NPL: A Graphical Programming Language for Motor Control and its Application to Bipedal Walking

by

Michael Alan Wessler

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2002

© Michael Alan Wessler, MMII. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author

Department of Electrical Engineering and Computer Science

May 19, 2002

Certified by..............

Gill Pratt
Associate Professor
Thesis Supervisor

Accepted by .......

Arthur C. Smith
Chairman, Department Committee on Graduate Students
NPL: A Graphical Programming Language for Motor Control and its Application to Bipedal Walking

by

Michael Alan Wessler

Submitted to the Department of Electrical Engineering and Computer Science on May 22, 2002, in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Abstract

Current methods for bipedal walking control include playback of recorded joint motion and the derivation of dynamic equations to map desired forces at the body to the required torques at the joints. Both methods require a significant amount of up-front knowledge about the structure and characteristics of the robot. This thesis presents an alternative method of control that removes the interdependence of the joint torques and simplifies the mathematics considerably. The simplification allows a programmer to create and tune a bipedal walk controller without requiring a complete model of the dynamics. The controller is implemented in a graphical programming language similar to fuzzy logic and neural networks, in which the algorithm is contained in the structure of the nodes rather than in the weights of the connections. The language and its development environment are specifically designed to assist the programmer to create and debug the algorithm in a live environment.

Thesis Supervisor: Gill Pratt
Title: Associate Professor
Acknowledgments

This thesis would not have been possible without the help and support of a number of very generous people. I first joined the Leg Lab just to be able to watch Jerry Pratt’s robot, Spring Flamingo, walk around the lab. Later, I swiped his simulation model to use as the basis of this thesis.

Jerry and other members of the Leg Lab, including Peter Dilworth, Dan Paluska, Max Berniker, Joaquin Blaya, Andreas Hofmann, Russ Tedrake, and Chee-Meng Chew, helped me understand motor control and provided excellent encouragement and sounding boards for ideas.

Most importantly, I want to thank my wife, Tina Bronkhorst, who has been a fountain of patience and understanding as I labored to finish this thesis.
# Contents

1 Introduction 13
   1.1 Thesis .............................................. 14
       1.1.1 Characteristics of NPL and the development environment .... 15
       1.1.2 Characteristics of the walking controller ..................... 17
   1.2 Motivation ............................................. 19
   1.3 Overview .............................................. 21

2 Background 23
   2.1 Robot Structure ....................................... 23
       2.1.1 Terms ........................................... 24
   2.2 Other Control Work .................................... 25
       2.2.1 ZMP and Trajectory Following ............................ 25
       2.2.2 Natural dynamics .................................. 27
       2.2.3 Virtual model control ............................... 27
       2.2.4 Mathematical analysis .............................. 28
       2.2.5 Learning algorithms ................................ 30
       2.2.6 Constructed networks ............................... 31
   2.3 Discussion ............................................. 33

3 The Neuronal Programming Language (NPL) 35
   3.1 Building blocks ...................................... 36
   3.2 Construction Theory ................................... 40
       3.2.1 Digital idioms .................................. 40
3.2.2 Analog idioms ................................................. 43
3.3 Development environment .................................... 49
3.4 Summary .......................................................... 51

4 The swing-up and balance task ................................ 53
4.1 Finding subtasks .................................................. 54
  4.1.1 The swingup task ........................................ 54
  4.1.2 The balance task ......................................... 55
4.2 Creating the controller in NPL ................................ 56
  4.2.1 Inputs and outputs ......................................... 56
  4.2.2 The swingup subcontroller ................................ 58
  4.2.3 The balance subcontroller .............................. 60
4.3 Debugging and tuning the controller ....................... 61
  4.3.1 Startup .................................................... 61
  4.3.2 Overdrive ................................................ 63
  4.3.3 Canceling drift ......................................... 64
4.4 Summary .......................................................... 66

5 Structure of the Bipedal Walk Controller .................. 69
5.1 The walk cycle .................................................. 70
5.2 The inverted pendulum model ............................ 71
5.3 Bipedal walking controller structure ...................... 73
  5.3.1 Inputs and coordinate transform ...................... 73
  5.3.2 State machine ........................................... 75
  5.3.3 Stance phase: Height and pitch maintenance .......... 77
  5.3.4 Swing phase: balance and control .................... 80
5.4 Tuning .......................................................... 85
  5.4.1 Gait parameters .......................................... 85
  5.4.2 Stability parameters .................................... 86
  5.4.3 The walking gait ......................................... 88
5.5 Additions ........................................................ 88
6 Analysis

6.1 Walking Gait
   6.1.1 Ankle torque output
   6.1.2 Knee torque output
   6.1.3 Hip torque output
   6.1.4 Gait Stability
   6.1.5 Noise

6.2 Effects of changes in selected parameters
   6.2.1 Velocity: the toeoff parameter
   6.2.2 Stability: step length and swing speed parameters

6.3 Running

6.4 NPL
   6.4.1 Building block decisions
   6.4.2 Programming style

6.5 Summary

7 Conclusion and Future Work

7.1 Future work
   7.1.1 The walking algorithm
   7.1.2 NPL

7.2 Conclusion
List of Figures

2-1 Structure of Spring Flamingo ........................................... 23

3-1 Neuronal Programming Language node types ...................... 36
3-2 NPL versions of the NOT, NOR and AND gates .................... 40
3-3 NPL flip-flop ............................................................ 41
3-4 An NPL state machine with four states ............................ 42
3-5 NPL proportional controller, and proportional controller with inhibitor 43
3-6 Function plots of the discriminator ................................ 44
3-7 PD controllers .......................................................... 45
3-8 Low pass filter .......................................................... 46
3-9 Oscillator and phase-lock loop ..................................... 47
3-10 Frequency and phase response of the phase lock loop ......... 49
3-11 Client windows ......................................................... 50

4-1 Schematic of the pole/cart system for the swingup and balance task . 53
4-2 Pumping energy into the pendulum .................................. 54
4-3 Balancing the pendulum ................................................. 55
4-4 Two wire encoding of an input variable ............................ 56
4-5 Gaussian encoding of an angle into eight segments ............... 57
4-6 Output decoding in NPL ................................................ 57
4-7 The swingup subcontroller .............................................. 59
4-8 The balance subcontroller .............................................. 60
4-9 Adjustments to the swingup controller ............................. 62
4-10 Hierarchical controller for velocity control once the pendulum is balanced 65
<table>
<thead>
<tr>
<th></th>
<th>Recorded data for a run of the swingup task.</th>
<th>66</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-1</td>
<td>Kajita inverted pendulum model</td>
<td>71</td>
</tr>
<tr>
<td>5-2</td>
<td>Transformation from input angles to horizontal and vertical displacements between the ankle and the hip.</td>
<td>74</td>
</tr>
<tr>
<td>5-3</td>
<td>Walking controller state machine</td>
<td>75</td>
</tr>
<tr>
<td>5-4</td>
<td>Hip control during stance phase</td>
<td>77</td>
</tr>
<tr>
<td>5-5</td>
<td>Knee control during stance phase</td>
<td>79</td>
</tr>
<tr>
<td>5-6</td>
<td>Ankle controller to keep the foot level</td>
<td>81</td>
</tr>
<tr>
<td>5-7</td>
<td>Knee controller during swing phase</td>
<td>82</td>
</tr>
<tr>
<td>5-8</td>
<td>Hip controller during swing phase</td>
<td>84</td>
</tr>
<tr>
<td>5-9</td>
<td>Stroboscopic representation of the gait produced by the NPL algorithm</td>
<td>88</td>
</tr>
<tr>
<td>5-10</td>
<td>Ankle controllers</td>
<td>90</td>
</tr>
<tr>
<td>6-1</td>
<td>Plot of the walking cycle</td>
<td>96</td>
</tr>
<tr>
<td>6-2</td>
<td>Combination plots of various joints. Initial conditions (marked with an x) quickly converge to the steady state walking cycle (red).</td>
<td>98</td>
</tr>
<tr>
<td>6-3</td>
<td>Limits to recoverable disturbances in different phases of a step</td>
<td>99</td>
</tr>
<tr>
<td>6-4</td>
<td>Response of the controller to a massive push from behind</td>
<td>100</td>
</tr>
<tr>
<td>6-5</td>
<td>Response of the controller to an impulse from above</td>
<td>101</td>
</tr>
<tr>
<td>6-6</td>
<td>Plot of step length and step time versus settings of toeoff parameter</td>
<td>103</td>
</tr>
<tr>
<td>6-7</td>
<td>Stable parameter values</td>
<td>105</td>
</tr>
<tr>
<td>6-8</td>
<td>Total contribution of swing speed parameters plotted against total contribution of step length parameters for stable walks.</td>
<td>106</td>
</tr>
<tr>
<td>6-9</td>
<td>Groucho running</td>
<td>107</td>
</tr>
<tr>
<td>6-10</td>
<td>NPL structures to enable walking and running transitions</td>
<td>110</td>
</tr>
<tr>
<td>6-11</td>
<td>Automatic transitions from walking to running and back</td>
<td>111</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Controlling a bipedal walker is a complex task. The controller needs to coordinate a large number of concurrent tasks into a coherent whole. Controllers are easiest to design when the controlled system has a simple continuous mathematical description for its dynamics. Portions of the walk cycle do fit this description. However, discontinuities at heel strike and toe-off change the dynamics of the leg and require delicate transitions from one set of functions to another.

There are several paths a controller designer can take to accommodate the different modes of motion and control during a single walk cycle. One path requires a complex mathematical formulation that encompasses all of the discontinuities and differences in impedances. Another path incorporates some kind of training or reinforcement algorithm to learn the correct controls to apply at any given time. A third path incorporates several simple controllers and contains a method to choose the appropriate subset of controllers to apply at any given time. Many techniques are hybrid controllers, applying portions of each of these paths.

The first, monolithic method requires a significant amount of work and complexity up front. A complete mathematical model may be too complex or may even be intractable. The onus is then on the designer to develop simplifications that make the model more amenable to a mathematical solution without breaking the correspondence to the real world.

The learning method has different requirements depending on the type of learning
being performed. In the case of back-propagation style learning, the correct set of motor commands must be known for every state of the system, or at least those states likely to be reached by a walking robot. The training must include not only the desired trajectory, but also enough deviations from that trajectory so that the system can recover from disturbances. For reinforcement style learning, the controller must be simple enough, or have few enough knobs that the exploration space is small enough to be tractable.

The third method is closer to the concept of programming. Instead of developing a monolithic mathematical formula, a programmer typically breaks a problem down into simpler chunks or functional blocks that can be solved easily, then connects the chunks together to form the complete solution. In the motor control domain, a programmer would break the control problem into small sub-control problems that can be combined to produce the full controller. Such a method is easier to design and implement than a monolithic one, but may be harder to prove correct.

1.1 Thesis

This thesis presents two contributions. The primary contribution is a new way of thinking about motor control for bipedal walking. It builds on the method of breaking down the problem into tasks that can be assigned to sub-controllers. The new idea is to have a one-to-one correspondence between joints and tasks, so that each joint has a negative feedback controller that does not interfere, or at least interferes very little, with the task of any other joint. The result is a very stable walking gait that recovers very well from disturbances to the system. The exact breakdown of the walking problem into independent subtasks is described in chapter 5.

The second contribution of this thesis is a new programming environment for developing motor control algorithms. The environment includes a visual programming language that captures some of the capabilities of biological neurons. The language is called Neuronal Programming Language, or NPL. NPL was designed to bring the rapid prototyping concept to motor control programming. One develops programs in
NPL by dragging small boxes that represent the computational units of the language into a drawing area, and drawing connections between them. The environment allows live editing of the algorithm while the simulation is running. The programmer can thus instantly see the effects of any changes made to the program.

1.1.1 Characteristics of NPL and the development environment

The need for a new programming environment grew out of frustration with traditional programming tools in the development of motor control algorithms. The ideal environment for algorithm exploration would allow rapid feedback in a live environment. High level compiled languages had a lengthy turnaround time, even for small changes. In addition, the algorithms were opaque; without debugging tools, it was unclear what any given part of an algorithm was doing at a particular instant. Even with debugging tools, it was impossible to interpret the interactions between the parts of an algorithm over time. NPL was developed to overcome these difficulties, and has the following characteristics:

- **Live development environment**

  The exploration of new ideas for a walking control algorithm requires a tremendous amount of trial and error. NPL drastically cuts down on the development cycle through the use of a live development environment. The programming language is connected at all times to the simulation of the robot. Changes to the program can be made at any time, even while the algorithm is running. The environment allows the programmer to instantly see the results of a change on the screen as a change in behavior of the robot. This is especially important when tuning controller parameters, where a large number of trials need to be conducted as quickly as possible.

- **Visual programming language**

  Part of the difficulty with traditional languages is that complex algorithms are
opaque. Apart from the inputs, outputs, and any debugging variables, it is not possible to watch the internal computation of the algorithm. While it is possible to record the states and play them back later, the rapid prototype paradigm suggests that the values be available while the program is running. NPL allows visibility of the entire computation by drawing the algorithm as a set of small boxes. Colored bars of various lengths indicate internal state values. Debugging tables or lists of values can be useful, but the human visual system is much better at processing colors and sizes than text. If the controller program is arranged efficiently on the screen, the programmer can watch as subcontrollers activate and deactivate at the appropriate times.

Note that the visual quality is dictated by the development environment, and not by the language itself. In fact, a perfectly good description of a program would be the file format in which the visual representation is saved, in this case, XML. One can edit the XML either directly in a text editor, or indirectly through the visual environment.

- **Explicit timing mechanism**

  From a mathematical standpoint, only some kind of feedback mechanism with time delay is necessary to allow a controller to incorporate a sense of time. This is how recurrent artificial neural networks handle time. However, the recurrent network encoding can be awkward for a human programmer to understand and work with, so NPL contains a specific type of node with the feedback state built in. It acts much like a capacitor or integrator, allowing the programmer to express the passage of time in a natural, intuitive way.

- **Implicit concurrency**

  With a sense of time comes the problem of extended events that overlap in time, and the attendant issues of synchronization and concurrency. Concurrency is fundamental to NPL: every part of the algorithm happens concurrently with every other part. There is no need to explicitly code for the ability to have two things happen at once.
• **Uniform processing**

An algorithm that consists of subcontrollers has two major categories of structures. One set of structures constitutes the subcontrollers themselves, while the other set constitutes the logic that enables and disables the appropriate subcontrollers. In NPL, the structures that compose the subcontrollers are identical to those that implement the switching logic. NPL is a "type free" language, in that there is a single type of value that describes both the inputs and outputs for the controllers and the inputs and outputs for the logic. Such a foundation increases the flexibility of the language, and makes rapid development and exploration easier.

• **Documentable**

NPL programs superficially look like some artificial neural networks. Such artificial neural networks are typically composed of a generic node structure with connection weights that are trained through example or exploration. Often, the inner workings of these trained networks are opaque, meaning that it is difficult to interpret what functional purpose a particular node serves. NPL programs differ in that they are built by hand like a traditional programming language. Every node in an NPL program is placed deliberately by the programmer, who therefore knows the purpose of the node, and can label it as such. In a well documented NPL program, the purpose of each node is labeled, and it is clear what a given change will do to the behavior of the program.

1.1.2 **Characteristics of the walking controller**

The design of the walking controller reflects the goals of simplicity and rapid development. The design stresses simple, intuitive building blocks, and de-emphasizes rigid mathematical formulations. The design takes some inspiration from biological designs, but biological plausibility was not one of the design goals.

• **Precision and accuracy aren't necessary**
Several prior motor control designs begin by describing mathematically the dynamics of the system. Such descriptions are vital if precise motions are required of the system. Walking, however, does not involve precise motions. A controller can recover from small errors that are made in one step during the next step. “Sloppy thinking” can even be a benefit: loose motion requires the designer to think more about error recovery than perfect path planning. As a result, the final controller is more likely to be able to recover from disturbances than one designed with a single trajectory in mind. The typical walking case of mostly level ground is a big advantage here. In most cases there is a lack of need for great foot-holds at every step. The precision necessary to plant the foot at an exact spot is usually unneeded.

- **Independent controllers**

  A lot of complexity arises from the interaction between different parts of the system. The walking controller of this thesis attempts to remove that interaction and complexity as much as possible. A single joint that can be used to perform a particular task is preferred over a set of interacting joints to accomplish the same task. In some tasks of motor control, for example balancing on one toe, this may not be possible. The walking algorithm presented here, however, is able to assign one and only one independent task to each of the major joints. This allows both simpler design and easier access points for higher layers to alter the overall trajectory.

- **Hierarchy and subsumption**

  The controller design technique lends itself quite well to a hierarchical or subsumption style architecture. In these architectures, complex behaviors are constructed through the layering of and interaction between basic behaviors. For instance, one controller might keep the swing leg a certain distance off the ground. Another controller layered on top could adjust the actual distance if the robot is walking on non-level ground.
• *Simple functions*

Both the relaxation of precision and accuracy and the use of independent controllers allow the programmer to avoid the use of complex equations. When multiple joints are not tied together, kinematic descriptions tend to be as simple as single sine or cosine functions. More complex equations can sometimes be reduced to something much simpler as well. Part of the reduction is attributable to approximations allowed by the lack of a need for precision. Other reductions can sometimes be made by knowing what the result of the equation will be used for. If the result of a complex equation is just being examined to see if it is too large or too small, that equation can be changed to a much simpler form whose output is monotonic with the original. An equation like $x^2 + \log x > 1$ is not necessary when $x > 1$ gives the same answer. Since NPL is not a mathematically expressive language, such reductions are necessary to implement an algorithm in NPL.

• *Use of momentum to simplify the controller*

Anyone who has gone hiking on a rocky path knows that it is much easier to traverse an entire set of rocks in one go than to come to a stop and balance on each individual rock. By diving through the motion instead of being hesitant, a rock climber saves effort that would otherwise have been used for the stop, balance and restart. A lot of complexity in maintaining balance can be eliminated by allowing momentum to carry the body through what would otherwise be difficult or complex portions of the trajectory. An application of this principle can be seen in the double-support phase of the walking controller in section 5.3.3.

1.2 Motivation

Traditionally, a walking controller is constructed through mathematical analysis. The programmer constructs the kinematic or dynamic equations of the robot and derives
a set of controllers from them. Such a system can produce very stable motion if the physics do not violate any of the assumptions in the simplifications. This style of controller is suitable if the programmer understands and can manipulate the underlying dynamics of the system.

Another class of controllers consist of learning algorithms. Most of these algorithms encode a stereotypical gait, and optimize a small number of parameters for that gait. Others attempt to learn the entire gait by optimizing a large number of weights. These algorithms are appropriate if the primary objective is adaptability, or if the dynamics are not fully understood or modeled.

Both of these categories of controllers inadvertently shield the programmer from being able to understand or debug the motion easily. If the programmer notices a strange gait artifact during testing, or intuits a new technique for the gait, it can be very difficult to change either type of controller. In the case of the mathematical controller, the dynamics may need to be reworked in order to accommodate the new idea. The learning algorithms may be even harder to alter, as it is often unclear what to change in a neural network, or what any given change will do to the system.

NPL provides the clarity of an algorithm in a programming language with some of the benefits of working with analog-style parts. The analog nature of the NPL language encourages “fuzzy” thinking, where the states of the system need not be discrete. Controllers can be partially activated, and state machines can gradually blend from one state to another. Blending is used in this thesis during foot placement, to avoid instantaneously applying a standing force while the foot is still in the air.

Another motivation for the structure of the NPL language comes from biology. Biological neurons have limits on the values they can represent: there are maximum and minimum firing rates in a neuron. In contrast, a mathematical formula has no inherent limits on its output values. Unless the programmer is extremely careful, some input states can produce unreasonable output torques. NPL alleviates the problem by limiting the outputs of its computational units to the range between zero and one. The different computational mappings of inputs to outputs in NPL were inspired by simple models of biological neurons and clusters of neurons.
1.3 Overview

The remainder of this thesis presents the NPL programming language and two motor controllers. One controller introduces the use of NPL in the cart–pole toy problem. The second controller implements the orthogonal decomposition walking algorithm.

Chapter 2 of this thesis provides some background into bipedal walking and presents solutions other teams have used to tackle the problem.

Chapter 3 introduces the NPL programming language. It describes the basic building blocks of the language, which are nodes that apply a function to the sum of the inputs into the node. Each of the seven types of nodes applies a different function, and is used for a different purpose.

The chapter then describes a set of digital and analog idioms that can be built with NPL nodes. These idioms comprise both the subcontrollers in a motor controller and the logic to switch them on and off at appropriate times.

Chapter 4 illustrates the use of NPL and the idioms in the cart-pole toy problem. This problem involves a cart on a rail with a freely swinging suspended pendulum. The task is to apply forces on the cart so as to swing the pendulum up to vertical and maintain its balance at the top.

The resulting controller illustrates the use of NPL to build subcontrollers and switching logic. It also illustrates how easy it is in NPL to use hierarchy and subsumption to correct for anomalous conditions and to add higher-level behaviors.

Chapter 5 describes the bipedal walking controller and its implementation in NPL. Stability under this algorithm is controlled by foot placement and timing. The algorithm exposes a small number of parameters that relate the step length and swing leg speed to the velocity of the body. As the velocity of the body gets faster, the algorithm causes the stride length and swing speed to increase, slowing the velocity back down. Each joint in the algorithm is contained within a high level negative feedback loop, enabling the system to recover from disturbances.

A desired velocity is never specified, nor is a specific foot placement location defined. Instead, the transition times and swing velocity work together to create a
steady-state limit cycle with a characteristic speed and step length. Different parameters produce different velocities and step lengths.

The simulator uses SD-fast, an accurate rigid-body physics simulator. The simulated robot is based on Spring Flamingo, a two-dimensional robot with six degrees of actuation designed and built by Jerry Pratt. The simulated structure is an identical copy of the one Pratt used to test his algorithms before applying them to the real robot, and has the characteristic masses and dimensions of the physical robot.

Chapter 6 analyzes the results of the walking controller. It discusses the stability and recovery characteristics of the robot, as well as the gait characteristics at different speeds. It illustrates the effects of changes in the high-level parameters of the algorithm on the behavior of the walking robot. Finally, it discusses the differences between building the controller in NPL and building the controller in similar techniques like neural networks and fuzzy logic.

Chapter 7 presents conclusions along with possibilities for future work, both in the programming language and with the walking controller.
Chapter 2

Background

This chapter introduces the structure of the robot used in this thesis, and describes some of the other methods researchers and roboticists have used to control walking robots. It provides brief descriptions of how the method presented in the following chapters differs from these techniques.

2.1 Robot Structure

![Figure 2-1: Structure of Spring Flamingo](image)

The robot used for this thesis is a physics-based simulation of Spring Flamingo, created by Jerry Pratt[Pra00]. Spring flamingo is a biped robot with six actuated
degrees of freedom. The physical robot was restrained from yawing or rolling by a boom; the simulation is simply constrained in software to two dimensions.

Figure 2-1 illustrates the coordinate system and angle measurements for the robot. Positive \( x \) is in the forward direction and positive \( z \) is up. However, the angles are all measured such that more forward angles are negative. This oddity is a result of the original design of Spring Flamingo, in which the robot walked in the other direction in a bird-like instead of human-like manner.

Spring Flamingo stands 88 centimeters from the bottom of its foot to its hip. Most of its mass is in the body, but the legs and feet are not weightless. The original robot had a squat body, rather than a tall torso as humans have. Again, the design originally reflected the structure of birds, rather than humans. As a result, the robot in this thesis has less use of the torso as a reaction weight or rotational inertia source as it would if it had been modeled in a more humanlike manner.

### 2.1.1 Terms

The structural terms for the parts of the robot are familiar to anyone who has ever sung the leg bone song. The foot is connected to the ankle, which is connected to the shin, then the knee, thigh, hip, and torso. The height of the robot is defined as the vertical distance from the ground to the location of the hip. The angle of the body with respect to vertical is the pitch.

Each joint can bend either in the positive direction, toward the back of the robot or in the negative direction, toward the front. A change in a joint angle that "straightens" the limb is called *extension*. In the model, these are all marked as positive changes in angle position. A negative change is called *flexion*. For the foot, flexion is sometimes called *dorsiflexion*, and extension is sometimes called *plantiflexion*.

There are a few terms associated with the contact between the robot and the ground. The *support points* are the locus of points on the robot that are in contact with the ground. The *support polygon* is the smallest convex polygon that completely encloses the support points. At each of the support points, there is a *ground reaction force*, which is the force and torque applied by the ground. The ground reaction
force does not necessarily point normal to the ground. Friction can apply a force perpendicular to the normal force.

Finally, the center of pressure is the point on the ground that is the weighted average of all the normal ground reaction forces at any instant in time. The center of pressure is not necessarily directly below the center of mass unless the robot is both stable and not moving.

2.2 Other Control Work

Several walking bipedal robots have been built to demonstrate various control algorithms. These control algorithms vary from playback of pre-recorded joint trajectories, to intuitive physics-based virtual controllers, to learning algorithms that attempt to mimic a demonstration.

2.2.1 ZMP and Trajectory Following

A major class of biped controllers relies on the computation of the ZMP, or zero moment point. Unfortunately, there is confusion in the literature over the definition of ZMP. A common definition is that the ZMP is the location at which the vertical reaction force intersects the ground. Goswami [Gos99] showed that this definition of ZMP is equivalent to the center of pressure, CoP. This view of the ZMP is therefore constrained to fall within the support polygon. Another use of ZMP indicates the Foot Rotation Indicator, or FRI, formalized by Goswami. The FRI is not restricted to lie within the support polygon. It is equivalent to the point that would have been the center of pressure if an infinitely large plate were bolted to the support foot. Goswami points out that the FRI is better at indicating foot rotation: where a CoP that lies on the support polygon boundary might indicate a foot that is rotating about that boundary or one that is just barely balanced, the FRI can distinguish between the two.

When the CoP lies strictly within the support polygon, the CoP and the FRI indicate the same point. The goal of most designers who use the term ZMP is to
keep the ZMP as far inside the boundary as possible, indicating foot stability. To these researchers, the terms CoP, ZMP, and FRI are all interchangeable, hence the sloppiness in terminology.

The Honda team \cite{Hir97, HHHT98} recorded actual human walking data for the various tasks they wanted the robot to perform. They analyzed and augmented the data offline so the ZMP remained well within the support polygon during the entire walk cycle. Their robot design incorporated the ability to play back the tweaked recordings perfectly, while adjusting certain joint angles on the fly to maintain the position of the actual ZMP measured in realtime compared to the ideal location specified by the recording. Huang et al. \cite{HKK+99} used a similar playback technique, although they used the reaction mass of the torso to adjust for errors in the ZMP.

The result of the edited recording on the Honda robot is a fairly realistic looking walk. With the right recordings, the robot can walk up and down stairs, walk and turn, and even kick a soccer ball. However, if the Honda team wants the robot to walk faster, they need to make a new recording, and transitions from one recording to another need to happen at exact locations. Furthermore, although ZMP tracking can handle small disturbances, a large push will cause the robot to fail. Without a set of recovery heuristics built into the controller, the team would require a recording to handle every conceivable disturbance.

Many other controllers focus on trajectory following. Zhou et al. \cite{ZJ96} build on the work of Shih \cite{SGZ91} to describe a hybrid fuzzy logic and neural network system. The system learns the proper torques to apply at the joints to make the biped system trace a given trajectory as accurately as possible.

The drive to reproduce an ideal trajectory as accurately as possible is an interesting goal, but not necessarily the most useful. Walking isn’t a game of accuracy. Joints are not stiff, like a pick-and-place robot arm, but loose. A natural walk comes from obeying natural dynamics and allowing the feet to come down somewhat arbitrarily. If you force a person to place their feet at specific locations, the walk becomes stilted and unnatural. The idea presented in the rest of this thesis is that a general type of trajectory can be specified, but the actual resulting trajectory should be caused by
other interacting forces.

2.2.2 Natural dynamics

Trajectory following controllers tend to be very stiff, in order to maintain strict control over the joint angles. At the other end of the scale, McGeer [McG90] created a completely passive robot with knees that can walk down a shallow incline with no active control whatsoever. The model contains loose joints with a damper to emulate the knee stop of a human leg.

When released with the right initial conditions, the stance knee automatically locks in place as the other leg swings forward. The system stabilizes when the impulse of the swing foot impact puts the state of the robot into the same conditions as the previous step. McGeer performed a small-perturbation linearization analysis of the system to show that small errors in the swing leg state are wiped away within one step, and small errors in step length or velocity smooth out over several steps.

The energy to propel the passive walker comes from gravity. McGeer’s walker can function on small declines, but cannot walk on level ground. Still, the fact that completely passive dynamics can stabilize a human-style biped are encouraging: complex control systems are not needed for walking down shallow slopes, and may not be needed on level ground, either.

2.2.3 Virtual model control

Pratt et. al [PDP97] designed a controller that falls halfway between stiff trajectory following and passive dynamics. Their system uses force control, rather than position control, as the basis of their actuators. The technique, called Virtual Model Control, decouples the programmer’s view of the control structures from the physical actuators of the robot. Semi-arbitrary springs and dampers can be emulated by the actual joints through the application of the forward kinematic equations. For example, an intuitive way to get a bipedal robot to stand up is to suspend it by springs from a large granny walker. The Virtual Model Control equations explain how to emulate those granny
walker springs with torques at the various joints. When the controller is active, a push on the body would feel exactly as if the granny walker and springs were really there.

One benefit of Virtual Model Control is that it allows the programmer to think of intuitive abstract controls, in any kind of reference frame. Such abstractions make it easier to come up with control schemes. A second benefit is that it allows loose controllers that have more realistic reactions to disturbances. The difficulty is that sometimes the equations can get very complex, and the controller can output extreme torques that make no sense in the real world. Furthermore, a bipedal walker needs three distinct sets of equations: one when the left foot is on the ground, one when the right foot is on the ground, and one when both feet are on the ground. The transitions between these sets of equations can make the robot unstable if they are not blended carefully.

2.2.4 Mathematical analysis

Most forms of mathematical analysis attempt to explain or simplify some aspect of bipedal walking by modeling an abstraction of a bipedal walking robot.

In [LL88], Lee and Liao present a complete mathematical analysis of a three link, or kneeless robot, and derive a controller for it. In [LC88], Lee and Chen present some simplifying assumptions of the dynamics of a more typical 5-link robot, and use gradient descent to derive a minimum-fuel trajectory. These papers illustrate the possible uses of simplification for walking control. In the first paper, the model itself is simplified by removing two degrees of freedom. Unfortunately, the resulting system becomes too simple, and while the final controller works, it does not lend any insight into the more complex system.

The second paper uses a full model similar to Spring Flamingo. It simplifies the problem by only examining a stretch of continuous motion between one foot strike and the next, with an instantancous changeover. The trajectory for each joint is defined as a set of continuous B-splines, with knots at a small number of uniform time intervals. A reward function based on positional constraints and energy usage allows gradient
descent to find the minimum-energy walk. This method cleverly reduces the problem space from finding correct values at every instant to finding optimal parameters for a very small number of splines. The result illustrates a good set of trajectories, but does not describe how to handle perturbations. Again, there is little insight into walking behavior in general.

Kajita [KTK90] created a simplified walking model that does deliver some interesting characteristics of bipedal walking control. He used a point-foot biped robot that maintains its height during the walk cycle. He describes a quantity called the “orbital energy” which is a combination of the kinetic energy of the body of the robot and the potential energy implied by the vector between the point foot and the center of mass: 

\[ E = \frac{v^2}{2} - \frac{gx^2}{2h} \]

where \( v \) is the velocity, \( x \) is the horizontal distance between the foot and the center of mass, and \( h \) is the height of the center of mass. He found that if the height remained fixed, \( E \) remained constant through the entire stance phase. Kajita used this equation to find a formula for foot placement to produce a stable walk.

The use of foot placement to control velocity was also worked out by Raibert [Rai86]. Although the Raibert robots were typically hopping and jumping robots instead of walking robots, the velocity control systems used similar principles. Another useful insight from the Raibert hoppers was the decoupling of controllers. The height controller, forward velocity controller, and pitch controller all worked independently, and considered any kind of coupling between them to be an error to be corrected in the appropriate controller. The decoupling simplified the locomotion problem considerably; we use the same technique here to reduce the complexity of the walking algorithm.

Kajita’s orbital energy is an important concept. It implies that both the stability and the speed of a bipedal walk can be controlled simply by the placement of the swing leg for the next step. If the controller for the full Spring Flamingo model were to use only the orbital energy for stability, then the stance ankle could be left entirely limp. This concept is foreign to the ZMP algorithms. The entire purpose behind the ZMP algorithms is to prevent ankle torque from driving the ZMP to the border of
the support polygon. With a limp ankle, that event never becomes an issue.

In reality, of course, three issues prevent the sole use of orbital energy to control the walk cycle. First, during a normal walk, the body bobs up and down, breaking the orbital energy requirement that the height remain constant. Second, the changeover between stance phase on one leg and stance phase on the other is not instantaneous. There is significant overlap, during which adjustments can be made that override or supplement foot placement. Finally, the ankle is used both for fine tuning balance during the entire stance phase, and for providing extra energy at the end of the stance phase during toe-off. However, by focusing primarily on foot placement, the issues surrounding ankle torque and ZMP can be minimized.

2.2.5 Learning algorithms

If the system is simple enough, a neural network can be trained to duplicate the functionality of an existing controller. Both Lee and Wang [LW91] and Lee and ElMaraghy [LE92] demonstrate a three link, or kneeless robot with neural nets that were trained using mathematically derived controllers. The low number of inputs and degrees of freedom make this kind of learning algorithm tractable.

Schaal [Sch97, SA94] demonstrated how reinforcement learning can be kick-started with a demonstration of the task, even if access to the state of the controller is denied. In one version of his devil stick juggler, he used a memory-based architecture to build up knowledge of the control space. In another, he primed the policy of a reinforcement learning system with examples from a human demonstration, and found the time to learn a stable controller was cut drastically from reinforcement learning without the examples.

Kun and Miller [KM96, KM98, Mil94a, Mil94b] posed a combination of trajectory following and learning. Their controller consisted of an idealized gait that was modified by several neural networks before being passed to the joints. Each network performed a specific modification to control some aspect of posture or balance. The networks were trained consecutively, so that forward motion did not begin until the robot could rock from side to side without falling over.
In all of these cases, the number of inputs and outputs to the learning system was kept to the absolute minimum. This is both a lesson and a warning. The warning is that learning problems become intractable very quickly when the search space becomes large. The trick of concentrating the learning examples around the desired final trajectory can increase learning speed, but cannot help build a good controller if the number of possible disturbance vectors gets too large.

The lessons to take from these examples involve the ways to keep the search space low. A good start involves building a complex controller that contains a very small number of adjustable controls or knobs. Even better, as demonstrated by Kun and Miller, is to make the knobs themselves orthogonal, so that they can be trained independently.

### 2.2.6 Constructed networks

Shih [SGZ91] presented a fuzzy logic controller to balance the forces in a robot during double support. The state variables were quantized and used as fuzzy inputs. Fuzzy rules defined force and position control based on mathematical formulas for the Jacobians at each quantized state.

The use of quantization for this problem is very similar to the method used by Lee and Chen above. Where Lee and Chen broke the timescale into discrete chunks, Shih broke the problem space into discrete chunks. Fuzzy math used in this way is similar to using a set of radial basis functions to approximate a true mathematical function. This method has an advantage when the complete formula is not known, but approximate values can be found by trial and error at specific points in the problem space.

Although Shih’s controller only handled a simple toy problem, it is useful in that it demonstrates how a complex mathematical problem can be approximated using much simpler functions. Through a combination of feedback and simple nonlinear functions, the system provided the same control that would otherwise have taken a difficult matrix inversion to compute.

Taga [Tag95] used neural oscillators [Mat85] as a central pattern generator to
control a seven-link biped simulation. Each pair of oscillators controlled the simulated muscles at the joints, with some muscles crossing multiple joints. A state machine reflected the phase of the walk cycle, and modulated the outputs from the oscillators to the muscles. Taga was able to hand tune the connection parameters for the oscillators and muscles to produce a stable walking gait.

The impressive feat of hand-tuning the enormous number of parameters in the system was made more manageable by Taga's adherence to the lessons learned in the previous section. In particular, he separated the parameters into distinct groups with specific purposes. For example, the neural oscillator parameters were tuned to match the natural frequency of the swing leg, and the muscle activation parameters were initially tuned to match values observed in human walking. The remainder of the parameters and fine tuning presumably took much trial and error work.

Another good idea in the Taga controller involved the use of a global state variable to select distinct sets of parameters for each of the phases of the walk cycle. This allowed each set of parameters to perform a simple task, instead of making a single set of parameters handle all of the different tasks. An alternative way of expressing the state concept is that Taga's controller contained several simpler sub-controllers, each of which described the outputs for a single phase of the walk cycle. The Taga controller can thus be seen as a switching algorithm among several simple controllers.

Although the use of time-based subcontrollers makes the parameter programming easier, the type of connections and abstractions made in the subcontrollers make them difficult to adjust. There is no way to build up the controller in a hierarchical or subsumption style to perform more complex tasks.

Another group of robots and controllers in this category are the BEAM style robots inspired by Mark Tilden [Til95]. These robots use hardware based oscillators called "bicores" as the basis for the control system. Each bicroe consists of two inverters with a motor or other load strung between them, and capacitors and resistors to control the oscillation rate. Bicores can be slaved to each other to produce coordinated movement, or can be disabled with external sensors to produce other behaviors.

The robots are small and insect-like. Tilden's bicores only produce full motion in
one direction or the other, so fine control is not possible. Instead, the robots employ four or more legs so that balance is not an issue. The controller changes the behavior of the robot by altering the phase difference between the motor bicores. The control is structured hierarchically, so that the behavior editing circuitry is separate from the low level motor control circuitry. This separation enables easy editing of behaviors.

2.3 Discussion

One of the fundamental concepts behind the design of the walking controller in this thesis is the idea of underdesigning. This principle crops up several times. One example is to resist the temptation to use a specific function when an arbitrary monotonic relationship to that function will suffice. It is not necessary to calculate the exact distance from the hip to the ground when a function that gets monotonically larger as the height increases is easier to compute. Both functions serve to identify when the robot is too high or too low; why waste computational power in calculating the more complex exact function?

Another instance of underdesigning can be illustrated by comparison to the overdesigning of the ZMP algorithms. Many of these algorithms spend an inordinate amount of computational energy attempting to keep the ZMP precisely in a desired location. A guarantee of ZMP position is equivalent to a guarantee of stability, which is a strong point of these algorithms. Foot stability is not a necessary condition to a stable walk however. Human walking involves a toe-off phase, where the rear foot rolls over the toe before pushing off. Such foot rotation is antithetical to the ZMP philosophy. Even so, transferring most of the stability problem in walking to foot placement eliminates the difficult mathematics required to solve the ZMP problem. As discussed later in the thesis, foot placement, taking some ideas from Kajita, is actually a fairly straightforward problem.
Chapter 3

The Neuronal Programming Language (NPL)

The walking controller in this thesis uses a style of programming called a “Neuronal Programming Language”, or NPL. NPL incorporates the ideas from section 1.1.1, specifically that the language and development environment:

- have a rapid prototyping model,
- encourage the subsumption style of programming,
- utilize simple parts with standard structures, and
- have a visual construction environment.

The result is a development environment much like Simulink or other circuit and network tools. The programmer drags individual computing boxes into a work area and connects them up by drawing lines between them. NPL uses building blocks reminiscent of traditional artificial neural networks, but NPL is actually a graphical programming language instead of a learning tool.

A traditional artificial neural network consists of a set of identical nodes, with a prescribed standard set of connections between the nodes. The function represented by the network is controlled solely by the weights of the connections. An NPL program is the reverse of this paradigm: the function is conveyed by the structure of the
network, and much less importance is given to the weights of the connections. There are no restrictions on creating cycles, and different paths from the network input to the network output may be of different lengths. Typically, the ratio of connections to nodes is much lower in an NPL program than in an artificial neural network.

NPL programs are designed in a process similar to that of a traditional programming language. Each structure in the network is built by hand by the programmer, and thus each cell can be documented with a meaning or purpose. NPL algorithms are not “trained.” Instead, the programmer sets the weights of the connections manually. Except for a few well-defined cases, the weights of the connections are +1, −1, or a large negative value. Like traditional programming languages, there are common structures, or idioms, that consist of several building blocks and can be adapted for specific tasks. The development environment enables the programmer to construct an NPL program visually. The program is always live, in the sense that the program can be running while it is under construction or being edited. This gives the programmer immediate feedback on changes.

3.1 Building blocks

<table>
<thead>
<tr>
<th>Const</th>
<th>Input</th>
<th>Output</th>
<th>Sigmoid</th>
<th>Gaussian</th>
<th>Accum</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>&gt;</td>
<td>&lt;</td>
<td>J</td>
<td>A</td>
<td>=</td>
<td>+</td>
</tr>
</tbody>
</table>

Figure 3-1: Neuronal Programming Language node types

The most basic building blocks of NPL are inspired by simplified models of biological neurons or clusters of neurons, and are very similar to the basic building blocks of artificial neural networks. Each node sums a series of inputs, applies a specific function to the sum, and provides the result of that function as output. Connections between nodes have a “weight,” which multiplies the output of the originating block before it is presented to the input of the destination block.

The value of each node is updated concurrently. At every time step, the inputs for each node are summed. After all the input values are calculated, the new output
values are generated. The time delay between controller updates is determined by the simulator. The default value for the simulations in this thesis is 2 ms of simulated time, or an update rate of 500 Hz. The response time of a controller is the update time multiplied by the number of computation nodes between the input node and output node.

The set of node types in NPL was designed to keep the number of different kinds of nodes to a minimum while enabling the characteristics outlined in section 1.1.1. In particular, each type of function both services a specific task, and provides the flexibility to be used in creative ways. Most of the functions limit their output between 0 and 1. This enables digital computation in addition to the typical analog calculations.

There are seven functions in use, as follows:

- **Constant.** This node ignores its inputs and instead outputs a constant value. Constants are used either as user-adjustable parameters, or as biases to nodes that do not have built-in adjustable biases.

- **Input.** This node ignores its inputs and instead outputs the value of some state variable from the simulation. The output of this node has no set minimum or maximum values, but ranges according to the domain of the state variable.

- **Output.** This node sums its inputs and sends the sum to some control variable in the simulation. In our robot controller, each output specifies the desired torque at a particular joint. The range of the output node is limited naturally, due to the fact that most of the computational nodes in NPL have a limited output range.

- **Sigmoid.** This node applies a sigmoid curve with a specified bias and sigma (width) to the sum of the inputs. The output will always fall between 0 and 1. Sigmoid nodes have two adjustable parameters: $\sigma$, the width of the sigmoid function; and $b$, the bias of the sigmoid. The bias is an implicit input: its value is added to the inputs before the sigmoid function is applied. If $x$ is the sum of
the inputs, the formula for the sigmoid function is

\[
\frac{1}{1 + e^{-(x+b)/\sigma}}
\]

The sigmoid is the basic function used in traditional artificial neural networks. By definition, any artificial neural network can be implemented as a program in NPL, and therefore any function that can be approximated with an artificial neural network can be approximated by NPL. Put in the hands of a programmer, the sigmoid function is extremely versatile, with uses that range from analog computation to digital logic. In the figures in this thesis, the width of the sigmoid is indicated by the steepness of the curve, and the location of the bias is indicated by a thin vertical dashed line.

- **Gaussian.** This node applies a gaussian curve with a specified center \((x_0)\) and standard deviation \((\sigma)\) to the sum of the inputs. The output will always fall between 0 and 1. The formula for the function is

\[
e^{-(x-x_0)/\sigma^2}
\]

Gaussians are useful for selecting regions of interest from a range of values. In this sense they are useful for coarse coding, with the added benefit that the selection regions are “fuzzy”. Overlapping regions of interest allow the programmer to perform various discrimination tasks between the regions in addition to using the regions as fuzzy selectors. A gaussian can also be used to approximate a portion of the cosine function.

- **Accumulator.** This node maintains an internal value between 0 and 1. The value is raised or lowered according to the sign and magnitude of the sum of the inputs. The action is as if the node is a capacitor connected to 0 or 1 through a resistor such that the time constant varies inversely with the magnitude of the input signal. If the input is zero, the output value of the accumulator does
not change. A parameter, \( r \), gives the overall rate of change, and a leak value biases the input signal. The formula for this function is:

\[
q_{t+1} = q_t + \begin{cases} 
(1 - q_t)a/(r + a) & \text{if } a > 0 \\
q_0a/(r - a) & \text{if } a \leq 0
\end{cases}
\]

where \( q_t \) is the value of the output at time \( t \), and \( a \) is the sum of the inputs and the leak value.

NPL includes the accumulator function in order to possess a node that has a native sense of timing. The accumulator’s capacitor-like properties allow it to behave as a first class citizen of NPL by restricting its output to the range \([0, 1]\). Accumulators can be used to perform smoothing operations, to act like integrators over small ranges, and to construct oscillators.

- **Sum.** This node simply outputs the sum of its inputs. It can be used as a single-time-step delay, or to reduce the number of connections in a functional block. Because it just passes the sum of the inputs, its output is not restricted to the range between 0 and 1.

There are a few other types of nodes that exist as an aid for debugging and testing. A set of trigger nodes have special functions that activate when their input rises over 1/2. They include a \textit{reset} node that resets the simulator and NPL program to an initial state, a \textit{recording} node that dumps the values of a set of simulation variables and NPL node values to a file whenever it is triggered, an \textit{exploration} node that sets the value of simulation variables and NPL node parameters to random values in a specified range, and a \textit{counting} node whose output value increments or decrements by one.

At every time step, all new outputs are computed concurrently. This means that the propagation time of a signal from an input to an output is determined by the number of nodes and connections between them.
3.2 Construction Theory

An NPL program typically consists of a set of simple feedback or feedforward controllers, and logic to turn them on or off at the appropriate times. The goal of the programmer is to break the task down to an appropriate set of controllers and to develop the switching logic between them.

Like any programming language, NPL has a set of idioms, or functional blocks, that are commonly used. The first set uses sigmoid nodes to create fuzzy analogues of digital circuits.

3.2.1 Digital idioms

![Diagram of NOT, NOR, and AND gates]

Figure 3-2: NPL versions of the NOT, NOR and AND gates. Connections are directional, as indicated by the arrowhead on the connection. Dashed connections have negative weights. The width of the connection reflects the magnitude of the weight. The thin vertical dashed line within the sigmoid nodes indicates the bias. All the connections in this figure have weights of +1 or −1.

The NOT gate is fairly straightforward. The bias of the node is set high so that the output is high in the absence of any inputs. Since the weight of the input connection
is negative, indicated by the dashed line in figure 3-2, a positive value from upstream will cause the output of the NOT gate to fall. A NOR gate is simply a NOT gate with two negated inputs. Any number of inputs can be combined in a NOR gate, as long as the “off” signals are kept clean, near zero.

If the bias is turned around and the input connections are positive, a NOR gate becomes an OR gate.

A few assumptions need to be met in order to build AND and NAND gates. Most importantly, the input values on each connection must remain between 0 and 1. An AND gate is similar to an OR gate, but the bias is moved in the negative direction far enough that all input connections need to be high in order to bring the output over the threshold. Thus, unlike the OR gate, the bias on the AND gate must be changed according to the number of inputs into the node. Note that in general, an AND gate does not give the same value as negating the inputs and output of a NOR gate via NOT gates. This is primarily because of the fact that the negation provided by a NOT gate clips the value, and some information is lost.

Care must be taken with values that are to be ANDed digitally. The values must be guaranteed to be clipped to a maximum value of one. In general, this caution does not apply to OR and NOR gates.

Figure 3-3: NPL flip-flop

Figure 3-3 illustrates a flip-flop. Two sigmoid “NOR”-style nodes inhibit each other and stimulate themselves. In the absence of any other input, the node that has a high value will retain a high value while inhibiting the other node. In this way,
the flip-flop retains its state. A large positive value on the “set” connection of the
dormant node can override the inhibition from the dominant node. This will cause
its output to rise, overriding the self-stimulation on the other node, and the state will
flip.

![Diagram](image)

**Figure 3-4: An NPL state machine with four states**

The multi-state extension of a flip flop is a “one hot” coded finite state machine.
In figure 3-4, there are four state nodes, but the procedure is generalizable to any
number of states. As in the flip-flop, each state node inhibits all the others and
stimulates itself. The transitions between states are carefully constructed so that
each state can only be triggered when the previous state is active.

The trigger nodes act as AND gates. The transition occurs only when both the
prior state node and an external input, shown in gray in the figure, are active. In
the example, there is only one transition out of any given state. Some care must
be taken if the state machine represented by the connections needs to make decision
branching. The designer must ensure that only one of the multiple transitions is
triggered to avoid splitting the state into two parts.

Upon startup, a mechanism is required to initialize a single active state. This can
be accomplished with a nor gate. The inputs to the nor gate come from each of the state nodes. The output of the nor stimulates the initial state. As long as at least one state is active, the reset node is dormant. When all states are off, the initial state is stimulated and the state machine gets initialized.

3.2.2 Analog idioms

When sigmoid nodes are not in their saturated region, they can be used as op-amps to perform analog calculations.

Figure 3-5: NPL proportional controller (left), and a proportional controller with inhibitor (right). The thicker dashed connections from the inhibit node indicate a large negative weight. The inhibit node is shown as a sigmoid to illustrate that its value is computed from other structures not shown in the picture.

Figure 3-5(a) illustrates a typical discriminator, or proportional controller. This small network takes an analog input, and produces an output in the range $[-1, 1]$. The output is zero at some discrimination point, indicated either by biases of equal magnitude but opposite sign, or by an external value which stimulates one node and inhibits the other.

The output of the pair of sigmoid nodes is

$$\frac{1}{1 + e^{-x/\sigma}} - \frac{1}{1 + e^{x/\sigma}} = \frac{2}{1 + e^{-x/\sigma}} - 1$$
In English, the output of the pair of sigmoid nodes is equivalent to an offset of a single sigmoid node with a doubled output. What is the use of using two sigmoid nodes instead of one? The benefit comes from being able to disable the output of the discriminator.

Figure 3-5(b) demonstrates the disabling technique. By inhibiting both nodes, the disabling signal can force the output towards zero from both nodes. This technique demonstrates the power of the agonist/antagonist structure of differential signaling. If only a single sigmoid with a range of $[-1, 1]$ was available, there would be no way to disable the output.

Inhibition is not a binary property. Because the sigmoid has a smoothly changing slope, a small amount of inhibition will cause the response of the discriminator near the discrimination point to decrease in slope instead of disappearing suddenly. Figure 3-6 illustrates this behavior. Note that the slope at the center of the inhibited plot is small, but non-zero. This effect can be useful when the programmer wishes a particular controller to contribute a small effect at some times and a larger effect at other times.

A discriminator with a sigma of about 0.4 in each node turns out to be a fairly
good approximation of a sine function out to about ±1. Similarly, a gaussian node with a standard deviation of about 3.9 is a fairly good approximation of \((\cos(x) + 1)/2\) out to ±1.

![Diagram of PD controllers]

Figure 3-7: PD controllers. (b) illustrates both the use of an inhibitor to turn off output, and the computation of the derivative of the input using two sum nodes. The sum node on the left simply delays the input signal by one time step. The sum node on the right approximates the derivative by computing the difference between the current and delayed signals. The weight of the connections to the second sum node must equal the desired weight times the frequency of the controller update.

Figure 3-7(a) shows a proportional-derivative (PD) controller. It is simply a discriminator with inputs from both a signal and its derivative. Typically, the weight on the derivative connections will be slightly smaller in magnitude than that of the direct connections for a critically damped system. Some hand tuning may be required to adjust the proper damping.

In a live interactive environment, tuning a damping parameter becomes fairly straightforward. The programmer needs only to watch for the end of vibrations as he or she lowers the damping parameter slowly.
If the derivative signal is not available, it may be computed as in figure 3-7(b). The sum node on the left delays the signal by one time step. The sum node on the right computes the difference between the current value and the value from one time step ago. Its computed value is actually the derivative of the input signal multiplied by the time step of the simulation. Either the input weights to this node or the output weights from this node must be multiplied by the update frequency in order to recover the correct derivative of the input signal.

![Diagram](image)

**Figure 3-8: Low pass filter**

With feedback, the accumulator node can be used as a kind of low-pass filter. Recall that the value of the accumulator increases when the sum of its inputs is positive, and decreases when the sum is negative. Figure 3-8 shows an accumulator with a negative feedback of weight $-1$ and a tracking input. The system is only at equilibrium when the tracking input and the output of the accumulator node are equal. Otherwise, the value of the accumulator will increase when the input is higher than the output, and decrease when the input is lower than the output.

The low pass filter idiom can be useful when a smooth transition from one value to another is required.

On its own, the output of a single accumulator node with a small positive leak value will drift towards unity. A large negative input will cause the output to sink rapidly. The upper right corner of figure 3-9 shows a set of three accumulator nodes
Figure 3-9: Oscillator (three nodes in upper right corner) and a phase-lock loop (larger structure). The node marked +/- is a special counting node, which increments on the rising edge of one input and decrements on the rising edge of the other input. The counting node leaks towards zero to prevent the magnitude of its output from becoming too large.

with positive leaks, where each node inhibits the next. Suppose the three nodes are labeled a, b, and c. If a is high, it inhibits b, driving the output of b to zero. This allows the value of c to drift up. After the value of c crosses the leak value, it drives a down, allowing b to rise up. The process results in a continuous oscillation, with each node 120 degrees out of phase with the next.

The rate of the oscillation is determined by the rate of the accumulator nodes, or equivalently, by the leak and the weight of the inhibition between the nodes. Recall that a discriminator can be inhibited smoothly. Replacing the simple inhibitory connection between the nodes of the oscillator with a discriminator causes the oscillator
to have a tunable frequency. The more the discriminators are inhibited, the slower
the oscillation becomes.

With a bit of tinkering, the tunable oscillator can be transformed into a phase
lock loop. The large structure in figure 3-9 illustrates the phase lock loop. The
three accumulator nodes inhibit each other through two-node discriminators. The
sum node in the center of the tunable oscillator supplies the inhibition signal to each
of the discriminators. The adjustments to the inhibition are provided by a fourth
accumulator node below and to the right of the sum node.

Because of their capacitive behavior, accumulator nodes only act like linear inte-
grators when their output value is close to 0.5. The constant node adjusts the range
of the accumulator so that an output of 0.3 from the accumulator maps to zero in-
hibition. A very tight sigmoid node activates when the sum node gets close to zero,
preventing the accumulator from discharging below a value of 0.3.

The correction signal to the accumulator comes from a specially constructed count-
ing node marked by the +/- notation in the figure. This node has three inputs: pos-
itive trigger, negative trigger, and reset. A rising edge on the positive trigger causes
the output to increment by one, while a rising edge on the negative trigger causes the
output to decrement by one. A parameter of the node causes the output to decay
towards zero at some given rate.

The phase lock loop is activated by connecting one of the oscillating nodes of the
phase lock loop to the positive trigger and the external tracking signal to the negative
trigger of the counting node. If the signal is faster than the oscillator, the counting
node will have a negative value, and the damping accumulator will decrease in value,
speeding up the oscillator. Once both oscillations have nearly the same frequency,
the difference in phase will trigger small adjustments in the oscillator speed. Only
when the oscillators both have the same frequency and phase will the adjustments
stop.

Some experimentation is required to tune the rate of the inhibition accumulator.
A sample of the frequency and phase tracking of the network is shown in figure 3-10.
3.3 Development environment

The development environment for NPL was implemented in two pieces. The physics simulation was initially created by Jerry Pratt for his Ph.D. thesis using the Creature Library and SD Fast[Pra00]. It was altered to run as a server on a Linux system. The physics server waits for a connection from client software, then sends a description of the model and its state variables to the client. The client then takes control, sending changes to the simulation variables and requesting the server to simulate for a specific period of time. When the client eventually quits, the server continues running, waiting for another client to connect.

The client was written in Java, and so can run on any machine with a Java VM. The client software, illustrated in figure 3-11, displays a window with a drawing of the current configuration of the robot, a list of the available variables, and a command line.

A second window displays the NPL development environment. A list of node types appears in the top right corner of the window. The programmer can drag a node from the palette to the work area, and connect nodes together by dragging from the lower right corner of any node to any other node. Clicking on a node or a connection brings
Figure 3-11: Client windows: state visualization and command line above, NPL development area below.
up appropriate input boxes under the node palette, where the programmer can change the name of a node, edit the properties of a node or a connection, and view its output value in real time.

Each node displays a coarse representation of its value as a colored bar filling the node. Red bars are positive, blue bars are negative, and the height of the bar indicates the magnitude between zero and one. The bars update in real time with the stick figure drawing of the robot, allowing the programmer to view the state of the algorithm as it runs.

At the command line, the programmer can run, stop, or reset the simulation, and can record up to eight seconds of data of any state variable. Once data has been recorded, the programmer can save it to a file, or view the data as a graph in the development environment. The command line can also be used to change the properties of any node in the NPL program.

The client/server relationship is communication intensive, but allows development from any computer without having to port the physics simulation code. On a high speed local network, the simulation runs at approximately one quarter real time. A command line option can slow down the simulation speed so that the algorithm may be observed at a greater time resolution.

3.4 Summary

NPL programs are very close in spirit to fuzzy logic programs. Fuzzy logic consists of a series of if\ldots then statements, where the truthfulness of a particular expression can take intermediate values. NPL and fuzzy logic share the concept of perspicuity: any given expression or node has a well-defined meaning. In the case of fuzzy logic, the rules are specified linguistically. In a well-documented NPL program, the purpose of a particular node is specified by its name, and its relationships to other nodes is immediately visible by its connections.

NPL differs from fuzzy logic primarily in the functions available. First, NPL defines an accumulator or integrator that explicitly encodes a time constant for rate.
of change of its output. Second, NPL uses a sigmoid function that is near-linear at its center, but has a smoothly changing slope. For the nonlinear nodes and rules, fuzzy logic typically encodes a linear function with clipped ends. The smooth transition enables gradual inhibition of a discriminator, as discussed in section 3.2.2, and allows the simple approximation of a sine function. These capabilities are not possible in a system that uses an abrupt clipping function.

The fact that both the weights of the connections and the width of the sigmoid or gaussian function are adjustable may seem redundant. After all, a change in the value of $\sigma$ is equivalent to dividing the weights of the input connections by the change. NPL allows both adjustments to be made. The paradigm in NPL is to restrict the weights to +1 and −1 except where one input must overpower another, as in the inhibition of a discriminator. The semantics of the $\sigma$ value are related to whether the node represents a sharp, digital distinction, or a smooth analog range. The default $\sigma$ for a sigmoid node is 0.2, which provides a large linear region while still producing outputs close to 0 and 1 for inputs at 0.5 and −0.5. For cleaner digital values, a $\sigma$ of 0.1 is adequate; the sine approximation uses a $\sigma$ of 0.48.

The swingup and bipedal walk controllers presented in the next chapters consist almost entirely of the digital and analog blocks described in this section.
Chapter 4

The swing-up and balance task

This chapter illustrates the use of NPL in the construction and debugging of a simple control problem: the swingup and balance task. In this task, a pendulum hangs freely from a cart that is confined to travel along a straight rail. The control task is to push the cart back and forth so as to swing the pendulum upright and maintain its balance vertically over the cart. A schematic of the setup appears in figure 4-1. In the simulation, the mass of the weight is about one quarter that of the cart. The final algorithm described in this chapter successfully accomplishes the swingup and balance task for a wide range of weight ratios.

There are four input variables: $\theta$, $\dot{\theta}$, $x$ and $\dot{x}$. The sole control output is the desired force on the cart in the $x$ direction, $f_x$. A large number of papers have described learned and optimal controllers for various versions of this task [AS97, sF96, CH95], but here we will treat the problem as a naive programming task to illustrate the use of NPL.

Figure 4-1: Schematic of the pole/cart system for the swingup and balance task
4.1 Finding subtasks

The first procedure in programming is to break the task into smaller subtasks. In this case, there are two obvious subtasks: swingup and balance. The swingup subtask requires the controller to pump energy into the pendulum, swinging it from its initial downward direction to approach vertical. Once the pendulum is near vertical, the second subtask takes over and balances the pendulum.

4.1.1 The swingup task

A simple method of pumping energy into a lightly swinging pendulum is to apply a force on the cart opposite to the direction of the swing velocity of the pendulum. A little experimentation with a ruler makes this intuitively obvious, but it is also readily apparent from the dynamic equations of the system. Equation 4.1 describes the angular acceleration of the pendulum, where $g$ is the gravitational acceleration, $\mu_{\theta}$ is the angular friction at the pivot point, and $l$ is the length of the pendulum.

$$\ddot{\theta} = \left( g\sin\theta - \dot{x}\cos\theta - \mu_{\theta}\dot{\theta} \right) / l$$

The major controllable component of this equation is $\dot{x}$, the acceleration of the cart. Since its contribution is negative, a force with the same sign as $-\dot{\theta}$ will accelerate the angular velocity of the pendulum. The $\cos\theta$ term implies that accelerations applied when the pendulum is close to horizontal will have little effect, and accelerations when the pendulum is above the cart will have the opposite effect.

The most efficient way to swing the pendulum up is therefore to apply a force to
the cart opposite to the angular velocity of the pendulum only when the pendulum is below horizontal.

4.1.2 The balance task

Once the pendulum is close to vertical, a balancing algorithm must take over the control. Balancing consists of two components, illustrated in figure 4-3. If the pendulum is vertical, but moving in a certain direction, the cart must be pushed in the same direction as the movement. This action keeps the cart underneath the pendulum. If the pendulum is not vertical, then the cart must be pushed in the direction of the lean. This action moves the cart to be underneath the pendulum, straightening it. Both of these corrections work in parallel, and need to be tuned relative to each other so that the effect of one correction does not swamp the correction of the other.

In a real system, the force that can be applied by an actuator is limited. A hard force limit is imposed by the simulation system, but as described below in section 4.2.1, the algorithm implemented in NPL has self-limiting outputs. In the case of the cart, the limited force causes a maximum value for acceleration, and therefore a limit to how fast the pendulum can be moving, or how far the pendulum can be tilted and still allow the controller to recover its balance. There is no benefit to running the balance controller when the pendulum is outside this range.
4.2 Creating the controller in NPL

The intuitive descriptions of the swingup and balance tasks form the design specification for the control algorithm that is to be written in NPL. Each task is fundamentally a discriminator. The swingup task amplifies the angular velocity of the pendulum to produce force. The balance task derives the output force by amplifying a combination of the angular position and angular velocity. The system selects which controller is active based on the position of the pendulum.

4.2.1 Inputs and outputs

Before creating the controller, the programmer must decide how to encode the inputs into signals in NPL and how to decode NPL signals into the output force. Options range from simple analog values to population density coding on multiple connections. The constructions developed in section 3.2.2 suggest a two wire encoding which can be viewed as a population density coding with a population restricted to the two extreme values.

The inputs of the NPL program are the four state values of the cart-pole system. Three of these values have an unbounded range and need to be transformed into bounded values for the rest of the system. The fourth, angular position, is problematic in that it wraps around in value, and jumps discontinuously from $2\pi$ to 0.
In the first three cases, the two wire encoding is appropriate. In the case of \( \dot{\theta}, x, \) and \( \dot{x} \), only the sign of the value is important to the controller. The encoding looks like figure 4-4. In fuzzy logic terminology, one node represents the positiveness of the value, while the other represents negativeness.

The fourth case requires some extra care. A simple positive/negative encoding will not work for an input value that can wrap around. A common method of solving this problem is to encode the angle as its sine and cosine. Here, we can use gaussian nodes that respond to ranges around specific values, as illustrated in figure 4-5. We use several nodes instead of just two so that each node strongly encodes a particular direction. This is similar to the fuzzy logic concept of dividing a range into many fuzzy subsets.

The output of the program is the commanded force on the cart. It is the sum of the outputs of each of the subcontrollers in the system, as illustrated in figure 4-6. Because of the encoding, there is an inherent limit to the magnitude of the output. Unlike the outputs of some purely mathematical controllers, the output of an NPL program constructed with the positive/negative encoding can never blow up.
4.2.2 The swingup subcontroller

The first task in building the program is to create the swingup controller. Its linguistic description, from the beginning of this chapter, is “apply a force to the cart opposite to the angular velocity of the pendulum only when the pendulum is below horizontal.” The sentence breaks down into a simple sign discriminator and a disabling mechanism.

The discriminator is similar to that from figure 3-5(b). It is simply a pair of sigmoid nodes, one of which notes the positiveness of the signal, while the other notes negativeness. We could provide the input from the $\dot{\theta}$ input node, but this would lead to a small problem. Since $\dot{\theta}$ does not have a specific maximum or minimum, it is not clear what weight is required for the inhibiting connections to inhibit the controller completely. Any choice for the inhibition weight could be overridden by a sufficiently large value of $\dot{\theta}$.

The solution is to process the input to limit it to the range $[-1, 1]$. This is illustrated in figure 4-7. The first set of nodes encountered from the qd.theta input limits the signal, while the second set provides an output that can be disabled. The cross connection between the two layers can be understood in fuzzy logic terms. The input signal is fuzzified by the first two nodes. That signal is then defuzzified into a limited range, then immediately refuzzified by the second set of nodes. Effectively, each of the first pair of nodes is providing a positive and negative stimulus to the second pair.

The output of the second pair of nodes is provided directly to the output force. The multiplier in the connections was somewhat arbitrarily chosen, based on the mass of the system, to make the swingup portion look “right.” The outputs provide a maximum force of 40 Newtons on a cart that has a mass of 16 kg.

The proper connections for the inhibitor node are obvious when the description is specified in logical terms: “disable the controller when the pendulum is not in the bottom half of its range.” The inhibitor node must be a NOT gate, with unit inputs from the three nodes tuned to the lower half of the angular range. The inhibitor is the node in the bottom left of figure 4-7.
Figure 4-7: The swingup subcontroller
4.2.3 The balance subcontroller

The balance subcontroller is just as straightforward to construct as the swingup subcontroller. Two rules combine to provide balance. The first corrects for a non-zero angular position of the pendulum. The second corrects for a non-zero angular velocity.

Figure 4-8 illustrates the detection of the two error conditions and their influence on the output. The controller computes position error by comparing the two gaussian sensors on either side of the vertical sensor. The gaussian curves overlap, providing a strong differential signal. The signal limiter nodes for the velocity provide only a weak distinction between positive and negative values close to zero, so the discrimination must be amplified. The nodes marked “velbal” in the figure illustrate the amplification.

Because of the limited force output, the balance controller will only be successful
within a small range of vertical. The node labeled “/top” in figure 4-8 takes account of this fact by inhibiting the balancer when the pendulum is sufficiently off vertical.

Experimentation with the running system revealed that the valid balance range was slightly narrower than the width of the vertical gaussian node. It is possible to decrease the apparent size of the gaussian by subtracting a portion of the gaussians on either side. Because the main vertical gaussian node contributes a negative value, the small links from the outer gaussian nodes is positive.

4.3 Debugging and tuning the controller

The combination of the swingup and balance subcontrollers is enough to accomplish the complete task under most initial conditions, but there are a few situations under which the system fails.

4.3.1 Startup

The first problem occurs when the pendulum initially points straight down with no angular velocity. With no angular velocity, the swingup controller will not produce any forces on the cart, and the system remains still.

To knock the system out of metastability, a temporary force needs to be provided. Figure 4-9 shows the logic for the switching mechanism. The “still” node is simply a very tight gaussian around $\dot{\theta} = 0$. The “/luffing” node indicates that the pendulum is both pointing down and not moving, and therefore needs to be kickstarted.

The actual kickstart is implemented by an oscillating ring. The oscillation allows the system to start up without drifting too far in any particular direction.

Note that a simple push cannot completely solve the metastability problem. Since the pendulum can approach the top from either side, there will always be some set of initial conditions for which any given destabilizer will fail. However, without the luffing node and oscillator, a startup delay is very likely. With it, the metastable initial conditions are much less likely to be encountered.
Figure 4-9: Adjustments to the swingup controller. Logic to determine when the pendulum is “luffing” appears in the lower right. Logic for disabling swingup when the pendulum is already swinging too fast is in the center.
4.3.2 Overdrive

A more serious problem occurs if the swingup is too vigorous. If the pendulum has too much energy, it will swing through the balance controller and pick up even more energy on the next downward pass. Ideally, the swingup should pump in only enough energy for the pendulum to just barely reach vertical.

The total energy in the pendulum can be expressed as a combination of the kinetic energy and potential energy: \( E = \alpha \cos \theta + \beta \dot{\theta}^2 \), where \( \alpha \) and \( \beta \) are some constants based on the mass and length of the pendulum. In a traditional controller, this energy would be computed at each time step and compared to the total energy \( E^* = \alpha \) where \( \theta = 0 \) and \( \dot{\theta} = 0 \). If \( E < E^* \), energy needs to be pumped in. If \( E > E^* \), energy needs to be pumped out.

Unfortunately, the energy formula is difficult to compute using only NPL nodes. Such precise math is contrary to the NPL philosophy, anyway. We only need a way to bring the total energy close to \( E^* \), where the balance controller can take over. We do not need to hit \( E^* \) exactly.

A solution suggests itself through some brief analysis. First, energy is only pumped into the system when the pendulum faces downward. Second, the pendulum hits its maximum velocity when it is pointing straight down. Third, the force on the cart has its greatest effect on the energy of the system when the pendulum is straight up or down. Therefore, we can use the pendulum itself as a governor. If the swing velocity becomes too high, the kick from the swingup controller should be reversed.

Figure 4-9 also illustrates the governor mechanism. The nodes labeled “too fast” override the “dtheta” nodes by having larger weights connecting them to the “swingup” nodes. The bias on the governor nodes can be determined either through trial and error, or simply by allowing the pendulum to drop from vertical and observing the angular velocity of the pendulum as it reaches the bottom of its swing.
4.3.3 Canceling drift

At this point the pendulum reliably swings itself up and balances at the top, solving the original problem. However, once the task is complete, the cart tends to drift off toward infinity because there is no input from the position or velocity of the cart. Aesthetically, it would be nice to bring the cart to a stop after it has balanced the pendulum.

Here is where hierarchical structures come into play. On first consideration, it would seem like another pair of nodes between the $\dot{x}$ input and the output force node would solve the problem. However, applying a force directly to the output node actually has the opposite of the desired effect. Suppose the pendulum is balanced, and the cart is moving to the left. If a force to the right is applied on the cart, the pendulum will start to fall to the left. This will cause the balance subcontroller to provide an even stronger force to the left, and the cart will end up going faster to the left, rather than slowing down.

Instead, the velocity control must move up a level to become more abstract. Consider the balance nodes, which actively keep the pendulum balanced. If they can be fooled into balancing the pendulum slightly to one side, the cart will automatically begin to accelerate to that side to keep the pendulum “balanced”. The velocity controller must reach inside the balance controller in order to affect it properly.

Figure 4-10 illustrates the velocity controller in place. The controller reaches in to alter nodes within the balance system. One of the principles of NPL is that the subcontrollers are not enclosed in “black boxes”, but have their constituent parts exposed for other systems to influence. This exposure may seem antithetical to the idea of abstraction, but it is actually an artifact of the way data is computed in NPL. We will discuss abstraction more completely in Chapter 6.

Figure 4-10 also includes a set of nodes that will bring the cart back to the origin. It does this by altering the desired velocity in the same way that the velocity controller alters the balance position. If the cart is to the left of the origin, the position nodes cause the stable velocity to be positive. If the cart is to the right, the stable velocity
Figure 4-10: Hierarchical controller for velocity control once the pendulum is balanced
Figure 4-11: Recorded data for a run of the swingup task.

will be negative. The sigmoidal shape causes the cart to slow down as it approaches
the origin, and finally stop there.

Figure 4-11 contains graphs of the position of the cart, the angle of the pendulum,
and the force on the cart computed by the NPL program. The force produced by
the swingup controller is evident in the first portion of the figure. The gaps in the
force are caused by the inhibitor on the swingup controller. The dips in the force
are evidence of the swingup energy limiter. Finally, the pendulum balances, and
the velocity and position controllers bring the pendulum asymptotically back to the
origin.

4.4 Summary

The swingup and balance toy problem illustrates the use of NPL in the implementa-
tion of intuitive control. Individual tasks are mapped to collections of NPL nodes
with one inhibitor node per task to enable or disable the collection at the appropriate
times. Once the system is complete, additional controllers can be layered on top, providing high level control without disturbing the lower level controllers.

The techniques learned in this chapter will make the walking controller in the next chapter easier to understand.
Chapter 5

Structure of the Bipedal Walk Controller

This chapter presents the major contribution of this thesis: the decomposition of walking control into individual orthogonal tasks such that each joint performs exactly one task during each phase of the walk cycle.

Typical walking controllers are designed from the top down. The designer specifies a trajectory for the body and for the extremities, and then solves the dynamics equations to find the torques that will provide exactly that trajectory. Typically, large simultaneous equations need to be solved to balance the torques at each joint against each other and the desired goal. When it works, this method is elegant, if a bit complex. Unfortunately, any discrepancies between the modeling equations and the actual robot can render the final solution inaccurate.

The controller presented here is designed from the bottom up. Each joint is assigned a single task during each phase, and each task is performed by a single joint. This restriction simplifies the problem in two ways. First, the equations for each joint do not depend on each other, and therefore are much simpler. Second, any problem with the resulting motion can be pinpointed and identified with a single controller for a single joint. A problem in the controller can be pinpointed easily simply by watching the gait and identifying parts that don’t look right.

Another aspect of the bottom-up design is that the final trajectory is not specified.
The controller is designed for error correction and stability through foot placement, which means that the location of the foot placement is determined by the current stability of the robot. Contrast this to the ZMP methods, which start with foot placement, and then adjust the rest of the body to restore stability.

5.1 The walk cycle

A bipedal walk cycle consists of two primary phases for each leg. During the stance phase, the foot is planted on the ground, supporting the weight of the body against gravity. During the swing phase, the foot lifts off the ground and swings forward to its new stance location. In human walking, the stance phase is just slightly longer than the swing phase. The overlap, where both feet are down and body support duties are shared by both legs, adds two more phases to the walk cycle. Thus, the four global phases are left support, transition to right support, right support, and transition to left support. Most gait analyses in the literature [Whi96] divide each leg into six or seven phases, but the main four are enough to understand the development of the controller presented here.

There are two ways to look at walking. One, used by the ZMP crowd and many others, is to view walking as an end-effector problem. They assume the stance foot is nailed to the floor, and work out the torques necessary in the joints to provide specific forces to the torso. The resulting torques may cause problems due to the fact that the stance foot is not actually affixed to the floor; the ZMP is a measure of whether the torques are in the safe range or not. To fix any excessive torques that cause foot rotation, the designer can either change the desired forces on the body, or use the torso as a reaction mass to balance out the excessive torques. In the reaction mass case, the final walker looks drunk as it throws its torso forward and back.

A much simpler way of looking at walking is to view the stance phase ballistically. In the pure form of this viewpoint, the feet can be reduced to single points, removing an entire degree of freedom. The walking algorithm relies on the momentum from the previous step to propel the body over the foot. Momentum for the next step builds
Figure 5-1: Kajita inverted pendulum model. All mass is concentrated at the body. $h$ is the height above the ground, $x$ is the horizontal displacement of the stance foot from the center of mass, $v$ is the velocity of the body, confined to a horizontal line, and $s$ is the length of the previous step. The $n$ subscript refers to the time of changeover at step $n$.

up during the second half of the stance phase, after the center of mass has passed over the foot. This view is called the inverted pendulum model.

### 5.2 The inverted pendulum model

The energy gained and lost during the stance phase of walking can be modeled by an inverted pendulum, as illustrated in figure 5-1. The base of the pendulum is located at the center of pressure. The weighted free end of the pendulum represents the center of mass of the body. During the first half of the stance phase, the foot is in front of the body, and the representative pendulum slopes backward. Momentum carries the body forward, continually slowing down until the pendulum crosses the vertical orientation. During the second half of the stance phase, the body falls forward, building up the momentum required for the next transfer. Bipedal walking is thus a transfer of energy between the kinetic energy of the body, and the potential energy represented by the horizontal distance from the center of mass to the center of pressure.

The pure inverted pendulum model [KYK92] represents an ankle that is completely limp during the stance phase. The motion during stance phase is effectively ballistic, as there is no way to control the location of the ZMP. The only control becomes the step length and time. Kajita describes a measure of “orbital energy”
that remains constant during the entirety of stance phase, as long as the legs adjust their length so that the body remains at a constant height.

\[ E = \text{constant} = \frac{1}{2}v^2 - \frac{g}{2h}x^2 \]

The change in orbital energy from one step to the next can be computed at the instant of changeover, by examining the difference between the horizontal displacement of the center of mass and the two feet, assuming the changeover is instantaneous and the horizontal velocity of the body does not change:

\[ \delta E = E_n - E_{n-1} = \frac{g}{2h}(x_n^2 - (s_n - x_n)^2) \]

The change in orbital energy is measured at the instant of transfer between one stance foot and the next. If the center of mass is closer horizontally to the front foot, the orbital energy increases and the velocity of the body during the next step will be faster. If the center of mass is closer to the rear foot, the orbital energy decreases and the next step will be slower. The actual step length is irrelevant in this model. Only the position of the center of mass relative to the two feet has any bearing on the change in orbital energy. Thus, accuracy in timing is more important than accuracy in position for foot placement.

The controller implemented in this thesis takes the general concepts of foot placement and timing from Kajita’s work, but does not use the exact orbital energy formula for a few reasons. First, the controller implemented here produces a “soft” robot, in the sense that its response to disturbances in its limit walk cycle are indicative of a weak spring rather than a stiff one. Weak springs prevent a disturbance from creating huge compensating torques at the joints, but they also cause the biped to bob quite a bit as it walks. This is a natural-looking gait, but it invalidates the assumptions Kajita uses to calculate orbital energy.

Second, Kajita assumes that the transfer of weight from one foot to the next is instantaneous. Again, this posits an extremely rigid controller. In the algorithm
implemented here, there is a significant period of weight transfer, where both legs share weight-bearing responsibility. Weight sharing allows the center of mass to move relative to the feet without much change in the velocity. Thus, the foot placement mechanism can be sloppy. A delay in the rear foot pickup corrects for a placement of the foot that was too far away or too early.

This sloppiness combined with a strategy for error correction allows the system to be robust, even under inexact control. One of the principles outlined in chapter 1 declares that there is no reason to use complex mathematical equations when simpler ones are sufficient. Since the Kajita model is partially invalidated by the system and there exists an error correcting mechanism, the function describing foot placement need only be vaguely representative of the ideal foot placement function.

Nevertheless, the inverted pendulum concept is extremely useful, and the idea that walking control is a matter of foot placement and timing forms the basis of the algorithm implemented in the NPL.

5.3 Bipedal walking controller structure

The walking controller consists of several small controllers, each defined for a single joint at a specific part of the walk cycle. To a first approximation, the stance phase acts as an inverted pendulum, while the swing phase implements the timing required for a stable walk. Neither of these tasks requires much activity from the ankle, leaving it available for fine control and adjustments. Ankle adjustments were added only as necessary in the debugging and tuning phase of the construction process.

5.3.1 Inputs and coordinate transform

The inputs to the controller consist of the positions and velocities of the six joints and body pitch angle, the forces on the foot, and the horizontal velocity of the body. From a programmer's perspective, it would be much easier to deal with the cartesian coordinates of the feet instead of the angles of the joints. The first part of the NPL controller uses the sine and cosine approximations from section 3.2.2 to transform the
Figure 5-2: Transformation from input angles to horizontal and vertical displacements between the ankle and the hip.

angles into horizontal and vertical displacements between the feet and the hip.

Figure 5-2 illustrates the transformation. The ankle, knee, hip, and body pitch angles are added appropriately to generate the global angles of the thigh, shin, and foot. Approximate sines and cosines of the global angle of the thigh and shin then generate the hip-centric cartesian coordinates of the ankle. The results are labeled foot.x and foot.z in the diagram. The height of the body can be approximated by the foot.z measurement of the stance foot. The distance between the feet is the difference between the foot.x measurements of the two feet.

Special mention should be made here about the cosine approximation and the vertical distance calculation. The gaussian curve actually approximates the function \((\cos \theta + 1)/2\). It is this value that appears at the foot.z node. Multiplying by 2 and subtracting 1 to get the proper value would be easy in NPL, but it is not done here. The reason has to do with the philosophy behind programming in NPL. In most cases, the vertical distances of the two feet will be subtracted from each other and compared against an arbitrary value. The subtraction removes the bias from the approximation, and the arbitrary value makes the multiplier irrelevant. The
vertical cartesian coordinate is viewed directly only to determine whether the body of the robot is too high. This case, presented in section 5.3.3, further transforms the computed “height” before comparing it to a fixed value. For such a comparison, the transformed data need only be monotonic with the original function.

The impulse to have perfectly accurate data at every step along a computational path is understandable, but as this thesis demonstrates, it is not always warranted. The computational effort required to compute exact values is often better used elsewhere.

5.3.2 State machine

There are four primary states in the walking controller, representing the two swing phases and the two transitions between them. The right swing phase begins with the lifting of the right foot off the ground. The foot swings forward until it reaches some
trigger point at which the state changes to the transition. During the transition, the right foot lowers to the ground, and begins to share the weight with the left foot while the body swings forward. At some point, the body has swung far enough, and the swing and transition states repeat with the other foot.

Figure 5-3 illustrates the complete state transition controller. The nodes labeled $L$ swing trig and $R$ swing trig identify the start of the swing phase. They examine the sum of the horizontal foot offsets. When the hip is closer to the rear foot, the negative contribution to the sum from the front foot outweighs the positive contribution from the rear foot. As the hip slides forward, the sum increases, crossing zero when the hip is exactly between the two ankles. A threshold is specified by the negative contribution from the toe-off constant parameter. When the sum crosses this threshold, the state transitions into the swing phase. The effect of changing the toe-off parameter is discussed in section 6.2.1.

The end of the swing phase is identified by the swing end trig nodes in figure 5-3. These nodes examine the difference between the two horizontal offset measurements, indicating the distance between the two feet. They compare the calculated distance to a value that is a function of the current body velocity: $s_c + s_d \dot{x}$, where $s_c$ is some constant distance and $s_d$ is a coefficient on the body velocity. Thus, the distance the swing foot travels before the end of the swing phase grows as the walk gets faster.

The values $s_c$ and $s_d$ are indicated by the nodes labeled dstep/const and dstep/vel, respectively. They constitute two of the four parameters that determine the stability of the walk. Their significance is discussed more fully in section 5.4.2.

The node with an 'x' in it is a special node that calculates the product of its inputs instead of the sum. It is used here only as a convenience for debugging. Ordinarily, the coefficient $s_d$ would show up as a weight on a connection. The node with a '!' in it is a special reset node. It is rigged so that the controller resets whenever the robot starts to fall backwards. On reset, the node initializes the state of the controller to $R$ swing.
5.3.3 Stance phase: Height and pitch maintenance

The primary purpose of the stance leg in an inverted pendulum controller is to maintain the height and pitch of the body. The hip joint is closest to the body, so we assign it the job of maintaining the desired pitch of the body. The knee joint handles the task of counteracting gravity on the body.

Pitch maintenance: Hip

Figure 5-4 shows the hip controller for the stance phase. The controller portion is a simple discriminator. The inhibitor is filtered to avoid sudden jerks when the state changes. The node in the center of the figure is the standard low pass filter biased so that its value is 0.5 in the absence of any inputs. The right swing state drives the
low pass filter to one, while the *left swing* state drives it to zero. The gaussian nodes are set up so that the right side shuts down during right swing, the left side shuts down during left swing, and both sides are damped to equal values if the low pass filter reaches 0.5 during the transitions.

The weight of the final connections to the force output reflects the desired maximum torque at the hip. It was made just barely large enough to balance the weight of the body when the body is pitched forward or backward by a few degrees. With smaller values, the body tends to flop over. Much larger values make the body so unnaturally stiff that it doesn’t change pitch at all during the walk, or causes the body to vibrate as the natural frequency approaches the time delay of the control network.

**Height maintenance: Knee**

The primary purpose of the knee during stance phase is to keep the body off the ground. Two components comprise this task. The major component counteracts the weight of the body. The minor component attempts to keep the body at a desired height.

The appropriate force on the knee to counteract the weight of the body is of the form $mgl \sin \theta$, where $\theta$ is the global angle of the thigh, $l$ is the length of the thigh, and $mg$ is the weight of the body. For the Spring Flamingo model, this works out to about 56Nm. Because of the $\sin \theta$ factor, the actual torque output due to this controller rarely reaches its maximum.

The gaussian inhibitors and the weight of the inhibition have been carefully tuned so that when the state is in transition, the force on each knee is close to one half of the normal output. This enables the controller to stand on two legs without collapsing or shooting into the air.

Simple anti-gravity is not enough to maintain a stable height off the ground. We must layer a weak height controller on top of the anti-gravity system in order to correct for errors in height. A measure of height off the ground is required for this system. The $Lfoot_z$ node in left stance and the $Rfoot_z$ node in right stance contain
Figure 5-5: Knee control during stance phase. The box at the top center contains logic to choose from $Lfoot_z$ or $Rfoot_z$ depending on the current state of the walk cycle.
this information. The computation block in the upper central portion of figure 5-5 selects the appropriate node depending on the state of the system. Its output is fed to a simple discriminator layered appropriately to add a very small adjustment to the knee force.

The height controller on the knee works properly only when the shin points forward. Because the height controller is very weak compared to the anti-gravity controller, brief periods where the shin points backwards will have little effect. However, both this controller and the swing controller force the gait into a kind of “Groucho walk.” This issue will be discussed more fully in the next chapter.

5.3.4 Swing phase: balance and control

During the swing phase, the swing leg needs to get the foot to the appropriate location for the next step without stubbing the toe on the ground. There are three tasks involved: swinging the leg forward, keeping the foot off the ground, and keeping the foot level with respect to the ground. Each of the three tasks can be assigned independently to the hip, knee, and ankle, respectively. The hip joint swings the foot forward, the knee joint keeps the foot a certain distance off the ground, and the ankle joint keeps the foot parallel to the ground. Similarly to the height adjustment in the stance knee controller, this simple assignment has consequences on the style of walking possible with this controller.

Foot control: Ankle

The simplest swing controller is the ankle controller. Its only task is to keep the foot level. It is simply a PD controller tuned for the impedance of the foot, as shown in figure 5-6. The impedance of the foot in the air is much less than that of the body when the foot is on the ground, so the output connections have a much smaller weight than the other controllers.

Because the foot is already level when it is grounded, there is no need to disable the controller during the stance phase of the walk cycle. For stance phase on non-
level ground, however, the ankle controller should be disabled. The angle of the foot resting on the ground would then give an indication of the general slope of the ground, which could be used to adjust other aspects of the swing leg, such as foot angle and height above the ground.

**Vertical control: Knee**

The knee controller handles the height of the foot off the ground. It performs its task by comparing the heights of the two feet against a constant, labeled `stepheight` in figure 5-7. The controller portion is the PD controller from figure 3-7(b) which calculates the derivative of an input by subtracting the value from a time-delayed version of itself. The inhibitor is set up so that the controller is only active during the swing phase.

In order for this controller to work properly, the gait must avoid singularities. The global angle of the shin must remain positive, sloping down and toward the back of the robot. Also, the distance from the knee to the desired height must remain smaller than the length of the shin. This constraint forces the hip to be lower than it would be at full leg extension. The combination of these requirements produces
Figure 5-7: Knee controller during swing phase. The derivative of height is calculated from the changing height value by the sum nodes in each of the blue side boxes.
a “groucho” walk, where the knees are bent throughout the walk cycle. In normal human walking, the knee straightens during the stance phase, and extends fully at the end of the swing phase. At heel strike in a normal walk, the knee flexes slightly both to absorb the shock and to guide the force of impact into forward momentum. Groucho walking avoids the sensitive timing issues around the knee flex, making the gait more resilient to errors in timing and placement.

Another issue with this controller is the amplification of noise in the calculated derivative. Part of the problem can be dealt with by filtering the inputs carefully. An alternative would be to measure the angle velocities directly and perform calculations on them. Unfortunately, the derivative of a function like $z = \cos \theta$ is $\dot{z} = -\dot{\theta} \sin \theta$, which is beyond the simple math implementable in raw NPL nodes. Much of the error introduced by the calculated derivative is absorbed by the bandwidth of the actuator and by the environment, which acts like a double integration. An analysis of the effect of sensor noise on the controller is discussed in section 6.1.5.

**Horizontal control: Hip**

The task of the hip controller is to swing the foot horizontally at a specified rate. The basic structure of the controller is similar to that of the knee controller except that the position connections have been removed. The hip controller does not care about position, only velocity.

The desired velocity is computed similarly to the desired step length in the state machine controller: a constant plus some multiple of the body velocity. In this controller, there are two sets of inhibitors. The upper one in figure 5-8 turns the entire controller on and off. The controller is active during the swing phase and the transition to stance phase. A toe sensor keeps the swing controller turned off until the foot has lifted off the ground. Without the sensor, the hip would attempt to drive the foot forward while it was still on the ground. The drag produced would slow down and destabilize the robot.

The lower inhibitor affects the desired velocity. It allows the desired velocity only when the state machine is in swing phase. During the transition phase, the inhibitor
Figure 5-8: Hip controller during swing phase. The blocks on the far left and right maintain a constant horizontal foot velocity, modulated by the swing velocity controller at the bottom.
drives the desired foot velocity to zero, slowing down the foot in preparation for heel
strike.

The connections running from the desired velocity blocks to the velocity controller
are very small because of the way the velocity is computed. The second sum node
is actually calculating \( v\Delta t \), where \( \Delta t \) is the update period of the controller. Thus,
the desired velocity must be premultiplied by \( \Delta t \), and is amplified by the velocity
controller by \( 1/\Delta t \) on its way to the output.

5.4 Tuning

With the structures above in place, the system is almost ready to take its first steps.
There are a few parameters that need to be tuned, however. They are the step length
constant and velocity factor, the swing speed constant and velocity factor, the step
height, the desired hip height, and the swing start trigger. All of these parameters
are easy to hand tune, because each of them has a specific effect on the robot.

5.4.1 Gait parameters

The step height parameter is the easiest to tune. The foot needs to be just high
enough during swing phase to avoid stubbing the toe. Higher values will cause the
robot to stomp when it puts its foot down at the end of the swing phase. Only a few
iterations were necessary to find the smallest safe value.

Hip height is a matter of preference. The robot turns out to be stable over a fair
range of values, and the failure modes at either end are obvious. When the robot is
too low, it spends a lot of extra motion trying to pick the swing foot up off the floor
and swing it forward. After a while, the robot sinks so low that it can’t get the foot
off the ground without tripping. When the desired height is too high, the robot snaps
its knees into the locked position. This destroys the “groucho walk” gait and breaks
the controllers. A good looking desired height is as high as possible without breaking
the groucho stance.

The swing start trigger is similarly flexible. Its effect is described more fully in
the analysis of chapter 6. If the swing starts too soon, the robot loses velocity very quickly and falls backwards. If the swing starts too late, the robot either stalls out with both feet planted firmly on the ground, or falls forward, depending on how much momentum was present.

5.4.2 Stability parameters

Finally, there are the four parameters that describe the swing leg motion and limits. They are the step length constant, the hip velocity contribution to the step length, the swing velocity constant, and the hip velocity contribution to the swing velocity. These parameters determine the stability of the walking gait, and are interrelated. Fortunately, each has a distinctive and intuitive effect on the appearance of the gait, so it is easy to determine which parameter needs to be changed in which direction simply by watching the robot fail. The online interactive system reflects changes immediately, so the debugging turnaround time is nearly instantaneous.

Step length parameters

In a natural human walk, both the stride length and swing speed grow as the speed of the walk increases. For an initial starting point, the step length parameters can be set to mimic the stride lengths for a human walk. Values of 0.1 for the step constant and 0.4 for the velocity coefficient reflect reasonable distances at various speeds up to the target of 1.2 m/s. Note that these values do not have known units associated with them. The distance measurements calculated by the coordinate transformation blocks are not in the same units as the inputs, as no effort was made to scale them to match the measurements of the robot. Instead, the parameters were selected by pushing the robot at various velocities and observing where the transition to stance phase occurred.

The constant term for the step length was chosen so that the initial step at low velocity fell approximately one foot-length from the starting position. The velocity coefficient was then adjusted so that at 1.2 m/s, the step length was approximately
two foot-lengths long. These distances reflect casual observations of human gaits at those speeds.

**Swing speed parameters**

For a given step length, stability is controlled by the swing speed. A faster swing speed causes the transition to stance to happen earlier. The center of mass of the robot will be further to the rear of the robot, allowing the robot to slow down. By contrast, a slower swing lands when the center of mass is close to the front foot, forcing the robot to speed up. The right velocity coefficient acts like a governor, slowing down the robot when it is too fast and speeding up the robot when it is too slow.

Adjusting the parameters is a very straightforward process. In the simulation, the initial velocity of the robot is very low. The first step is most sensitive to the constant parameter. If the parameter is too small and the swing foot takes too long to get into position, the robot will fall over forwards. If the parameter is too large, the foot snaps into place, locking the knee. This action breaks the controller, as described in section 5.3.4. The knee never unlocks, and the robot falls backwards. In either case, the change to be made to the parameter is obvious.

Once the robot takes a few steps, the speed increases, and the velocity coefficient term begins to have more of an effect. If the velocity coefficient term is not tuned properly, the robot will once again fall over. Tuning is performed exactly the same as with the velocity constant parameter. If the coefficient is too large, the knees will straighten, and the robot will stop or fall backwards. If the coefficient is too small, the robot will either fall forwards, or gradually sink as the feet fail to keep up with the center of mass of the body.

Using the methods above, the constant and coefficient parameters for a stable swing velocity were found. They had the values 0.3 and 0.7, respectively. Again, the numbers are of arbitrary units, depending on the exact measurements of the robot and the scaling implied by an NPL discriminator.
5.4.3 The walking gait

Figure 5-9 illustrates the gait produced by the NPL algorithm with the parameters set to the values described above. It represents a walking gait with a duty cycle of 60%. In other words, each leg is on the ground for 60% of each cycle, and in the air 40% of the cycle. The right and left legs are both on the ground for 10% of a cycle twice per cycle. Walks always have a duty cycle of 50% or more. Runs have a duty cycle of less than 50%.

Because there is no torque on the ankle except when it is in the air, the center of pressure always lies below or near the ankle during single support. Unlike the ZMP algorithms which spend a lot of computational effort attempting to maintain a non-rotating foot, the NPL walking algorithm gets a stable foot for free. Stability is maintained not through clever contortions of the torso nor through adjustments in ankle torque, but through error corrections in foot placement.

5.5 Additions

The walking algorithm as presented above is stable and can handle some minor perturbations, but it suffers from a few deficiencies. First, it does not handle vertical perturbations very well. If the hip height is too low, there is only so much adjustment the knee can perform. The angle of the shin gets steeper and steeper over time until
the algorithm can’t lift the foot high enough or bring the foot forward fast enough to remain stable. A possible solution is to provide some torque at the ankle during stance phase to aid in height adjustment.

The second problem with the algorithm is that it tends to stub and lightly drag its toe when it initially lifts the swing foot off the ground. This has two effects. It slows the robot down, and it can cause the hip height to sink suddenly. The solution here is to provide for toe-off. Toe-off is the phase in normal human walking during which the rear stance foot rises onto its toes just before the swing foot lands. This action provides some energy to initiate the swing phase, and also provides a small push to start the stance phase on the other foot.

Each of these problems can be resolved with a new subcontroller for the ankle torque in the NPL algorithm. These controllers are illustrated in figure 5-10. Also included in the figure is the original level foot angle maintenance controller from figure 5-6.

5.5.1 Ankle adjustments to height

The central portion of the figure contains the adjustments to the height. At the exact center is a copy of the height discriminator that was explained in section 5.3.3. The outputs from this discriminator originally influenced the knee during stance phase. The same discriminator is used here again to add a gentle torque to the ankle during stance phase. Two nodes inhibit the discriminator for separate reasons.

First, these adjustments are only applicable when the appropriate foot is on the ground. The inner of the two inhibitor nodes disables the adjustment controller when the foot is in the wrong state.

Second, it was found experimentally that the system was vulnerable to collapse if adjustments were made in the first part of the stance phase, before the center of mass crossed over the stance foot. The outer of the two inhibitor nodes compares the horizontal displacement between the foot and the hip, disabling the controller until the hip crosses a point indicated by the ank adj parameter. This parameter was tuned to be as small as possible without causing the system to collapse. In the figure, the
Figure 5-10: Complete set of ankle controllers with foot levelers at lower outside, toe-off mechanism at upper outside, and height adjustments in middle.
connection from *ank adj* looks like it connects to the inner inhibitor; the connection actually passes behind it to the outer inhibitor.

The contribution of this controller to the ankle torque is small. In particular, the weight of the connection was chosen so that at maximum output, the torque at the ankle is not enough to cause the foot to start rotating. This is an example where the built-in clipping functions of NPL sigmoid nodes becomes very useful.

### 5.5.2 Toe-off

The outer top portion of figure 5-6 illustrates the toe-off controller. Toe-off in this algorithm is a function of the horizontal distance between the hip and the ankle. This reflects toe-off in human walking gaits. When the step size is very small, there is no toe-off. Instead, the foot lifts straight off the ground as it did prior to the addition of this controller. For longer steps, however, the controller waits for some trigger point, then provides a very strong torque to the ankle. Unlike the height adjustment output, the toe-off output is large enough to cause the foot to rotate onto its toes. The start of the rotation causes the *footang* node from figure 5-3 to trigger the transition state, turning off the toe-off controller. This interaction ensures that the torque applied to the ankle will never cause the foot to spin out of control.

The result of the toe-off is a cleaner transition from stance phase to swing phase, reducing toe-stubbing behavior and providing a boost to the height of the hip in preparation for the next phase.

### 5.6 Summary

This chapter described a common-sense analysis of how a stable walk might be performed. The NPL language allows straightforward transcription of the verbal description to a motor controller algorithm.

The exposed parameters of the algorithm were carefully chosen to have intuitive physical meanings. This allowed easy tuning of the parameters through common sense reasoning and observation of the failure modes of the gait.
Table 5.1: List of node types in the final walking controller. Last four columns indicate the number of adds and subtracts, multiplies, divides, and exp() calls required to compute the function of the node.

5.6.1 Statistics

The final controller consists of 158 nodes, with a total of 321 connections between them. Table 5.1 lists the type and number of nodes used in the controller. Although there are 23 input nodes, some of them register the same variable to make the controller easier to read. The actual input variables are the six joint angles and the pitch of the robot, the derivatives of those variables, the force on the toes of each foot to detect toe-stubbing, and the body velocity in the horizontal and vertical direction, for a total of 18 distinct input variables. The two multiply nodes and the reset node are special nodes described at the end of section 5.3.2.

Of the 321 connections, 246 are either +1, -1, or large negative inhibition con-
nections. Of the remaining 75 connections, 30 have unusual weights because they connect to the outputs. A total of 45 connections, or 14%, have specialized weights. Most of these are hierarchical connections that alter other parts of the system, or connections that have to do with calculating derivatives.

5.6.2 Computational complexity

Once an NPL program is complete, the computation represented by the nodes and the connections can be extracted. Each node needs to perform \( n - 1 \) additions, where \( n \) represents the number of source connections into the node. Once the sum of the sources has been computed, each type of node performs further operations to compute its output value. The number and type of operations required by each category of node is listed in table 5.1.

In the final NPL algorithm, 32 nodes have just one input, 31 have two inputs, 22 have three inputs, 26 have four inputs, 7 have five inputs, 3 have six inputs, and the remainder have no inputs. This makes a total of 196 additions to compute the sums of the inputs to the nodes. A total of 321 connections would suggest that 321 multiplications are required to compute the connection values, but 214 connections have weights of +1 or −1, which do not require multiplications. The remaining 107 multiplications are required.

Adding those sums and products to the requirements specified in table 5.1 yields the total amount of computation required in each time step: 361 additions and subtractions, 200 multiplies, 78 divides, and 82 \( \text{exp()} \) calls. The simulation employed a time step of 2 milliseconds, which by modern standards is plenty of time to perform the computation required.
Chapter 6

Analysis

In this chapter, we discuss several aspects of the walking gait and the controller. In particular, we will examine the stability of the walk, explore the space of the parameters of the algorithm that produce stable gaits, and explore the variation in gaits that can be produced by the algorithm.

6.1 Walking Gait

The walk cycle is illustrated in figure 6-1. It shows the torque outputs of the NPL walking controller for a single leg. The leg begins in stance phase, transitions to swing phase, then back to stance.

6.1.1 Ankle torque output

The ankle torque is the easiest plot to understand. It is nearly flat except for two places. The negative bump just after heel strike is the result of the attempt by the derivative portion of the PD controller to slow the rotation of the foot just after heel strike. The positive rise at the end of the stance phase is the result of the toeoff controller as it transfers the weight of the robot to the toe in preparation for the transition to swing phase. The drop in this torque, which occurs at the state change, allows the toe to leave the ground.
Figure 6-1: Plot of the walking cycle. Output torques for the hip, knee, and ankle are shown with the active state and a representation of the pose of the robot. The toe-off and heel strike instants are indicated by vertical lines through all graphs. The small anomaly just after toe-off in the hip torque was caused by a brief toe stub on this particular step.
6.1.2 Knee torque output

The knee is similarly easy to understand. During stance phase, the force on the knee is proportional to the cosine of the angle of the shin. It is at its greatest magnitude just after the transition to stance phase when the foot is in front of the body and the shin is at its steepest angle.

Just after the transition to swing phase, the knee flexes back to bring the foot off the ground. Once the foot is stabilized at its desired height, only minor corrections are needed until the stance phase begins again.

6.1.3 Hip torque output

The hip has torque contributions from several different controllers, so its curve is more complex. The major action in the hip controller occurs during the transition to stance phase on the right side of the plot. When the controller changes to the transition state, the desired horizontal velocity of the swing foot drops to zero. The hip controller throws a large torque on the hip to attempt to push the leg backwards. Once the transition to the stance state occurs, the pitch controller takes over. The two humps in the hip torque at this point are from the attempt to straighten the body as the other leg lifts off the ground and begins to swing forward.

The action in the hip torque on the left hand side begins when the toeoff mechanism kicks in. The hip begins to push forward as the toe pushes down. Part of the purpose of the toeoff mechanism is to give the stance leg a head start in preparation for swing phase. The hip extension here is part of that preparation. Just after toeoff, there is a sudden drop in the hip torque. This was due to the fact that the robot briefly stubbed its toe at this point, and the toe sensor turned off the hip torque to avoid jamming it into the ground.

6.1.4 Gait Stability

One measure of gait stability is the controller’s response to disturbances. Figure 6-2 shows various combination plots of the hip, knee, and ankle joints. The initial state
Figure 6-2: Combination plots of various joints. Initial conditions (marked with an x) quickly converge to the steady state walking cycle (red).
Figure 6-3: Limits to recoverable disturbances in different phases of a step. The stick figure indicates the state of the robot at each phase. The middle line indicates the measured velocity at that instant, while the upper and lower lines indicate the maximum and minimum velocities for which the robot recovers into a stable cycle.

of the robot is with its feet together, moving forward at a small velocity. The initial state can be considered a disturbance, as the state of the robot in its stable walking cycle does not pass through the initial state. It is clear from the diagrams that the controller very quickly converges from the initial state to the stable walking cycle.

The steady state cycle can be envisioned as a valley in phase space. If the state of the system falls anywhere within the valley, it will converge down to the stable cycle. Another way to examine stability is to perturb the robot from its steady cycle at various points and determine whether it returns to the steady cycle. This process determines the distance between the bottom of the valley and the crest on either side.

To explore the region of recoverability, we allowed the robot to settle into its stable cycle, then abruptly altered the velocity of the hips in some direction. A search was performed to find the maximum velocity change in the given direction for which the controller was able to recover. The robot and controller combination is more sensitive to disturbances in some phases of the walk cycle than others, so the response to disturbances was examined at different times during a single step.

Figure 6-3 illustrates the results of the disturbance test. Seven equally spaced instants within the single step are indicated by the stick figure sketch. Horizontal
and vertical disturbances were tested at each instant, and the maximum positive and negative change in velocity were recorded.

The plot on the left shows the maximum and minimum recoverable horizontal speeds. Note that the maximum recoverable impulse is at its greatest when the swing leg is nearly halfway through its swing phase. The leg swings out so fast that it snaps straight and locks, as illustrated by the arrow in figure 6-4. The jolt when it lands is enough to slow the system down to about 2m/s, from which it is able to stabilize back to 1m/s normally. The worst point in the cycle to push the robot is when both feet are on the ground. After a very large push, the controller is not able to get the rear foot off the ground fast enough to swing it into place for the next step.

For impulses slowing the robot down, the controller is at its most sensitive during and just before the double support phase. Force control of the body when both feet are on the ground is actually a very complex task. The NPL controller ignores the complexity almost completely by relying on momentum to carry it through the double support stage. The controller is relatively helpless if the forward velocity is stopped during double support: the controller is not sophisticated enough to know how to generate forward forces with both legs on the ground. If the robot is on one leg in the early or middle part of the swing phase, however, the controller is very good at recovering, as long as the push does not send the robot backwards.

The plot on the right in figure 6-3 illustrates the maximum and minimum recoverable vertical disturbances over the different phases of the walk. The robot is
reasonably and consistently good at recovering from downward disturbances. The controller is very loose in the vertical direction, as illustrated in figure 6-5. It takes several steps to recover its original height.

Upward pushes from below are trickier, as the robot is already at almost full extension, and a large enough push will actually launch the robot into the air. The large spike in the figure for the double support stage is a bit of a fluke. The controller in this phase is pushing with reduced strength on each knee; the resulting airborne spasm is diminished compared to other phases of the walk. By the time the robot lands again, the legs have settled down, and the controller can recover into a normal walk.

6.1.5 Noise

Because of the way that derivatives are computed in the knee and hip controller systems, the controller is susceptible to noise on the inputs. For the most part, the controller relies on the characteristics of the physical world to damp out the fluctuations caused by input noise. The double integration implicit in the translation from force to position smoothes some high frequency jitter in the output. Although not modeled, the limited bandwidth of the actuators would correct much more noise.

We tested the effect of sensor noise on the stability of the system by applying a random offset to every sensor value at every time step. At a maximum offset of about 1% of the normal range of the sensors, the robot was still stable. As we increased the
noise, it caused the robot to look like it had weak knees: steps varied in length, and
the hip height wandered around a bit. At about 1.5% noise, the robot was unable to
take more than a few steps before sagging into the ground.

6.2 Effects of changes in selected parameters

One advantage to the method of hand-designing a controller in NPL is that the
parameters of the algorithm are both meaningful to the human programmer and
have distinct and predictable effects on the robot. Some parameters, like the step
height, have a narrow range of stable values. Too low, and the robot regularly stubs
its toe. Too high, and the foot stomps with too much velocity when the algorithm
gains the stance controller on the knee. Other parameters, however, have a wide
range of useful values, which produce various effects on the gait.

6.2.1 Velocity: the toeoff parameter

The toeoff parameter was introduced in section 5.3.2. It specifies the start of the
swing phase in terms of the offset of the hip from exactly between the feet. As the
parameter gets larger, the controller waits longer in double support before lifting the
back foot. The robot does not lose as much kinetic energy during the weight-sharing
double support phase as it would if all of the weight were carried on the front foot.
In terms of the inverted pendulum model, the base of the abstract pendulum is closer
to vertical under double support than under single support. Therefore, under double
support, the kinetic energy of the body does not change drastically, whereas under
single support, the body would slow down as it attempts to rise over the support leg.

The further the hip travels under double support, the less time it will take under
single support for the center of mass to pass over the center of pressure, and con-
sequently, the faster the robot will travel. Therefore, the toeoff parameter acts as a
kind of speed knob. The effects of this speed knob can be demonstrated by slowly
increasing the value of the toeoff parameter and recording the characteristics of each
step.
Figure 6-6: Plot of step length and step time versus settings of the \textit{toeoff} parameter. Projections onto the sides of the graph cube illustrate the various two-parameter relationships. Dark layers in the central plot illustrate the trumpet-shape structure. Data are taken from 41,580 consecutive steps in a single run.
The results of this test are shown in figure 6-6. The \textit{toeoff} parameter was initialized to have the value 0.2, and slowly increased as the robot took 41,580 consecutive steps. The step length and step duration were recorded for each step. Each dot in the figure represents the measurements from a single step. The vertical axis represents the value of the \textit{toeoff} parameter, and the horizontal axes represents the step length and step duration for that step. The central mass in the figure is the 3-D plot; the other clusters are the projections of that data into each of the pairs of axes. The dark bands in the central mass highlight the fine structure at equally spaced parameter intervals. The velocity of a particular step can be read off the bottom projection by dividing the step length by the step duration. Velocities ranged from 0.4m/s to just over 1m/s for this trial.

At low velocities, the packing of the dots is very tight, indicating a very stable walk. At about 0.8m/s, however, the step locations begin to diverge. The walk remains completely stable, but the step length and time undergo chaotic bifurcation. The 3-D plot is actually trumpet-shaped. Eventually, the steps become erratic enough to topple the robot.

6.2.2 Stability: step length and swing speed parameters

In designing the walking controller in the previous chapter, we initially selected values for the step length parameters that would produce an aesthetically pleasing step length, then tuned the swing speed parameters until the walk became stable. The question can be asked: were we just lucky that the step length parameters happened to work out, or is there a wide range of parameters that produce a stable walk?

To answer this question, we performed a Monte Carlo test. Random values were presented to each of the four parameters, and the controller was allowed to run for 30 seconds. If the robot fell over within those 30 seconds, it was classified as unstable; otherwise it was classified as stable. Approximately 1% --- 402 of 35,057 trials --- produced stable walks.

Understanding the results requires visualizing four-dimensional data. Figure 6-7 shows two projections of the same data set. The projection on the left shows the
Figure 6-7: Stable parameter values. Data is four-dimensional; each plot is a projection of the same data onto two of the four dimensions. Dots of the same color correspond to each other.

step length parameters. It is clear from this picture that we were not just lucky with our choices of step parameter values in the previous chapter; any pair of parameters whose sum is less than about 1.1 has a corresponding set of swing speed parameters that will stabilize the robot.

Figure 6-7 also shows how the step length and swing speed parameters are related. Each corresponding pair of dots in the two graphs is marked in the same color. A low sum of parameters for the step length pairs with a high sum in the swing velocity, and vice versa.

At first glance, this result might seem counter-intuitive. Ordinarily, one would think that a large step length would require a large swing velocity in order to swing the foot across for the next step in time. The fallacy is the assumption that the velocity of the body is the same in both cases. In a stable walk for this algorithm, the swing speed is closely related to the velocity of the hip. The angle velocity of the stance leg is determined by the velocity of the body. The angle velocity of the swing leg is determined by the swing speed. If the position of the hips at changeover is to be near the midpoint of the two feet, then one must vary with the other.

At slower velocities, the body needs to build up more momentum in order to carry its mass over the new stance foot. The only way to build up that momentum is to take
longer steps. Again this appears to be the reverse of the description in the previous chapter, but remember that we are describing the steady state velocity for a fixed set of parameter values. Within a given set of parameters, fluctuations in stability are handled by lengthening and shortening the steps in the normal manner, and speed control is handled with the toeoff value, as described in the previous section.

At faster steady state velocities, the step length parameter values need to be smaller. One reason is that the equation for determining step length includes the velocity, and the coefficient needs to be small to prevent the step length from becoming overly large. Another reason is that the step length parameters only determine the start of the foot placement phase. By the time the foot hits the ground, it will have travelled an extra distance proportional to the swing velocity.

Figure 6-8 plots the computed swing speed against the computed step length, taking into account the body velocity for each stable set of parameters. The graph clearly shows the inverse relationship between desired step length and desired swing speed.
speed for the stable cycle of the groucho walk. It also illustrates two distinct clusters of stable walk parameters. The lower linear group corresponds to a normal groucho walk with a duty cycle of greater than 50%. The upper cluster corresponds to a groucho run. In the groucho run, the duty cycle is less than 50%. Here, the start of the swing phase for the rear foot begins instantly after the end of the swing phase of the front foot. The result is that the rear foot lifts up into the air before the front foot has a chance to land, providing for an extremely brief aerial phase.

6.3 Running

Groucho running is very different from normal running. The aerial portion of the run is primarily horizontal instead of parabolic. The duty cycle of the run is very close to 50%; very little time is spent airborne. And, like the groucho walk, the hips stay
relatively level, whereas in normal running there is a lot of vertical movement in the hips.

Groucho running, like normal running, allows much greater speeds than walking by allowing much larger strides. The step length need not be limited by the geometry of the robot, but can be extended by the distance travelled in the air. Unlike normal running, however, the running enabled by the NPL algorithm cannot wait arbitrary lengths of time in the air. Because of the state ordering of the state machine, the front foot must be on its way down for contact with the ground as the rear foot comes up; there is no way to “glide” in the air.

Because groucho walking and running are so similar, the controller that implements one also implements the other for free. In fact, some sets of step length and swing speed parameters have two stable cyclic attractors, one for running and one for walking. This fact was discovered accidentally while running the tests to generate figure 6-3. Some forward disturbances caused the controller to settle back into a running cycle instead of a walking cycle.

Instead of requiring an external “kick” to transfer from one gait to the other, it would be useful to build an addition to the NPL algorithm that can change from a walk to a run and back on command. In order to do this, we used one NPL flip-flop to remember the desired gait of the robot, and another NPL flip-flop to record the current gait style. Nodes between the two detect differences and apply corrections to various parts of the system to change the gait.

The first task is to automatically determine which gait the robot is currently displaying. To do this, we used hip height: the hip height is significantly lower during running than during walking. The lower hip height is an indicator for two reasons. First, the longer stride in a run requires more extreme angles in the legs, necessitating a lower hip height. Second, a greater vertical force is required during a run than a walk; the lower hip height engages the height correction controller, which provides that extra force.

To transition from a walk to a run, the robot needs to speed up. This can be accomplished by leaning forward and increasing the toeoff parameter value. Leaning
forward serves two purposes. First, it causes the center of mass to move forward, reducing the time the abstract pendulum spends slowing down and increasing the time it spends speeding up. Second, leaning forward engages the pitch controller, which is active on the hip during stance phase. This causes the stance leg to push backwards harder, encouraging the speedup.

To transition from a run to a walk, the robot needs to slow down. This is accomplished by leaning backwards and increasing the speed of the swing leg. Both operations increase the distance between the center of mass and the center of pressure at landing. This forces the robot to spend more time slowing down as it tries to rise over the stance leg. As the velocity slows, the step length decreases. Eventually, the foot placement is close enough that the body can stand up on the stance leg and resume the normal walk cycle.

The change to the desired pitch of the body cannot be instantaneous, as the sensitive pitch controller would go unstable. Instead, we send the desired pitch change through an NPL lowpass filter, causing the change to be gradual. Figure 6-10 illustrates the addition to the NPL controller to enable the run to walk and walk to run transitions. The NPL flip-flop on the left indicates the desired gait. The outputs of these nodes can be set from the command line of the NPL development environment.

Figure 6-11 illustrates the transitions between walking and running. Each transition takes approximately eight steps between the initial change in desired state and the actual change in gait.

6.4 NPL

This thesis introduced a new language and development environment for use as a programming tool for the walking controller. The NPL visual programming area is very similar to functional schematic programs like Simulink. In fact, NPL can be implemented as a restricted library of functional blocks within Simulink. The only differences are that the NPL development environment allows tighter visual packing of a program, offers more real-time visual debugging information, and encourages
Figure 6-10: NPL structures to enable walking and running transitions. The large box near the top contains the desired gait flip-flop on the left, and the current gait sensors and flip-flop on the right. The two nodes between the flip-flops implement the actions required to effect the change.
Figure 6.11: Automatic transitions from walking to running and back. The upper curve indicates horizontal velocity. The central curve indicates pitch of the body.
a qualitative, rather than quantitative way of thinking about motor control. NPL demonstrates that a highly restricted set of functional blocks is sufficient to implement motor control in a highly non-linear system.

6.4.1 Building block decisions

The restricted set of functional blocks was developed with some trial and error. Biological plausibility guided, rather than dictated the final structures. The most interesting case was the accumulator. It is possible to build a linear accumulator out of a sum node and positive feedback with delay. Unfortunately, such a structure does not play nice with the rest of NPL because its output is not restricted to the range between 0 and 1. One possible way to make the output behave is to pass it through a sigmoid node. This fixes the range problem, but the internal value can still blow up. If the internal value is extremely high or low, the output can be unresponsive, looking as if it is stuck at 0 or 1.

Another option to correct the output range is to make a leaky accumulator that gets pulled proportionately back to zero, given no inputs. This effectively limits the output range if the input domain is also limited. Unfortunately, such an accumulator cannot “remember” a value for very long, unless the restorative leak is very small. A small leak, however, leads to a large output range, which we were trying to avoid.

Simply clipping the internal value of a linear accumulator goes against the conscious decision to make all NPL structures have smooth functions. The final capacitor-like structure for the accumulator smoothly restricts the output between 0 and 1, allows the accumulator to hold a value for an indefinite amount of time, and retains near-linear properties close to the middle of its range.

6.4.2 Programming style

Programming in NPL is a lot like programming with op-amps in an electrical circuit. Its closest relatives are neural nets and fuzzy logic. Like most programming languages, it is not any more or less powerful than these languages. However, it is well structured
for the algorithm presented here.

One measure of the usefulness of a language for a particular algorithm is to examine the difficulty of implementing that algorithm in other similar languages. The difference between NPL and traditional artificial neural nets is fairly obvious. The algorithm developed in this thesis was based on knowledge of how the robot should react to changes in its physical state. The programming methodology involved building simple subcontrollers with specific tasks. The final relationship between input state and output torques falls out of the interaction between the subcontrollers; it was not a known relationship a priori. Traditional artificial neural networks use back propagation to find the weights in the network. They require explicit knowledge of the “correct” output values given a set of inputs.

Artificial neural networks differ from NPL programs in that the intelligence of the network is encoded in the weights between nodes rather than in the structure of the network. It is often extremely difficult to assign a meaning to any particular node in a neural network once it has been trained. Nodes in NPL programs have specific meanings. This makes it much more straightforward to add behaviors to an NPL program: the effects of a change in the system are obvious to the programmer.

NPL is much closer in style to fuzzy logic. A fuzzy logic program is straightforward to translate into NPL. The differences stem from the extra functions implemented by NPL nodes, and also by the style of programming that comes most naturally to each language.

Typically, fuzzy logic rules are defined as piecewise linear functions. The use of a sigmoid function for NPL allows constructions like the discriminator introduced in section 3.2.2 to be disabled smoothly.

Another difference between NPL and fuzzy logic is stylistic. It is possible to think of the two nodes in an NPL discriminator as members of a fuzzy set, one describing the positiveness of the input, the other describing its negativeness. However, in NPL, the outputs of constructions like these can be treated either as fuzzy set logic, or as actual values. In fuzzy logic, fuzzification and defuzzification are viewed as “heavy” operations, typically only performed at the beginning and end of the fuzzy logic
process. The conversion in NPL is much "lighter", meaning that values are often converted back and forth within an NPL program.

This mode of thinking defines how a programmer using each language would approach a particular problem. A fuzzy logic programmer typically thinks of breaking each input variable into several fuzzy ranges, and providing a table lookup of output values for each combination of ranges. Such table lookups are awkward in NPL, forcing the programmer to think of simpler functions to approximate. A fuzzy logic program would have output sets that included fuzzy terms such as "high output," "medium output" and "low output." NPL programs can do this too, but typically code output values directly.

Finally, NPL has an explicit method of recording the passage of time in the accumulator node. This node allows low pass filtering and oscillations to be implemented directly in the language. Similar constructions are possible in fuzzy logic networks, but require careful manipulation of recurrent functions.

6.5 Summary

The walking algorithm implemented in NPL has a number of desirable characteristics. It can recover from a wide range of disturbances and jolts to the robot. It has a simple parameter that can adjust the speed, although desired velocity is not a specific control in the algorithm. Also, the parameters that control the stability of the gait have a wide range of valid values, which allow the finding of a stable set of values in a short time.

The groucho walking gait controller is also able to handle running. Between walking and running, the controller is able to produce velocities that range from approximately 0.3 m/s through 2.4 m/s. NPL allows the programmer to build an intuitive controller to switch from running to walking and vice versa by making it easy to build hierarchical systems.

NPL is very similar to fuzzy logic, but stresses a different style of programming than is typically seen with fuzzy logic. NPL encourages the programmer to use
defuzzified values within the algorithm, and not just at the beginning and end of the program.
Chapter 7

Conclusion and Future Work

This chapter summarizes the new style of motor control design expressed by the walking algorithm, and discusses the role of NPL in enabling and encouraging the design. It also discusses future directions for the walking controller and for the NPL development environment.

7.1 Future work

Future work that can be done with the concepts in this thesis involve changes both to the walking algorithm and to the NPL programming environment.

7.1.1 The walking algorithm

The walking algorithm enabled both running and walking, but was constrained to a groucho-style gait in two dimensions on level terrain. Insights acquired during the development of the algorithm point to solutions for each of the possible extensions.

Non-level terrain

Chee-Meng Chew [CPP99] used the simulation of Spring Flamingo to explore one possible method for handling rough terrain in the absence of vision or other knowledge of the ground structure. As long as the ground is relatively smooth, the angle of the
stance foot provides an estimate of the slope of the ground in the immediate vicinity. This angle can be used to drive the desired height of the swing leg as the distance between the swing and stance foot changes.

The height of the robot needs to change as well. On steep uphill slopes, the height should remain constant until the center of mass has passed over the center of pressure. This would prevent excessive backward force by the stance leg. On downhill slopes, the same rule applies. Lowering the body after it has passed the center of pressure means that excess speed from the downhill motion can be partially absorbed in the first part of stance phase. The swing leg might also have to swing faster on downhill slopes than on uphill slopes to absorb or provide orbital energy, respectively.

**Straight-knee walking**

Normal human walking, in contrast to groucho walking, requires a complete rethinking of the knee controller for swing and stance phases. In particular, the knee is not used for weight support in straight-leg walking as much as it is in groucho walking. The stance controller needs to allow a partial collapse of the knee at the beginning of the stance phase, quickly followed by extension and locking. For the swing phase, a new controller is needed that does not suffer from the singularity issue of the swing controller presented here. Otherwise, the state transition logic for swing speed and step length should still hold.

**Full 3-D walking**

A full three-dimensional model requires the ability to stabilize itself in the lateral plane as well as in the sagittal plane. The same principles that enable stability in the two-dimensional walker, step length and swing speed, should also work to stabilize the robot laterally.

Another difference in the 3-D walker is the ability of the hips to shift the relative height of the legs on either side. This can be used to flatten the natural bob of the body during the walk, or it can be used to help lower the foot from swing phase to stance phase.
7.1.2 NPL

Improvements to the NPL programming environment primarily involve enabling more complex structures through the use of abstraction.

Libraries

Chapter 3 illustrated several common structures that were used frequently in both the swingup controller and the walking controller. The NPL editing environment would benefit from a library of such structures, so that the programmer would not have to recreate them by hand each time they were needed.

The presence of such structures in a library brings up some interesting management issues. If the programmer has several copies of a particular structure in the program, how should changes to the structure propagate? Sometimes the programmer will want to individualize a copy of the structure to customize it for a particular purpose. Other times, the copies should remain identical, where changes in one are immediately reflected in the others. An example of this is the symmetry of the legs in the walking controller. A change to the left leg controller should immediately be reflected in the right leg controller. The mechanics of the copying and the user interface make an interesting problem.

Views

The concepts of abstraction and libraries naturally lead to the concept of visual hierarchy: the desire to enclose a functional unit in a “black box” so that it can be used without concern for its internal construction.

Unfortunately, the black box concept falls down when confronted with the style of programming in NPL. Part of the power of NPL is the ability to reach into a structure that has already been built, so as to modify its properties from a higher level. A good example of this was the addition of the position control in the swingup task in section 4.3.3. If the balance control had been a black box, it would have been difficult to add the ability to cancel drift by faking out the balance controller.
Insight into why this style of design abstraction is a bad idea comes from an analysis of the structure of an NPL program. An appropriate question is, “what plays the role of a function in NPL?” The question is confusing because functional computation in an NPL program runs in the opposite direction to the flow of data. The value of the torque at the knee is computed by two “functions.” One is the block of nodes that is active during stance phase, and the other is the block that is active during swing phase. Each of these in turn “calls” on computational blocks further back in the data stream, for example the state machine and the cartesian coordinate transform block.

It is clear from the mess depicted in figure 3-11 that some kind of organizational abstraction is necessary, but the traditional visual hierarchical tool of black boxes with fixed inputs and outputs is inappropriate. Instead, a better tool would be a concept of “views.”

Views are slices of a complete NPL program that perform as a functional unit. All of the images in chapter 5 fit the concept of a view. Every node in a program belongs to a particular view, but any node in a view is available for use in another view. For example, the state machine appears in most views in the implementation chapter, but the state machine itself belongs in the state machine view of figure 5-3. Similarly, the $Foot_x$ node appears in several views as a source, but it belongs in the coordinate transform view of figure 5-2.

Views offer a way to reduce the complexity of an NPL program while still retaining the visibility and perspecuity that make NPL useful.

Learning

NPL is a pure programming language: there are no structures that allow tuning of the weights of any of the connections. Pure reinforcement learning algorithms are not appropriate for the large number of state variables involved in bipedal walking. However, the decomposition of the walking task into NPL presented in chapter 5 reduced the number of important parameters to a more manageable size. Here, learning algorithms are ideal, and can be used to tune the parameters to achieve speed, efficiency,
or any other characteristic that can be implemented as a reward function.

7.2 Conclusion

The primary concept that was presented in this thesis is a new way of thinking about dynamic motor control programming. The idea is to break the problem into independent subcontrollers so that a one-to-one correspondance can be constructed between joints and tasks. Such a correspondence reduces the mathematical complexity of the algorithm, and allows the programmer to create a very stable controller through the use of tight negative feedback loops. The resulting controller is robust, extensible, and easily editable. Furthermore, because of the joint-to-task correspondance, it is easy for the programmer to identify and correct problems in the control simply by observing the system in action.

The NPL language encourages this style of control by enforcing the use of simple functions, by allowing the operation of the entire algorithm to be visible at a glance, and by enabling a focus on rapid prototyping and intuitive structures. NPL consists of a small set of functions, which have limited ranges, vary smoothly, and include a sense of time. This thesis demonstrated that these simple functions are sufficient to implement the control of a complex non-linear control problem.

Because NPL is a qualitative language rather than a quantitative one, NPL is most suited to problems that do not require precision. Bipedal walking is an ideal problem because there are many definitions of a “successful” gait. Errors in one step can be corrected in the next. Designing for errors leads to an extremely robust controller that can recover from large disturbances.
Bibliography


[HKK+99] Qiang Huang, Shuji Kajita, Noriho Koyachi, Kenji kaneko, Kazuhito Yokoi, Hirohiko Arai, Kiyoshi Komoriya, and Kazuo Tanie. A high stabil-


