

A DYNAMIC MODEL OF LOCOMOTION FOR
COMPUTER ANIMATION

BY

MICHAEL ALLEN MCKENNA

Bachelor of Science
Massachusetts Institute of Technology
Cambridge, MA. 1987

Submitted to the Media Arts and Sciences Section
in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE


at the
Massachusetts Institute of Technology
February 1990

© Massachusetts Institute of Technology 1990, all rights reserved


Signature of the Author _____

 Michael A. McKenna
Media Arts and Sciences Section
January 12, 1990

Certified by _____

 David L. Zeltzer
Thesis Supervisor
Associate Professor of Computer Graphics

Accepted by _____

 Stephen A. Benton
Chairman, Departmental Committee on Graduate Studies

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

FEB 26 1990

LIBRARIES

Botch



Room 14-0551
77 Massachusetts Avenue
Cambridge, MA 02139
Ph: 617.253.2800
Email: docs@mit.edu
<http://libraries.mit.edu/docs>

DISCLAIMER NOTICE

The accompanying media item for this thesis is available in the MIT Libraries or Institute Archives.

Thank you.

**A DYNAMIC MODEL OF LOCOMOTION FOR
COMPUTER ANIMATION**

BY

MICHAEL ALLEN MCKENNA

Submitted to the Media Arts and Sciences Section
on January 12, 1990, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

A computational model of legged locomotion was developed in which all motions are physically-based. A dynamic simulator for articulated figures forms the basis of all motion in the system, where forces are applied to bodies, and their accelerations are then computed. A gait controller coordinates the activity of stepping and stance, and is based on biological mechanisms found in vertebrates and invertebrates. Dynamic motor programs, based on spring and damper combinations, provide the forces required to move the limbs in order to step and to propel the body forward. The system successfully computes the motions of a simulated six-legged insect negotiating level and uneven terrain.

A VHS videotape containing sample animations accompanies this thesis.

Thesis Supervisor: David L. Zeltzer
Title: Associate Professor of Computer Graphics

This work was supported in part by the National Science Foundation (Grant IRI-8712772), and equipment grants from Hewlett-Packard Co., Gould Electronics, Inc., and Apple Computer, Inc.

Acknowledgements

I would like to first thank Dave Small, the person I can rely on more than any other.

And thanks to my Mom, for sending me to MIT, and for so much encouragement.

Thanks to my advisor, David Zeltzer, for introducing me to so many new concepts.

And of course, thanks to all the Snakepit guys for stimulating conversation, and much needed diversion.

And thanks to all the people of the Media Lab, who made the roach possible.

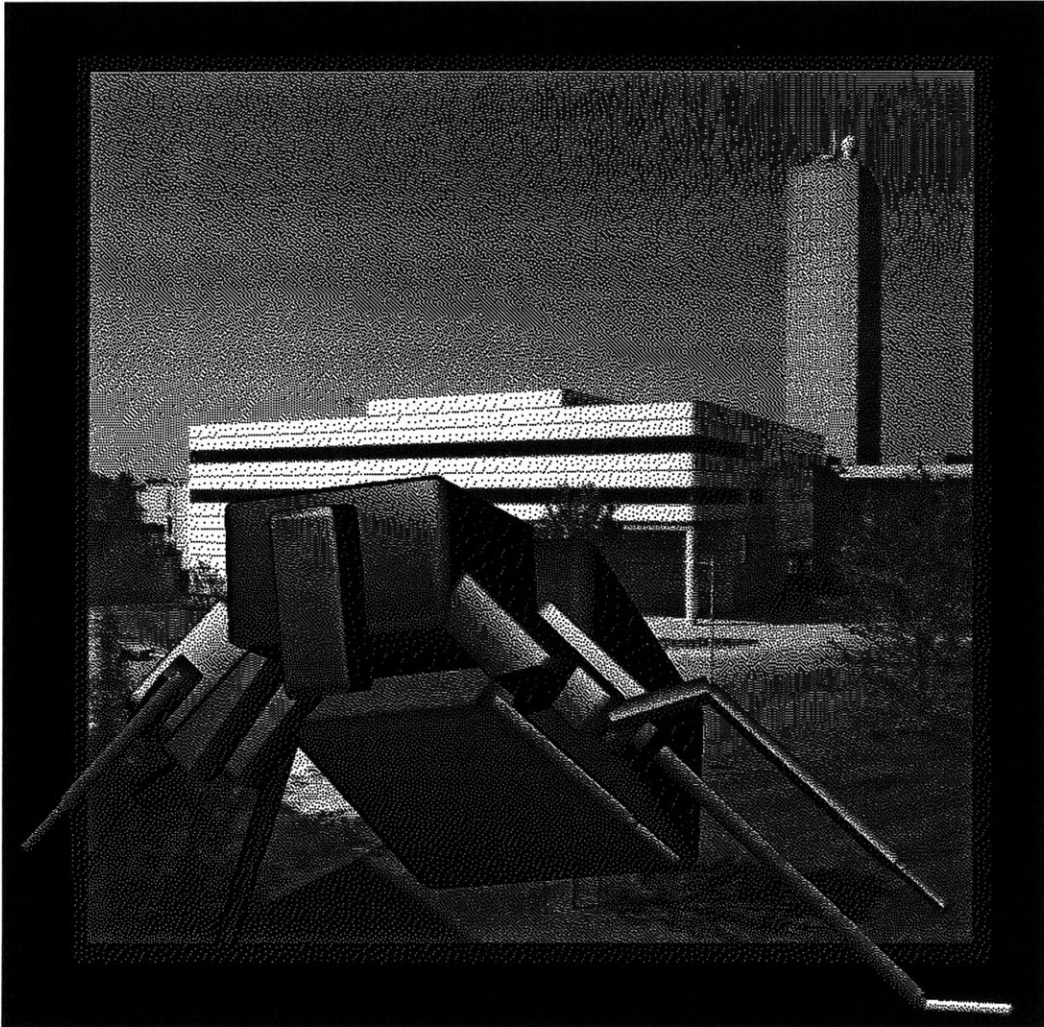


Table of Contents

Chapter 1: Introduction.	5
Chapter 2: Background and Related Work.	8
2.1: Dynamic Simulation.	8
2.2: Gait Generation.	14
2.3: Motor Control.	18
2.4: Walking Machines and Simulations.	25
Chapter 3: Approach.	27
3.1: Dynamic Simulation.	28
3.2: Gait Control.	31
3.3: Motor Programs.	33
3.4: Roach Description.	36
Chapter 4: Implementation: <i>Corpus</i>	38
4.1: Dynamic Simulation.	38
4.2: Integration.	40
4.3: Articulated figures in corpus	42
4.4: External Forces.	44
4.5: Joints and Joint Forces.	47
4.6: Gait Controller.	48
4.7: Hexapod Data.	50
Chapter 5: Results and Analysis.	52
5.1: Dynamic Simulator.	52
5.2: Gait Controller.	54
5.3: Walking Experiments.	56
5.4: Computer Animation and Dynamic Locomotion.	59
Chapter 6: Future Work.	63
Appendix A: Spatial Algebra.	65
Appendix B: The Articulated Body Method for Dynamic Simulation.	70
Appendix C: Roach Construction Scripts.	76
Appendix D: <i>Corpus</i> Commands.	83

1. Introduction

In recent years, the field of computer animation has been turning more and more to dynamic simulation in order to produce realistic motion. Dynamic simulation presents new problems when compared to kinematic simulations, however. Most notably, the problem of controlling motion becomes more complex because controlling forces must be supplied at the joints, instead of directly manipulating joint angles. However, physically-based approaches, being based on real-world phenomena, encourage the development of control strategies which can be used in the field of robotics, and which can test the validity or usefulness of theories on how biological systems control motion.

This thesis develops a strategy to coordinate and control the motion of a hexapod in a physically-based manner. A completely dynamic model of locomotion has not been previously reported in the computer graphics literature. There are three major components to the approach: a *dynamic simulator* which creates physical motions, a *gait controller* which coordinates the activity of stepping and stance, and *dynamic motor programs*, which produce forces to control the motions of the limbs. The dynamic simulator computes the accelerations of rigid bodies, jointed into articulated figures, in response to applied forces. The gait controller sequences stepping such that coherent patterns are created, suitable to move the body forward at a given velocity. The motor programs control the desired motions of stepping and stance, providing forces to propel the body forward.

Dynamic simulation presents many problems, such as slow computation time, stability problems, and complex algorithms to implement, but it has many desirable features. Many behaviors are automatically produced in dynamic systems, such as: falling under gravity, bouncing collisions, supporting contacts, interaction among different objects, and velocity and momentum effects. Dynamic simulations attempt to mimic the physical properties of the real world, and thus animations which employ dynamics appear very realistic.

One of the most interesting aspects of dynamic simulations is the way in which dynamic elements can adapt to their surroundings through *mechanical compliance*. A simple example of such compliance would be a dynamic linkage which falls to a (possibly curved) surface under gravity. After it has fallen and settled, it has conformed, or complied, to the shape of the surface. The resulting configuration of the linkage follows the surface, without the system explicitly computing the joint angles needed to conform. A more complex example, demonstrated in this thesis, is locomotion over uneven terrain. The springy properties of the limbs allow a leg to conform to the differing heights of the terrain, without actually planning the different joint angles needed to adapt to the different heights.

The gait controller is a biologically-inspired model, using theories from neurology and physiology. Basically, the gait controller employs coupled oscillators, or pacemakers, which generate basic stepping patterns. Reflexes serve to reinforce the basic stepping pattern, while increasing the adaptability of the gait to external disturbances. Walking speed is set by simply assigning the appropriate oscillator frequency—high frequencies for fast walking, low frequencies for slow walking—and the gait controller creates the appropriate stepping pattern.

The motion control technique used in this thesis is based on exponential springs and dampers placed at the joints in the hexapod figure. The springs supply forces to position the limbs, against the disturbances of other forces, such as impacts with the ground. Motion is controlled by moving the rest position of the springs over time, using motor programs. In this manner, the legs are driven to lift up and forward during stepping, and to support the body and drive it forward during stance.

The layout of this thesis is basically as follows. Chapter 2 provides background material for the discussion, and presents the related work of other researchers. Chapter 3 recounts the approach I have taken to create physically-simulated locomotion. Chapter 4 details the implementation of my approach, the program

corpus. Chapter 5 presents the results of my walking experiments, and analyzes the properties of the locomotion. Finally, chapter 6 concludes with a summary of the research, and puts forth ways in which the work can be continued and improved upon.

2. Background and Related Work

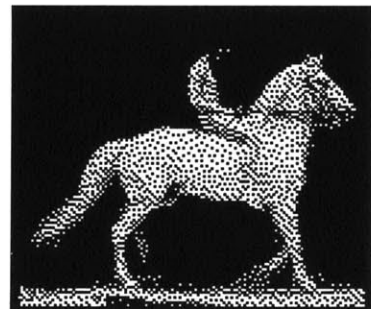
As there are three main problems to be addressed by this thesis, the background chapter will be divided into three main sections: *dynamic simulation*, *gait generation* and *motor control*. An additional section, *walking machines and simulations*, will present other researchers' synthetic locomotion results.

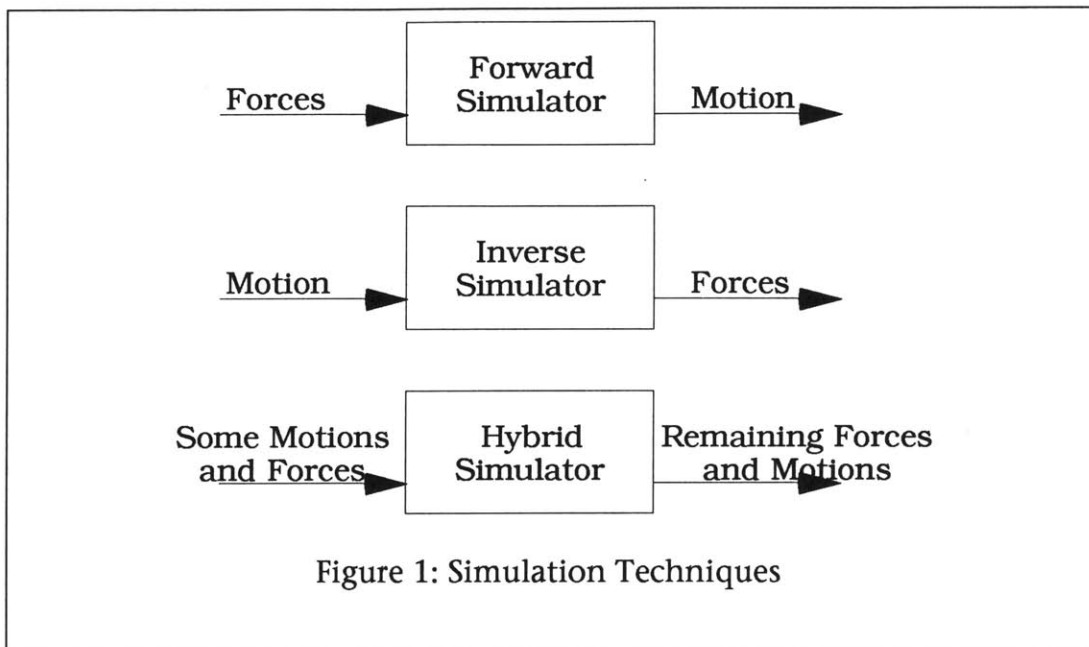
2.1. Dynamic Simulation

The physical simulation of motion plays a key role in this thesis, because we are interested in achieving a *realistic* model of locomotion. Realistic not only in the sense of *appearing* convincing, but also in the sense of accurately representing the physical world in which we live. Because of this quest for realism, dynamic simulation forms the foundation of the walking system presented in this thesis. By *dynamic simulation* we mean that all movements occur according to the laws of Newtonian physics, as they would in the real world, under the same conditions. We *simulate* the world, using the laws of *dynamic motion*.

Dynamic simulation covers a broad category of techniques, only some of which are employed in this thesis. Two main classes of techniques can be specified, *inverse* and *forward* dynamics. Inverse dynamics solves the problem of determining what forces would be required to produce a specific motion. Forward dynamics, on the other hand, calculates what motion would result given applied forces. However, many dynamic simulation techniques allow both forces and motions to be specified, then solve for the unspecified forces and motions. We will term the simulators which employ this technique *hybrid simulators*

Extended figure 1: On the bottom right of each page in this chapter, you will find a photograph of a horse with rider, taken by Eadweard Muybridge in the late 1800's [1]. Muybridge was one of the first researchers of the stepping patterns of animals and people. By *slowly* flipping the pages of this chapter, you will observe the "amble" gait of the horse.





(see Figure 1). In this thesis, we are not interested in specifying the final motions involved in locomotion. If we knew the motions ahead of time, there would be little point in simulation, unless we wanted to study the forces involved in the specified locomotion. Instead, we will specify the *goal* of locomotion and employ calibrated force-producing agents to create approximate limb motions. Thus, a forward dynamics simulator is required.

Another division of simulation techniques can be made: *flexible object simulation* and *rigid body simulation*. As its name implies, *flexible object simulation* deals with objects capable of deformation, such as skin, Jello™ and cloth. *Rigid body simulation*, on the other hand, deals with objects incapable of deformation. Although most real objects deform in some way, bone, robot linkages, insect exoskeletons and many other nearly-rigid bodies can often be approximated as rigid bodies.

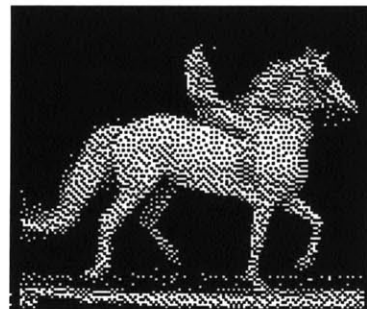
It should be noted that potentially important factors are ignored when the rigid body approximation is made. For example, the deer, dog and other quadrupeds are believed to store energy, as elastic strain



energy, in the aponeurosis (a spinal bone) during galloping [2]. In addition, modal vibrations of robot parts can induce movements very different from predicted results which are based solely on a rigid body simulation. A large body of research has recently been performed on correctly modeling the elastic properties of robot links and joints [3; 4]. Plastic deformations and breakage due to stress and strain are also not accounted for in basic rigid body simulators.

Rigid body simulators typically allow bodies to be connected together to form *articulated figures*. These figures are comprised of rigid links connected by joints, which allow the links to move relative to each other with one or more degrees of freedom. With each new link, the overall equations of motion for the articulated figure change; i.e. the motion of each link is dependent on the motion of the other links. Therefore, articulated figure simulators must be generalized to deal with different link and joint configurations.

One way to achieve this generality is to employ a *constraint-based* approach. In constraint methods, the relationships between different links is defined, and the entire system is solved simultaneously. For example, two links could be constrained to be connected together at a joint which does not allow them to separate, but does allow them to rotate freely around the connection point. The constraint equations would then solve for the force required to hold together the connection. This solves an inverse dynamics problem in the sense that forces are being computed from a specified motion (or lack of motion). However, the rotary motion at the joint would remain unconstrained, and the resulting motion at the joint would be determined by other applied forces, solving the forward dynamics problem. Therefore, since both motion and forces can be specified, constraint methods are classed as hybrid simulators, as described above. An advantage of constraint based methods is that constraints can be established not only between links, but also between links and the environment. For example, one end of a link could be fixed to a point on the ground. Constraint methods also allow complex

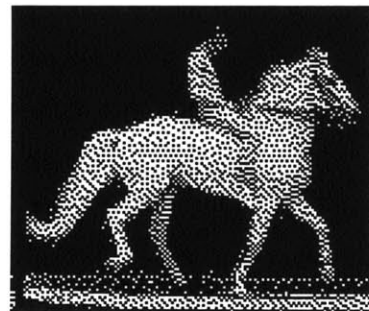


linkage geometries, such as kinematic loops, to be created. The main drawback of constraint methods is that they require more computation than other methods (described below). Another problem is that they can be more numerically unstable [5; 6].

Isaacs and Cohen describe a straightforward method of constraint simulation based on a matrix formulation [7]. Joints are configured as kinematic constraints, and either accelerations or forces can be specified for the links. An equation is established for each degree of freedom, yielding n simultaneous equations to solve. These equations form a matrix which needs to be inverted, yielding $O(n^3)$ complexity. Interdependencies among the constraints typically make the matrix non-sparse, such that sparse matrix solutions cannot be employed to reduce the complexity.

Barzel and Barr have described constraint-based simulators in [8]. Their methods allow for the “self-assembly” of linkage structures by satisfying the constraint equations using a critically damped function. For example, two links which are separated could be constrained to connect together, and because the constraint equation is critically damped, the parts would move together and connect in the fastest possible time, without overshooting each other. The published Barzel-Barr method is of order $O(n^3)$ computational complexity, where n is the number of constraints. This can be reduced to approximately $O(n^2)$ using a sparse matrix solution [6].

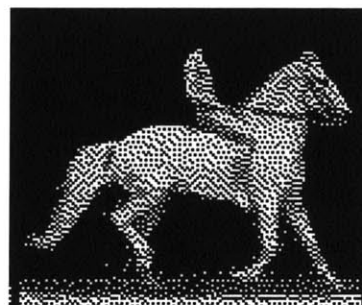
Witkin and Kass describe a method they term *spacetime constraints* in which the constraint equations are solved not only for the joint/link geometry, but also for the applied control forces [9]. Another important feature of their system is that the constraint equations are solved simultaneously for the entire time span of the simulation. Because of this, energy (or some other function) can be minimized over all of time, unlike Barzel and Barr’s system which can introduce extra momentum in links as self-assembly



occurs. Solving over the entire span of time incurs a very large computational cost, however. Spacetime constraints will be discussed further in the **Motor Control** section below.

Other methods for solving the articulated, forward dynamics problem directly encode the link/joint relationships into the dynamics equations. Instead of solving the constraint forces required to hold joints together, these methods solve the equations of motion for the figure using the joint geometries as part of the formulation. Therefore, self-assembly (as in Barzel-Barr) is not possible in these systems. However, these methods can use the joint relationships to reduce the number of computations required for the dynamics of the figure, making them more efficient than the constraint methods.

The non-constraint methods fall into two main categories: ones that form a set of simultaneous equations for the accelerations and then solve them, and ones that form a recursive relationship propagating force and movement information through the linkage, solving directly for accelerations [10]. The first type of formulations typically yield $O(n^3)$ complexity, because there are n simultaneous equations to solve (forming a matrix which requires inversion). The recursive formulations, however, can reach $O(n)$, although the benefit of the recursive methods is not without cost. They are more difficult to develop, and the cost of the computations per link is higher than the simultaneous equation methods. Articulated figures with few joints are less expensive to compute using simultaneous equations; figures with many joints are more efficiently computed using the recursive methods. According to Featherstone [10], the Walker-Orin method [11] is the most efficient of the simultaneous methods, and Featherstone's Articulated Body Method is the most efficient of the recursive methods. The point at which Walker and Orin's method becomes more efficient than Featherstone's is when $n < 9$. Because the articulated figure to be modeled in this thesis has many links and joints, I chose the Articulated Body Method (ABM) as the basis for our simulation

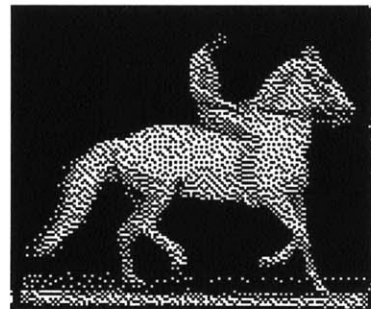


method.

Armstrong has developed a recursive dynamic formulation, which has approximately the same computational efficiency as the ABM [12; 10]. Armstrong *et al* have experimented with this method to simulate a moving human figure in near-real time [13]. However, the Armstrong method is only capable of efficiently simulating spherical joints, unlike the ABM which allows fully general joint types.

Another efficient recursive method has been formulated by Lathrop [14], and implemented by Schröder [6; 15]. The advantage of Lathrop's method over Featherstone's is that it allows for kinematic constraints at the end-effectors. For example, Lathrop's formulation would allow the end-effector of a linkage to maintain connection with a point on the ground. Lathrop's method can also be extended to handle kinematic loops [6]. It is not clear, however, that these constraints would be generally useful for the physically-accurate simulation of locomotion. This topic will be discussed more in the **Future Work** chapter.

An important aspect of dynamic simulation is the computational numerics involved in executing the various algorithms. Most forward dynamics algorithms compute accelerations. These accelerations need to be numerically integrated into velocities and positions. Various integration methods have been developed for computational use; many of these are discussed in [16; 17; 18]. Whatever integration method is used, the integrator must sample the dynamics solution at various discrete points in time to estimate the continuous solution. The better the integration technique, the fewer times it will have to sample the accelerations in order to get a *stable* solution for the velocities and positions. An *unstable* solution will be dominated by incorrect results, very often exhibited by high frequency perturbations. A common problem in dynamic simulations is that the system is often *stiff*. Stiffness results when low frequency components and high frequency components



(which decay to zero) are mixed in a solution [18]. Typically, the high frequency components are not of interest, but can cause certain integration techniques to closely sample them in order to remain stable. Stiffness results when accelerations are (partially) a function of velocities and positions. This can be due to functions such as dampers, springs and friction.

2.2. Gait Generation

The dynamic simulation techniques discussed above form the physical basis for simulated locomotion, but are independent of the control of locomotion. This section will introduce coordination methods used in natural and synthetic locomotion, and the following section will discuss techniques used to produce the forces required for physically-based motion.

The Soviet physiologist Nicolai Aleksandrovitch Bernstein developed a theory concerning the control and coordination of movements which has been termed the *Bernstein perspective* [19]. One of the major aspects of the Bernstein perspective is the question of how movements are performed when there are so many degrees of freedom to control. For example, the human arm (including the wrist but not the hand) has 7 degrees of freedom of movement due to its joints. In addition, there are 26 muscles active on these joints. Further, there are hundreds of motor units per muscle. If one desires to simply move one's wrist forward in a straight line, then, essentially, only one degree of freedom in Cartesian space should be affected. However, the 7 joint angles must be simultaneously controlled by the 26 muscle groups which are composed of thousands of muscle fibers. At the lowest level of control, thousands of units must be successfully activated to correctly achieve the motion.

Given so many degrees of freedom to control, it becomes desirable to effect some mechanism which allows the highest levels to concern themselves with only the few degrees of freedom required to specify an action. It appears from much research that natural movements are coordinated by a hierarchical



structuring of functional units which adapt to each other as well as external influences [20].

In order to move from place to place, an animal must coordinate its limbs to bring about coherent motion. Legs are alternately controlled between *step* and *stance*. Stepping, of course, brings the leg up and forward, while stance supports the body and drives it forward. The overall sequence of the various legs stepping and standing is termed the *gait*.

Natural gait has been the focus of much research for many years. One of the first researchers of gait was Eadweard Muybridge, who in the 1870 and 80's took time sequenced photographs of various animals and humans at differing speeds and gaits [1]. (See Extended figure 1 throughout this chapter). Muybridge, Hildebrand [21], and others have cataloged and analyzed the stepping patterns of the legs during different mammalian gaits, such as galloping, cantering, trotting, etc. These running gaits are discrete, e.g., either the animal is trotting, or galloping. The transition period between different gaits is almost nonexistent. This is in contrast to insect gaits, which the work of Hughes [22] and Wilson [23] shows to be continuous in nature. To elaborate, for each unique speed at which an insect travels, there is a unique gait associated with that speed. A smooth shift through different speeds results in a smooth transition of gaits. One possible reason for the apparent difference between mammalian and insect gait continuity may be that larger animals have certain mechanical resonances in their skeletal and musculo-tendon structure[24; 25]. The discrete mammalian gaits correspond to these resonances, and gaits which would fall outside of the resonances would cause the animal to expend more energy and to experience discomfort. Because insects are so much smaller than mammals, inertial effects play much smaller roles [26]. The mechanisms which generate mammalian gaits may actually be continuous in nature, and indeed, computational mechanisms which generate continuous insect gaits are also capable of generating many of the discrete mammalian gaits [27].



The gait-generating mechanisms employed by the cockroach have been studied by many researchers. Wilson analyzed the stepping patterns cockroaches exhibited under a variety of conditions [23]. He then developed five rules which describe the gait behavior of many insects:

1. A wave of steps runs from rear to head (and no leg steps until the one behind is placed in a supporting position).
2. Adjacent legs across the body alternate in phase.
3. Stepping time is constant.
4. The frequency with which each leg steps varies.
5. The interval between steps of adjacent legs on the same side of the body is constant, and the interval between the stepping of the foreleg and hind-leg varies inversely with the stepping frequency.

Wilson made hypotheses about the neurological mechanisms which could generate these rules, and his ideas were confirmed by the experimental work of Pearson [28]. Each leg in the cockroach has a pacemaker or *oscillator*, which rhythmically triggers the leg to step. The oscillators are coupled together, and their interaction generates the various gaits. At slow oscillator frequencies, slow “wave” gaits are generated, and at high frequencies faster wave gaits and the “tripod” gait is generated. As the oscillator frequencies smoothly change, a smooth gait change is effected. The concept of coupled oscillators is not a new one; in the 1930’s, von Holst proposed the coupled oscillator model to explain the wide range of interacting cyclic behaviors he observed in nature [29].

In addition to the coupled oscillators, *reflexes* also play an important role in gait generation. Reflexes can both trigger or retard the stepping of limbs [28]. In the cockroach, the *step reflex* causes a leg to step when hair receptors detect that the leg has nearly reached its back-reaching extent. Another cockroach reflex employs cuticle stress-receptors, which measure the load that a leg is bearing, and prevent a leg from stepping if it is sup-



porting the insect. In general, reflexes reinforce the stepping pattern generated by the coupled oscillators, while increasing the adaptability of the creature under changing environmental conditions. Reflexes seem to play an even more important role during locomotion over uneven terrain. A study by Pearson of locusts walking over uneven terrain shows that a rigid gait is not employed over rough terrain [30]. To find suitable footholds, the legs employ searching tactics, and an *elevator reflex* causes the leg to lift higher if it encounters an obstacle during a stepping movement.

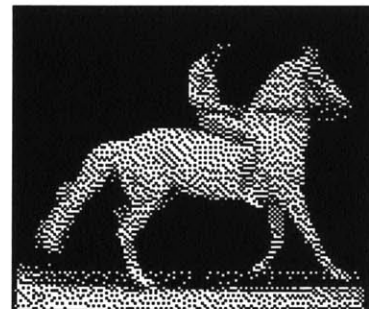
In order to synthesize gait for walking robots or simulations, researchers have created the field of computational gait. McGhee [31] derived the following formula which describes the number of distinct gait patterns possible for a k -legged system:

$$N = (2k - 1) ! \quad \text{Equation 1}$$

For a quadruped, therefore, $7! = 5040$ distinct gait patterns are possible. For a hexapod, $11! = 3,916,800$ gaits can be generated. Selecting which gait to use from this large number can seem overwhelming. Using a matrix analysis of gait, however, McGhee and Jain showed that the 5040 quadruped gaits can be reduced to 492 temporally equivalent gaits. They further showed that the 492 gaits can be reduced to 45 equivalence classes. Sun employed further matrix transformations to reduce quadruped gaits to 14 equivalence classes [32]. Sun also showed that the 3,916,800 hexapod gaits can be reduced to 148 equivalence classes. In addition, all 288 regular symmetric hexapod gaits can be reduced to 7 equivalence classes.

Many of the methods used to actually generate gaits are based on *finite state control* [31; 24; 33]. Under this method of control, each leg cycles through a set of states, such as step and stance. Stepping and stance is typically broken down into smaller sets of states such as step up and forward, and step down and forward.

Communication between the individual leg state machines, and commands from higher levels of



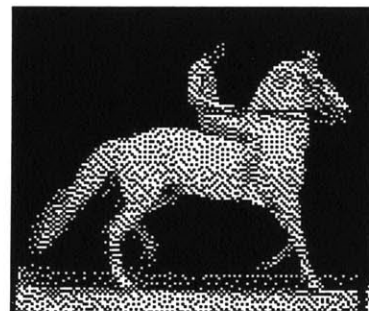
control determine the stepping pattern.

Beer, Chiel and Sterling employ a heterogeneous neural network to simulate the stepping patterns exhibited by the cockroach [34; 35]. The network is created by a set of coupled differential equations, which model a neuron cell membrane. The cell membrane acts as a resistor-capacitor circuit and sums the input potentials, generating an output frequency. The gait generator employs 37 neurons. For each of the six legs there are three motor neurons, two sensory neurons and one pacemaker neuron. Overall walking speed is set by one command neuron. By varying the firing frequency of the command neuron, a variety of gaits is generated by the network. Slow walking speeds employ the wave gait, and the highest speed employs the tripod gait, the same behavior exhibited by many insects.

2.3. Motor Control

Once the overall pattern of step and stance sequencing has been coordinated, it must be performed in some manner. In a kinematic simulation, the geometry of the desired motions alone determines the movements. In a forward dynamic simulation, however, forces must be delivered to the limbs to create the movements. The problem of dynamic motor control is determining what forces need to be supplied over time in order to create a specified motion. For the fine control of motion, exact forces are required. These forces can be difficult to compute, due in part to the forces introduced by the interaction of the different linkages and joints in an articulated figure. For a broader scope of motion, such as "swing leg up and forward," less demanding methods can be employed.

In biological systems, the basic producer of bio-mechanical forces is the muscle. McMahon [36] contains an excellent review of the force-producing properties of muscle, under varying types of stimulation. In 1922, V. A. Hill proposed a mechanical model for the active muscle which is depicted in Figure 2. Hill's model is based on a contractile ele-



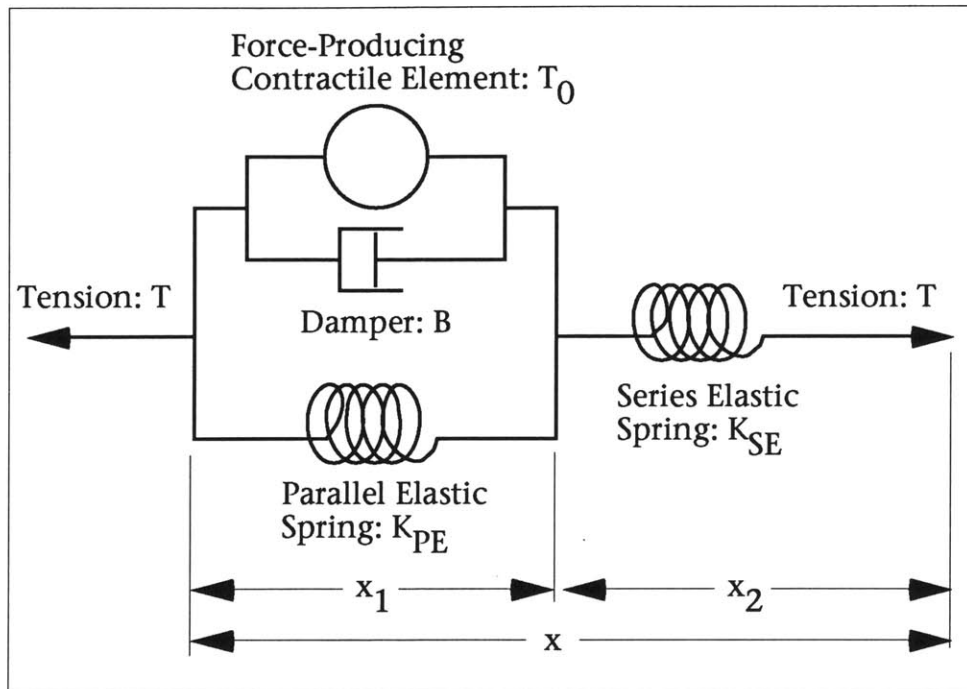


Figure 2: The Hill model for the force response of the active muscle.

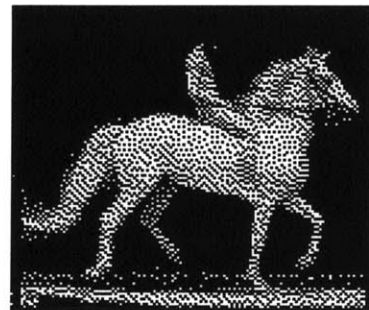
ment which consists of a force generator in parallel with a non-linear damper. The damper exerts a force as a function of the velocity of muscle shortening (\dot{x}_1). The contractile element exerts a force, T_0 , as a function of the contractile element length, x_1 , and time, t . The contractile force rises and falls with the muscle stimulation, and during tetanus (the highest efficiency of stimulation and force development) the force T_0 rises to a constant level, equal to the isometric tension on the muscle. In addition, a spring in parallel with the contractile element exerts a force as a function of x_1 . Another spring lies in series with the contractile element and parallel spring, and produces a force as a function of the length x_2 . The parallel spring models the elastic properties of the muscle fibers, while the series spring models the elastic properties of the connective tendon. The Hill model has proven very useful in describing the mechanical properties of muscle acting under a load in the skeletal system. In some ways, however, the Hill Model is misleading. From



the diagram in Figure 2, one would be led to believe that a mechanical damper, such as the viscous water medium in the muscle, is acting as a dashpot. In fact, the mechanical properties of the muscle's water does not match the damping effects seen in the Hill Model. The Hill model explains the muscle's force response in mechanical terms, however, it should be noted that muscle is more accurately modeled as biochemical reactions which release energy.

Reflexes and feedback are used to regulate the action of the muscle [36]. The *stretch reflex* increases the amount of stimulation a muscle receives when it is stretched beyond its equilibrium length, as an attempt to return the muscle to its previous length. This reflex is initiated by the muscle spindle organ, which lies in parallel with the main muscle fibers. Within the spindle organ lies a small set of muscle fibers, and a nuclear bag fiber which detects changes in length. When the muscle is stretched by some outside force, the spindle organ senses the change in length and increases stimulation to the muscle fibers. During voluntary movements, the muscle fibers in the spindle organ are co-stimulated with the main muscle mass, from higher centers. This keeps the muscle and spindle organ at approximately the same length, so that the stretch reflex does not interfere with a stretch controlled by higher centers.

Sensors in the tendon organs are responsible for *reflex stiffness*. The sensors measure muscle force, and send inhibitory signals to the muscles. It is thought that this reflex allows the skeletal-muscular system to present a constant stiffness to the world, in spite of changing environmental conditions. A balance is achieved between the spindle organ and tendon organ signals, so that neither length nor force regulation has complete control of the muscle system (see Figure 3). The tendon organs are also responsible for the *clasp-knife reflex*. This reflex prevents the muscle from over-exerting itself and causing damage to tissues. The force sensors in the tendon organ detect strong force and send strong inhibitory signals to the muscle. When the reflex becomes active, the limb it controls suddenly loses tension, which



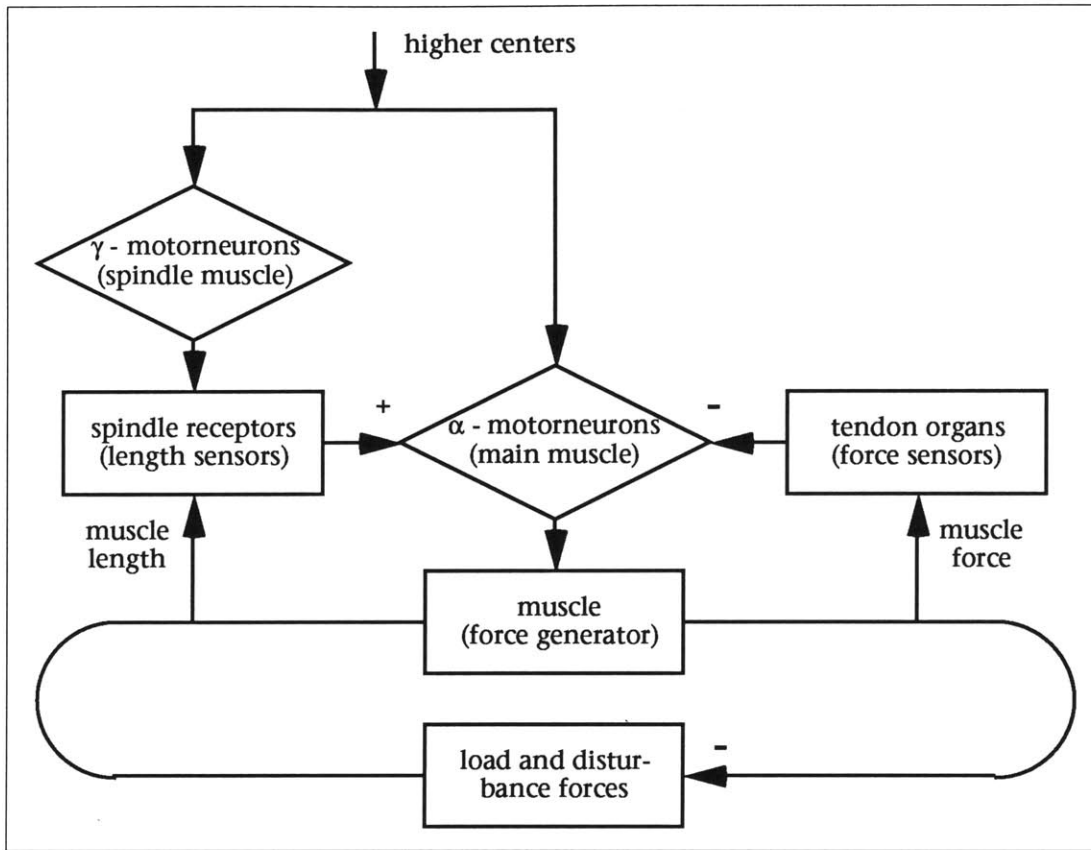
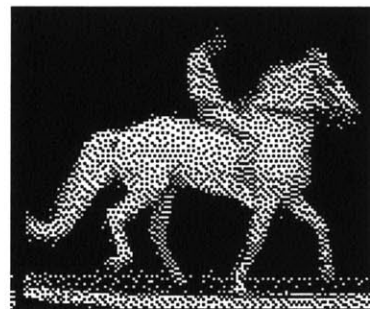


Figure 3: A diagram of stiffness regulation in the stretch reflex [36].

looks somewhat like a clasp-knife returning to its sheath.

Starting with the assumption that muscles are tunable, spring-like force generators, motor control researchers have come up with an *equilibrium-point hypothesis* to explain how controlled movements are produced [37; 38; 39; 40]. This model treats the muscle, along with its feedback system, as a single, tunable unit, with measurable, spring-like properties. Postures are controlled by establishing an equilibrium between agonist and antagonist muscle groups. This equilibrium configuration forms a point in a controlling space, which can be specified by the neuromuscular system. The equilibrium-point hypothesis states that movements are produced by changing the equilibrium point from one posture to another. Exactly how the change from

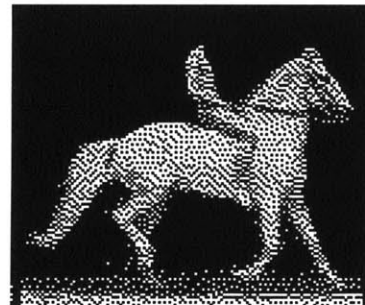


one point to another is accomplished is not fully understood. Experiments by Bizzi, Chapple and Hogan show that it is not the case that the equilibrium point is instantly switched from one point to another [37]. Hogan describes a “virtual trajectory” of equilibrium points which control movements[39].

One of the basic controlling means in nature is the servomechanism [20], which employs negative feedback to steer a system toward some equilibrium. Sensors are used to measure some signal, and an error term is constructed based on the displacement of the system from its goal. The error term is used to stimulate the system in the direction of the goal. The way in which moths orient towards the light is a servomechanism; unless both eyes detect light, the moth turns more towards the light. If one eye of a moth is blinded, it turns continually in a circle, since only one eye is ever illuminated. The controlling means of the stretch reflex, described above, can also be thought of as a servomechanism, since it employs active feedback to regulate the length of the muscle system.

The study of *computational motor control* has arisen mostly from the need to control robotic systems. Again, the problem comes down to what forces are required to produce specific motions. In many cases, the demands on precision can be very high. In some applications, not only the motion of the system needs to be controlled, but also the force applied by the robot onto its environment.

Before describing computational motor control, an interesting example of *interactive* motor control will be presented. An interactive graphical editor, *Virya*, developed by Wilhelms, allows controlling functions to be specified for an articulated figure simulator [41]. The user draws and edits curves to indicate what forces or torques are to be applied at a degree of freedom. Alternately, the motion of the degree of freedom can be defined with a curve, and an approximate force will be automatically calculated to achieve approximately that motion. Results from using *Virya* and similar experi-



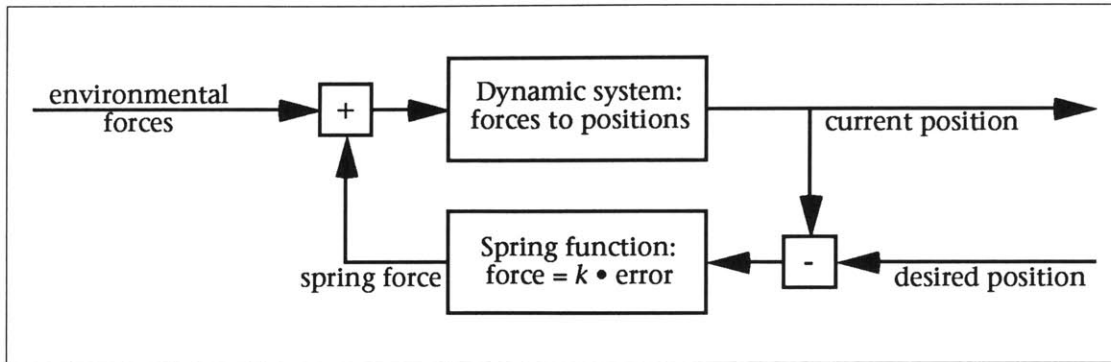
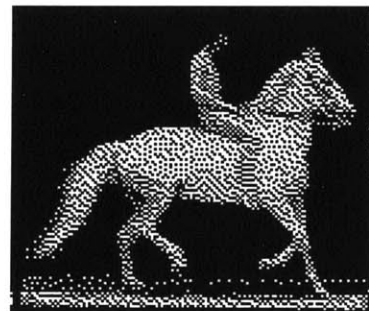


Figure 4: a simple servomechanism implementing spring position control

ments by Armstrong, Green and Lake [13], reveal that direct manipulation of the forces and torques needed to achieve certain motions or goals is very difficult. One problem is that there are typically many degrees of freedom to control. Another problem is that forces applied at one degree of freedom influence not only that DOF, but also (potentially) the entire dynamic system. In addition, Coriolis and centripetal forces arise due to the motion of the bodies in the system, and must be counteracted for accurate motion.

A basic servomechanism might be used to automate motion control. A simple feedback system which employs a spring to draw a dynamic system towards a specified position is shown in Figure 4. A linear spring function produces a force proportional to the displacement of the dynamic system from the target position. The feedback system will automatically respond to external perturbations which cause a drifting from the target. The problem with this basic approach is that the system will not necessarily ever reach the target. Once an equilibrium is reached between the spring force and external forces, the spring cannot draw the system any closer to the goal. Increasing the spring strength, k , will make the error displacement smaller and smaller, but not zero. In addition, the simple spring system depicted in Figure 4, in the absence of damping (from external forces), will actually fight itself; the spring will bring the system closer to the target, but in the process the system will pick



up momentum and overshoot the target. The spring will then have to pull the system back again, and so on.

Inverse dynamic techniques (including constraint based systems) can calculate the exact force required to bring the system to a target, without any displacement. However, in order to exactly counteract external disturbance forces, those forces must be known. In a simulation system, it is reasonable to say that these forces will be known, but it is not always a valid assumption that the controller should have knowledge of them. For example, if a simulated human was standing and received a push from the side, the controlling system should not know exactly what force is being applied and immediately counteract it, if a humanly-realistic motion is desired. Instead the human should begin to fall to the side, and when the controller discovers the error in balance, it would try to correct and rebalance.

An, Atkeson and Hollerbach employ *model-based control* [42], in which a descriptive model of the kinematics, dynamics and motors of a system are used to compute accurate controlling forces for robot manipulators. The complete robot system, including motor properties, linkage elasticity, joint viscosity, etc., is often termed the *plant*. A model of the *inverse-plant* is used to predict and plan how the system will respond to controlling input. Determining the exact values for a plant, and thus its inverse, can be difficult, however.

Another class of inverse-dynamics controlling methodologies are *optimization techniques*. With these techniques, a goal is specified, along with the details of the physical system. Some measurable quantity, such as energy expenditure, is chosen to be minimized. The system is then analyzed over a period of time to determine what controlling parameters will yield the optimal result. Because optimization techniques solve the entire dynamic system with controlling parameters over an interval of time, they can be quite computationally expensive. Brotman and Netravali have devel-



oped a system using optimal control to interpolate between a set of motion key-frames [43]. Witkin and Kass term their method of optimal control *spacetime constraints* [9]. Their method allows goals to be specified in kinematic and energy terms, such as “jump from here to there, clearing a hurdle in between, and don’t waste energy.”

Optimization techniques along with other constraint techniques can be termed *key event simulation*, because specific events must be satisfied. This is in contrast to *forward simulation* in which a system is established, and then simulated forward in time [44]. Controlling techniques for key event simulation require global knowledge about the system, sometimes over the entire time span of the simulation. Forward simulation controllers, however, need only partial knowledge of the world, presumably derived from a simulated sensor. The more sophisticated the controller, the more knowledge of the world it will require. External influences, such as interactive input, can be easily incorporated into a forward simulation at any point in time, unlike methods such as spacetime constraints, which must be solved over the entire span of time.

2.4. Walking Machines and Simulations

Walking machines have been the subject of research for many years. A human-controlled, four-legged vehicle was developed at General Electric by Liston and Moser in the mid-1960 [45]. Using force-feedback, the driver could “feel” obstacles and negotiate uneven terrain. The first example of an autonomous legged vehicle, controlled by a digital computer, was a hexapod robot developed by McGhee [46]. The hexapod could employ a number of gaits, and negotiate rough terrain with areas marked as forbidden. A similar hexapod was developed by Sutherland [47]. Its gait control system is similar to the one presented in this thesis [24]. Legged robots designed by Raibert differ from the above robots in that they employ dynamic balance; i.e. they can run and hop without enough legs always on the ground to statically support the body [48].



In the computer graphics field, simulations of legged locomotion have also been an area of active research. A kinematic simulation of human walking was developed by Zeltzer using a finite state machine approach [33]. The biped model was capable of walking over uneven terrain. However, since the motions lacked a physical basis, some of the walking appeared overly simplified, especially when adapting to terrain. Girard developed a system for kinematically simulating the locomotion of creatures with various numbers of legs [49]. In his system, gaits and leg motions are interactively created, and gait shifts are accomplished by simply interpolating between defined gaits. Some dynamic aspects were also added, such as parabolic trajectories while in the air and banking during turns. However, because the motions are interactively defined, the overall motions lack a complete dynamic basis, which can make them appear artificial. Sims created a walking system with an interactive figure editor which allows the user to quickly and easily define new figures [50]. The figure can then automatically employ defined gaits over uneven terrain, and also hop or “pronk” using a dynamic trajectory. A dynamic model of snake and worm locomotion was developed by Miller [51]. Although not legged locomotion, Miller’s work is interesting in this context because of its dynamic nature. The forward motion of the snakes is ultimately due to friction, as is the motion of the roach in this thesis.

3. Approach

This chapter describes the approach that I have taken to develop a dynamic model of locomotion. My approach is basically one of *forward simulation*, where a physically accurate model of a hexapod and its environment is constructed, and a control algorithm for walking is established (without global knowledge of its environment), and the system is then simulated forward in time. The success or failure of the system depends on the accuracy and robustness of the control system, as the dynamic simulation will faithfully incorporate all controlling forces, whether they help or hinder locomotion.

My approach breaks down into three main *procedural* components which describe how the physical and controlling mechanisms will operate, and a set of *structural* components which describe the parameters of the hexapod (see Figure 5). The three procedural components are: a dynamic simulator, a gait controller and motor programs. Dynamic simulation forms the foundation of the system, since all motions are computed by the simulator. Because a forward dynamics simulator is employed, forces must be applied to create movement.

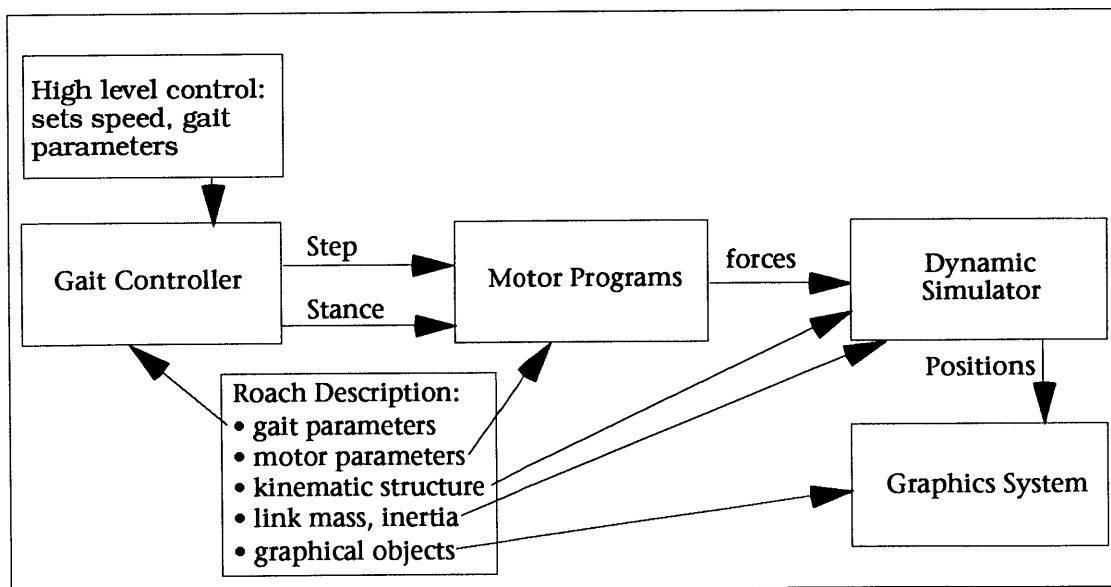


Figure 5: breakdown of approach for dynamic locomotion

Featherstone's Articulated Body Method (ABM) [10; 52] was selected as the simulation algorithm, because it is one of the most efficient methods for forward dynamics. The gait controller coordinates the sequences of stepping and stance for the hexapod. The controller used in this thesis is based on the coupled oscillator model with reflexive feedback [28]. The motor programs generate the forces required for stepping and stance. They are based on exponential spring and damper combinations, which are tuned over time. In addition, there is another procedural unit, the graphics system, which is not directly involved in the creation of locomotion, but is required for display and animation of the simulation.

The structural components describe the parameters that the functional units will operate on. The structural model of the hexapod describes how it mechanically constructed (i.e. the kinematic structure of the links and joints, and the mass and inertia of the links), how it appears when graphically rendered, and the specifics of how it will move (i.e. the maximum speed, motor program parameters, etc.) Some of the structural elements were designed from a study of insect physiology, others were developed by trial and error methods.

3.1. Dynamic Simulation

As has been previously noted, the dynamic simulator forms the basis of all motion in this dynamic locomotion system. The simulator accepts forces acting on an articulated body as input, and computes the acceleration of the degrees of freedom of the figure as output. These accelerations are numerically integrated over time to give rise to the velocities and position of the DOF's.

Featherstone's ABM is introduced in [52], and developed further in [10]. Featherstone claims (with substantial proof) that the ABM is the most efficient algorithm yet developed for forward dynamics simulation. Deyo and Ingebretsen [53] claim that Bae's recursive algorithm [54] is the most efficient, although the claim is not substantiated. The main reason for the algorithm's efficiency is because it is recursive in nature. Instead of solving a set of simultaneous equations describing the kinematic and dynamic relationships between

different links in an articulated figure, recursive dynamics formulations establish a recursive, linear relationship between parent and child links. The simultaneous equations methods have a computational complexity of approximately $O(n^3)$, whereas recursive methods can achieve a complexity of $O(n)$, as the ABM does.

The ABM operates on branching, articulated figures, comprised of links connected by joints. The joint type in the ABM is flexible in that many different joint types can be specified. For example, rotary and prismatic (translational) joints are supported, as well as combinations of both, such as screw and cylindrical joints. A joint can have from one to six degrees of freedom.

One of the links in the articulated figure is chosen to be the *root body*, or base. This link can actually be any of the links in the figure, but it is logical to choose the most central body in the branching structure. If the recursive algorithm were implemented on a parallel processor machine, then the most efficient root body would be the most central, since all branches could be processed in parallel. The root body is somewhat special in the ABM since kinematic constraints can be specified for it. The other links in the figure, including the end-effectors (last link in a branch of the figure), do not support kinematic constraints; their motion is determined by the applied forces only. However, an inverse dynamics algorithm can be used to compute those forces needed for kinematic constraints.

The key feature of the Articulated Body Method is the concept of the articulated body inertia. Just as the inertia tensor of a rigid body determines its acceleration when the applied force is known, the articulated body inertia tensor of a rigid body, within an articulated figure, determines the acceleration when the applied force is known. Formally, the equation of motion for a free rigid body is:

$$\hat{f} = \hat{I} \hat{a} + \hat{p}^v \quad \text{Equation 2}$$

where \hat{f} is the applied force, \hat{I} is the inertia tensor, \hat{a} is the acceleration and \hat{p}^v is the bias force (centripetal and Coriolis) produced by the body's velocity. The

equation of motion for rigid body in an articulated figure reads:

$$\hat{f} = \hat{I}^A \hat{a} + \hat{p} \quad \text{Equation 3}$$

where \hat{I}^A is the articulated body inertia tensor, and \hat{p} is the bias force which incorporates the \hat{p}^v from above, as well as joint reaction forces. The $\hat{}$ above these quantities denote that they are represented in *spatial notation*. This notation, developed by Featherstone, is based on screw calculus [55], and essentially unites

the rotational and translational aspects of motion into a single vector quantity. Appendix A gives an introduction to the spatial algebra required to implement the ABM, however, Featherstone should be referred to in [10] for a tutorial in and reference for spatial notation.

The articulated body inertia (\hat{I}^A) gives directional properties to the *apparent* mass of a body. To use an example from Featherstone [52], a simple two dimensional block and gantry system will be described (see Figure 6). The first mass, m_1 , is a block free to move up and down in y along the second mass, m_2 , to which it is attached by a prismatic joint. The second mass is free to move from side to side in x along another prismatic joint. A force applied to m_1 in the y direction will yield an acceleration in the y :

$$a_y = \frac{f_y}{m_1} \quad \text{Equation 4.}$$

A force applied to m_1 or m_2 in the x direction, however, needs to overcome the combined mass of both the block and gantry, producing an acceleration in x :

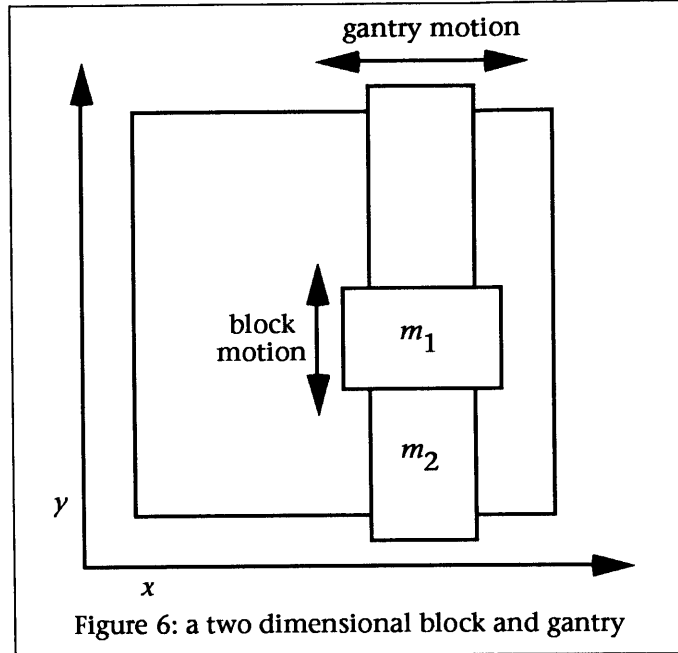


Figure 6: a two dimensional block and gantry

$$a_x = \frac{f_x}{m_1 + m_2} \quad \text{Equation 5.}$$

These two equations can be combined into a single matrix equation:

$$\begin{bmatrix} f_x \\ f_y \end{bmatrix} = \begin{bmatrix} m_1+m_2 & 0 \\ 0 & m_1 \end{bmatrix} \begin{bmatrix} a_x \\ a_y \end{bmatrix} \quad \text{Equation 6.}$$

It can be seen that the apparent mass of the system is higher in the x direction than in the y . It is the difference in the degrees of freedom between the two bodies due to the joints which creates this effect. The mass matrix in Equation 6 is the articulated body inertia for this two dimensional system.

The full development of the ABM is left to Featherstone [10], however, the basic equations needed to implement the ABM are described in **Appendix B**. The basic operation of the algorithm progresses as follows: first the algorithm passes from the leaf bodies of the figure tree in to the root body, accumulating the articulated body inertia (\hat{I}^A) and bias forces (\hat{p}), then Equation 3 is used to compute the acceleration of the root body as in:

$$\hat{a} = (\hat{I}^A)^{-1} (\hat{f} - \hat{p}) \quad \text{Equation 7}$$

finally, the algorithm passes from the root out to the leaves computing the accelerations of the joints.

3.2. Gait Control

In order to coordinate the stepping and stance of the hexapod, a biologically-inspired approach has been used. A computational model of coupled oscillators is used to generate the overall stepping patterns. In addition, feedback via reflexes reinforces the general stepping pattern while increasing the adaptability of the gait. The coupled oscillator model is employed in this thesis because it is adaptive and robust, and because it lends itself well to hierarchical structuring. The higher centers of control simply specify walking speed, and the gait controller selects the appropriate gait, and sends motor commands to the levels below it.

The oscillators will be modeled as pacemaker units which rhythmically trigger

an action. Each leg receives an oscillator, and the action which it triggers is stepping. The coupling between oscillators will be modeled as phase and time relationship rules which the oscillators maintain between each other. These relationships are mathematical translations of the stepping rules observed in the cockroach by Wilson, presented in the **Background and Related Work** chapter. The oscillators cycle with the same frequency, such that a master frequency is set, and all oscillators match that frequency (see Figure 7). Because of the coupling rules between oscillators (most importantly, that legs across from each other step 180° out of phase) different gaits and walking speeds result when the oscillator frequency is varied. Further details of these models will be given in the **Implementation** chapter.

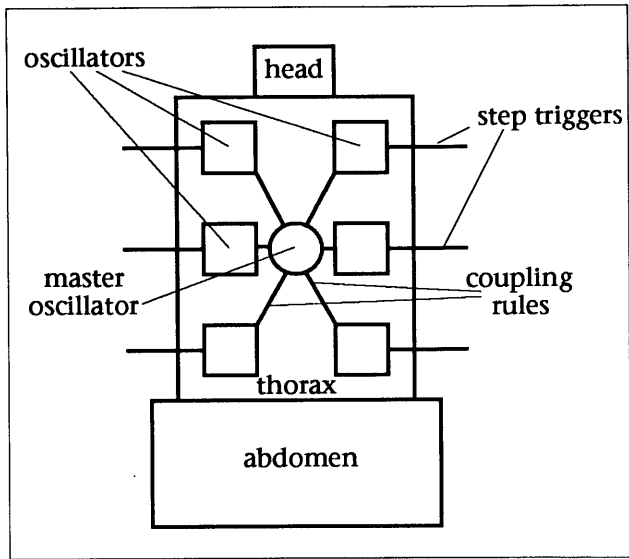


Figure 7: the coupled oscillator configuration

Reflexes are modeled as conditional units. When a certain condition is met, the reflex can inhibit or trigger different actions. For example, the step reflex triggers stepping when a leg is bent back past a specified angle. This helps the hexapod avoid over-extending the reach of its legs. An inhibitory reflex prevents stepping if an immediate neighbor (front, back and side) of a leg is already stepping. In general, adjacent legs should not step simultaneously, since it would decrease the stability of the supporting legs. An exception to this rule is seen in the locust when negotiating uneven terrain [30]. The two middle legs of the locust are lifted simultaneously, while the other legs support, in order to lift the legs out of a large depression in the terrain. A load bearing reflex inhibits stepping if a leg is currently bearing too much weight (see Figure 8). This prevents the hexapod from stepping with a leg that is currently an important support site. Again, more details on the models will be presented in the

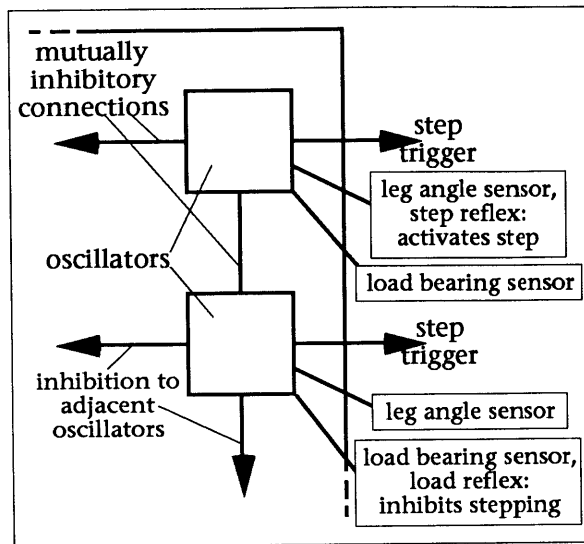


Figure 8: reflex feedback to the oscillators

Implementation chapter.

It should be noted that this modeling is not a fine-grained neural network approach as Beer, et al, have employed [34]. Instead, the oscillators and reflexes are modeled as computational units, and the oscillator coupling is modeled as rules for the oscillators units to follow.

The oscillators and reflexes trigger the stepping motor programs for the legs. One stepping is initiated, it continues to completion and stance begins again. The gait controller only generates the pattern of stepping, and is not directly responsible for the movements of the legs or body. However, the movements of the legs, due to the motor programs and dynamic simulation, do feedback into the gait controller through the reflexes.

3.3. Motor Programs

The dynamic motor programs are responsible for delivering forces, through the joints of the hexapod, to create the movements required for locomotion. There are two motor programs: step and stance. The stepping program must compute the forces necessary to lift the leg up and forward, and place it in a position to take up the load of the body when stance begins. The stance program supplies the forces needed to support the body via the legs, and propel it forward. Stepping programs are triggered by the gait controller, as described above. Stance programs automatically begin when stepping has completed.

The dynamic motor programs create forces by using exponential springs. As their name implies, These springs have an exponential relationship between the displacement, x , of the spring from its rest position (or angle) and the force

generated, f , such that:

$$f = \alpha (e^{\beta|x|} - 1) \quad \text{Equation 8}$$

where α controls linear strength, and β controls exponential rise. The force response of an exponential spring is shown in Figure 9. The exponential response creates a steep potential well; with a large displacement, the force becomes extremely high. The β parameter controls the width of the well, and the α parameter controls how fast a well of a given width will linearly rise. When an exponential spring is used for position control, the degree of freedom it controls will very likely stay within the lower parts of the well, since the forces grow so large outside of the lower region.

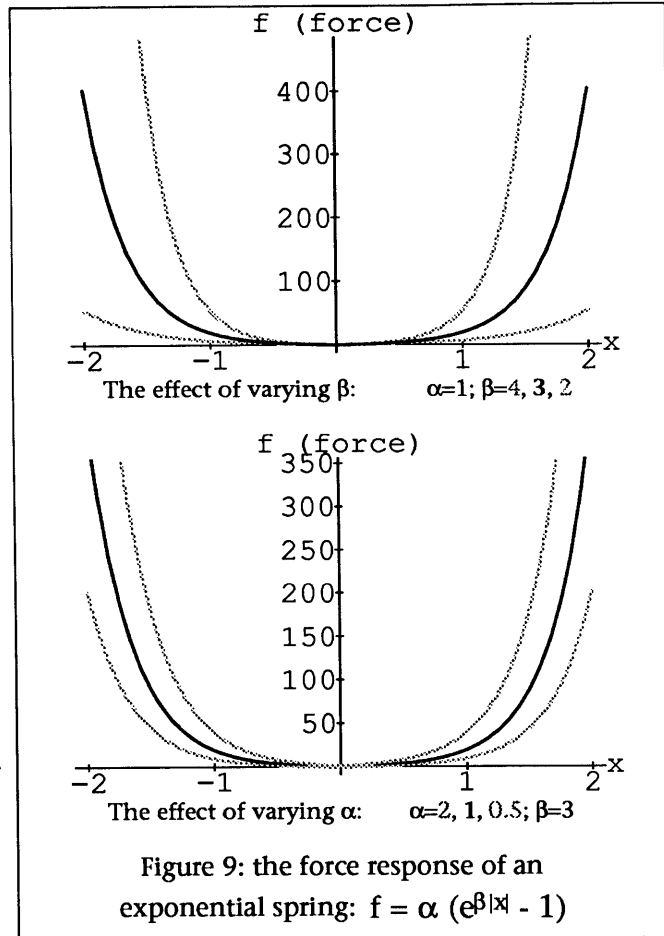


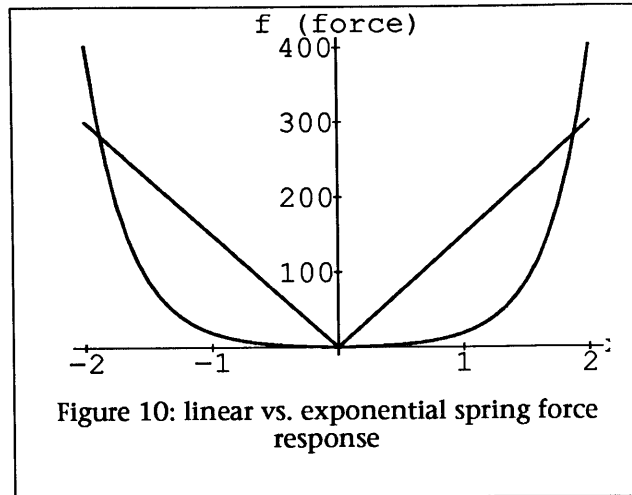
Figure 9: the force response of an exponential spring: $f = \alpha (e^{\beta|x|} - 1)$

A motor program controls motion by changing the rest position (or angle) of its exponential spring over time. This causes the force potential-well to travel along the degree of freedom, “dragging” the controlled limb along with it. The rest position is modified using a linear interpolation from the current position to the target position. In more physical terms, the rest position travels with a constant velocity to the target. This is not an exacting method of motion control, and would be inappropriate for a movement such as a controlled reach and grasp.

Although this springy method of motor control does not allow specified trajectories to be exactly executed, it does have an interesting advantage over meth-

ods which do control exact movements (especially kinematic systems). The potential wells created by the springs lead to a *compliant* system, which allows the final motion to fall within a range of possible motions. For example, to negotiate uneven terrain, a kinematic locomotion system would need to compute the leg joint angles needed to place a foot at different heights at different points on the

terrain. Using a dynamic, compliant system however, the legs can automatically adjust to the different heights of the terrain, which is demonstrated in this thesis. For example, if a foot were to land in a hole, the joint angles would be rotated beyond their normal values, but would still fall within the force potential well created by the springs. When a foot lands on a rise, the joint angles would be compressed more than usual, but the compliant response in the joints would allow this. In the hill case, the force response of the springs would lie on the opposite side of the potential well from the hole case.



A disadvantage of using exponential springs is that they create a *stiff* system. As the force response of the springs is pushed further and further up the steep walls of the potential well, the numerical sampling of the integrator must take smaller and smaller time steps to get an accurate result. Linear springs would not create such a stiff system for a given force output, but exponential springs have the advantage that at small displacements, they are *less* stiff than linear springs (see Figure 10). In addition, linear springs need to be very strong to create similar forces to the exponential springs at large displacements.

In some basic structural ways, this method of motor control is similar to the equilibrium position hypothesis, presented in the **Background and Related Work** chapter. I am not claiming to model that hypothesis, but rather claim that the gross hierarchical structuring is comparable. In the equilibrium posi-

tion hypothesis, motion is created by “interpolating,” in some manner, between different postures. This interpolation occurs in a neuromuscular controlling space. In my dynamic motor program model, different postures are “calibrated” (discussed in the **Implementation** and **Results and Analysis** chapters) and motion is achieved by interpolating between them. The controlling space of the motor programs is the rest length of the exponential-spring actuators.

3.4. Roach Description

The final component necessary for a simulation of dynamic locomotion is the set of data needed to describe the hexapod/roach model. This *model* includes not only the graphical objects used for display, but also other information specifying the mechanical design of the articulated figure, as well as data describing *how* the roach will walk. Essentially this data is a set of static information which describes the physical properties, control parameters and graphical models for the hexapod. All of this data will be detailed in the **Implementation** chapter.

The physical properties include the size, shape and density (and thus mass and inertia) of each link in the figure, as well as the figure’s mechanical design. The mechanical design describes how the figure is constructed: the joint axes’ locations, directions and degrees of freedom, and the branching tree structure of the figure. This kinematic design is based on studies of cockroach physiology, from diagrams and text descriptions [22; 56].

The control parameters determine the basic gait features and the motor program parameters. Constant gait features are the stepping speed, the time between stepping of adjacent legs on the same side of the body, the number of legs and default values for other parameters such as oscillator frequency. The motor programs for step and stance define what joint angles are traversed by the exponential springs during those actions. These programs are “tuned” via a trial-and-error method to determine appropriate spring strengths and joint angle values. In general, this trial-and-error approach is not the appropriate method to determine the operating dynamic parameters, since it requires an “expert” tuner (i.e.

the programmer) to make “educated” guesses as to the parameters, based on the experience gained from previous experiments. In some sense, the expert tuner acts as natural selection in an “evolutionary” process which increases the robustness of the locomotion. Other methods for determining the motor parameters will be discussed in the **Future Work** section.

The same data that is used for the computation of the physical inertia (the size and shape of the body parts) is used as the geometric data for graphical rendering. In addition, rendering information such as the color, shading parameters and lighting models is specified for all of the graphical objects.

4. Implementation: *Corpus*

In order to implement the methods described in the **Approach** chapter, the program *corpus* was designed and programmed by the author. *Corpus* is a system for simulating the forward dynamics of articulated figures, with gait control mechanisms, force-producing motor programs and rendering support. A “corpus” is the body of a man or animal (Webster’s 7th), and thus the articulated-figure simulator, *corpus* is so-named. *Corpus* is implemented in the c programming language, and uses several support libraries, also implemented in c: *rendermatic* (David Chen, MIT), a rendering library with geometric collision detection; *retepmatic* (Peter Schröder, MIT), additional rendering code; and *robotlib* (David Chen, MIT), a matrix manipulation package. The control of *corpus* is implemented through a scripting command language, which allows figures to be constructed, dynamically simulated, gait controlled, motor controlled and rendered.

The data which describes the mechanical structure of the hexapod and the specifics of how the hexapod will walk (gait and motor parameters) form the remaining parts of the implementation of this thesis. Although not directly a part of *corpus*, this data describes the operating parameters for the program execution.

4.1. Dynamic Simulation

Corpus employs the Articulated Body Method (ABM) described in the **Approach** chapter and **Appendix B**. Underlying the ABM is a set of spatial algebra functions (see **Appendix A** for a discussion of spatial algebra). These functions include:

- spatial operators (e.g. spatial transpose, spatial cross),
- spatial transformation matrix operators,
- and operators used to build spatial quantities from more intuitive data and to recover that data from spatial quantities.

These latter operators include functions such as:

- build a spatial force from a linear force applied at a point,
- extract the linear velocity of a point from the spatial velocity,
- and build a spatial joint axis from the axis direction and location, and joint type.

The dynamics code is a straightforward implementation of the ABM, computed in body-local coordinates. Computing in body-local coordinates instead of world space (or the *inertial frame*) has several advantages. Local coordinate computation allows certain optimizations to be made to the code [10]. These optimizations have not, as yet, been done in *corpus*, but the transformation code which would be modified is localized to allow such operations to be easily made. A problem with inertial coordinates, discovered by the author, is that they increase numerical noise. This is discussed further in the **Integration** section, below. Lastly, local coordinates can provide, at times, more intuitive values for spatial quantities; for example, the joint axis direction would constantly change in inertia coordinates, but is a constant in body-local coordinates.

Corpus dynamics operate on articulated figures with branching structure, without kinematic loops. Loops can be simulated using forces, which does not provide exact closures (there are small gaps of variable size between the links), and can lead to a stiff system. The links in the articulated figure are connected by one degree of freedom joints. The joints can be rotary or sliding (prismatic). Multiple degree of freedom joints and different joint types are supported by the underlying algorithm, and only simple changes to the joint data structure are needed to implement this.

External forces and joint forces are specified before the dynamics algorithm begins, using functions described below. Forces generators include gravity, collisions, motor program springs, joint dampers, etc. The dynamics algorithm then passes from the leaves bodies to the root body, summing articulated-body inertia and bias forces, then progresses from the root body to the leaves, solving

for accelerations.

4.2. Integration

Once the accelerations have been computed by the dynamics algorithms, they must be numerically integrated to compute velocities and positions. *Corpus* supports three methods for integration: Euler; fourth order Runge-Kutta; and fifth order, adaptive Runge-Kutta. The most basic Euler method would simply increment the integrand by the derivative times the time step, as in:

$$v_{\text{new}} = v_{\text{old}} + a \, dt \quad \text{Equation 9}$$

and

$$p_{\text{new}} = p_{\text{old}} + v_{\text{new}} \, dt \quad \text{Equation 10.}$$

This assumes constant acceleration and velocity over the time step. Furthermore, the term due to acceleration is not included in the position (p) computation. A more accurate method for Euler integration assumes constant acceleration over the time step, but uses an average of the new and old velocity to compute the new position. This accounts (in approximation) for the fact that velocity is, in fact, changing over the time interval. Also, the term due to acceleration (by doubly integrating acceleration) should be accounted for in the position computation. In the more accurate Euler integration, Equation 9 remains the same, but Equation 10 is rewritten as:

$$p_{\text{new}} = p_{\text{old}} + \frac{(v_{\text{new}} + v_{\text{old}})}{2} \, dt + \frac{a}{2} \, dt^2 \quad \text{Equation 11.}$$

This is the Euler method employed in *corpus*. For each joint, the one DOF acceleration is integrated to joint velocity and position. The root spatial acceleration is component-wise integrated to a spatial velocity, and then to spatial “position” [6]. This position is used to compute the incremental transformation which updates the local coordinate frame of the root body. The transform is constructed by forming a spatial transformation matrix which first rotates around the 3D axis specified by the upper three elements of the spatial position by the angle specified by the axis magnitude, then translates by the lower three elements. Since the local frame continuously updates to follow the moving body, the spatial position can be discarded after the new transform is made.

Euler integration allows for very rapid execution of the simulation, since only one call to the dynamics is required per time step. Also, since Euler integration is so simple, makes only one call to the dynamics, and does not require any non-integrated state to be maintained (discussed below), it can be very useful for de-bugging code. However, Euler integration can become unstable very rapidly, and the simple system outlined above does not adjust the time step for varying stability conditions. A more stable integration method is Runge-Kutta, in which multiple sub-steps are taken for each desired time step. A fourth-order Runge-Kutta has been implemented in *corpus*, as described in Press [16]. In this method, four sub-steps are taken per time step and the results are averaged with different weights for the different sub-steps. Because this integration is much more accurate than Euler (fourth order vs. first order), it is possible to take larger time steps and still remain stable.

The most accurate and robust integration method in *corpus* is an adaptive fifth order Runge-Kutta. The implementation uses the basic structure from Press [16], but the way in which the sub-steps are laid out in time, and the coefficients used to weigh the results are due to Fehlberg [17]. This method takes six sub-steps and weighs the results in different manners to compute both a fourth and fifth order solution. The two solutions are compared, and the difference is used as a gauge of the accuracy of the fifth order solution. If the difference is beyond a designated value, then the algorithm backs up to the beginning of the time step and restarts with a smaller step size (selected using the accuracy estimation).

Because the integrator must be able to back up in time in order to adapt to less stable conditions, the state (positions and velocities) data must be backed up also. This is trivial, since the integrator is initially passed this information. However, the problem arises that time-varying state data which is not integrated must also be backed up. This non-integrated state includes such parameters as motor program state, breakable springy connections to the world or a body, and the root body's transformation matrix. The integrator structure was modified to support the storage of this state, which must be passed to the dynamics algo-

rithm.

Integration of spatial quantities presents a special problem when inertial coordinates are used instead of local coordinates. Due to the nature of spatial algebra, rotational values are specified with regard to the origin of the coordinate frame which they are expressed in. Other methods often express rotation with respect to the axis about which the body is rotating. In spatial algebra, as a body travels further and further from the inertial origin, any “local” rotation it undergoes is expressed with respect to the origin, such that the body actually rotates far away from its correct position. Spatial algebra then corrects for this by translating the body back to the position it should occupy. Mathematically, this is completely valid. However, as the integrator attempts to “sum” this motion over an interval of time, the rotation-then-translation method becomes less and less accurate the further the body is from the inertial origin. The integrator “notices” this in its error estimation, and sub-divides more and more, until many sub-samples are needed for simply rotary motions. Integrating in local coordinates solves this problem, since the local origin is a (presumably) small distance from the point about which rotation is occurring. Note that this is only an issue for the root body, since it is the only body undergoing full 6 DOF motion. For all other bodies, only the joint motion is integrated, and joint motion is constant across coordinate frames.

4.3. Articulated figures in *corpus*

Articulated figures are “built” in *corpus* using scripting commands. An interactive, graphical method, which would supply the necessary commands to *corpus*, could be employed to construct figures. However, currently only script files are used to build figures. *Corpus* breaks down figures into *bodies*, *parts*, and *corpora*.

Bodies are the individual links in a figure which are based on graphical objects. Bodies also have a specified density, joint axis, joint type, and initial transformation, which takes them from graphical modeling space to their own local frame. Local frames are defined for non-root bodies by placing their local origin

at the location of their joint (the one which connects to their parent body). The bodies are translated, rotated and scaled using simple scripting commands, the same commands generally used in *corpus* to manipulate graphical objects. Using the object shape and specified density, the inertia tensor is automatically calculated by a function coded by Peter Schröder (MIT). This tensor is then built into a spatial inertia tensor.

Parts are used to rigidly attach extra graphical objects to bodies. They are oriented using scripting commands into the local space of the body to which they will attach. Their density is specified, and their computed inertia is added to the body.

Both bodies and parts can optionally be declared non-inertial and/or non-colliding. If they are non-inertial, then they contribute no inertia to the body. If they are non-colliding, they are not collision detected against any other object. Using combinations of these two options, parts can be added to bodies to be used as simple “bounding” objects, which have fewer polygons, but represent the general shape of the body, to speed up collision detection (discussed below). Bodies and parts are also declared as convex or concave. If a body is purely convex, collision detection can be performed more rapidly.

Bodies are attached to one another by specifying the parent and child bodies, and the transformation which places the child body in the space of the parent. This transform is again specified using the transformation commands in *corpus*. For example, if a child is attached to the end of a parent, and that end is located 5 meters along the x axis, in the parents space, then the specified transform would be “translate 5 in x” (specifically “move 5 0 0” in *corpus*).

Corpora are articulated figures (one is a “*corpus*”). *Corpora* are initialized after all parent/child connections have been specified, and a root body is designated. *Corpora* can be placed on and off the *integration list* so that they can be simulated or not when the dynamics is called. The root body can be specified as *fixed* or *moving*; when fixed the acceleration and velocity of the root are always zero.

The velocity and acceleration of the base can also be explicitly set, to produce constrained motion at the root.

4.4. External Forces

Any external force can be added to any body in a *corpus*. One of the most simple and useful is *gravity*. To add gravity, a force is applied to each body, at its center of mass, equal to the body's mass times 9.8, in the negative z direction (positive z in *corpus* is "up"). Other gravitational constants can be specified, if desired.

Other external forces applied to bodies are *collision forces*. Detection of collisions is described below. When a collision occurs, a force is applied at the point of penetration/contact. The force can be decomposed into two components, a *normal* force and a *tangential* force. The normal force pushes the point out of the penetrating surface, along the normal of contact, and the tangential forces account for surface friction. This is basically the approach outlined by Wilhelms and Barsky in [41]. The normal force can be generated by linear springs, exponential springs and dampers. In general, only exponential springs are employed, and damping effects are accounted for by applying a *coefficient of restitution* to the spring force. The action of the coefficient directly models the energy loss when an object rebounds from a collision. The coefficient ranges from zero to one, where zero is total energy loss, and one is a perfectly elastic collision. The coefficient is used when the collision point's velocity is upward, out of the surface. When this occurs, the spring force is scaled by the coefficient.

Tangential forces are also applied at the points of contact and are used in *corpus* to model sliding friction. These forces are applied in the plane of the contact (tangent to the contact normal). Several models of friction have been experimented with in *corpus*, with generally equal results. In general, only one model is employed, in which the normal force is scaled by the coefficient of friction, and is applied in a direction to oppose the tangential velocity of the contact point. This tends to bring any tangential motion to a halt, unless the force producing the motion is greater than the generated force, in which case sliding

occurs. The problem with this method is that when sliding does not occur (i.e. when the force producing tangential motion is less than the friction force) a stiff system is created. The friction force applied to prevent sliding should exactly counteract the other tangential forces, but in a strictly forward dynamic system, that force is not known. The stiff system arises as the approximate friction force over-compensates, producing motion in the opposite direction. This new motion must then be opposed by the friction forces, which again reverses the motion direction, and so on.

The most basic collision detection available in *corpus* is detecting penetration of the bounding boxes of bodies against a ground plane. This is actually sufficient for many of the walking experiments performed for this thesis, since the objects which comprise the hexapod are rectangular solids, and most of the tests were performed on level surfaces. The collision detection operates by simply testing the worldspace location of each of the eight corners of a body's graphical bounding box against a specified z value, which forms the level plane. For each corner below the z value, a spring force is applied to the body at that corner, using the distance below the z plane as the spring displacement.

Collision detection in *corpus* also supports detection of body bounding boxes against uneven terrain, parametrized as a regular "grid" of triangular polygons [33; 50; 57]. Because the grid is regularly spaced, a look-up table of the heights can be made from the triangular polygonal object. The table allows for rapid testing for points under the grid. To test a point, its x - y location is checked against the table's corresponding entries which surround that point. The table values are interpolated to get the exact height of the triangular polygon at the test point's x - y location. If the test point is below the polygon, a spring is applied as described above. The normal force direction must match the normal of the polygon, and the tangential forces must lie in the plane of the polygon. Note that body bounding box edges are not tested against the terrain edges, so the collision detection is incomplete. However, the hexapod supports itself on small bodied "feet" such that edges play very little role in its normal walking collisions.

Inter-body collision detection is also supported in *corpus*. Body bounding boxes are tested against other body bounding boxes. When a corner point is detected within another box, spring forces are applied as previously described, using the normal of the penetrated surface as the force normal. The force is applied equally and oppositely to each dynamic body. Again, edge collisions are not tested, leaving the detection incomplete. This form of detection is not used for the walking experiments.

A more complete form of collision detection is available in *corpus* through *rendermatic*, the graphical rendering library (David Chen, MIT). Using this method, arbitrarily shaped objects can be collision detected against each other, and all penetrating points and edges are detected. Collision detection can be performed more rapidly for purely convex objects, so bodies are marked as convex or concave upon their creation. The depths of the penetrations are not reported, however, so this method cannot be used with repulsive springs, which require displacement values. This method can be used with an analytic collision response, which does not require depth information. This response is supported in *corpus*, but will be described only briefly, since it is not used in the walking experiments. Basically, the response method operates using conservation of momentum principles, as described by Moore and Wilhelms in [58] and Hahn in [59]. The algorithm was reformulated in spatial algebra, and implemented for single body (non-articulated) collisions. The algorithm operates faster than the spring method, since only one time step is required to evaluate the collision, instead of the multiple sub-steps required to compute a springy elastic collision. The method can be extended to articulated bodies, using a large matrix approach, as Moore describes (Hahn's articulated body approach does not retain dynamic integrity, as the entire figure is treated as one completely rigid object), which has an efficiency of approximately $O(n^3)$. This analytic method does not work well with continuous support (as opposed to instantaneous collisions), and objects sink into the ground. Because walking involves large intervals of support, this method was not explored further.

Another external force available in *corpus* is an *attach force*. This force allows

bodies to be attached to other bodies or the ground using exponential springs. In the multiple body case, the force is applied equally and oppositely to the two bodies. The spring is allowed to break if it is stretched beyond a specified distance (which corresponds to a certain force). Instead of damping, a coefficient of restitution is specified, such that if the connection points of two bodies are moving towards each other, the spring force is scaled down by the coefficient, providing for energy loss.

4.5. Joints and Joint Forces

Internal forces in articulated figures are generated at the joints. Some of these forces are those which hold the figure together at the joints, but these are calculated implicitly as part of the ABM. Other joint forces are those forces applied in the direction (rotary or linear) of the joint. These forces are those which actively control the motion of a figure in *corpus*.

Basic parameters for a joint are its position (or angle) (\mathbf{q}), velocity ($\dot{\mathbf{q}}$), and acceleration ($\ddot{\mathbf{q}}$). Any of these parameters can be directly set through *corpus* commands, and acceleration is computed during the dynamic simulation. Most of the joint forces applied in *corpus* are parametrized by the values of these parameters.

All forces applied at the joints are summed into the total joint force, \mathbf{Q} . Damping can be applied at joints to slow their motion. The damping force is equal to the damping constant times the joint velocity. Linear and exponential springs can also be placed at joints. Linear springs have a spring constant and a rest position; the force generated is equal to the spring constant times the difference between the joint position and the rest position. Similarly, exponential springs generate a force as a function of joint displacement from the spring rest position. Exponential springs and their parameters are discussed in the **Approach** chapter. Joint limit springs are also available in *corpus*, such that spring and damper combinations are activated when the joint position exceeds a specified value. Joint limits are not employed in the walking experiments, however. Finally, any arbitrary bias (or “extra”) force can be added into \mathbf{Q} using

corpus commands.

Dynamic motor programs in *corpus* operate using the underlying joint force mechanisms described above. Motor programs are activated at a joint by specifying a target joint rest position for its exponential spring and the time to take to reach that rest position. From this information, the program computes the constant velocity required for the exponential spring to travel from its current rest position to the target position, in the time specified. The motor program passes the velocity information directly to the integrator, with the other derivatives, such that the rest position is automatically updated over time. The motor program deactivates itself when the target position has been reached.

4.6. Gait Controller

The gait controller is a straightforward implementation of the mechanisms described in the **Approach** chapter. The coupled oscillators are implemented as one master oscillator with phase shifts for the six leg oscillators. The oscillators are modeled implicitly as sine waves which trigger when they reach their peaks. Explicitly, the peaks are detected when the phase input to the sine function (which is not actually computed) passes beyond 90° , the phase at which the sine wave reaches its peak. Each frame, the phase input to the oscillators is incremented by the time step divided by the oscillator cycling time (the inverse of the oscillator frequency) times one complete cycle (360°). The gait computations are performed once per frame, not once per sub-step that the dynamics algorithm might take. Otherwise, the oscillator and gait state would have to become part of the integrator's stored "non-integrated state" (as discussed above), so that the gait information could be backed-up in time when a dynamic instability is detected.

The coupling phase offsets (from the master oscillator) are based on mathematical translations of Wilson's rules (see the **Background and Related Work** chapter). Because the wave of stepping runs from rear to front along one side of the body (rule 1), the phase offsets for the oscillators must be larger, the further back along the body. The amount of this phase offset is based on the current os-

cillator frequency, and is of constant time duration (rule 5) . The phase offset is equal to the stepping delay divided by the oscillator cycling time, times one complete phase cycle (360°). Optimally, the time offset should be equal to the stepping time, so that as soon as a leg finishes stepping, the leg in front of it will begin to step. Legs across each other step 180° out of phase (rule 2).

When an oscillator triggers, it changes the motor state of the leg it controls. Stepping is not directly triggered, because stepping can be inhibited by the load reflex, as described below. Instead, oscillators trigger the potential stepping state, which directly changes to the stepping state, if not inhibited.

The step reflex can also trigger the potential stepping state. The reflex is modeled simply as a conditional unit which triggers when the upper limb segment of the leg is bent back, relative to the body, beyond a specified angle. The implementation of the step reflex allows the programmer to specify the joint of any body as the joint to monitor. A “trigger” angle and direction is specified, and when the joint angle passes the trigger angle, in the given direction (either greater than or less than), the potential stepping state of that leg is set.

The load-bearing reflex is implemented in a similar manner. This reflex inhibits the stepping state, such that the potential stepping state cannot change to the stepping state, and stance continues. The reflex inhibits stepping when the force generated in the upper supporting joint in the leg (second joint from the body) exceeds some specific value. For each leg, the programmer specifies a body (and thus its joint) and a trigger force value. When that joint force is exceeded, stepping is inhibited. The potential stepping state does not change to stepping until the reflex is inactive.

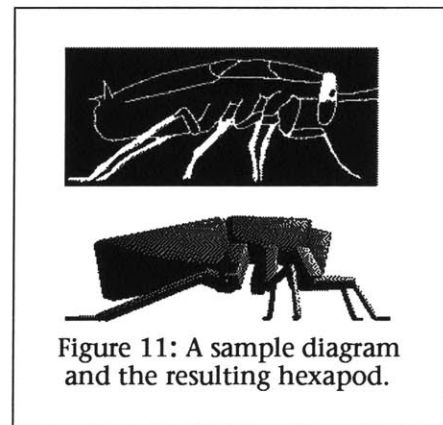
When the potential stepping state for a leg changes to stepping, the stepping motor program is activated. Stepping occurs for a specified, constant time, and when it is complete, the stance state is set and the stance motor program for that leg is activated. The high-level stepping and stance motor programs are specified for each leg as sets of *corpus* commands. In general, these commands

will be joint motor program commands, which specify target angles for exponential springs, and times to reach the targets. For each joint in the leg, a motor program is set to control its motion.

Over the time span of its activity, a higher-level motor program (such as step or stance) might require multiple lower-level motor commands for a given joint. For example, the step motor programs for the front legs first move the limbs up and forward, then down and forward. This is accomplished by using *timers*. A timer in *corpus* allows commands to be set to act when the simulation clock advances a given amount. The step motor program for the front legs, therefore, starts the initial joint motor programs and also sets timers for the second joint motor programs.

4.7. Hexapod Data

The kinematic structure of the hexapod was derived from text descriptions and diagrams from insect physiology textbooks. A set of diagrams of the insect *Blatta* from Hughes [22] was used to parametrize the sizes of the limb parts of the hexapod. A reproduction of one of the diagrams is shown in Figure 11. The lengths and widths of the limb parts were measured, and a rectangular solid was constructed to represent that limb (the width measurement from the 2D diagrams was used for two width directions in the 3D model). A further refinement in the shape of the limb and body parts, beyond the rectangular solid, was not attempted. Study instead focuses on the basic motions and physical parameters involved in locomotion.



The joint connections were established from the diagrams and text [22]. There are four segments to each leg. The uppermost (most proximal to the thorax) is called the coxa, and generally allows one DOF relative to the thorax. Continuing down the limb, the next segment is formed by the trochanter and

femur, which has a one DOF joint with the coxa (at the trochanter). The two one DOF joints (between the thorax and coxa, and between the coxa and trochanter) are oriented roughly at right angles to each other, so that the distal parts of the limb have a large range of motion. The femur attaches to the next segment, the tibia, by a one DOF joint. The most distal segment is formed by the tarsi (multiple small “foot” structures), which are more flexible than the other limb segments because of multiple joints between the femur and one another. In *corpus* the tarsi are reduced to two rigid bodies, each possessing a one DOF joint, such that two DOF motion is present at the “foot.”

The head, thorax and abdomen body parts were also parametrized from the *Blatta* diagrams. The head is jointed to the thorax so that it can turn from side to side (superfluous to locomotion). The thorax is jointed to the abdomen so that they can move up and down relative to each other. Although the connection between thorax and abdomen is not simply jointed in the insect, a rotary joint between the thorax and abdomen in the hexapod accounts, in approximation, for overall body flexibility and curvature. The overall scaling of the roach gives it a total length of approximately 2.9 cm.

The density of the body and limb parts was set to the density of water, 1 gm/cm^3 . Animal tissues in general are composed of mostly water, so this seemed a reasonable approximation. The total mass of the hexapod is 2.1 gm.

A *corpus* script which constructs the hexapod is given in **Appendix C**. The script also sets the dynamic parameters of the environment and the hexapod (for example the ground spring constants, and hexapod joint dampers), and sets typical operating parameters for the gait controller and dynamic motor programs. A partial listing of *corpus* commands (used in the script) is given in **Appendix D**.

5. Results and Analysis

5.1. Dynamic Simulator

The ABM implementation provides accurate and robust results for the simulation of many systems. For well-behaved systems (systems which are not stiff or very quickly varying), Euler integration provides stable results, and the simulations continue for an indefinite time. For more complex systems, the Runge-Kutta 5th order integrator will adapt the time-step size to the complexity of the problem, providing good results. The more complex a system is, the more “tuning” it will typically require. This tuning consists partly of:

- setting spring constants,
- creating a balance between springs and dampers,
- setting the speeds and ranges of motion for motor programs,
- setting elasticity and friction values for collisions, etc.

Even the adaptive integrator requires “tuning,” in order to set the error tolerance. Simulations with very high accelerations and velocities, such as the hexapod, can undergo much larger error values than slower moving simulations.

Careful tuning can mean the difference between a stable and unstable simulation. However, tuning is often not critical to stability, and can be used as the creative input to simulations. Physical systems typically operate “automatically” once set in operation, and tuning can be used to change their characteristics. For example, in a sequence from the animation *Grinning Evil Death* [60], 40 cereal puffs bounce about on a table. When they first drop into the scene, they have a low elasticity, and their bouncing quickly dies out. After a few moments, they are given a partially-random upward velocity, setting them again in motion. The puffs’ elasticity is then set above 1.0, such that they actually gain energy during collisions, creating a very active and lively look for the puffs.

The dynamics system has been tested successfully in a variety of fashions. Some of the tests include:

- letting a body fall freely under gravity, and verifying numerically that its acceleration is indeed 9.8m/sec^2 ,
- letting a rotating body fall freely under gravity, and numerically verifying its straight-line acceleration,
- giving a body a random velocity (linear and rotational), and verifying its straight-line motion, and constant rotational velocity. These last two tests are more robust than they may first appear, since the bodies undergo rotation in their local frames, such that their “straight-line motion” is actually the accumulation of many rotating motions in the local frame.
- comparing the motion of simple articulated bodies against the Virtual Erector Set [6], another articulated body simulator,
- comparing the motion of a body whose local frame origin has been shifted far away from the body versus a body whose local origin is at its center of mass. The motion of an origin far from the body will be greatly exaggerated during rotations of the body, verifying that complex integration of a local frame proceeds correctly.
- verifying the results of simple articulated figure tests against values attained by directly computing the motions “by hand.” For example, a three body linkage was created, with all bodies aligned along one axis. One of the end bodies was given an initial joint velocity, and all other bodies were at rest. On the first time-step, the instantaneous velocity of the end body created centripetal forces which accelerated the linkage system exactly along the axis along which all bodies were aligned, just as a “hand” analysis of such a system revealed.

In addition, many complex simulations have been computed which “look right,” revealing not that the simulations are numerically correct, but that they appear intuitively correct to the human psychophysical system. These visual verifications do not validate the motions. However, many systems which are incorrectly computed (due to an error in the dynamics code) definitely appear incorrect.

5.2. Gait Controller

The coupled oscillators produce gait stepping patterns which appear very similar to the recorded patterns of insect stepping [23; 28; 22]. The wave gait results for slow oscillator frequencies (see Figure 12 and Animation 2). The fastest allowed oscillator frequency produces the tripod gait (see Figure 13 and Animation 3). Comparisons of the stepping patterns for the tripod gait created by the computational model and by real cockroaches can be seen between Figure 13 and Figure 14. Frequencies faster than the tripod gait frequency would trigger legs to step before their neighbors had completed their stepping. Smooth changes in oscillator frequency result in smooth changes in gait (see Animation 4). As the oscillator frequency increases, the waves of stepping activity which travel up each side of the body occur with a higher frequency, and the two waves begin to overlap in time. Eventually, the wave of stepping begins at the rear while the front leg is stepping, and the two sides of the body overlap completely in time

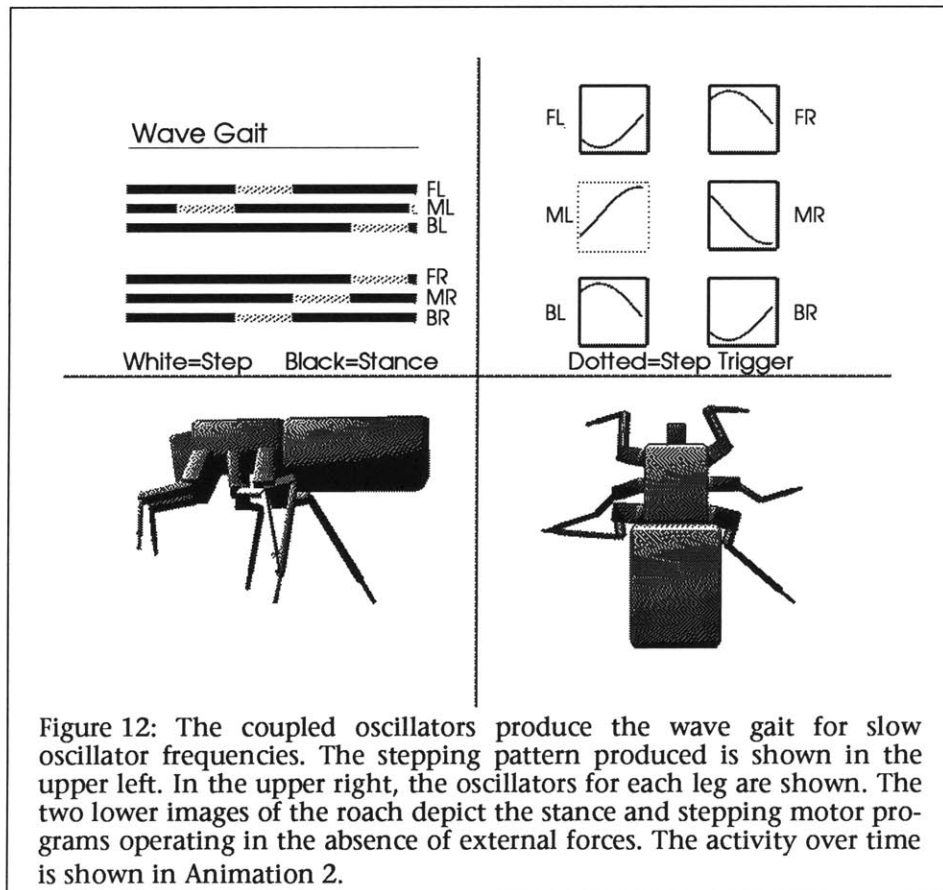
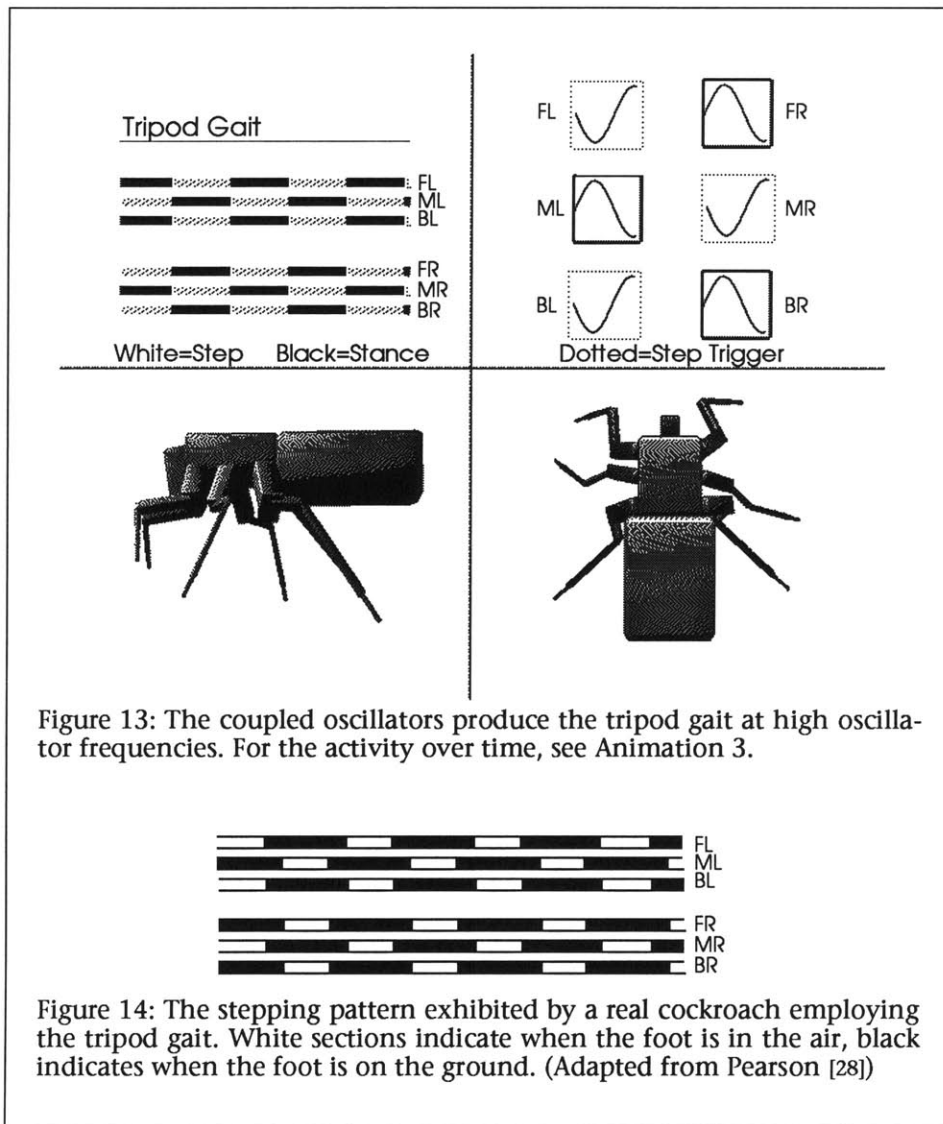


Figure 12: The coupled oscillators produce the wave gait for slow oscillator frequencies. The stepping pattern produced is shown in the upper left. In the upper right, the oscillators for each leg are shown. The two lower images of the roach depict the stance and stepping motor programs operating in the absence of external forces. The activity over time is shown in Animation 2.

(but 180° out of phase), resulting in the tripod gait. The oscillators trigger motor programs appropriately for stepping and stance.

The step reflex and load bearing reflexes function correctly, but require calibration. They should not function during undisturbed walking, but should instead reinforce stable stepping patterns under disturbances. The calibration procedure is to observe and analyze undisturbed walking, and then set reflex trigger values beyond the norm. Different leg pairs (front, middle, and back) will require different values since their ranges of motion are different, and they support different loads. This calibration has not yet been performed for the hexapod



model. However, these two reflexes have been studied for a simple kinematic hexapod [61]. The step reflex increases the robustness of the gait, especially during turning and speed changes. The load-bearing reflex (implemented in the kinematic model as a table lookup of stable stepping patterns) increases stability when limbs are missing, and prevents the step reflex from triggering a supporting leg to step. Up to this point, the thesis dynamics experiments have studied adaptive locomotion not through “nervous” or control adaptation, but through the mechanical compliance offered by the physical simulation.

5.3. Walking Experiments

Using the initial joint and spring angles established from the *Blatta* diagrams, the hexapod was “dropped” onto level ground in several dynamic simulations. The first attempts employed linear springs at the joints, and on every attempt the hexapod would collapse, as the supporting forces generated at the joints were not strong enough. Eventually, the springs were set so strong, that a very stiff system was created, and the adaptive integrator sub-divided the frame rate time step many times while the hexapod was simply falling through the air, and the spring strengths were still insufficient to support the hexapod. It was at this point that exponential springs, in place of linear springs, were introduced at the joints. On the first attempt, the exponential springs created forces sufficient to support the hexapod as it was dropped on the ground. In addition, while the hexapod was falling through the air, the integrator only slightly sub-divided the frame rate, since the exponential springs generate less force than the linear springs at low displacements (see the **Approach** chapter, especially, Figure 10 on page 31).

During the first walking experiment, the initial posture was found to be too low, and the hexapod dragged its abdomen along the ground behind it. Although the cockroach frequently drags its abdomen along the ground [22], we desired a model of locomotion in which the body was fully supported as in many other insects. Therefore the posture was raised up higher, by using joint motor programs to move the exponential spring rest angles to values which further extended the limbs. These spring angles were used as the new initial posi-

tion for further walking experiments.

Dozens of walking simulations have been executed, often successively “tuning” the action of the motor programs or other parameters. For example, the posture might be modified (as explained in the preceding paragraph), or the springs might be made stronger to better support the body, or the range of motion and timing of the stepping and stance programs might be refined. For example, the step program originally did not lift the foot fast enough or high enough to avoid dragging it along the ground for much of the stepping time, so the motor program was modified to lift the leg up higher, and more rapidly at the beginning of the step.

Figure 15 and Animation 5 show the hexapod employing the tripod gait over level terrain. The interval between steps of successive legs employed was 50 msec, compared to approximately 120 msec for the beetle *Chrysomela* which has a “relatively long” stepping interval [22]. The walking speed exhibited by the hexapod was approximately 5.5 cm/sec. Insects show a wide variety of walking speed, varying from 2.0–9.8 cm/sec in the Earwig, 3.2–17.5 cm/sec for *Blatta*, and 1.0–20.0 cm/sec for *Periplaneta* [22]. The walking speed of our simulated roach falls well within these ranges, but is considerably slower than real insects walking at their top speeds. This experiment employed a sliding model of friction with a fairly low coefficient of friction (0.7).

A different walking experiment, also employing the tripod gait, used a ground contact model in which the “feet” were modeled as having sticky pads, under active control of the hexapod, like many insects [56]. During stance, the feet would stick

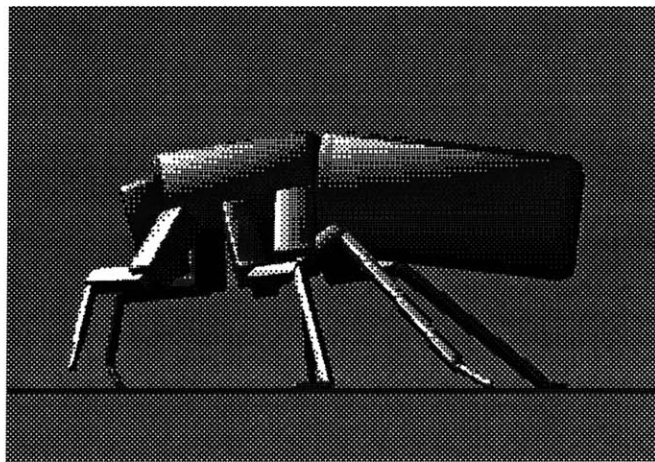


Figure 15: The hexapod employs the tripod gait over level terrain.

to the ground using exponential springs. The springs were allowed to break, if the force rose above a specified limit, allowing the feet to slide slightly and stick again. The walking speed of the hexapod increased to approximately 8.0 cm/sec, using the sticky foot model.

An interesting observation of our hexapod is that it exhibits a side-to-side “wobble” as it progresses forward, using the tripod gait. In fact, the same sort of zig-zag path is seen in real insects [56]. The phenomenon can be explained when the propulsive forces are analyzed. The front supporting leg acts as a tractor, pulling the center of mass forward, and towards the point of support. The rear supporting leg (on the same side of the body as the front support) pushes the body forward, and produces turning forces in the direction of either “left” or “right,” depending whether the line of force produced by the limb (the line passing from the point of contact on the ground and the point of contact with the body) passes in front of or behind the body’s center of mass. At the beginning of stance, the rear leg will tend to rotate the body in the same direction as the front leg. As stance continues and the rear foot moves back relative to the body, the line of force produced by the leg will shift further and further forward, and its turning forces will tend toward the opposite direction. The middle supporting leg, on the opposite side of the body, serves to support that side, propel the body forward, and to counteract part of the rotary forces produced by the other two supporting limbs.

Locomotion over uneven terrain is shown in Figure 16 and Animation 7. The sticky-foot model of contact was used for this simulation, to prevent the hexapod from sliding down the hill. The hexapod adapts to the terrain purely by the mechanical compliance of the springs and dampers in the legs. The stepping and stance motor programs were not modified for the terrain; a more complete system should adapt its motor control for different environmental conditions. Certain reflexes and control strategies which insects use to adapt to uneven terrain are presented from Pearson’s work with locusts in [30]. However, it is interesting to note how dynamic simulation and mechanical compliance can lead to adaptive behavior, without special planning.

The computation time involved in simulating the walking motion of the hexapod is relatively high, especially compared to kinematic models. On a Hewlett-Packard Series 9000 Model 835 (a RISC based workstation, rated at 12 MIPS) one videoframe at 1/40 real time (1/1200 sec simulation time) takes approximately 4 minutes of computation time. The dynamics algorithm is called approximately 100 times in that interval by the adaptive step-size integrator. A simple kinematic model of the hexapod [61] operates in real time, but has fewer degrees of freedom (20 DOF vs. 38 DOF) and does not display complex, realistic motion. The dynamics code does not currently take advantage of several numerical optimizations, which could increase speed by an order of magnitude. In addition, a stiff-system integrator could increase speed greatly by saving many calls to the dynamics algorithm [18].

5.4. Computer Animation and Dynamic Locomotion

The accompanying VHS videotape shows several simulations of the hexapod. Animation 1 shows the hexapod falling under gravity to the ground. Supporting

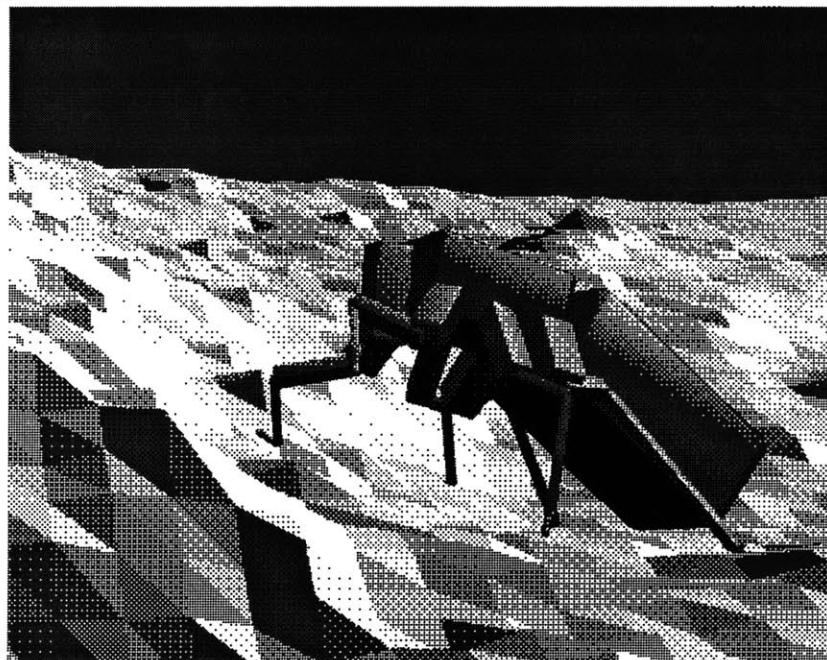


Figure 16: Locomotion over uneven terrain. Mechanical compliance in the limbs allows the legs of the hexapod to adapt to the different heights of the terrain.

forces are generated at contact points with the ground using exponential springs. The hexapod supports its weight through the joints using exponential springs and dampers. By instantaneously changing the rest length of the springs, very strong, sudden joint forces are generated, causing the hexapod to jump.

Animation 2 shows an animated diagram of the wave gait. In the upper left, a stepping pattern shows the time-varying activation of the step and stance motor programs. Step is shown as white, stance as black. The upper right shows a diagram of the coupled oscillators. When stepping is triggered by the oscillator, it flashes white. On the bottom of the screen two views are shown of the hexapod executing the motor programs in the absence of external forces (i.e. no gravity and no ground). Animation 3 shows a similar diagram of the tripod gait. And Animation 4 shows a transition from a slow wave gait to the tripod gait, as the oscillator frequency increases. The time indicators (i.e. "1/20 real time") indicate playback rate, not computation time. The hexapod is modeled as being very small, and small things move relatively quickly, hence the slow playback rate.

Animation 5 shows a normal tripod gait on level terrain. A sliding model of friction was used in this simulation, with a fairly low coefficient of friction (0.7). Animation 6 also shows a tripod gait, this one employing the sticky-foot model of ground contact. Animation 7 depicts the hexapod using the wave gait over uneven terrain. Mechanical compliance allows the legs to conform to the terrain.

Creative usage of the hexapod model is shown in the computer animation *Grinning Evil Death (GED)* [60]. Sections from *GED* can be seen in Animation 8. The story concerns a giant robotic cockroach from outer space. The roach crashes its space-pod into a city, and begins to wreak havoc there. All motions of the roach were dynamically simulated, without any direct manipulations from the animator. Instead, high level walking commands were issued, and in some cases, special-purpose motor programs were scripted by the animator.

In the first animated sequence, the roach is activated in its space-pod. Mechanical compliance allows the roach to lie and stand in the curved egg shape, without carefully planning the joint angles for the legs. The activation sequence is animated by instantaneously changing the rest length of the springs at the joints, creating sudden strong forces and the resulting “twitch” of the figure.

The next scene, after the roach has exited its space-pod, shows a normal tripod gait. This sequence uses the sticky-foot model for the foot-ground contacts. The police car seen at the end of the sequence is a dynamic simulation employing the Virtual Erector Set [6; 15], an efficient dynamics simulation system.

The roach is next seen breaking through a set of high-voltage wires, in a sequence displaying interaction between different dynamic elements. The wires are constructed as dynamic linkages in *corpus*, and the loops are closed at the middle of the linkages by *attach* forces, described in the **Implementation** chapter. The loop-closure exponential springs are set to break at a particular force, and the roach is instructed to walk forward, through the wires. Collision detection is performed between the roach parts and the wire parts, and exponential springs are used to repel parts in contact. As the roach pushes through the wires, the tension at the closure points grows greater and greater, until the breakage force is exceeded, and the closure is broken. A particle system is generated at each point of contact between the roach and wires simulating sparks. Particle systems are also placed at the broken end of each wire.

The next animated sequence (skipping over several scenes from *GED*) shows the roach reacting violently to a spray of insecticide (not depicted in the test sequence). The roach is using a tripod gait in this scene, however, the middle legs are placed on backwards, so that the front and back legs push the roach forward, but the middle legs absorb that energy and push the roach backwards again, giving the roach bouncing and flailing motions.

The roach’s adverse reaction to the spray continues in the next scenes, as the roach flips over on its back. At the beginning of the flip sequence, the friction

on the ground is turned down to nearly zero, and without any traction, but with the legs striking the ground, the roach pushes itself up and over. The roach recovers from the spray, and attempts to right itself. First, the roach is programmed to use a fast tripod gait. Then motor programs are set at the abdomen-thorax joint to wrench the body back and forth. A corner of the body strikes the ground with a large velocity, and the impact forces rotate the body over.

6. Conclusion

A dynamic simulator underlies all motion in *corpus*, the dynamic locomotion system. The simulator computes the motion of articulated figures in response to applied forces, such as gravity, collision/support forces, friction, joint springs and dampers, etc. The primary role of the simulator in this thesis is to compute the physically-based motion of a simulated six-legged insect. The kinematic structure of the hexapod is based on studies of insect physiology.

To coordinate the hexapod's walking, a gait controller is used to produce stepping patterns. The gait controller is based on hypothesized control mechanisms in vertebrates and invertebrates. Coupled oscillators, with reflexive feedback produce the stepping patterns, based on a simple high level input command, the oscillator frequency. For different frequencies, different gait are produced, the same gaits exhibited by real insects. Smooth gaits changes are effected by smoothly changing the oscillator frequencies.

The motion of the limbs are controlled by dynamic motor programs. When stepping is triggered by an oscillator, the step motor program is activated. When the step of a limb is complete, the stance motor program takes over. These programs move the rest angles of exponential springs in the joints of the limbs to produce forces in order to create movements. The step program lifts the leg up and forward, and the stance program drives the leg down and back. Propulsive forces are supplied by the stance program, and the contact forces with the ground. Either friction or sticky forces at the "feet" of the hexapod ultimately supply the forces needed to drive the body forward, through the supporting limbs.

The hexapod walks with a speed comparable to real insects, and displays walking phenomena observed in insects. For example, a side-to-side wiggle is observed in the hexapod, as it uses the tripod gait, due to the dynamic nature of the propulsive forces supplied through the supporting limbs.

One of the most interesting results of this thesis is way in which dynamic compliance can be used to provide adaptability to different environmental conditions. Due to the nature of the dynamic simulation and the springyness of the articulated figure, the hexapod can automatically adapt the configuration of its limbs to different terrains, without explicit calculation of the different joint angles required. The “give” in the hexapod structure allows the figure to comply to environmental influences.

This thesis has demonstrated that a dynamic model of legged locomotion is computationally feasible. On a Hewlett-Packard Series 9000 Model 835 (a RISC based workstation, rated at 12 MIPS) one videoframe at 1/40 real time (1/1200 sec simulation time) takes approximately 4 minutes of computation time. The dynamics algorithm is called approximately 100 times in that interval by the adaptive step-size integrator. Although the computation time is somewhat slow, simulation results can be obtained on “overnight” runs of the code.

The most immediate shortcoming of this thesis is the lack of calibration for the step and load-bearing reflexes. Once these reflexes are calibrated (by examining undisturbed walking, and setting the trigger parameters beyond the range of normal motions and stresses) the adaptive effect of the reflexes can be studied, during locomotion which receives external disturbances, and during locomotion over uneven terrain.

Computation time remains a problem for the dynamic simulation in this thesis. Speed could likely be increased by using a stiff-method integrator, since fewer calls would be made to the dynamics algorithm. For some types of locomotion, such as the sticky-foot model, might be better be modeled using a constraint method, such as Lathrop’s endpoint-constraint prorogation method [14; 6]. The sticky-foot model essentially imposes a kinematic constraint on the end-effectors, and a constraint method which deals directly with this effect should reduce the stiffness of the system, and decrease computation time.

Appendix A: Spatial Algebra

This appendix gives an overview of the spatial quantities and operators required to implement the Articulated Body Method employed in this thesis. For a full tutorial in spatial algebra, however, the reader is referred to Featherstone [10]. Basically, spatial algebra is based on screw calculus [55], and uses 6 dimensional vectors to encode not only direction and magnitude (as 3 dimensional vectors do), but also position. For example, the spatial representation of a rotary joint axis specifies the axis about which rotations occur, and also the location of that axis in 3 space. Using its ability to encode direction and position, spatial algebra unifies the rotational and translational aspects of motion into single vector quantities. For example, spatial velocity contains both the angular and linear velocity, and is transformed and manipulated as a single quantity. Contrast this with more traditional dynamics formulations which treat the angular and linear velocities separately, and use different equation sets for each. It should be noted that spatial vectors are column vectors, typical to robotics notation. These vectors are the transpose of computer graphics row vectors. Similarly, matrixes (such as rotation matrixes) in spatial notation are the transpose of computer graphics style matrixes.

Before discussing spatial vectors, the three dimensional *cross operator* is introduced:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} \times = \begin{bmatrix} 0 & -z & y \\ z & 0 & -x \\ -y & x & 0 \end{bmatrix} \quad \text{Equation 12.}$$

The cross operator allows cross product operations to be folded into matrixes, as will be seen below.

The general form of a spatial vector is as follows:

$$\hat{\mathbf{a}} = \begin{bmatrix} \mathbf{a} \\ \mathbf{a}_0 \end{bmatrix} \quad \text{Equation 13}$$

where \mathbf{a} is a 3 dimensional vector which specifies a magnitude and direction, and \mathbf{a}_O specifies the location of \mathbf{a} in the following manner:

$$\mathbf{r} \times \mathbf{a} = \mathbf{a}_O \quad \text{Equation 14}$$

where \mathbf{r} is the 3 vector from the coordinate frame origin to the location of vector \mathbf{a} (the \mathbf{r} vector can actually point to any location on the line defined by the direction of \mathbf{a} , positioned in space).

The first spatial quantity we will discuss is spatial velocity. Spatial velocity contains the angular and linear components of velocity as follows:

$$\hat{\mathbf{v}}_O = \begin{bmatrix} \boldsymbol{\omega} \\ \dot{\mathbf{r}} + \mathbf{r} \times \boldsymbol{\omega} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\omega} \\ \mathbf{v}_O \end{bmatrix} \quad \text{Equation 15}$$

where $\boldsymbol{\omega}$ is the angular velocity of the body, and $\hat{\mathbf{v}}_O$ is the linear velocity of the point on the body which is traveling through the origin. This point on the body may be imaginary, if the body is not actually coincident with the origin. The subscript $_O$ denotes that the spatial value is taken with respect to the coordinate frame O . If the linear velocity of some other point in the body is known, $\hat{\mathbf{v}}_O$ may be calculated from the location of that point, given by the vector \mathbf{r} , and the linear velocity at that point, $\dot{\mathbf{r}}$, as shown in Equation 15.

Spatial acceleration contains the angular and linear accelerations of a rigid body as follows:

$$\hat{\mathbf{a}}_O = \begin{bmatrix} \dot{\boldsymbol{\omega}} \\ \ddot{\mathbf{r}} + \dot{\mathbf{r}} \times \boldsymbol{\omega} + \mathbf{r} \times \dot{\boldsymbol{\omega}} \end{bmatrix} = \begin{bmatrix} \dot{\boldsymbol{\omega}} \\ \mathbf{a}_O \end{bmatrix} \quad \text{Equation 16}$$

where $\dot{\boldsymbol{\omega}}$ is the angular acceleration of the body, and \mathbf{a}_O is the linear acceleration of the point in the body passing through the origin. If the linear acceleration ($\ddot{\mathbf{r}}$) and the linear velocity ($\dot{\mathbf{r}}$) of some other point in the body (specified by \mathbf{r}) is known, \mathbf{a}_O can be calculated as in Equation 16.

The 6x6, spatial rigid-body inertia tensor is given as:

$$\hat{\mathbf{I}}_O = \begin{bmatrix} m (\vec{PO} \times) & m \mathbf{1} \\ \mathbf{I}^* + (\vec{OP} \times) m (\vec{PO} \times) & (\vec{OP} \times) m \end{bmatrix} = \begin{bmatrix} \mathbf{H}^T & \mathbf{M} \\ \mathbf{I} & \mathbf{H} \end{bmatrix} \quad \text{Equation 17}$$

where m is the scalar mass of the body, \vec{OP} denotes the vector from the origin to the location of the point from which the 3x3 inertia tensor, \mathbf{I}^* , was derived. \vec{OP} might typically point to the center of mass of the body, where the \mathbf{I}^* calculation can be simplified for certain simple cases, such as symmetrical boxes, rods or spheres.

Spatial forces are given by:

$$\hat{\mathbf{f}} = \begin{bmatrix} \mathbf{f} \\ \mathbf{f}_O \end{bmatrix} = \begin{bmatrix} \mathbf{f} \\ \vec{OP} \times \mathbf{f} \end{bmatrix} \quad \text{Equation 18}$$

where \mathbf{f} denotes the linear force applied at point P , and \mathbf{f}_O denotes the resulting torque at the origin, due to the linear force. A pure torque applied at the origin can also be added in to \mathbf{f}_O .

Joint axes are represented in spatial notation by:

$$\hat{\mathbf{s}} = \begin{bmatrix} \mathbf{s} \\ \mathbf{s}_O \end{bmatrix} \quad \text{Equation 19.}$$

A one DOF rotary joint is represented such that \mathbf{s} defines the direction of the axis about which rotation occurs (this value should be normalized), and \mathbf{s}_O defines the location of the rotation axis where:

$$\mathbf{s}_O = \mathbf{r} \times \mathbf{s} \quad \text{Equation 20}$$

where \mathbf{r} is the vector from the origin to a point on the \mathbf{s} axis. A one DOF prismatic (sliding) joint is defined where $\mathbf{s} = \mathbf{0}$, and \mathbf{s}_O is the normalized axis along which translation will occur.

Two special operators need to be defined in spatial algebra. The first, the *spatial cross operator*, denoted by $\hat{\times}$, is used just as the regular cross operator (Equation 12), and is given by:

$$\begin{bmatrix} \mathbf{a} \\ \mathbf{b} \end{bmatrix} \hat{\times} = \begin{bmatrix} \mathbf{a} \times & \mathbf{0} \\ \mathbf{b} \times & \mathbf{a} \times \end{bmatrix} \quad \text{Equation 21.}$$

The second spatial operator is the *spatial transpose*, denoted by a superscript S , and is used in place of the normal matrix transpose operator. The spatial transpose of a spatial vector is given as:

$$\hat{\mathbf{a}}^S = \begin{bmatrix} \mathbf{a} \\ \mathbf{a}_O \end{bmatrix}^S = [\mathbf{a}_O^T \quad \mathbf{a}^T] \quad \text{Equation 22.}$$

The spatial transpose of a spatial matrix (either inertia tensor or coordinate transform) is given as:

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}^S = \begin{bmatrix} \mathbf{D}^T & \mathbf{B}^T \\ \mathbf{C}^T & \mathbf{A}^T \end{bmatrix} \quad \text{Equation 23.}$$

Spatial transformation matrixes bring spatial quantities from one coordinate frame to another, in the following manner:

$$\hat{\mathbf{a}}_P = {}^P\hat{\mathbf{X}}_O \hat{\mathbf{a}}_O \quad \text{Equation 24}$$

where the value of $\hat{\mathbf{a}}$ is transformed from coordinate frame O to P . Spatial inertia tensors require a pre- and post-multiplication with spatial transformation matrixes as follows:

$$\hat{\mathbf{I}}_P = {}^P\hat{\mathbf{X}}_O \hat{\mathbf{I}}_O {}_O\hat{\mathbf{X}}_P \quad \text{Equation 25.}$$

A translation from one coordinate frame to another is constructed as follows:

$$\hat{\mathbf{a}}_P = \begin{bmatrix} 1 & \mathbf{0} \\ (\mathbf{r} \times)^T & 1 \end{bmatrix} \hat{\mathbf{a}}_O \quad \text{Equation 26}$$

where \mathbf{r} is the translation vector \overrightarrow{OP} . A rotation matrix is formed as:

$$\hat{\mathbf{a}}_P = \begin{bmatrix} \mathbf{E} & \mathbf{0} \\ \mathbf{0} & \mathbf{E} \end{bmatrix} \hat{\mathbf{a}}_O \quad \text{Equation 27}$$

where \mathbf{E} is the 3x3 rotation matrix which takes values from frame O to P . Spatial transformation matrixes may be concatenated to form compound transforms, just as in computer graphics notation, except that pre-multiplies become post-multiplies. The spatial transformation which corresponds to a change in coordinate frames where a translation is followed by a rotation is given as:

$$\hat{a}_p = \begin{bmatrix} \mathbf{E} & \mathbf{0} \\ \mathbf{E} (\mathbf{r} \times)^T & \mathbf{E} \end{bmatrix} \hat{a}_o \quad \text{Equation 28.}$$

This corresponds to the standard computer graphics transformation matrix, where:

$$a_p = a_o \begin{bmatrix} \mathbf{E}^T & 0 \\ & 0 \\ & 0 \\ r_x \ r_y \ r_z \ 1 \end{bmatrix} \quad \text{Equation 29}$$

where \mathbf{E} and \mathbf{r} are the same as in Equation 28.

Appendix B: The Articulated Body Method for Dynamic Simulation

In spatial algebra, the equations of motion are succinctly stated. They can be derived, starting from the conservation of momentum, which is stated:

$$\hat{\mathbf{I}} \hat{\mathbf{v}} = \text{constant} \quad \text{Equation 30}$$

where $\hat{\mathbf{I}}$ is the spatial rigid-body inertia tensor, and $\hat{\mathbf{v}}$ is its spatial velocity. Taking the derivative with respect to time describes how an applied force changes the momentum:

$$\hat{\mathbf{f}} = \frac{d}{dt} (\hat{\mathbf{I}} \hat{\mathbf{v}}) = \hat{\mathbf{I}} \hat{\mathbf{a}} + (\hat{\mathbf{v}} \times) \hat{\mathbf{I}} \hat{\mathbf{v}} \quad \text{Equation 31}$$

where $\hat{\mathbf{f}}$ is the applied spatial force, and $\hat{\mathbf{a}}$ is the resulting spatial acceleration. Featherstone rewrites Equation 31 as:

$$\hat{\mathbf{f}} = \hat{\mathbf{I}} \hat{\mathbf{a}} + \hat{\mathbf{p}}^v \quad \text{Equation 32}$$

where $\hat{\mathbf{p}}^v$ is the *bias force*, which contains the velocity dependent forces of the body (centripetal and Coriolis).

The basic equation used in the ABM appears very similar to Equation 32, and is defined as:

$$\hat{\mathbf{f}} = \hat{\mathbf{I}}^A \hat{\mathbf{a}} + \hat{\mathbf{p}} \quad \text{Equation 33}$$

where $\hat{\mathbf{I}}^A$ is the *articulated body inertia*, and $\hat{\mathbf{p}}$ is the *bias force*. The articulated body inertia gives the inertia of a body directional properties, such that the *apparent* mass of a body might be different, in different directions. The bias force in Equation 33 incorporates the velocity dependent terms (as the $\hat{\mathbf{p}}^v$ term does in Equation 32) as well as the forces needed to hold the joints together, and forces transmitted through the joints. When the articulated body inertia, bias force and desired acceleration of a body is known, the force needed to create that acceleration can be found using Equation 33. The acceleration of a body can be found when $\hat{\mathbf{I}}^A$, $\hat{\mathbf{p}}$ and the externally applied force, $\hat{\mathbf{f}}$ are known using:

$$\hat{\mathbf{a}} = (\hat{\mathbf{I}}^A)^{-1} (\hat{\mathbf{f}} - \hat{\mathbf{p}}) \quad \text{Equation 34.}$$

The key to developing the ABM them is deriving the recursive equations which determine the articulated body inertias and bias forces, as is done in [52] and [10]. The final equations for the ABM, computed in body local space, is given below. Because these computations occur in body local space, all values are represented in a body's own local coordinate frame, which travels along with the body. These local frames might have their origin at the body's center of mass or the body's joint axis. Different optimizations can be made for different selections of local frames, for a discussion, see [10] and [6].

The algorithm below operates on a kinematic linkage forming a single linear chain. A simple extension to branching figures follows the equations. The bodies in the chain are numbered 0 to n , where body 0 is the *root body* and body n is the *leaf body*. Body i is any body along the linkage, from 0 to n . Body $i-1$ is termed the *parent* of body i , and body $i+1$ is named the *child* of body i . The algorithm operates basically as follows:

1. Starting with the leaf body, the articulated body inertia and bias force is "accumulated" along the chain, in to the root body.
2. The acceleration of the root body is computed using the inverse of its articulated body inertia, and its bias force.
3. The joint accelerations are computed out from the root body to the leaf body.

Four common sub-expressions, $\hat{\mathbf{c}}_i$, $\hat{\mathbf{h}}_i$, \mathbf{d}_i and \mathbf{u}_i are derived which occur in multiple locations in the ABM equations. They can be computed once, and used in multiple equations to improve efficiency. The subscript, i , indicates that the value is represented in the coordinate frame of body i . The first sub-expression

$$\hat{\mathbf{c}}_i = (\hat{\mathbf{v}}_i \times) \hat{\mathbf{s}}_i \dot{\mathbf{q}}_i \quad \text{Equation 35}$$

describes the acceleration of body i , due to its joint velocity, $\dot{\mathbf{q}}_i$. The joint position (or angle) is given as \mathbf{q} , and the joint acceleration is given as $\ddot{\mathbf{q}}$. These values are scalars for one DOF joints. For multiple DOF joints, these values must take on the dimension n , where n is the number of DOFs. Additionally, for mul-

multiple DOF joints, the body joint axis, \hat{s}_i , must have n rows, defining a *motion sub-space* [10]. The next three sub-expressions do not have physical interpretations as meaningful as the first; they are simply recurring terms in other expressions. They are given as:

$$\hat{h}_i = \hat{I}_i^A \hat{s}_i \quad \text{Equation 36}$$

$$d_i = \hat{s}_i'^S \hat{h}_i \quad \text{Equation 37}$$

$$u_i = Q_i - \hat{h}_i^S \hat{c}_i - \hat{s}_i^S \hat{p}_i \quad \text{Equation 38}$$

where Q_i is the pure force (or torque) applied at the joint. This force is due to a joint actuator, damper, spring, or any other force function. For single degree of freedom joints, Q_i is a scalar, and for multiple degree of freedom joints, Q_i takes on a dimension equal to the number of DOFs.

The velocity of body i , is defined in terms of its parent (body $i-1$) velocity and the joint velocity, \dot{q}_i :

$$\hat{v}_i = {}_i\hat{X}_{i-1} \hat{v}_{i-1} + \hat{s}_i \dot{q}_i \quad \text{Equation 39.}$$

This equation “reads” that the spatial velocity of body i is equal to the spatial velocity of the parent (transformed from its own coordinate frame into the body i frame) plus the velocity due to the joint velocity.

The motion (velocity) of body i creates forces acting on that body. This spatial force (*bias force*) is given as:

$$\hat{p}_i^v = (\hat{v}_i \times) \hat{I}_i \hat{v}_i \quad \text{Equation 40}$$

The articulated body inertia of body i is calculated from its own spatial inertia tensor, plus the articulated body inertia of its child minus the amount of inertia that cannot be “seen” through the child’s joint axis (transformed from the frame of body $i+1$ to i).

$$\hat{I}_i^A = \hat{I}_i + {}_i\hat{X}_{i+1} \left(\hat{I}_{i+1}^A - \frac{\hat{h}_{i+1} \hat{h}_{i+1}^S}{d_{i+1}} \right) {}_{i+1}\hat{X}_i, \quad (\hat{I}_n^A = \hat{I}_n) \quad \text{Equation 41}$$

The articulated body inertia of a *leaf* body at the end of a chain of length n is defined to be simply its spatial inertia.

The bias force of body i consists of its own velocity-dependent force, minus the net external force, $\hat{\mathbf{f}}_i^{ext}$, applied to that body, plus the amount of bias force of its child which is transmitted through the child's joint, plus the force required to hold the child body ($i+1$) connected to the joint, plus the forces due to the child's joint force.

$$\hat{\mathbf{p}}_i = \hat{\mathbf{p}}_i^v - \hat{\mathbf{f}}_i^{ext} + {}_i\hat{\mathbf{X}}_{i+1} (\hat{\mathbf{p}}_{i+1} + \hat{\mathbf{I}}_{i+1}^A \hat{\mathbf{c}}_{i+1} + \frac{\mathbf{u}_{i+1}}{\mathbf{d}_{i+1}} \hat{\mathbf{h}}_{i+1}), \quad (\hat{\mathbf{p}}_n = \hat{\mathbf{p}}_n^v - \hat{\mathbf{f}}_n^{ext})$$

Equation 42.

The external force could be due to gravity, global damping, a springy collision force, etc. The bias force of the leaf body (body n) is defined to be its velocity dependent force minus its externally applied force).

The spatial acceleration of body i is given as its parent acceleration, plus the acceleration due to the joint velocity, plus the joint acceleration term:

$$\hat{\mathbf{a}}_i = {}_i\hat{\mathbf{X}}_{i-1} \hat{\mathbf{a}}_{i-1} + \hat{\mathbf{c}}_i + \hat{\mathbf{s}}_i \ddot{\mathbf{q}}_i \quad \text{Equation 43.}$$

The spatial acceleration of the *root* body (body 0) is computed from the inverse of its articulated body inertia times the (negative) bias force (which includes the externally applied forces):

$$\hat{\mathbf{a}}_0 = (\hat{\mathbf{I}}_0^A)^{-1} (-\hat{\mathbf{p}}_0) \quad \text{Equation 44}$$

This solves the forward dynamics problem for the root body. Alternately, the motion of the root body can be constrained to follow a specified acceleration, simply by setting the spatial acceleration, $\hat{\mathbf{a}}_0$. The constraint force required to achieve the specified acceleration is calculated as:

$$\hat{\mathbf{f}}_0^{constraint} = \hat{\mathbf{I}}_0^A \hat{\mathbf{a}}_0 + \hat{\mathbf{p}}_0^v \quad \text{Equation 45.}$$

This solves the inverse dynamics problem for the root body *only*. The joint accelerations of all other bodies in the linkage are still determined by the applied forces. However, the joint accelerations do respond in the correct physical manner from the constrained base acceleration.

The joint acceleration of body i is determined by the parent acceleration, the joint force, the velocity-dependent acceleration (involving the articulated body inertia) and the bias force of body i .

$$\ddot{q}_i = \frac{u_i \hat{h}_i^S \hat{X}_{i-1} \hat{a}_{i-1}}{d_i} \quad \text{Equation 46}$$

To reiterate and expand upon the algorithm operation:

- 0) As an initializing step, the spatial velocity of the links is computed from the root body (whose velocity is known) out to the leaf body, using Equation 39. In addition, all transformation matrixes between links must be known, and joint forces and external forces should be established.
- 1) Beginning with the leaf body, and operating in to the root body, the articulated body inertia and bias force is computed:
 - a) the four sub-expressions are calculated (Equation 35–Equation 38).
 - b) the velocity-dependent force is calculated using Equation 40.
 - c) the articulated body inertia is calculated with Equation 41.
 - d) the bias force is computed using Equation 42.
- 2) The acceleration of the root body is computed using Equation 44.
- 3) The joint accelerations of the bodies is computed out from the root body to the leaf, using Equation 46, and the spatial acceleration is computed using Equation 43. (The spatial acceleration of body i is needed to compute the joint acceleration of body $i+1$.)

The algorithm is easily extended to handle branching kinematic structures, as all of the above equations hold true for branches. The root body becomes the body at which all branches *eventually* converge. This body can actually be any body in the structure, but it is logical to choose the most central, and/or massive body to be the root. There are multiple leaf bodies, one at the end of each branch. Articulated body inertias and bias forces simply sum at a branch node

(the parent body where two or more child branches converge). This is the only modification necessary to the algorithm as it is described in the equations above, besides a required change in the body-numbering subscript notation (i.e. the subscript i is inadequate to place a body in a branching structure).

Appendix C: Roach Construction Scripts

The following script defines the kinematic structure of the hexapod, and sets the operating parameters for the gait and dynamic motor programs.

```

# SETUP THE WORLD
addworld w

groundk 0
groundb 0
# strong exp ground
groundea 17.5
groundeB 3000
grounde .5
groundfric 0.7
groundemaxz .01

integration rkf
eps 10

# 1/2400 of a sec
dt 0.000416666666666667
h 0.000033333

# CONSTRUCT THE ROACH
addcorpus broach

# The unit_cubeb is a beveled cube,
#   size 1 in each dimension,
#   centered about the origin
get head from ../data/unit_cubeb
get thorax from ../data/unit_cubeb
get abdomen from ../data/unit_cubeb

# first make them all 1 cm long
localxforms
scale head .01 .01 .01
scale thorax .01 .01 .01
scale abdomen .01 .01 .01

# size each object
scale head 0.265 0.25 0.635
scale thorax 1.06 0.875 0.529
scale abdomen 1.63 1.25 0.706

# move joint to origin
move thorax 0.00529 0 0
move head 0.0013525 0 -.00141

# make into bodies
addbody abdomen abdomen 0 0 1 rotary 1000
addbody thorax thorax 0 1 0 rotary 1000
addbody head head 1 0 0 rotary 1000

# move to parent position
xformcenter 0 0 0
localxforms
move thorax 0.00815 0 .001

xformcenter 0 0 0
rotate head y 10
# 0.01058 = 2.0 * .0059
#   (thorax half-length)
localxforms
move head 0.01058 0 0

# create tree structure
linkbodies abdomen thorax
linkbodies thorax head

setroot abdomen

# CONSTRUCT FRONT LEFT LEG
get leg_fl0 from ../data/unit_cubeb
get leg_fl1 from ../data/unit_cubeb
get leg_fl2 from ../data/unit_cubeb
get leg_fl3 from ../data/unit_cubeb
get leg_fl4 from ../data/unit_cubeb

# leg_fl3 is small joint used for
#   making 2 dof end effector

postmult
move leg_fl0 0 .5 0
move leg_fl1 0 .5 0
move leg_fl2 0 .5 0
move leg_fl3 0 .5 0
move leg_fl4 0 .5 0

scale leg_fl0 .01 .01 .01
scale leg_fl1 .01 .01 .01
scale leg_fl2 .01 .01 .01
scale leg_fl3 .01 .01 .01
scale leg_fl4 .01 .01 .01

scale leg_fl0 0.21 0.49 0.21
scale leg_fl1 0.11 0.6 0.11
scale leg_fl2 0.053 0.60 0.053
scale leg_fl3 0.035 0.035 0.035
scale leg_fl4 0.035 0.18 0.035

addbody leg_fl0 leg_fl0 0 0 1 rotary 1000
# leg_fl0 joint gets reset in MakeJoints

```

```

addbody leg_fl1 leg_fl1 1 0 0 rotary 1000
addbody leg_fl2 leg_fl2 1 0 0 rotary 1000
addbody leg_fl3 leg_fl3 0 0 1 rotary 1000
addbody leg_fl4 leg_fl4 1 0 1 rotary 1000

rotate leg_fl0 x -45
move leg_fl0 0.009 0.0044 -0.0015
linkbodies thorax leg_fl0

rotate leg_fl1 x 90
rotate leg_fl1 y 90
move leg_fl1 0 0.0049 0
linkbodies leg_fl0 leg_fl1

rotate leg_fl2 x -90
move leg_fl2 0 0.0060 0
linkbodies leg_fl1 leg_fl2

move leg_fl3 0 0.0060 0
linkbodies leg_fl2 leg_fl3

move leg_fl4 0 0.00035 0
linkbodies leg_fl3 leg_fl4

# CONSTRUCT FRONT RIGHT LEG
get leg_fr0 from ../data/unit_cubeb
get leg_fr1 from ../data/unit_cubeb
get leg_fr2 from ../data/unit_cubeb
get leg_fr3 from ../data/unit_cubeb
get leg_fr4 from ../data/unit_cubeb

postmult
move leg_fr0 0 -.5 0
move leg_fr1 0 -.5 0
move leg_fr2 0 -.5 0
move leg_fr3 0 -.5 0
move leg_fr4 0 -.5 0

scale leg_fr0 .01 .01 .01
scale leg_fr1 .01 .01 .01
scale leg_fr2 .01 .01 .01
scale leg_fr3 .01 .01 .01
scale leg_fr4 .01 .01 .01

scale leg_fr0 0.21 0.49 0.21
scale leg_fr1 0.11 0.6 0.11
scale leg_fr2 0.053 0.60 0.053
scale leg_fr3 0.035 0.035 0.035
scale leg_fr4 0.035 0.18 0.035

addbody leg_fr0 leg_fr0 0 0 1 rotary 1000
# leg_fr0 joint gets reset in MakeJoints
addbody leg_fr1 leg_fr1 1 0 0 rotary 1000
addbody leg_fr2 leg_fr2 1 0 0 rotary 1000
addbody leg_fr3 leg_fr3 0 0 1 rotary 1000
addbody leg_fr4 leg_fr4 1 0 1 rotary 1000

rotate leg_fr0 x 45
move leg_fr0 0.009 -0.0044 -0.0015
linkbodies thorax leg_fr0

rotate leg_fr1 x -90
rotate leg_fr1 y 90
move leg_fr1 0 -0.0049 0
linkbodies leg_fr0 leg_fr1

rotate leg_fr2 x 90
move leg_fr2 0 -0.0060 0
linkbodies leg_fr1 leg_fr2

move leg_fr3 0 -0.0060 0
linkbodies leg_fr2 leg_fr3

move leg_fr4 0 -0.00035 0
linkbodies leg_fr3 leg_fr4

# CONSTRUCT MIDDLE LEFT LEG
get leg_ml0 from ../data/unit_cubeb
get leg_ml1 from ../data/unit_cubeb
get leg_ml2 from ../data/unit_cubeb
get leg_ml3 from ../data/unit_cubeb
get leg_ml4 from ../data/unit_cubeb

postmult
move leg_ml0 0 .5 0
move leg_ml1 0 .5 0
move leg_ml2 0 .5 0
move leg_ml3 0 .5 0
move leg_ml4 0 .5 0

scale leg_ml0 .01 .01 .01
scale leg_ml1 .01 .01 .01
scale leg_ml2 .01 .01 .01
scale leg_ml3 .01 .01 .01
scale leg_ml4 .01 .01 .01

scale leg_ml0 0.18 0.60 0.18
scale leg_ml1 0.071 0.42 0.071
scale leg_ml2 0.053 0.95 0.053
scale leg_ml3 0.035 0.035 0.035
scale leg_ml4 0.035 0.20 0.035

addbody leg_ml0 leg_ml0 0 0 1 rotary 1000
# leg_ml0 joint gets reset in MakeJoints
addbody leg_ml1 leg_ml1 1 0 0 rotary 1000
addbody leg_ml2 leg_ml2 1 0 0 rotary 1000
addbody leg_ml3 leg_ml3 0 0 1 rotary 1000
addbody leg_ml4 leg_ml4 1 0 1 rotary 1000

rotate leg_ml0 x -45
move leg_ml0 0.0055 0.0044 -0.0015
linkbodies thorax leg_ml0

rotate leg_ml1 x 90
rotate leg_ml1 y -45
move leg_ml1 0 0.0060 0
linkbodies leg_ml0 leg_ml1

rotate leg_ml2 x -90
move leg_ml2 0 0.0042 0

```

```

linkbodies leg_ml1 leg_ml2

move leg_ml3 0 0.0095 0
linkbodies leg_ml2 leg_ml3

move leg_ml4 0 0.00035 0
linkbodies leg_ml3 leg_ml4

# CONSTRUCT MIDDLE RIGHT LEG
get leg_mr0 from ../data/unit_cubeb
get leg_mr1 from ../data/unit_cubeb
get leg_mr2 from ../data/unit_cubeb
get leg_mr3 from ../data/unit_cubeb
get leg_mr4 from ../data/unit_cubeb

postmult
move leg_mr0 0 -.5 0
move leg_mr1 0 -.5 0
move leg_mr2 0 -.5 0
move leg_mr3 0 -.5 0
move leg_mr4 0 -.5 0

scale leg_mr0 .01 .01 .01
scale leg_mr1 .01 .01 .01
scale leg_mr2 .01 .01 .01
scale leg_mr3 .01 .01 .01
scale leg_mr4 .01 .01 .01

scale leg_mr0 0.18 0.60 0.18
scale leg_mr1 0.071 0.42 0.071
scale leg_mr2 0.053 0.95 0.053
scale leg_mr3 0.035 0.035 0.035
scale leg_mr4 0.035 0.20 0.035

addbody leg_mr0 leg_mr0 0 0 1 rotary 1000
# leg_mr0 joint gets reset in MakeJoints
addbody leg_mr1 leg_mr1 1 0 0 rotary 1000
addbody leg_mr2 leg_mr2 1 0 0 rotary 1000
addbody leg_mr3 leg_mr3 0 0 1 rotary 1000
addbody leg_mr4 leg_mr4 1 0 1 rotary 1000

rotate leg_mr0 x 45
move leg_mr0 0.0055 -0.0044 -0.0015
linkbodies thorax leg_mr0

rotate leg_mr1 x -90
rotate leg_mr1 y -45
move leg_mr1 0 -0.0060 0
linkbodies leg_mr0 leg_mr1

rotate leg_mr2 x 90
move leg_mr2 0 -0.0042 0
linkbodies leg_mr1 leg_mr2

move leg_mr3 0 -0.0095 0
linkbodies leg_mr2 leg_mr3

move leg_mr4 0 -0.00035 0
linkbodies leg_mr3 leg_mr4

```

```

# CONSTRUCT BACK LEFT LEG
get leg_bl0 from ../data/unit_cubeb
get leg_bl1 from ../data/unit_cubeb
get leg_bl2 from ../data/unit_cubeb
get leg_bl3 from ../data/unit_cubeb
get leg_bl4 from ../data/unit_cubeb

postmult
move leg_bl0 0 .5 0
move leg_bl1 0 .5 0
move leg_bl2 0 .5 0
move leg_bl3 0 .5 0
move leg_bl4 0 .5 0

scale leg_bl0 .01 .01 .01
scale leg_bl1 .01 .01 .01
scale leg_bl2 .01 .01 .01
scale leg_bl3 .01 .01 .01
scale leg_bl4 .01 .01 .01

scale leg_bl0 0.25 0.56 0.25
scale leg_bl1 0.11 0.35 0.11
scale leg_bl2 0.071 1.41 0.071
scale leg_bl3 0.035 0.035 0.035
scale leg_bl4 0.035 0.35 0.035

addbody leg_bl0 leg_bl0 0 0 1 rotary 1000
# leg_bl0 joint gets reset in MakeJoints
addbody leg_bl1 leg_bl1 1 0 0 rotary 1000
addbody leg_bl2 leg_bl2 1 0 0 rotary 1000
addbody leg_bl3 leg_bl3 0 0 1 rotary 1000
addbody leg_bl4 leg_bl4 1 0 1 rotary 1000

rotate leg_bl0 x -45
move leg_bl0 0.0015 0.0044 -0.0015
linkbodies thorax leg_bl0

rotate leg_bl1 x 90
rotate leg_bl1 y -90
move leg_bl1 0 0.0056 0
linkbodies leg_bl0 leg_bl1

rotate leg_bl2 x -90
move leg_bl2 0 0.0035 0
linkbodies leg_bl1 leg_bl2

move leg_bl3 0 0.0141 0
linkbodies leg_bl2 leg_bl3

move leg_bl4 0 0.00035 0
linkbodies leg_bl3 leg_bl4

# CONSTRUCT BACK RIGHT LEG
get leg_br0 from ../data/unit_cubeb
get leg_br1 from ../data/unit_cubeb
get leg_br2 from ../data/unit_cubeb
get leg_br3 from ../data/unit_cubeb
get leg_br4 from ../data/unit_cubeb

postmult

```

```

move leg_br0 0 -.5 0
move leg_br1 0 -.5 0
move leg_br2 0 -.5 0
move leg_br3 0 -.5 0
move leg_br4 0 -.5 0

scale leg_br0 .01 .01 .01
scale leg_br1 .01 .01 .01
scale leg_br2 .01 .01 .01
scale leg_br3 .01 .01 .01
scale leg_br4 .01 .01 .01

scale leg_br0 0.25 0.56 0.25
scale leg_br1 0.11 0.35 0.11
scale leg_br2 0.071 1.41 0.071
scale leg_br3 0.035 0.035 0.035
scale leg_br4 0.035 0.35 0.035

addbody leg_br0 leg_br0 0 0 1 rotary 1000
# leg_br0 joint gets reset in MakeJoints
addbody leg_br1 leg_br1 1 0 0 rotary 1000
addbody leg_br2 leg_br2 1 0 0 rotary 1000
addbody leg_br3 leg_br3 0 0 1 rotary 1000
addbody leg_br4 leg_br4 1 0 1 rotary 1000

rotate leg_br0 x 45
move leg_br0 0.0015 -0.0044 -0.0015
linkbodies thorax leg_br0

rotate leg_br1 x -90
rotate leg_br1 y -90
move leg_br1 0 -0.0056 0
linkbodies leg_br0 leg_br1

rotate leg_br2 x 90
move leg_br2 0 -0.0035 0
linkbodies leg_br1 leg_br2

move leg_br3 0 -0.0141 0
linkbodies leg_br2 leg_br3

move leg_br4 0 -0.00035 0
linkbodies leg_br3 leg_br4

# INITIALIZE THE CORPUS FIGURE
corpusinit
rootmotion free

# SETUP JOINT PARAMETERS
# revise leg joints using parent
# coordinate frames
setjointp leg_bl0 0 0 1 rotary
setjointp leg_br0 0 0 1 rotary
setjointp leg_ml0 0 0 1 rotary
setjointp leg_mr0 0 0 1 rotary
setjointp leg_fl0 0 0 1 rotary
setjointp leg_fr0 0 0 1 rotary

joint head Q_type 3
joint head k .1

joint head b .01
joint head k_q 0
joint head springtype mass

joint thorax Q_type 17
joint thorax e_q 0
joint thorax ea 0.024
joint thorax eB 10
joint thorax b .01
joint thorax springtype mass

joint leg_bl0 q 0.3
joint leg_bl0 Q_type 17
joint leg_bl0 springtype distalmass
joint leg_bl0 e_q 0.3
joint leg_bl0 ea 1
joint leg_bl0 eB 16
joint leg_bl0 b 0.03

joint leg_bl1 q -0.2
joint leg_bl1 e_q -0.2
joint leg_bl1 Q_type 17
joint leg_bl1 springtype distalmass
joint leg_bl1 ea 1
joint leg_bl1 eB 16
joint leg_bl1 b 0.03

joint leg_bl2 q .4
joint leg_bl2 e_q .4
joint leg_bl2 Q_type 17
joint leg_bl2 springtype distalmass
joint leg_bl2 ea 1
joint leg_bl2 eB 16
joint leg_bl2 b 0.03

joint leg_bl3 Q_type 17
joint leg_bl3 springtype distalmass
joint leg_bl3 e_q 0
joint leg_bl3 ea 0.2
joint leg_bl3 eB 8
joint leg_bl3 b 0.03

joint leg_bl4 Q_type 17
joint leg_bl4 springtype distalmass
joint leg_bl4 e_q 0
joint leg_bl4 ea 0.2
joint leg_bl4 eB 8
joint leg_bl4 b 0.03

joint leg_br0 q -.3
joint leg_br0 e_q -.3
joint leg_br0 Q_type 17
joint leg_br0 springtype distalmass
joint leg_br0 ea 1
joint leg_br0 eB 16
joint leg_br0 b 0.03

joint leg_br1 q 0.2
joint leg_br1 e_q 0.2
joint leg_br1 Q_type 17

```



```

joint leg_br1 springtype distalmass
joint leg_br1 ea 1
joint leg_br1 eB 16
joint leg_br1 b 0.03

joint leg_br2 q -.4
joint leg_br2 e_q -.4
joint leg_br2 Q_type 17
joint leg_br2 springtype distalmass
joint leg_br2 ea 1
joint leg_br2 eB 16
joint leg_br2 b 0.03

joint leg_br3 Q_type 17
joint leg_br3 springtype distalmass
joint leg_br3 e_q 0
joint leg_br3 ea 0.2
joint leg_br3 eB 8
joint leg_br3 b 0.03

joint leg_br4 Q_type 17
joint leg_br4 springtype distalmass
joint leg_br4 e_q 0
joint leg_br4 ea 0.2
joint leg_br4 eB 8
joint leg_br4 b 0.03

joint leg_ml0 q .15
joint leg_ml0 e_q .15
joint leg_ml0 Q_type 17
joint leg_ml0 springtype distalmass
joint leg_ml0 ea 1
joint leg_ml0 eB 16
joint leg_ml0 b 0.03

joint leg_ml1 q -.05
joint leg_ml1 e_q -.05
joint leg_ml1 Q_type 17
joint leg_ml1 springtype distalmass
joint leg_ml1 ea 1
joint leg_ml1 eB 16
joint leg_ml1 b 0.03

joint leg_ml2 q 0
joint leg_ml2 e_q 0
joint leg_ml2 Q_type 17
joint leg_ml2 springtype distalmass
joint leg_ml2 ea 1
joint leg_ml2 eB 16
joint leg_ml2 b 0.03

joint leg_ml3 q 0
joint leg_ml3 e_q 0
joint leg_ml3 Q_type 17
joint leg_ml3 springtype distalmass
joint leg_ml3 ea 0.2
joint leg_ml3 eB 8
joint leg_ml3 b 0.03

joint leg_ml4 q 0

```

```

joint leg_ml4 e_q 0
joint leg_ml4 Q_type 17
joint leg_ml4 springtype distalmass
joint leg_ml4 ea 0.2
joint leg_ml4 eB 8
joint leg_ml4 b 0.03

joint leg_mr0 q -.15
joint leg_mr0 e_q -.15
joint leg_mr0 Q_type 17
joint leg_mr0 springtype distalmass
joint leg_mr0 ea 1
joint leg_mr0 eB 16
joint leg_mr0 b 0.03

joint leg_mr1 q 0.5
joint leg_mr1 e_q 0.5
joint leg_mr1 Q_type 17
joint leg_mr1 springtype distalmass
joint leg_mr1 ea 1
joint leg_mr1 eB 16
joint leg_mr1 b 0.03

joint leg_mr2 q 0
joint leg_mr2 e_q 0
joint leg_mr2 Q_type 17
joint leg_mr2 springtype distalmass
joint leg_mr2 ea 1
joint leg_mr2 eB 16
joint leg_mr2 b 0.03

joint leg_mr3 q 0
joint leg_mr3 e_q 0
joint leg_mr3 Q_type 17
joint leg_mr3 springtype distalmass
joint leg_mr3 ea 0.2
joint leg_mr3 eB 8
joint leg_mr3 b 0.03

joint leg_mr4 q 0
joint leg_mr4 e_q 0
joint leg_mr4 Q_type 17
joint leg_mr4 springtype distalmass
joint leg_mr4 ea 0.2
joint leg_mr4 eB 8
joint leg_mr4 b 0.03

# new upper joint positions

joint leg_fl0 q -.45
joint leg_fl0 e_q -.45
joint leg_fl0 Q_type 17
joint leg_fl0 springtype distalmass
joint leg_fl0 ea 1
joint leg_fl0 eB 6
joint leg_fl0 b 0.03

joint leg_fl1 q 0
joint leg_fl1 e_q 0
joint leg_fl1 Q_type 17

```

```

joint leg_fl1 springtype distalmass
joint leg_fl1 ea 1
joint leg_fl1 eB 6
joint leg_fl1 b 0.03

joint leg_fl2 q 0
joint leg_fl2 e_q 0
joint leg_fl2 Q_type 17
joint leg_fl2 springtype distalmass
joint leg_fl2 ea 1
joint leg_fl2 eB 6
joint leg_fl2 b 0.03

joint leg_fl3 q 0
joint leg_fl3 e_q 0
joint leg_fl3 Q_type 17
joint leg_fl3 ea 0.2
joint leg_fl3 eB 8
joint leg_fl3 springtype distalmass
joint leg_fl3 b 0.03

joint leg_fl4 q 0
joint leg_fl4 e_q 0
joint leg_fl4 Q_type 17
joint leg_fl4 springtype distalmass
joint leg_fl4 ea 0.2
joint leg_fl4 eB 8
joint leg_fl4 b 0.03

joint leg_fr0 q .45
joint leg_fr0 e_q .45
joint leg_fr0 Q_type 17
joint leg_fr0 springtype distalmass
joint leg_fr0 ea 1
joint leg_fr0 eB 6
joint leg_fr0 b 0.03

joint leg_fr1 q 0
joint leg_fr1 e_q 0
joint leg_fr1 Q_type 17
joint leg_fr1 springtype distalmass
joint leg_fr1 ea 1
joint leg_fr1 eB 6
joint leg_fr1 b 0.03

joint leg_fr2 q 0
joint leg_fr2 e_q 0
joint leg_fr2 Q_type 17
joint leg_fr2 springtype distalmass
joint leg_fr2 ea 1
joint leg_fr2 eB 6
joint leg_fr2 b 0.03

joint leg_fr3 q 0
joint leg_fr3 e_q 0
joint leg_fr3 Q_type 17
joint leg_fr3 springtype distalmass
joint leg_fr3 ea 0.2
joint leg_fr3 eB 8
joint leg_fr3 b 0.03

```

```

joint leg_fr4 q 0
joint leg_fr4 e_q 0
joint leg_fr4 Q_type 17
joint leg_fr4 springtype distalmass
joint leg_fr4 ea 0.2
joint leg_fr4 eB 8
joint leg_fr4 b 0.03

# SETUP GAIT PARAMETERS AND MOTOR
# PROGRAMS
address broach
metabolism 0.05
speed 1000
gaitinit

procom 0
# Leg FrontLeft: leg 0
! Step: Leg FrontLeft: leg 0
motor leg_fl0 etarget -.7 *1.0
# lift leg 1/5 time
motor leg_fl1 etarget .35 *0.2
# lower leg during step - last 1/2 of
time
timer *0.5 motor leg_fl1 etarget -0.3
*0.5
motor leg_fl2 etarget -.2 *1.0
timer *0.5 !StepFLb:
timer *0.5 !motor leg_fl1 etarget -0.3
*0.5
time
.

retcom 0
! Stance: Leg FrontLeft: leg 0
motor leg_fl0 etarget -.05 *1.0
# should be a simple maintain
motor leg_fl1 etarget -0.3 *1.0
motor leg_fl2 etarget 0.1 *1.0
time
.

# Leg FrontRight: leg 1
procom 1
! Step: Leg FrontRight: leg 1
motor leg_fr0 etarget .7 *1.0
# lift leg 1/5 time
motor leg_fr1 etarget -.35 *0.2
# lower leg during step - last 1/2 of
time
timer *0.5 motor leg_fr1 etarget 0.3 *0.5
motor leg_fr2 etarget .2 *1.0
time
.

retcom 1
! Stance: Leg FrontRight: leg 1
motor leg_fr0 etarget .05 *1.0
# should be a simple maintain

```

```

motor leg_fr1 etarget 0.3 *1.0
motor leg_fr2 etarget -0.1 *1.0
time
.

# Leg MiddleLeft: leg 2
procom 2
! Step: Leg MiddleLeft: leg 2
motor leg_ml0 etarget -.25 *1.0
# lift leg in 2/5 time
motor leg_ml1 etarget 0 *0.4
# do this in last 1/2 of step
timer *0.5 motor leg_ml1 etarget -0.5
*0.5
motor leg_ml2 etarget .2 *1.0
time
.

retcom 2
!Stance: Leg MiddleLeft: leg 2
motor leg_ml0 etarget .25 *1.0
motor leg_ml1 etarget -0.9 *1.0
motor leg_ml2 etarget 0.8 *1.0
time
.

# Leg MiddleRight: leg 3
procom 3
! Step: Leg MiddleRight: leg 3
motor leg_mr0 etarget .25 *1.0
# lift leg in 2/5 time
motor leg_mr1 etarget 0 *0.4
# do this in last 1/2 of step
timer *0.5 motor leg_mr1 etarget 0.5 *0.5
motor leg_mr2 etarget -.2 *1.0
time
.

retcom 3
! Stance: Leg MiddleRight: leg 3
motor leg_mr0 etarget -.25 *1.0
motor leg_mr1 etarget 0.9 *1.0
motor leg_mr2 etarget -0.8 *1.0
time
.

# Leg BackLeft: leg 4
procom 4
! Step: Leg BackLeft: leg 4
motor leg_bl0 etarget -.1 *1.0
# lift leg in 1/3 time
motor leg_bl1 etarget 1 *0.3333333333
# descend leg in last 1/2
timer *0.5 motor leg_bl1 etarget 0.4 *0.5
motor leg_bl2 etarget -.35 *1.0
time
.

```

```

retcom 4
! Stance: Leg BackLeft: leg 4
motor leg_bl0 etarget .3 *1.0
motor leg_bl1 etarget -0.2 *1.0
motor leg_bl2 etarget 0.4 *1.0
time
.

# Leg BackRight: leg 5
procom 5
! Step: Leg BackRight: leg 5
motor leg_br0 etarget .1 *1.0
# lift leg in 1/3 time
motor leg_br1 etarget -1 *0.3333333333
# descend leg in last 1/2
time *0.5 motor leg_br1 etarget -0.4 *0.5
motor leg_br2 etarget .35 *1.0
time
.

retcom 5
! Stance: Leg BackRight: leg 5
motor leg_br0 etarget -.3 *1.0
motor leg_br1 etarget 0.2 *1.0
motor leg_br2 etarget -0.4 *1.0
time
.

```

Appendix D: *Corpus* Commands

The following is a partial listing of commands available in *corpus* to manipulate graphical objects, to create articulated figures, to set dynamic properties of the environment and of figures, and to set gait parameters.

```
# COMMENT

Transformation/Object control commands:
premult
postmult
localxforms
xformcenter x y z

get obj_name [from] instance_file_name
closeobj obj_name (kill object)

pushobj obj_name
popobj obj_name

init obj
scale obj x y z
center obj (puts centroid at origin)
move obj <obj> <x y z>
rotate obj {x|y|z} angle
rotateaxis obj x y z angle

Gait control commands:
addroach roach_name
setroach roach_name (set current roach for other commands below)
metabolism time (set's protraction, dleg, retraction, cycle time)
speed val (1.0=top speed, 2.0=1/2 speed of 1.0)
deltaspeed val
incrspeed
topspeed val
bottomspeed val
prottime time
rettime time
cycletime time
dlegtime time
gaitgo
gaitinit
procom leg_number [list of commands for protraction]
retcom leg_number [list of commands for retraction]
stepreflex {on|off}
legstepreflex leg_number body_name trigger_angle trigger_dir(`+' or `-' )
loadreflex {on|off}
legloadreflex leg_number body_name joint_force
legstatus leg_number

Dynamics:
addworld world_name
setworld world_name

alarm time command_string
```

```

timer delta_time command_string
timerflush (activate all timers which are triggered by current time)

addcorpus corpus_name
setcorpus corpus_name
corpusinit
addbody body_name instance_name joint_axis_x y z { sliding | rotary }
  density [1=inertial 1=colliding 1=convex 1=colliding_vel_normal_test]
addpart part_name inst_name body_name density
  [1=inertial 1=colliding 1=convex 1=colliding_vel_normal_test]
setroot body_name
setjointp b_name joint_axis_x y z
  (set joint axis direction in parent's frame)
bodygroundstick b_name on|off
maxv body_name max_linear max_angular
addv body_name linear_x y z angular_x y z
setrootmatrix [4 lines of 4 floats, the transform matrix for the local space]
setrootpos (harden current position of the corpus)
updatev (update all spatial velocities in the current corpus,
  useful after reading in a state file)
linkbodies parent child
integrate corpus_name [on|off]
go { # of frames }
motor body { etarget | ... } value1 value2
jointmatchexp body_name (set exp spring to the current joint angle)
joint body { q | dq | ... } value (set a joint parameter)
jointstatus body_name
rootmotion {fixed|free}
totalmass (print total mass of corpus)

dumps file_name (dump main state of corpus)
loads file_name
dumpQs file_name (dump joint force values)
dumpas file_name (dump acceleration values)
loadas file_name
dumpmotors file_name (dump motor programs state)
loadmotors file_name
dumpcontacts file_name (dump special contact data)
loadcontacts file_name
dumpcontactforces file_name
dumpmatcorpus file_name (dump script to position all objects for rendering)

iter int (loop each frame int times)
dt [time] (set time step for frame)
h [time] (set integrator time step)
eps [val] (set integrator error tolerance)
time [val] (set current time)
integration [rkfixed | rkvar | rkf | euler]

setgrav val (set gravitational acceleration value)
grav [on|off]

ground [on|off]
groundtype [flat | trigrid]
groundkrf [val] (set ground velocity-dependent reaction force)
groundk [val] (set ground linear spring const)
grounde [val] (set ground coefficient of elasticity)
groundfric [val] (set coefficient of sliding friction)
groundfric2 [val] (coefficient of friction model 2)
groundfric2k [val] (friction model 2 spring strength)

```

```
groundfric2e [val] (friction model 2 restitution val)
groundb [val] (set ground damping const)
groundea [val] (set ground exponential spring linear strength)
groundeB [val] (ground exp rise val)
groundstickeA [val] (sticky ground exp spring linear strength)
groundstickeB [val] (sticky ground exp spring rise strength)
groundstickemaxz [val] (maximum force penetration val)
groundsticke [val] (ground stick force coefficient of restitution)
groundz [val] (set world space value for z-ground plane)

collide body1 body2 (set collision detection between b1 and b2)
collision [on|off]
collisionanalysis [on|off]
collisone [val] (set collision restitution)
collisionfric [val] (sliding friction value for collisions)
collisonea [val] (exp spring linear strength)
collisoneB [val] (exp spring rise)
collisionemaxz [val] (maximum force depth value)

addrigrd grid_name instance_name

attach body1 body2 body1_x y z body2_x y z k ea eB e [break_length]
```

Bibliography

- [1] Muybridge, E. (1957). *Animals in Motion*. New York, Dover.
- [2] Alexander, R. M., N. J. Dimery and R. F. Ker. (1985). "Elastic structures in the back and their role in galloping in some mammals." *J. Zool., Lond.* (207): 467-482.
- [3] Pfeiffer, F. and B. Gebler. (1988). A Multistage-Approach to the Dynamics and Control of Elastic Robots. *IEEE International Conf. on Robotics and Automation*. 1: 2-8.
- [4] Yang, G. and M. Donath. (1988). Dynamic Model of a One-Link Robot Manipulator with Both Structural and Joint Flexibility. *IEEE Int. Conf. on Robotics and Automation*. 1: 476-481.
- [5] Barzel, R. and A. H. Barr. (1988). "Controlling Rigid Bodies with Dynamic Constraints." ACM SIGGRAPH '88 Course Notes #27: *Developments in Physically-Based Modeling*, Section E.
- [6] Schröder, P. (1990). *The Virtual Erector Set*, Master of Science Thesis, Massachusetts Institute of Technology.
- [7] Isaacs, P. M. and M. F. Cohen. (July 1987). "Controlling Dynamic Simulation with Kinematic Constraints, Behavior Functions and Inverse Dynamics." *Computer Graphics*. 21(4): 215-224.
- [8] Barzel, R. and A. H. Barr. (August 1988). "A Modeling System Based on Dynamic Constraints." *Computer Graphics*. 22(4): 179-188.
- [9] Witkin, A. and M. Kass. (August 1988). "Spacetime Constraints." *Computer Graphics*. 22(4): 159-168.
- [10] Featherstone, R. (1987). *Robot Dynamics Algorithms*. Kluwer Academic Publishers.
- [11] Walker, M. W. and D. E. Orin. (1981). Efficient dynamic computer simulation of robotic mechanisms. *Joint Automatic Contr. Conf.*
- [12] Armstrong, W. W. (1979). Recursive solution to the equations of motion of an n-link manipulator. *Proc. 5th World Congress Theory Mach. Mechanisms*, Montreal: ASME, 1343-1346.
- [13] Armstrong, W. W., M. Green and R. Lake. (June 1987). "Near-Real-Time Control of Human Figure Models." *IEEE Computer Graphics and*

Bibliography

Applications. 7(6): 52-61.

- [14] Lathrop, R. H. (1986). Constrained (Closed-Loop) Robot Simulation By Local Constraint Propagation. 1986 *IEEE Int. Conf. on Robotics and Automation*. 2: 689-694.
- [15] Schröder, P. and D. Zeltzer. (1990). The Virtual Erector Set: Dynamic Simulation with Linear Recursive Constraint Propagation. *Proc. 1990 Symposium on Interactive 3D Graphics*. Snowbird, Utah.
- [16] Press, W. H., B. P. Flannery, S. A. Teukolsky and W. T. Vetterling. (1988). *Numerical Recipes in C*. Cambridge, Cambridge University Press.
- [17] Forsythe, G. E., M. A. Malcolm and C. B. Moler. (1977). *Computer Methods for Mathematical Computations*. New Jersey, Prentice-Hall, Inc.
- [18] Green, M. (1989). Using Dynamics in Computer Animation: Control and Solution Issues. *Proc. Mechanics, Control and Animation of Articulated Figures*.
- [19] Turvey, M. T., H. L. Fitch and B. Tuller. (1982). The Problems of Degrees of Freedom and Context-Conditioned Variability. *Human Motor Behavior*. Hillsdale, New Jersey, Lawrence Erlbaum Associates.
- [20] Gallistel, C. R. (1980). *The Organization of Action: A New Synthesis*. Hillsdale, New Jersey, Lawrence Erlbaum Associates.
- [21] Hildebrand, M. (1976). Analysis of Tetrapod Gaits: General Considerations and Symmetrical Gaits. *Neural Control of Locomotion*. New York, Plenum Press.
- [22] Hughes, G. M. and P.J. Mill. (1974). Locomotion: Terrestrial. In M. Rockstein (Ed.), *The Physiology of Insecta*. New York and London: Academic Press. 335-379.
- [23] Wilson, D. M. (1966). "Insect Walking." *Annual Review of Entomology*. 11: 162-169.
- [24] Donner, M. D. (1984). *Control of Walking: Local control and real time system*, Doctoral Thesis, Carnegie-Mellon University.
- [25] Raibert, M. personal correspondence, Massachusetts Institute of Technology.

Bibliography

- [26] McMahon, T. A. (1975). "Using body size to understand the structural design of animals: quadruped locomotion." *Journal of Applied Physiology*. 39(4): 619-627.
- [27] Zeltzer, D. (in preparation). Kinematic Gait control Using Coupled Oscillators.
- [28] Pearson, K. "The Control of Walking." *Scientific American* 235.6 (December 1976): 72-86.
- [29] Holst, E. von. On the Nature of Order in the Central Nervous System. *The behavioral physiology of animals and man: Selected papers of E. von Holst*. University of Miami Press, 1973. (original publication 1937).
- [30] Pearson, K. G. and R. Franklin. (1984). "Characteristics of Leg Movements and Patterns of Coordination in Locusts Walking on Rough Terrain." *The International Journal of Robotics Research*. 3(2): 101-112.
- [31] McGhee, R. B. (1976). Robot Locomotion. *Neural Control of Locomotion*. New York, Plenum Press.
- [32] Sun, S. S. (1974). *A theoretical study of gaits for legged locomotion systems*, Ph.D. Dissertation, Ohio State University
- [33] Zeltzer, D. (August 1984). *Representation and Control of Three Dimensional Computer Animated Figures*, Ph.D. Thesis, Ohio State University.
- [34] Beer, R. D., L. S. Sterling, and H. J. Chiel. (January 1989) *Periplaneta Computatrix: The Artificial Insect Project*. Case Western Reserve University, Technical Report TR 89-102.
- [35] Chiel, H. J. and R. D. Beer. (February 1988). *A lesion study of a heterogeneous artificial neural network for hexapod locomotion*, Case Western Reserve University, Technical Report TR-108.
- [36] McMahon, T. A. (1984). *Muscles, Reflexes, and Locomotion*. Princeton University Press.
- [37] Bizzi, E., W. Chapple and N. Hogan. (1982). "Mechanical Properties of Muscle: Implications for Motor Control." *Trends in Neuroscience*. (November).
- [38] Bizzi, E. (1980). Central and peripheral mechanisms in motor control. *Tutorials in Motor Behavior*. North-Holland Publishing Co.

Bibliography

- [39] Hogan, N. (1985). "The Mechanics of Multi-Joint Posture and Movement Control." *Biol. Cybern.* (52): 315-331.
- [40] Feldman, A. G. (1986). "Once More on the Equilibrium-Point Hypothesis (λ Model) for Motor Control." *Journal of Motor Behavior.* 18(1): 17-54.
- [41] Wilhelms, J. (June 1987). "Using Dynamic Analysis for Realistic Animation of Articulated Bodies ." *IEEE Computer Graphics and Applications.* 7(6): 12-27.
- [42] An, C. H., C. G. Atkeson and J. M. Hollerbach. (1988). *Model-Based Control of a Robot Manipulator.* Cambridge, MA, MIT Press.
- [43] Brotman, L. S. and A. Netravali. (August 1988). "Motion Interpolation by Optimal Control." *Computer Graphics.* 22(4): 309-315.
- [44] Zeltzer, D., S. Pieper and D. Sturman. (1989). An Integrated Graphical Simulation Platform. *Proc. of Graphics Interface 89.* 266-274.
- [45] Liston, R. A. and R. S. Moser. (1968). A Versatile Walking Truck. *Proc. Transportation Engineering Conf,* Institution of Civil Engineers, London.
- [46] McGhee, R. B. and G. I. Iswahdhi. (April 1979). "Adaptive Locomotion of a Multilegged Robot over Rough Terrain." *IEEE Trans. on Systems, Man, and Cybernetics.* SMC-9(4): 176-182.
- [47] Sutherland, I. E. (1983). *A Walking Robot.* Pittsburgh, PA, Marcian Chronicles, Inc.
- [48] Raibert, M. H. (1986). *Legged Robots That Balance.* Cambridge, MA, MIT Press.
- [49] Girard, M. (June 1987). "Interactive Design of 3D Computer-Animated Legged Animal Motion." *IEEE Computer Graphics and Applications.* 7(6): 39-51.
- [50] Sims, K. (June 1987). *Locomotion of Jointed Figures over Complex Terrain,* M.S.V.S Thesis, Massachusetts Institute of Technology.
- [51] Miller, G. (August 1988). "The Motion Dynamics of Snakes and Worms." *Computer Graphics.* 22(4): 169-178.

Bibliography

- [52] Featherstone, R. (1983). "The Calculation of Robot Dynamics Using Articulated-Body Inertias." *Robotics Research*. 2(1): 13-29.
- [53] Deyo, R. and D. Ingebretsen. (1989). *Notes on Real-Time Vehicle Simulation*. Salt Lake City, Evan & Sutherland.
- [54] Bae, D. and E. J. Haug. (1987). "A Recursive Formulation for Constrained Mechanical System Dynamics: Part I. Open Loop Systems." *Mech. Struct. & Mach.* 15(3): 359-382.
- [55] Ball, R. S. (1900). *A treatise on the theory of screws*. London, Cambridge Univ. Press.
- [56] Wigglesworth, V. B. *The Principles of Insect Physiology*. London, Chapman and Hall.
- [57] Franklin, W. R. (June 1981). *3-D Geometric Databases Using Hierarchies of Inscribing Boxes*. Proc. Conf. Canadian Society for Man-Machine Interaction. 173-180.
- [58] Moore, M. and J. Wilhelms. (August 1988). "Collision Detection and Response for Computer Animation." *Computer Graphics*. 22(4): 289-288.
- [59] Hahn, J. K. (August 1988). "Realistic Animation of Rigid Bodies." *Computer Graphics*. 22(4): 299-308.
- [60] McKenna, M. and B. Sabiston, (1990) *Grinning Evil Death*, computer animation produced at the Media Laboratory, Massachusetts Institute of Technology. Cambridge, MA.
- [61] McKenna, M. A., S. Pieper and D. Zeltzer. (1990). Control of Virtual Actor: The Roach. *Proc 1990 Symposium on Interactive 3D Graphics*, Snowbird, Utah.