

Towards Robotic Bipedal Walking:
An Experimental System for Designing Control
Software for Dynamically Stable Robots

by
Richard S. LaBarca

Submitted to the School of Computer Science in partial fulfillment
of a Computer Science Senior Thesis

at

Carnegie Mellon University

May, 1998

Towards Robotic Bipedal Walking:
An Experimental System for Designing Control Software for
Dynamically Stable Robots

by
Richard S. LaBarca

Submitted to the School of Computer Science on May 11, 1998 in partial
fulfillment of the requirements of a Computer Science Senior Thesis

ABSTRACT

Bipedal robots and other dynamically stable machines require reliable, adaptable and expandable control algorithms in order to be useful. Creating such algorithms from scratch requires a significant amount of testing, feedback and reconfiguration. This paper describes a software architecture and experimental setup that facilitates the design and testing of control software for dynamically stable robots by using concurrently executing modules as a software paradigm. A module-based, expandable control architecture will be described along with existing and theoretical control schemes that can be implemented using it. An experimental robot infrastructure, both software and hardware will also be described, as well as the link between this higher-level software paradigm and lower level control.

This research is the first step toward creating a fully functional bipedal robot. Therefore, the emphasis of this paper is on the ease and power of its experimentation features and the robustness of its design. However, mathematical control models of dynamically stable robots are explored in the context of their implementation using this system.

Thesis Supervisor: Andrew W. Moore

Title: Assistant Professor, Robotics Institute and School of Computer Science

ACKNOWLEDGEMENTS

I would like to thank my partners, Johnathan Hurst, whose talent and many hours of painstaking work produced the leg and pelvis assemblies of the biped, and Brian Olson, who crammed a lot of electronics into very little space. Thanks also to Charlie Reverte, who created all of the 3-D renderings used in this report and helped with some machining and assembly.

Thanks are also due to the CMU RoboClub, especially Gabe Brisson, my friend and robotics partner in crime, for providing knowledge and manual labor, as well as the faculty and staff of the Robotics Institute for giving us funding, advice and support. Also thanks to Costa Nikou who helped me configure the Optotrak targets, though he won't admit it.

Finally, I would like to thank Robyn Thomlinson, for ruthlessly editing this document, saving me from numerous embarrassing grammatical errors, and showing me how to spell my name right.

This thesis and my biped work in general is dedicated to Bill McLaughlin, who introduced me to the robotics world back in sixth grade and who remains a constant supporter and good friend to this day.

INTRODUCTION

Why Bipedal Walking?

The desire for robotic bipedal walking machines stems from the fact that the world's infrastructure has been designed to be human-accessible. Stairways, ladders and devices proportional to average human size are all put in place to make humans' lives more comfortable. However, these same devices often make robotic navigation quite troublesome. Statically stable robots¹, such as wheeled and frame-walking machines, are often limited in the types of terrain they can successfully traverse and therefore are much less versatile than dynamically stable robots² such as bipeds and quadrupeds.

For a dramatic example, consider the Three Mile Island reactor incident. Decontamination of the site using robotics proved to be extremely challenging, due to the fact that the robots that were sent in were wheeled, and therefore could not reach many of the higher-level structures accessible through only narrow stairways. As a result, human lives were put in jeopardy so that those areas could be cleaned.

On a more everyday level, if general-purpose robotic devices were eventually to be put in situations requiring substantial human interaction and human environment navigation, a bipedal walker would be the most suitable configuration. Given human proportions, interacting with a biped would be much like interacting with another human, and the robot's use of devices like elevators and doorknobs would be within natural reach of its design.

An example of this would be the continuing research that is put towards office assistant robotics. Projects like Carnegie Mellon University's Xavier are wheeled, and designed to use elevators to move between floors. A robot that is humanoid shaped would have no problem accessing elevator controls, as well as taking stairs as a more efficient route. If temporary obstacles such as delivery boxes are in its way, it will be

¹ Robots that only move through defined positions of stability in their walking gait and are never in an unstable configuration.

² Robots that move through unstable positions in their walking gait and need to intelligently adjust and plan their movements to remain stable at any given time.

able to step over them, rather than navigating around them as wheeled robots do. Its usefulness would be enhanced due to the increased amount of terrain it can traverse.

Static vs. Dynamic Stability

Bipedal locomotion and dynamic and static stability have been studied for a long time, though only with the advent of fast computers could dynamic models be studied by building artificial mechanisms.

Before this, static walkers were created, employing control software that ensures the projection of the center of gravity on the ground is always inside the foot support area. On a biped, this is the minimum convex hull containing both foot surfaces in the case of both feet on the ground and the surface of the supporting foot in the case of one supporting leg. This is depicted in Figure 1. Static walking assumes that if at any time the robot's motion is stopped, it will stay indefinitely in a stable position. For this to work effectively, walking speed must be slow so that inertial forces are negligible. Therefore, this type of walking requires large feet, strong ankle joints, and can only achieve slow velocities.

In bipedal dynamic walking, the center of gravity can be outside the support region for a finite amount of time. There is no absolute criterion for determining the stability of a given dynamic walking gait, though walkers can be designed to recover from instabilities and use stability criteria as their guides. One such criterion, ZMP control will be discussed later in this paper.

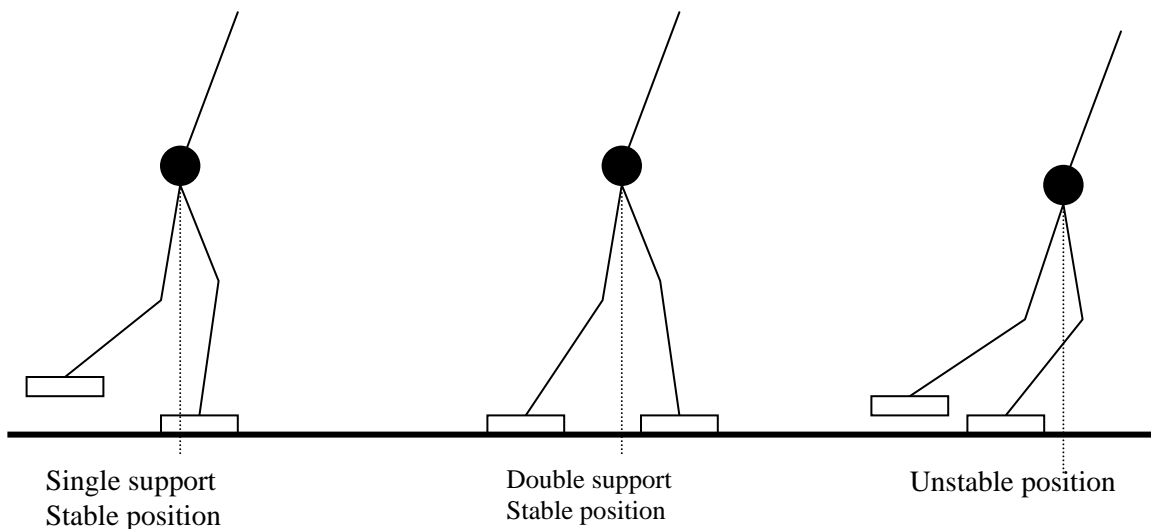


Figure 1: Static walking

Difficulty of Dynamic Control

As one may assume, a dynamically stable robot will be extremely hard to control reliably and effectively. As small children, we had to learn the properties of our own bodies and consequences of each move that we made in order to learn to balance upright. The same learning process applies to control software in dynamically gaited robotic systems. However, humans have the advantage of possessing some inherent knowledge of the configuration of their appendages, as well as an extensive nervous system to provide feedback, and an extremely powerful and adaptive brain to process that feedback. Robots can only adapt to the extent that we program them, and are provided with feedback from only the sensors we build into them. The fact that humans accomplish walking almost unconsciously after it is learned suggests that there is at least one good, computationally non-heavy solution to the problem.

Humans make use of many problem-solving techniques in almost everything they do. For example, driving a car requires the use of several types of problem-solving methods in order to be successful. Staying a constant distance from a car in front resembles a PID control method, while using the brakes effectively requires accessing acquired knowledge about the state of the brakes and their behavior, and handling turns depends on previous experience. The use of multiple techniques applies to walking as well, and this fact is addressed heavily by this research.

Bipedal Research Goals

The primary goal that will be discussed in this paper is the development of a software architecture that will allow programmers to explore various dynamic control schemes easily. It accomplishes this by providing a modular, easily reconfigurable development environment as well as a robust, fully customizable interface to a physical device's motors and sensors. This, along with a bipedal robotic testing platform, also discussed here, will make up a complete hardware and software biped experimental setup that will serve as a testing and development infrastructure for future dynamically stable robots.

This paper will present the details of this software architecture and the philosophy behind the design paradigm it provides to the programmer. It will also discuss the electrical and mechanical details of the robotic experimental setup and describe how the higher level control software interacts with lower level software and hardware.

EXPERIMENTAL ROBOTIC PLATFORM

The design philosophy behind the testing biped centers on reconfigurability, convenience of experimentation and low-cost. For the robot to be useful, it needs to be well built and protected to allow the programmer to make mistakes in control without worrying about damaging it. The experimental setup also needs to incorporate a programmable embedded computer so that low-level behavior can be easily modified regardless of the high-level software used to control it. If the robot is not well built or has weak motors or loose joints, no high-level control scheme will be able to keep it stable. This section will discuss the mechanics and electronics that make up the experimental biped, named *Iria*, depicted in Figure 4, and the design decisions that went into them.

Mechanics

Requirements

A biped that is physically unable to control or propel itself will never be able to walk. This suggests the importance of the physical construction of the robot. In order to design a robot that, given the right control scheme, can balance and walk successfully, care must be taken to meet and preferably surpass the necessary motor torque, joint tightness and range of motion needed to physically accomplish stable walking. With this in mind, the requirements designed into the physical attributes of the robot are:

- Enough degrees of freedom to balance without support (discussed later)
- Tight joints and accurate motor positioning
- Able to lift its own weight easily from any position
- Big enough to incorporate low-level electronics onboard
- Reconfigurable length of upper and lower legs
- Light enough to fall without damaging itself (for testing purposes)
- Provide protection for electronics and delicate sensors.

These requirements are satisfied using a set budget of \$1040 by employing readily available material and inexpensive motors. The key features of the mechanical design, as well as its advantages and shortcomings will now be discussed.

Mechanical Overview

Iria is built to maintain as light a weight as possible while maintaining stiffness and strength to safely survive rigorous testing. It incorporates the use and is built around the proportions of a bicycle helmet, which limits its size for ease of testing, while the helmet serves its typical duty of protecting vital parts upon accidental impact. Its design is kept simple, with only the legs and helmet making up the entire robot. It is designed to have off-board servo power provided through a jack mounted in the back of the helmet and onboard computer power, provided by a battery array, mounted inside the helmet. All mechanical and electrical components are mounted as symmetrically as possible in order to keep the center of mass along lines of symmetry in the robot.

Construction Materials

In order to keep the legs reconfigurable to facilitate the testing of new leg dimensions and replacing damaged components, the only machined parts are the ankles and pelvis. The upper and lower legs, as well as the feet are made primarily from easily replaceable Plexiglas, reinforced with aluminum bars. If these legs were to break, replacing them would consist of cutting new pieces out, drilling the appropriate holes, and assembling the leg. The aluminum used is milled from solid pieces rather than assembled from separate components. This provides strength and eliminates the weight that bolts and other joining methods would add.

Kinematics

Since *Iria* is an experimental platform, its kinematic model is defined only to the point of joint relationships, not to the extent of necessary motions required for walking. It is the point of future research to solidify these motions.

A rough kinematic model for one leg will now be presented. This model is rough in that joints are assumed to have all degrees of freedom centered around one pivot point, when in fact the mechanics require small offsets. In Figures 2 and 3, a coordinate system

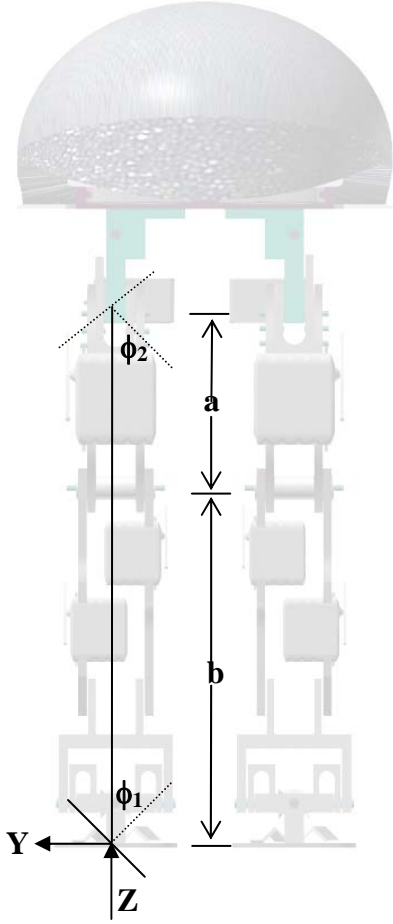


Figure 2: Biped rear view definitions

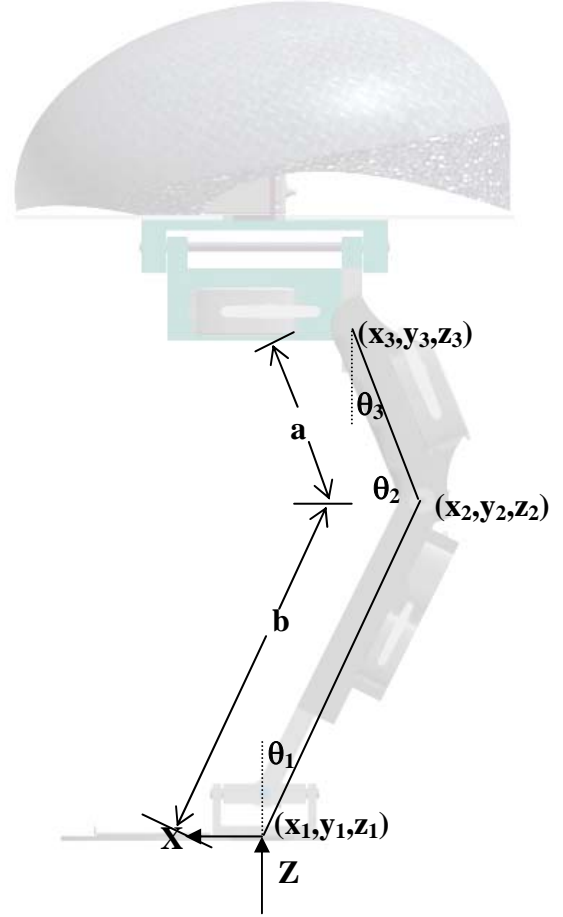


Figure 3: Biped left side view definitions

for a leg, reference variables for angles and lengths, and 3-dimensional coordinate variables of joint positions are defined. The positions of all leg points are in reference to the point of contact of the center of the foot and ankle joint with the ground, (x_1, y_1, z_1) , which is equal to $(0, 0, 0)$ since the origin of the leg coordinate system is positioned at this point. Following these conventions, the positions of the knee and hip joints have the mathematical form:

$$x_2 = b \sin(\theta_1)$$

$$y_2 = b \sin(\phi_1)$$

$$z_2 = b \sin(\theta_1) \sin(\phi_1)$$

$$x_3 = b \sin(\theta_1) + a \cos(\theta_2 + \theta_1 - 90)$$

$$y_3 = (b + a) \sin(\theta_1)$$

$$z_3 = b \sin(\theta_1) \sin(\phi_1) + a \sin(\theta_1 + \theta_2 - 90)$$

Though simplified, these mathematical models are good approximations to the actual positions of the joints, and can be used in control schemes such as “stupid walking,” described later, to help calculate pose information.

This method of determining pose, however, can become unreliable with the current electronics configuration, since joint position information is determined by the angles at which each servo is theoretically positioned. In reality, these angles may be different, due to heavy forces and the undetermined time needed to move to new angles. For this information to be totally accurate, a passive-sensing device on each joint, such as potentiometers must obtain joint angles.

The coordinate system used by the accelerometers and eventually by other pose-tracking mechanisms is centered at the top of the pelvis assembly, in the center of the

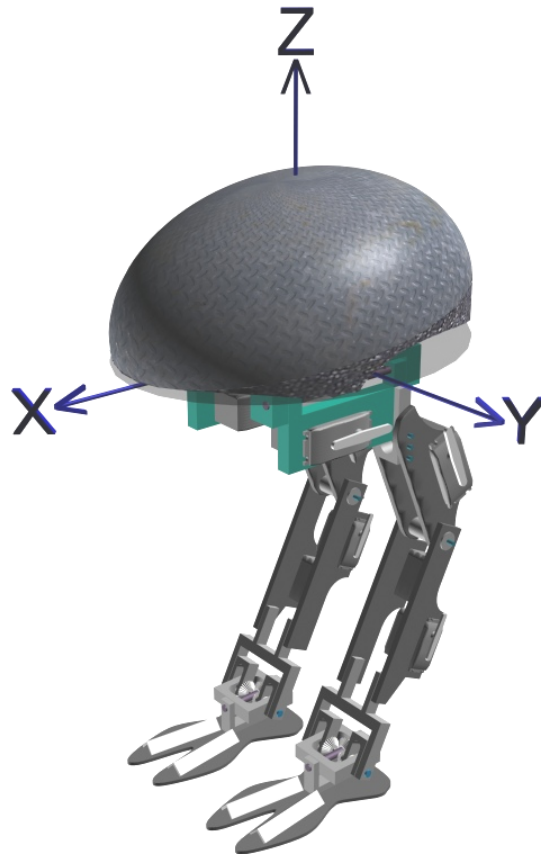


Figure 4: Biped perspective view and sensor coordinate system

body. This is approximately where the accelerometers are positioned and is a point that is affected by movement of any joint. This coordinate system is depicted in Figure 4.

Actuation

Iria's actuation is accomplished by the use of 1/4-scale model aircraft servos for the pelvis and knee joints and 1/8-scale model aircraft servos for the ankle joints. These servos are powered by an external 6V DC source, isolated from the computing power supply. Each servo has bearings on its drive shaft, and bearings are incorporated into each joint on the robot in order to provide wear resistance and smooth travel. In order to minimize the amount of play in all joints, each linkage is designed to provide the necessary range of motion while maintaining a tight connection. This is accomplished differently for each joint.

Pelvis actuation is exclusively pull-pull, using adjustable but tight opposing connections to their corresponding joint. Since pull-pull exerts relatively more stress on a joint than other drive connections, the bearings incorporated into the pelvis joints are also used to provide resistance against long-term damage.

Knee actuation is push-push, with oil damped springs providing adjustable compliance. This compliance can be totally eliminated for rigid response, and added to perform damped experiments.

The ankle joints use a gear differential, keeping both servos parallel to one plane along the length of the leg, but providing ankle motion in two planes.

All servos are positioned as high as possible above their corresponding joints in order to reduce the mass on the lower extremities, and hence decreasing the load on a given servo. This placement does not compromise the tightness of their linkages, since all motors are connected directly to their joints, with no intermediate components used to deliver power. Also, the servos use standard model airplane connectors on their shafts, which provide connection points at various radii from the center shaft. This provides a way to vary the mechanical advantage between servos and joints.

Electronics

Overview

Iria's onboard electronic infrastructure is designed to provide reconfigurable behavior and expandable abilities including the addition of motors and various sensors while still taking up a small enough area to be mounted inside its helmet. The basic design consists of an embedded computer, connected through A/D converters to three accelerometers and eight foot pressure sensors, and through a single serial line to servo controllers and to an external laptop. These components will be discussed in the following sections.

Embedded Computer

Iria uses a Motorola 68HC11A based Botboard II for its main onboard computing. Its purpose is to provide a link between high and low-level control via server software communicating via a 9600bps serial link to the laptop, as well as to monitor and provide data from all connected sensors.

The HC11 is powered by 5V, delivered by a pack of four AA batteries mounted in the helmet, independent of the servo power system. It runs at 2MHz and has 33024 bytes of RAM. This is more than sufficient, since the low-level code executing on the HC11 is written in assembly and commands servos with the assistance of off-board servo controllers.

One RS232 serial port handles communication with both the laptop as well as the servo controllers. This does not present a problem, since the servo control protocol differs significantly from the high-level to low-level software protocol, and each recipient can easily identify information relevant to itself.

A valid concern about separating the lower and upper (decision making) layers by a 9600bps serial connection is that at this speed the upper layer will not be able to send commands and receive data quickly enough to keep the robot stable in all situations. But, given that 9600bps sends approximately 1200 bytes per second, a realistic computation can be done to predict response time. A given motor command or sensor reading averages 4 bytes, and there are 10 motors to command and at most 11 sensors (3

accelerometers and 8 foot pressure sensors). If one assumes half of the serial time is spent sending commands and half is receiving data, each motor can be commanded approximately 15 times a second, and each sensor can be read approximately 13 times a second. Unless the robot is moving at a high rate of speed, this is most likely sufficient enough for good control. If more data needs to be sent, or experimentation determines that faster communication is needed, the HC11 and Botboard II can communicate at up to 14.4Kbps, but a new solution for servo control must be implemented since the servo controllers accept communication through serial at only 9600bps.

Servo Controllers

The two Mini Servo Controllers on *Iria* are small modules that use PIC Microcontrollers to generate pulse width modulated signals for the model aircraft servos. They have accuracy of 180 degree travel at .72 degree precision and can control up to 8 servos each, for a total of 16 servos. To use these controllers, each servo is assigned an identifier, determined by the port on which controller it is plugged into. The HC11 can then command any servo to move to a specific position with one 3-byte serial command, and let the servo controllers take over. This leaves the HC11 free to handle other low-level functions, such as monitoring sensors.

Sensors

The two types of sensors included in *Iria's* hardware configuration are accelerometers and electrostatic foot pressure sensors. These sensors are used due to their low cost and the value of the data they provide.

The accelerometers used are Analog Devices ADXL05 acceleration measurement system ICs. They are force balanced capacitive accelerometers with the ability to measure both vibrational and gravitational accelerations within less than +/- 5 milli-g. They are mounted on a breadboard along three orthogonal axes representing the X, Y, and Z-axes of the biped. The decision to use accelerometers rather than a 3-axis gyroscope package is based on two factors. The first is cost, with the accelerometers priced at about a factor of five lower than a sufficient gyroscope package. Second is the fact that accelerometer data can be used to measure the acceleration of the center of mass of the robot, as well as its angle by calculating the fractional gravitational pull along the

z-axis. A gyroscope package will only measure roll, pitch, and yaw, and can only give acceleration or velocity data if values are tracked continuously over a known time period. The roll, pitch, and yaw of the center of mass can be calculated on flat surfaces using inverse kinematics, discussed previously, if passive joint position sensors are used.

Though position and acceleration can be determined with accelerometers alone, forward acceleration due to walking will be difficult to filter out. Therefore, gyroscope packages will need to be integrated in future design revisions, but for initial experimentation, accelerometers will work satisfactorily.

Since acceleration of the center of mass will produce a predicted force on the bottom of its feet, it can be said that both accelerometers and foot pressure sensors are redundant. However, foot pressure sensors can provide information about small instabilities in the robot's stance that the accelerometers cannot pick up by accurately monitoring the force exerted by the robot on various points of its feet. In order to provide this data at low-cost, electrostatic foam was sandwiched by copper plates in order to create pressure sensors. As the distance between the copper decreases when pressure is applied, the resistance the foam exhibits decreases. This is similar to the operation principal behind a carbon microphone. If a voltage is applied across the foam, this resistance can be measured and relative pressure values can be obtained. The accuracy of this implementation will be determined with further experimentation.

THE *TRAVELER* SOFTWARE ARCHITECTURE

In order to truly create a bipedal experimental walking system, the software architecture used to design and test control algorithms must be as open-ended as possible, while still diverting the attention away from tedious software to hardware interfacing and other logistics of implementing a total control system. Details of this architecture, named *Traveler*, will be provided in this section, and the advantages and disadvantages of using it will be discussed.

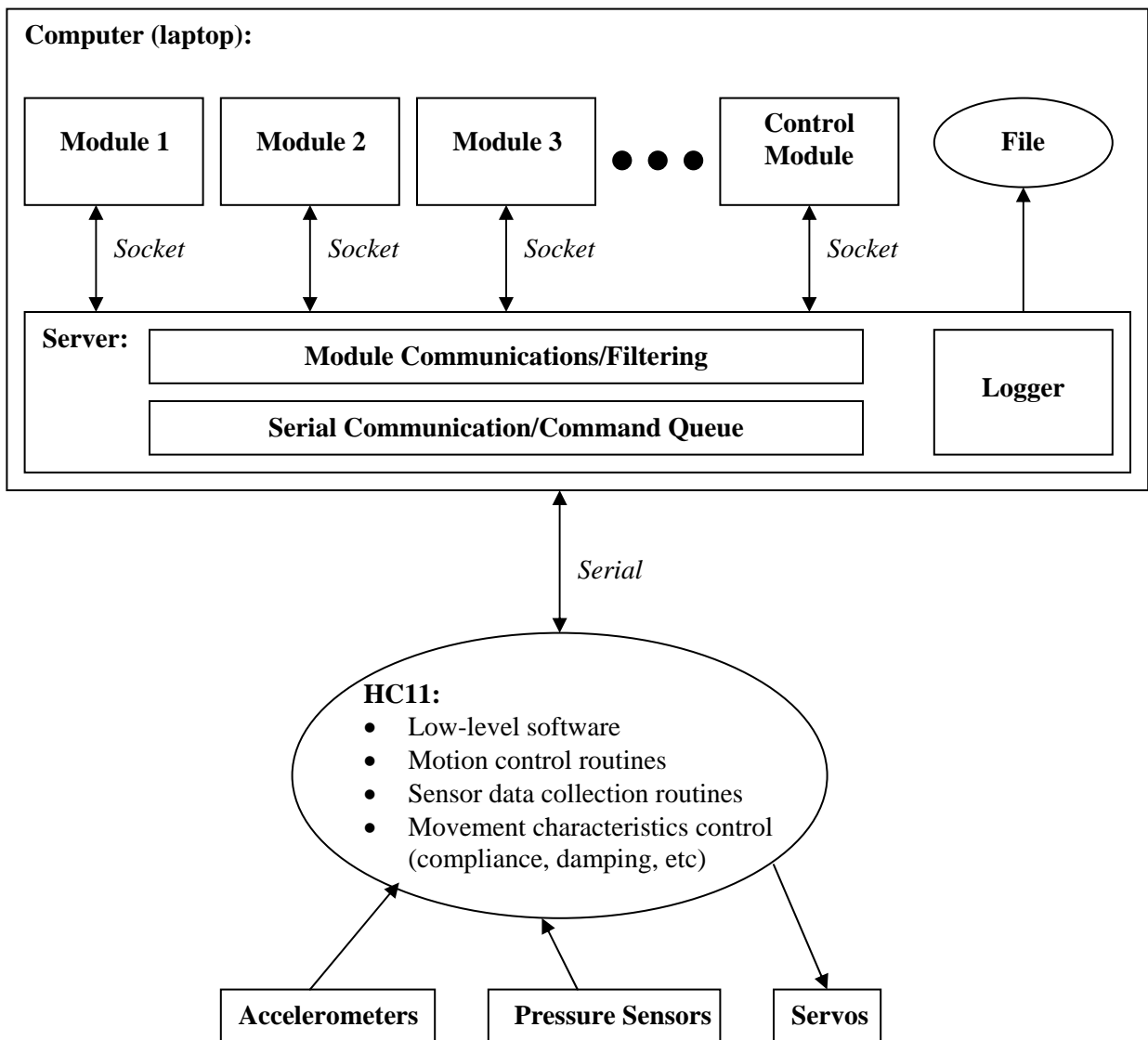


Figure 5: *Traveler* software infrastructure block

Layout

The complete software infrastructure consists of both low and high level software, with the low-level software running on the embedded HC11 on the robot, and the higher level software running on a computer connected via a 9600bps serial line to the embedded machine. A diagram of this can be seen in Figure 5. The system is clearly layered with strictly defined interfaces. This provides logical organization, as well as allows the programmer to modify specific layers for porting or working with a different robot, without having to worry about destroying the functionality of other layers in the process.

On the computer side, the software is layered internally by being divided into modules and a server to organize and filter all module commands and requests. The roles of each of these components will be discussed later. By dividing the software this way, programmers are free to implement whatever control schemes they would like, and the majority of the system can remain unchanged.

Low-level Software

The software running on the HC11 inside the robot is directly responsible for issuing commands to the servos and reading sensor data. It is configured to group specific motions together, as well as provide individual motor control and provide access to all of its functions via a command line-like serial interface. Specific commands the current low-level software provides are:

- Move individual motor to absolute position at a given velocity
- Move individual motor to position relative to current position at a given velocity
- Move motor group to absolute position at a given velocity (i.e. bend both knees to a specific position)
- Move motor group to position relative to current position at a given velocity (i.e. both hips, both knees, both ankles)
- Get all or individual accelerometer data
- Get all or individual foot sensor data

An example of these commands in use is the action of crouching. Assuming the robot is in a stable upright configuration, crouching involves bending the knees to lower its stance, and adjusting its ankle pitch to keep the center of mass of the robot over its feet. In order to do this, the higher level software must first issue a command to move the knees to the desired position at a reasonable velocity. This involves moving the knee motor group to a specific position, sending the desired velocity as a parameter, then, while the knee motion is occurring, reading the accelerometer data and adjusting the relative positions of the ankles to keep the balance correct.

By modifying the serial layer of the server and the low-level software, more commands and requests can easily be added to the system. Also, the implementation of the low-level software is independent of the high-level control scheme, so entirely new robots can be hooked up to the computer, as long as they follow the same communications protocol.

In the future, the low-level software will be expanded to include velocity curve specification and built-in compliance. This will provide a more natural walking motion and help with traversing uneven terrain.

High-level Software

As can be seen in Figure 5, the high-level software infrastructure is divided into two distinct layers: the server and the modules, with the server layer subdivided even further. Each module and the server are separate processes that communicate with each other through UNIX sockets. The design philosophy behind this is concerned primarily with four important factors.

First, the software must be expandable to as much a degree as possible. If the entire high-level control program were written as one monolith entity, a fixed language (C/C++) would limit expansion. Also, the act of recompiling the entire program each time a small piece of the control algorithm was modified would prevent it from being an efficient and convenient testing environment. Since the control algorithm is implemented primarily distributed among smaller, standalone programs, and communicates via a common, infrequently-changing server, development time is decreased and the limitations of a fixed programming language are removed. At the time of this paper,

module libraries are implemented in C, but due to the cross-language attributes of UNIX socket communications, modules can be written in any language and incorporated relatively easily into the architecture.

The next factor has less to do with experimentation but is essential in implementing effective control schemes. Walking and balancing dynamically involves handling of several different types of information and dealing with them separately, though a particular decision about a given piece of information can be based on the current status of other parts of the system. Though these decision-makers require information on each other, they must do their own jobs individually, and, most importantly, concurrently. To solidify this point, consider a simple, general case of controlling forward as well as lateral balance. If a programmer were able to come up with two simple, single-threaded algorithms for handling balance in each direction, these two algorithms must run at the same time in order for the robot to stay completely balanced. In reality, control algorithms will require many more concurrent processes in order to function effectively, so the architecture must provide the ability to do so easily. The modular design of the high-level software handles this issue. Since it is implemented under a UNIX system in as many modules as the programmer needs, the operating system will take care of concurrency, which leaves the programmer free to concentrate on control issues rather than figuring out how to make concurrent decisions in one process.

The third factor in designing the high-level architecture is the ability to selectively log activity to a file or other destination. Since the modules that make up the system need to communicate with each other as well as the robot, the server provides a common place to monitor all system activity and filter out the events that should be logged. This functionality is discussed later.

Finally, the server should provide a layer of abstraction between high-level communication and interaction with the robot. The modules should have no knowledge of the communications protocol, but still have access to the state of the robot and have the ability to issue commands.

There are also other advantages of a separate process paradigm that are handled almost invisibly, rather than explicitly implementing them internally to the software. One such advantage is the ability to set resource priorities on different parts of the system,

forcing infrequently used modules to stay out of the way of more important ones. Also, the ability to eliminate parts of the system for isolation in testing without breaking subtle code dependencies, and the ability to incorporate a user interface that runs independently from the rest of the system are provided almost for free.

The following sections will describe each part of the high-level architecture, and discuss their relationship with the rest of the system.

Modules and the Module Design Paradigm

Modules are simply standard UNIX programs with the ability to communicate via sockets to the server. In order for these modules to interact with the server, a specific protocol must be used to communicate and certain rules must be adhered to. If a module is written in C or C++, this communication is transparent and taken care of by a module library. If not, this library must be emulated, or a new one must be written.

The communication between module and server consists of message passing. Each message has the form:

#<sender-ID><command><parameters>

where *sender-ID* is a unique two byte identifier for the module, *command* is one byte representing the action wished to be performed with the server, and *parameters* is a string representing specific parameters used with the given command. The parameter string can include spaces.

In order for a module to interact successfully with the server, it must be “registered.” This is accomplished by providing an ID to the server in its module header file. In future versions of the software, this will be done via a text file that is read in upon server execution, eliminating the need for recompilation. Before performing its normal duties, each module must first “check-in” with the server. That is, each module registered with the server must notify the server of its presence by sending a “check-in” command to it. After doing so, it must wait until the server replies that it has received notification from all configured modules. At this point, the module is free to interact with the server. If for some reason a module is forced to exit or wishes to disable the system, it can perform a “check-out,” which forces the server and other modules back into a wait state. The specific interactions allowed between modules and the server are described later in the *module control layer* section.

For modules to cooperate and operate effectively, they should all be designed around the communications paradigm used by the server. Registering themselves is only part of this. While being programmed to perform their intended duties, each module must be designed with its priority and control level in mind. That is, it should be prepared to restrict hardware access to other modules and/or expect to be restricted itself. Though the server is the “meeting area” where final decisions are made on whether actions are executed or not, ignoring the server’s access limitations will make the system chaotic, and most likely, unsuccessful. A well-written module will recognize when it needs to restrict access, or when access is restricted to it, and will adjust its internal behavior according to a global understanding of why the access change occurred.

For an example of this, consider a very general case of two modules. One module controls the normal actions of the robot, while the other only handles emergency situations, such as when an external force throws the robot heavily off balance. While the robot is operating under normal conditions, the emergency module is prevented from interacting with the robot. During this time, the emergency module should not keep trying to read the robot’s sensor data, or control its motors. It will only be ignored, but processor cycles will be wasted in handling these ignored requests. If designed poorly, the emergency module could also force itself into a state where, when it is allowed access, will operate incorrectly.

Rather than blindly going about its duties, the emergency module should wait until the server sends a message to it saying that access has been granted. This means that the robot is in an emergency situation, and it the emergency module should begin taking action. At this same time, the normal control module should recognize that it just relinquished control to the emergency module, and wait quietly until the emergency module stabilizes the robot and turns control back over to it.

Server

The server layer of *Traveler* is named not due to its relationship with the robot, but due to its function as the center of communication and organization of the modules that make up the high-level control software. As discussed previously, the server is able to restrict access to hardware, as well as restrict and filter message passing between modules. It consists of three layers: serial communications, module communications and

filtering, and logging. Though the server is implemented as a single process, these separate layers can be configured independently, and their functionality can be modified with little if any modification to other layers. All command, request, messaging and logging actions occur serially in the server, since a given action can directly affect the way future actions are treated. Therefore concurrent operation in the server would defeat its organizational purpose. The next few sections will discuss these three layers that make up the server, what functionality they provide, and will present examples of their usage.

Serial Communications Layer

The server's serial communications layer handles all communication with the robot, as well as manages a queue of pending robot control instructions. This layer provides specific interface functions to access its available features, restricting the high-level software from sending arbitrary commands over the serial line.

The serial communication itself is full duplex and event driven. When information comes in over the serial line, the server process is interrupted briefly to deal with it. This prevents the need of blocking until a response to a query or command is received from the robot.

To modify or add functionality to the serial layer, new interface functions must be added to support it. Also, a tight relationship must exist between the serial layer and the low-level software in the types of functions they allow. Specifically, the server cannot offer functionality that is not implemented in the low-level software.

Module Communications Layer

As discussed earlier, the module design paradigm used by *Traveler* relies on a powerful module communication mechanism provided by the server. The module communications layer is where this mechanism is implemented. The functionality it provides consists of:

- Module check-in/check-out
- Module hardware restriction
- Module messaging
- Module message restriction and isolation

- Logging requests
- Robot hardware command or query requests

The module communications layer can be looked at as having the “final word” over what modules are able to do and what inter-module or hardware commands are executed. It is in this layer where certain modules can be restricted from sending messages to other modules, issuing commands to the robot hardware and from logging events to the global log file. Having this central decision-maker, prevents the system from being chaotic, and provides a common point to track all system activity.

Logging Layer

Testing a dynamic robot requires the analysis of a large amount of selected data in order to isolate and observe specific behaviors of the control scheme and to determine the state of the system when instabilities occur. In order to facilitate the recording of this data, a layer dedicated to logging is incorporated into the *Traveler* server.

This layer can be accessed two ways: configuration prior to execution and logging requests during execution. The former of which is done currently through the modification of a header file. In later versions of the software, this will be done using a text file, read in upon startup. The latter is done at runtime, via a command provided through the module communications layer.

By configuring the logging layer through its header file, any event that passes through the server can be logged automatically. Events such as module restrictions, module to module messages, and robot hardware commands can all be tracked and stored in a file without specific requests to do so during execution. In many cases, however, modules may want to log events occurring only within themselves to the common log device. This can be accomplished by sending a message to the server requesting a log along with the text to log.

In the current version of the software, all logs go directly to a file, specified at compile time. In future versions this will be made more robust, providing the ability to output to multiple files, as well as other devices like a pipe to a user interface.

MATHEMATICAL MODELS FOR BIPEDAL WALKING

In order to understand what types of control schemes can be implemented under this module-based architecture, some mathematical models for bipedal walking will be described. Later, examples of how these models have been implemented in software will be given, and the advantages the *Traveler* system could provide to these implementations will be discussed.

Zero Momentum Point Model

The zero momentum point (ZMP) is the point where the robot's total momentum at the ground is zero, and can be used as long as the robot keeps at least one foot flat on the ground at all times. If the ZMP is inside the support region, the robot is considered dynamically stable since it is the only case where the foot (ankle actuators) can control the robot's posture. If the robot has no ankle joints, this criterion for stability cannot be applied.

By ensuring that the ZMP is in the support region, this means that the robot is always rotating around a point in the support region. If this is not the case and the robot rotates around a point outside the support region, the supporting foot will tend to be removed from the ground or pressed into it, leading to instability. This is depicted in Figure 6. If all motion were stopped in this case, the robot would rotate around the ZMP

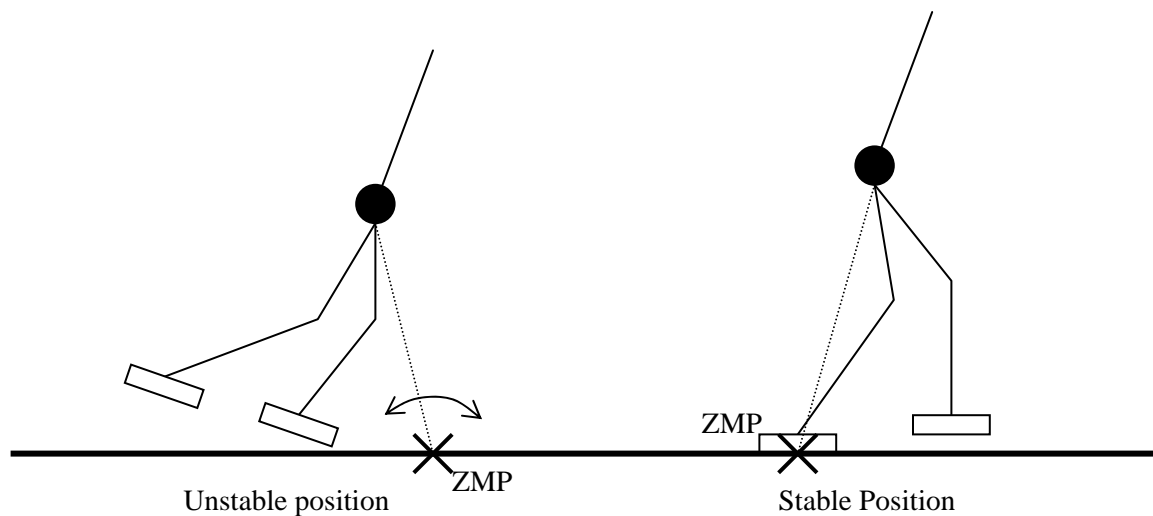


Figure 6: ZMP model

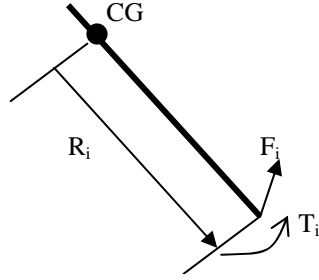


Figure 7: Forces on a link

and fall over.

Finding the ZMP is accomplished by finding the point (X,Y,Z) where the total torque is zero. Since we are interested in this point on the ground plane, we'll set $Z=0$. Assume a robot with n links; each link can experience a total force F_i applied at a point determined by the vector R_i relative to the center of gravity of the link. T_i determines the total motor torque applied to the link. R_z is the ZMP vector and T is the robot's total torque. This setup is depicted in figure 7. Given that the force, torque, and position vectors have the following coordinates:

$$F_i: (F_{xi}, F_{yi}, F_{zi})$$

$$T_i: (T_{xi}, T_{yi}, T_{zi})$$

$$R_i: (x_i, y_i, z_i)$$

$$R_z: (X, Y, Z)$$

Then the total torque is computed as:

$$T = \sum_{i=1}^n (R_i + R_z) \times F_i + \sum_{i=1}^n T_i = 0 \quad (1)$$

This can be expanded to:

$$\sum_{i=1}^n (y_i + Y)F_{zi} - \sum_{i=1}^n (z_i + Z)F_{yi} + \sum_{i=1}^n T_{xi} = 0 \quad (2)$$

$$\sum_{i=1}^n (x_i + X)F_{yi} - \sum_{i=1}^n (y_i + Y)F_{xi} + \sum_{i=1}^n T_{zi} = 0 \quad (3)$$

Making $Z=0$ and solving for X and Y , the ZMP coordinates become:

$$X = \frac{\sum_{i=1}^n (z_i F_{xi} - x_i F_{zi}) + \sum_{i=1}^n T_{yi}}{\sum_{i=1}^n F_{zi}} \quad (4)$$

$$Y = \frac{\sum_{i=1}^n (z_i F_{yi} - y_i F_{zi}) + \sum_{i=1}^n T_{xi}}{\sum_{i=1}^n F_{zi}} \quad (5)$$

Inverted Pendulum Model

A simple way to look at bipedal motion during slow walking, assuming only one foot on the ground, is a model employing the physics of an inverted pendulum. In this model, depicted in Figure 8, the robot is represented by a point mass equal to the total mass of the robot, positioned at the center of gravity and connected by a massless beam to the foot on the ground. In this case, the ZMP is the point of contact with the ground. The dynamic equation for this model is:

$$x(\ddot{y} + g) - y\ddot{x} = 0 \quad (6)$$

where x and y are the coordinates of the point mass, g is gravity and the second derivatives are with respect to time, providing acceleration. Note that in this model, only two dimensions will be discussed, so the robot will be viewed from the sagittal plane with a standard x - y coordinate axis. This equation is highly nonlinear and cannot be used effectively for control, so a simplified equation can be constructed assuming constant height.

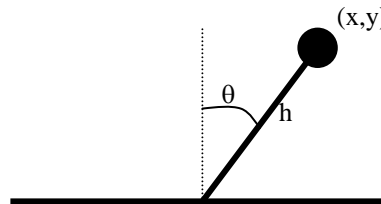


Figure 8: Inverted pendulum model

$$\ddot{x} = \left(\frac{g}{h} \right) x \quad (7)$$

where h is the constant body height. This assumption is commonly used and valid since most humans try to keep their center of mass at a constant height to save energy. This constant height is usually changed by the knee or whatever mechanism on the robot replaces it. As said before, it is insufficient for fast motion where for shock absorbency and power delivery, the knee is bent and extended, changing the height stance. But at slow walking speeds, this model is a good approximation.

Lateral Balancing

Most bipedal research papers have focused on models to control the two-dimensional, forward motion of the robot. MIT, for example, restricts many of their robots to two dimensions by having them run in a circle while attached to a radial support beam [Raibert, 86], [Pratt, 95]. This is most likely because lateral balancing can be accomplished using the same or simplified versions of the models discussed previously.

The simplest case is the inverted pendulum model, depicted in Figure 8. By using Equation 7 while the feet are in a transition period, the behavior of the robot while supported by one leg can be modeled. If the robot undergoes a significant change in height during this transition, Equations 1 through 5 can be used to calculate the ZMP

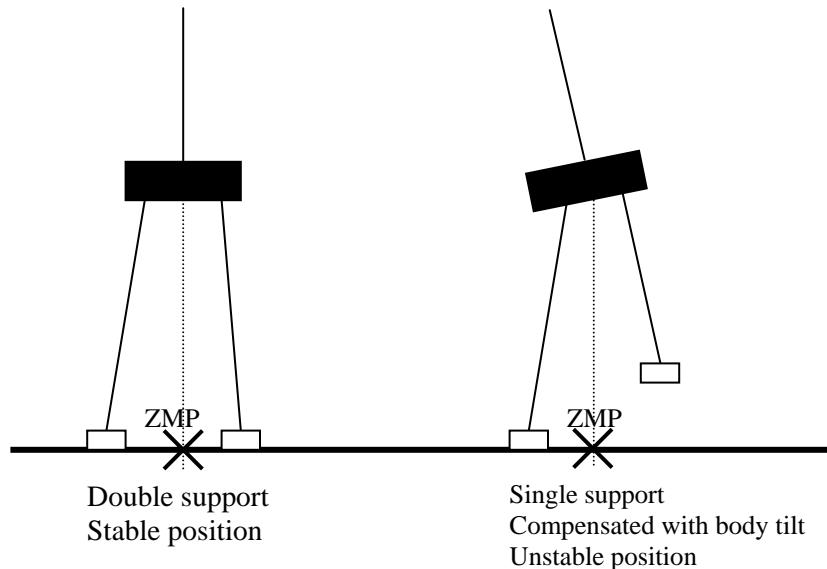


Figure 9: Lateral balancing

location. By having this information, lateral hip actuators, as well as the lateral angle of the free leg can be used to control the ZMP and keep the robot stable until the free foot is on the ground again.

When both feet are on the ground, the ZMP is located between them and therefore the robot is in a stable position. One and two foot support cases are depicted in Figure 9.

Walking Gait Study

In order to develop efficient ways to move and position legs during a walking gait, a study was designed as part of this research effort to collect data on human and reverse knee walking by using optical tracking targets affixed to parts of a human leg. This setup, depicted in Figure 10, uses an *Optotrak* optical tracking camera, developed by Northern Digital Inc. to continuously record the positions of several rigid-body configurations of LED's with sub-millimeter accuracy. Plots of relative positions over time can then be obtained as a subject walks past the camera and used as guides on how to move the legs to achieve certain gait characteristics.



Figure 10: Walking gait study setup

CONTROL SCHEME EXAMPLES

“Stupid Walking”

“Stupid Walking,” is a control method used by the *Spring Turkey* two-dimensional biped at MIT [Pratt, 95]. This biped is supported laterally by a fixed radial arm that guides it around a circular track, providing the ability to isolate and control only forward motion control (along its x-axis). As an attempt at making this robot walk unintelligently and unreliably forward, a simple Finite State Machine is used, the details of which are given in Table 1, along with a simple set of rules:

- Maintain a constant height and pitch.
- Switch state from double support to left or right single support if the body’s x position becomes too close to that foot.
- Switch state from single support to double support if the body’s x position becomes too far away from the support foot.
- Servo the free leg so that its foot is placed the normal stride length away from the support foot when switching to double support.
- During double support, push in the direction of desired travel with a constant force.

State	Trigger Event
Double Support	Delay after left or right support 2
Left Support	Body nearly over left foot
Left Support 2	Body away from left foot
Right Support	Body nearly over right foot
Right Support 2	Body away from right foot

Table 1: “Stupid Walking” FSM details [Pratt, 95]

Using this control scheme, *Spring Turkey* has been able to walk approximately 3 meters in 5 seconds. This is far from ideal, since the model used is so simplistic. This “stupid walking” scheme can be implemented in one module, in that it only makes decisions serially using one decision making component.

In more successful applications of FSMs to bipedal walking, several state machines are used, each focusing on a specific task. Since these state machines are trying to accomplish the same goal, interactions between them must occur, but they also must be able to execute independently and have individual access to motor control and sensor data. This is tedious and often inefficient when implemented in a single process.

One solution to this problem is to create a multithreaded process, giving each FSM its own thread as well as allocating a thread to serial communications code and other layers of logic. The main disadvantage to this is the amount of programming overhead it creates, along with the disadvantage of having to modify communications and other code that should otherwise be left unchanged.

Creating each FSM as a separate process produces the same effect as multithreading, but at the same time providing operating system scheduling priority assignment and keeping static code unchanged. Communication between separate FSMs is handled by module message passing through UNIX sockets. This communication method provides a great deal of priority control due to the server's ability to filter messages and restrict access to hardware. By configuring the server with the correct filtering and access restriction logic, and planning FSM interaction carefully, a given FSM can be assigned a high priority or control position over other FSM's in the system.

Neural Networks

Neural network use in robotic bipedal walking is generally based on the assumption that there exists a nominal behavior governed by a minimal number of inputs. That is, there are "correct" sets of values for sensors and motor positions that guarantee stable walking. In this fashion, this behavior can be learned and generally repeated so that the robot can "learn" how to walk correctly.

A powerful learning architecture should take advantage of any available knowledge, however, there are some cases where a simple rule or linear controller can achieve the desired behavior. With this in mind, many researchers have used different controllers in parallel, and a switching mechanism that activates the appropriate controller [Jacobs, 91], [Narendra, 94], [Sklansky, 66], [Widrow, 60].

In a study by researchers at the University of New Hampshire [Benbrahim, 1996], instead of switching mechanisms, a “melting pot” is used. This is a central controller that uses the experience of other controllers in order to learn an average control policy. The central controller controls the robot in nominal situations, and peripheral controllers intervene only when they consider the actions of the central controller contradictory to their individual control policies. These peripheral controllers are able to update and correct the central controller’s policy and issue an evaluation of the general behavior of the system.

In order to achieve this, many controller tasks are required to run in parallel, and must be able to communicate to the central controller and sometimes each other in order to pass training information. They must also be able to take control over the system at certain times. The *Traveler* architecture lends nicely to this with its ability to run as many processes as needed at the same time, its ability to restrict hardware access, and the availability of programmer-defined message passing routines.

Also, the ability to isolate, and log sensors and messages will help in debugging any modular neural network system, as well as allow the programmer to block off parts of the system in order to train them specifically.

CONCLUSIONS

Accomplished Goals

As of the writing of this paper, *Iria* is fully operational in that it can move all of its joints and sense acceleration and foot pressure. The Traveler software is operational, but needs to have many of its configuration mechanisms made easier to access, preferably without having to recompile after a change is made. For initial demos of the system, a stripped down version of the server with a user interface is used to interactively move joints and show sensor readings.

Due to budget constraints, only acceleration data could be incorporated into the initial system. This is enough to get the electronic, mechanical and software bugs worked out and gain experience working with the system. In future revisions of *Iria*, as well as in new bipeds, gyroscopes will be used to provide angular positional data, and external visual tracking systems such as the *Optotrak*, discussed earlier, will be used to accurately track pose while developing walking gaits.

Future Work

This experimental software will continue to be developed until control scheme research begins. During this research, the experimental software will most likely be expanded and updated to provide any new functionality that will be needed.

Iria will be used as a testing platform while the design of a new biped commences. This will allow for the concurrent development of a control scheme and new hardware. Due to the strict protocol used for serial communication, the low-level software and electronics can be implemented in any way necessary for the new biped, and still be linked seamlessly with the high-level software.

It is the hope of everyone involved in this research project that this research will help bipedal walking become a viable tool in the challenge of walking mobile robots.

References

Benbrahim, H., *Biped Dynamic Walking Using Reinforcement Learning*, Doctor's Thesis, University of New Hampshire, December, 1996.

Dunn, Eric R., Howe, Robert D., "Towards Smooth Bipedal Walking," *Proceedings of the IEEE International Conference on Robotics and Automation*, Volume 3, pp. 2489-2494, San Diego, May 1994.

Jacobs, R.A., Jordan, M.I., Barto, A.G., "Task Decomposition Through Competition in a Modular Connectionist Architecture: The What and Where Vision Tasks," *Cognitive Science*, Volume 15, pp. 219-250, 1991.

Mason, Matthew T., *Mechanics of Manipulation*, Course Text Draft, Carnegie Mellon University, April 1998.

Narendra, K.S., Balakrishnan, J., "Intelligent Control Using Switching and Tuning," *Proceedings of the 8th Yale Workshop on Adaptive and Learning Systems*, 1994.

Pratt, Jerry E., *Virtual Model Control of a Biped Walking Robot*, Doctor's Thesis, Massachusetts Institute of Technology, August, 1995.

Raibert, M. H., *Legged Robots That Balance*, MIT Press, Cambridge, 1986.

Raibert, M. H., Brown, H. B., Jr., Chepponis, M., Hastings, E., Shreve, S. T., Wimberly, F. C. 1981. *Dynamically Stable Legged Locomotion, First Annual Report*. CMU--RI--81--9, Robotics Institute, Carnegie Mellon University.

Sklansky, J., "Learning Systems for Automatic Control," *IEEE Transactions on Automatic Control*, Volume AC-11, pp. 6-19, 1966.

Troy, James J., *Interactive Simulation and Control of Planar Biped Walking Devices*, Technical Paper, Iowa State University.

Widrow, B., Hoff, M.E., "Adaptive Switching Circuits," *1960 IRE WESCON Convention Record*, New York: IRE, pp. 96-104.