP) to ensure that forces, controls and contact

variables are consistent with each other.

Hierarchical methods enjoy great popularity in the control literature. Si et al. [70] developed a swimming controller in which neural networks produced rhythmic motor patterns based on high-level commands (such as amplitude, frequency and phase) to give desired muscle lengths, while a low-level PD controller produced activations to give desired muscle lengths. In the work of Deshpande et al. [18], a high-level controller determined the desired muscle length for a target joint angle in the index finger, and a low-level PID controller computed the corresponding muscle forces. Deisenroth and Rasmussen [16] employed a hierarchical formulation to decompose the motor learning problem into a hierarchy of three sub-problems.

2.2.4 Control of Stochastic Systems

Control tasks are often subject to some degree of stochasticity: noise corrupts observations of the environment, the dynamics of the system's movement, or the activation signals computed by the controller. In biological systems, this is always the case [85].

Several control algorithms have been designed for stochastic systems. Jones [36] gave an optimal control law for performing saccades where the cost function depended on the statistical distribution of the eye's position, obtained with Monte Carlo sampling. Geijtenbeek et al. [22] incorporated noise in the form of temporal delay into their feedback control formulation. In Huh and Todorov's paper [33], both controller and body were assumed to be stochastic dynamical systems, and activations were computed using a risk-sensitive objective function. A reinforcement learning framework (see Section 2.3.3) can handle Markov Decision Processes (MDPs) with stochastic rewards, observations, and/or state transitions [74].

2.2.5 Control Methods for Sensorimotor Tasks

Subsections 2.2.1 - 2.2.4 deal with general control principles. In this section, we discuss some considerations that arise specifically in the control of motor tasks.

Managing multiple phases of movement Practically speaking, many motor tasks – such as locomotion and object manipulation – involve several stages of movement. For complicated, multi-stage tasks, researchers commonly use finite state machine models. This involves building different controllers at each stage of movement. For example, locomotion controllers are often divided into four states, corresponding to the ground contact and swing phases for each foot [83, 87]. Generating the desired behaviour in different states is accomplished either by building conceptually different controllers for each, as in de Lasa et al. [14], or by having one controller framework and changing the parameters for each state. Wang et al. [83], Yin et al. [87] and Geijtenbeek et al. [22] accomplished the latter by changing the weights of an objective function, which was optimized to yield the control signal. Zhang et al. [88] and Malhotra et al. [46] changed the per-joint controller gains at each state, while Andrews and Kry [2] changed the controller's joint stiffness and damping matrices.

Enforcing realism of movements A common concern in control problems – particularly in computer graphics – is ensuring that movements look natural. Animation techniques generally have a trade-off between motion naturalness and the control that can be exerted over the motion [79].

Several techniques can be used to encourage natural motions. For methods that compute activation signals by optimizing an objective function, one can add to this function terms that penalize unnatural-looking behaviour – or, more generally, anything in the movement (besides control error) that is undesirable. For example, the locomotion controller of Wang et al. [83] included penalty terms to reduce head movement and penalize unnatural – i.e. significantly different from human – joint torque ratios between the hips, knees and ankles.

Another approach, which has tie-ins to biological learning, is to parametrize the control task in terms of muscle synergies or, more broadly, high-level features that result in coordinated muscle movements. Compared to activating each muscle individually, this results in more natural, controlled movements. For examples, see [14] for walking, [70] for swimming, and [88] for playing the piano.

When the motion's appearance is the primary concern, it is possible to forego physical fidelity to achieve the desired plausible motions – for example, by blend-

ing kinematic and dynamic motions, as in [91]. Another approach is the direct imitation of motion capture data. Grochow et al. [24] and Wang et al. [82] both fit dynamical models to motion capture frames to learn the most likely poses and motions satisfying a set of constraints.

2.3 Machine Learning Approaches in Motor Control

Learning is a crucial component in the performance of motor tasks. Because of the high dimensions of the actuation and movement spaces and the complexity of the mappings between them, there is great potential for techniques from statistics and machine learning. In this section, we focus on the control of simulated – rather than robotic – systems. We also focus more on problems where the output is deterministic given the input, rather than on stochastic problems. Hence, our learning process can be categorized as a *computer experiment*: as the name suggests, these are experiments involving simulated processes. These differ from physical experiments in that they are assumed to have no random error: the system's output is a deterministic function of its input. According to Sacks et al. [61], the role of statistics in computer experiments is:

- Data analysis
- Quantification of uncertainty associated with prediction of fitted models
- Experimental design

Section 2.3.1 describes supervised learning methods that can be used in control, while Section 2.3.2 discusses design principles for computer experiments. Section 2.3.3 concludes with an overview of reinforcement learning.

2.3.1 Supervised Learning

This section deals with supervised learning, specifically regression – namely, learning a mapping from inputs to continuous outputs. Regression for computer experiments involves modelling a complicated deterministic function (such as the simulation of the dynamical system in Equation 2.1) as the realization of a stochastic process. This allows the use of statistical techniques [61].

Gaussian process regression A Gaussian process is a family of statistical distributions where every finite collection of random variables within an input space follows a multivariate normal distribution. Because of properties inherited from the normal distribution, the distributions of various derived quantities can be obtained analytically [56]. The model is defined by its mean and covariance functions.

Suppose we have inputs $\{x^{(1)},...,x^{(n)}\}$ and corresponding responses $\{Y^{(1)},...,Y^{(n)}\}$. We focus here on the case in which the response is modelled as a linear function of the inputs:

$$Y(x) = \sum_{j=1}^{k} \beta_j \phi_j(x) + Z(x) \tag{2.8}$$

where ϕ are previously specified regression functions, k is the number of regression functions ϕ_j, β is a vector of regression coefficients, and $Z(\cdot)$ is a random process assumed to have mean 0 and covariance $V(w,x) = \sigma^2 R(w,x)$ for some correlation function R and variance σ^2. There are two ways to interpret Equation 2.8. One is to view the responses as departures from a simple regression model [61]. The second is that the $Y(\cdot)$ are a Bayesian prior on the true response functions, with the β's either specified a priori or given a prior distribution [12].

Suppose now that, given a design $\mathcal{S} = \{s^{(1)},...,s^{(n)}\}$ and data $Y_s = \{Y(s^{(1)}),...,Y(s^{(n)})\}'$, we wish to construct a linear predictor $\hat{Y}(x) = c'(x)Y_s$ of $Y(x)$ at untried x. In *kriging*, we select the Best Linear Unbiased Predictor (BLUP): in other words, we want to choose a vector $c(x)$ minimizing the Mean Squared Prediction Error (MSPE):

$$\text{MSPE}(\hat{Y}) \equiv \text{E}\{(\hat{Y} - Y)^2\} \tag{2.9}$$

subject to the unbiasedness constraint

$$\text{E}\{c'(x)Y_s\} = \text{E}\{Y(x)\} \tag{2.10}$$

We now introduce some notation. Let $\phi(x) = [\phi_1(x),...,\phi_k(x)]'$ denote the vector of the k functions in the regression; $\Phi = \begin{pmatrix} \phi'(s^{(1)}) \\ \vdots \\ \phi'(s^{(n)}) \end{pmatrix}$ the $n \times k$ expanded de-

sign matrix; $R = \{R(s^{(i)}, s^{(j)})\}, 1 \leq i \leq n, 1 \leq j \leq n$ the $n \times n$ matrix of stochastic-process correlations between Z values at the design sites; and $r(x) = [R(s^{(1)}, x), ..., R(s^{(n)}, x)]'$ the vector of correlations between Z values at the design sites and at an untried input x.

The BLUP is given by:

$$\hat{Y}(x) = \phi'(x)\hat{\beta} + r'(x)R^{-1}(Y_s - \Phi\hat{\beta}) \tag{2.11}$$

where

$$\hat{\beta} = (\Phi' R^{-1}\Phi)^{-1}\Phi' R^{-1} Y_s \tag{2.12}$$

is the least-squares estimate of β. For proof that Equation 2.11 is in fact the BLUP, we refer the reader to [63]. The fit occurs in two stages: we first obtain the generalized least-squares predictor (the first term on the right-hand side of Equation 2.11) and then interpolate the residuals as if there were no regression model (the second term). Hence, the kriging regression model differs from the standard regression model in that the model interpolates the values at the design points. This is obviously desirable for computer experiments, where the output is assumed to be noise-free. For a visual depiction of kriging, see Figure 2.3.

Gaussian process regression is rather time-consuming ($O(n^3)$), but can be sped up with matrix inverse CUDA parallelization [15] or sparsification methods [9]. Because of the need to calculate R^{-1}, the method may become unstable when R is poorly conditioned.

Gaussian process methods are common in the control literature [44, 86]. It was used in Wang et al. [82] to fit dynamical models to human motion, and in Grochow et al. [24] for inverse kinematics. Deshpande et al. [18] used Gaussian processes for learning the moment-arm matrix (which gives the relationship between muscles and joints) of the ACT hand based on motion capture data.

Nearest-neighbours The idea behind nearest-neighbours prediction is to predict the value at a new input site x based on some function of the values of the k nearest neighbours. Some choices for this function are weighted averages and local regression models. k-nearest-neighbours with an inverse distance-weighted average

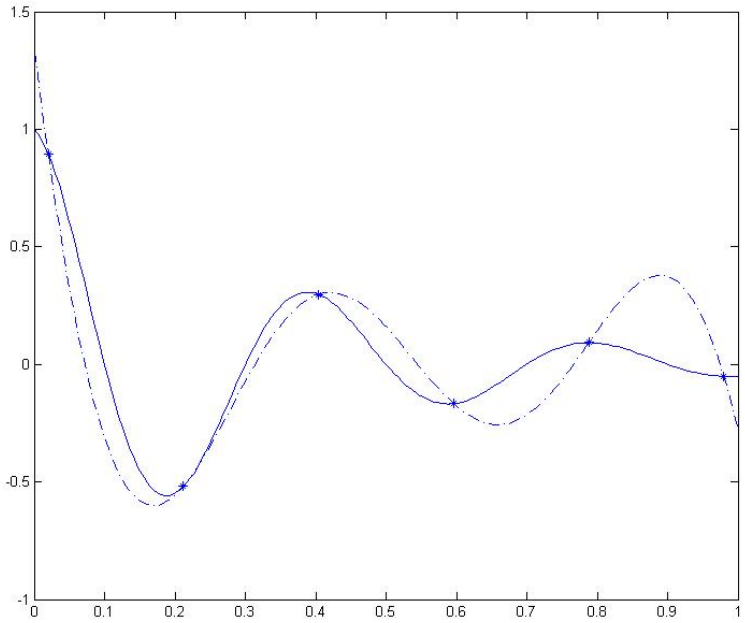


Figure 2.3: Kriging in 1D for $f(x) = \exp(-3x) \cdot \cos(5\pi x)$, given by the solid line. Design points are shown by asterisks and the dashed line gives the predicted values obtained with a kriging estimator using an exponential correlation function [61] with parameters $\theta = 4, \rho = 2$

.

is used for policy value estimation in Andrews and Kry [2].

Most standard nearest-neighbours methods, for example *kd*-trees [89], have either space or query time that is exponential in the number of points and/or the dimension of the data. However, Locality-Sensitive Hashing (LSH) algorithms can perform approximate *k*-nearest-neighbours computations with both space and query time that is linear in the number of points to search [1].

Neural networks A neural network is represented by a series of interconnected units whose behaviour is inspired by that of neurons. Learning occurs by adjusting the weightings between these neurons. This is analogous to learning in the brain,

which is effected by creating new synapses and strengthening existing ones [59].

Neural networks are ubiquitous in the control literature. For example, Grzeszczuk et al. [25] performed approximations of physics-based animations by replacing the entire dynamical system with a neural network, which could be easily controlled by gradient-based methods (as the gradient of a neural network can be computed analytically). Recurrent neural networks were used for equilibrium control in Huh and Todorov [33], and the swimming controller of Si et al. [70] produced rhythmic outputs using a special class of neural networks called Central Pattern Generators. Deep learning methods are often used (e.g. in Mnih et al. [49]) because some functions cannot be efficiently represented by architectures that are too shallow: they may need exponentially many more units if the network depth is even 1 too low [6].

2.3.2 Experimental Design

The experimental design problem addresses how we should sample from our input space to obtain training data for our learning algorithms. In computer experiments, uncertainty arises only from not knowing the exact function between inputs and response variables. Therefore, designs should take no more than one observation at a set of inputs, allow for a variety of models, and provide information about all portions of the output space [61].

Space-filling designs Methods like kriging are interpolators; so intuitively, we want space-filling designs. A simple strategy is to select points according to a regular grid pattern superimposed on the experimental region. However, in this case, the number of points required grows exponentially with the dimension of the experimental region, making it infeasible for high-dimensional problems.

Another straightforward option is a simple random sample (fig. 2.4a). These are eventually space-filling, but don't work well for small sample sizes. A more sophisticated approach is a stratified random sample [63], in which we obtain n points by dividing the region into n evenly spread strata and randomly select a single point from each (fig. 2.4b).

A popular approach is the Latin Hypercube Design (LHD) [48], in which we

spread observations evenly over the range of each input separately (fig. 2.4c). These designs possess desirable marginal properties, but only a subset of them are truly "space-filling" [63]. A variation of LHD is the cascading LHD: essentially, we generate an LHD, consider a small region around each point of the design, and generate a second LHD in that region. This allows the exploration of both the local and global correlation structure of the model for the response. Other variations of LHDs are based on orthogonal arrays [31].

Criterion-based designs In space-filling designs, we use geometric criteria measuring the amount of "spread" of the data points. More classical methods of choosing an experimental design are based on some statistical criterion [63]. One such approach is called *D-optimality*, in which we wish to minimize the determinant $D = |X^T X|$, where X is the design matrix [13]. This is equivalent to minimizing the variance of the least-squares estimates of the regression parameters.

Another strategy is *maximum entropy design*, which seeks to maximize the expected Shannon entropy of observed responses at the design points [65]. Assume that our model for the response function is parametrized by θ, and let $[\theta|\mathcal{D}]$ denote the posterior distribution of θ given the data obtained with the design $\mathcal{D}$. Before the experiment, the amount of information about θ (given by its prior distribution, $[\theta]$) is:

$$I = \int [\theta] log[\theta] \mathrm{d}\theta \tag{2.13}$$

The amount of information about θ after the experiment using the design $\mathcal{D}$ is:

$$I_{\mathcal{D}} = \int [\theta|\mathcal{D}] \log([\theta|\mathcal{D}]) \mathrm{d}\theta \tag{2.14}$$

The change in information after using $\mathcal{D}$ is given by $I - I_{\mathcal{D}}$. The expected change in information is maximized by the design that maximizes the entropy of the observed responses at points in the design: this is what is meant by maximum entropy design. An algorithm for finding such designs is described by Malakar and Knuth [45].

Lastly, *mean-squared prediction error designs* are based on minimizing some functional of the MSPE [63]. The Integral Mean-Squared Error (IMSE) criterion

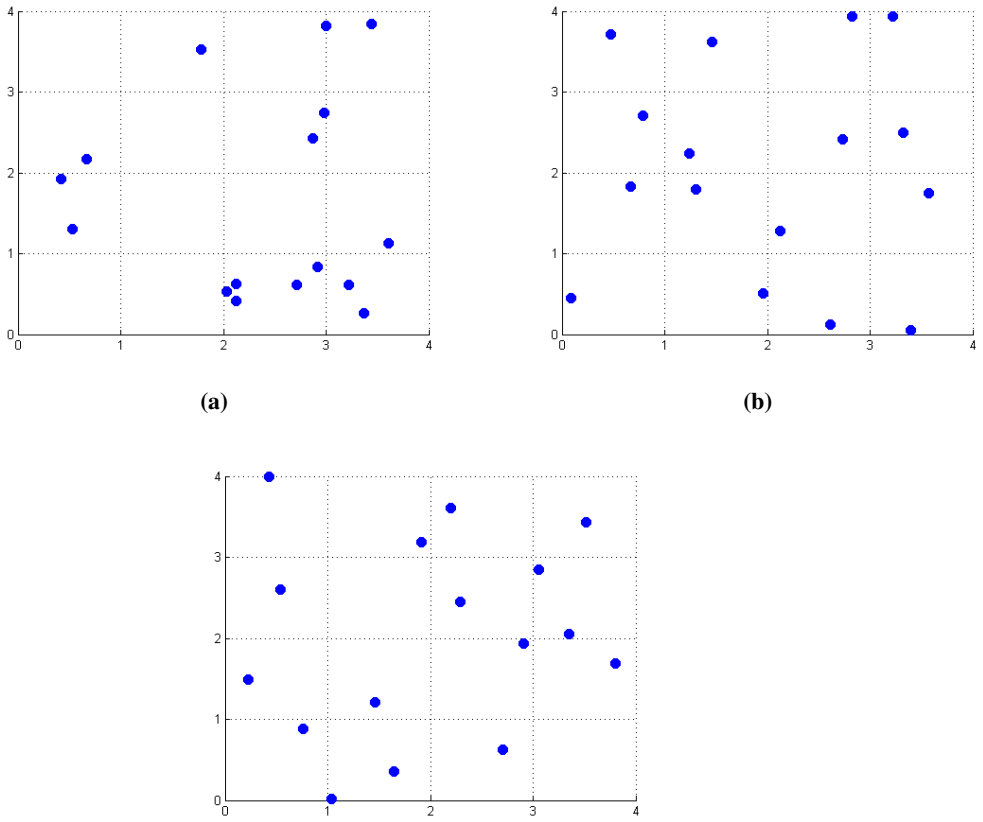


Figure 2.4: Illustration of different space-filling design techniques for 16 sample points in 2 dimensions. (a) Simple random sample. (b) Stratified random sample, where each stratum is a square comprising one-sixteenth of the design space. (c) Latin hypercube sampling – the projection of the points onto either the x or y axes is spaced uniformly.

minimizes the average MSPE over the experimental domain $\mathcal{X}$, and the Maximum Mean-Squared Error (MMSE) minimizes the worst-case prediction error over $\mathcal{X}$. Both the IMSE and MMSE tend to locate points away from the boundaries of the design space.

Motor babbling The motivation of experimental design methods in the statistical literature is similar to that of *goal babbling* in control theory. This approach differs from random motor babbling – in which we gather data by supplying random actuation signals – in that we sample points using goal-directed exploration. This is similar to how infants learn motor skills [57]. Rolf [58] presented a goal babbling method for exploring an unknown space. It is based on successive attempts to move as far as possible in a particular target direction until there are large deviations between the current state and the target – a hint that the target is outside range and we have gone as far as possible in that direction.

2.3.3 Reinforcement Learning

Sections 2.3.1 and 2.3.2 have dealt with supervised learning. However, because the solution space in control problems is frequently non-convex, we may encounter problems if we employ supervised learning methods without further consideration [54].

We now introduce the *reinforcement learning* framework to address some of these problems. Reinforcement learning involves mapping situations to actions so as to maximize reward. The most important features of reinforcement learning are trial-and-error search and delayed reward [74]. It differs from supervised learning in that correct input/output pairs are never presented, and "incorrect" actions are never explicitly corrected. Reinforcement learning is mainly used to compute optimal controls for problems that are framed as a Markov Decision Process (MDP), as in Wampler et al. [81].

Value functions The key computational problem for most reinforcement learning algorithms is estimating *value functions*: these are functions of states (or state-action pairs) that estimate how good it is to be in a particular state (or how good it is to perform a given action in a particular state) [74]. To quantify this, we define the notion of a *return*: this is some function of the sequence of rewards experienced at each timestep, which we wish to maximize. The return R_T is generally a sum $\left(R_T = \sum_{k=0}^{T} r_k\right)$ or a discounted sum $\left(R_T = \sum_{k=0}^{T} \gamma^k r_k, 0 < \gamma < 1\right)$ of the instantaneous reward r_k at each timestep k. We also define a *policy* π as a function mapping states

to actions.

The value of a state s under policy π, denoted $V^\pi(s)$, is the expected return when starting in s and following π thereafter. For MDPs:

$$V^\pi(s) = \mathrm{E}_\pi\{R_t|s_t = s\} = \mathrm{E}_\pi\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s\} \tag{2.15}$$

We call V^π the *value function for the policy* π. The value of taking an action a in state s under policy π is defined by the action-value policy $Q^\pi(s,a)$:

$$Q^\pi(s,a) = \mathrm{E}_\pi\{R_t|s_t = s, a_t = a\} = \mathrm{E}_\pi\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s, a_t = a\} \tag{2.16}$$

For any policy π and state s, the following consistency condition holds between the value of s and the value of its successor states:

$$V^\pi(s) = \mathrm{E}_\pi\{R_t|s_t = s\} = \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V^\pi(s')] \tag{2.17}$$

where $\mathcal{P}^a_{ss'}$ is the probability of transition to state s' given that action a was taken in state s and $\mathcal{R}^a_{ss'}$ is the expected value of the next reward r given a current state s, action a and next state s'. Equation 2.17 is called the *Bellman equation* for V^π. V^π is the unique solution to its Bellman equation [5].

The value function provides a method for comparing different policies. We say that a policy π is optimal if its expected return $V^\pi(s)$ is greater than or equal to that of any other policy π' for all states. All optimal policies π^* share an optimal state-value function $V^*(s) \equiv \max_\pi V^\pi(s)$ for all $s \in \mathcal{S}$. They also share an optimal action-value function, denoted Q^* and defined by $Q^*(s,a) \equiv \max_\pi Q^\pi(s,a)$. We can write Q^* in terms of V^* as:

$$Q^*(s,a) = \mathrm{E}\{r_{t+1} + \gamma V^*(s_{t+1})|s_t = s, a_t = a\} \tag{2.18}$$

Policy iteration Assume we have a given policy π with a corresponding (known) value function V^π. Suppose that in state s, we want to know whether we should change the policy to choose an action $a \neq \pi(s)$. We can determine how good it

would be to follow a and then return to the policy π with the value

$$Q^{\pi}(s,a) = \mathrm{E}_{\pi}\{r_{t+1} + \gamma V^{\pi}(s_{t+1}) | s_t = s, a_t = a\} = \sum_{s'} \mathcal{P}^{a}_{ss'}[\mathcal{R}^{a}_{ss'} + \gamma V^{\pi}(s')] \quad (2.19)$$

If this value is greater than $V^{\pi}(s)$, this means it's better to select a once in s than to follow π all the time. As such, we would expect that if we select a every time we encounter s, the new policy would be better overall. The process of making a new policy that improves on the original policy by making it greedy with respect to the original policy's value function is called *policy improvement* [74].

Repeating this process leads to *policy iteration* [75] – once π has been improved using V^{π} to yield a better policy π', we can the compute $V^{\pi'}$ and use it to improve π', and so on. By alternating policy evaluation and policy improvement steps, we obtain a sequence of monotonically improving policies and value functions [74].

Monte Carlo methods We now turn to the problem of estimating value functions. Since the value of a state is its expected return starting from that state, an obvious way to estimate it is from experience: simply average the returns observed after visits to that state. This is the *Monte Carlo method*. Unlike dynamic programming approaches, which compute the value function by solving the Bellman equation (Equation 2.17), Monte Carlo methods require no knowledge of the transition probabilities $\mathcal{P}^{a}_{ss'}$ or expected rewards $\mathcal{R}^{a}_{ss'}$.

However, these methods use samples inefficiently because we use a long trajectory to improve the estimate of just the single state-action pair that started the trajectory. This means that the methods converge slowly, and generally work only for small, finite MDPs [74].

Temporal difference methods Temporal difference methods [74] are similar to Monte Carlo methods, but they update value estimates based in part on other learned estimates, rather than waiting for the final outcome.

The simplest method, TD(0), updates the value function after each step with:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{2.20}$$

where α is a constant step-size parameter. In other words, we improve our estimate of the value of s_t using both the instantaneous reward and the expected value of the next state.

Some more sophisticated temporal difference methods include SARSA [23], Q-learning [49], and actor-critic methods [53].

Policy gradient methods Policy gradient methods frame policy optimization as a stochastic optimization problem and perform a local search. These methods are quite versatile: Kohl and Stone [38] performed policy gradient estimation for a non-MDP with no notion of a state, but which instead has a direct mapping from policy parameters to expected return. Fast convergence is seen with vanilla and natural policy gradients [52], and methods like natural actor-critic [53] achieve good results by combining policy gradients with temporal difference learning.

Chapter 3

Control of Elastodynamic Skin

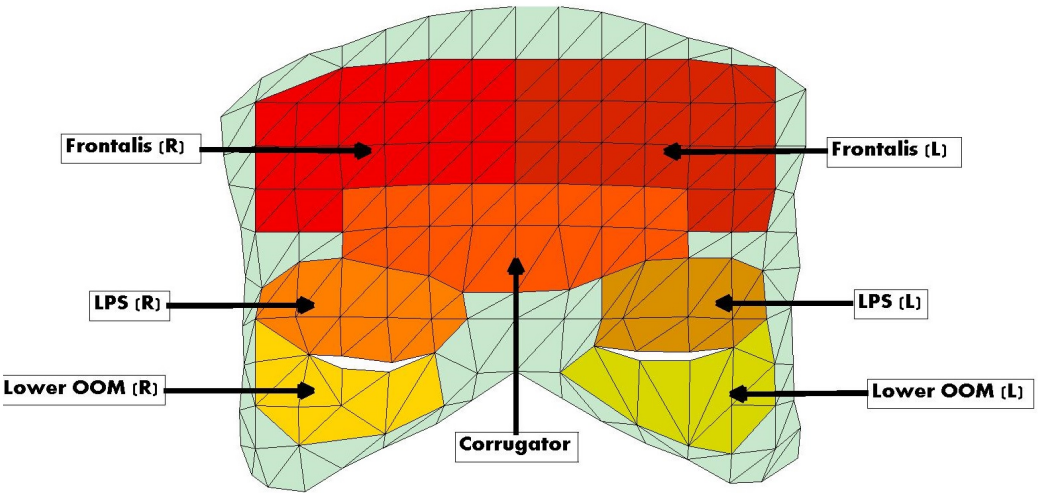
This chapter describes the control of a quasistatic elastodynamic model of the skin of the human mid-face. This model was developed by Debanga Raj Neog under the supervision of Dinesh Pai for Neog's forthcoming doctoral thesis, and is used here with permission. Section 3.1 provides a high-level overview of the simulator, while Section 3.2 and Section 3.3 present two methods for its control.

3.1 System Description

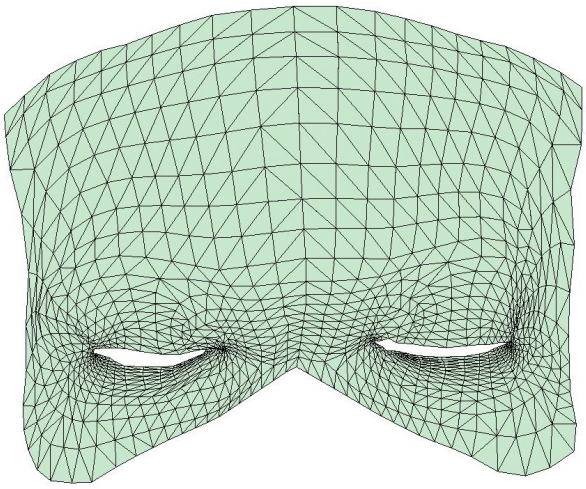
The model (Figure 3.1) is a triangular mesh of the human midface, which is controlled by specifying the activation levels of seven facial muscles: right and left frontalis; corrugator; right and left levator palpebrae superioris (LPS); and right and left lower orbicularis oculi (OOM).

The skin is modelled as a two-dimensional hyperelastic membrane and represented with an Eulerian discretization [42]. Unlike other representations, this approach avoids the need to constrain the skin to lie on the body surface and allows the use of a single mesh to represent both the body and the skin.

The face has two major types of muscles: linear and sphincter [76]. In linear muscles, the fibers lie in a line, and the muscle produces force along the fiber direction. Sphincter muscles' fibers contract tangentially, producing the net force along the radial direction and closing the orifice in the muscle. Since both types of muscles are sheet-like, the 2D formulation is suitable for simulation. The muscle



(a)



(b)

Figure 3.1: (a) low-resolution skin mesh with location of muscle sheets labelled. (b) high-resolution skin mesh.

model is anisotropic and hyperelastic: the passive muscle stress-strain profile is shifted along the muscle fiber direction with an amount proportional to the muscle's

activation level (see Figure 3.2).

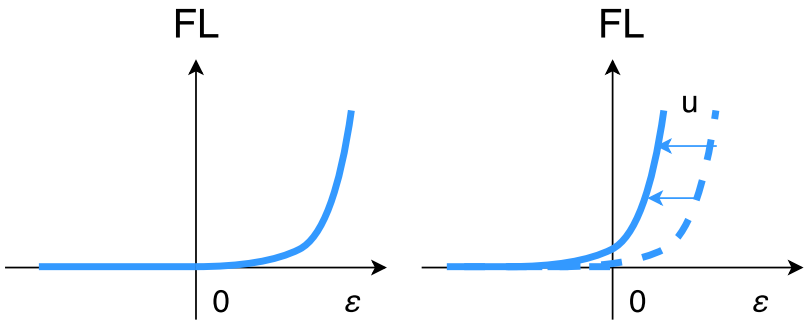


Figure 3.2: Example of passive and active muscle force-length (FL) curves. *Left*: passive force. *Right*: active force, given by shifting passive curve leftward according to the muscle activation u.

The model is quasistatic: that is, given a set of muscle activations, it generates a simulated pose without the need of a full dynamic simulation. If we supply a constant muscle activation, the model will reach a steady-state configuration when the stress due to the muscle activations is balanced by the external forces from the attached tendons and muscles. Steady-state configurations are computed in the 2-dimensional skin atlas space, with corresponding 3-dimensional mesh configurations obtained by a barycentric mapping.

3.2 A Supervised Learning Approach

Because of the model's quasistatic formulation, we can fully describe its behaviour with a mapping from muscle activations to resulting mesh configurations. This means we can address the control problem with supervised learning methods. The muscle activation is given by a 7-dimensional vector u, with each $u_i \in [0,1]$ representing the activation level of one of the seven facial muscles. Equilibrium configurations are computed in the 2-dimensional skin atlas space; this is the space we will use for learning and prediction. This will be faster to predict than 3-dimensional mesh positions and, because the skin is modelled as a 2-dimensional surface, there is no loss of information in the 2D configuration compared to the 3D. The skin atlas consists of 163 points, which means the dimension of our output is 326. We

refer to the mesh configuration as the *pose*, denoted by the variable P.

3.2.1 Regression Methods

Training data We perform our sampling in the 7-dimensional activation space using a Latin hypercube design. Recall from Section 2.3.2 that a Latin hypercube design is one that distributes points evenly over each input dimension separately. Designs are generated with the *maximin criterion* [37], which aims to maximize the minimum distance between points. In all the demonstrations in this section, we use 350 training points.

Kriging prediction We model the mapping $u \to P$ using kriging, which was discussed in Section 2.3.1. Because kriging methods assume scalar output variables, we fit a separate predictor to each component of P. We assume that each component P_i of our response P is of the form $P_i(u) = \sum_{j=1}^{k} \beta_{i,j}\phi_j(u) + Z(u)$, where ϕ are previously specified regression functions, k is the number of regression functions, and $Z(\cdot)$ is a random process with mean 0 and covariance function $\sigma^2 R(w,u)$. For each component P_i we use the same regression and correlation functions, but we fit different least-squares parameters β.

The predicted value $\hat{P}_i$ of u_{new}, given inputs $\mathcal{U} = u^{(1)}, ..., u^{(n)}$ and corresponding outputs $P_{U,i} = P_i(u^{(1)}), ..., P_i(u^{(n)})$, is given by:

$$\hat{P}_i(u_{new}) = \phi'(u_{new})\hat{\beta}_i + r'(u_{new})R^{-1}(P_{U,i} - \Phi\hat{\beta}_i) \tag{3.1}$$

where

$$\hat{\beta}_i = (\Phi' R^{-1}\Phi)^{-1}\Phi' R^{-1} P_{U,i} \tag{3.2}$$

In Equation 3.1, $\phi(u_{new}) = [\phi_1(u_{new}), ..., \phi_k(u_{new})]'$ denotes the vector of the k functions in the regression; $\Phi = \begin{pmatrix} \phi'(u^{(1)}) \\ \vdots \\ \phi'(u^{(n)}) \end{pmatrix}$ the $n \times k$ expanded design matrix; $R = \{R(u^{(i)}, u^{(j)})\}, 1 \le i \le n, 1 \le j \le n$ the $n \times n$ matrix of stochastic-process cor-

relations between Z values at the design sites; and $r(u_{new}) = [R(u^{(1)}, u_{new}), ..., R(u^{(n)}, u_{new})]'$ the vector of correlations between Z values at the design sites and an untried input u_{new}. For our regression functions, we use $\phi(u) = [u_1, u_2, ..., u_k]$, where k is the dimension of u. In other words, we model our response P as a simple linear function of u.

Choice of correlation function A question we have not yet addressed is the value of the correlation function, $R(u^{(i)}, u^{(j)})$. These functions generally take the form $R(w,u) = \prod_{j=1}^{m} g(w_j - u_j)$ for some function g, where m is the dimension of the input data [61]. A common choice is the *exponential correlation function*, of the form:

$$R(w,u) = \prod_{j=1}^{m} \exp(-\theta_j |w_j - u_j|^{\rho}) \qquad 0 < \rho \leq 2 \tag{3.3}$$

where θ, ρ are correlation parameters. Another option is the *linear correlation function*, which results in a linear spline:

$$R(w,u) = \prod_{j=1}^{m} (1 - \theta_j |w_j - u_j|)_+ \tag{3.4}$$

Lastly, the correlation function

$$R(w,u) = \prod_{j=1}^{m} [1 - a_j (w_j - u_j)^2 + b_j |w_j - u_j|^3] \tag{3.5}$$

gives a cubic spline for certain values of a_j and b_j.

For all problems in this chapter, we use the exponential correlation function. In initial experiments, we will assume known values for the parameters θ, ρ (practically speaking, this means trying different values until we obtain good results). Afterwards, we will compare the accuracy of prediction with hand-chosen parameters to prediction with the Maximum Likelihood Estimate (MLE) parameters.

Maximum Likelihood Estimation of correlation parameters MLE methods [63] assume that the correlation function is of the form $R = R(\cdot|\psi)$, where ψ is a

vector of parameters. In the case of the exponential correlation function, $\psi \equiv \{\theta_1, ..., \theta_m, \rho\}$. The MLE chooses the parameter vector $\hat{\psi}$ that minimizes

$$n\log\hat{\sigma}_z^2(\psi) + \log(\det(R(\psi))) \tag{3.6}$$

where $R(\psi)$ is the correlation matrix built from the existing training data with parameters ψ, and

$$\hat{\sigma}_z^2 = \hat{\sigma}_z^2(\psi) = \frac{1}{n}(P_{U,i} - \Phi\hat{\beta})^T R^{-1}(\psi)(P_{U,i} - \Phi\hat{\beta}) \tag{3.7}$$

For large n, it would be impractical to compute the gradient of Equation 3.6, so we perform the optimization for $\hat{\psi}$ using a quasi-Newton method [3].

Measuring prediction error The next two subsections deal with forward and inverse prediction. Forward prediction computes the map from muscle activations to poses; inverse prediction maps from poses to the muscle activations that yield them. Here we describe how we assess the quality of predictions. For inverse prediction, we have a pose that was generated by an activation u_{true}; if our inverse predictor returns $\hat{u}$ as the estimate of u_{true}, we define the *activation error* by:

$$\text{Err}_{\text{act}}(\hat{u}, u_{\text{true}}) = \frac{||\hat{u} - u_{\text{true}}||}{||u_{\text{true}}||} \tag{3.8}$$

Note that when we omit the subscript from vector norms, we are referring to the L_2-norm.

For both forward and inverse prediction, it is useful to have a measure of the error between poses. This is obviously necessary to assess the quality of forward predictions, which predict a pose given a set of muscle activations. However, it will also be a useful metric for inverse prediction. Recall that for control problems, we want the plant to exhibit some desired behaviour. In this case, we want to be able to use inverse predictions to mimic poses. So, if the simulator maps $u_{\text{true}} \rightarrow P_{\text{true}}$, and our inverse predictor estimates an activation $\hat{u}$ corresponding to P_{true}, we can compute the pose that results from applying $\hat{u}$ as a simulator input: $\hat{u} \rightarrow \hat{P}$. We then assess the quality of the choice $\hat{u}$ by computing the error between P_{true} and $\hat{P}$. We denote this quantity by $\text{Err}_{\text{pose}}(\hat{P}, P_{\text{true}})$.

To describe how we compute Err_{pose}, recall that we perform all pose predictions on the 163 points in the (2D) physical atlas space. Since we would like a notion of relative error, rather than absolute error, we ran the simulator with 500,000 random activations and approximate the range of each point as follows:

- Define point i on the mesh by p_i, for $1 \leq i \leq 163$, with coordinates $[p_{i,(1)}, p_{i,(2)}]$
- Over the 500,000 trials, obtain minimum and maximum possible values for each coordinate: $\{p^{\min}_{i,(1)}, p^{\max}_{i,(1)}, p^{\min}_{i,(2)}, p^{\max}_{i,(2)}\}$
- Define $r(p_i) = ||p^{\max}_{i,(1)} - p^{\min}_{i,(1)}, p^{\max}_{i,(2)} - p^{\min}_{i,(2)}||$

114 of the 163 points were near-stationary (defined as having $r(p_i) < 0.05$). To avoid artificially low error estimates, we only compute the pose error over the 49 points with a higher range of movement. These points are shown in Figure 3.3.

Given two poses $P = \begin{pmatrix} P_1 \\ \vdots \\ P_{163} \end{pmatrix}$ and $\hat{P} = \begin{pmatrix} \hat{P}_1 \\ \vdots \\ \hat{P}_{163} \end{pmatrix}$, let $\tilde{P} = \begin{pmatrix} \tilde{P}_1 \\ \vdots \\ \tilde{P}_{49} \end{pmatrix}$ and $\tilde{\hat{P}} = \begin{pmatrix} \tilde{\hat{P}}_1 \\ \vdots \\ \tilde{\hat{P}}_{49} \end{pmatrix}$ denote the respective reduced matrices consisting of the 49 non-stationary points.

We define the *average relative per-point error* by:

$$\text{Err}^{\text{avg}}_{\text{pose}}(P, \hat{P}) = \frac{1}{49} \sum_{i=1}^{49} \frac{||\tilde{\hat{P}}_i - \tilde{P}_i||}{r(\tilde{p}_i)} \tag{3.9}$$

And the *maximum relative per-point error* by:

$$\text{Err}^{\max}_{\text{pose}}(P, \hat{P}) = \max_{1 \leq i \leq 49} \frac{||\tilde{\hat{P}}_i - \tilde{P}_i||}{r(\tilde{p}_i)} \tag{3.10}$$

3.2.2 Forward Prediction

Here we attempt to learn the mapping from input activations to resulting poses. Since kriging assumes a scalar output, we build a separate predictor for each di-

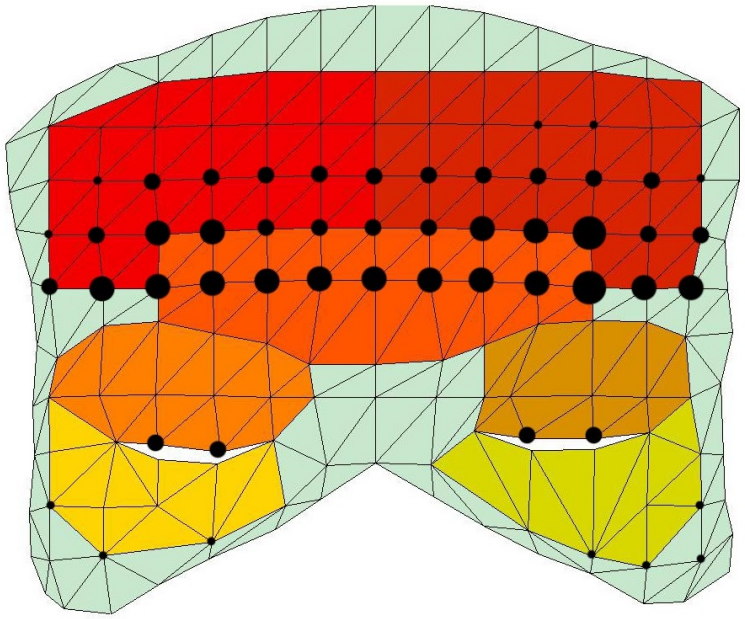
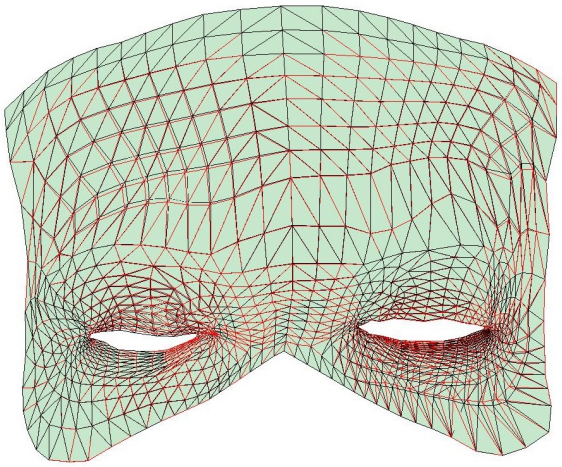


Figure 3.3: Depiction of the 49 low-resolution mesh points for which $r(p_i) > 0.05$. The black circles denote the points, with larger circles representing points with proportionally larger ranges of movement. The coloured areas of the mesh represent the locations of the different muscle sheets.

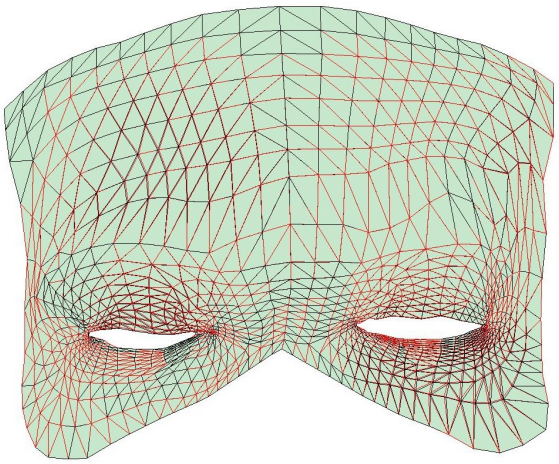
mension of the pose. This is a valid method for multidimensional regression, but will ignore relationships between different dimensions of P. We could probably achieve better results with methods that learn in parallel for different dimensions of vector output [10], but such methods are beyond the scope of this thesis.

Manual correlation parameters – results We use an exponential correlation function with parameters $\theta_i = 2$ for all i, $\rho = 2$. Using 350 training points in a Latin hypercube design and evaluating on 50 testing points, the mean value of the average pose error over the testing points was 0.1093 and the mean value of the maximum pose error was 0.2478. This means that, for the average point in the test set, each point of the predicted pose deviated by 10.93% (relative to its full movement range) on average, and the worst point deviated by 24.78%. Graphical depiction of

a specific example from the testing set is given in Figure 3.4a.



(a)



(b)

Figure 3.4: An example of forward prediction with manually selected correlation parameters (a) and MLE correlation parameters (b). The true pose is shown in black and the predicted poses are overlaid in red. (a) The average relative pose error is 0.062 and the maximum error is 0.3152 – notice the large deviations in the bottom right. (b) The average pose error is 0.0107, and the maximum is 0.0570.

	Pose Error	
	Average	Maximum
Manual parameters	0.1093	0.2478
MLE parameters	0.0246	0.060

Table 3.1: Table of mean pose errors over test set for forward prediction with manual and MLE correlation parameters. Reported errors represent the average and maximum relative errors per mesh point in the testing set.

MLE correlation parameters – results We use the same training and testing data here as for the previous example. The MLE gives correlation parameters $\theta_i = 0.669$ for all i, and $\rho = 1.95$. Unsurprisingly, these parameters result in better prediction accuracy than the manual correlation parameters: average and maximum per-point errors of 0.0243 and 0.06, respectively. For an illustration, see Figure 3.4b.

3.2.3 Inverse Prediction

Here we deal with the control (or inverse prediction) problem: we wish to learn the mapping from poses P to activations u. As in the previous section, we gather our training data in a Latin hypercube over the activation space.

We use only the 49 non-stationary mesh points for prediction, and we standardize each dimension so that all observations cover the range $[0,1]$. As before, the quantity we are predicting is a vector (in this case, the 7-dimensional activation), so we fit a separate kriging predictor to each dimension of the activation. Because we have 98 input dimensions (49 points times 2 dimensions per point), this could make it challenging to find the MLE correlation parameters. So, we make the simplifying assumption that, for each of our seven predictors, $\theta_i = \theta$ for all i – in other words, we assume that the vector θ is constant for all 98 dimensions of the pose.

This means we have seven predictors (one for each dimension u_i), each with two correlation parameters θ and ρ. The respective MLE values of θ for each of the seven predictors are $[2.02, 2.02, 9.89, 3.03, 2.63, 10.30, 2.02]$ and the corresponding values of ρ are $[0.74, 1.1, 2, 0.84, 0.84, 2, 1.18]$.

To test our predictors, we draw 50 activations from a random uniform distribution and apply them to the simulator to obtain the corresponding poses. We apply the predictor to the poses to obtain predicted activations and clamp the results so

they are all in the legal range $[0, 1]$. We then measure both how close the predicted activations are to the true activations (activation error), as well as how closely the poses resulting from applying the predicted activations match the target poses in the testing set (pose error). The results are:

- Average relative activation error: 0.1423
- Average relative per-point pose error (mean over points in test set): 0.0183
- Maximum relative per-point pose error (mean over points in test set): 0.1132

At first glance, these results seem satisfactory. When we mimic a target pose, the average point on the mesh deviates by 1.8%, and in typical cases, no single point is off by more than 11.3%. There is, however, one class of pose on which this method performs quite poorly, and this is for points achieved with mostly saturated activations – i.e. with most or all values of u very close to 0 or 1. This is perhaps to be expected, as these points are at the boundaries of the input space, and therefore at or beyond the boundaries of the training set.

To illustrate this point, let us consider the same inverse predictor obtained earlier, but instead of testing it on poses resulting from random activations, we will test it on:

1. The 7 possible ways to fully activate one muscle while setting the others to 0, and
2. the 21 possible ways to fully activate two muscles while setting the others to 0

In this case, the results are:

- Average relative activation error: 0.4856
- Average relative per-point pose error (mean over points in test set): 0.2862
- Maximum relative per-point pose error (mean over points in test set): 0.7394

Clearly, performance is much worse than in the random case. Consider the example given in Figure 3.5, in which the target is obtained by fully activating the corrugator and right lower OOM. The correct activation to generate this pose is $[0,0,1,0,0,0,1]$, while the inverse predictor generates the activation $[0,1,1,1,0.39,0,1]$. The average and maximum per-point errors in the pose are 0.6363 and 0.9999, respectively. That second number merits some attention: recall that the maximum per-point error is a measure of how far off-target the worst point on the mesh is, relative to its range of motion. The fact that this number is 0.9999 tells us that there is at least one point in the mesh that is *as far away from its target position as it could possibly be*.

3.3 Combining Regression and Local Search

The poor performance of our control method for poses generated by boundary cases in the activation space tells us that we need to improve the prediction method of Section 3.2. However, with any experimental design we select, points near the boundary of the input space will always have less information for an interpolated method (such as kriging) to use to make a prediction. If we want adequate boundary performance without covering the entire boundary during training, we need to move beyond standard supervised learning methods. Hence, this section presents a method for adding local optimization to the regression method in Section 3.2 to improve control performance.

3.3.1 Method

Problem formulation To borrow a term from reinforcement learning, we define our *policy*, denoted by U, by the choice of activations: $U = \{u_1, ..., u_7\}$. Let P_{target} denote our pose, and P_U denote the pose obtained by running the simulator with activation U. We then define our value function, which we seek to maximize, by:

$$V^U = -\frac{1}{2}\left(\text{Err}^{\text{avg}}_{\text{pose}}(P_U, P_{\text{target}}) + \text{Err}^{\text{max}}_{\text{pose}}(P_U, P_{\text{target}})\right) \tag{3.11}$$

with $\text{Err}^{\text{avg}}_{\text{pose}}(P_U, P_{\text{target}})$ and $\text{Err}^{\text{max}}_{\text{pose}}(P_U, P_{\text{target}})$ denoting the average and maximum per-point errors, defined in Equation 3.9 and Equation 3.10, respectively.

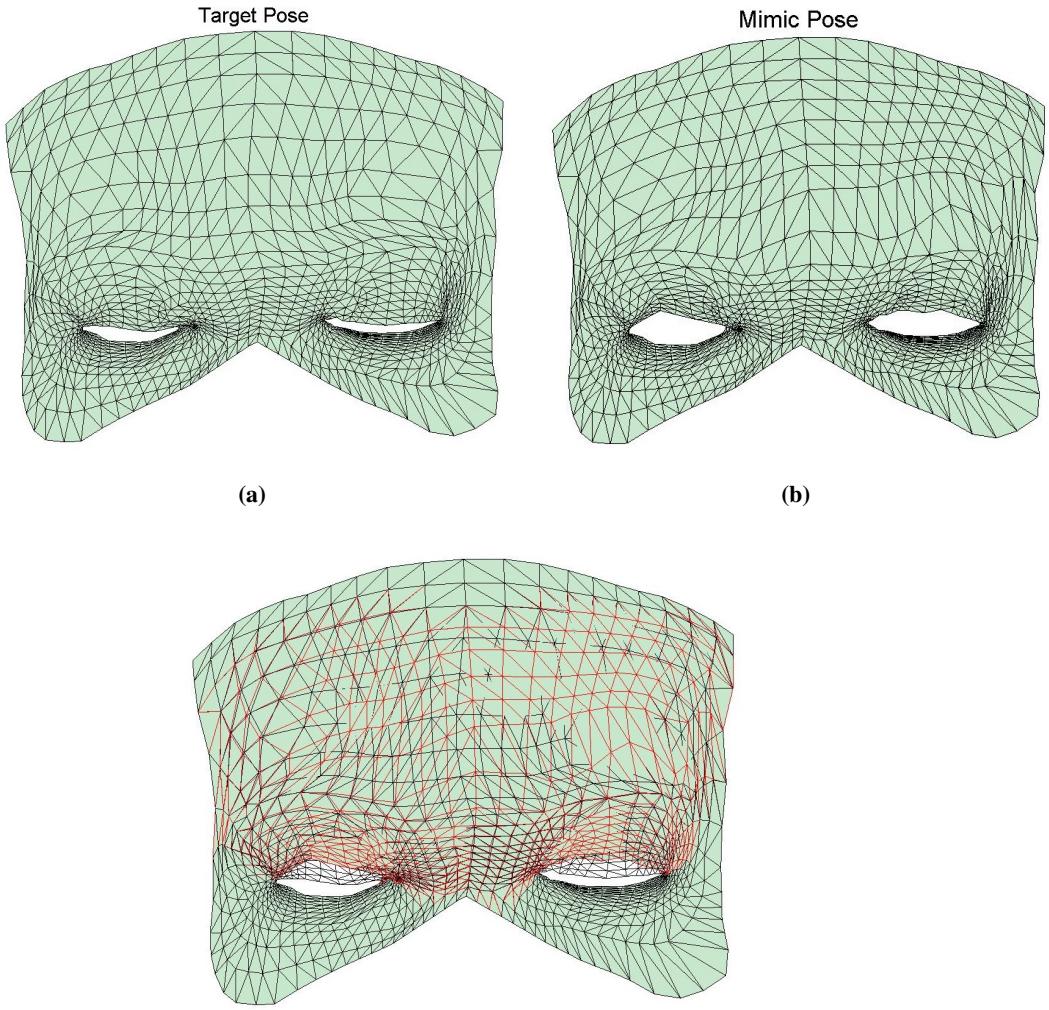


Figure 3.5: Inverse prediction on pose generated by full activation of the corrugator and right lower OOM. (a) Target position. (b) Position obtained by predictor. (c) Overlay of the two, with the target in black and predicted position in red.

Gradient descent To compute a policy U to maximize the value function, we adapt Kohl and Stone's [38] method to estimate the *policy gradient*. This is the gradient of the value function with respect to the parameters of U; once it is com-

puted, we can use gradient descent to improve our estimate.

We begin with an initial activation $U = \{u_1, ..., u_7\}$, obtained by kriging. We then estimate the partial derivative of V^U with respect to each parameter. We do this by first evaluating t randomly generated policies $\{W_1, W_2, ..., W_t\}$ near U – specifically, each $W_i = \{u_1 + \Delta_1, ..., u_7 + \Delta_7\}$, where each Δ_j is chosen randomly to be either $-\varepsilon_j, 0$ or $+\varepsilon_j$. We require all u_j to be in the range $[0,1]$, so if $u_j < \varepsilon_j$, then we choose Δ_j to be one of $\{0, +\varepsilon_j, -u_j\}$, and if $u_j > 1 - \varepsilon_j$, we choose Δ_j from $\{-\varepsilon_j, 0, 1 - u_j\}$. This will ensure that all dimensions of U remain in the legal range (note that, since our kriging predictor clamps all values to lie in $[0,1]$, we are guaranteed to begin our iterations with legal values of U).

After evaluating the value V^W of each W_i, we group each W_i into one of three sets for each dimension ($n = 1, ..., 7$):

$$W_i \in \begin{cases} S_{+\varepsilon,n} & \text{if the } n^{\text{th}} \text{ parameter perturbation of } W_i \text{ is positive} \\ S_{+0,n} & \text{if the } n^{\text{th}} \text{ parameter perturbation of } W_i \text{ is 0} \\ S_{-\varepsilon,n} & \text{if the } n^{\text{th}} \text{ parameter perturbation of } W_i \text{ is negative} \end{cases} \tag{3.12}$$

We then compute the average scores $Avg_{+\varepsilon,n}$, $Avg_{+0,n}$ and $Avg_{-\varepsilon,n}$ of $S_{+\varepsilon,n}$, $S_{+0,n}$ and $S_{-\varepsilon,n}$, respectively, to give us an estimate of the value of altering the n^{th} parameter by ε. We then use the average scores to compute a 7-dimensional adjustment vector D where:

$$D_n = \begin{cases} 0 & \text{if } Avg_{+0,n} > Avg_{-\varepsilon,n} \text{ and } Avg_{+0,n} > Avg_{+\varepsilon,n} \\ Avg_{+\varepsilon,n} - Avg_{-\varepsilon,n} & \text{otherwise} \end{cases} \tag{3.13}$$

In Kohl and Stone [38], the adjustment vector was normalized and multiplied by a scalar stepsize parameter η, and then added to the policy. However, we found that this often led to a policy with lower value V^U than the previous one and sometimes prevented the iterations from converging to a suitable value of U. So, we added a check to see if the value of $U + \eta * \frac{D}{||D||}$ is higher than the value of U. If so, we accept the update and set $U \leftarrow U + \eta * \frac{D}{||D||}$, and then continue with the iter-

	Pose Error	
	Average	Maximum
Regression only	0.2862	0.7394
Local search only (random starts)	0.0156	0.0316
Regression + Local search	0.0003	0.0028

Table 3.2: Table of mean pose errors over 28 boundary points for regression method, policy gradient (PG) with random starts, and regression combined with policy gradients. Reported errors represent the average and maximum relative errors per mesh point in the testing set.

ations. However, if the value function decreased after adding the displacement, we reject the proposed update – in other words, we implement greedy policy iterations by choosing the action (either acceptance or rejection of a proposed update) that leads to a higher value V^U. We attempt another update after decreasing η by a factor of $\sqrt{2}$, to account for the possibility that the stepsize is too large. Once we found a D which improved the results, we would reset η to its original value. If we had 10 consecutive failed updates, we would stop the algorithm. For pseudocode, see Algorithm 1.

3.3.2 Results

We perform gradient descent updates on the 28 boundary cases discussed in Section 3.2.3, with parameters $t = 30, \eta_0 = 0.1$, and $\varepsilon_i = 0.01$ for all i. Prediction accuracy increases considerably (see rows 1 and 3 of Table 3.2); in fact, for 20 of the 28 cases, the local search converges exactly to the correct activations. This is the case with the example given in Figure 3.5, which reaches the correct activation (and therefore 0 mesh error) after 114 iterations (see Figure 3.6).

Do we need the regression step? Given the effectiveness of the local optimization, a natural question is to ask is whether we even need to perform the regression step first. To investigate, we compare the performance of the local search method with random initial policies to its performance when starting with the values obtained by kriging. We compare on the same set of 28 test points used earlier.

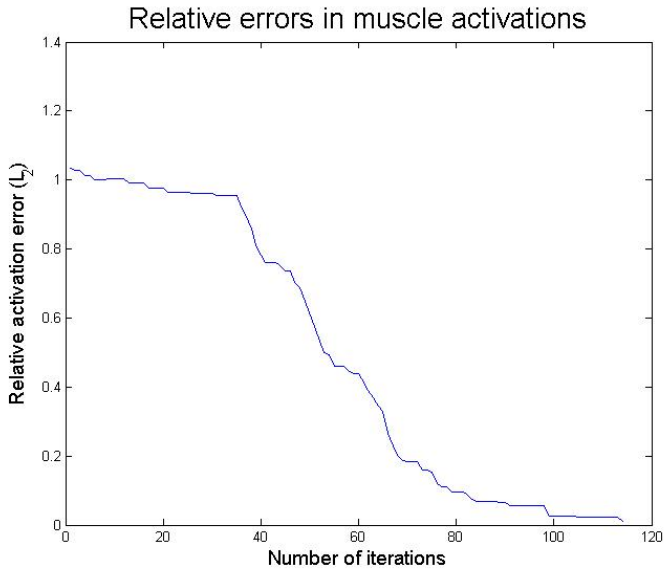
We start by considering the pose errors for both methods. Results are shown in

Algorithm 1 Gradient descent for inverse prediction

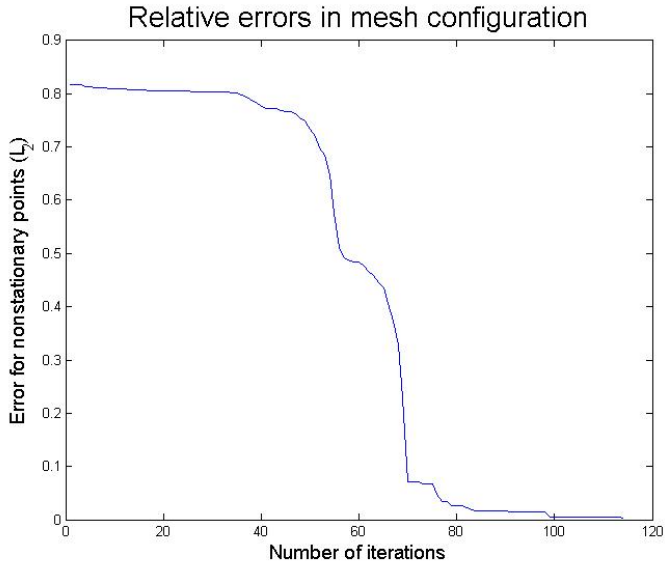
$U \leftarrow$ initial activation
$nIter \leftarrow 0$
$\eta \leftarrow \eta_0$ ▷ Set to initial stepsize value
$nFails \leftarrow 0$ ▷ Number of consecutive rejected updates
while $nIter < maxIter$ and $nFails < maxFails$ **do**
    $\{W_1, W_2, ..., W_t\} \leftarrow t$ random perturbations of U
    for $n = 1$ to $dim(U)$ **do**
        $Avg_{+\varepsilon,n} \leftarrow$ average score for W_i that have positive perturbation in
        dimension n
        $Avg_{-\varepsilon,n} \leftarrow$ average score for W_i that have negative perturbation in
        dimension n
        $Avg_{+0,n} \leftarrow$ average score for W_i that have 0 perturbation in
        dimension n
        if $Avg_{+0,n} > Avg_{+\varepsilon,n}$ and $Avg_{+0,n} > Avg_{-\varepsilon,n}$ **then**
            $D_n \leftarrow 0$
        else
            $D_n \leftarrow Avg_{+\varepsilon,n} - Avg_{-\varepsilon,n}$
        end if
    end for
    $D \leftarrow \frac{D}{||D||} * \eta$
    if $V^{U+D} < V^U$ **then** ▷ Update worsens policy
        $nFails \leftarrow nFails + 1$
        $\eta \leftarrow \frac{\eta}{\sqrt{2}}$
    else
        $U \leftarrow U + D$ ▷ Accept update
        $nFails \leftarrow 0$ ▷ Reset failure counter
        $\eta \leftarrow \eta_0$ ▷ Reset stepsize
    end if
    $nIter \leftarrow nIter + 1$
end while

Table 3.2. As we can see, local with random starts performs reasonably well – still better than regression without further optimization – but the error is higher than when we combine both methods.

Another consideration is time to convergence. The simulator and learning algorithms were implemented in MATLAB and run on a PC with an AMD A10-6800K 4.10 GHz processor and 10 GB of memory. Each gradient descent update requires computing the objective function for 30 different activations, which involves computing the mesh position for a given activation 30 times. This calculation takes between 1 and 2 seconds, so the updates are somewhat expensive. With random starts, the gradient descent method required an average of 128.61 iterations, taking 35.68 minutes; with the kriging starts, it's 73.57 iterations and 19.98 minutes. So, we gain a significant efficiency boost with the kriging starts. Given that it only takes 75 seconds to build a predictor and 0.6 seconds to predict the value for a new point, the kriging starts are certainly worth our time.



(a)



(b)

Figure 3.6: Example of local search updates on initial policy obtained by regression. (a) Plot of relative activation errors. (b) Plot of the mean of the average and maximum per-point pose errors (the negative of the value function).

Chapter 4

Reaching Control

This chapter deals with reaching control for a model of the human index finger. We frame this as an equilibrium control problem, which we discussed in Section 2.2.1. To recap, an equilibrium point is an state-activation pair $\{x_{eq}, u_{eq}\}$ with the property that $f(x_{eq}, u_{eq}) = x_{eq}$. In other words, the activation u_{eq} keeps the system in equilibrium at the point x_{eq}. With equilibrium control, we attempt to reach the state x_{eq} by applying the constant activation u_{eq}, with the rationale that an activation that stabilizes the plant at a given point may be all we need to drive the plant to that point from an arbitrary starting state.

This chapter is similar to Chapter 3 in that we obtain a mapping from activations to final (equilibrium) configurations, and we don't consider the dynamics at intermediate points. In this chapter, we deal with supervised learning approaches for reaching control. We will deal with general-purpose methods for more complicated control tasks in Chapter 5.

4.1 Plant Description: Human Index Finger

The plant we use in this chapter is a biomechanical simulation of a human index finger, described in detail in [41]. It was originally created by Duo Li and Shinjiro Sueda under the supervision of Dinesh Pai, and developed further by Prashant Sachdeva [60]. It is a tendon-based simulator built using the *Strands* framework [72], in which bones are rigid bodies controlled by tendons, which are represented

by thin strands. The motion of the tendon is constrained by the network of sheaths and pulleys present in the hand. These constraints are handled using the approach of Sueda et al. [73].

Model description The model (shown in Figure 4.1) has four bones: the metacarpal bone; the proximal phalanx, which is connected to the metacarpal bone via the metacarpophalangeal (MCP) joint; the middle phalanx, which is connected to the proximal phalanx by the proximal interphalangeal (PIP) joint; and the distal phalanx, connected to the middle phalanx by the distal interphalangeal (DIP) joint. The plant has four degrees of freedom: flexion-extension for all joints, and abduction-adduction for the MCP. The finger is controlled by the activation of six muscles: the distal interosseous (DI), extensor digitorum communis (EDC), flexor digitorum profundus (FDP), flexor digitorum superficialis (FDS), lumbrical, and palmar interosseous (PI). Thus, the activation for the plant is given by a six-dimensional vector u, where $0 \leq u_i \leq 1$ for $i = 1...6$.

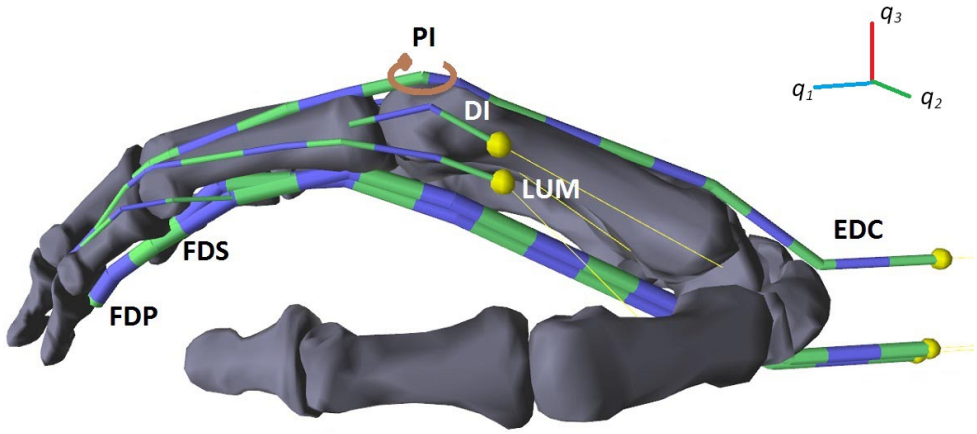


Figure 4.1: Biomechanical model of the human index finger.

Muscle model The force exerted by a muscle is assumed to be a sum of the passive and active forces: $f = f_P + f_A$. The muscle model relates the force at the origin of the tendon to the tendon excursion (i.e. the displacement of the tendon origin). Passive and active forces are modelled as piecewise linear functions [60], as shown in Figure 4.2. The passive force consists of a horizontal line concatenated with a positive-sloping line beginning at 0 force, and the active force is the piecewise linear curve shifted by an amount proportional to the muscle's activation. Thus, the muscle model is equivalent to that of the skin simulator of Chapter 3, but with the simplifying assumption of piecewise linearity.

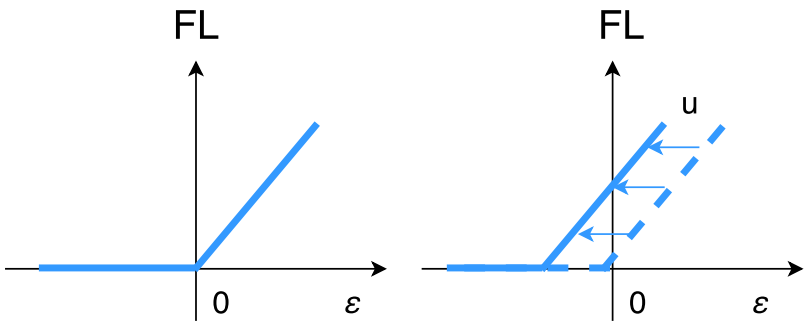


Figure 4.2: Passive and active muscle force-length (FL) curves. *Left*: passive force, given by a piecewise linear curve. *Right*: active force, given by shifting the piecewise linear curve leftward according to the muscle activation u.

4.2 Task Description

Configuration variable In all index finger control examples in this and the following chapter, our configuration variable q is chosen to be the 3-dimensional fingertip position, as in the work of Sueda et al. [72] and Fain [20]. Unlike joint angles or tendon excursions, which were used for control of the ACT hand in Deshpande et al. [18] and Malhotra et al. [46], respectively, the fingertip position doesn't fully describe the finger's configuration. There is, however, some physiological basis for this choice of configuration variable, as evidence suggests that humans do use fin-

gertip position directly for reaching movements [4]. This choice of configuration variable has the added advantage of making it more intuitive to specify controller goals.

Training and testing sets In the reaching control problem, we learn the mapping from configurations q to activations u. That is, for a target fingertip position, we estimate the activation that, when applied constantly to the plant, will take us there.

Our training data for this problem consists of 39,000 activation-configuration pairs, $\{u^{(i)}, q^{(i)}\}$. We obtained these by sampling from a uniform grid in the activation space, and we applied each activation $u^{(i)}$ to the plant until it reached a stable configuration $q^{(i)}$. A scatter plot of the fingertip positions in the training data set is shown in Figure 4.3. Note the clear non-convexity of the configuration space: this could cause problems for some supervised learning algorithms. An illustration of different plant configurations is shown in Figure 4.4.

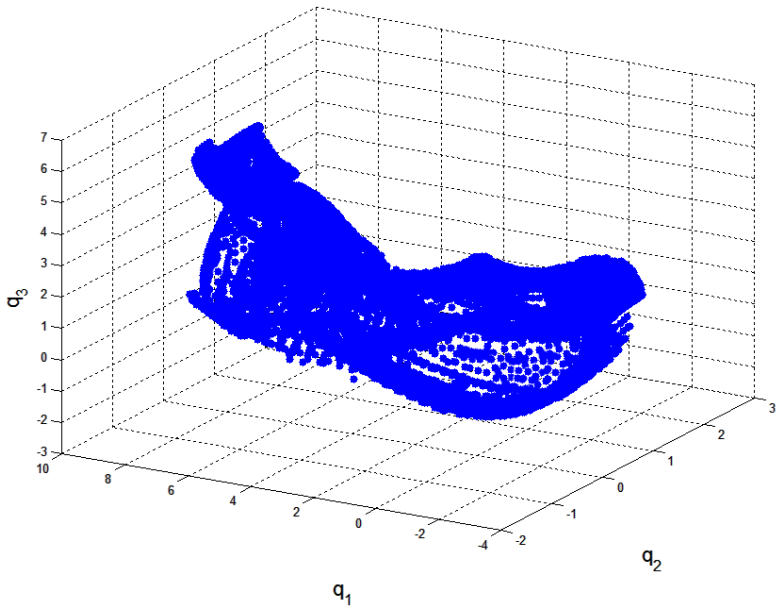


Figure 4.3: 3-dimensional scatter plot of plant configurations (i.e. fingertip positions) in the training data set for the reaching control problem. Axis units are in centimetres.

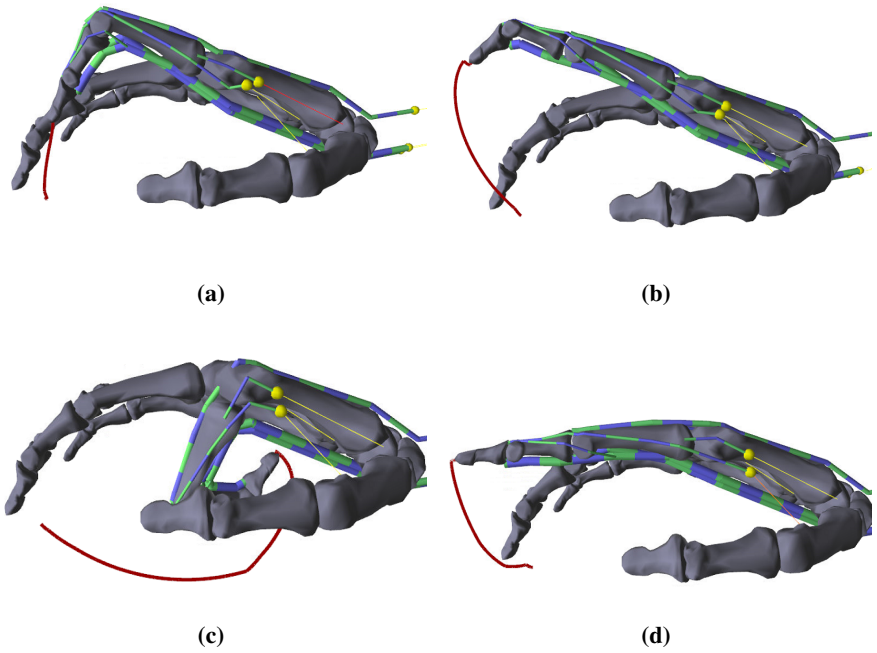


Figure 4.4: Different configurations of the index finger, with the plant's trajectory from the initial configuration shown in red. (a) full activation of the DI. (b) full activation of the EDC. (c) full activation of the FDS. (d) full activation of the EDC and half activation of the lumbrical.

To assess the quality of a predicted activation $\hat{u}$ for a target configuration q_r, we apply $\hat{u}$ to the plant until stabilization and measure the Euclidean distance (in centimetres) of the plant configuration from q_r.

For our test set, we would like plant configurations that are (a) not in the training set, and (b) in the space of reachable configurations. To achieve a high probability of a testing point meeting both criteria, we first choose a random configuration in the training set. We then take the average of this configuration and the three next configurations in the training data. Because of the order in which we performed the sampling, consecutive points in the training data are generally close together in activation space. This implies that the corresponding configurations are probably close together, as well. This means that the average of the four points is likely to be a reachable configuration. Because of the non-convexity of the configuration

space, we would have no guarantees of this if we took the average of completely random points (with consecutive points, we still don't have a guarantee, but we have a higher probability).

We gather 500 such points to create the test set. The training and testing data are the same for all experiments conducted in this chapter.

For all methods, we compare the build time for the data structure and the query time to predict the activation for a single point. All learning algorithms were implemented in MATLAB and run on a commodity PC with an Intel Core i5 3570 processor and 16GB of memory.

4.3 k-Nearest-Neighbours

4.3.1 Method

To estimate the activations leading to a target point q_r, we begin by computing q_r's k nearest neighbours in the training set, $\{q_{nn}^{(1)}, ..., q_{nn}^{(k)}\}$. We predict the $\hat{u}$ to lead the plant to q_r with a weighted average of the activations $\{u_{nn}^{(1)}, ..., u_{nn}^{(k)}\}$ corresponding to the configurations $\{q_{nn}^{(1)}, ..., q_{nn}^{(k)}\}$. The weight of each activation $u_{nn}^{(i)}$ is given by the reciprocal of the distance between q_r and $q_{nn}^{(i)}$. In the case where the distance between q_r and its nearest neighbour $q_{nn}^{(1)}$ is 0 – i.e. when q_r is in the training data – then we simply set $\hat{u} = u_{nn}^{(1)}$. Hence, this method interpolates the training data.

Locality-sensitive hashing We improve the efficiency of our k-nearest-neighbour search using Locality-Sensitive Hashing (LSH) algorithms [34, 67]. This approach has space and query time that are both linear in the number of points to search.

These algorithms are based on *locality-sensitive* hashing functions, which possess the property that, for any two points $p, r \in \mathbb{R}^d$:

1. If $||p - r|| \leq D$ then $\Pr[h(r) = h(p)] \leq P_1$
2. If $||p - r|| \geq cD$ then $\Pr[h(r) = h(p)] \geq P_2$

where $P_1 > P_2$ and $c \geq 1$.

We concatenate several LSH functions to amplify the gap between P_1 and P_2. For parameters m and L, we choose L functions

$$g_j(r) = (h_{1,j}(r), ..., h_{m,j}(r)) \tag{4.1}$$

where the hash functions $h_{l,j}(1 \leq l \leq m, 1 \leq j \leq L)$ are chosen at random from a family of LSH functions.

We then construct a hash table structure by placing each point $q^{(i)}$ from the input set into a bucket $g_j(q^{(i)}), j = 1, ..., L$. To process a query point q_r, we scan through the buckets $g_1(q_r), ..., g_L(q_r)$ and retrieve the points stored in them. For each retrieved point, we compute the distance between it and q_r and report the k closest points.

For our hash functions, we project a point q onto a random 1-dimensional line in $\mathbb{R}^d$, which is then partitioned into uniform segments. The bucket into which a particular point is hashed corresponds to the index of the segment containing it. For Euclidean spaces, these functions are locality sensitive; for proof, we refer the reader to [1]. The algorithm parameters we use are $m = 20, L = 30$ and varying values of k.

4.3.2 Results

We examined the performance of k-nearest-neighbours for $k = 1, 4, 7, ..., 34$ on the training and testing datasets described in Section 4.2. We build a single LSH table that is used for all the predictors; the build time was 2.20 seconds. Summary statistics of the k-nearest-neighbours predictor performance are shown in Table 4.1. Surprisingly, prediction accuracy is highest with a single nearest neighbour: we discuss why this may be the case in Section 4.6.

4.4 Neural Networks

4.4.1 Method

We now attempt our reaching task using a feedforward neural network with Levenberg-Marquardt backpropagation [27]. This method is designed to approach second-order training speed without having to compute the Hessian matrix. The Hessian is approximated as $H = J^T J$, and the network gradient is computed as $g = J^T e$.

Neighbours	Error (cm)	Query time (s)	Activation size
1	0.19	0.0058	0.948
4	0.41	0.0059	0.939
7	0.40	0.0059	0.942
10	0.39	0.0058	0.942
13	0.41	0.0059	0.940
16	0.40	0.0059	0.941
19	0.36	0.0061	0.940
22	0.36	0.0061	0.940
25	0.36	0.0060	0.940
28	0.36	0.0062	0.940
31	0.34	0.0062	0.940
34	0.36	0.0065	0.939

Table 4.1: Table of summary statistics for reaching task using k-nearest-neighbours with locality-sensitive hashing. Error denotes the average distance (in centimetres) of final configurations from target positions; query time gives the time (in seconds) to compute the predicted activation $\hat{u}$ for a target q_r; and activation size is the average L_2-norm of the predicted activations $\hat{u}$.

Here J is the Jacobian matrix (much simpler to compute than the Hessian), which contains the first derivatives of the network errors with respect to the weights and biases, and e is the vector of network errors.

Let Θ_k denote the network weight and bias variables at training iteration k. We then use the approximation to the Hessian matrix in the update rule:

$$\Theta_{k+1} = \Theta_k - [J^T J + \mu I]^{-1} J^T e \tag{4.2}$$

The scalar parameter μ is set to an initial value of 0.001. After a successful step (i.e. where the sum of the squared network errors decreases), μ is multiplied by 0.1; after an unsuccessful step, μ is multiplied by 10.

Training is done in parallel on the GPU. Note that each time the network is trained, it can result in different solutions (and different build times) due to different initial weight and bias values, as well as different divisions of the data into training, validation and test sets.

Hidden layers	Error (cm)	Build time (s)	Query time (s)	Activation size
1	4.64	2035	0.012	0.820
2	2.26	3631	0.016	0.921
3	2.28	3066	0.015	0.988
4	1.91	10802	0.016	1.011
5	1.53	1449	0.014	1.045
6	1.53	10802	0.014	1.059
7	1.38	5328	0.015	1.103
8	1.54	10802	0.015	1.093
9	1.17	10802	0.014	1.116
10	1.05	3266	0.016	1.121
11	1.41	7053	0.020	1.155
12	1.24	7572	0.019	1.160

Table 4.2: Table of summary statistics for reaching task using neural networks. Error denotes the average distance (in centimetres) of final configurations from target positions, build time gives the neural network build time (in seconds); query time gives the time (in seconds) to compute the predicted activation $\hat{u}$ for a target q_r; and activation size is the average L_2-norm of the predicted activations $\hat{u}$.

4.4.2 Results

We tested the performance of neural networks with $n = 1, 2, ..., 12$ hidden layers. Results are given in Table 4.2. Several of the networks have a build time of 10,802 seconds: this is because we specified one of our stopping criteria for training to be a maximum of 3 hours' (10,800 seconds') training time.

4.5 Random Forests

4.5.1 Method

We now turn our attention to random forest predictors. In a survey of different classification methods, Fernandez et al. [21] found them to be the most effective class of predictor over a wide variety of datasets.

Our focus here is on random forests for regression, which consist of an ensemble of *regression trees* [29]. Suppose our data consists of n observations $\{q^{(i)}, u^{(i)}\}$,

each with three input dimensions $q^{(i)} = \{q_1^{(i)}, q_2^{(i)}, q_3^{(i)}\}$, and an output consisting of one dimension of the activation, $u_k^{(i)}$. (Because regression trees assume a scalar output, we fit predictors separately to each dimension of u).

Regression trees partition the input space into M regions and model the response as a constant, c_m, in each region, i.e.:

$$f(q) = \sum_{i=1}^{M} c_m I(q \in R_m) \tag{4.3}$$

where I is an indicator function for $q \in R_m$.

To minimize the mean-squared prediction error, the best $\hat{c}(m)$ is the average of the responses $u_k^{(i)}$ corresponding to the observations $q^{(i)}$ in the region R_m:

$$\hat{c}_m = \text{ave}(u_k^{(i)} | q^{(i)} \in R_m) \tag{4.4}$$

Finding the best binary partition in terms of the minimum sum of squares is generally infeasible, so we use a greedy algorithm. Starting with all data, we consider a splitting variable j and split point s, and define a pair of half-planes:

$$R_1(j,s) = \{Q | Q_j \leq s\} \text{ and } R_2(j,s) = \{Q | Q_j > s\} \tag{4.5}$$

We then seek the splitting variable j and split point s that solve

$$\min_{j,s} \left(\min_{c_1} \sum_{q^{(i)} \in R_1(j,s)} \left(u_k^{(i)} - c_1\right)^2 + \min_{c_2} \sum_{q^{(i)} \in R_2(j,s)} \left(u_k^{(i)} - c_2\right)^2 \right) \tag{4.6}$$

For any choice j,s the inner minimization is solved by setting $\hat{c}_1$ and $\hat{c}_2$ according to Equation 4.4. For each splitting variable j, the determination of the split point s can be determined easily; so, choosing the optimal j and s at each iteration is computationally feasible.

Next, we must determine how large to grow the tree. A tree that is too large may overfit the data, while one that is too small may be too simple to effectively capture the relationship between inputs and outputs. The most common approach is to stop when some minimum node size is reached [29]. In our case, we choose this value to be 5: in other words, there will be no fewer than 5 observations $q^{(i)}$ in any region R_j.

Trees	Error (cm)	Build time (s)	Query time (s)	Activation size
10	0.55	4.39	0.11	0.736
20	0.50	7.30	0.18	0.737
30	0.49	11.19	0.28	0.736
40	0.46	14.57	0.37	0.737
50	0.47	18.34	0.47	0.735
60	0.46	22.18	0.58	0.735
70	0.45	25.76	0.68	0.736
80	0.46	29.19	0.77	0.734
90	0.45	33.13	0.88	0.736
100	0.44	36.87	1.01	0.736
110	0.45	40.24	1.07	0.734
120	0.45	46.89	1.21	0.734

Table 4.3: Table of summary statistics for reaching task using random forests. Error denotes the average distance (in centimetres) of final configurations from target positions, build time gives the time to build the six random forests (in seconds); query time gives the time (in seconds) to compute the predicted activation $\hat{u}$ for a target q_r; and activation size is the average L_2-norm of the predicted activations $\hat{u}$.

A random forest is simply an ensemble of regression trees: the forest's prediction is the average of the trees'. This will result in estimates with a lower variance than those generated by a single tree.

4.5.2 Results

We tested the random forest predictor for $n = 10, 20, ..., 120$ regression trees. Numerical results are given in Table 4.3.

4.6 Methods Comparison/Discussion

We compare the best predictors from each of Sections 4.3 - 4.5 in Table 4.4. In a rather striking illustration of the fact that more sophisticated is not always better, we find that single-nearest-neighbour prediction outperforms all other predictors in terms of both accuracy and efficiency.

What accounts for this surprising fact? Well, notice that 1-nearest-neighbour

is the only method in which the predicted activation u isn't based on some combination of activations for multiple values of q. This could be important for this particular problem: due to the non-convexity of the configuration space, we could have two points q that are located close together in Euclidean space, but if points on the line segment joining them aren't in the legal configuration space, it could be that we require very different paths to get to them. So, two points that are close together may be quite dissimilar in how we can reach them.

Similarly, by basing our prediction on a single observation, we can mitigate the effects of redundant mappings: since we are considering a relationship between 6 muscle activations and a 3-dimensional position, we can assume that there are multiple activations u that map to the same or similar configurations q. (If nothing else, we can achieve this by co-activating agonist-antagonist muscle pairs). If two very different activations $u^{(1)}, u^{(2)}$ result in nearly identical configurations q, but a value $u^{(3)}$ on the path joining $u^{(1)}$ and $u^{(2)}$ maps to a different configuration, this could lead to problems for any methods that base their predictions on more than one activation-configuration pair.

The computation of equilibrium controls, which is what we have dealt with in these experiments, form an important component of the controller discussed in Chapter 5. If prediction accuracy was our only concern, random forests would still be a competitive option. While the average error is consistently higher than for k-nearest-neighbours, the random forests' performance seem to be less affected by the parameter choice (i.e. the number of trees) than k-nearest-neighbours. However, in the next chapter, we will examine problems where equilibrium controls will need to be computed every 3 milliseconds of controlled simulation: in this case, the random forests' 1-second query time is unacceptably slow. Thus, for the remainder of this thesis, nearest-neighbour will be our prediction method of choice.

Predictor	Error (cm)	Build time (s)	Query time (s)
k-**NN**	0.19	2.20	0.0058
NN	1.05	3266	0.027
RF	0.44	36.87	1.01

Table 4.4: Comparison of best predictors from each of the k-nearest-neighbours, neural network, and random forest experiments – parameter values for best predictors from each class are 1 neighbour, 10 hidden layers, and 100 trees, respectively. Error denotes the average distance (in centimetres) of final configurations from the target positions; build time gives the time (in seconds) to build the data structure; and query time gives the time (in seconds) to compute the predicted activation $\hat{u}$ for a target q_r.

Chapter 5

Trajectory Tracking with a Human Index Finger

This chapter deals with more sophisticated control mechanisms for the index finger model introduced in Chapter 4.

Section 5.1 begins with an implementation of an existing controller, developed by Malhotra et al. [46]. Sections 5.2 - 5.5 each present an additional component to improve the performance of this controller, and Section 5.6 ends with a discussion of the results.

We focus in this chapter on two fingertip precision tasks. The simpler of the two involves tracing a circle when the fingertip is constrained to lie in the q_1q_2 plane: this is similar to drawing on a flat horizontal surface, such as a tablet. The more difficult task involves tracing a circle in free space. In this case, the target is a circle in the q_1q_3 plane but, unlike the planar writing task, the position of the fingertip is unconstrained. The performance of each control method is given in its corresponding subsection, and tracking errors for all methods are summarized in Table 5.1 (for 2D tracking) and Table 5.2 (3D tracking).

5.1 Existing Implementation: Baseline Hierarchical Control

5.1.1 Method

The controller in this section – which we will call the *baseline controller* – was developed by Malhotra et al. [46]. It is a hand-tuned feedback controller with a hierarchical formulation. The high-level controller computes the desired change in plant velocity based on the instantaneous movement errors, and the low-level controller estimates activations to yield the velocity change sought by the high-level controller.

High-level controller Suppose that, at time t, our plant has configuration $\begin{pmatrix} q_t \\ \dot{q}_t \end{pmatrix}$ – comprised of fingertip position and velocity – and a target (or reference) configuration $\begin{pmatrix} q_r \\ \dot{q}_r \end{pmatrix}$. The desired change in plant velocity is given by the *feedback command*, defined as:

$$v_b = K_P(q_r - q_t) + K_D(\dot{q}_r - \dot{q}_t) \tag{5.1}$$

where K_P and K_D are manually chosen scalar gains. For this problem, we found that controller performance was sensitive to the choice of gains and that the selection process was non-trivial. We implemented a procedure to automatically select K_P and K_D off-line from a user-specified set of possibilities, based on which values gave the best performance on a trial task. The chosen values were $K_P = 1.9$, $K_D = 7.6\frac{\Delta t}{2}$, where Δt is the controller timestep. The $\frac{\Delta t}{2}$ term brings the position and velocity terms to a common scale, as the absolute value of the instantaneous velocity is $\frac{2}{\Delta t}$ larger than the absolute value of the change in position for a body accelerated for time Δt.

Low-level controller Given a velocity command v_b computed by the high-level controller, the low-level controller computes an activation u_t to produce the desired change in plant velocity.

To describe how this is done, we first define the notion of the Activation-Velocity Matrix (AVM), denoted by R. This is a $dim(q) \times dim(u)$ matrix that maps activations to corresponding changes in plant velocity: $R \cdot u_t \rightarrow \Delta \dot{q}_t$.

The controller computes u_t by the constrained least-squares equation:

$$u_t = \underset{u}{\operatorname{argmin}} ||R \cdot u - v_b||^2 \quad \text{such that } 0 \leq u \leq 1 \tag{5.2}$$

AVM computation We compute R using the *identification method* of [46]. Starting from the plant's initial configuration, we compute the change in velocity resulting from fully activating one muscle at a time. The change in velocity given by fully activating u_i (the i^{th} dimension of u) becomes the i^{th} column of R.

Algorithm 2 Identification method for AVM estimation

1: $f \leftarrow$ function such that: $f\left(\begin{pmatrix} q_t \\ \dot{q}_t \end{pmatrix}, u_t\right) \rightarrow \begin{pmatrix} q_{t+1} \\ \dot{q}_{t+1} \end{pmatrix}$
2: $q_{init} \leftarrow$ arbitrary initial configuration
3: $\begin{pmatrix} q_{(0)} \\ \dot{q}_{(0)} \end{pmatrix} \leftarrow f\left(\begin{pmatrix} q_{init} \\ \dot{q}_{init} \end{pmatrix}, \mathbf{0}\right)$ ▷ Apply 0 muscle activation
4: **for** $i = 1$ to $dim(u)$ **do**
5: $\quad u_{(i)} \leftarrow$ vector with value 1 at index i, 0 elsewhere
6: $\quad \begin{pmatrix} q_{(i)} \\ \dot{q}_{(i)} \end{pmatrix} \leftarrow f\left(\begin{pmatrix} q_{init} \\ \dot{q}_{init} \end{pmatrix}, u_{(i)}\right)$
7: $\quad R[i] \leftarrow \dot{q}_{(i)} - \dot{q}_{(0)}$ ▷ i^{th} column of R gets difference in velocities
8: **end for**

5.1.2 Results

2D tracing We begin with trajectory-tracking constrained to a plane. A screen-shot of this is shown in Figure 5.1a. A plot of the trajectory followed compared to the target trajectory is given in Figure 5.2a, with corresponding muscle activation levels shown in Figure 5.2b.

There is a striking translational error in the 2D tracking experiment. This error could be from one of two sources: it could result from the feedback command

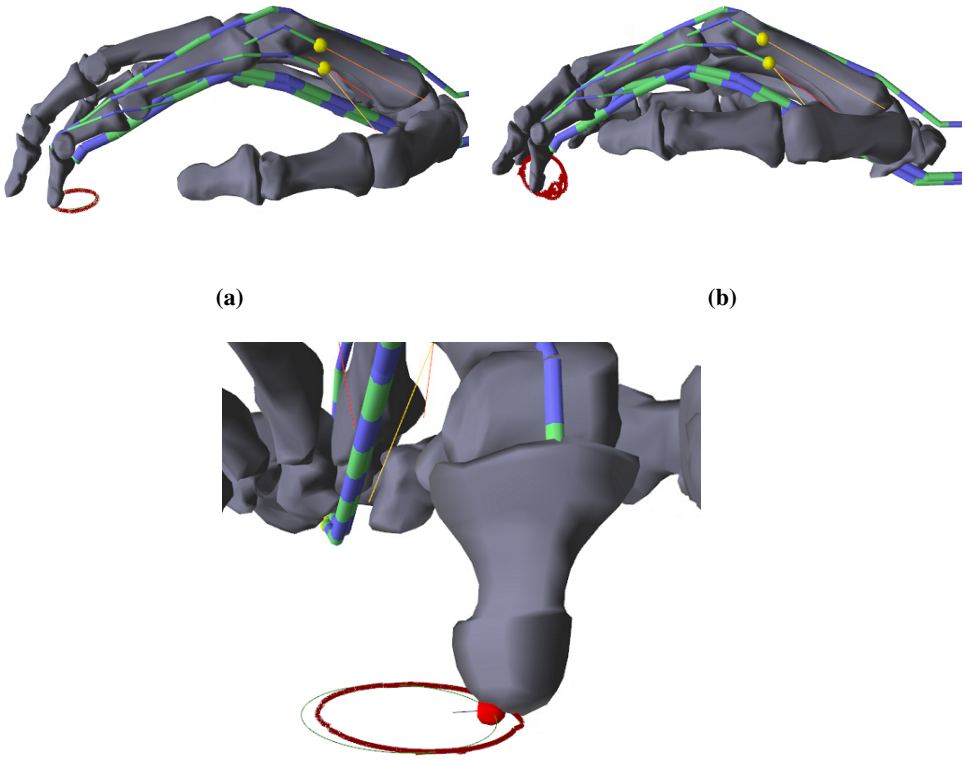


Figure 5.1: Screenshots for baseline controller. (a) tracing a circle in the plane. (b) tracing a circle in free space. (c) close-up of the planar tracing task: the red dot shows the target position and the blue line represents the high-level controller's desired change in position, given by $\Delta t(\dot{q}_t + v_b)$.

(see Equation 5.1), which would suggest a poor choice of gains parameters; or from the activation computation (Equation 5.2), which suggests a poor choice of AVM. To investigate this, see the close-up screenshot in Figure 5.1c: the red ball represents the target point and the blue line represents the high-level controller's desired change in position for the next timestep, computed as follows.

The controller's desired plant velocity over the next timestep is equal to the current velocity $\dot{q}_t$ plus the velocity command v_b. Note that this desired plant velocity

is determined by the current error in the plant configuration (see Equation 5.1), and as such is not generally equal to the reference velocity $\dot{q}_r$. We multiply the desired velocity $\dot{q}_t + v_b$ by Δt to get the position to which the high-level controller is attempting to guide the plant, and illustrate this attempted correction with the blue line.

From the screenshot, we see that the feedback command is trying to compensate for the positional error: the blue line pointing to the (plant's) right is present for the duration of the experiment. This tells us that the high-level controller is (appropriately) instructing the low-level controller to produce activations to bring the plant to the right, but the low-level controller is producing the wrong activations. This tells us that the AVM we've computed is not a good choice for this section of the configuration space; this is the motivation for Section 5.2, in which we will make the AVM configuration-dependent.

The average tracking error per trajectory point in this experiment was 0.13 centimetres, and the maximum error at any single point was 0.21 centimetres.

3D tracing Next, we considered tracking a circular trajectory in 3-dimensional space, without constraining the position of the fingertip. Unsurprisingly, the results are much less smooth than for the planar example: a screenshot is shown in Figure 5.1b, and plots of the followed trajectory and muscle activation are shown in Figures 5.2c and 5.2d, respectively. The controller gave an average per-point tracking error of 0.17 centimetres and a maximum error of 0.55 centimetres.

5.2 AVM Estimation

5.2.1 Method

In this section, we make the AVM configuration-dependent: in other words, instead of having a matrix R, we have a matrix $R(q_t)$. As before, we estimate the AVM at a given configuration using the identification method (Algorithm 2). But, unlike the previous section, in which we estimated R at one point and used this as the AVM over the entire configuration space, we now estimate $R(q_t)$ at multiple design points and interpolate the value at non-design points. We will call this controller

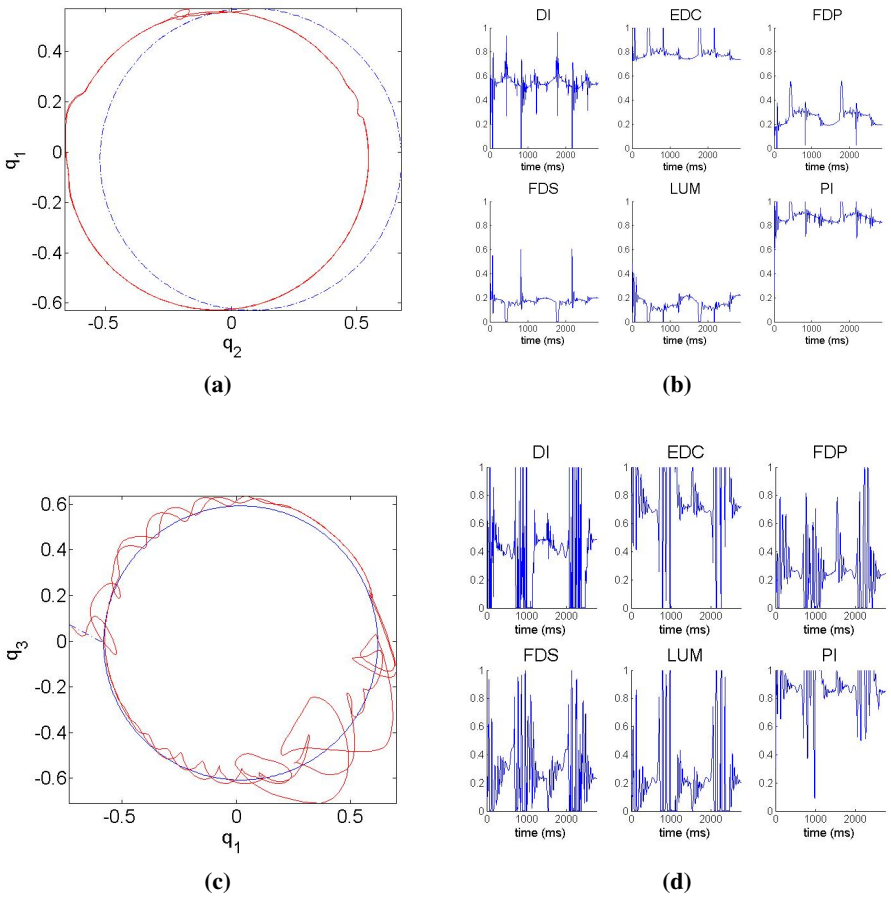


Figure 5.2: Depiction of circle-tracing results for baseline controller. (a) trajectory tracked for 2D experiments (as viewed from above): target trajectory is shown by the dashed blue line and the actual trajectory taken is given in solid red. (b) muscle activations for 2D trajectory tracking. (c-d) trajectory (as viewed from the left) and muscle activations for 3D (unconstrained) tracking.

the *variable AVM controller*.

Specifically, we estimate $R(q^{(i)})$ for 154 stationary plant configurations $\mathcal{Q} = \{q^{(1)}, q^{(2)}, ..., q^{(154)}\}$ (see Figure 5.3). To approximate the AVM at a new point q, we compute q's three nearest neighbours in $\mathcal{Q}$, $\{q_{nn}^{(1)}, q_{nn}^{(2)}, q_{nn}^{(3)}\}$, and compute a distance-weighted average of the values $R\left(q_{nn}^{(i)}\right), i = 1,2,3$ with the weights given

by the inverse of the distance between q and $q_{nn}^{(i)}$. We could use any number of neighbours k. In practice, there is a trade-off between smoothness of activation patterns and tracking accuracy. For smaller k, the controller parameters change more drastically with configuration changes, which can lead to better tracking accuracy but tends to cause jerkiness due to large parameter shifts. We found that three neighbours worked well. The controller can now be interpreted as a linearly blended composition of controllers with different matrices R, which are valid in the neighbourhood of corresponding configurations q [8].

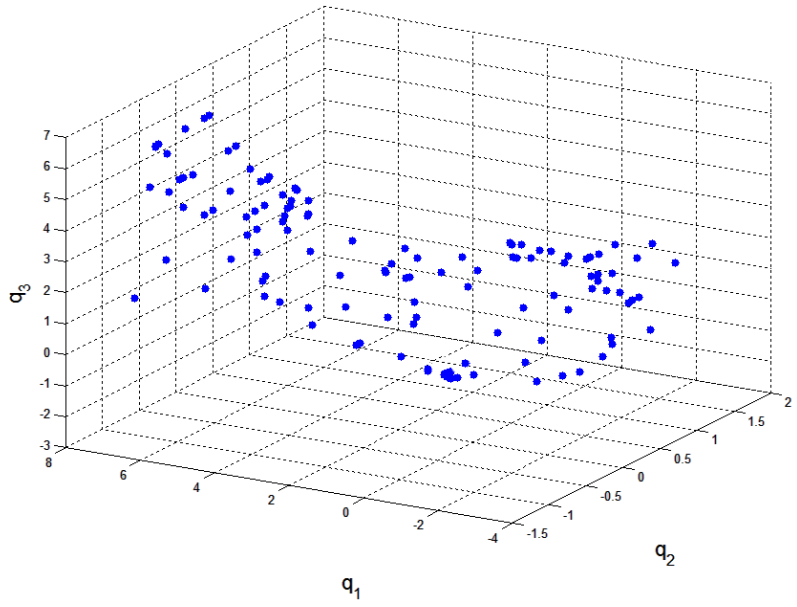


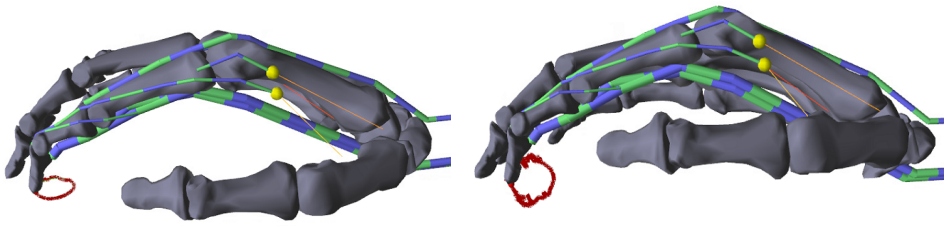
Figure 5.3: 3-dimensional scatter plot of plant configurations (i.e. fingertip positions) at which we compute the AVM. Axis units are in centimetres.

Compared to the control formulation of Section 5.1, the computation of the high-level velocity command is the same as before, but the low-level controller now solves for u_t by:

$$u_t = \underset{u}{\operatorname{argmin}} ||R(q_t) \cdot u - v_b||^2 \quad \text{such that } 0 \leq u \leq 1 \tag{5.3}$$

5.2.2 Results

2D tracing A screenshot for the 2D tracing task is given in Figure 5.4a, while Figures 5.5a and 5.5b plot the trajectory and muscle activation levels. Compared to the baseline controller, we have jerkier patterns of movement and muscle activations – which we address in the next section – but improved tracking accuracy, as we have overcome the persistent translational error given by the baseline controller. The variable AVM controller gives an average trajectory tracking error of 0.042 centimetres and a maximum of 0.14 centimetres.



(a) **(b)**

Figure 5.4: Screenshots for the variable AVM controller. (a) tracing a circle in the plane. (b) tracing a circle in free space.

3D tracing As we can see from the screenshot in Figure 5.4b and trajectory plot in Figure 5.5c, the variable AVM controller still encounters difficulty with tracing a circle in free space. It does, however, improve over the baseline controller: average tracking error was 0.085 centimetres and the maximum is 0.28 centimetres.

5.3 Smoothing

5.3.1 Method

Configuration-dependent AVMs improve the tracking accuracy of the baseline controller, but lead to more oscillatory activation patterns. In this section, we try to

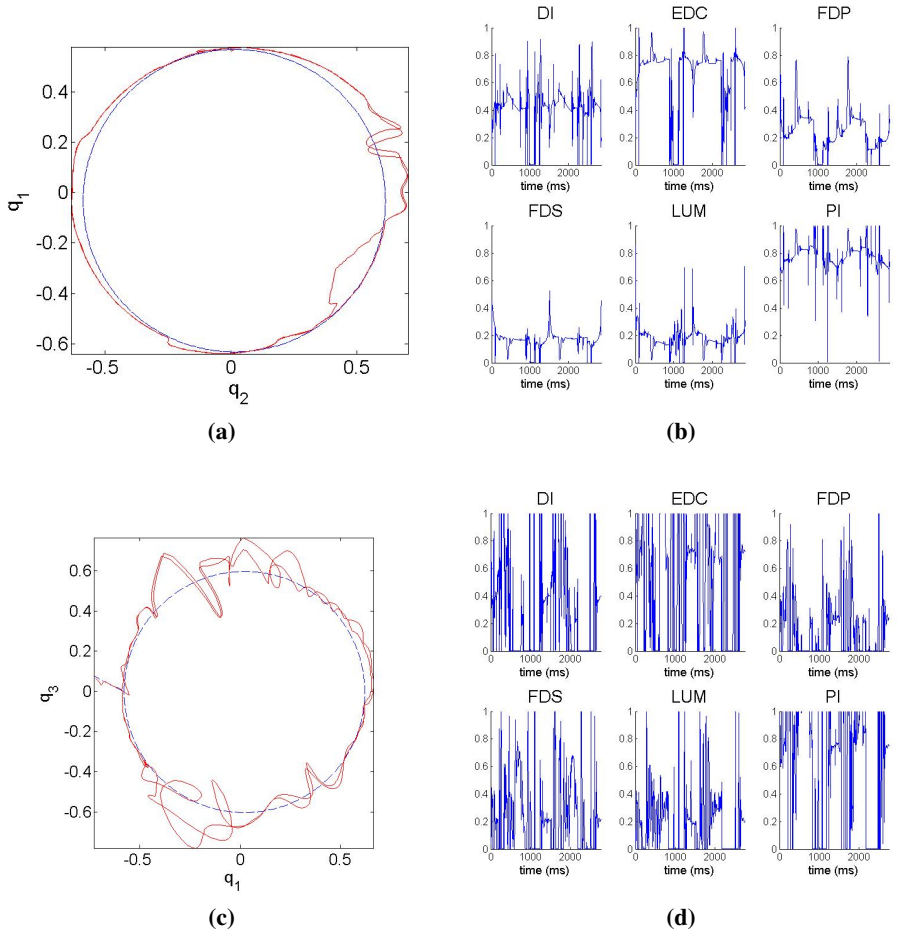


Figure 5.5: Depiction of circle-tracing results for variable AVM controller. (a) trajectory tracked for 2D experiments (as viewed from above): target trajectory is shown by the dashed blue line and the actual trajectory taken is given in solid red. (b) muscle activations for 2D trajectory tracking. (c-d) trajectory (as viewed from the left) and muscle activations for 3D (unconstrained) tracking.

remedy this by introducing activation regularization to the low-level controller.

To that end, we replace the constrained least-squares formulation of Equation 5.3 with the formulation used in [72]:

$$u_t = \underset{u}{\operatorname{argmin}}\, \alpha||R(q_t)\cdot u - v_b||^2 + \beta||u||^2 + \gamma||u - u_{t-1}||^2 \quad \text{such that } 0 \leq u \leq 1 \tag{5.4}$$

where α, β and γ are blending weights. The first term encourages the computed activations to follow the feedback commands. The second term penalizes large activations, and the third penalizes large changes in activations between timesteps. We scale the weights such that $\alpha + \beta + \gamma = 1$. With $\alpha = 1, \beta = \gamma = 0$, this formulation reduces to that of the unsmoothed variable AVM controller.

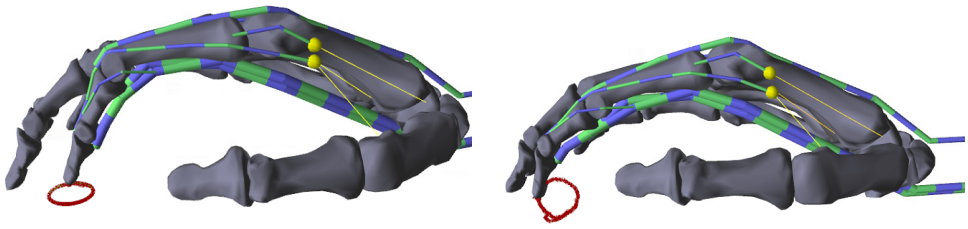
The smoothing parameters are chosen empirically, using the same automatic selection procedure we used to select the feedback gains in Section 5.1. For the remaining demonstrations in this chapter, we use the values $\alpha = 0.984, \beta = 0.014, \gamma = 0.002$.

5.3.2 Results

2D tracing Here we notice a marked improvement over the baseline and unsmoothed variable AVM controllers. Figures 5.6a and 5.7a show that, apart from a deviation in the bottom right quadrant, the controller is able to follow the circle very closely. The muscle activations produced by this controller are much lower than for the unsmoothed controller: Figure 5.7b shows that only the palmar interosseous (PI) muscle ever has an activation level above 0.4.

The average tracking error for this task was 0.011 centimetres, and the maximum was 0.16 centimetres.

3D tracing Figures 5.6b and 5.7c illustrate that, while far from perfect, the 3D circle tracing is noticeably improved over that of the previous controllers. Despite the smoothing, the muscle activation patterns (shown in 5.7d) are still fairly saturated, though not to the degree of the unsmoothed controller. The average tracking error was 0.062 centimetres, and the maximum error was 0.31 centimetres.



(a) **(b)**

Figure 5.6: Screenshots for the variable AVM controller with smoothing. (a) tracing a circle in the plane. (b) tracing a circle in free space.

5.4 Gain Scheduling

5.4.1 Method

In this section we implement gain scheduling for the variable AVM controller. Recall that gain scheduling automatically selects configuration-dependent values for the controller gains K_P and K_D in Equation 5.1. The general idea behind the method is similar to the one we used for AVM estimation: we move the plant to different configurations, and at each configuration we compute the optimal set of gains parameters (more on what we mean by "optimal" shortly). Then, during control operations, we compute the gains at each timestep by interpolating the data we obtained during training. Our approach – analytically computing optimal gains at design points and interpolating them throughout the configuration space – is similar to the methods used in [28, 64, 69], though as far as we know nobody has used our specific method for computing gains at the design points.

Defining optimal gains In defining optimal gains parameters, we assume that we are computing the feedback command v_b according to Equation 5.1 and the motor command u_t according to Equation 5.3 (in other words, we assume a variable AVM and no smoothing of the activations).

Intuitively, the purpose of the feedback term is to correct deviations of the plant from its desired configuration; thus, the ideal v_b is one that will result in an

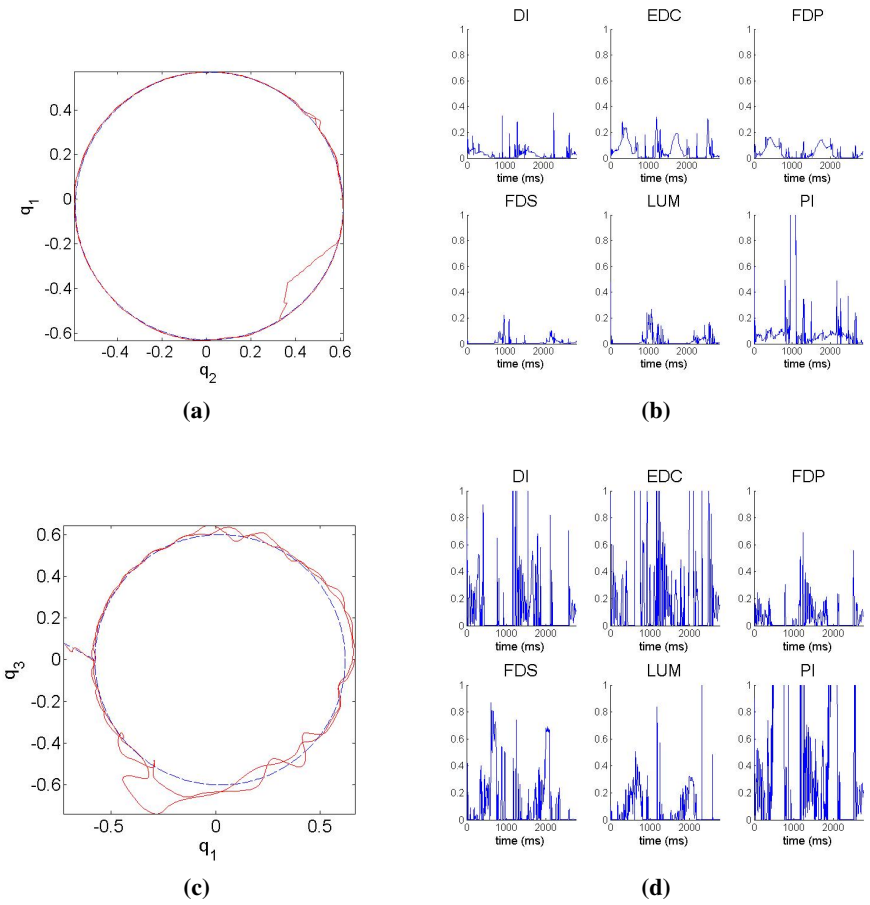


Figure 5.7: Depiction of circle-tracing results for the variable AVM controller with smoothing. (a) trajectory tracked for 2D experiments (as viewed from above): target trajectory is shown by the dashed blue line and the actual trajectory taken is given in solid red. (b) muscle activations for 2D trajectory tracking. (c-d) trajectory (as viewed from the left) and muscle activations for 3D (unconstrained) tracking.

activation u_t that will bring the plant back to its target configuration $\begin{pmatrix} q_r \\ \dot{q}_r \end{pmatrix}$.

So, for a given $\begin{pmatrix} q \\ \dot{q} \end{pmatrix}$, how do we find a K_P and K_D that will give us such a v_b? To approach this problem, we begin with a known, arbitrary activation vector $\tilde{u}$. If

we apply that activation to the system for one controller timestep, we obtain the mapping:

$$f\left(\begin{pmatrix} q \\ \dot{q} \end{pmatrix}, \tilde{u}\right) = \begin{pmatrix} \tilde{q} \\ \dot{\tilde{q}} \end{pmatrix} \tag{5.5}$$

where f is the function that governs the dynamical system and $\tilde{q}$ is the configuration that results from applying $\tilde{u}$ to the system in configuration q for one timestep. Since we now know that $\tilde{u}$ will bring the system from $\begin{pmatrix} q \\ \dot{q} \end{pmatrix}$ to $\begin{pmatrix} \tilde{q} \\ \dot{\tilde{q}} \end{pmatrix}$, we know that an ideal feedback controller, given a current state $\begin{pmatrix} q \\ \dot{q} \end{pmatrix}$ and a target state $\begin{pmatrix} \tilde{q} \\ \dot{\tilde{q}} \end{pmatrix}$, would compute $\tilde{u}$ as an activation. Thus, in this particular case, our ideal feedback function $\hat{v}_b$ (parametrized by the ideal gains $\hat{K}_P$ and $\hat{K}_D$) will return the value:

$$\begin{aligned} \hat{v}_b &:= \hat{K}_P(q - \tilde{q}) + \hat{K}_D(\dot{q} - \dot{\tilde{q}}) \\ &= R(q_t) \cdot \tilde{u} \end{aligned} \tag{5.6}$$

Note that we have not accounted for the possibility that there may be more than one activation u that will lead the plant to the target configuration – in particular, by co-activating agonist and antagonist muscles, there could be many so-called "optimal" activations u. However, we would argue that, in such instances, a truly ideal controller would give the solution that activates as few muscles as possible (so as to minimize biomechanical cost). Hence, when we estimate the optimal gains at configurations throughout the state space (the procedure for which we will describe shortly) we will only use values of $\tilde{u}$ that activate a single muscle.

Estimating optimal gains for a given configuration We estimate the optimal gains at a given configuration by beginning at the configuration and fully activating one muscle at a time for one timestep (this is similar to our procedure for estimating the AVM). We obtain the next states corresponding to each of these activations and, treating these new states as the target configurations in the feedback parametrization, solve a non-negative least-squares problem for $\hat{K}_P$ and $\hat{K}_D$. Pseudocode is given in Algorithm 3.

Algorithm 3 Estimation of PD gains

Given: current configuration $\begin{pmatrix} q \\ \dot{q} \end{pmatrix}$
$N \leftarrow dim(u)$
for $i = 1$ to N **do**
    $u_{(i)} \leftarrow$ vector with value 1 at index i, 0 elsewhere
    $\hat{v}_b^{(i)} \leftarrow R(q) \cdot u_{(i)}$ ▷ Ideal feedback command for $q, u_{(i)}$
    $\begin{pmatrix} \tilde{q}^{(i)} \\ \dot{\tilde{q}}^{(i)} \end{pmatrix} \leftarrow f\left(\begin{pmatrix} q \\ \dot{q} \end{pmatrix}, u_{(i)} \right)$ ▷ Result of applying $u_{(i)}$ for one timestep
    $\Delta q^{(i)} \leftarrow \tilde{q}^{(i)} - q$ ▷ Store resulting position change
    $\Delta \dot{q}^{(i)} \leftarrow \dot{\tilde{q}}^{(i)} - \dot{q}$ ▷ Store resulting velocity change
end for
$\hat{V} \leftarrow \left[\hat{v}_b^{(1)^T}, \hat{v}_b^{(2)^T}, ..., \hat{v}_b^{(N)^T} \right]^T$ ▷ Column vector of concatenated $\hat{v}_b^{(i)}$
$\Delta Q \leftarrow \left[\Delta q^{(1)^T}, \Delta q^{(2)^T}, ..., \Delta q^{(N)^T} \right]^T$ ▷ Column vector of concatenated $\Delta q^{(i)}$
$\Delta \dot{Q} \leftarrow \left[\Delta \dot{q}^{(1)^T}, \Delta \dot{q}^{(2)^T}, ..., \Delta \dot{q}^{(N)^T} \right]^T$ ▷ Column vector of concatenated $\Delta \dot{q}^{(i)}$
$\hat{K}_P, \hat{K}_D \leftarrow$ solution of non-negative least-squares problem: $\hat{V} = [\Delta Q \ \Delta \dot{Q}] \begin{bmatrix} K_P \\ K_D \end{bmatrix}$

Gathering and interpolating gains data Like with AVM estimation, we compute $\hat{K}_P, \hat{K}_D$ at various configurations q and, to estimate these values at a new configuration, we use an inverse distance-weighted average of the k nearest neighbours. However, the crucial difference is that, for AVM estimation, we considered only the position of the plant and ignored the velocity; plant velocity is crucial to PD-control, so ignoring it for gains estimation would be unwise. Hence, we interpolate with respect to velocity as well as position.

This requires a slightly different scheme for gathering training data than we used for the AVM problem. For the AVM, we specified different activation vectors, and the configurations at which the AVM was sampled were those obtained by applying these activations until the plant stabilized. For the gains, we first sampled 500 activations in a Latin hypercube and 100 activations activating either the EDC alone, or the EDC and lumbrical. The reason for the 100 EDC and lumbrical activations was to ensure that we had adequate sampling of points with a high vertical coordinate, as we found that randomly chosen activations tended to oversample the lower points in the configuration space. We reach the configurations by applying

the specified activations for 50, 100, 200, 300, 400, 600, 800, 1000, 1300, 1600, and 2000 milliseconds. If the plant stabilized in less than 2 seconds (for most activations it did so in less than 1), we stopped the iterations early and moved on to the next activation. This resulted in a gains lookup table of 3,272 distinct locations in the combined position-velocity space (scatter plot of the positions is shown in Figure 5.8).

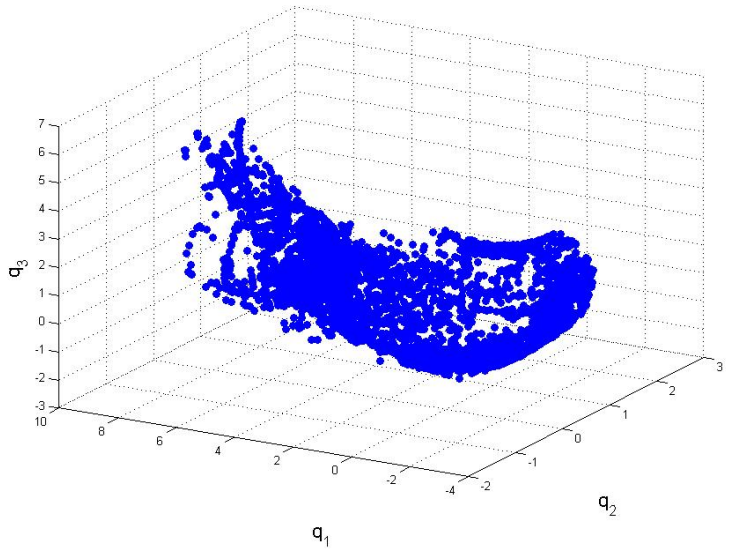


Figure 5.8: 3-dimensional scatter plot of plant positions at which we compute the gains K_P and K_D. Axis units are in centimetres.

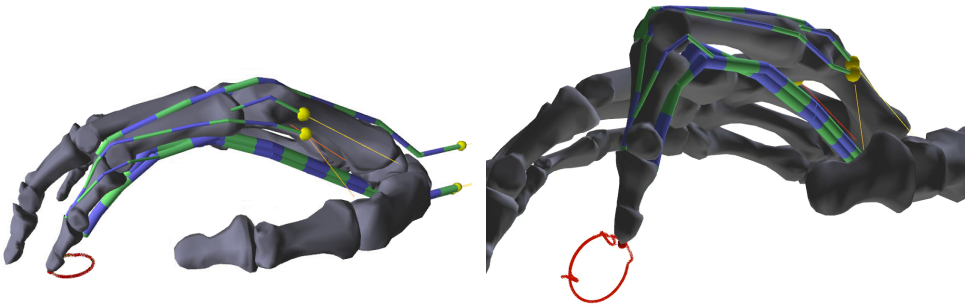
Like AVM interpolation, in selecting k there is a trade-off between tracking accuracy and smoothness of movement, as the controller parameters change more rapidly for smaller values of k. We found that $k = 30$ worked well, and this is the value used for the remaining experiments in this chapter.

5.4.2 Results

2D tracing A screenshot of the 2D tracing is shown in Figure 5.9a, and plots of the tracked trajectory and muscle activations are given in Figures 5.10a and 5.10b, respectively. In terms of tracking accuracy, performance is slightly improved over

the variable AVM controller with smoothing: there is noticeably decreased deviation in the bottom right corner of the circle. However, there is some increase in rapid oscillations in the muscle activations, and this controller activates the muscles more strongly for this task.

The average tracking error for this task was 0.028 centimetres and the maximum was 0.13 centimetres.



(a) **(b)**

Figure 5.9: Screenshots for the gain scheduled controller. (a) tracing a circle in the plane. (b) tracing a circle in free space.

3D tracing From Figures 5.9b and 5.10c, we see that there is still a problem with the plant dragging away from the target trajectory in the bottom left quadrant of the circle, though the error is less for the gain scheduled controller than for the smoothed variable AVM controller. The average tracking error is 0.038 centimetres, and the maximum is 0.22 centimetres.

Interestingly, if we look at the activations shown in Figure 5.10d and compare them to the activations produced by the smoothed variable AVM controller for the same task, we notice more of a distinct pattern for this controller – the activations for all muscles display a noisy but distinct curve, while the non-gain-scheduled controller tends to oscillate frequently between very high and very low activations (particularly for the EDC and PI). This is in contrast to the 2D tracing task, on which the non-gain-scheduled controller produced smaller, smoother activations. The fact that gain scheduling yields greater improvements for 3D tracing than 2D

should perhaps not come as a surprise: since unconstrained tracing is bound to explore a larger portion of the configuration space, we expect the importance of variable gains to be greater.

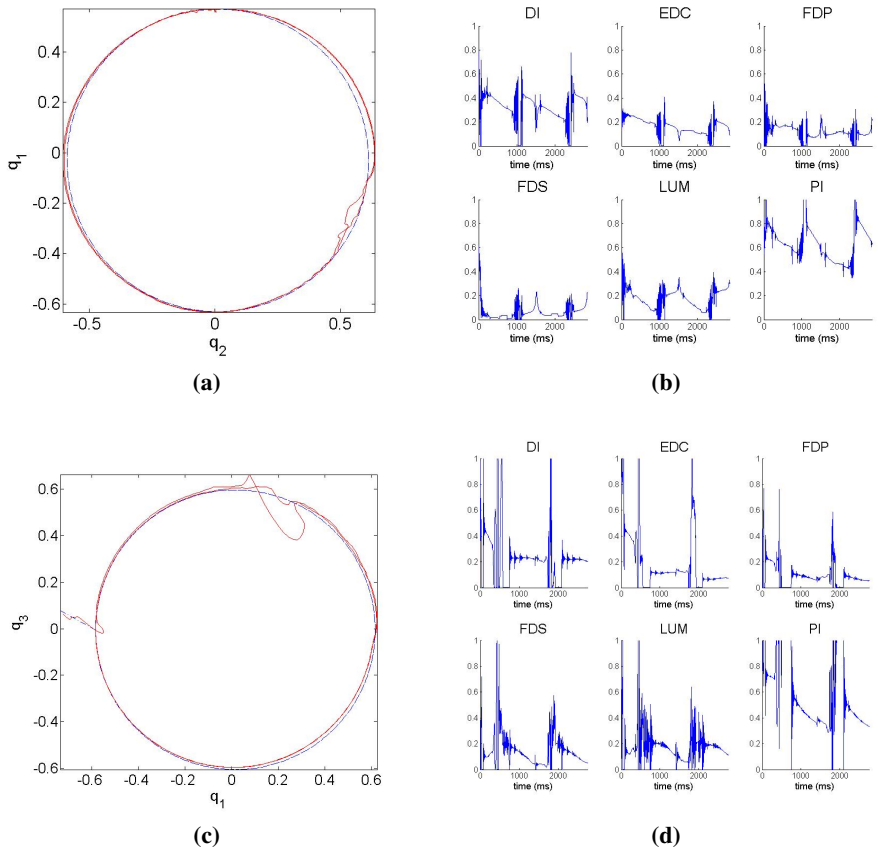
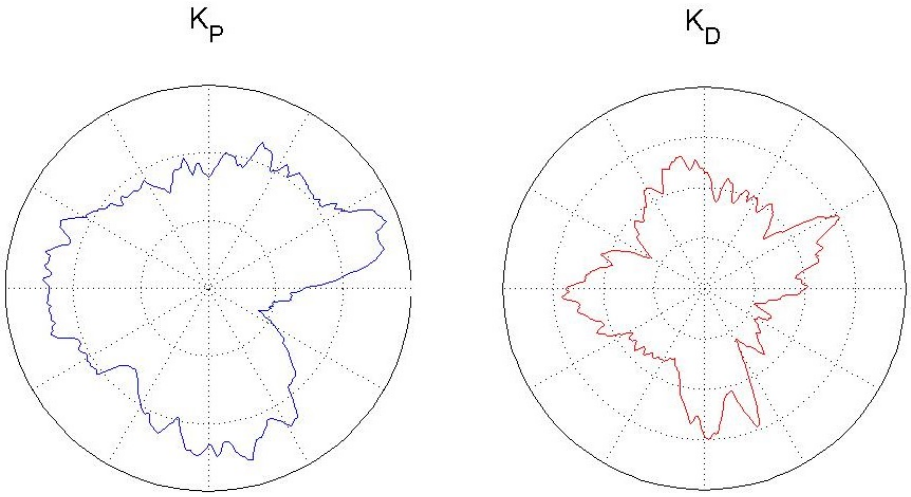


Figure 5.10: Depiction of circle-tracing results for the gain scheduled controller. (a) trajectory tracked for 2D experiments (as viewed from above): target trajectory is shown by the dashed blue line and the actual trajectory taken is given in solid red. (b) muscle activations for 2D trajectory tracking. (c-d) trajectory (as viewed from the left) and muscle activations for 3D (unconstrained) tracking.

Gains values The values of the computed gains along the 3D circle trajectory are shown in Figure 5.11. The computed values for K_P ranged between 1.651 and

2.19 while the values of K_D (before being multiplied by $\frac{\Delta t}{2}$) ranged from 11.324 to 14.062. In the plots, both K_P and K_D are mean-shifted and divided by their respective ranges so that the variation in the values can be seen clearly.



(a) **(b)**

Figure 5.11: Polar plots of the values of the controller gains K_P (a) and K_D (b) along the 3D circle trajectory. Gains values are mean-shifted and scaled by their ranges so that the variations can be seen clearly on a polar plot.

5.5 Passive Dynamics Compensation

5.5.1 Method

In the previous subsections, we used only feedback control to compute the velocity command, which we denoted by v_b. In this section, we update the velocity command to $v_t := v_p + v_b$, where $v_p = v_p(q_t, q_r)$ is a 3×1 *passive dynamics* term designed to compensate for passive forces acting on the plant and capture some non-linearities in the dynamics.

The low-level controller computes activations in the same way as before, but

now aims to follow v_t at each timestep instead of v_b:

$$u_t = \underset{u}{\operatorname{argmin}}\, \alpha||R(q_t)\cdot u - v_t||^2 + \beta||u||^2 + \gamma||u_t - u_{t-1}||^2 \quad \text{such that } 0 \leq u \leq 1 \tag{5.7}$$

We compensate for passive dynamics using equilibrium controls, the computation of which was discussed at length in Chapter 4. The rationale behind this is that equilibrium controls will hold the plant's position against gravity and other passive forces; as such, they cancel out the passive forces acting on the plant at a given configuration. The use of equilibrium controls for this purpose is seen in [18, 20, 43].

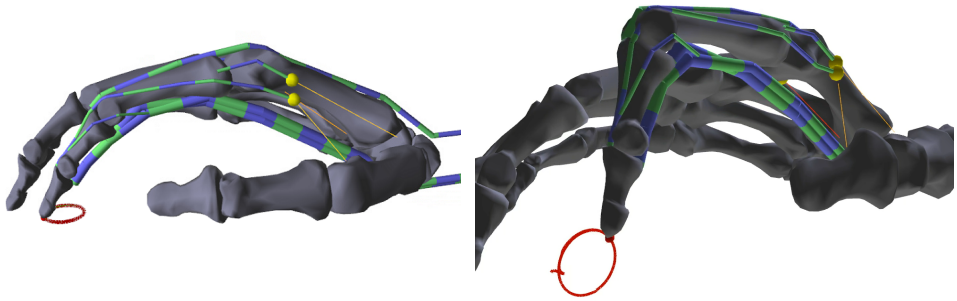
Specifically, for a target position q_r we estimate the activation u_{eq} that will lead the plant there by nearest-neighbour prediction on the 39,000 activation/equilibrium position pairs discussed in Chapter 4. In that chapter, we found nearest-neighbours to be the most effective method for predicting an activation that would lead to a target configuration; hence, that is the method we use here. For a detailed description of our methodology, see Section 4.3.

Once we have computed u_{eq} corresponding to the target q_r, the passive dynamics term is set to $v_p = R(q_t)\cdot u_{eq}$, where q_t is the current configuration of the plant. Note that, in computing v_p, we consider only the positional component of the target and ignore the target velocity.

5.5.2 Results

2D tracing As we can see from Figures 5.12a and 5.13a, the controller traces a nearly perfect circle. The average tracking error is 0.022 centimetres and the maximum is 0.032. Interestingly, adding the passive dynamics term has also had the effect of producing much smaller muscle activations for this task as compared to gain-scheduled controller without the passive dynamics term.

3D tracing Figures 5.12b and 5.13c show that, after adding five new features to our original controller, we are finally able to track a circle in free space without any large deviations. The activation patterns (Figure 5.13d) have a similar pattern



(a) (b)

Figure 5.12: Screenshots for the gain scheduled controller with passive dynamics compensation. (a) tracing a circle in the plane. (b) tracing a circle in free space.

to those produced by the original gain-scheduled controller, though with somewhat fewer oscillations. Activation of the FDS is also significantly reduced.

The average tracking error for this task was 0.030 centimetres, and the maximum instantaneous tracking error was 0.16 centimetres.

5.6 Summary

In this section, we compare the performance of the different control methods. Tables 5.1 and 5.2 summarize controller tracking errors on the 2D and 3D circle tracing tasks, respectively.

For both tasks, going from the baseline to the variable AVM controller cuts tracking errors by about half. In going from the unsmoothed to smoothed variable AVM controller, there is a sizeable decrease in the average error (about 30 to 50%), but a slight increase in the maximum error. However, an advantage of the smoothed controller is more natural-looking movement: there is reduced oscillation in the activation patterns, which results in a less jittery trajectory.

In going from constant to scheduled gains, there is little difference for the 2D task: the maximum error is decreased by reducing the kink to the right of the circle, but there is an increase in average error effected by added drift to the left of the circle. The more significant difference (30-40% error reduction) is seen for the

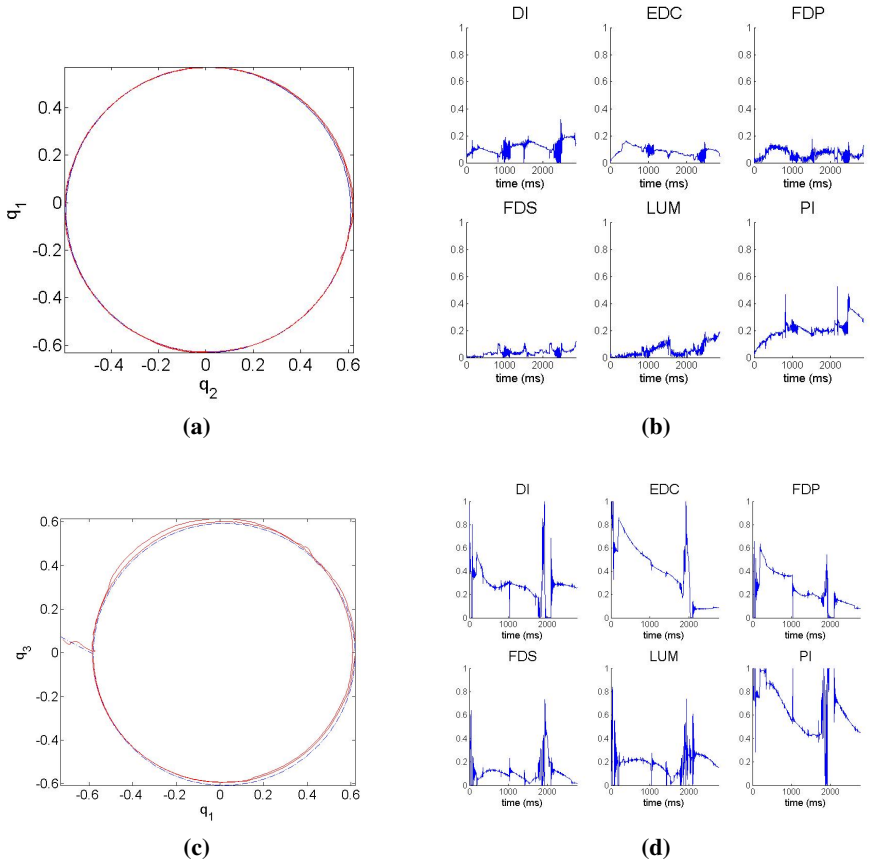


Figure 5.13: Depiction of circle-tracing results for the gain scheduled controller with passive dynamics compensation. (a) trajectory tracked for 2D experiments (as viewed from above): target trajectory is shown by the dashed blue line and the actual trajectory taken is given in solid red. (b) muscle activations for 2D trajectory tracking. (c-d) trajectory (as viewed from the left) and muscle activations for 3D (unconstrained) tracking.

3D task. As mentioned earlier, this makes sense as the 3D task explores a larger portion of the configuration space due to being unconstrained in its movement, so configuration-dependent gains are more likely to be useful. Another consideration is that our gain-scheduling algorithm is performed without constraining the plant to a plane, so the higher average error for the 2D task could be due to the fact that

	2D tracking errors	
Controller	**Average error (cm)**	**Max error (cm)**
Baseline	0.133	0.208
Variable AVM	0.0420	0.142
Variable AVM + Smoothing	0.0221	0.160
Gain-scheduled	0.0282	0.131
Gain-scheduled + Passive Dynamics	0.0218	0.0321

Table 5.1: Summary of controllers' performance on the 2D circle tracing task. Average error denotes the average positional error between the target and actual plant position at each point on the trajectory, in centimetres. Max error denotes the maximum positional error at any single point on the trajectory, in centimetres.

	3D tracking errors	
Controller	**Average error (cm)**	**Max error (cm)**
Baseline	0.168	0.549
Variable AVM	0.0849	0.280
Variable AVM + Smoothing	0.0621	0.306
Gain-scheduled	0.0378	0.215
Gain-scheduled + Passive Dynamics	0.0296	0.160

Table 5.2: Summary of controllers' performance on the 3D circle tracing task. Average error denotes the average positional error between the target and actual plant position at each point on the trajectory, in centimetres. Max error denotes the maximum positional error at any single point on the trajectory, in centimetres.

the ideal gains parameters are different when the fingertip is constrained to a plane.

When we adapt our gain-scheduled controller to include passive dynamics, there is a considerable reduction in errors for both the 2D and 3D tasks. Because the passive dynamics term gives the activation required to hold our target pose against passive forces, it removes some non-linearities in the dynamics. Our feedback control formulation assumes that system dynamics are locally linear: hence, passive dynamics compensation simplifies the problem in such a way that our linear control methods are more suitable.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Summary This thesis aimed to develop novel learning algorithms for motor control. Chapter 2 reviewed the necessary background concepts and related work in control literature. Section 2.1 reviewed the properties of biological motor systems, including how learning occurs in the brain and how biomechanical systems are represented as dynamical systems (either simulated or robotic). Section 2.2 covered some basic methods for controlling these systems, and Section 2.3 went over concepts in machine learning with discussions of where these concepts are used in control literature.

In Chapter 3 we developed a learning method for control of a quasistatic model of the elastodynamic skin of the human mid-face. Section 3.1 gave properties of the model, and Section 3.2 introduced regression methods for its control. In Section 3.3, we developed a stochastic gradient descent formulation to improve the performance of the supervised learning methods, and showed that the use of regression as a pre-processing step significantly improves the accuracy and convergence time of the optimization.

Chapter 4 investigated reaching control for the human index finger. Section 4.1 described the index finger model. Section 4.2 framed the task as a supervised learning problem, which was then addressed using k-nearest-neighbours (Section 4.3),

neural networks (Section 4.4) and random forests (Section 4.5). Section 4.6 compared the performance of the methods studied and concluded that 1-nearest-neighbour was the best prediction method for this problem.

Finally, in Chapter 5 we incrementally developed a control algorithm for trajectory tracking with the index finger model. Section 5.1 reviewed the controller of [46], which was additionally improved upon in the succeeding subsections by introducing configuration-dependent AVMs (Section 5.2), activation smoothing (Section 5.3), gain scheduling (Section 5.4), and passive dynamics compensation (Section 5.5).

Contributions Our first contribution is a novel combination of regression and local search for control of quasistatic systems. We improve on the policy gradient method of Kohl and Stone [38] by implementing greedy policy updates and incorporating regression as a pre-processing step. With the combined regression/policy gradient method, we have significantly better control performance than with either method alone, and we are able to improve the convergence time of the gradient descent optimization by nearly 50%.

Next, Chapter 5 develops several techniques for trajectory-tracking problems, using the controller formulation of Malhotra et al. [46] as a starting point. By framing the model as a Linear Parameter-Varying (LPV) system with respect to the AVM, we obtain a controller with significantly improved tracking abilities. While the LPV assumption is fairly common in control literature, our particular method for estimating the AVM has (as far as we know) not been implemented elsewhere. By adapting Malhotra et al.'s [46] low-level controller to include the activation-smoothing method of Sueda et al. [72], we obtain less oscillatory muscle activations, which results in fewer sudden, jerky movements in trajectory tracking.

We then proposed a novel gain-scheduling method for PD control that is both simple to implement and applicable to a broad class of feedback problems. To date, the vast majority of control methods for tendon-based systems rely on hand-tuned gains, so our method could be of use. Lastly, we introduce the use of equilibrium controls for passive dynamics compensation. This is not a new idea – see, for example, [20, 66] – but by using nearest-neighbour prediction we are able to avoid issues with redundant and non-convex mappings that cause problems for other pre-

diction methods, and the use of locality-sensitive hashing algorithms makes our method extremely efficient.

6.2 Future Work

Online learning Our current controller implementation performs all learning offline: we determine in advance where to gather the training data, we do it, and then we learn the model parameters accordingly. This is different from how humans learn: we learn as we perform movements, and our performance at a task improves the more time we spend doing it.

If we want our controller to be adaptable to changes in its environment, we need an online learning method so that parameters can be changed as new situations are encountered. Online learning/data collection could also improve the efficacy of training by allowing the controller to identify portions of the plant's configuration space that are worth exploring further – for example, areas of the space we expect to occupy often, or regions of rapid change in plant configuration with respect to muscle activations.

Noise and sensory delay Our feedback controller assumes noiseless, instantaneous feedback on the state of the system. This is not at all what is seen in biological systems, in which sensory streams are affected by noise and temporal delay. We also assume that motor commands work deterministically: we can compute a certain activation and, when we apply that command to the plant in a certain configuration, we will always see the same result. Again, this is not what we see in real biological systems: in these cases, motor signals are generated imprecisely and corrupted as they travel to the muscle. For example, if you hit a hole-in-one at a particular hole on a golf course, that doesn't mean that you have now "figured out" how to do it and can replicate your earlier action perfectly every time: variance in movements is unavoidable even for highly practiced individuals.

It remains to be seen how (or if) our controller could be used to control systems with some kind of noise or delay involved.

Scaling We need to consider how our method would scale to a larger dynamical system – for example, learning to control the entire hand or performing full-body tasks like locomotion. Due to our use of k-nearest-neighbours prediction with locality-sensitive hashing, the computation of controller parameters is efficient and robust (i.e. not prone to numerical instability), so the determination of these values given a set of training data should scale reasonably well.

A potential problem arises in solving for low-level commands: with a large number of muscles, the solution of the QP in Equation 5.7 could become difficult to solve – specifically, if the matrix $R(q_t)$ is indefinite, the problem becomes NP-hard [62].

However, the real bottleneck of our learning process is data collection. The sizes of our problem's input and configuration spaces grow exponentially with the system's degrees of freedom. This could be partially addressed with more efficient experimental design and online learning techniques, but ensuring sufficient coverage of the entire problem space poses a challenging computational problem.

Object manipulation Interaction with the environment is a crucial function of motor movements, particularly of the hands. For our method to be useful for real-world tasks, we need to ensure that it is capable handling contact with external objects. This will likely require some modification of our existing framework, particularly in how we parametrize our control goals. Our method for dynamical system control works with fully specified, spline-based trajectories: this framework makes it difficult to incorporate elements such as force modulation and reaction to environmental cues, both of which are critical to object interaction.

Imitating human movements All experiments performed for this thesis were formulated using simulated data: for the mid-face, we assessed prediction quality by attempting to reach certain simulated poses, and for the index finger we attempted to trace geometric trajectories. We have not addressed the problem of imitating or learning from the movements of human subjects. We are currently working on testing the controller of Chapter 3 on real facial expressions obtained from video capture data.

Bibliography

[1] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, Jan. 2008. ISSN 0001-0782. doi:10.1145/1327452.1327494. URL http://doi.acm.org/10.1145/1327452.1327494. → pages 18, 50

[2] S. Andrews and P. Kry. Goal directed multi-finger manipulation: Control policies and analysis. *Computers and Graphics*, 37(7):830 – 839, 2013. ISSN 0097-8493. doi:http://dx.doi.org/10.1016/j.cag.2013.04.007. → pages 11, 14, 18

[3] U. Ascher and C. Greif. *A First Course on Numerical Methods*. Computational Science and Engineering. Society for Industrial and Applied Mathematics, 2011. ISBN 9780898719970. → pages 31

[4] P. Bédard and J. Sanes. Gaze and hand position effects on finger-movement-related human brain activation. *J. Neurophysiol.*, 101(2): 834–842, Feb 2009. → pages 47

[5] R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1957. → pages 23

[6] Y. Bengio. Learning deep architectures for AI. *Found. Trends Mach. Learn.*, 2(1):1–127, Jan. 2009. ISSN 1935-8237. doi:10.1561/2200000006. URL http://dx.doi.org/10.1561/2200000006. → pages 19

[7] D. J. Berger, R. Gentner, T. Edmunds, D. K. Pai, and A. d'Avella. Differences in adaptation rates after virtual surgeries provide direct evidence for modularity. *The Journal of Neuroscience*, 33(30):12384–12394, 2013. doi:10.1523/JNEUROSCI.0122-13.2013. URL http://www.jneurosci.org/content/33/30/12384.abstract. → pages 5

[8] R. R. Burridge, A. A. Rizzi, and D. E. Koditschek. Sequential Composition of Dynamically Dexterous Robot Behaviors. *Int. J. Robot. Res.*, 18(6): 534–555, 1999. → pages 63

[9] Y. Cao, M. A. Brubaker, D. J. Fleet, and A. Hertzmann. Efficient optimization for sparse Gaussian process regression. *CoRR*, abs/1310.6007, 2013. URL http://arxiv.org/abs/1310.6007. → pages 17

[10] R. Caruana. Multitask learning. *Machine Learning*, 28(1):41–75, 1997. ISSN 0885-6125. doi:10.1023/A:1007379606734. → pages 33

[11] P. Cisek and J. Kalaska. Neural mechanisms for interacting with a world full of action choices. *Annual Review of Neuroscience*, 33(1):269–298, 2010. doi:10.1146/annurev.neuro.051508.135409. URL http://dx.doi.org/10.1146/annurev.neuro.051508.135409. → pages 5

[12] C. Currin, T. Mitchell, M. Morris, and D. Ylvisaker. A Bayesian approach to the design and analysis of computer experiments. Technical Report ORNL–6498, Oak Ridge Laboratory, 1988. → pages 16

[13] P. de Aguiar, B. Bourguignon, M. Khots, D. Massart, and R. Phan-Than-Luu. D-optimal designs. *Chemometrics and Intelligent Laboratory Systems*, 30(2):199 – 210, 1995. ISSN 0169-7439. doi:http://dx.doi.org/10.1016/0169-7439(94)00076-X. URL http://www.sciencedirect.com/science/article/pii/016974399400076X. → pages 20

[14] M. de Lasa, I. Mordatch, and A. Hertzmann. Feature-Based Locomotion Controllers. *ACM Transactions on Graphics*, 29(3), 2010. → pages 11, 14

[15] E. G. de Ravé, F. Jiménez-Hornero, A. Ariza-Villaverde, and J. Gómez-López. Using general-purpose computing on graphics processing units (GPGPU) to accelerate the ordinary kriging algorithm. *Computers and Geosciences*, 64(0):1 – 6, 2014. ISSN 0098-3004. doi:http://dx.doi.org/10.1016/j.cageo.2013.11.004. URL http://www.sciencedirect.com/science/article/pii/S0098300413002914. → pages 17

[16] M. Deisenroth and C. Rasmussen. Efficient reinforcement learning for motor control. In *Proceedings of the 10th International Workshop on Systems and Control*, 2009. → pages 13

[17] A. Deshpande, Z. Xu, M. Weghe, B. Brown, J. Ko, L. Chang, D. Wilkinson, S. Bidic, and Y. Matsuoka. Mechanisms of the Anatomically Correct Testbed hand. *Mechatronics, IEEE/ASME Transactions on*, 18(1):238–250, Feb 2013. ISSN 1083-4435. doi:10.1109/TMECH.2011.2166801. → pages 2, 6

[18] A. D. Deshpande, J. Ko, D. Fox, and Y. Matsuoka. Control strategies for the index finger of a tendon-driven hand. *Int. J. Rob. Res.*, 32(1):115–128, Jan. 2013. ISSN 0278-3649. doi:10.1177/0278364912466925. URL http://dx.doi.org/10.1177/0278364912466925. → pages 2, 5, 9, 11, 12, 13, 17, 46, 75

[19] J. Doyle, B. Francis, and A. Tannenbaum. Feedback control theory, 1990. → pages 10, 11

[20] M. Fain. Control of complex biomechanical systems. Master's thesis, University of British Columbia, 2013. → pages 5, 46, 75, 80

[21] M. Fernández-Delgado, E. Cernadas, S. Barro, and D. Amorim. Do we need hundreds of classifiers to solve real world classification problems? *Journal of Machine Learning Research*, 15:3133–3181, 2014. URL http://jmlr.org/papers/v15/delgado14a.html. → pages 52

[22] T. Geijtenbeek, M. van de Panne, and A. F. van der Stappen. Flexible muscle-based locomotion for bipedal creatures. *ACM Transactions on Graphics*, 32(6), 2013. → pages 1, 5, 6, 11, 13, 14

[23] A. Gosavi. Reinforcement learning: A tutorial survey and recent advances. *INFORMS Journal on Computing*, 21(2):178–192, 2009. doi:10.1287/ijoc.1080.0305. → pages 25

[24] K. Grochow, S. L. Martin, A. Hertzmann, and Z. Popović. Style-based inverse kinematics. *ACM Trans. Graph.*, 23(3):522–531, Aug. 2004. ISSN 0730-0301. doi:10.1145/1015706.1015755. URL http://doi.acm.org/10.1145/1015706.1015755. → pages 15, 17

[25] R. Grzeszczuk, D. Terzopoulos, and G. Hinton. Neuroanimator: Fast neural network emulation and control of physics-based models. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, pages 9–20, New York, NY, USA, 1998. ACM. ISBN 0-89791-999-8. doi:10.1145/280814.280816. URL http://doi.acm.org/10.1145/280814.280816. → pages 19

[26] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *J. Mach. Learn. Res.*, 3:1157–1182, Mar. 2003. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=944919.944968. → pages 6

[27] M. Hagan and M. Menhaj. Training feedforward networks with the Marquardt algorithm. *Neural Networks, IEEE Transactions on*, 5(6): 989–993, Nov 1994. ISSN 1045-9227. doi:10.1109/72.329697. → pages 50

[28] H. Hannoun, M. Hilairet, and C. Marchand. High performance current control of a switched reluctance machine based on a gain-scheduling {PI} controller. *Control Engineering Practice*, 19(11):1377 – 1386, 2011. ISSN 0967-0661. doi:http://dx.doi.org/10.1016/j.conengprac.2011.07.011. URL http://www.sciencedirect.com/science/article/pii/S0967066111001626. → pages 12, 67

[29] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition*. Springer Series in Statistics. Springer, 2009. ISBN 9780387848587. URL https://books.google.ca/books?id=tVIjmNS3Ob8C. → pages 52, 53

[30] F. Haugen. The good gain method for simple experimental tuning of PI controllers. *Modeling, Identification and Control*, 33(4):141–152, 2012. ISSN 1890-1328. → pages 11

[31] A. Hedayat, N. Sloane, and J. Stufken. *Orthogonal Arrays: Theory and Applications*. Springer Series in Statistics. Springer New York, 1999. ISBN 9780387987668. URL https://books.google.ca/books?id=HrUYIlbI2mEC. → pages 20

[32] Z. Hou, M. M. Gupta, P. N. Nikiforuk, M. Tan, and L. Cheng. A recurrent neural network for hierarchical control of interconnected dynamic systems. *IEEE Transactions on Neural Networks*, 18(2):466–481, 2007. doi:10.1109/TNN.2006.885040. URL http://doi.ieeecomputersociety.org/10.1109/TNN.2006.885040. → pages 12

[33] D. Huh and E. Todorov. Real-time motor control using recurrent neural networks. In *Adaptive Dynamic Programming and Reinforcement Learning, 2009. ADPRL '09. IEEE Symposium on*, pages 42–49, March 2009. doi:10.1109/ADPRL.2009.4927524. → pages 9, 13, 19

[34] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages

604–613, New York, NY, USA, 1998. ACM. ISBN 0-89791-962-9. doi:10.1145/276698.276876. URL http://doi.acm.org/10.1145/276698.276876. → pages 49

[35] J. N. Ingram, K. P. Kording, I. S. Howard, and D. M. Wolpert. The statistics of natural hand movements. *Experimental brain research*, 188(2):223–236, 2008. doi:10.1007/s00221-008-1355-3. → pages 6

[36] G. L. Jones. Noisy optimal control strategies for modelling saccades. Master's thesis, University of British Columbia, 2011. → pages 5, 6, 9, 13

[37] V. R. Joseph and Y. Hung. Orthogonal-maximin Latin hypercube designs, 2008. → pages 29

[38] N. Kohl and P. Stone. Policy gradient reinforcement learning for fast quadrupedal locomotion. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2619–2624, May 2004. URL http://www.cs.utexas.edu/users/ai-lab/?kohl:icra04. → pages 5, 25, 38, 39, 80

[39] S.-H. Lee and D. Terzopoulos. Heads up!: Biomechanical modeling and neuromuscular control of the neck. *ACM Trans. Graph.*, 25(3):1188–1198, July 2006. ISSN 0730-0301. doi:10.1145/1141911.1142013. URL http://doi.acm.org/10.1145/1141911.1142013. → pages 2, 5, 6, 11

[40] W. Leithead. Survey of gain-scheduling analysis design. *International Journal of Control*, 73:1001–1025, 1999. → pages 12

[41] D. Li. Biomechanical simulation of the hand musculoskeletal system and skin. Master's thesis, University of British Columbia, 2013. → pages 44

[42] D. Li, S. Sueda, D. R. Neog, and D. K. Pai. Thin skin elastodynamics. *ACM Trans. Graph.*, 32(4):49:1–49:10, July 2013. ISSN 0730-0301. doi:10.1145/2461912.2462008. URL http://doi.acm.org/10.1145/2461912.2462008. → pages 26

[43] C. K. Liu. Dextrous manipulation from a grasping pose. *ACM Trans. Graph.*, 28(3):59:1–59:6, July 2009. ISSN 0730-0301. doi:10.1145/1531326.1531365. URL http://doi.acm.org/10.1145/1531326.1531365. → pages 5, 6, 9, 75

[44] X. Liu, C. Ma, M. Li, and M. Xu. A kriging assisted direct torque control of brushless DC motor for electric vehicles. In *Natural Computation (ICNC),*

2011 Seventh International Conference on, volume 3, pages 1705–1710, July 2011. doi:10.1109/ICNC.2011.6022349. → pages 17

[45] N. K. Malakar and K. H. Knuth. Entropy-Based Search Algorithm for Experimental Design. *AIP Conference Proceedings*, 1305(1):157–164, 2011. doi:10.1063/1.3573612. URL http://dx.doi.org/10.1063/1.3573612. → pages 20

[46] M. Malhotra, E. Rombokas, E. Theodorou, E. Todorov, and Y. Matsuoka. Reduced dimensionality control for the ACT hand. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 5117–5122, May 2012. doi:10.1109/ICRA.2012.6224651. → pages 2, 5, 11, 14, 46, 57, 58, 59, 80

[47] J. Martins, E. Pires, R. Salvado, and P. Dinis. A numerical model of passive and active behavior of skeletal muscles. *Computer Methods in Applied Mechanics and Engineering*, 151(34):419 – 433, 1998. ISSN 0045-7825. doi:http://dx.doi.org/10.1016/S0045-7825(97)00162-X. URL http://www.sciencedirect.com/science/article/pii/S004578259700162X. Containing papers presented at the Symposium on Advances in Computational Mechanics. → pages 7

[48] M. D. McKay, R. J. Beckman, and W. J. Conover. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 21(2):pp. 239–245, 1979. ISSN 00401706. URL http://www.jstor.org/stable/1268522. → pages 19

[49] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing Atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL http://arxiv.org/abs/1312.5602. → pages 19, 25

[50] I. Mordatch, Z. Popović, and E. Todorov. Contact-invariant optimization for hand manipulation. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '12, pages 137–144, Aire-la-Ville, Switzerland, Switzerland, 2012. Eurographics Association. ISBN 978-3-905674-37-8. URL http://dl.acm.org/citation.cfm?id=2422356.2422377. → pages 5, 12

[51] J. Najemnik and W. Geisler. Optimal eye movement strategies in visual search. *Nature*, 434:387–391, 2005. → pages 5

[52] J. Peters and S. Schaal. 2008 special issue: Reinforcement learning of motor skills with policy gradients. *Neural Netw.*, 21(4):682–697, May 2008. ISSN 0893-6080. doi:10.1016/j.neunet.2008.02.003. URL http://dx.doi.org/10.1016/j.neunet.2008.02.003. → pages 9, 25

[53] J. Peters and S. Schaal. Natural actor-critic. *Neurocomputing*, 71(79):1180 – 1190, 2008. ISSN 0925-2312. doi:http://dx.doi.org/10.1016/j.neucom.2007.11.026. URL http://www.sciencedirect.com/science/article/pii/S0925231208000532. Progress in Modeling, Theory, and Application of Computational Intelligenc 15th European Symposium on Artificial Neural Networks 2007 15th European Symposium on Artificial Neural Networks 2007. → pages 25

[54] J. R. Peters. *Machine Learning of Motor Skills for Robotics*. PhD thesis, University of Southern California, 2007. → pages 22

[55] R. Poole and N. G. S. (U.S.). *The Incredible Machine*. National Geographic Society, 1995. ISBN 9780792227298. → pages 5

[56] C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005. ISBN 026218253X. → pages 16

[57] M. Rolf. *Goal Babbling for an Efficient Bootstrapping of Inverse Models in High Dimensions*. PhD thesis, Bielefeld University, 2012. → pages 22

[58] M. Rolf. Goal babbling with unknown ranges: A direction-sampling approach. In *Development and Learning and Epigenetic Robotics (ICDL), 2013 IEEE Third Joint International Conference on*, pages 1–7, Aug 2013. doi:10.1109/DevLrn.2013.6652526. → pages 22

[59] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, Nov. 1958. → pages 19

[60] P. Sachdeva, S. Sueda, S. Bradley, M. Fain, and D. K. Pai. Biomechanical simulation and control of hands and tendinous systems. *ACM Trans. Graph.*, 34(4):42:1–42:10, July 2015. ISSN 0730-0301. doi:10.1145/2766987. URL http://doi.acm.org/10.1145/2766987. → pages 2, 6, 44, 46

[61] J. Sacks, W. J. Welch, T. J. Mitchell, and H. P. Wynn. Design and analysis of computer experiments. *Statist. Sci.*, 4(4):409–423, 11 1989. doi:10.1214/ss/1177012413. URL http://dx.doi.org/10.1214/ss/1177012413. → pages ix, 15, 16, 18, 19, 30

[62] S. Sahni. Computationally related problems. *SIAM Journal on Computing*, 3 (4):262–279, 1974. doi:10.1137/0203021. URL http://dx.doi.org/10.1137/0203021. → pages 82

[63] T. J. Santner, W. B., and N. W. *The Design and Analysis of Computer Experiments*. Springer-Verlag, 2003. → pages 17, 19, 20, 30

[64] I. Sardellitti, G. Medrano-Cerda, N. Tsagarakis, A. Jafari, and D. Caldwell. Gain scheduling control for a class of variable stiffness actuators based on lever mechanisms. *Robotics, IEEE Transactions on*, 29(3):791–798, June 2013. ISSN 1552-3098. doi:10.1109/TRO.2013.2244787. → pages 12, 67

[65] P. Sebastiani and H. P. Wynn. Maximum entropy sampling and optimal Bayesian experimental design. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 62(1):145–157, 2000. ISSN 1467-9868. doi:10.1111/1467-9868.00225. URL http://dx.doi.org/10.1111/1467-9868.00225. → pages 20

[66] R. Shadmehr. Equilibrium point hypothesis. In *The handbook of brain theory and neural networks*, pages 370–372. MIT Press, 1998. → pages 8, 80

[67] G. Shakhnarovich. Locality sensitive hashing (LSH). http://ttic.uchicago.edu/~gregory/download.html, 2009. → pages 49

[68] J. Shamma and M. Athans. Gain scheduling: potential hazards and possible remedies. *Control Systems, IEEE*, 12(3):101–107, June 1992. ISSN 1066-033X. doi:10.1109/37.165527. → pages 12

[69] X. Shu, P. Ballesteros, W. Heins, and C. Bohn. Design of structured discrete-time LPV gain-scheduling controllers through state augmentation and partial state feedback. In *American Control Conference (ACC), 2013*, pages 6090–6095, June 2013. doi:10.1109/ACC.2013.6580793. → pages 12, 67

[70] W. Si, S.-H. Lee, E. Sifakis, and D. Terzopoulos. Realistic biomechanical simulation and control of human swimming. *ACM Trans. Graph.*, 34(1): 10:1–10:15, Dec. 2014. ISSN 0730-0301. doi:10.1145/2626346. URL http://doi.acm.org/10.1145/2626346. → pages 5, 6, 9, 11, 13, 14, 19

[71] D. L. Sparks. The brainstem control of saccadic eye movements. *Nature Reviews Neuroscience*, 3:952–964, Dec 2002. doi:10.1038/nrn986. → pages 9

[72] S. Sueda, A. Kaufman, and D. K. Pai. Musculotendon simulation for hand animation. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 27(3), 2008. → pages 2, 6, 44, 46, 65, 80

[73] S. Sueda, G. L. Jones, D. I. W. Levin, and D. K. Pai. Large-scale dynamic simulation of highly constrained strands. *ACM Trans. Graph.*, 30(4): 39:1–39:10, July 2011. ISSN 0730-0301. doi:10.1145/2010324.1964934. URL http://doi.acm.org/10.1145/2010324.1964934. → pages 45

[74] R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998. ISBN 0262193981. → pages 13, 22, 24

[75] C. Szepesvari and M. L. Littman. Generalized Markov decision processes: Dynamic-programming and reinforcement-learning algorithms. Technical report, University of Alberta, 1996. → pages 24

[76] D. Thalmann and M. van de Panne. *Computer Animation and Simulation 97: Proceedings of the Eurographics Workshop in Budapest, Hungary, September 2–3, 1997*. Eurographics. Springer Vienna, 2012. ISBN 9783709168745. → pages 26

[77] E. Theodorou. *Iterative Path Integral Stochastic Optimal Control: Theory and Applications to Motor Control*. PhD thesis, University of Southern California, 2011. → pages 12

[78] L. H. Ting and J. L. McKay. Neuromechanics of muscle synergies for posture and movement. *Current Opinion in Neurobiology*, 17(6):622 – 628, 2007. ISSN 0959-4388. doi:http://dx.doi.org/10.1016/j.conb.2008.01.002. URL http://www.sciencedirect.com/science/article/pii/S0959438808000044. Motor systems / Neurobiology of behaviour. → pages 5, 12

[79] H. Van Welbergen, B. J. H. Van Basten, A. Egges, Z. M. Ruttkay, and M. H. Overmars. Real time animation of virtual humans: A trade-off between naturalness and control. *Computer Graphics Forum*, 29(8):2530–2554, 2010. ISSN 1467-8659. doi:10.1111/j.1467-8659.2010.01822.x. URL http://dx.doi.org/10.1111/j.1467-8659.2010.01822.x. → pages 14

[80] N. Vitiello, T. Lenzi, S. M. M. D. Rossi, S. Roccella, and M. C. Carrozza. A sensorless torque control for antagonistic driven compliant joints. *Mechatronics*, 20(3):355 – 367, 2010. ISSN 0957-4158. doi:http://dx.doi.org/10.1016/j.mechatronics.2010.02.001. URL

http://www.sciencedirect.com/science/article/pii/S0957415810000309. → pages 1

[81] K. Wampler, J. Popović, and Z. Popović. Animal locomotion controllers from scratch. In *EUROGRAPHICS*, volume 32, 2013. → pages 22

[82] J. Wang, D. Fleet, and A. Hertzmann. Gaussian process dynamical models for human motion. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 30(2):283–298, Feb 2008. ISSN 0162-8828. doi:10.1109/TPAMI.2007.1167. → pages 15, 17

[83] J. M. Wang, D. J. Fleet, and A. Hertzmann. Optimizing walking controllers. *ACM Trans. Graph.*, 28(5):168:1–168:8, Dec. 2009. ISSN 0730-0301. doi:10.1145/1618452.1618514. URL http://doi.acm.org/10.1145/1618452.1618514. → pages 5, 6, 11, 14

[84] D. Wolpert and M. Kawato. Multiple paired forward and inverse models for motor control. *Neural Networks*, 11(78):1317 – 1329, 1998. ISSN 0893-6080. doi:http://dx.doi.org/10.1016/S0893-6080(98)00066-5. URL http://www.sciencedirect.com/science/article/pii/S0893608098000665. → pages 2, 5, 6, 10

[85] D. M. Wolpert, J. Diedrichsen, and J. R. Flanagan. Principles of sensorimotor learning. *Nature reviews. Neuroscience*, 12(12):739–751, 2011. ISSN 1471-0048. URL http://view.ncbi.nlm.nih.gov/pubmed/22033537. → pages 5, 6, 13

[86] H. Wu and F. Sun. Adaptive kriging control of discrete-time nonlinear systems. *Control Theory Applications, IET*, 1(3):646–656, May 2007. ISSN 1751-8644. → pages 17

[87] K. Yin, K. Loken, and M. van de Panne. SIMBICON: Simple biped locomotion control. *ACM Trans. Graph.*, 26(3):Article 105, 2007. → pages 5, 6, 11, 14

[88] A. Zhang, M. Malhotra, and Y. Matsuoka. Musical piano performance by the ACT hand. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 3536–3541, May 2011. doi:10.1109/ICRA.2011.5980342. → pages 2, 11, 12, 14

[89] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics (TOG)*, 27(5):126, 2008. → pages 18

[90] J. G. Ziegler and N. B. Nichols. Optimum Settings for Automatic Controllers. *Transactions of ASME*, 64:759–768, 1942. → pages 11

[91] V. B. Zordan, A. Majkowska, B. Chiu, and M. Fast. Dynamic response for motion capture animation. *ACM Trans. Graph.*, 24(3):697–701, July 2005. ISSN 0730-0301. doi:10.1145/1073204.1073249. URL http://doi.acm.org/10.1145/1073204.1073249. → pages 15