# CONTROL PRIMITIVES FOR ROBOT SYSTEMS

by

D. Curtis Deno, Richard M. Murray, Kristofer S.J. Pister,
and S. Shankar Sastry

Memorandum No. UCB/ERL M90/39

4 May 1990

# CONTROL PRIMITIVES FOR ROBOT SYSTEMS

by

D. Curtis Deno, Richard M. Murray, Kristofer S.J. Pister,
and S. Shankar Sastry

Memorandum No. UCB/ERL M90/39

4 May 1990

**ELECTRONICS RESEARCH LABORATORY**

College of Engineering
University of California, Berkeley
94720

# CONTROL PRIMITIVES FOR ROBOT SYSTEMS

by

D. Curtis Deno, Richard M. Murray, Kristofer S.J. Pister,
and S. Shankar Sastry

Memorandum No. UCB/ERL M90/39

4 May 1990

**ELECTRONICS RESEARCH LABORATORY**

College of Engineering
University of California, Berkeley
94720

## Abstract

Inspired by the organization of the mammalian neuro-muscular system, this report develops a methodology for description of hierarchical control in a manner which is faithful to the underlying mechanics, structured enough to be used as an interpreted language, and sufficiently flexible to allow the description of a wide variety of systems. We present a consistent set of primitive operations which form the core of a robot system description and control language. This language is capable of describing a large class of robot systems under a variety of single level and distributed control schemes. We review a few pertinent results of classical mechanics, describe the functionality of our primitive operations, and present several different hierarchical strategies for the description and control of a two-fingered hand holding a box. An implementation of the primitives in the form of a Mathematica package is given in an appendix.

# Contents

# Chapter 1

# Introduction

The complexity of compound, redundant robotic systems, both in specification and control, continues to present a challenge to engineers and biologists. Complex robot actions require coordinated motion of multiple robot arms or fingers to manipulate objects and respect physical constraints. As we seek to achieve more of the capability of biological robots, additional descriptive structures and control schemes are necessary. A major aim of this work is to propose such a specification and control scheme. The ultimate goal of our project is to build a high level task programming environment which is relatively robot independent.

In Chapter 2 we review the dynamics and control of coupled, constrained rigid robots in a Lagrangian framework. Chapter 3 contains definitions of the primitives of our robot control environment. Chapter 4 illustrates the application of our primitives to the description of a two fingered robot hand. We show that our environment can be used to specify a variety of control schemes for this hand, including a distributed controller which has a biological analog. Chapter 5 extends the basic primitives to include specification and control of constraint forces and redundant motion. In Chapter 6 we discuss future avenues of research. The remainder of this introduction presents motivation and background for our work, and an overview of the primitives we have chosen to use.

## 1.1 The Musculoskeletal System: Metaphor for a Robotic System

Motivation for a consistent specification and control scheme may be sought in our current knowledge of the hierarchical organization of mammalian motor systems. To some degree of accuracy, we may consider segments of limbs as rigid bodies connected by rotary joints. Muscles and tendons are actuators with sensory feedback which enter into low level feedback control at the spinal level [9].

Further up the nervous system, the brainstem, cerebellum, thalamus, and basal ganglia integrate ascending sensory information and produce coordinated motor commands. At the highest levels, sensory and motor cortex supply conscious goal-related information, trajectory specification, and monitoring.

The hierarchical structure of neuromuscular control is also evident from differences in time scale. The low level spinal reflex control runs faster (loop delays of about 30 ms) than the high level feedback loops (100–200 ms delays). This distinction may be exploited by control schemes which hide information details from high level controllers by virtue of low level control enforcing individual details. These concepts are shown in Figure 1.1 where a drawing of neuromuscular control structures for a finger is juxtaposed with a block diagram to emphasize the hierarchical nature of the thumb-forefinger system for picking up objects.

Biological control systems commonly operate with constraints and redundancy. Kinematic constraints arise not only from joints which restrict the relative motion of adjacent limb segments, but also from contact with objects which leads to similar restrictions. Many musculo-skeletal subsystems possess kinematic and actuator redundancy which may be imagined to be resolved by effort and stability considerations. In any event, the neural controller directs a specific strategy and so expands a reduced set of control variables into the larger complete set.

In the sequel we shall see these concepts expressed in a notation which is faithful to the laws of mechanics and flexible enough to permit concise descriptions of robot motion control at various hierarchical levels.

## 1.2 Background

The robotics and control literature contains a number of topics which are related to the specification and control scheme of this paper.

### Robot programming languages

Two directions of emphasis may be used to distinguish robot programming languages: traditional programming languages (perhaps including multitasking), and dynamical systems based descriptions of systems and control structures.

More traditional task specification languages include VAL II, AML, and Robot-BASIC [6, 20, 8, 19]. These languages are characterized by C-, BASIC-, or Lisp-like data structures and syntax, coordinate frame specification and transformation primitives, sensor feedback conditionally controlling program flow, and motion between specified locations achieved through via points and interpolation. In a two stage hierarchy, low level controllers usually control joint angle trajectories which are specified by high level language statements and kinematics computations.

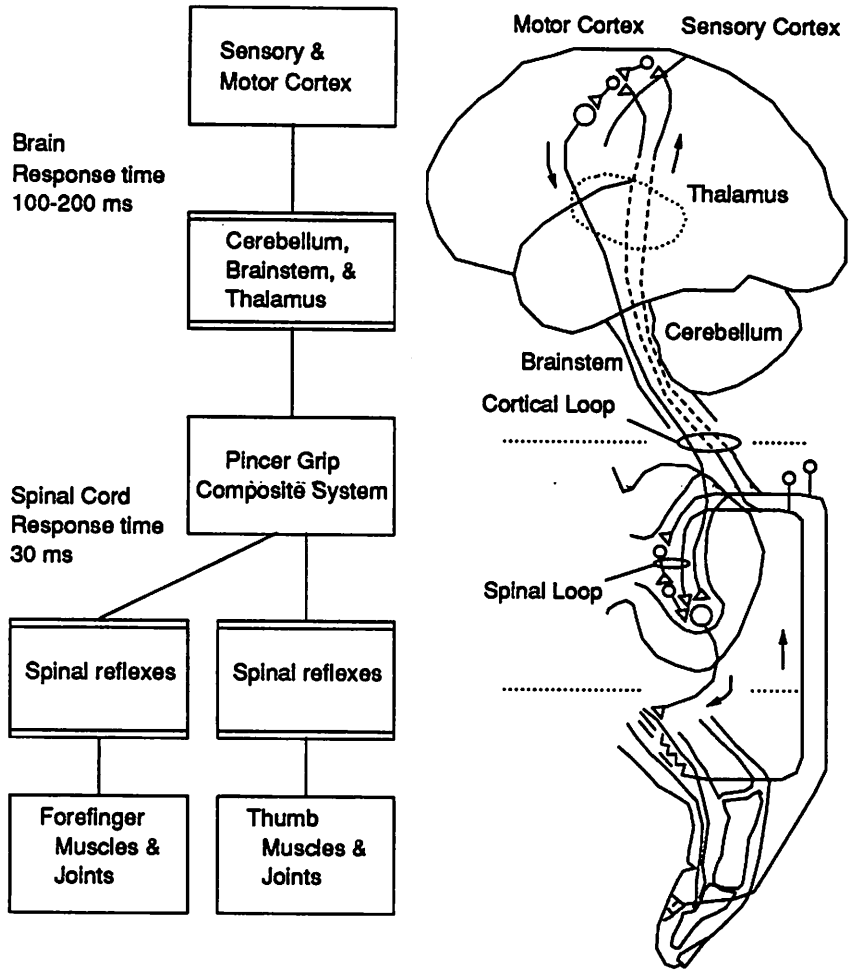


Figure 1.1: Hierarchical control scheme of a human finger. At the highest level, the brain is represented as sensory and motor cortex (where sensory information is perceived and conscious motor commands originate ) and brainstem and cerebellar structures (where motor commands are coordinated and sent down the spinal cord). A pair of fingers forms a composite system for grasping which is shown integrated at the level of the spinal cord. The muscles and sensory organs of each finger form low level spinal reflex loops. These low level loops respond more quickly to disturbances than sensory motor pathways which travel to the brain and back. Brain and spinal feedback controllers are represented by double lined boxes.

Brockett's Motion Description Language [3, 7] (MDL) is more closely aligned with dynamical systems theory. MDL employs sequences of triples $(u, k, T)$ to convey trajectory information, feedback control information, and time interval to an extensible Forth/PostScript like interpreter. The scheme described in this paper was inspired partly by descriptions of MDL. Our work explicitly utilizes geometric and inertial parameters together with the equations of motion to describe the organization and control of complex robots. MDL is less explicit on this matter but is more completely developed in the matter of sequences of motions.

### Distributed control, hierarchical control

The nervous system controls biomechanical robots using both distributed controllers and hierarchical organization [9]. For example, spinal reflex centers can direct portions of gait in cats and the wiping motions of frog limbs without the brain. One reason for a hierarchical design is that high level feedback loops may respond too slowly for all of motor control to be localized there. Indeed the complexity and time delays inherent in biological motor control led the Russian psychologist Bernstein to conclude the brain could not achieve motor control by an internal model of body dynamics [10].

Centralized control has been defined as a case in which every sensor's output influences every actuator. Decentralized control was a popular topic in control theory in the late 1970's and led to a number of results concerning weakly coupled systems and multi-rate controllers [23]. Graph decomposition techniques, applied to the graph structures employed in a hierarchical scheme, permitted the isolation of sets of states, inputs, and outputs which were weakly coupled. This decomposition facilitated stability analyses and controller design. Robotic applications of hierarchical control are exemplified by HIC [4] which manages multiple low level servo loops with a robot programming language from the "traditional" category above. One emphasis of such control schemes concerns distributed processing and interprocess communication.

## 1.3 Overview of Robot Control Primitives

The fundamental objects in our robot specification environment are objects called robots. In a graph theoretic formalism they are nodes of a tree structure. At the lowest level of the tree are leaves which are instantiated by the `define` primitive. Robots are dynamical systems which are recursively defined in terms of the properties of their daughter robot nodes. Inputs to robots consist of desired positions and conjugate forces. The outputs of a robot consist of actual positions and forces. Robots also possess attributes such as inertial parameters and kinematics.

There are two other primitives which act on sets of robots to yield new

robots, so that the set of robots is closed under these operations. These primitives (`attach` and `control`) may be considered as links between nodes and result in composite robot objects. Thus nodes closer to the root may possess fewer degrees of freedom, indicating a compression of information upon ascending the tree.

The `attach` primitive reflects geometrical constraints among variables and in the process of yielding another robot object, accomplishes coordinate transformations. `Attach` is also responsible for a bidirectional flow of information: expanding desired positions and forces to the robots below, and combining actual position and force information into an appropriate set for the higher level robot. In this sense the state of the root robot object is recursively defined in terms of the states of the daughter robots.

The `control` primitive seeks to direct a robot object to follow a specified "desired" position/force trajectory according to some control algorithm. The controller applies its control law (several different means of control are available such as PD and computed torque) to the desired and actual states to compute expected states for the daughter robot to follow. In turn, the daughter robot passes its actual states through the controller to robot objects further up the tree.

The block diagram portion of Figure 1.1 may be seen to be an example of a robot system comprised of these primitives. Starting from the bottom: two fingers are `defined`; each finger is `controlled` by muscle tension/stiffness and spinal reflexes; the fingers are `attached` to form a composite hand; the brainstem and cerebellum help `control` and coordinate motor commands and sensory information; and finally at the level of the cortex, the fingers are thought of as a pincer which engages in high level tasks such as picking.

# Chapter 2

# Review of robot dynamics and control

In this chapter we selectively review the dynamics and control of robot systems. The basic result is that even for relatively complicated robot systems, the equations of motion for the system can be written in a standard form. This point of view has been used by Khatib in his operational space formulation [12] and in some recent extensions [13]. The results presented in this chapter are direct extensions of those works, although the approach is different.

The dynamics for a robot manipulator with joint angles $\theta \in \mathbf{R}^n$ and actuator torques $\tau \in \mathbf{R}^n$ can be derived using Lagrange's equations and written in the form

$$M(\theta)\ddot{\theta} + C(\theta, \dot{\theta})\dot{\theta} + N(\theta, \dot{\theta}) = \tau \tag{2.1}$$

where $M(\theta)$ is a positive definite inertia matrix and $C(\theta, \dot{\theta})\dot{\theta}$ is the Coriolis and centrifugal force vector. The vector $N(\theta, \dot{\theta}) \in \mathbf{R}^n$ contains all friction and gravity terms and the vector $\tau \in \mathbf{R}^n$ represents generalized forces in the $\theta$ coordinate frame. For systems of this type, it can be shown that $\dot{M} - 2C$ is a skew symmetric matrix with proper choice of $C$ (such as that in [25]).

## 2.1 Constrained manipulators

Constrained robot systems can also be represented by dynamics in the same form as equation (2.1). As our main example, consider the control of a multi-fingered hand grasping a box (Figure 2.1) where $\theta$ is a vector of all the joint angles and $x$ is a vector describing the position and orientation of the box. The grasping constraint may be written as

$$J(q)\dot{\theta} = G^T(q)\dot{x} \tag{2.2}$$

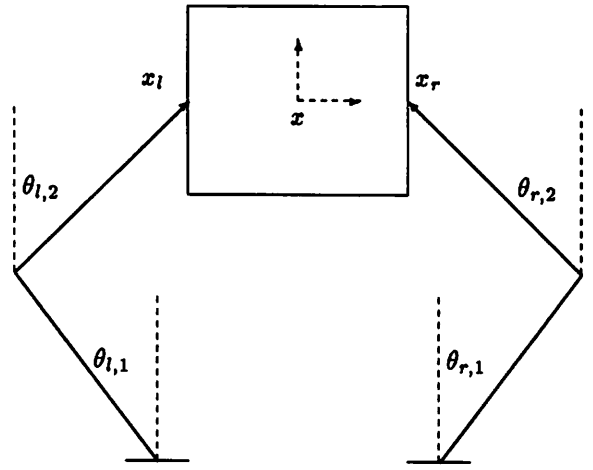


Figure 2.1: Planar two-fingered hand. Contacts are assumed to be maintained throughout the motion. For this particular system the box position and orientation, $x$, form a generalized set of coordinates for the system.

where $q = (\theta, x) \in \mathbf{R}^m \times \mathbf{R}^n$, $J$ is the Jacobian of the finger kinematic function and $G$ is the "grasp map" for the system. We will assume that $J$ is bijective in some neighborhood and that $G$ is surjective. This form of constraint can also be used to describe a wide variety of other systems, including grasping with rolling contacts, surface following and coordinated lifting. For the primitives presented in the next chapter, we also assume that there exists a forward kinematic function between $\theta$ and $x$; that is, the constraint is holonomic. A more complete derivation of grasping kinematics can be found in a recent paper by Murray and Sastry [18].

To include velocity constraints we again appeal to Lagrange's equations. Following the approach in Rosenberg [21], the equations of motion for our constrained system can be written as

$$\tilde{M}(q)\ddot{x} + \tilde{C}(q,\dot{q})\dot{x} + \tilde{N}(q,\dot{q}) = F_e \tag{2.3}$$

where

$$\begin{aligned}
\tilde{M} &= M + GJ^{-T}M_\theta J^{-1}G^T \\
\tilde{C} &= C + GJ^{-T}\left(C_\theta J^{-1}G^T + M_\theta \frac{d}{dt}\left(J^{-1}G^T\right)\right) \\
\tilde{N} &= GJ^{-T}N \\
F_e &= GJ^{-T}\tau \\
M, M_\theta &= \text{inertia matrix for the box and fingers, respectively}
\end{aligned}$$

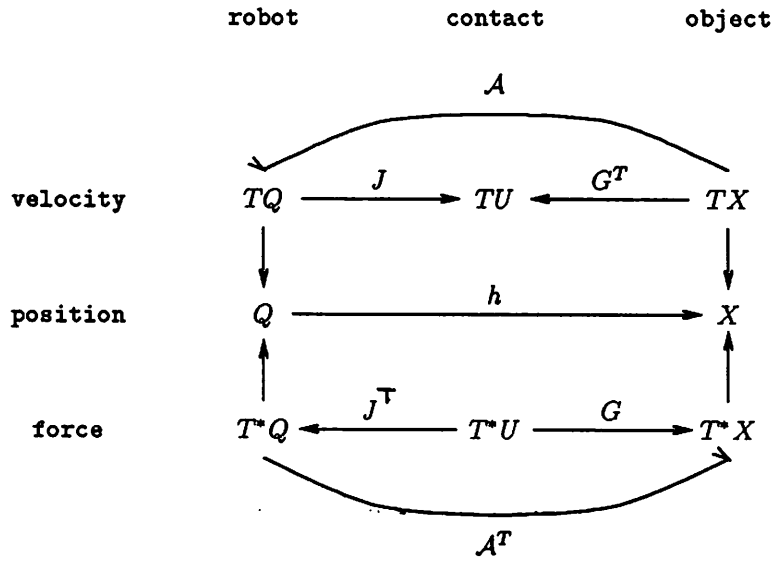


Figure 2.2: Commutative diagram for robot coordinate frames. In this figure, $Q$ represents the configuration manifold of the robot, $X$ the configuration manifold of the object. $TU$ is the space in which the velocity constraints live.

$$C, C_\theta \quad = \quad \text{Coriolis and centrifugal terms}$$

Thus we have an equation with a similar form to our "simple" robot. In the box frame of reference, $\tilde{M}$ is the mass of the effective mass of the box, and $\tilde{C}$ is the effective Coriolis and centrifugal matrix. These matrices include the dynamics of the fingers, which are being used to actually control the motion of the box. However the details of the finger kinematics and dynamics are effectively hidden in the definition of $\tilde{M}$ and $\tilde{C}$. The skew symmetry of $\dot{\tilde{M}} - 2\tilde{C}$ is preserved by this transformation.

A diagram of the relations between the various coordinates frames is show in Figure 2.2.

## 2.2 Internal forces

Although the grasp map $G$ was assumed surjective, it need not be square. From the equations of motion (2.3), we note that if fingertip force $J^{-T}\tau$ is in the null space of $G$ then the net force in the object frame of reference is zero and causes no net motion of the object. These forces act against the constraint and are generally termed *internal* or *constraint* forces. We can use these internal forces

to satisfy other conditions, such as keeping the contact forces inside the friction cone (to avoid slipping) or varying the load distribution of a set of manipulators rigidly grasping an object.

To include the internal forces in our formulation, we extend the grasp map by defining an orthonormal matrix $H(\theta)$ whose rows form a basis for the null space of $G(\theta)$. As before we assume that $G(\theta)$ has constant rank and we break all forces up into an external and an internal piece, $F_e$ and $F_i$. Given these desired forces, the torques that should be applied by the actuators is

$$\tau = J^T \begin{pmatrix} G \\ H \end{pmatrix}^{-1} \begin{pmatrix} F_e \\ F_i \end{pmatrix} = J^T \begin{pmatrix} G^+ & H^T \end{pmatrix} \begin{pmatrix} F_e \\ F_i \end{pmatrix} \tag{2.4}$$

## 2.3 Redundant manipulators

Some manipulators contain more degrees of freedom than are necessary to specify the position of the end effector. Mathematically, these robots can be represented by a change of coordinates $f : \mathbf{R}^m \to \mathbf{R}^n$ where $m > n$. In this case $J := \frac{\partial f}{\partial \theta}$ is not square and hence $J^{-1}$ is not well defined so the derivation of equation (2.3) does not hold.

It is still possible to write the dynamics of redundant manipulators in a form consistent with equation (2.3). To do so, we first define a matrix $K(\theta)$ whose rows span the null space of $J(\theta)$. As before we assume that $J(\theta)$ is full row rank and hence $K(\theta)$ has constant rank $m - n$. The rows of $K(\theta)$ are basis elements for the space of velocities which cause no motion of the end effector; we can thus define an *internal motion*, $\dot{x}_i \in \mathbf{R}^{m-n}$ using the equation

$$\begin{pmatrix} \dot{x}_e \\ \dot{x}_i \end{pmatrix} = \begin{pmatrix} J \\ K \end{pmatrix} \dot{\theta} = \bar{J}\dot{\theta} \tag{2.5}$$

By construction, $\bar{J}$ is full rank (and square) so we can use the previous derivation to conclude that

$$\tilde{M}(q) \begin{pmatrix} \ddot{x}_e \\ \ddot{x}_i \end{pmatrix} + \tilde{C}(q, \dot{q}) \begin{pmatrix} \dot{x}_e \\ \dot{x}_i \end{pmatrix} + \tilde{N}(q, \dot{q}) = F \tag{2.6}$$

where $\tilde{M}$ and $\tilde{C}$ are obtained from equation (2.3) replacing $J$ with $\bar{J}$ and having augmented $G$ with a block diagonal identity matrix to preserve the $\dot{x}_i$'s. If we choose $K$ such that its rows are orthonormal then $\bar{J}^{-1} = (J^+ \; K^T)$ where $J^+ = J^T(JJ^T)^{-1}$ is the least-squares inverse of $J$. This approach is related to the *extended Jacobian* technique for resolving kinematic redundancy [1].

## 2.4 Control

To illustrate the control of robot systems, we look at two controllers which have appeared in the robotics literature. We start by considering systems of the form

$$M(q)\ddot{x} + C(q,\dot{q})\dot{x} + N(q,\dot{q}) = F \tag{2.7}$$

where $M(q)$ is a positive definite inertia matrix and $C(q,\dot{q})\dot{x}$ is the Coriolis and centrifugal force vector. The vector $N(q,\dot{q}) \in \mathbf{R}^n$ contains all friction and gravity terms and the vector $F \in \mathbf{R}^n$ represents generalized forces in the $x$ coordinate frame.

For the case of more complicated manipulators the dynamics look essentially the same with appropriate definition of $x$ and $F$. For redundant manipulators we define $x$ as $(x_e, x_i)$ where it is understood that only the derivatives of $x_i$ can actually be used in a control law (since these are well defined). In particular, when we refer to a vector of position errors, $e$, it is assumed that the portion of the error vector corresponding to $x_i$ is taken to be zero. For manipulators which contain constraints, $F$ is actually $F_e$ and there is an additional force term, $F_i$, which causes no net motion of the system. Both internal motions and forces are specified in terms of the basis vectors for the appropriate null spaces.

### Computed torque

Computed torque is an exactly linearizing control law that has been used extensively in robotics research. It has been used for joint level control [2], Cartesian control [16], and most recently, control of multi-fingered hands [15, 5]. Given a desired trajectory $x_d$ we use the control

$$F = M(q)\left(\ddot{x}_d + K_v\dot{e} + K_p e\right) + C(q,\dot{q})\dot{x} + N(q,\dot{q}) \tag{2.8}$$

where error $e = x_d - x$ and $K_v$ and $K_p$ are constant gain matrices. The resulting dynamics equations are linear with exponential rate of convergence determined by $K_v$ and $K_p$. Since the system is linear, we can use linear control theory to choose the gains ($K_v$ and $K_p$) such that they satisfy some set of design criteria.

The disadvantage of this control law is that it is not easy to specify the interaction with the environment. From the form of the error equation we might think that we could use $K_p$ to model the stiffness of the system and exert forces by commanding trajectories which result in fixed errors. Unfortunately this is not uniformly applicable as can be seen by examining the force due to a quasi-static displacement $\Delta x$:

$$\Delta F = M(q)K_p\Delta x \tag{2.9}$$

Since $K_p$ must be constant in order to prove stability, the resultant stiffness will vary with configuration. Additionally, given a desired stiffness matrix it may not be possible to find a positive definite $K_p$ that achieves that stiffness.

## PD + feedforward control

PD controllers differ from computed torque controllers in that the desired stiffness (and potentially damping) of the end effector is specified, rather than its position tracking characteristics. Typically, control laws of this form rely on the skew-symmetric property of robot dynamics, namely $\alpha^T \left(\dot{M} - 2C\right) \alpha = 0$ for all $\alpha \in \mathbf{R}^n$. Consider the control law

$$F = M(q)\ddot{x}_d + C(q,\dot{q})\dot{x}_d + N(q,\dot{q}) + K_v\dot{e} + K_p e \tag{2.10}$$

where $K_v$ and $K_p$ are symmetric positive definite. Using a Liapunov stability argument, it can be shown that the actual trajectory of the robot converges to the desired trajectory asymptotically [14]. Extensions to the control law result in exponential rate of convergence [24, 22].

This PD control law has the advantage that for a quasi-static change in position $\Delta x$ the resulting force is

$$\Delta F = K_p \Delta x \tag{2.11}$$

and thus we can achieve an arbitrary symmetric stiffness. Experimental results indicate that the trajectory tracking performance of this control law does not always compare favorably with the computed torque control law [17]. Additionally there is no simple design criteria for choosing $K_v$ and $K_p$ to achieve good tracking performance. While the stability results give necessary conditions for stability they do not provide a method for choosing the gains. Nonetheless, PD control has been used effectively in many robot controllers and has some computational features which make it an attractive alternative.

# Chapter 3

# Primitives

In this chapter we describe a set of primitives that gives us the mathematical structure necessary to build a system and control specification for dynamical robot systems. We do not require any particular programming environment or language, borrowing instead freely from languages such as C, Lisp and C++. As much as possible, we have tried to define the primitives so that they can be implemented in any of these languages.

As our basic data structure, we will assume the existence of an object with an associated list of attributes. These attributes can be thought of as a list of name-value pairs which can be assigned and retrieved by name. A typical attribute which we will use is the inertia of a robot. The existence of such an attribute implies the existence of a function which is able to evaluate and return the inertia matrix of a robot given its configuration.

Attributes will be assigned values using the notation *attribute* := *value*. Thus we might define our inertia attribute as

$$M(\theta) := \begin{bmatrix} m_1 l_1^2 + m_2 l_2^2 & m_2 l_1 l_2 \cos(\theta_1 - \theta_2) \\ m_2 l_1 l_2 \cos(\theta_1 - \theta_2) & m_2 l_2^2 \end{bmatrix} \tag{3.1}$$

In order to evaluate the inertia attribute, we would call $M$ with a vector $\theta \in \mathbf{R}^2$. This returns a $2 \times 2$ matrix as defined above. The Coriolis/centrifugal attribute, $C$, and friction/gravity/nonlinear attribute, $N$, are defined similarly.

To encourage intuition, we will first describe the actions of the primitives for the case of non-redundant robots. Additionally, we ignore the internal forces that are present in constrained systems. Extensions to these cases are presented in Chapter 5.

## 3.1 The robot object

The fundamental object used by all primitives is a *robot*. Associated with a robot are a set of attributes which are used to define its behavior:

| | |
|---|---|
| $M$ | inertia of the robot |
| $C$ | Coriolis/centrifugal vector |
| $N$ | friction and gravity vector |
| *rd* | return position and force information about the robot |
| *wr* | send position and information to the robot |

The *rd* function returns the current position, velocity, and acceleration of the robot, and the forces measured by the robot. Each of these will be a vector quantity of dimension equal to the number of degrees of freedom of the robot. Typically a robot may only have access to its joint positions and velocities, in which case $\ddot{x}$ and $F$ will be *nil*.

The *wr* function is used to specify an expected position and force trajectory that the robot is to follow. In the simplest case, a robot would ignore everything but $F$ and try to apply this force/torque at its actuators. As we shall see later, other robots may use this information in a more intelligent fashion. We will often refer to the arguments passed to write by using the subscript $e$. Thus $x_e$ is the expected or desired position passed to the *wr* function.

The task of describing a primitive is essentially the same as describing how it generates the attributes of the new robot. The following sections describe how each of the primitives generates these attributes. The new attributes created by a primitive are distinguished by a tilde over the name of the attribute.

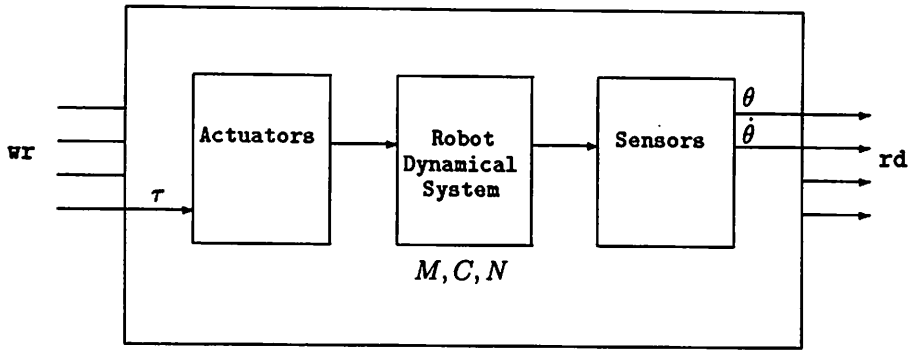


Figure 3.1: Example of the `define` primitive. The robot shown here corresponds to a robot with torque driven motors and only position and velocity sensing.

## 3.2 DEFINE primitive

`Synopsis:`
`DEFINE(`$M$`,` $C$`,` $N$`,` $rd$`,` $wr$`)`

The `define` primitive is used to create a simple robot object. It defines the minimal set of attributes necessary for a robot. These attributes are passed as arguments to the `define` primitive and a new robot object possessing those attributes is created:

$$\begin{aligned} \tilde{M}(\theta) &:= M(\theta) \\ \tilde{C}(\theta,\dot{\theta}) &:= C(\theta,\dot{\theta}) \\ \tilde{N}(\theta,\dot{\theta}) &:= N(\theta,\dot{\theta}) \\ \tilde{rd}() &:= rd() \\ \tilde{wr}(\theta_e,\dot{\theta}_e,\ddot{\theta}_e,\tau_e) &:= wr(\theta_e,\dot{\theta}_e,\ddot{\theta}_e,\tau_e) \end{aligned}$$

Several different types of robots can be defined using this basic primitive. For example, a DC motor actuated robot would be implemented with a $wr$ function which converts the desired torque to a motor current and generates this current by communicating with some piece of hardware (such as a D/A converter). This type of robot system is shown in Figure 3.1. On the other hand, a stepper motor actuated robot might use a $wr$ function which ignores the torque argument and uses the position argument to move the actuator. Both robots would use a $rd$ function which returns the current position, velocity, acceleration and actuator torque. If any of these pieces of information is missing, it is up to the user to insure that they are not needed at a higher level. We may also define a *payload*

as a degenerate robot by setting the *wr* argument to the *nil* function. Thus commanding a motion and/or force on a payload produces no effect.

## 3.3 ATTACH primitive

Synopsis:

```
ATTACH(J, G, h, payload, robot-list)
```

`Attach` is used to describe constrained motion involving a payload and one or more robots. `Attach` must create a new robot object from the attributes of the payload and of the robots being attached to it. The specification of the new robot requires a velocity relationship between coordinate systems ($J\dot{\theta} = G^T\dot{x}$), an invertible kinematic function relating robot positions to payload position ($x = h(\theta)$), a payload object, and a list of robot objects involved in the contact.

The only difference between the operation of the `attach` primitive and the equations derived for constrained motion of a robot manipulator is that we now have a *list* of robots each of which is constrained to contact a payload. However, if we define $\theta_R$ to be the combined joint angles of the robots in `robot-list` and similarly define $M_R$ and $C_R$ as block diagonal matrices composed of the individual inertia and Coriolis matrices of the robots, we have a system which is identical to that presented previously. Namely, we have a "robot" with joint angles $\theta_R$ and inertia matrix $M_R$ connected to an object with a constraint of the form

$$J\dot{\theta}_R = G^T\dot{x} \tag{3.2}$$

where once again $J$ is a block diagonal matrix composed of the Jacobians of the individual robots. To simplify notation, we will define $\mathcal{A} := J^{-1}G^T$ so that

$$\dot{\theta}_R = \mathcal{A}\dot{x} \tag{3.3}$$

The attributes of the new robot can thus be defined as:

$$\tilde{M} := M_p + \mathcal{A}^T M_R \mathcal{A} \tag{3.4}$$

$$\tilde{C} := C_p + \mathcal{A}^T C_R \mathcal{A} + \mathcal{A}^T M_R \dot{\mathcal{A}} \tag{3.5}$$

$$\tilde{N} := N_p + \mathcal{A}^T N_R \tag{3.6}$$

$$\tilde{rd}() := (h(\theta_R),\ \mathcal{A}^+\dot{\theta}_R,\ \mathcal{A}^+\ddot{\theta}_R + \dot{\mathcal{A}}^+\dot{\theta}_R,\ \mathcal{A}^T\tau_R) \tag{3.7}$$

$$\tilde{wr}(x_e, \dot{x}_e, \ddot{x}_e, F_e) := wr_R(h^{-1}(x_e),\ \mathcal{A}\dot{x}_e,\ \mathcal{A}\ddot{x}_e + \dot{\mathcal{A}}\dot{x}_e,\ \mathcal{A}^{+T}F_e) \tag{3.8}$$

where $M_p, C_p, N_p$ are attributes of the payload, $M_R$ and $C_R$ are as described above and $N_R$ is a stacked vector of friction and gravity forces. This construction is illustrated in Figure 3.2.

The $\tilde{rd}$ attribute for an attached robot is a function which queries the state of all the robots in `robot-list`. Thus $\theta_R$ in equation (3.7) is constructed by calling the individual *rd* functions for all of the robots in the list. The $\theta$ values for each of these robots are then combined to form $\theta_R$ and this is passed to the forward kinematic function. A similar computation occurs for $\dot{\theta}_R$, $\ddot{\theta}_R$ and $\tau_R$. Together, these four pieces of data form the return value for the *rd* attribute.

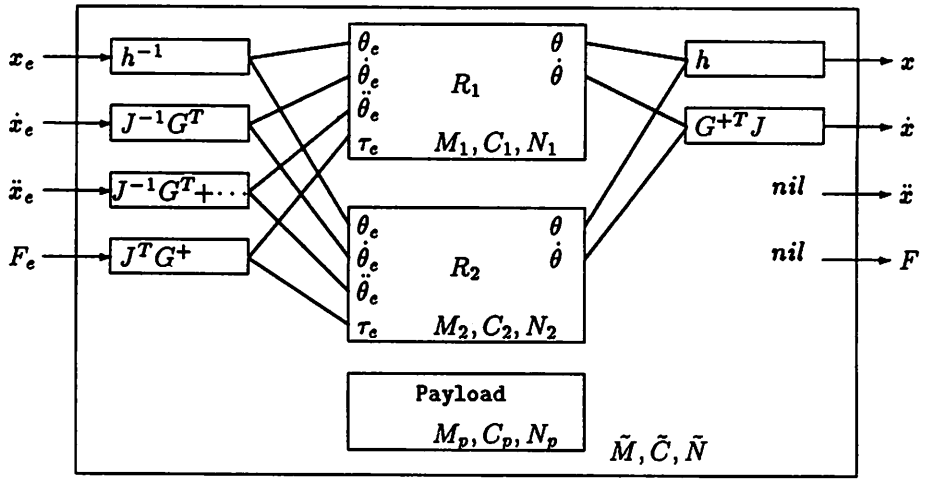


Figure 3.2: Data flow in a two robot **attach**. In this example we illustrate the structure generated by a call to **attach** with 2 robots and a payload (e.g. a system like Figure 2.1). The two large interior boxes represent the two robots, with their input and output functions and their inertia properties. The outer box (which has the same structure as the inner boxes) represents the new robot generated by the call to **attach**. In this example the robots do not have acceleration or force sensors, so these outputs are set to *nil*.

In a dual manner, the $\tilde{wr}$ attribute is defined as a function which takes a desired trajectory (position and force), converts it to the proper coordinate frame and sends each robot the correct portion of the resultant trajectory. A special case of the **attach** primitive is its use with a *nil* payload object and $G = I$. In this case, $M_p$, $C_p$, and $N_p$ are all zero and the equations above reduce to a simple change of coordinates.

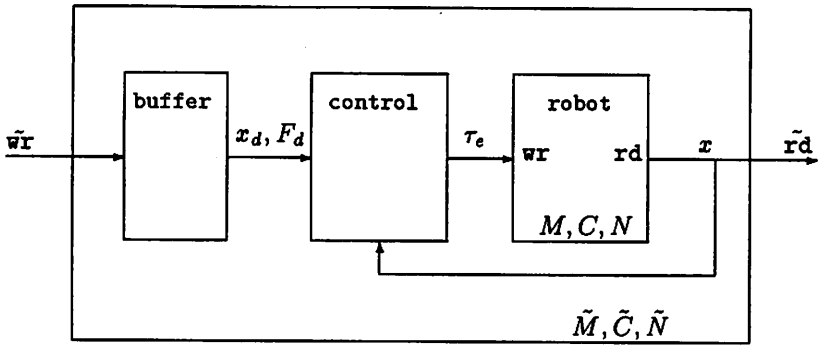


Figure 3.3: Data flow in a typical controlled robot. Information written to the robot is stored in an internal buffer where it can be accessed by the controller. The controller uses this information and the current state of the robot to generate forces which cause it to follow the desired trajectory.

## 3.4 CONTROL primitive

**Synopsis:**

```
CONTROL(robot, controller)
```

The `control` primitive is responsible for assigning a controller to a robot. It is also responsible for creating a new robot with attributes that properly represent the controlled robot. The attributes of the created robot are completely determined by the individual controller. However, the *rd* and *wr* attributes will often be the same for different controllers. Typically the *rd* attribute for a controlled robot will be the same as the *rd* attribute for the underlying robot. That is, the current state of the controlled robot is equivalent to the current state of the uncontrolled robot. A common *wr* attribute for a controlled robot would be a function which saved the desired position, velocity, acceleration and force in a local buffer accessible to our controller. This configuration is shown in Figure 3.3.

The dynamic attributes $\tilde{M}$, $\tilde{C}$ and $\tilde{N}$ are determined by the controller. At one extreme, a controller which compensates for the inertia of the robot would set the dynamic attributes of the controlled robot to zero. This does not imply that the robot is no longer a dynamic object, but rather that controllers at higher levels can ignore the dynamic properties of the robot, since they are being compensated for at a lower level. At the other end of the spectrum, a controller may make no attempt to compensate for the inertia of a robot, in which case it should pass the dynamic attributes on to the next higher level. Controllers which lie in the middle of this range may partially decouple the dynamics of the

manipulator without actually completely compensating for them. To illustrate these concepts we next consider one possible controller class, computed torque. However, many control laws originally formulated in joint space may also be employed since the structure of equation (2.3) has been preserved.

### Computed torque controller

As we mentioned in Chapter 2, the computed torque controller is an exactly linearizing controller which inverts the nonlinearities of a robot to construct a linear system. This linear system has a transfer function equal to the identity map and as a result has no uncompensated dynamics. The proper representation for such a system sets the dynamical attributes $\tilde{M}$, $\tilde{C}$, and $\tilde{N}$ to zero and uses the *rd* and *wr* attributes as described above. We introduce $x_d$ to refer to the buffered desired trajectory.

The control process portion of the controller is responsible for generating input robot forces which cause the robot to follow the desired trajectory (available in $x_d$). Additionally, the controller must determine the "expected" trajectory to be sent to lower level robots. For the computed torque controller we use the resolved acceleration [16] to generate this path. This allows computed torque controllers running at lower levels to properly compensate for nonlinearities and results in a linear error response. The methodology is similar to that used in determining that the dynamic attributes of the output robot should be zero. The control algorithm is implemented by the following equations:

$$
\begin{aligned}
&(x, \dot{x}, \cdot, \cdot) = rd() \\
&\ddot{x}_e = \ddot{x}_d + K_v(\dot{x}_d - \dot{x}) + K_p(x_d - x) \\
&\dot{x}_e = \int_0^t \ddot{x}_e \\
&x_e = \int_0^t \dot{x}_e \\
&F_e = M(q)\ddot{x}_e + C(q, \dot{q})\dot{x} + N(q, \dot{q}) + F_d \\
&wr(x_e, \dot{x}_e, \ddot{x}_e, F_e)
\end{aligned}
$$

where *rd* and *wr* are attributes of the robot which is being controlled.

Note the existence of the $F_d$ term in the calculation of $F_e$. This is placed here to allow higher level controllers to specify not only a trajectory but also an force term to compensate for higher level payloads. In essence, a robot which is being controlled in this manner can be viewed as an ideal force generator which is capable of following an arbitrary path.

The computed torque controller defines two new attributes, $K_p$ and $K_v$, which determine the gains (and hence the convergence properties) of the controller. A variation of the computed torque controller is the feedforward controller, which is obtained by setting $K_p = K_v = 0$. This controller can be used to distribute nonlinear calculations in a hierarchical controller, as we shall see in Chapter 4.

# Chapter 4

# Examples

To make the use of the primitives more concrete we present some examples of a planar hand grasping a box (Figure 2.1) using various control structures. For all of the controllers, we will use the following functions

| | |
|---|---|
| $M_b$ | inertia matrix for the box in Cartesian coordinates |
| $M_l, M_r$ | inertia matrix for the left and right fingers |
| $C_b, C_l, C_r$ | Coriolis/centrifugal vector for box and fingers |
| $f$ | finger kinematics function, $f : (\theta_l, \theta_r) \mapsto (x_l, x_r)$ |
| $g$ | grasp kinematics function, $g : (x_l, x_r) \mapsto x_b$ |
| $J$ | finger Jacobian, $J = \frac{\partial f}{\partial \theta}$ |
| $G$ | grasp map, consistent with $g$ |
| `rd_left`, `rd_right` | read the current joint position and velocity |
| `wr_left`, `wr_right` | generate a desired torque on the joints |

where $\theta_l, \theta_r, x_l, x_r$, and $x_b$ are defined as in Figure 2.1.

## 4.1 High level computed torque control

In this example we combine all systems to obtain a description of the dynamic properties of the overall system in box coordinates. Once this is done we can move the box by directly specifying the desired trajectory for the box. This structure is illustrated in Figure 4.1.

In terms of the primitives that we have defined, we build this structure from the bottom up

```
left  = DEFINE(M_l, C_l, 0, rd_left, wr_left)
right = DEFINE(M_r, C_r, 0, rd_right, wr_right)
fingers = ATTACH(J, I, f, nil, left, right)
```

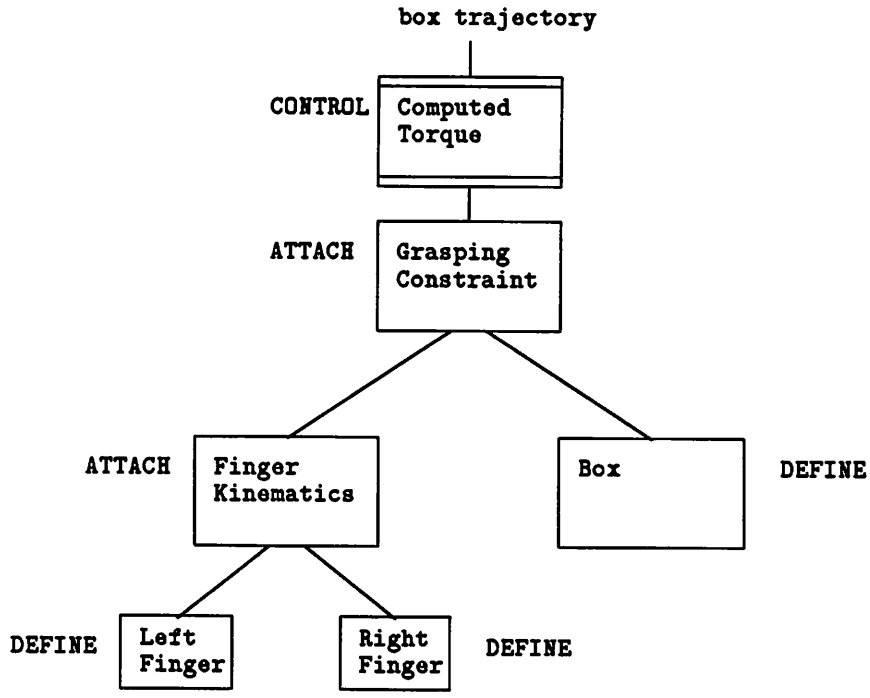


Figure 4.1: High level computed torque. The primitives listed next to the nodes in the graph indicate the primitive that was used to created the node. In this structure all dynamic compensation and error correction occurs at the top of the graph, using a complex dynamic model for the underlying system.

```
box  = DEFINE(M_b, C_b, 0, nil, nil)
hand = ATTACH(I, G, g, box, fingers)

ct_hand = CONTROL(hand, computed-torque)
```

It is useful to consider how the data flows to and from the control law running at the hand level. In the evaluation of $x_b$ and $\dot{x}_b$, the following sequence occurs (through calls to the *rd* attribute):

hand: asks for current state, $x_b$ and $\dot{x}_b$
  finger: ask for current state, $x_f$ and $\dot{x}_f$
    left: read current state, $\theta_l$ and $\dot{\theta}_l$
    right: read current state, $\theta_r$ and $\dot{\theta}_r$
  finger: $x_f, \dot{x}_f \leftarrow f(\theta_l, \theta_r), J(\dot{\theta}_l, \dot{\theta}_r)$
hand: $x_b, \dot{x}_b \leftarrow g(x_f), G^T \dot{x}_f$

Similarly, when we write a set of hand forces using the $wr$ attribute, it causes another chain of events to occur: call sequence is generated

box: generate a box force $F_b$
  hand: generate finger force $GF_b$
    finger: generate joint torques $J^T GF_b$
      left: output torques conjugate to $\theta_l$
      right: output torques conjugate to $\theta_r$

Using the transformations given above it is straightforward to calculate the torque generated by the control law by expanding the structure of Figure 4.1 using the definition of the primitives.

$$\begin{aligned}
\begin{pmatrix} \tau_l \\ \tau_r \end{pmatrix} &= J^T F_{f,d} \\
&= J^T G^+ F_{b,d} \\
&= J^T G^+ \left[ M_h \left( \ddot{x}_{b,d} + K_v \dot{e} + K_p e \right) + C_h \dot{x}_b \right] \\
&\vdots \\
&= J^T G^+ \Big[ \left( M_b + G J^{-T} M_\theta J^{-1} G^T \right) \left( \ddot{x}_{b,d} + K_v \dot{e} + K_p e \right) + \\
&\quad \left( C_b + G J^{-T} C_\theta J^{-1} G^T \right) \dot{x}_b + G J^{-T} M_\theta \frac{d}{dt} \left( J^{-1} G^T \right) \dot{x}_b \Big]
\end{aligned}$$

This control law corresponds exactly to the generalized computed torque control algorithm presented by Li, *et al.* [15], with the omission of internal forces (see Section 5.1).

## 4.2 Low level computed torque control

Another common structure for controlling robots is to convert all trajectories to joint coordinates and perform computed torque at that level. In a crude implementation one might assume the dynamic effects of the box were negligible and construct the following structure shown in Figure 4.2. The primitives used to define this structure are

```
left  = DEFINE(M_l, C_l, 0, rd_left, wr_left)
right = DEFINE(M_r, C_r, 0, rd_right, wr_right)
ct_left  = CONTROL(left, computed-torque)
ct_right = CONTROL(right, computed-torque)

fingers = ATTACH(J, I, f, nil, ct_left, ct_right)
box     = DEFINE(M_b, C_b, 0, nil, nil)
```

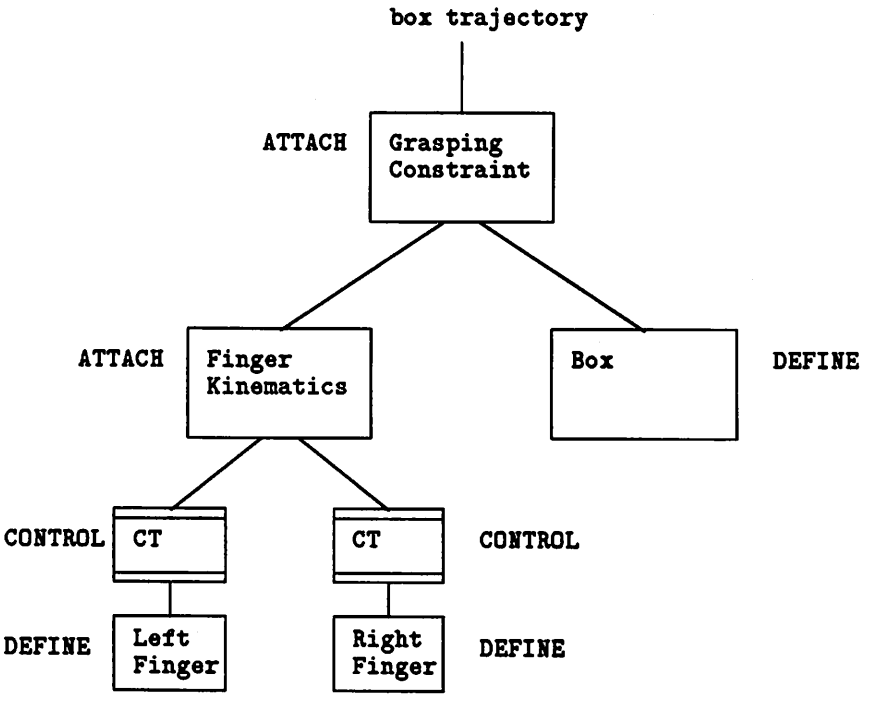


Figure 4.2: low level computed torque. Computed torque controllers are used for the individual fingers to provide trajectory following capability in joint space. Since no controller is positioned above the box, the dynamics of the box are ignored even though the path is given in the box's frame of reference.

```
hand = ATTACH(I, G, g, box, fingers)
```

This controller is provably exponentially stable when the mass of the box is zero. However, this controller does not compensate for the mass of the box. As a result, we expect degraded performance if the mass of the box is large. Experimental results on a system of this form confirm our intuition [17].

## 4.3 Multi-level computed torque/stiffness control

As a final example, we consider a control structure obtained by analogy with biological systems in which controllers to run at several different levels simultaneously (see Figure 4.3). At the lowest level we use simple PD control laws

attached directly to the individual fingers. These PD controllers mimic the stiffness provided by muscle coactivation in a biological system [11]. Additionally, controllers at this level might be used to represent spinal reflex actions. At a somewhat higher level, the fingers are attached and considered as a single unit with relatively complicated dynamic attributes and Cartesian configuration. Here we employ a feedforward controller (computed torque with no error correction) to simplify these dynamic properties, as viewed by higher levels of the brain. With respect to these higher levels, the two fingers appear to be two Cartesian force generators represented as a single composite robot.

Up to this point, the representation and control strategies do not explicitly involve the box, a payload object. These force generators are next attached to the box, yielding a robot with the dynamic properties of the box but capable of motion due to the actuation in the fingers. Finally, we use a computed torque controller at the very highest level to allow us to command motions of the box without worrying about the details of muscle actuation. By this controller we simulate the actions of the cerebellum and brainstem to coordinate motion and correct for errors.

The structure in Figure 4.3 also has interesting properties from a more traditional control viewpoint. The low level PD controllers can be run at high servo rates (due to their simplicity) and allow us to tune the response of the system to reject high frequency disturbances. The Cartesian feedforward controller permits a distribution of the calculation of nonlinear compensation terms at various levels, lending itself to multiprocessor implementation. Finally, using a computed torque controller at the highest level gives the flexibility of performing the controller design in the task space and results in a system with linear error dynamics. Another feature is that if we ignore the additional torques due to the PD terms, the joint torques generated due to an error in the box position are the same as those of the high level computed torque scheme presented earlier.

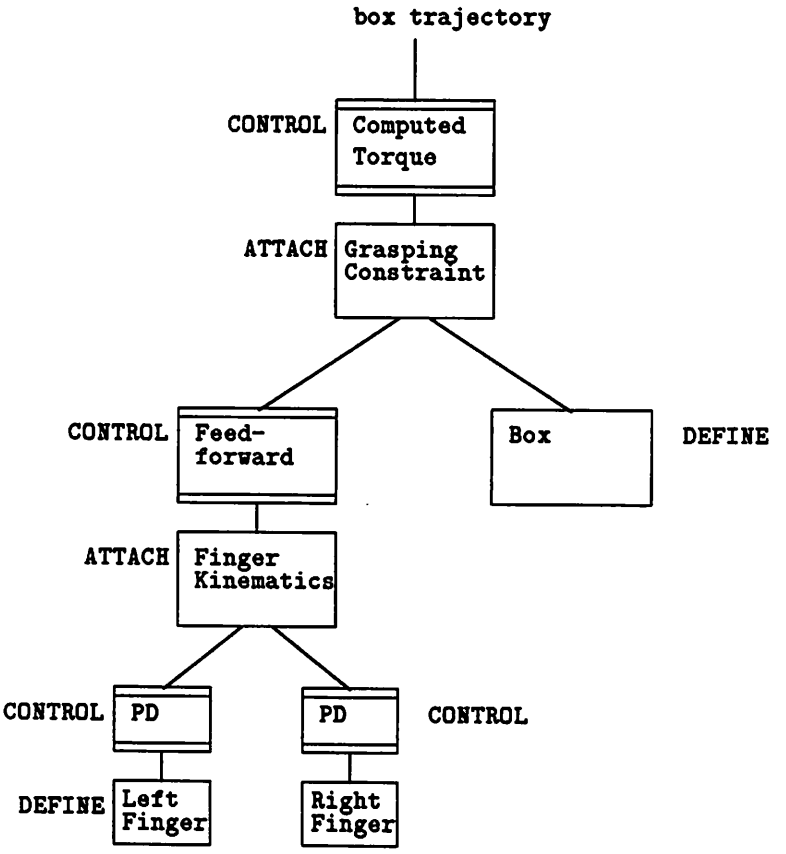


Figure 4.3: multi-level computed torque and stiffness (PD). Controllers are used at each level to provide a distributed control system with biological motivation, desirable control properties, and computational efficiency.

# Chapter 5

# Extensions to the basic primitives

Having presented the primitives for non-redundant robot systems in which we ignore internal forces, we now describe the modifications necessary to include both internal motion and internal forces in the primitives. As before, these extensions are based on the dynamic equations given in Chapter 2 and rely on the fact that the equations of motion of this class of systems can be expressed in a unified way.

Internal motion and force can be thought of as manifestations of redundancies in the manipulator, and both can be used to improve performance. A classical use of redundant motion in robotics is to specify a cost function and use the redundancy of the manipulator to attempt to minimize this cost function. If we extend our definition of the *wr* function so that it takes not only an "external" trajectory, but also an internal trajectory (which might be represented as a cost function or directly as a desired velocity in the internal motion directions) then this internal motion can be propagated down the graph structure. A similar situation occurs with internal or constraint forces.

The matrices $J(q)$ and $G(q)$ in equation (2.2) embody the fundamental properties of the constrained system. We begin by assuming that $J(q)$ and $G(q)$ are both full row rank. The null space of $J(q)$ corresponds to motions which do not affect the configuration of the object, i.e., internal motions. Likewise, the null space of $G(q)$ describes internal forces—the set of forces which cause no motion of the object. A complete trajectory for a robot must specify not only external motion and force for a robot but also the internal motion and force which lie in these subspaces.

## 5.1 Internal forces

To allow internal forces to be specified and controlled, we must first add them to the *rd* and *wr* attributes. This is done by simply adding an extra value to the list of values returned by *rd* and adding an extra argument to *wr*. Thus the *wr* attribute is called as

$$wr(x_e, \dot{x}_e, \ddot{x}_e, F_e, F_i) \tag{5.1}$$

where $F_i$ is the desired internal force.

Internal forces are "created" by the **attach** primitive. The internal force directions for a constraint are represented by an orthonormal matrix $H(\theta)$ whose rows form a basis for the null space of $G(\theta)$. Since any of the daughter robots may itself have an internal force component, the internal force vector for a robot created by attach consists of two pieces: the internal forces created by this constraint and the combined internal forces for the daughter robots. We shall refer to these two components as $F_{i,1}$ and $F_{i,2}$, respectively. The force transformations which describe this relationship are

$$\tau_R = \left(\frac{\tau_{R,e}}{\tau_{R,i}}\right) = \left(\begin{array}{cc|c} J^T G^+ & J^T H^T & 0 \\ 0 & 0 & I \end{array}\right) \left(\begin{array}{c} F_e \\ \hline F_{i,1} \\ F_{i,2} \end{array}\right) \tag{5.2}$$

where $\tau_{R,e}$ is the vector of external forces for the daughter robots and $\tau_{R,i}$ is the vector of internal forces. This equation is analogous to equation (2.4) in Section 2.2. Note that $\tau_{R,i}$ is identical to $F_{i,2}$, thus internal force specifications required by the daughter robots are appended to the internal force specification required due to the constraint. Expanding equation (5.2) we see the appropriate definition for the new *wr* attribute generated by **attach** is

$$\tilde{wr}(x_e, \dot{x}_e, \ddot{x}_e, F_e, F_i) := wr_R(\cdots, J^T G^+ F_e + J^T H^T F_{i,1}, F_{i,2}) \tag{5.3}$$

The inclusion of internal forces in the *rd* attribute is similar. The sensed forces from the robots, $\tau_R$, are simply split into external and internal components and converted to the appropriate internal and external forces for the new robot. This is equivalent to inverting equation (5.2):

$$F = \left(\begin{array}{c} F_e \\ \hline F_{i,1} \\ F_{i,2} \end{array}\right) = \left(\begin{array}{cc|c} GJ^{-T} & & 0 \\ HJ^{-T} & & 0 \\ \hline 0 & & I \end{array}\right) \left(\frac{\tau_{R,e}}{\tau_{R,i}}\right) \tag{5.4}$$

It follows that

$$\tilde{rd}() := (\cdots, GJ^{-T}\tau_{R,e}, \left(\begin{array}{c} HJ^{-T}\tau_{R,e} \\ \tau_{R,i} \end{array}\right)) \tag{5.5}$$

Internal forces are resolved by the **control** primitive. In principle, a controller can specify any number of the internal forces for a robot. Internal forces

which are not resolved by a controller are left as internal forces for the newly defined robot. In practice, controllers will often be placed immediately above the attached robots since internal forces are best interpreted at this level. Unlike external motions and forces, internal forces are not subject to coordinate change and so leaving such forces unresolved causes higher level controllers to use low level coordinates.

## 5.2 Internal motions

Internal motions are also created by the **attach** primitive, this time due to a non-square Jacobian matrix. As before, we must add arguments to the *rd* and *wr* attributes of robots to handle the extra information necessary for motion specification. We only assume that the redundant velocities and accelerations are defined, so we add only those quantities to *rd* and *wr*. Since the notation becomes quite cumbersome, we won't actually define the *rd* and *wr* primitives, but just specify the internal and external motion components.

Given a constraint which contains internal motions, the **attach** primitive must again properly split the motion among the robots attached to the object. Define $K(\theta)$ to be a matrix whose rows span the null space of $J(\theta)$. Then we can rewrite our constraint as

$$\left(\begin{array}{c|c} J & 0 \\ K & 0 \\ \hline 0 & I \end{array}\right)\left(\begin{array}{c} \dot{\theta}_{R,e} \\ \hline \dot{\theta}_{R,i} \end{array}\right) = \left(\begin{array}{cc|c} G^T & 0 & 0 \\ 0 & I & 0 \\ \hline 0 & 0 & I \end{array}\right)\left(\begin{array}{c} \dot{x}_e \\ \dot{x}_{i,1} \\ \hline \dot{x}_{i,2} \end{array}\right) \tag{5.6}$$

Defining $\bar{J}$ and $\bar{G}$ as the extended Jacobian and grasp matrices,

$$\bar{J} = \begin{pmatrix} J \\ K \end{pmatrix} \qquad \bar{G}^T = \begin{pmatrix} G^T & 0 \\ 0 & I \end{pmatrix} \tag{5.7}$$

we see that $\bar{J}$ is full rank and so we can use it to define $\mathcal{A} = \bar{J}^{-1}\bar{G}^T$ in equations (3.4–3.8). This then defines the dynamics attributes created by **attach**. Note that the dimension of the constrained subspace (where internal forces act) is unchanged by this extension.

The input and output attributes are described in a manner similar to those used for internal forces. For *wr* the external component of the motion is given by

$$\dot{\theta}_{R,e} = \mathcal{A}\begin{pmatrix} \dot{x}_e \\ \dot{x}_i \end{pmatrix} \tag{5.8}$$

$$= J^{+}G^T\dot{x}_e + K^T\dot{x}_i \tag{5.9}$$

$\ddot{\theta}_{R,e}$ is defined similarly. $\theta_{R,e}$ is only defined if an inverse kinematic function, $h^{-1}$, is given. Otherwise that information is not passed to the daughter robots.

As before, if the robots themselves have internal motions then these should be split off and passed unchanged to the lower level robots.

The *rd* attribute is defined by projecting robot motions into an object motion component and an internal motion component. That is

$$x_e = h(\theta_{R,e}) \tag{5.10}$$

$$\dot{x}_e = G^+ J \dot{\theta}_{R,e} \tag{5.11}$$

$$\dot{x}_i = \begin{pmatrix} K\dot{\theta}_{R,e} \\ \dot{\theta}_{R,i} \end{pmatrix} \tag{5.12}$$

$\ddot{x}_e$ and $\ddot{x}_i$ are obtained by differentiating the expression for $\dot{x}_e$ and $\dot{x}_i$.

Controllers must also be extended to understand redundant motion. This is fundamentally no different than control of an ordinary manipulator except that position information is not available in redundant directions. Thus the computed torque law would become

$$F = M(q) \begin{pmatrix} \ddot{x}_{e,d} + K_v \dot{e}_e + K_p e_e \\ \ddot{x}_{i,d} + K_v \dot{e}_i \end{pmatrix} + C(q,\dot{q}) \begin{pmatrix} \dot{x}_e \\ \dot{x}_i \end{pmatrix} + N(q,\dot{q}) \tag{5.13}$$

Motion specification for such a control law would be in terms of a position trajectory $x_e(\cdot)$ and a velocity trajectory $\dot{x}_i(\cdot)$. If a controller actually resolves the internal motion (by specifying $\dot{x}_{i,d}(\cdot)$ based on a pseudo inverse calculation for example), then the internal motion will be masked from higher level controllers; otherwise it is passed on.

Control laws commonly use the position of the object as part of the feedback term. This may not always be available for systems with non-integrable constraints (such as grasping with rolling contacts). If the object position cannot be calculated from $\theta$ then we must retrieve it from some other source. One possibility is to use an external sensor which senses $x$ directly, such as a camera or tactile array. The function to "read the sensor" could be assigned to the payload *rd* function and `attach` could use this information to return the payload position when queried. Another possible approach is to integrate the object velocity (which is well defined) to bookkeep the payload position.

Some care must also be taken with the evaluation of dynamic attributes for robots which do not have well defined inverse kinematic functions. There are some robot control laws which use feedforward terms that depend on the desired output trajectory, e.g., $M(x_d)\ddot{x}_d$. The advantage of writing such control laws is that this expression can be evaluated offline, increasing controller bandwidth. This calculation only makes sense if the desired configuration, $q_d$, can be written as a function of $x_d$ and more generally if $q$ can be written as a function of $x$. One solution to this problem is to only evaluate dynamic attributes of a robot at the current configuration. Assuming each robot in the system can determine its own position, these attributes are then well defined. For all the control laws presented in this paper, $M$, $C$ and $N$ are always evaluated at $q$, the current configuration.

# Chapter 6

# Discussion

Working from a physiological motivation we have developed a set of robot description and control primitives consistent with Lagrangian dynamics. Starting from a description of the inertia, sensor, and actuator properties of individual robots, these primitives allow for the construction of a composite constrained motion system with control distributed at all levels. Robots, as dynamical systems, are recursively defined in terms of daughter robots. The resulting hierarchical system can be represented as a tree structure in a graph theoretic formalism, with sensory data fusion occurring as information flows from the leaves of the tree (individual robots and sensors) toward the root, and data expansion as relatively simple motion commands at the root of the tree flow down through contact constraints and kinematics to the individual robot actuators.

One of the major future goals of this research is to implement the primitives presented here on a real system. This requires that efforts be made toward implementing primitives in as efficient fashion as possible. The first implementation choice is deciding when computation should occur. It is possible that the entire set of primitives could be implemented off-line. In this case, a controller-generator would read the primitives and construct suitable code to control the system. A more realistic approach is to split the computation burden more judiciously between on-line and off-line resources. Symbolically calculating the attributes of the low level robots and storing these as precompiled functions might enable a large number of systems to be constructed using a library of daughter robot systems. Although the expressions employed are continuous time, in practice digital computers will be relied upon for discrete time implementations. This raises the issue of whether lower computation rates may be practical for higher level robots/controllers.

In addition to implementation issues, there are still several theoretical issues which we hope to address. We would like to have stability proofs for classes of control hierarchies, e.g. any hierarchy with a computed torque controller at the highest level and only feedforward controllers below it can be shown

to be exponentially stable. There is also no provision in the primitives for dynamics which can not be written in the form of equation (2.1). Adaptive identification and control techniques may be useful in cases where unmodeled dynamics substantially affect system performance.

## Acknowledgements

D.C. Deno was supported by NEI EY05913, the Smith-Kettlewell Eye Research Foundation, and the Rachael C. Atkinson Fellowship Award. K.S.J. Pister, R.M. Murray, and S.S. Sastry were supported in part by the NSF under grant numbers DMC 84-51129, ECS 87-19298 and the Air Force Office of Scientific Research (AFSC) under grant number F49620-87-C0041. R.M. Murray was also supported in part by an IBM Manufacturing Fellowship.

# Appendix A

# Mathematica implementation

As an example of how the primitives migth be implemented, we present an offline version of the primitives written as a Mathematica package [26]. The primitives here do not include the extensions presented in Chapter 5, nor do they attempt to incorporate the control law calculations of the `CONTROL` primitive. The primary purpose of this code is to indicate one way in which the primitives might be implemented.

The first two examples used in Chapter 4 are constructed in `Examples.m`. The syntax is almost identical to that used in the body of the paper, with the exception of an additional argument to `DEFINE` and `ATTACH` which gives the number of degrees of freedom of the robot. This term was added to eliminate the need to check the dimensions of the other arguments (which are passed as *functions* not matrices). Also, we note that the kinematics functions $f$ and $g$ have a rule associated with them that gives the inverse kinematics function. This function is accessed as `RobotInverse[f]` and is defined in `StyxDefinitions.m`.

The primitives are defined in `RobotPrimitives.m`. A robot is represented as a list of rules with head `Robot`. All robots have the basic dynamic and I/O attributes. Additional attributes are used to store values needed by the different primitives (such as $J$, $G$ and $h$ in `ATTACH`). Note also that when evaluating attributes, we use intermediate functions which take a robot object as the first argument. This is redundant for `DEFINE`d and `CONTROL`led robots, but is necessary for `ATTACH`ed robots where we must decide how to partition data based on the size of the children.

The other files listed here are definitions and utility functions which are needed to construct the examples and the primitives. Their definitions are straightforward.

The code listed in this section can be obtained via anonymous ftp from

robotics.berkeley.edu in the directory pub/RobotPrimitives.

## Examples.m

```
(*
 * Examples.m - use RobotPrimitives to define two-fingered robot hand
 *
 * Richard M. Murray
 * April 27, 1990
 *
 *)

<<RobotPrimitives.m             (* definitions of the primitives *)
<<StyxDefinitions.m             (* definitions for two-fingered hand *)

(*
 * Example #1 - High level computed torque
 *
 * Connect everything together and strap a computed torque controller
 * on to the top of the whole thing.
 *
 *)

left = RobotDefine[2, Mleft, Cleft, {0, 0}&, RDleft, WRleft];
right = RobotDefine[2, Mright, Cright, {0, 0}&, RDright, WRright];

fingers = RobotAttach[4, J, IdentityMatrix[4] &, f, nil, {left, right}];

box = RobotDefine[3, Mbox, Cbox, Nbox, nil, nil];
hand = RobotAttach[3, IdentityMatrix[4] &, G, g, box, {fingers}];

HighCThand = RobotControl[hand, ComputedTorque];

(*
 * Example #2 - Low level computed torque
 *
 * Add a computed torque controller just above fingers to simplify
 * calculations
 *
 *)

LowCTfingers = RobotControl[fingers, ComputedTorque];
LowCThand = RobotAttach[3, IdentityMatrix[4] &, G, g, box, {LowCTfingers}];

(* Sample computations - run these interactively
RobotMass[left]                 (* mass matrix at the current position *)
RobotMass[hand]                 (* this one takes a while (2-3 min) *)
RobotCoriolis[fingers,          (* evaluate coriolis at a given point *)
  {x1,y1, x2,y2},
  {dx1,dy1, dx2, dy2}]

RobotWrite[hand,                (* run through the inverse kinematics *)
  {x,y,phi}, nil, nil, nil]
RobotRead[hand]                 (* forward kinematics *)
```

```
RobotMass[HighCThand]          (* this should be zero *)
RobotMass[LowCThand]           (* just the box mass *)
*)
```

## RobotPrimitives.m

```
(*
 * RobotPrimitives - Primitives for Robot Control
 *
 * Richard M. Murray
 * May, 1990
 *
 * This package implements the primitives described in:
 *
 *    D.C. Deno, R.M. Murray, K.S.J. Pister and S.S. Sastry,
 *    "Control Primitives for Robot Systems",
 *    ERL memo #90-???, UC Berkeley, May 1990
 *
 * :Context: RobotPrimitives`
 * :Package Version: 1.0
 * :Mathematica Version: 1.1
 * :History:
 *    Version 1.0 by Richard Murray (UCB), May 1990
 *
 * :Discussion:
 *  A "robot" is just a list of rules that stores the attributes
 *  associated with a robot (mass, I/O, etc). Send comments and bug
 *  reports to murray@united.berkeley.edu.
 *)

BeginPackage["RobotPrimitives`", "Jac`"]

RobotDefine::usage = "RobotDefine[size, M, C, N, rd, wr] - define a new robot."

RobotAttach::usage = "RobotAttach[size, J, G, h, payload, list] - attach
several robots to a payload via a kinematic constraint."

RobotControl::usage = "RobotControl[robot, controller] - attach a controller
to a robot.  Currently the only controller defined is ComputedTorque."

RobotInfo::usage = "RobotInfo[robot] - Print the attributes of a robot."

RobotRead::usage = "RobotRead[robot] - print current state of a robot."

RobotWrite::usage = "RobotWrite[robot, x, dx, ddx, f] - write state information
to a robot.  x = position, dx = velocity, ddx = acceleration, f = force."

RobotMass::usage = "RobotMass[robot, x] - return the mass matrix of a robot
at position x.  If x is nil or not specified, the current position is used."

RobotCoriolis::usage = "RobotCoriolis[robot, x, v] - return the coriolis
matrix of a robot evaluated at a given position and velocity.  If x and
v are not specified, the current values are used."

RobotNonlinear::usage = "RobotNonlinear[robot, x, v] - return the nonlinear
```

```
force vector for a robot.  If x and v are not specified, the current values
are used."

RobotSimplify::usage = "RobotSimplify[expr] - apply some basic simplification
rules to expr.  Supposedly optimized for robot type expressions."

nil::usage = "The nil function is used by several RobotPrimitives routines to
indicate a lack of data."

RobotInverse::usage = "RobotInverse[f] is used to define the inverse of f.  A
typical application is defining the inverse kinematics associated with f."

ComputedTorque::usage = "ComputedTorque - controller for use with RobotControl"

(* Set the default simplification operation to nothing *)
Options[RobotPrimitives] = {simplify -> RobotSimplify};

Begin["Private`"]

(* Redefine the output format to keep listings small *)
(* Use RobotInfo to get a full listing of attributes *)
Format[robot_Robot] := "-RobotObject-";

(*
 * DEFINE - create a new robot given its basic attributes
 *
 * Usage: DEFINE[size, mass, coriolis, nonlinear, rd, wr]
 *
 *)

SetAttributes[RobotDefine, HoldAll];
RobotDefine[size_, mass_, cori_, nonl_, rd_, wr_] :=
    (* Return the attribute list with Robot head *)
    Robot @@ {
      size$ -> size, type$->"Define",
      ReadFcn$ -> defineRead,
      WriteFcn$ -> defineWrite,
      MassFcn$ -> defineMass,
      CoriFcn$ -> defineCori,
      NonlFcn$ -> defineNonl,

      M$->mass, C$->cori, N$->nonl,
      Rd$->rd, Wr$->wr
    };

(* Mass matrix evaluation for a defined robot *)
defineMass[robot_Robot, pos_] :=
  Block[
    {theta},
    (* Get the robot position; if nil is specified, then read it *)
    If [SameQ[pos, nil], theta = (RobotRead[robot])[[1]], theta = pos];

    (* Now evaluate the mass matrix at that position *)
    (getAttribute[robot, M$])[theta]
  ];
```

```
(* Coriolis matrix evaluation for a defined robot *)
defineCori[robot_Robot, pos_, vel_] :=
  Block[
    {theta, dtheta},
    (* Get the robot position; if nil is specified, then read it *)
    If [SameQ[pos, nil],
        {theta, dtheta} = (RobotRead[robot])[[{1,2}]],
        {theta, dtheta} = {pos, vel}];

    (* Now evaluate the mass matrix at that position *)
    (getAttribute[robot, C$])[theta, dtheta]
  ];

(* Nonlinear vector evaluation for a defined robot *)
defineNonl[robot_Robot, pos_, vel_] :=
  Block[
    {theta, dtheta},
    (* Get the robot position; if nil is specified, then read it *)
    If [SameQ[pos, nil],
        {theta, dtheta} = (RobotRead[robot])[[{1,2}]],
        {theta, dtheta} = {pos, vel}];

    (* Now evaluate the mass matrix at that position *)
    (getAttribute[robot, N$])[theta, dtheta]
  ];

(* Read function for a defined robot *)
defineRead[robot_Robot] := (getAttribute[robot, Rd$])[];

(* Write function for a defined robot *)
defineWrite[robot_Robot, x_, dx_, ddx_, f_] :=
  (getAttribute[robot, Wr$])[x, dx, ddx, f];

(*
 * ATTACH - attach two or more objects together with a kinematic constraint
 *
 * Usage: ATTACH[size, J, G, h, payload, robot-list]
 *
 *)

SetAttributes[RobotAttach, HoldAll];
RobotAttach[size_, J_, G_, h_, payload_, robotlist_] :=
    Robot @@ {
      size$ -> size,
      type$ -> "Attach",
      MassFcn$ -> attachMass,
      CoriFcn$ -> attachCori,
      NonlFcn$ -> attachNonl,
      ReadFcn$ -> attachRead,
      WriteFcn$ -> attachWrite,
      G$ -> G, J$ -> J, h$ -> h,
      RobotList$ :> robotlist,
      Payload$ :> payload}
```

```
(* Mass matrix evaluation for an attached robot *)
attachMass[robot_, q_] :=
  Block[
    {daughter = getAttribute[robot, RobotList$],
     simple = (simplify /. Options[RobotPrimitives] /. {simplify->Identity}),
     x, theta, thpart, Afn, Mr,Ma, i},

    (* Get the position of the robot *)
    If[SameQ[q, nil],
      (* Figure out the current object location and velocity transformation *)
      {Afn, x} = Take[ attachForward[robot], 2 ];

      (* Use current location (nil) for mass matrix evaluation *)
      thpart = Array[nil &, Length[daughter]],

    (* Else *)
      (* Use the inverse kinematics to find the robot location (slow) *)
      {Afn, theta} = Take[ attachReverse[robot, q], 2 ];
      thpart = partition[theta, getAttribute[daughter, size$]];
      x = q;
    ];

    (* Stack the mass matrices on top of each other *)
    Mr = blockDiagonal[
      Table[ RobotMass[daughter[[i]], thpart[[i]] ], {i, Length[daughter]} ]
    ];

    (* Now transform the mass matrix into object coordinates *)
    Ma = simple[ Transpose[Afn] . Mr . Afn ];

    (* Add in the payload mass if there is a payload *)
    If[Not[ SameQ[getAttribute[robot, Payload$], nil] ],
        Ma += RobotMass[getAttribute[robot, Payload$], x]];

    (* Return the mass matrix *)
    Ma
  ]

(* Coriolis matrix evaluation for an attached robot *)
attachCori[robot_, q_, v_] :=
  Block[
    {daughter = getAttribute[robot, RobotList$],
     simple = (simplify /. Options[RobotPrimitives] /. {simplify->Identity}),
     x, dx, theta, dtheta, thpart, dthpart, Afn, Cr, Co, i},

    (* Get the position of the robot *)
    If[SameQ[q, nil],
      (* Figure out the current object location and velocity transformation *)
      {Afn, x, dx} = Take[ attachForward[robot], 3 ];

      (* Use current location (nil) for evaluation *)
      dthpart = thpart = Array[nil &, Length[daughter]],

    (* Else *)
      (* Use the inverse kinematics to find the robot location (slow) *)
```

```
      {Afn, theta, dtheta} = Take[ attachReverse[robot, q, v], 3 ];
      thpart = partition[theta, getAttribute[daughter, size$]];
      dthpart = partition[dtheta, getAttribute[daughter, size$]];
      x = q; dx = v;
    ];

    (* Stack the coriolis matrices on top of each other *)
    Cr = blockDiagonal[ Table[
      RobotCoriolis[daughter[[i]], thpart[[i]], dthpart[[i]] ],
      {i, Length[daughter]} ] ];

    (* Now transform the matrix into object coordinates *)
    Co = simple[ Transpose[Afn] . Cr . Afn ];

    (* Add in the payload mass if there is a payload *)
    If[Not[ SameQ[getAttribute[robot, Payload$], nil] ],
        Co += RobotCoriolis[getAttribute[robot, Payload$], x]];

    (* Return the Coriolis matrix *)
    Co
  ]

(* Nonlinear vector evaluation for an attached robot *)
attachNonl[robot_, q_, v_] :=
  Block[
    {daughter = getAttribute[robot, RobotList$],
     simple = (simplify /. Options[RobotPrimitives] /. {simplify->Identity}),
     x, dx, theta, dtheta, thpart, dthpart, Afn, Cr, Co, i},

    (* Get the position of the robot *)
    If[SameQ[q, nil],
      (* Figure out the current object location and velocity transformation *)
      {Afn, x, dx} = Take[ attachForward[robot], 3 ];

      (* Use current location (nil) for evaluation *)
      dthpart = thpart = Array[nil &, Length[daughter]],

    (* Else *)
      (* Use the inverse kinematics to find the robot location (slow) *)
      {Afn, theta, dtheta} = Take[ attachReverse[robot, q, v], 3 ];
      thpart = partition[theta, getAttribute[daughter, size$]];
      dthpart = partition[dtheta, getAttribute[daughter, size$]];
      x = q; dx = v;
    ];

    (* Stack the nonlinear matrices on top of each other *)
    Nr = stackVector[ Table[
      RobotNonlinear[daughter[[i]], thpart[[i]], dthpart[[i]] ],
      {i, Length[daughter]} ] ];

    (* Now transform the nonlinear into object coordinates *)
    Nl = simple[ Transpose[Afn] . Nr ];

    (* Add in the payload mass if there is a payload *)
    If[Not[ SameQ[getAttribute[robot, Payload$], nil] ],
```

```
        Nl += RobotNonlinear[getAttribute[robot, Payload$], x, dx]];

    (* Return the nonlinear vector *)
    Nl
  ]

(* Read routine - just solve the forward kinematics *)
attachRead[robot_Robot] := Take[attachForward[robot], -4];

(* Write routine - solve inverse kinematics and split up solution *)
attachWrite[robot_Robot, x_, dx_, ddx_, force_] :=
  Block[
    {daughter = getAttribute[robot, RobotList$],
     theta, dtheta, ddtheta, tau,
     thpart, dthpart, ddthpart, taupart},

    (* Figure out the desired information for the daughter robots *)
    {theta, dtheta, ddtheta, tau} =
      Take[ attachReverse[robot, x,dx,ddx,force], -4];

    (* Partition the data *)
    thpart =   partition[theta, getAttribute[daughter, size$]];
    dthpart =  partition[dtheta, getAttribute[daughter, size$]];
    ddthpart = partition[ddtheta, getAttribute[daughter, size$]];
    taupart =  partition[tau, getAttribute[daughter, size$]];

    (* Write to the daughter robots *)
    For[i = 1, i <= Length[daughter], ++i,
      RobotWrite[daughter[[i]],
        thpart[[i]], dthpart[[i]], ddthpart[[i]], taupart[[i]]]
    ];
  ];


(* Forward kinematics for an attached robot *)
attachForward[robot_Robot] :=
  Block[
    {daughter = getAttribute[robot, RobotList$],
     simple = (simplify /. Options[RobotPrimitives] /. {simplify->Identity}),
     states, theta, dtheta, ddtheta, tau,
     x, dx, ddx, force, Gfn, Jfn, Afn, Apl, Adt},

    (* Get the states of the child robots *)
    states = Array[(RobotRead[daughter[[#1]]]) &, Length[daughter]];

    (* Now extract out the various pieces *)
    theta = Flatten[ Array[ (states[[#1,1]]) &, Length[daughter] ] ];
    dtheta = Flatten[ Array[ (states[[#1,2]]) &, Length[daughter] ] ];
    ddtheta = Flatten[ Array[ (states[[#1,3]]) &, Length[daughter] ] ];
    tau = Flatten[ Array[ (states[[#1,4]]) &, Length[daughter] ] ];

    (* Use the forward kinematics to find the object location *)
    x  = (getAttribute[robot, h$])[theta];

    (* Figure out the various transformations needed for the velocity *)
```

```
    Gfn = getAttribute[robot, G$];
    Jfn = getAttribute[robot, J$];
    Afn = simple[ Inverse[Jfn[theta]] ] . Transpose[Gfn[x]];
    Apl = simple[ Transpose[pseudoInverse[Gfn[x]]] ] . Jfn[theta];

    (* Now calculate the object velocity *)
    dx =  Apl . dtheta;

    (* Object acceleration requires that we differentiate Apl *)
    Adt = simple[
      directDeriv[ Transpose[pseudoInverse[Gfn[#1] ]]&, x, dx] . Jfn[theta] +
      Transpose[pseudoInverse[Gfn[x]]] . directDeriv[Jfn, theta, dtheta]
    ];
    ddx = Apl . ddtheta + Adt . dtheta;
    force = Transpose[Afn] . tau;

    {Afn, x, dx, ddx, force}
  ];

(* Reverse kinematics for an attached robot *)
attachReverse[robot_Robot, x_, dx_:nil, ddx_:nil, force_:nil] :=
  Block[
    {daughter = getAttribute[robot, RobotList$],
     simple = (simplify /. Options[RobotPrimitives] /. {simplify->Identity}),
     theta, dtheta = nil, ddtheta = nil, tau = nil,
     Gfn, Jfn, Afn, Apl},

    theta = RobotInverse[getAttribute[robot, h$]][x];

    Gfn = getAttribute[robot, G$];
    Jfn = getAttribute[robot, J$];
    Afn = simple[ Inverse[Jfn[theta]] ] . Transpose[Gfn[x]];

    (* Figure out the robot velocity *)
    If[!SameQ[dx, nil], dtheta = Afn . dx];

    (* Figure out the robot acceleration *)
    If[!SameQ[ddx, nil],
      (* Object acceleration requires that we differentiate Afn *)
      Adt = simple[
        directDeriv[ Inverse[Jfn[#1]]&, theta, dtheta] . Tranpose[Gfn[x]] +
        Inverse[Jfn[theta]] . directDeriv[ Transpose[Gfn[#1]]&, x, dx]
      ];
      ddtheta  = Afn . ddx + Adt . dx;
    ];

    (* Figure out the robot torques *)
    If[!SameQ[force, nil],
      Apl = simple[ Transpose[pseudoInverse[Gfn[x]]] ] . Jfn[theta];
      tau = Transpose[Apl] . force];

    (* Send pack a list of what we know *)
    {Afn, theta, dtheta, ddtheta, tau}
  ];
```

```
(*
 * CONTROL
 *
 * Usage:  RobotControl[robot, controller]
 *
 *)

RobotControl[robot_Robot, controller_Symbol] := controller[robot];

(* Computed torque control law *)
ComputedTorque[robot_Robot] :=
    Robot @@ {
      size$ -> getAttribute[robot, size$],
      type$->"ComputedTorque",
      MassFcn$ -> zeroMass,
      CoriFcn$ -> zeroCori,
      NonlFcn$ -> zeroNonl,
      ReadFcn$ -> controlRead,
      WriteFcn$ -> controlWrite,
      desired$ -> {nil, nil, nil, nil},
      child$ :> robot
    }

(* Dynamic attributes for computed torque controlled robots *)
zeroMass[robot_, pos_]   := 0 IdentityMatrix[getAttribute[robot, size$]];
zeroCori[robot_, q_, v_] := 0 IdentityMatrix[getAttribute[robot, size$]];
zeroNonl[robot_, q_, v_] := 0 Range[getAttribute[robot, size$]];

(* Read function for controlled robots *)
controlRead[robot_Robot] := RobotRead[getAttribute[robot, child$]];

(* Write function for controlled robots *)
controlWrite[robot_Robot, x_, dx_, ddx_, force_] :=
  (* Not implemented *)
  Null;

(*
 * RobotSimplify
 *
 * This command attempts to use simplification rules which are fine
 * tuned to a robot.  For now we just built in simplification rules.
 *
 * TrigExpand is too brittle in version 1.1 to be used * reliably, so
 * we are forced to omit it, even though it is the obvious thing to use.
 *)

RobotSimplify[expr_] := Together[expr]

(*
 * Functions for returning various attributes of a robot
 *
 * RobotMass            return the mass matrix of a robot
 * RobotCori            return the coriolis/centrifugal matrix
 * RobotNonl            return gravity/friction vector
 * RobotRead read the current state of a robot
```

```
 * RobotWrite write the desired state of a robot
 * RobotInfo            return general information about a robot
 *
 *)

SetAttributes[getAttribute, Listable]
getAttribute[robot_Robot, attr_Symbol] := (attr /. (List @@ robot))

RobotMass[robot_Robot] := RobotMass[robot, nil];
RobotMass[robot_Robot, x_] := getAttribute[robot, MassFcn$][robot, x];

RobotCoriolis[robot_Robot] := RobotCoriolis[robot, nil, nil];
RobotCoriolis[r_Robot, x_, v_] := getAttribute[r, CoriFcn$][r,x,v];

RobotNonlinear[robot_Robot] := RobotNonlinear[robot, nil, nil];
RobotNonlinear[r_Robot, x_, v_] := getAttribute[r, NonlFcn$][r,x,v];

RobotRead[robot_Robot] := (getAttribute[robot, ReadFcn$])[robot];

RobotWrite[robot_Robot, x_, dx_, ddx_, f_] :=
  (getAttribute[robot, WriteFcn$])[robot, x, dx, ddx, f];

(* List the information contained in a robot *)
(* Replace the head (Robot) with List to bypass formatting *)
RobotInfo[robot_Robot] := List @@ robot;

(*
 * Utility functions used by the primitives
 *
 *)

(* Partition a list into a bunch of sublists *)
partition[list_List, size_List] :=
  Block[
    {offset = 0, index, part = {}},
    For[index = 1, index <= Length[size], ++index,
      part = Join[part, {list[[ offset + Range[size[[index]]] ]]} ];
      offset += size[[index]];
    ];
    part
  ]

(* Crufty implementation of block diagonalization *)
blockDiagonal[list_List] :=
  Block[
    {mat, index, row, rowoff = 0, col, coloff = 0},
    mat = Array[0 &, Plus @@ Map[Dimensions, list]];
    For[index = 1, index <= Length[list], ++index,
      For[row = 1, row <= Dimensions[list[[index]]][[1]], ++row,
        For[col = 1, col <= Dimensions[list[[index]]][[2]], ++col,
          mat[[row+rowoff, col+coloff]] = list[[index]][[row, col]]
        ];
      ];
      {rowoff, coloff} += Dimensions[list[[index]]];
    ];
```

```
    mat
  ];

(* Stack vectors on top of each other *)
stackVector[list_List] := Flatten[list, 1]

(* Quick and dirty Moore-Penrose inverse (assumes m is full rank) *)
pseudoInverse[m_] := Transpose[m] . Inverse[m . Transpose[m]];

(* Take the derectional derivative of a function at x in the v direction *)
directDeriv[f_, x_, v_] :=
  Block[
    {args, i},
    args = Table[Unique[], {Length[x]}];
    Sum[
      v[[i]] *
      Map[ D[#1, args[[i]]]&, f[args], {TensorRank[f[args]]} ]
      /. listRule[args, x],
      {i, Length[x]}
    ]
  ];

(* Listable version of Rule *)
SetAttributes[listRule, Listable];
listRule[lhs_, rhs_] := Rule[lhs, rhs];

End[]                          (* end of private context *)
EndPackage[]                   (* end of RobotPrimitives context *)
```

## RobotDynamics.m

```
(* RobotDynamics.m - functions for generating dynamic eqs for robots *)
(* RMM 2/16/90 *)

BeginPackage["RobotDynamics`", "Jac`"];

(* Calculate the Coriolis matrix from the mass matrix *)
Coriolis[M_, Q_, W_] :=
  Block[
    {m, Dm, Co, x, y},

    (* Define a functional form of the mass tensor *)
    m[q_, x_, y_] := x.M.y;

    (* Coriolis and centrifugal forces *)
    Dm[q_, x_, y_, z_] := Jac[m[q,x,y], q].z;
    Co[q_, x_, y_, z_] := (Dm[q, x,z,y] + Dm[q, y,z,x] - Dm[q, x,y,z]) / 2;

    (* Return the coriolis matrix *)
    { {Co[Q, W, {1,0},{1,0}], Co[Q, W, {0,1},{1,0}]},
      {Co[Q, W, {1,0},{0,1}], Co[Q, W, {0,1},{0,1}]} }
  ];

(* Calculate the inertia matrix from the kinetic energy *)
MassFromKinetic[K_, v_] := Jac[Jac[K, v], v];
```

```
EndPackage[];
```

## StyxDefinitions.m

```
(*
 * StyxParameters.m - parameters for two-fingered hand
 *
 * Richard M. Murray
 * April 27, 1989
 *
 * This file defines the dynamic attributes for two fingers and a box.
 * The following symbols are used:
 *
 * Ml1, Ml2, Mr1, Mr2 Finger masses (point mass; end of link)
 * Ll1, Ll2, Lr1, Lr2 Finger lengths
 * Mb, Jb box mass and inertia
 * r box radius
 *
 * The inverse kinematics for the two link are from Craig's book, with more
 * or less arbitrary determination of elbow up/down configurations.
 *)

<<RobotDynamics.m (* utilities for robot dynamics *)
<<Jac.m (* definitions for Jacobian *)

(* Mass matrix for a 2 link manipulator with point masses at the link tips *)
TwoLinkMass[theta_, {m1_, m2_}, {l1_, l2_}] :=
  {{ m1 l1^2 + m2 l2^2, m2 l1 l2 Cos[theta[[1]] - theta[[2]]] },
   { m2 l1 l2 Cos[theta[[1]] - theta[[2]]], m2 l2^2}}

TwoLinkKinematics[{th1_, th2_}, {l1_, l2_}] :=
  {l1 Cos[th1] + l2 Cos[th2], l1 Sin[th1] + l2 Sin[th2]}

TwoLinkInverseKin[{x_, y_}, {l1_, l2_}, soln_] :=
  Block[
    {c2, s2, phi2},
    Print["solving two link inverse kinematics"];

    (* First solve for the outer joint angle *)
    c2 = (x^2 + y^2 - l1^2 -l2^2) / (2 l1 l2);
    s2 = If[soln == Up, Sqrt[1 - c2^2], -Sqrt[1 - c2^2]];
    phi2 = ArcTan[c2, s2];

    (* Now solve for the inner joint angle *)
    phi1 = ArcTan[x, y] - ArcTan[l1 + l2 c2, l2 s2];

    (* Return absolute coordinates *)
    {phi1, phi1 + phi2}
  ];

(* Box dynamic attributes *)
Mbox[pos_] :=  DiagonalMatrix[{Mb, Mb, Jb}];
Cbox[pos_, vel_] :=  DiagonalMatrix[{0, 0, 0}];
Nbox[pos_, vel_] := {0, 0, 0}
```

```
(* Left finger attributes *)
Fleft[{th1_, th2_}] =     TwoLinkKinematics[{th1, th2}, {L11, L12}];
RobotInverse[Fleft][{x_,y_}] :=   TwoLinkInverseKin[{x, y}, {L11, L12}, Up];
Mleft[{th1_, th2_}] =      TwoLinkMass[{th1, th2}, {M11, M12}, {L11, L12}];
Cleft[{th1_, th2_}, {dth1_, dth2_}] =
  Coriolis[Mleft[{th1, th2}], {th1, th2}, {dth1, dth2}];

(* For read and write, just return some symbols and write to stdout *)
RDleft[] := {{th1, th2}, {dth1, dth2}, {ddth1, ddth2}, {tau1, tau2}};
WRleft[th_, dth_, ddth_, tau_] :=
  Block[
    {},
    Print["Left:"];
    Print["theta: ",   Short[th, 10]];
    Print["dtheta: ",  Short[dth, 10]];
    Print["ddtheta: ", Short[ddth, 10]];
    Print["tau: ",     Short[tau, 10]];
  ];

(* Right finger attributes *)
Fright[{th1_, th2_}] =     TwoLinkKinematics[{th1, th2}, {Lr1, Lr2}];
RobotInverse[Fright][{x_,y_}] :=  TwoLinkInverseKin[{x, y}, {L11, L12}, Up];
Mright[{th1_, th2_}] =      TwoLinkMass[{th1, th2}, {Mr1, Mr2}, {Lr1, Lr2}];
Cright[{th1_, th2_}, {dth1_, dth2_}] =
  Coriolis[Mright[{th1, th2}], {th1, th2}, {dth1, dth2}];
RDright[] := {{th3, th4}, {dth3, dth4}, {ddth3, ddth4}, {tau3, tau4}};
WRright[th_, dth_, ddth_, tau_] :=
  Block[
    {},
    Print["Right:"];
    Print["theta: ",   Short[th, 10]];
    Print["dtheta: ",  Short[dth, 10]];
    Print["ddtheta: ", Short[ddth, 10]];
    Print["tau: ",     Short[tau, 10]];
  ];

(* Finger kinematics - combines left and right *)
f[{th1_,th2_,th3_,th4_}] = Join[ Fleft[{th1, th2}], Fright[{th3, th4}] ];
RobotInverse[f][{x1_, y1_, x2_, y2_}] :=
  Join[ RobotInverse[Fleft][{x1, y1}], RobotInverse[Fright][{x2, y2}] ];
J[{th1_,th2_,th3_,th4_}] = Jac[f[{th1, th2, th3, th4}], {th1,th2,th3,th4}];

(* Grasp kinematics - for attach box to fingers *)
g[{xl_, yl_, xr_, yr_}] := {(xl+xr)/2, (yl+yr)/2, ArcTan[xl-xr, yl-yr]};
RobotInverse[g][{x_, y_, psi_}] :=
  {x-r Cos[psi], x-r Sin[psi], x+r Cos[psi], x+r Sin[psi]}
G[{x_, y_, psi_}] :=
  {{1,0,1,0}, {0,1,0,1}, {r Sin[psi], -r Cos[psi], -r Sin[psi], r Cos[psi]}};
```

## Jac.m

```
(*
 * Jac.m - jacobians and Lie derivatives
 *
```

```
 * John Hauser
 * 1989
 *
 *)

BeginPackage["Jac`"]

Jac::usage = "Jac[f,x] computes the derivative of f with respect to x.";
Lie::usage = "Lie[f,g,x] computes the Lie bracket of f and g wrt x.";
LieD::usage = "LieD[f,h,x] compute the Lie derivitive of h wrt f.";
Adj::usage = "Adj[v1,v2,x,k] calculate the kth bracket of v2 wrt v1.";

Begin["Private`"]

Jac[f_, x_] :=
        If[
                VectorQ[ f ],
                Table[ D[ f[[i]], x[[j]] ], {i, Length[f]} , {j, Length[x]} ],
                Table[ D[ f, x[[j]] ], {j, Length[x]} ]
          ]

Lie[v1_, v2_, x_] := Jac[v2,x].v1 - Jac[v1,x].v2

Adj[v1_, v2_, x_, k_] :=
        If[ k==0, v2, Lie[ v1, Adj[ v1, v2, x, k-1 ], x] ]

LieD[f_, h_, x_] := Jac[h,x].f

End[]
EndPackage[]
```

# Bibliography


[1] J. Baillieul. Kinematic programming alternatives for redundant manipulators. In *IEEE International Conference on Robotics and Automation*, pages 722–728, 1985.

[2] A. K. Bejczy. Robot arm dynamics and control. Technical Report 33-699, Jet Propulsion Laboratory, 1974.

[3] Rodney A. Brooks. A hardware retargetable distributed layered architecture for mobile robot control. In *IEEE International Conference on Robotics and Automation*, pages 106–110, 1987.

[4] Dayton Clark. HIC: An operating system for hierarchies of servo loops. In *IEEE International Conference on Robotics and Automation*, pages 1004–1009, 1989.

[5] A. Cole, J. Hauser, and S. Sastry. Kinematics and control of multifingered hands with rolling contact. In *IEEE International Conference on Robotics and Automation*, pages 228–233, 1988.

[6] John J. Craig. *Introduction to Robotics: Mechanics and Control.* Addison-Wesley, second edition, 1989.

[7] Victor Eng. *The MDL Programmer's Reference Manual.* Harvard Robotics Laboratory, 1988. First Revision.

[8] K.S. Fu, Raphael C. Gonzalez, and C. S. George Lee. *Robotics: control, sensing, vision, and intelligence.* McGraw-Hill, Inc., 1987.

[9] Claude Ghez. Introduction to the motor system. In Eric R. Kandel and James H. Schwartz, editors, *Principles of Neural Science, 2nd Ed.*, chapter 33. Elsevier, 1985.

[10] G. Hinton. Some computational solutions to Bernstein's problems. In H. T. A. Whiting, editor, *Human Motor Actions — Bernstein Reassessed*, chapter 4b. Elsevier Science Publishers B.V., 1984.

[11] N. Hogan. Stable execution of contact tasks using impedance control. In *IEEE International Conference on Robotics and Automation*, 1987.

[12] O. Khatib. A unified approach for motion and force control of robot manipulators: The operational space formulation. *IEEE RA Journal*, RA-3(1):43–53, February 1987.

[13] O. Khatib. Augmented object and reduced effective inertia in robot systems. In *American Control Conference*, 1988.

[14] D. Koditschek. Natural motion for robot arms. In *IEEE Control and Decision Conference*, pages 733–735, 1984.

[15] Z. Li, P. Hsu, and S. Sastry. On kinematics and control of multifingered hands. In *IEEE International Conference on Robotics and Automation*, pages 384–389, 1988.

[16] J. Y. S. Luh, M. W. Walker, and R. P. Paul. Resolved acceleration control of mechanical manipulators. *IEEE AC Transactions*, AC-25, 1980.

[17] R. Murray and S. Sastry. Control experiments in planar manipulation and grasping. In *IEEE International Conference on Robotics and Automation*, 1989.

[18] R. Murray and S. Sastry. Grasping and manipulation using multifingered robot hands. In *Mathematical Question in Robotics, Lecture Notes*, Providence, RI, 1990. American Mathematical Society.

[19] R. P. Paul and V. Hayward. Robot control and computer languages. In *Theory and Practice of Robots and Manipulators*, 1984. Proceedings of RoManSy '84: The Fifth CISM-IFTOMM Symposium.

[20] Richard P. Paul. *Robot manipulators: mathematics, programming, and control.* MIT Press, 1981.

[21] R. M. Rosenberg. *Analytical Dynamics of Discrete Systems.* Plenum Press, New York, 1977.

[22] N. Sadegh. *Adaptive Control of Mechanical Manipulators: Stability and Robustness Analysis.* PhD thesis, Department of Mechanical Engineering, University of California, Berkeley, California, 1987.

[23] N. R. Sandell, Jr., P. Varaiya, M. Athans, and M. G. Safonov. Survey of decentralized control methods for large scale systems. *IEEE Transactions on Automatic Control*, AC-23:108–128, 1978.

[24] J. E. Slotine and W. Li. On the adaptive control of robot manipulators. *International Journal of Robotics and Control*, 6:49–59, 1987.

[25] M. W. Spong and M. Vidyasagar. *Dynamics and Control of Robot Manipulators.* John Wiley, 1989.

[26] Stephen Wolfram. *Mathematica: A System for Doing Mathematics by Computer.* Addison-Wesley, 1989.