Design and Control of the BlueFoot Platform : A Multi-terrain Quadruped Robot

A Thesis
Submitted in Partial Fulfillment
of the Requirements for the
Degree of
Master of Science (Electrical Engineering)
at the Polytechnic University

by

Brian Cairl
May 2015

Master's Examination Committee:
Copy Number:

Approved by:

_____
Advisor

_____
Date

_____
Department Head

_____
Date

VITA

Mar 23, 1992	Born
Sep, 2010	Entered the NYU Polytechnic School of Engineering as a Mechanical Engineering major and an Honors student.
Jun, 2012	Completed a minor in Mechanical Engineering.
Mar, 2014	Admitted to the NYU Polytechnic School of Engineering as PhD Fellow under the advisement of Professor Farshad Khorrami.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

Introduction

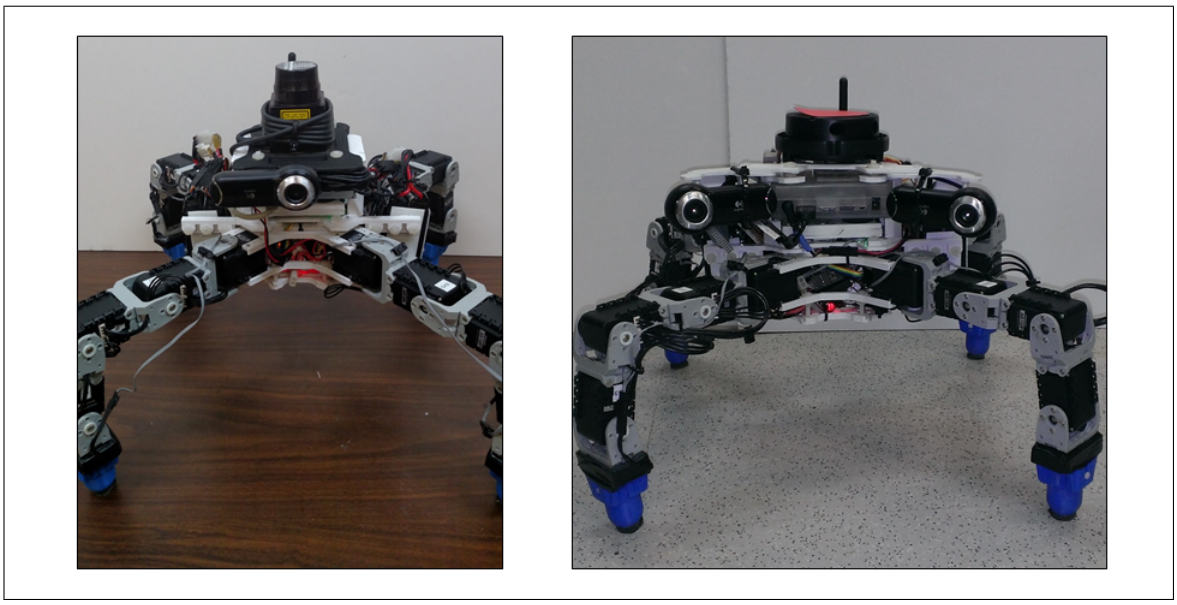

Figure 1: The BlueFoot Quadruped Robot: single-camera configuration *(left)*; stereo-camera configuration *(right)*

The design of legged robots and associated methods of locomotion control has been an area of interest spanning the past several decades, as shown by [**?**, 1–3]. Quadruped robotic systems have gained popularity in studies pertaining to variable terrain navigation and full-body stability adaptation. Well known examples of this from the past decade are the Tekken [4], Kolt [5], BigDog [6], and HyQ [7] quadrupeds. Many of these systems have been implemented on a larger scale so that they can carry substantial payloads while maintaining adequate system bandwidth for fast gaits and robustness to rough terrains. Few, however, have been implemented on the scale of a hobby-robot platform while still maintaining an aptitude for rough terrain navigation and comparable sensory prowess.

The BlueFoot quadruped is a self-contained, fully-actuated platform with the dexterity to perform stabilization and repositioning maneuvers on variable terrains along the same lines as the LittleDog platform [8]. BlueFoot includes a sizable array of onboard sensors for feedback and control. Using the computational capacities at hand, BlueFoot has the ability to utilize similar control mechanisms to those implemented on larger quadruped systems. BlueFoot is gaited via a central pattern generator (CPG) based gaiting algorithm augmented with a foothold controller along the same lines as [9] and [10]. Additionally, active platform stabilization is performed via a zero-moment point (ZMP) based body-posture controller which actively stabilizes the system during arbitrary gaiting sequences. The controllers presented in this paper make use of virtual-forces to drive system reference commands. An outer-loop controller supplies commands and corrections used in system navigation control.

Not complete

CHAPTER II

Hardware and Design

2.1 Overview and Design Goals

The BlueFoot quadruped robot is designed as a small-scale, general-purpose legged mobile platform with enough physical dexterity and on-board computational/sensory power to perform complex tasks in variable environments. BlueFoot's hardware configuration is aimed at performing of tasks as a standalone unit, i.e. without power tethering or off-board processing. BlueFoot's sensory, computational and power-source outfit make it fit to complete tasks in both settings fully and semi-autonomous modes.

The implementation of a legged robot which meets these general specification is inherently bottlenecked by several well-known shortcomings which plague legged robot design. These drawbacks can be summarized as follows: relatively low payload capacity, as leg joints are often subjected to substantial dynamic torque loading during gaiting; and higher power consumption due to a, typically, larger number of total actuators. Thus, a general-purpose, multi-legged system like the BlueFoot platform must ultimately achieve a balance between payload-carrying capacity (i.e. maximum joint-servo output torque); actual on-board payload; and on-board energy supply. It is desirable that the sensory and computational power; as well as overall mobility of a legged system are simultaneously maximized along with the aforementioned characteristics.

The design goals which have guided the implementation of the BlueFoot quadruped have been tailored to a yield an overall system design which is both feature rich, computationally powerful, and exploits the natural dexterity and terrain handling of legged robotic systems. Namely, the core design requirements which have guided BlueFoot's development can be summarized as follows:

- The use of legs with joint redundancy for improved dexterity
- The use of smart servos for extended joint feedback and control
- A distributed on-board and computing architecture for hierarchal task handling

- A vision sensor array including a camera and laser-ranging sensors
- 30+ minutes of total battery life

This chapter will outline how an implementation meeting these design goals is achieved, starting with the structural layout of the system. Next, major system payloads and the associated interfacing of major devices will be described. This section which will include details about BlueFoot's actuators, computational modules, and sensory mechanisms. Lastly, the system power routine, energy requirements, and runtime will be detailed.

2.2 Robot Structure

BlueFoot's body is designed in a modular fashion and is comprised of mostly custom designed, 3D printed parts. The use of 3D printing as a fabrication method allowed for rapid design iterations the early stages of system prototyping, and has kept the weight of the robot's overall structure relatively low. Parts were mainly printed from both PLA and SLA plastics. BlueFoot's overall weight (when fully outfitted) is $1.85 - 1.98$ kg, depending on configuration.

The modularity of BlueFoot's overall structure arises from the inherent design requirements associated with 3D printing and general design practices aimed at keeping the system reconfigurable for the incorporation of updated sensory and computational hardware. Moreover, parts are designed to fit future replacements while conforming to the constraints imposed by the 3D printing fabrication method, i.e. particular part size and orientation requirements. Such constraints had to be met by each designed part to ensure print feasibility.

The BlueFoot platform has undergone several minor redesign phases since its inception. These redesigns were necessary to bring the BlueFoot platform to its final structural and hardware state and were performed to accommodate changes in sensory/computational hardware. The sections that follow will mainly focus BlueFoot's final hardware configurations.

2.2.1 Main Body (Trunk) Design

BlueFoot's trunk consists of the three main sections: a lower module which interfaces the legs with the main body; a center chassis, designed to hold computational and battery payloads; and a top platform, which interfaces the system's visions sensors to

the trunk. These sections will be referred to as the *Root* module, the *Main* module, and the *Head*module, respectively. The full trunk (not including sensor dimensions) fits within a 21.6 by 21.6 by 15.3 cm bounding box.

The Root Module

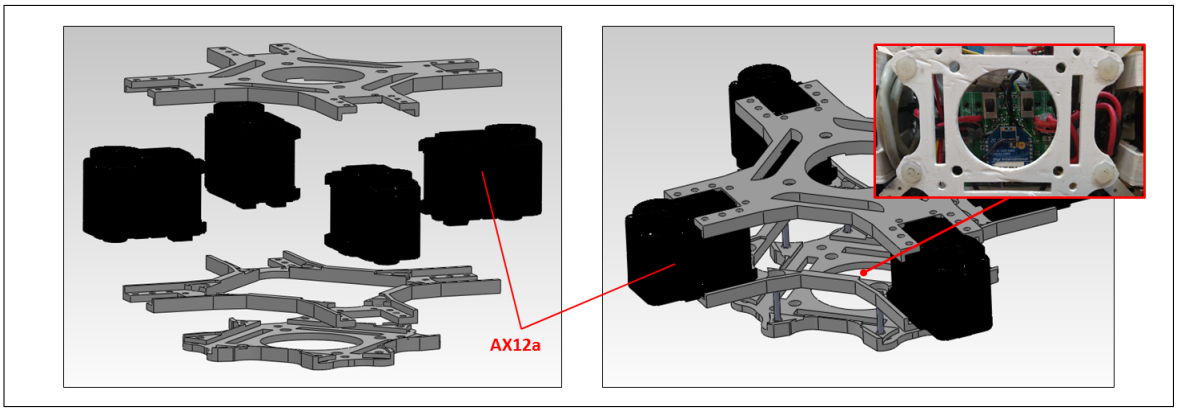

Figure 2: Root section of trunk. Call-out in top-right shows main-switch access through the bottom of the root module.

The Root module, consists of three plates, as shown in Figure 2. Each plate is designed with a central opening to allow for wired connections to pass to other trunk modules. Two such plates directly interface with four servos, which are mounted to four symmetric arms which extend from the center of each plate. These servos are the first joint (hip-joint) of each leg. Each servo mounts to the top a bottom plates via mounting holes located at the top and underside of each servo chassis. The assembly is mated with small steel bolts. A third, smaller plate is attached to the bottom of the module to provide more space for power components and associated wiring. This plate is attached to the bottom of the module via plastic standoffs. An opening in the middle of this plate provides access to the system's main power switches, as well as a removable XBEE wireless radio unit.

The Main Module

The Main module of BlueFoot's trunk includes compartments for an in-house designed AutoPilot unit and a main computer unit, an ODROID-XU. The Main module is designed such that the AutoPilot and ODROID-XU computer slide in and out of the body. The computer payloads are locked into position when the Head module is added

to the assembly. The Main body section is designed to fit both computers when stacked upon one another, as depicted in Figure 3. The computer stack is positioned directly in the center of the module when inserted.

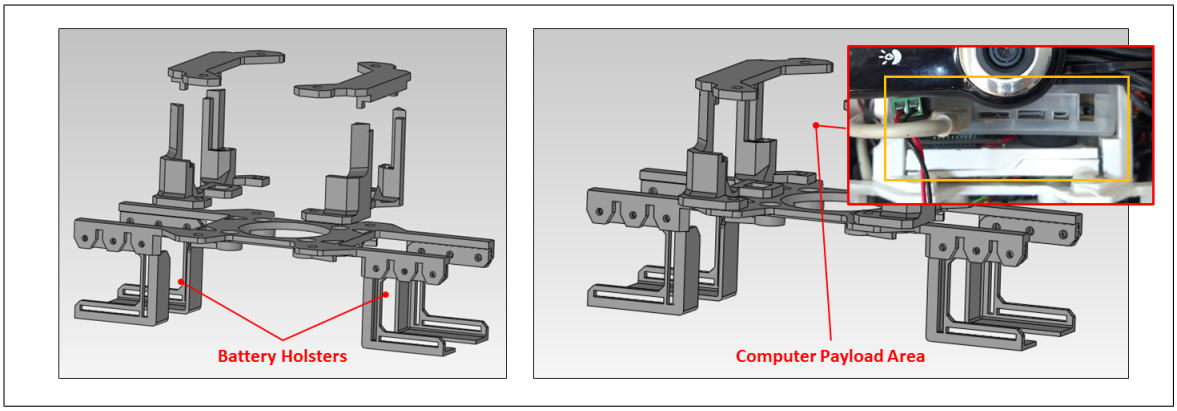

Figure 3: Main section of trunk. Call-out in the top-right shows how the ODROID-XU and AutoPilot computers fit within the module.

The Main module also includes two battery holsters, which hang over its left and right sides. The holsters align the battery packs with the center of Root module. This battery placement serves to lower the center of mass (COM) of the trunk. Doing so serves to lower the magnitude of dynamic torques imparted upon the leg servos during gaiting by decreasing the net moment due to gravity imparted upon the system when the body is oriented away from the direction of gravitational force. The entirety of the Main module is attached to the Root module through the battery holster sub-assembly by four plastic bolts.

The Head Module

Two separate Head modules have been designed for the BlueFoot system : one of which features a stereo camera pair and a Piccolo LIDAR sensor (PLDS); and a monocular design, which features a camera and a Hokuyo-URG LIDAR sensor, as shown in Figure 4. Each head module is attached to Main module via four plastic mounting screws. In the stereo-camera design, two adjustable wings are attached to either side of a top platform which hold cameras. These wings were designed to be adjustable to aid in stereo-camera configuration and calibration. The position of each camera on the trunk allows for a persistent field of view by each camera during mobilization. The PLDS unit is positioned such that the center of its rotating laser head is aligned to the center of the trunk.

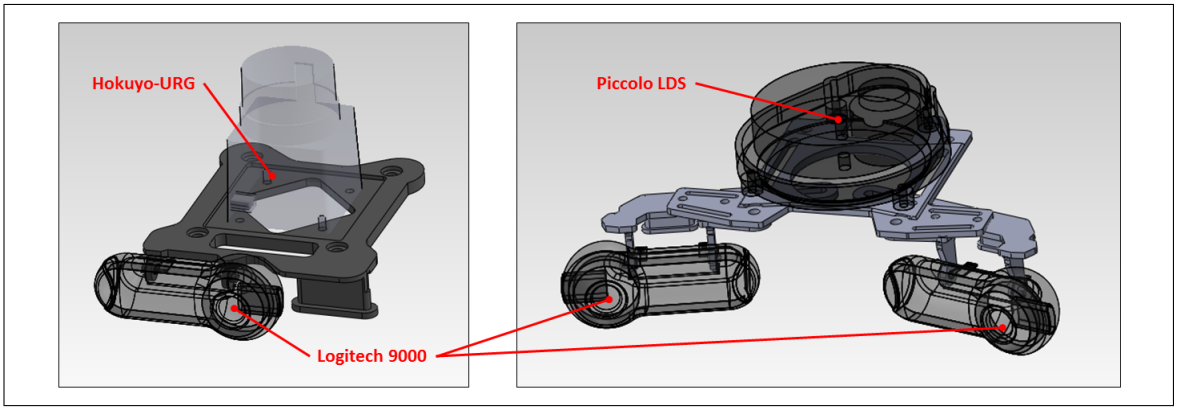

Figure 4: Head section of trunk. Monocular camera configuration with Hokuyo-URG *(left)*; and Stereo configuration with PLDS *(right)*.

In the monocular design, a single camera is mounted such that the lens of the camera is aligned to the sagittal plane of the trunk. This configuration is currently be used as BlueFoot's *primary* head configuration and is mainly being used for 3D point-cloud building and surface reconstruction via 2D LIDAR scans. This is because the Hokuyo-URG laser scanner used in this configuration offers higher-resolution laser-scan outputs, which will be covered in more detail later in this chapter.

2.2.2 Leg Designs

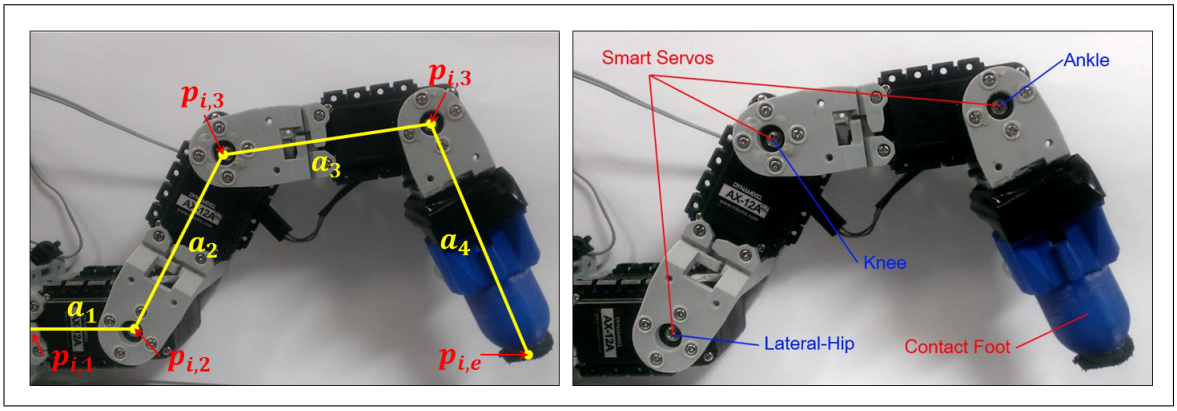

Figure 5: Closeup of BlueFoot's leg. *(left)* shows effective link lengths and the location of defined joint positions.

Each of BlueFoot's legs are identical and are comprised of four Dynamixel AX12a smart-servo actuators (see Figure 5). These actuators are connected via dedicated Dynamixel mounting brackets. Feet are attached to the ends of each leg which contain an

embedded, two-state contact sensors. Each foot is designed with a spherical tip, which is rubberized to provide extra grip. The ankle joint of the platform has been added such that the platform can reconfigure its foot orientation while retaining a constant spatial position during gaiting. Additionally, this configuration allows for a considerable amount of independent body re-orientation and repositioning. This capability extends itself to the stabilization and gimbaling of vision sensors mounted on the upper body of the platform while the platform is in motion.

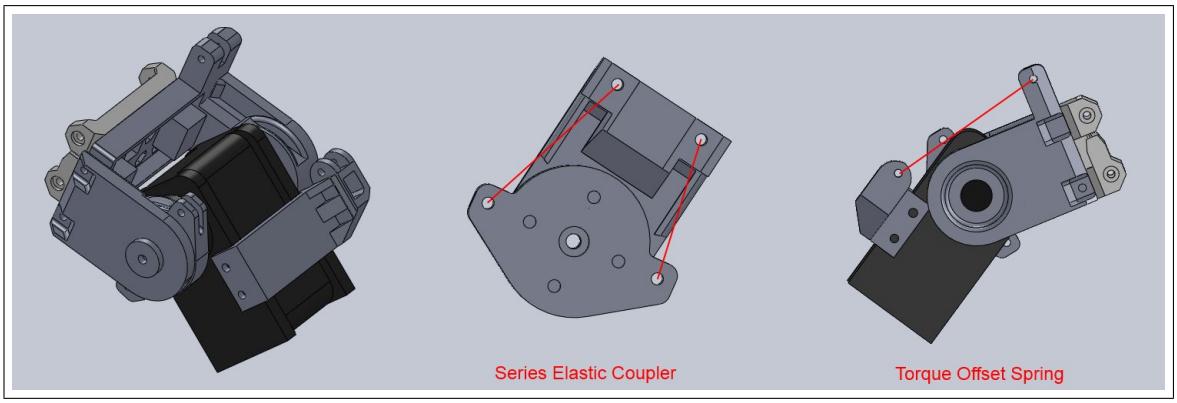

Figure 6: Series elastic brackets.

Though not kept in the system's final design, some experimentation was performed with the incorporation of series elastic joints, which were designed to relieve joint impact during gaiting. Series elastic actuation was achieved by replacing bracket interfacing the first and second hip joints of each leg with an elastic-compliant mounting bracket. This bracket includes spring loaded member which was mounted to the horn of the second hip servo on each leg, as shown in Figure 6 and allowed the leg to deflect a small amount at the lateral hip (second joint).

Link lengths, a_1, a_2, a_3 and a_4; and offset from the center for the Root module to the first joint of each leg, ν are defined in Table 1. These parameters are identical for each leg, and are corresponded with physical leg members labeled in Figure 5.

2.3 Computational and Sensory Hardware

Major payloads on-board the BlueFoot robot are as follows:

- Dual processor AutoPilot unit with a 12-axis inertial measurement unit
- ODROID-XU Computer

Link	Length, m
a_1	0.06500
a_2	0.06500
a_3	0.06500
a_4	0.06500
ν	0.09215

Table 1: Link and body-offset lengths for each leg.

- Logitech 9000 Web-cameras
- Hokuyo-URG / Piccolo LIDAR Units (configuration dependent)
- Two-state foot contact sensors (x4)
- Dynamixel AX12a Smart Serial Servos (x16)
- XBEE Wireless radio

Device selection has remained mostly consistent since the platform's inception and initial design, with the exception of computing its main computing units. The AutoPilot unit was updated from an older model, and the ODROID-XU computer replaced a Beaglebone computer for the sake of improving overall computing power.

2.3.1 Device Descriptions

AutoPilot

A dual processor AutoPilot unit performs BlueFoot's low-level gaiting and actuator control tasks, as well as handles communications with a computer running ground-station software. Given the set of low-level sensory and motor-handling tasks it performs, this module has been named the the "Lower Brain" (LB) of the system. The AutoPilot consists of two processing units: a TM4C and RM48 micro-controller (MCU), which operate at 80 MHz and 220 MHz, respectively. These processors communicate over a single UART line, which is used to transfer packeted data between the two processors using a unified inter-processor data transfer protocol, EXI. This protocol which will be described later in more detail. One UART of the RM48 MCU is also connected to an on-board computer, an ODROID-XU, through a USB-to-serial connection. The AutoPilot is powered via an external 12 V supply.

This AutoPilot unit includes a 12-axis inertial measurement unit (IMU) which consists of two, 3-axis accelerometers; one 3-axis rate gyro; and a 3-axis magnetometer unit. This sensor is used for acquiring angular rate data of BlueFoot's trunk and estimating of trunk orientation states using an Extended Kalman Filter (EKF).

ODROID-XU

An ODROID-XU performs many of system's high-level planning tasks, such as navigation, image processing and terrain reconstruction; and handles data data acquisition from both camera and LIDAR sensor units. Given that this unit performs mostly high-level planning tasks, it has been given the name "Upper Brain" (UB). This computer contains a 1.6 GHz, quad-core processor with 2 Gb of RAM. The ODROID-XU can be communicated with over WiFi via a USB WiFi antenna. Currently, SSH tunneling is used to start processes on the ODROID remotely and stream data. The ODROID-XU is powered via an external 5V connection.

Logitech 9000 Web Cameras

Logitech 9000 web cameras have been selected for creating a stereo camera pair, as well as for use in a single camera configuration. These cameras are high-definition web cameras and have a maximum frame rate of 30 fps and a max resolution of 1280 by 720. Cameras are currently read at a the max rate of 30 fps at a more conservative resolution of 640 by 480. These settings are adequate for image processing tasks and have been chosen to reduce nominal data throughput. These cameras are interfaced with the UB (OROID-XU) over a USB connection.

Laser Distance Sensors (PLDS and Hokuyo-URG)

The Piccolo Laser distance sensor (PLDS), which is used in BlueFoot's stereo-camera type configuration, is a 4 meter spinning-head laser range finder. The PLDS has a resolution of a point per degree and covers a range of 360 degrees. Ranging frames (which covers a full rotation) are acquired at a rate of 5 Hz, and are dispatched over a serial connection at 115200 baud. An FTDI break-out board is used to convert the sensor's raw serial output to USB protocol so that the sensor can be interfaced with the UB unit. The PLDS is powered via and external power 5 V source, which is regulated to a 3.3 V voltage level for powering the motor which spins the laser head, and 1.8 V for internal logic. Regulation is performed by an auxiliary power circuit.

The Hokuyo-URG Laser Distance sensor, which is used in BlueFoot's single-camera head configuration, has a a range of 5.6 metets and an angular resolution of 0.38 degrees per point (628 points per scan). This scanner covers a total angular range of 240 degrees. Ranging frames are acquired at a rate of approximately 10 Hz and dispatched directly over a USB connection at 115200 baud. The unit is powered directly over USB.

Foot Contact Sensors

Binary-state contact sensors are embedded in each foot. These contact sensors are essentially limit-switches which generate an active-low signal when the foot comes in contact with the ground. Each sensors is connected to ground and a GPIO pin on the TM4C MCU of the AutoPilot. A 500 Ω is added in series with the limit-switch for the purpose of pin protection.

Dynamixel AX12a Smart Serial Servos

BlueFoot uses 16 Dynamixel AX12a servo units (4 per leg). These servos are position-controlled and commanded over a daisy-chained, half-duplex serial bus (i.e. single wire) at a rate of 1 Mbps. These servos have a maximum holding torque of 1.618 N m and top speed of 306 degrees/s. The AX12s provide position, velocity and loading feedback, however velocity feedback is not used. Servo velocities are, instead, estimated in real time from position feedback because velocity readings provided by the AX12 is relatively noisy by comparison.

Commands are sent to the servos via an aggregate command packet which contains goal-position values for all servo units. Feedback is collected from each servo using individual data-request packets. Servos respond to each request with a response packet containing a corresponding feedback value. Given the number of servos in the network; communication overhead; and the one-wire communication configuration, servo updates are limited to a maximum update rate of 50 Hz over a half-duplex communication line. Gathering feedback over the half-duplex communication bus is particularly expensive because feedback requests require that the host processor wait after each dispatched for a response from each targeted servo. Moreover, each request/response cycle must finish to completion before a feedback request is made to another servo on the communication bus.

A dedicated circuit has been designed for use with these servos which converts a full-duplex serial line to a half-duplex AX12 bus. The circuit uses a two-state tri-

state buffer which is switched via a general-purpose I/O line. This switching circuit is integrated into the system's main power switching and distribution board. Each servo is powered via an fused, software-switched 12 V supply line.

XBEE Wireless Radio

An XBEE Wireless Radio, shown in Figure This could be the picture from before, is used for communication between the LB and an external computer ground-station. The radio is interfaced with the LB via 57600 baud serial connection. This radio has a range of outdoor range of 27 meters and a maximum one-way transfer-rate of 115200 bps. Transfer rates between the LB and ground station are currently being limited to 57600 bps to compensate for a lack of hardware flow-control, which is required for stable, two-way communication between two XBEE radios at maximum communication rates. However, the selected communication rate is more than adequate for transferring necessary control information to and from the system without the need for additional flow-control hardware.

This wireless endpoint is used currently used interchangeably with the ODROID-XU's wireless WiFi radio, but will soon be retired to simplify hardware design and increase the platform's data streaming capabilities by switching to a WiFi-based line of communication. Ground station software, as well as the system's internal command-routing and networking software, is designed in such a way to easily accommodate this change.

2.3.2 Device Networking

Figure 8 depicts how each major device is connected within the system and details the communication rates (f_{com}) between networked devices. Accompanying specifications are detailed in Table 2, which summarizes all device communication pairs and their corresponding baud rates.

2.4 System Power

2.4.1 Power Routing

System power routing is handled via an integrated power switching and distribution board. This board includes physical, main power switches which connects external power to two main, internal 12 volt buses for computer power (Net-1) and motor power (Net-2), respectively. The board also regulates system input voltage to a 5 V bus for use with

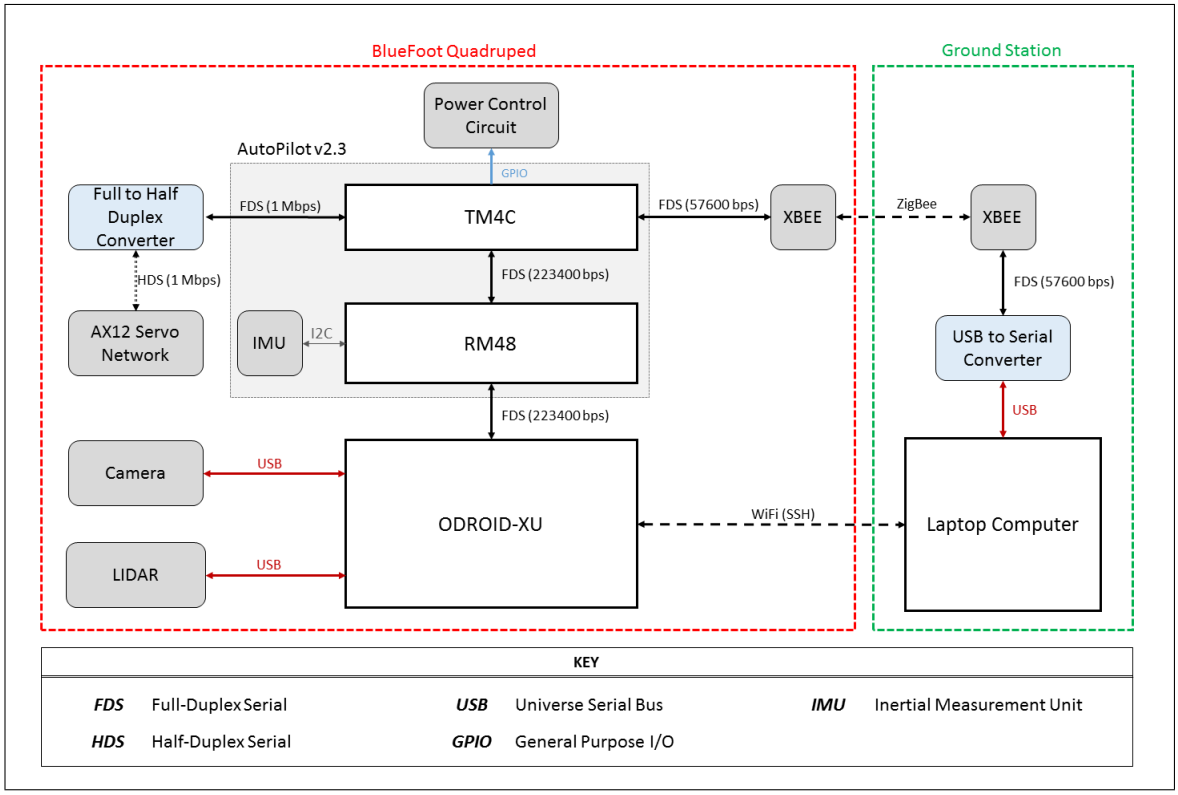

Figure 7: BlueFoot device networking diagram.

Port$_A$	**Source**$_A$	**Port**$_B$	**Source**$_B$	f_{com}, **kbps**
UART0	TM4C	DIN/DOUT	XBEE*	55.7
UART2	TM4C	DIN+	AX12 Net.	1000
UART1	TM4C	LINSCI	RM48	223.4
SCI	RM48	USB (FTDI)	ODROID-XU	223.4
USB	ODROID-XU	DIN/DOUT	LIDAR	115.2
USB	ODROID-XU	USB	Cameras	No Spec.

Table 2: System communication port-pairs and corresponding data transfer rate*Refers to the on-board XBEE module which communicates with the ground station.

on-board ICs and 3.3 V bus for powering the XBEE radio. Regulated power and power the servo motors of each leg controlled via three, two-channel power-switching IC's, which are toggled using six digital I/O pins on the TM4C processor of the LB. These power-switching chips allow for software-controlled power configuration, and further, software controlled emergency power cutoff to the servo motors. System main power is

supplied via four 12 V (3 cell), 2 Ah Lithium Polymer battery packs.

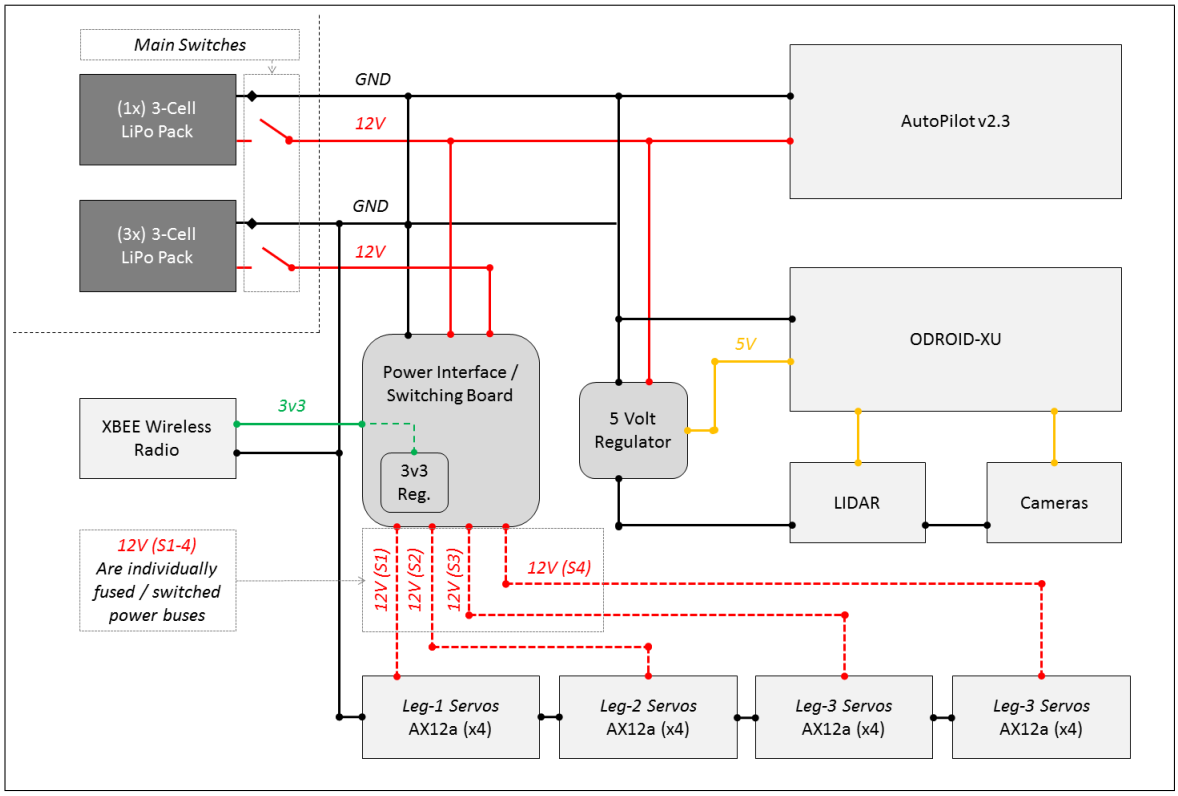

Figure 8: BlueFoot power routing.

2.4.2 Energy Requirements and Runtime

The power consumptions of BlueFoot's component device's are summarized in Table 3, which provides the operating voltage, V_{op}, and nominal current draw, I_{nom}, of each active, on-board component. Table 4 details battery specification (output voltage and amp-hour rating) and BlueFoot's estimated run-time under nominal operating conditions.

Net	**Device**	V_{op}, V	I_{nom}, A
1	AutoPilot (RM48, TM4C)	12.0	0.40
1	XBEE Radio	3.3	0.25
1	ODROID-XU	5.0	2.0
1	Logitech-9000	5.0	0.1
1	Hokuyo-URG	5.0	0.5
2	AX12a Servos (x16)	12.0	9.6 (0.6 ea.)

Table 3: Power consumption summary by device (for single-camera with Hokuyo-URG).

Net	**Battery Pack**	V_{out}, V	**Rating,** $A.hr$
1	3S LiPo Pack (x1)	12.0	2.0
2	3S LiPo Pack (x3)	12.0	6.0
Total Estimated Runtime		35-40 minutes	

Table 4: Battery power supply and estimated runtime summary, assuming low regulator losses.

CHAPTER III

Software

3.1 System Software Architecture

BlueFoot is controlled using a multi-processor software architecture which incorporates several independent control-software cores. Each software core handles specific subsets of operations essential to the macro-system. This distributed system architecture allows independent tasks (such as actuator command/feedback handling, battery monitoring, etc.) to be decoupled from more computationally heavy tasks by offloading them to physically separate computing modules. Therefore, each control unit handles an assigned task set in an independent control loop which contribute updates to the overall macro-system in an asynchronous fashion. System control tasks are divided into four main categories, which can be summarized as follows:

- *Low-Level Control* : power monitoring/switching, actuator command handling, communications routing, sensor data acquisition, script parsing and evaluation
- *Motion/Motor Control* : gait planning, gait adaptation, trunk pose adaptation
- *High-Level Control* : vision sensor handling, perception, motion planning, surface reconstruction, navigation, localization
- *Human-Operator Control* : joystick/keyboard commands, scripting commands

As previously mentioned in Chapter I, low-level and motor/motion control tasks are handled, exclusively, by the Lower Brain (LB), which is comprised of a software collective spanning over the RM48 and TM4C processors on-board the AutoPilot. High-level control tasks are handled by the Upper Brain (UB), which is the software-core which runs on the ODROID-XU module. Lastly, a human operator can interface with the system wirelessly from a personal computer running ground-station software. This ground station software which communicates directly with the TM4C processor. The ground-

station also interfaces with the UB over an SSH connection. This secondary wireless connection is used, mainly, for on-board data-logging configurations.

Since this software architecture is distributed over several separate computational units, an integral part of this control architecture is an efficient, reconfigurable inter-processor communication protocol. Namely, BlueFoot utilizes data packets transfered over serial lines to update system states between processors. These data packets are formatted using a binary-XML protocol, called EXI. This protocol facilitates a highly customizable packeting structure for asynchronous inter-module communication and utilizes robust packet-error checking routines. This sections will detail the specifics of BlueFoot's interprocessor communication protocol, namely the composition of packets transferred between processor.

This section will also detail the specific software-level tasks handled by each of BlueFoot's processor; the speed at which each core software element is run (update frequency); and what data must be communicated between software elements for operation. Additionally, this section will describe the ground-station software and corresponding user-interface used to control the BlueFoot Quadruped and administer high-level commands.

3.1.1 Processor Tasks Descriptions

Figure 9 depicts how core software/control elements are related within the BlueFoot software macro-system. This section will detail a general description of the tasks carried out by each major software module implemented on the BlueFoot quadruped.

TM4C (Lower Brain)

As previously mentioned, the TM4C processor on-board the AutoPilot module is responsible for *Low-Level* tasks and can be viewed a safety/communications routing co-processor within the overall system. Within its main program loop, the TM4C polls the system's main battery voltage via ADC interface routines; handles transmit and receive (packet decoding) routines between the ground-station and the RM48 system nodes; and handles command dispatching and feedback polling with the system's 16 servo actuators. The TM4C is directly interfaced with two dual-channel power switching IC's and is used to control power supply to each leg by toggling general purpose IO pins in software. The state of these pins is administered as part of a periodic packet command/update packet sent from the ground-station. Since the TM4C has this control

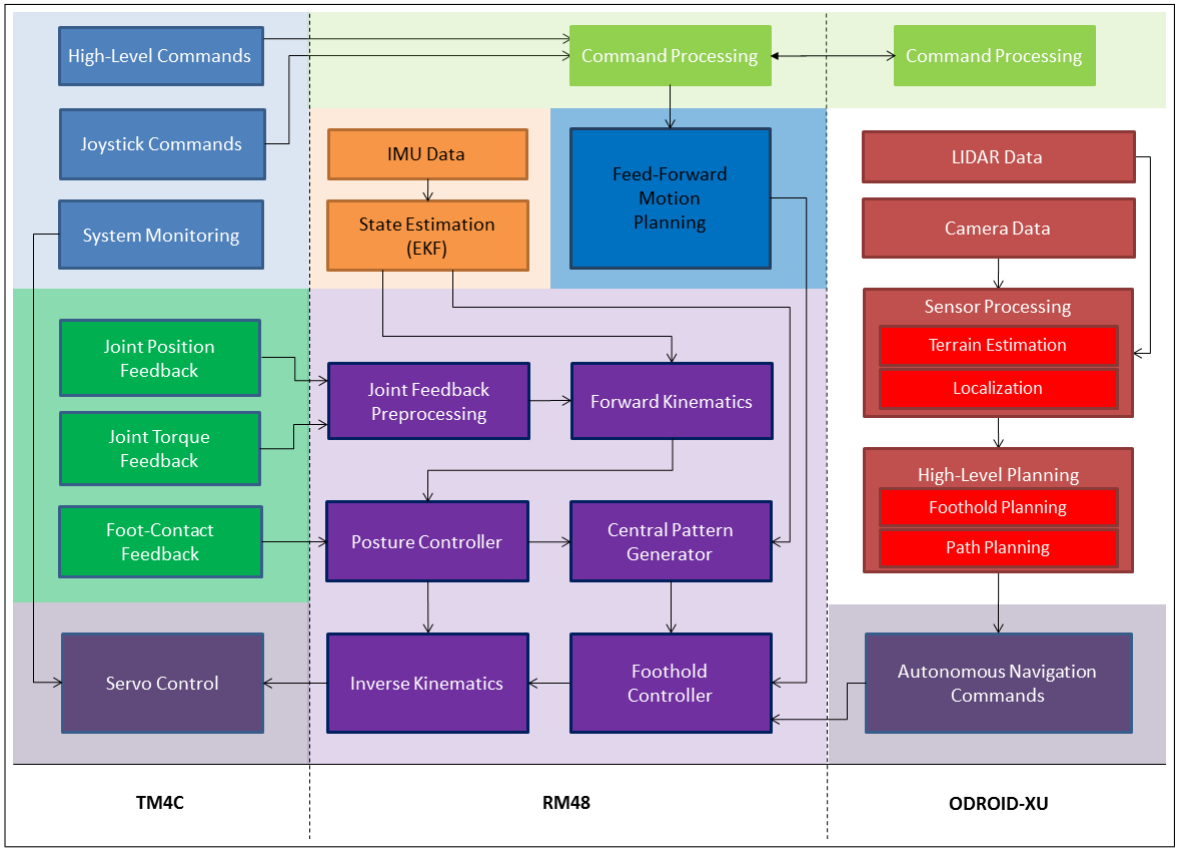

Figure 9: BlueFoot's main processes and their relationships.

over the system's actuators (which consume most of the system's power) and battery monitoring capabilities, it runs a safety routine which is responsible for halting motor activity and/or cutting system power on low-battery or power-fault conditions, as well as during unexpected breaks in communication with the ground-station.

As previously mentioned, the TM4C handles communication routing between system processing modules; as well as with controllers on-board each smart-servo. Administering servo commands and collection servo feedback it the TM4C's highest priority task. This process, which involves both commanding and requesting feedback from each servo, is relatively expensive and limits the TM4C's loop frequency to roughly 50 Hz. Thus, it is particularly important that this task is offloaded to this processor, as its other safety and communication-related tasks are much less expensive, by comparison, and allow the servo actuators to be updated quickly as possible without encumbering other system control operations.

RM48 (Lower Brain)

The RM48 is responsible for several *Low-Level* tasks, including IMU polling and handling communication with the TM4C and ODROID-XU. Each collected IMU sample is passed along to an extended Kalman Filter routine, which generates a trunk orientation estimate, $\hat{\theta}_b$ in the world frame, O_0.

The RM48's primary function is to carry out motion control and gait-planning tasks. To achieve this, the RM48 handles a state machine which switches between planned motion execution and trajectory control; and gait control via a Central Pattern Generator (CPG) based gaiting controller, which will be discussed in more detail in Chapter V. Additional functions for body and posture (position and orientation) control, including trunk leveling procedures, and gait-stabilization are run in tandem with the aforementioned gait-control task.

Motion and gait controls, which are performed in the robot task-space, are converted into joint-space reference angles, q^r, via an inverse kinematics (IK) routine. The IK routine is exectued at all times when the legs are engaged for the purpose of issuing servo position commands, given desited task-space configurations for body and feet positions, which will be more formally defined in Chapter IV. The RM48 also maintains BlueFoot's forward kinematic model (specifically, foot position relative to the trunk), which relies on an EKF-generated trunk orientation estimate, $\hat{\theta}_b$, and joint position feedback, q. BlueFoot's inverse and forward kinematics models will be detailed in Chapter IV. The RM48 runs its full control loop at approximatively 100 Hz (twice the speed of the TM4C control loop) to facilitate higher integration stability when updating gait related controller dynamics, dynamic motion controls, task-space reference trajectories.

Lastly, the RM48 handles an on-board scripting engine (based on the MIT Squirrel Scripting), which interprets lexical commands. This scripting engine is capable of handling a large number of high-level commands [http://squirrel-lang.org/] and is complex enough to handle function and class definitions in real time. The scripting engine currently being used to evaluate BlueFoot's core user command set, ranging from simple state toggling and parameter modification, to the prescription of user-specified waypoints for navigation, among other high-level command itams. Scripting commands are passed from the ground station (via terminal) and routed through the TM4C to the RM48, where they are finally evaluated.

ODROID-XU (Upper Brain)

The ODROID-XU runs software upon a Debian (Linux) operating system distribution "Jessie." The use of an Linux operating system extends itself to a number of programmatic conveniences, such as to ability to run several tasks in parallel threads. Inbuilt USB drivers are used in functions which are used to acquire data from USB-interfaced vision sensors. Namely, the ORDOID runs sensor handling elements used for acquiring and buffering camera images and controlling camera frame-rate control; as well as LIDAR scan frames. The ORIOID uses these sensor inputs, in conjunction with orientation estimates and inertial data passed from the RM48 to perform several navigation-related tasks (*e.g.*, potential fields, mapping, localization, terrain-reconstruction). These tasks will be described in more detail in Chapter VI.

The ODROID utilizes 2D-LIDAR scans (frames) and trunk-pose estimates to form organized 3D point clouds. These point-clouds are further processed to reconstruct 3D terrain surfaces and height-maps, which are then used for step-planning. LIDAR frames are utilized in potential fields-based navigation tasks. These navigation modes also incorporate camera data for the purpose of goal-targeting (where the goal is typically an object of particular shape or color). Image processing and image feature detection is run as a separate process on the ODROID, which is incorporated with the aforementioned processed sensor data to produce a set of forward and turning velocity commands, v_c and ω_c, respectively; as well as foot-hold positions generated from step-planning algorithms. In particular, the open-source libraries OpenCV (Open Computer Vision Library), OpenPCL (Open Point-Cloud Library), and Boost are heavily used in the software developed to carry out the aforementioned tasks. Software written for this platform was generated using a mixture of C++ and Python.

As previously mentioned, the ODROID can handle a limited set user-command on its own, which are administered directly to the ODROID from an SSH terminal on the ground-station computer. These commands include core-program start-ups and data-logging configurators. Essentially, the ODROID's software core is designed as a completely independent software module which replaces the roll a human director, as it handles the bulk of the systems high-level planning and navigation tasks. Moreover, if the ODROID is removed from the BlueFoot system, the system can still be operated via remote-control heading commands provided from human operator (*i.e.*, ground-station joystick control).

3.1.2 Inter-processor Communication

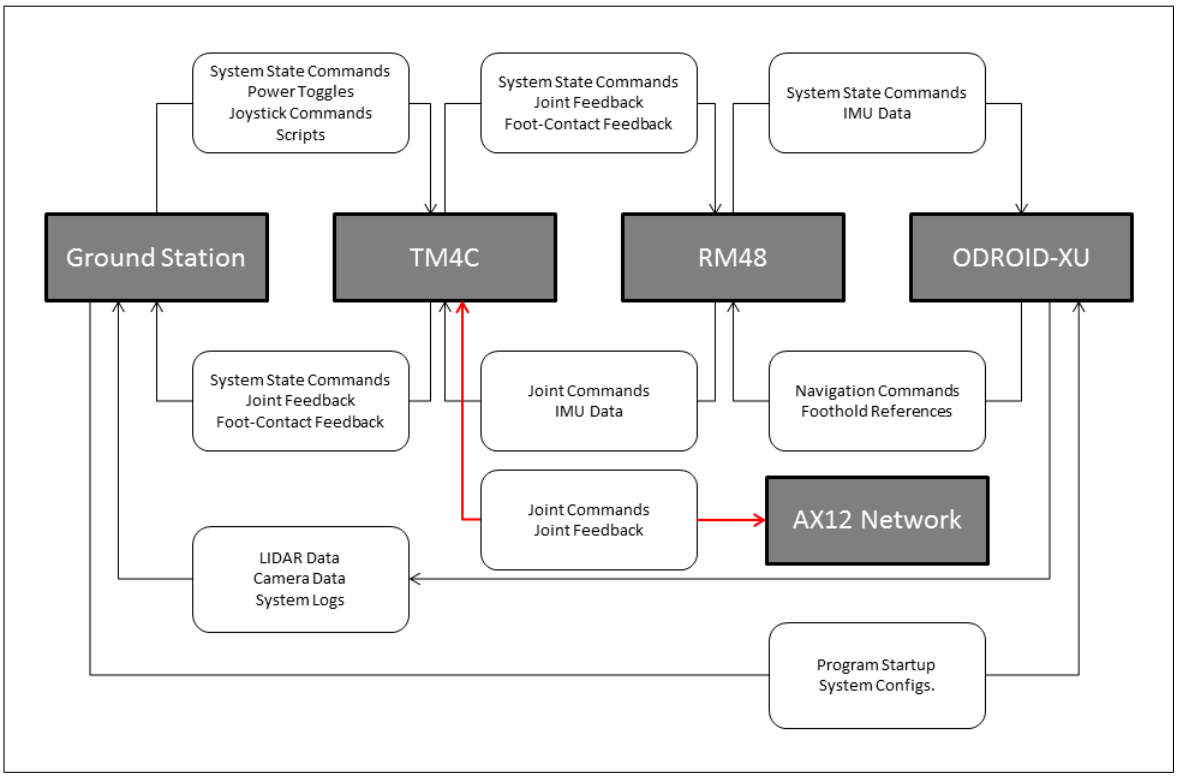

Figure 10: Communication flow between processors on the BlueFoot Platform.

This section will detail the contents of the data packets transfered between processors, which is summarized in Figure 11. System directives, generated by a human operator who interacts with the robot via a graphical user interface and/or joystick controllers, are generated from a ground-station computer. Packets (without padding) sent from the ground station to the BlueFoot robot (TM4C) are composed as shown in Table 5:

32-bits	*8-bits*	*8-bits*	*16-bits*	*16-bits*
HEADER	Master-Tog.	Power-Tog.	Unused	Network Info
Variable				
Scripts				

Table 5: Structure of the packets sent from Ground-Station to TM4C.

Every packet issued using the EXI protocol has a 4-byte (32-bit) header. As part of the internal system protocol, every packet sent between system nodes contains a “Master

Status Vector", which is comprised of a fixed length, 7-byte sequence of essential system information/control items. The "Master Toggles" section (first 8-bits after header) enumerate major systems states, including *On-line, Standby, Off-line* and *Suspended* system state designations. The next 8-bits are used to toggle on-board power, namely the power supplied to each leg. The remaining 32-bits are used for specifying battery voltage (8-bits), power-fault states (8-bits), generic binary feedback toggles (8-bits), and system networking information (16-bits). The last section of this packet contains scripting commands, which can be of varying lengths. For example, foward velocity and turning rate commands (gathered form a joy-stick controller) are administered in the form of scripted commands.

Packets sent from the TM4C to the Ground-station contain status items generated on board the robot, and appear as shown in Table 6: The *Joint Pos. FB* (joint position

32-bits	*8-bits*	*8-bits*	*8-bits*	*16-bits*
HEADER	Master-Tog.	Unused	Foot-Contacts	Network Info
2048-bits				
Joint Pos. FB				

Table 6: Structure of the packets sent from the TM4C to the Ground-Station.

feedback) element is composed of 16 2-byte sequences corresponding to the joint positions of read-back by each actuator. To avoid redundancy, the structure of the packets communicated from the TM4C to the RM48 and vice-versa will not be depicted explicitly. Like all system packets, these packets contain a 7-byte master status vector with only the *Master-Toggle* and *Network Info* fields populated. The TM4C sends the same joint feedback information to the RM48 as it does the Ground Station. For packets sent from the TM4C to the RM48, this field is replaced with corresponding joint-position commands for each of the 16 servo actuators. This field is also 2048 bits in length.

Packets sent from the RM48 to the ODROID contain additional dynamical-state fields for use in planning on the ODROID. State information is sent in the form a vectors with 32-bit, single precision floating-point elements. This set of information includes trunk a orientation estimation, angular rate, and global position (generated from open-loop command integration), each of which are represented as 3-element vector; and foot-position estimates (four, 3-element vectors.) generated. These packets have a structure which is depicted in Table 7. Packets sent form the ODROID to the RM48 contain

32-bits	*8-bits*	*8-bits*	*8-bits*	*16-bits*
HEADER	Master-Tog.	Unused	Foot-Contacts	Network Info
96-bits	*96-bits*	*328-bits*	*328-bits*	
Orientation	Angular Rate	Trunk Pos.	Foot Positions	

Table 7: Structure of the packets sent from the RM48 to the ODROID.

command items, such as forward velocity, turning rate and trunk-pose commands, as well as foothold-references and corrected global trunk position estimates. These packets are constructed as shown in Table 8 :

32-bits	*8-bits*	*8-bits*	*8-bits*	*16-bits*
HEADER	Master-Tog.	Unused	Unused	Network Info
32-bits	*32-bits*	*96-bits*	*328-bits*	*96-bits*
Velocity	Turning Rate	Trunk Pose	Footholds.	Trunk Pos.

Table 8: Structure of the packets sent from the ODROID to the RM48.

3.2 Ground Station

Figure 11: The BlueFoot ground-station GUI.

Ground-station software used for controlling the BlueFoot platform was generated in C++ using an open-source graphical user interface design library called *wxWidgets* [Need ref]. *wxWidgets* is used, primarily, for the ground-station's front-end design. Namely, the ground-station code is designed such that that its US design is reconfigurable via an XML-based design specification file. Furthermote, this allows for UI reconfigurations without the need to change internal, back-end processes. In terms of the GUI backend, *wxWidgets* handles general UI signal processing, and allows for easy registration of user callbacks on events such as button presses. Additionally, *wxWidgets* provides an interface for collective joystick inputs, which are interpretted and sent to the BlueFoot as remote-control commands. *Boost*, a C++ utility library, is utilized to provide socket and serial-port IO handling. This library is particularly important for use with the ground-station's wireless radio, which transmits information to the robot from data sent to the radio module over USB.

The ground station is composed of several main UI sections which allow for system master-state, opterating-state and power toggling; administering system commands and script. Basic system state controls are used to periodically populate a fixed-structure, robot-command packet, shown in Table 5. This packet is used to refresh the robot operatin state with used commands are a rate of 25 Hz and is sent to the robot using the aforementioned XBEE radio module. BlueFoot replies to each ground-station update with a packet of internal configurations, as shown in Table 6, which is then used to ensure that system updates and command are being adminstered properly and that the system is live.

BlueFoot also sends back battery volatge levels; power fault coniditons; foot contact feedback; and joint position feedback, which are used to update several key portions of the UI. Firstly, battery levels are used to update a battery meter, which indicates the current system volatge level. Additionally, a dynamic text box displays the system's power-fault state to the user. Joint position commands are used to update a visualization of the robot in real time. The foot-color of the vizualized BlueFoot robot changes from blue to red to represent foot-contact conditions. Joint positions, foot-contact states and battery levels are time-stamped and automatically logged for each ground-station session for later inspection.

CHAPTER IV

System Modeling

4.1 Kinematic Model

The kinematic model of the BlueFoot platform is paramount for foot/body trajectory planning, localization, and adaptation. In particular, inverse position and velocity solutions are used to perscribe joint-space commands from particular foot trajectories planned within the world coordinate frame. Additionally, foward kinematic solutions are utilized to localize the position of each foot using a combination of localized trunk position/orientation and joint position feedbback. This section will fully describe the foward and inverse kinematic models which describe the BlueFoot platform and are how these models are used in motion planning and control tasks.

4.1.1 Forward Position Kinematics

To formulate the kinematics model, a set of coordinate systems have been defined and are described by **??**. Note that the frame O_0 represents the world coordinate frame; and O_b is the coordinate frame, centered at p_b attached to the platform and is always aligned with O_0. O_b represents a body frame rigidly attached to the center of the trunk. The orientation and position of the trunk are defined by vectors of $\theta_b \in R^3$ and $p_b \in R^3$, which relate the frame O_b to the world frame O_0. Coordinate frames $O_{i,0}$ are attached to the first joint of each i^{th} leg. The j^{th} joint position of each i^{th} leg is represented by the points $p_{i,j}$ in the frame O_0. These spatial locations are generated from a combination of the body orientation, θ_b, and joint positions for each i^{th} leg, $q_i = [q_{i,1}, q_{i,2}, q_{i,3}, q_{i,4}]^T$. $q_{i,1}$ represents the position of the hip-joint (joint closest to the center platform), which rotates in the direction of the transverse body plane. The joint variables $q_{i,2}, q_{i,3}$ and $q_{i,4}$ represent the lateral-hip, knee and ankle joint rotations, respectively.

The coordinate transformation between world coordinate frame, O_0, and the zeroth Denavit-Hartenberg (DH) coordinate frame of leg i, $O_{i,0}$ (located at the origin of joint-1),

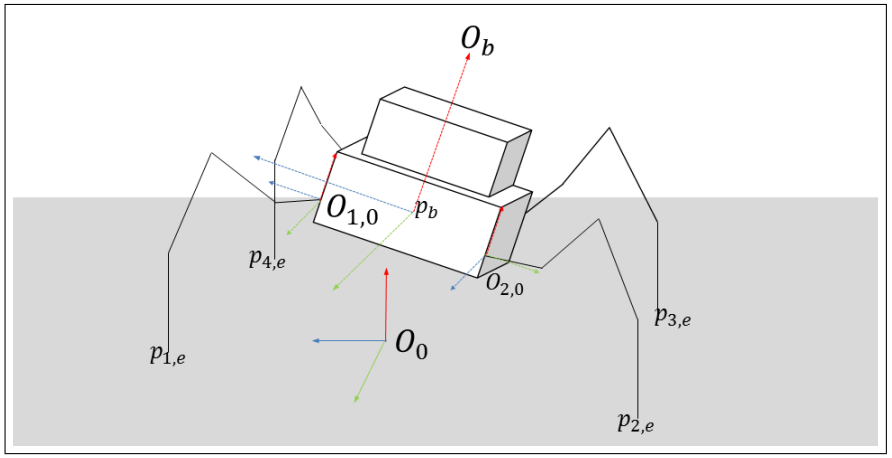

Figure 12: Coordinate frame setup.

is given by

$$H_0^{i,0} = \left[\begin{array}{c|c} R_{zyx}(\theta_b)R_z(\sigma_i) & R_{zyx}(\theta_b)\nu + p_b \\ \hline 0 & 1 \end{array}\right] \tag{4.1}$$

where $\sigma_i \equiv \frac{\pi}{2}(i-1) + \frac{\pi}{4}$ and $\nu \equiv R_z(\sigma_i)\beta$ with β defining an offset from O_b to where the first joint of each leg is attached to the body. R_{zyx} represents a rotation associated with the pitch (x-axis), roll (y-axis), and yaw (z-axis) angles of the main body about the platform frame, θ_b. R_z represents a rotation about the (z-axis) of the frame O_b.

A transformation from the zeroth DH frame of the to the $(j+1)^{th}$ joint of leg i is described, in general, by

$$H_{i,j}^{i,0} = \left[\begin{array}{c|c} R_{i,j}^{i,0} & p_{i,j}^{i,0} \\ \hline 0 & 1 \end{array}\right]. \tag{4.2}$$

The kinematics of each leg are identical. Thus, the transformations $H_{i,j}^{i,0}$ are of the same form and are derived by the DH parameters given by Table 9. Actual values for the link lengths a_{1-4} and body-offset, ν, are provided in Table 1.

Link	a_i	α_i	d_i	θ_i
1	a_1	$\pi/2$	0	$q_{i,1}^*$
2	a_2	0	0	$q_{i,2}^*$
3	a_3	0	0	$q_{i,3}^*$
4	a_4	0	0	$q_{i,4}^*$

Table 9: DH parameters for all legs.

Using these DH parameters, the transformations $H_{i,1}^{i,0}$, $H_{i,2}^{i,0}$, $H_{i,3}^{i,0}$, and $H_{i,4}^{i,0}$ can be computed explicitly as follows:

$$H_{i,1}^{i,0} = \left[\begin{array}{ccc|c} c_{1,i} & 0 & s_{1,i} & c_{1,i}a_{1,i} \\ s_{1,i} & 0 & -c_{1,i} & s_{1,i}a_{1,i} \\ 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \tag{4.3}$$

$$H_{i,2}^{i,0} = \left[\begin{array}{ccc|c} c_{1,i}c_{2,i} & -c_{1,i}s_{2,i} & s_{1,i} & c_{1,i}(a_{1,i} + a_2c_{2,i}) \\ s_{1,i}c_{2,i} & -s_{1,i}s_{2,i} & -c_{1,i} & s_{1,i}(a_{1,i} + a_2c_{2,i}) \\ s_{2,i} & c_{2,i} & 0 & a_2s_{2,i} \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \tag{4.4}$$

$$H_{i,3}^{i,0} = \left[\begin{array}{ccc|c} c_{1,i}c_{23,i} & -c_{1,i}s_{23,i} & s_{1,i} & c_{1,i}(a_{1,i} + a_2c_{2,i} + a_3c_{23,i}) \\ s_{1,i}c_{23,i} & -s_{1,i}s_{23,i} & -c_{1,i} & s_{1,i}(a_{1,i} + a_2c_{2,i} + a_3c_{23,i}) \\ s_{23,i} & c_{23,i} & 0 & a_2s_{2,i} + a_{3,i}s_{23,i} \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \tag{4.5}$$

$$H_{i,4}^{i,0} = \left[\begin{array}{ccc|c} c_{1,i}c_{234,i} & -c_{1,i}s_{234,i} & s_{1,i} & c_{1,i}(a_{1,i} + a_2c_{2,i} + a_3c_{23,i} + a_4c_{234,i}) \\ s_{1,i}c_{234,i} & -s_{1,i}s_{234,i} & -c_{1,i} & s_{1,i}(a_{1,i} + a_2c_{2,i} + a_3c_{23,i} + a_4c_{234,i}) \\ s_{234,i} & c_{234,i} & 0 & a_2s_{2,i} + a_3s_{23,i} + a_4s_{234,i} \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \tag{4.6}$$

The position of joint 1 of leg i in O_0, $p_{i,1}$ may now be computed with respect to frame O_0 by:

$$p_{i,1} \equiv E_p H_0^{i,0} e_p. \tag{4.7}$$

where

$$E_p = [I_{3\times3}, 0_{3\times1}]$$
$$e_p = [0_{1\times3}, 1]^T.$$

The position of joints 2-4 of leg i, may now be computed with respect to frame O_0 by:

$$p_{i,j} \equiv E_p H_0^{i,0} H_{i,(j-1)}^{i,0} e_p. \quad \forall \quad j \in 2, 3, 4 \tag{4.8}$$

Finally, the position of the end-effector (foot) of i^{th} leg. $p_{i,e}$, is achieved as follows:

$$p_{i,e} \equiv E_p H_0^{i,0} H_{i,4}^{i,0} e_p. \tag{4.9}$$

A Note about Foot Localization in World and Robot-Relative Frames

Using the previously defined forward kinematics model and the relationships defined in and , the position of each joint and foot can be computed assuming the p_b and θ_b are known (or can be estimated) in the frame O_0. Provided inertial feedback, trunk orientation, θ_b, can be estimated by the use of a Extended Kalman Filter (EKF), or one of many other orientation estimation methods (*i.e.*, complimentary filter, etc.). p_b, however, requires more sophisticate localization measures, such as a Simultaneous Localization and Mapping (SLAM) scheme or absolute positioning via overhead camera or GPS. In lieu of an implementation for such a localization routine, it is often convenient to generate foot positions estimate in a *robot-relative* frame, $O_b^{'}$. This frame is not rigidly attached to the robot but moves with the robot in O_0. Additionally, this frame has its own position, relative to O_0, defined by the translation $p_b^{'}$. $O_b^{'}$ is constantly aligned with respect to O_0, which fixed the rotation of $O_b^{'}$ with respect to O_0 as the identity matrix.

In this frame, the trunk position is regarded as an offset from the origin of $O_b^{'}$, $p_{b,rel}^{'}$. Since there is zero rotation between O_0 and $O_b^{'}$, the trunk rotation vector $\theta_{b,re;}^{'}$ is directly equivalent to θ_b in O_0. Using these translation and rotation definitions in place of their world frame counterparts, the robot-relative joint and foot-positions, $p_{i,j}^{'}$ and $p_{i,e}^{'}$ of each i^{th} leg can be computed by defining a transformation from the robot relative frame to the first DH-frame for each leg, as follows:

$$H_{b'}^{i,0} = \left[\begin{array}{c|c} R_{zyx}(\theta_b)R_z(\sigma_i) & R_{zyx}(\theta_b)\nu + p_b^{'} \\ \hline 0 & 1 \end{array}\right] \tag{4.10}$$

which follows the same definitions in (4.1). The transformation defined in (4.10) is used as a direct replacement for the matrix $H_0^{i,0}$ in 4.1.1 and 4.1.1 in computing the corresponding enitities, $p_{i,j}^{'}$ and $p_{i,e}^{'}$. Knowledge of $p_{i,e}^{'}$ and $p_{i,j}^{'}$ is particularly useful for planning foot and body-placements which depend directly on the location other foot positions and the relative body location, but do not necessarily depend on the position of the robot in the world coordinate frame.

4.1.2 Inverse Position Kinematics

A foot configuration is specified by its coordinates p_e and an ankle orientation, γ_i, which represents a rotation about the axis of rotation of the second joint (lateral hip). Given a desired platform configuration, $\{p_b, \theta_b\}$, and desired i^{th} foot configuration, $\{p_{i,e}, \gamma_i\}$, the inverse kinematics solution for each i^{th} leg, q_i, is derived to be:

$$
\begin{aligned}
q_{i,1} &= \cos(i\pi)\left(\frac{\pi}{4} - \psi_i\right) \\
q_{i,2} &= \tan^{-1}\left(\frac{\zeta_{i,z}}{\sqrt{\zeta_{i,x}^2 + \zeta_{i,y}^2}}\right) \mp \cos^{-1}\left(\frac{a_3^2 - a_2^2 - \|\zeta_i\|^2}{2a_2\|\zeta_i\|}\right) \pm \pi \\
q_{i,3} &= \mp\cos^{-1}\frac{\|\zeta\|^2 - a_2^2 - a_3^2}{2a_2a_3} \\
q_{i,4} &= \gamma_i - q_{i,2} - q_{i,3}
\end{aligned} \tag{4.11}
$$

where

$$
\begin{aligned}
p_{i,e}^{i,0} &= E_p\left(H_{i,k}^0\right)^{-1}\begin{bmatrix} p_{i,e} \\ 1 \end{bmatrix} \\
\\
\psi_i &\equiv \tan^{-1}\left((p_{i,e}^{i,0})_2/(p_{i,e}^{i,0})_1\right) \\
\zeta_{i,x} &\equiv (p_{i,e}^{i,0})_2\sin(\psi_i) + (p_{i,e}^{i,0})_1\cos(\psi_i) - a_4\cos(\gamma_i) - a_1 \\
\zeta_{i,y} &\equiv (p_{i,e}^{i,0})_2\cos(\psi_i) - (p_{i,e}^{i,0})_1\sin(\psi_i) \\
\zeta_{i,z} &\equiv (p_{i,e}^{i,0})_3 - a_4\sin(\gamma_i).
\end{aligned} \tag{4.12}
$$

Here, $p_{i,e}^{i,0}$ represent the position of each i^{th} foot with respect to the zeroth DH frame of each i^{th} leg; and $(p_{i,e}^{i,0})_k$ with $k \in \{1,2,3\}$ represents the k^{th} element of $p_{i,e}^{i,0}$.

It is important to note that the ankle specification, γ_i, adds extra constraints on the system kinematics and, thus, reduces the number of inverse kinematics solutions per foot position to two.

4.1.3 Velocity Kinematics

Using the DH-coordinate transformation, $H_{i,4}^{i,0}$, the velocity kinematics of each i^{th} leg are derived as the Jacobian $J_{i,e}^{i,0} \in R^{6\times4}$ where $\dot{x}_{i,e}^{i,0} = J_{i,e}^{i,0}\dot{q}_i$, with $\dot{x}_{i,e}^{i,0} \in R^6$ being stacked vector of translational and rotational velocities, $\dot{p}_{i,e}^{i,0} \in R^3$ and $\dot{\theta}_{i,e}^{i,0} \in R^3$, respectively, of each i^{th} foot with respect to frame $O_{i,0}$. The matrix $J_{i,e}^{i,0}$ is defined explicitly in Appendix [B].

Assuming the translational and rotational velocity of the trunk, $\dot{p}_b$ and $\dot{\theta}_b$, respectively; and the translational and rotational of each i^{th} foot, $\dot{p}_{i,e} \in R^3$ and $\dot{\theta}_{i,e} \in R^3$, respectively, are known, the translation velocity of each i^{th} foot can be written with respect to frame $O_{i,0}$ by:

$$
\dot{p}_{i,e}^{i,0} \equiv \left(R_{i,0}^0\right)^T\left(\dot{p}_{i,e} - \dot{p}_b - S\left(\dot{\theta}_b\right)\left(p_{i,e} - p_b - R_{i,0}^0\vec{o}_\nu\right)\right) \tag{4.13}
$$

where $\vec{o}_\nu = [\nu, 0, 0]^T$, $R^0_{i,0}$ is the rotation-matrix component of the transformation $H^0_{i,0}$ defined in (4.1), and $S(*)$ is the standard skew-symmetric matrix operator, which takes a 3×1 vector input. The corresponding rotational velocity of each i^{th} foot can be computed with respect to $O_{i,0}$ by:

$$\left(R^0_{i,0}\right)^T S\left(\dot{\theta}_{i,e} - \dot{\theta}_b\right) R^0_{i,0} = S\left(\dot{\theta}^{i,0}_{i,e}\right) \tag{4.14}$$

where the rotational velocity vector $\dot{\theta}^{i,0}_{i,e}$ is recovered by:

$$\dot{\theta}^{i,0}_{i,e} \equiv \left[-\left[S\left(\dot{\theta}^{i,0}_{i,e}\right)\right]_{1,2}, \left[S\left(\dot{\theta}^{i,0}_{i,e}\right)\right]_{1,3}, -\left[S\left(\dot{\theta}^{i,0}_{i,e}\right)\right]_{2,3}\right]^T. \tag{4.15}$$

Joint velocities required to attain $\dot{x}^{i,0}_{i,e}$ can be computed using $J^{i,0}_{i,e}$. However, since each of BlueFoot's of legs had 4 degrees of freedom, $J^{i,0}_{i,e}$ is rank deficient and $\dot{q}_i$ cannot be obtained by direct inversion. Instead, $\dot{q}_i$ can be obtained from $\dot{x}^{i,0}_{i,e}$ by the psuedo-inverse of

4.2 Dynamical Model

4.2.1 System State Vector and General-Form Dynamics

To fully defined the state of the BlueFoot platform, we consider a general, free-floating, four legged robotic system with four degrees of freedom per leg. This system is fully described by the state vector $z \equiv [\eta^T, \dot{\eta}^T]^T \in R^{44}$ and its dynamics are:

$$M(\eta)\ddot{\eta} + C(\eta, \dot{\eta})\dot{\eta} + G(\eta) + \Delta H = \tau + J^T(\eta) f_{ext} \tag{4.16}$$

where $M(\eta)$, $C(\eta, \dot{\eta})$, $G(\eta)$ and $J(\eta)$ represent the system mass matrix, Coriolis matrix, gravity matrix and Jacobian, respectively [11]. ΔH has been included as a lump term to account for dynamical uncertainties, such as friction or unmodeled coupling effects. Additionally, $f_{ext} = [f^T_{1,ext}, f^T_{2,ext}, f^T_{3,ext}, f^T_{4,ext}]^T \in R^{24}$ represents a stacked vector of force-wrenches, $f_{i,ext} \in R^6$, applied to the system through each i^{th} foot. The state vector, η, can be partitioned as follows: $\eta = [p^T_b, \theta^T_b, q^T]^T$ with $p_b \in R^3$ and $\theta_b \in R^3$ representing the position and orientation, respectively, of the quadruped's trunk in an arbitrarily placed world coordinate frame, and $q \in R^{16}$ is a vector of joint variables, m of which are contributed by each leg. $\tau \in R^{22}$ represents a vector of generalized torque inputs and takes the form $\tau = [0_{1x6}, \tau^T_q]^T$ where τ_q represents a set of torque inputs to each joint. It is important to note that the states we are most interested in controlling, p_b and θ_b, are not directly actuated, and must be controlled via composite leg joint motions.

BlueFoot's dynamics, from (4.16), can be realized in a general, compact, state-space form by:

$$\begin{aligned} \dot{z}_1 &= z_2 \\ \dot{z}_2 &= M^{-1}(z_1)(\tau + \Phi(z_1, z_2, f_{ext})) \\ \Phi(z_1, z_2, f_{ext}) &= J^T(z_1) f_{ext} - C(z_1, z_2) z_2 - G(z_1) - \Delta H \end{aligned} \tag{4.17}$$

where $z_1 = \eta$ and $z_2 = \dot{\eta}$. The notation $\Phi(z_1, z_2, f_{ext})$ is introduced for convenience as a composite dynamical term. This term will be referred to, simply, as Φ in the sections that follow. The system dynamics are also considered in an approximate, discrete-time (first-order) form as follows:

$$\begin{aligned} z_{1,k+1} &= z_{1,k} + (e_{1,k}^{\Delta_s} + z_{2,k})\Delta_s \\ z_{2,k+1} &= z_{2,k} + M_{1,k}^{-1}(e_{2,k}^{\Delta_s} + \tau_k + \Phi_k)\Delta_s \\ t &= \Delta_s k \end{aligned} \tag{4.18}$$

where $M_{1,k} = M(z_{1,k})$ and $\Delta_s \equiv (f_s)^{-1}$ with f_s defining a uniform sampling frequency in Hz. The terms $e_{1,k}^{\Delta_s}$ and $e_{2,k}^{\Delta_s}$ are used to explicitly account for system discretization errors, which vary with respect to the step-size, Δ_s.

4.2.2 Joint-Servo Dynamics

The motor dynamics driving each joint need to be considered for use in control design since the input to BlueFoot's servo motors at each joint is a reference position command. In model-based control schemes to follow, a simple model of the motor dynamics will be utilized. Moreover, servos are considered as simple torque generators of the following form:

$$\tau_q = k_s(q^r - q) \tag{4.19}$$

where $k_s > 0$ is a constant, scalar gain and q^r is a joint position reference. The servos we are utilizing to drive the leg joints of the BlueFoot quadruped have high-gain position feedback which allows us to model the motors, simply, as a static block which transform reference trajectories to torque outputs. All of these servos are identical, and thus have identical gains. One could instead consider the full motor dynamics for computing reference positions given a desired torque. The simple model stated above was adequate for achieving desired results with all proposed control schemes which use this torque-generator model.

4.2.3 Single Leg Dynamics

The dynamics of each independent leg (with each base-frame fixed) can be derived in closed form. These dynamics have been included in Appending [B] using the previously defined notations.

4.3 BlueFoot Simulator

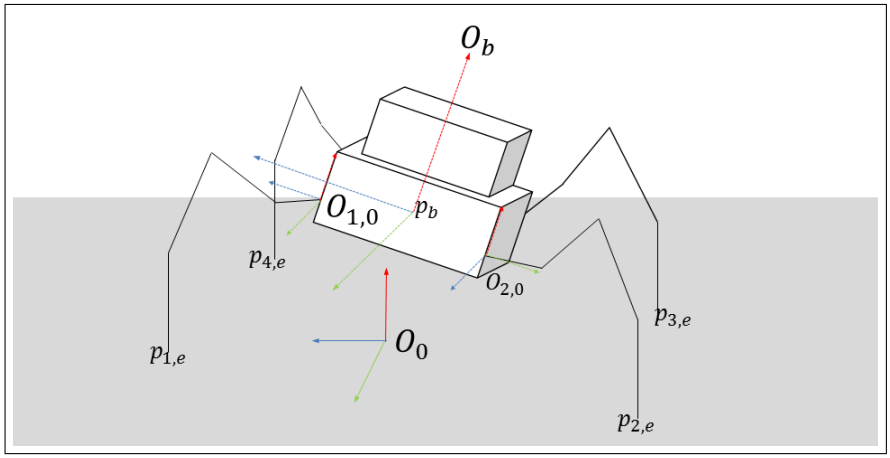

Figure 13: Coordinate frame setup.

The BlueFoot platform is comprised of 17 rigid bodies, 16 of which are linkages between joints; and a main platform. Since each limb is formed from four revolute joints, the system has a total of 22 DOF, 16 of which are actuated. The main platform's configuration is generated through particular configurations of the legs. Furthermore, because each linkage is constrained to a rotation about a single axis, the rigid-body dynamic model of the BlueFoot quadruped is represented by 44 states. These states represent the position and velocity of each joint, q_i and $\dot{q}_i$, respectively. Additionally, foot contact states are represented by binary scalars, $\mu_i \in (0, 1)$, which describes whether the foot is not in contact or in contact, respectively.

The dynamics of this system are somewhat complex because the quadruped's legs continuously make and break contact with the ground during gaiting. Additionally, the surface attributes and contact effects may vary significantly during gaiting and for various environments. A numerical dynamics engine, Open Dynamics Engine (ODE) [12], has been utilized to implement a dynamics simulator for the BlueFoot platform. The simulator allows for various reconfigurations of environmental parameters, such as contact friction and physical obstacles.

The simulator's numerical engine, based on ODE is updated at 1000 Hz to attain high-fidelity, especially during contact phases.The input to the simulator is the desired main body configuration (i.e., position and orientation), from which an inverse kinematic model is solved to attain all the desired joint configurations for the legs. Servos at each joint have built-in, tunable proportional controllers that effectively render them as ideal torque generators responding to the command input. To mimic the behavior of the system, the commands to each joint motor are updated at 50 Hz, which matches the update rates of the P-controller of the actual robot. This rate has been chosen to account for the speed of inter-processor communications, and is adequate given the operating bandwidth of the robot.

The BlueFoot simulator utilizes a dynamical model of the platform which consists of purely rectangular bodies. To match the dynamics of the true system, the individual mass and inertia parameters of each rigid body have been generated using a 3D model analysis software. This software takes into account geometric irregularities and variation of materials when computing the inertia of each body. The simulator also contains IMU and LIDAR sensor models for gathering feedback from the simulated environment. Each sensor is modeled with appropriate measurement noise so as to more closely match the performances of the actual IMU and LIDAR sensors on-board the BlueFoot platform.

Careful, manual tuning was performed on simulated robot parameters, including joint update gains, joint-friction and surface-friction model coefficients to achieve closer matches between simulated results and the actual robot motions. A performance comparison can be seen in [NEED FIG], which shows joint position outputs for all joints of the front-right leg of the actual and simulated robot while performing a forward gait at 0.080 m/s. It can be seen in [NEED COMPARISON FIGURE] that the simulator matches true system performance for most of each joint trajectory. The discrepancies between the two data sets occur at the minima of each series when the robot makes contact with the ground after a step. These differences can be attributed to ground-contact model inaccuracies. Work is still left to be done in improving the simulator's contact model, but this is more the fault of the simulation software libraries being used, namely in its low-order contact models. These contact models are very simplistic for the sake of simulation speed. Contact models could certainly be improved such that they more closely matches the effects of real-world surfaces. All of the other simulated leg joints exhibited similar results to the actual motions.

CHAPTER V

Gaiting and Gait-Stability Control

5.1 Overview

Legged robotic systems have long employed motion controllers based on limit cycle oscillators and, more recently, Central Pattern Generators (CPGs) for the purpose of generating bio-mimetic gaits [9, 13–20]. Since these motion control methods are open-loop motion planners (*i.e.*, not inherently formulated to incorporate feedback) they do not perform any active syste stabilization on their own accord. As a result, implementations involving these locomotion methods often require auxiliary control mechanisms which provide gait stability. Fixed point methods, which include considerations of the system's zero-moment point (ZMP) and center of gravity (COG), are utilized in the design of stable oscillator driven gaits. These methods are summarized in [21].

Developments in CPG-based gait controllers have led to the incorporation of "reflexive" feedback mechanisms aimed at correcting foot-placement during gaiting on uneven terrain or various surfaces. One such approach involves active compliance to each leg by directly modifying CPG oscillators units through feedback-driven modulations. In [4] and [15], a CPG for each leg is modified by a neural oscillator with one tuning parameter.

This chapter will detail a similar, reflex-adaptive CPG gait generation method which utilizes IMU feedback to modify CPG limit-cycle parameters. Additionally, this section will present related motion-control methods implemented on the BlueFoot platform which aid in gait stabilization, including a virtual force-based foot placement planner and a ZMP-based body placement controller; as well as a nuero-control technique used to level BlueFoot's trunk.

5.2 Central Pattern Generator (CPG) Based Gaiting

A central pattern generator (CPG), which includes a reflexive mechanism using sensory feedback, is utilized as the core of the robot's gait generation algorithm. CPGs are a form of neural oscillator network which mimic biological mechanisms for repetitive mo-

tor tasks [17], [14]. CPGs commonly consist of a network of multi-state unit-oscillators. These unit-oscillators are coupled such that the motion of one oscillator drives or attenuates the motion of other oscillators it is connected to, creating phase-locked limit cycles.

In this context, the limit-cycles generated with a CPG will be utilized to drive a quadruped walking gait by mapping oscillator outputs to foot position controls. CPGs are widely used in this way as they provide a compact method for prescribing rhythmic gaits with variable stepping sequences [**?**, 16, 22]. CPG-driven gait controllers are convenient as they allow for continuous transitioning between gaiting patterns through the modification of oscillator coupling. Reflexes are built into the oscillators which use IMU measurements to modulate the CPGs unit oscillators, as will be outlined later in this section.

The CPG implemented on BlueFoot consists of four modified two-state $(y_{1,i}, y_{2,i})$ Hopf Oscillators connected through a coupling matrix K. The oscilator output vector, $y_{2,i}$, parameterize the trajectories of each i^{th} foot. The resulting reflexive CPG system is described by

$$\begin{aligned} \dot{y}_1 &= A_1 \left(\Psi_M M - \Gamma\right) y_1 + \Psi_\omega W y_2 \\ \dot{y}_2 &= A_2 \left(\Psi_M M - \Gamma\right) y_2 - \Psi_\omega W y_1 + K y_2 \end{aligned} \tag{5.1}$$

where M is a diagonal matrix in $R^{4\times4}$ with diagonal elements $m_{i,i} = y_{1,i}^2 + y_{2,i}^2 \quad \forall i = \{1, 2, 3, 4\}$, $(A_1, A_2, \Psi_M, \Psi_\omega) \in R^{4\times4}$, $\{y_1, y_2\} \in R^{4\times1}$, and Γ and W are diagonal matrices in $R^{4\times4}$ which represent nominal oscillator amplitudes and frequencies, respectively.

5.2.1 Reflexive Gait Adaptations

Feedback signals are incorporated into the CPG through the frequency modulation matrix Ψ_ω and amplitude modulation matrix Ψ_M. These modulation parameters are generated using state estimates of the platform's orientation and angular rate. The platform orientation state-estimate, $\hat{\theta}_b$, is supplied to the controller from an Extended Kalman Filter utilizing inertial measurement and magnetometer feedback (which are separately calibrated). The platform's angular rate is output by an angular rate-gyro sensor.

Feedback-based corrections to the CPG aid in tracking a specified body orientation θ_b^r during gaiting. To achieve higher system velocities, it is often necessary to perform a gait wherein multiple legs leave contact with the ground. Configurations such as these

would cause a quadruped robot to tip in the direction of the non-planted legs, thus disturbing θ_b. These disturbances are counteracted, in part, by adjusting the CPG such that the amount of torque applied on the main body by legs in flight is actively limited. This is done by adjusting stepping height and time-of-flight according to a measure of disturbance (essentially, the amount and rate of tipping). These adjustments have been formulated to mimic reflexive behaviors that might be performed by a biological system.

When the robot's body begins to fall in particular direction (*i.e.*, is disturbed by legs in-flight during gaiting), the disturbance signal $\dot{\epsilon}_\theta = R_{z_b}\left(\frac{\pi}{2}\right)\left(\dot{\hat{\theta}}_b - \dot{\theta}_b^r\right)$ is non-zero. $\dot{\epsilon}_\theta$ represents a measure of translational drift recovered from gyroscopic sensor measurements. $R_{z_b}\frac{\pi}{2}$ represents a rotation by $\frac{\pi}{2}$ about the z-axis of the body frame O_b. $\dot{\epsilon}_\theta$ is mapped to the parameters Ψ_ω and Ψ_M by

$$\begin{aligned} \psi_i &= \text{sig}\left(w_i T_i - w_i c_i\right)\mu_i \\ \Psi_\omega &= I + A_\omega \text{diag}\left(\psi_i\right) \\ \Psi_M &= I - A_\mu \text{diag}\left(\psi_i\right) \end{aligned} \tag{5.2}$$

where

$$\begin{aligned} T_i &= \|\dot{\epsilon}_\theta\|\left(1 + \frac{\Delta x_i}{\|\Delta x_i\|}^T \frac{\dot{\epsilon}_\theta}{\|\dot{\epsilon}_\theta\|}\right) \\ \Delta x_i &= p_e - p_b \end{aligned} \tag{5.3}$$

with $\{w_i, c_i\} \in R$ and $\{A_\mu, A_\omega\} \in R^{4\times 4}$. $\text{sig}\left(*\right)$ represents the standard sigmoid step function. The signal T_i represents a projection of $\dot{\epsilon}_\theta$ into the unit-vector emanating from p_b to each i^{th} foot. This projection delegates the level of adjustment to each i^{th} oscillator as a result of $\dot{\epsilon}_\theta$.

Adjusting Ψ_ω by the method delineated in (**??**) serves to shorten the stepping period of a leg in-flight given greater values of $\dot{\epsilon}_\theta$. Likewise, Ψ_M is adjusted to decrease the height of each foot in-flight.

The frequency of a full gaiting cycle is prescribed via ω_s and duty-factor $\alpha \in [0, 1]$ [18]. The matrix W is formed from ω_s and α as a diagonal matrix with diagonal elements

$$w_{ii} = \frac{\alpha\omega_s}{1 + e^{+\zeta y_{2,i}}} + \frac{(1-\alpha)\omega_s}{1 + e^{-\zeta y_{2,i}}}, \;\; i = 1, ..., 4 \tag{5.4}$$

and ζ being a sensitivity tuning parameter. A linear mapping between commanded platform velocity, v_c, and ω_s is prescribed such that stepping-cycle frequency is adjusted proportionally with respect to the desired velocity of the system, *i.e.*, $\omega_s = a_v v_c$.

In BlueFoot's CPG implementation, the coupling matrix K takes on the values $k_{i,j} \in [-1, 1]$. Each element of the coupling matrix, $k_{i,j}$, is utilized to adjust the phase offsets between the unit-oscillators.Furthermore, setting $k_{i,j} = 1$ causes the j^{th} oscillator to attract the i^{th} oscillator towards a positive peak, and vice-versa. Setting $k_{i,j} = 0$ effectively disconnects i^{th} and j^{th} oscillators.

Figures **??** and **??** correspond to the output of the oscillator dynamics, $y_{2,i}$, given by (**??**) with the feedback mechanism given in (**??**) used during simulations of BlueFoot's default two-pace trotting gait. The associated K matrix prescribing this gaiting pattern is a modified version of K for a 2-pace trot gait presented in [10], since this generates a more effective and fluid gait when applied to BlueFoot's gait controller

$$K \equiv \begin{bmatrix} 0 & -1 & 1 & -0.5 \\ -1 & 0 & -0.5 & 1 \\ -1 & -0.5 & 0 & -1 \\ -0.5 & 1 & -1 & 0 \end{bmatrix}. \tag{5.5}$$

A comparable four-pace gait can be achieved by a slight modification of (5.5), which yields (5.6) as follows:

$$K \equiv \begin{bmatrix} 0 & -1 & 1 & -0.5 \\ -1 & 0 & -0.5 & 1 \\ -1 & 0.5 & 0 & -1 \\ 0.5 & -1 & -1 & 0 \end{bmatrix}. \tag{5.6}$$

The output state of each i^{th} oscillator is shown in **??** as a function of time, whereas **??** depicts a phase portrait for each i^{th} oscillator.

5.3 Foot Placement Control

A foot placement controller has been implemented which utilizes CPG outputs (defined in equation **??**) in concert with a virtual-force controller. This controller is designed such that each foothold of the robot tracks the foot positions of a virtual robot generated through a model reference. The virtual robot is described by the foot positions, $\tilde{p}_{i,e}$, a virtual reference configuration corresponding to the position, $\tilde{p}_c$, and orientation, $\tilde{\theta}_c$, of the main body in O_0. The robot follows the foot placement of the virtual robot to achieve a commanded ground velocity $v_c \in R^3$; and turning velocity

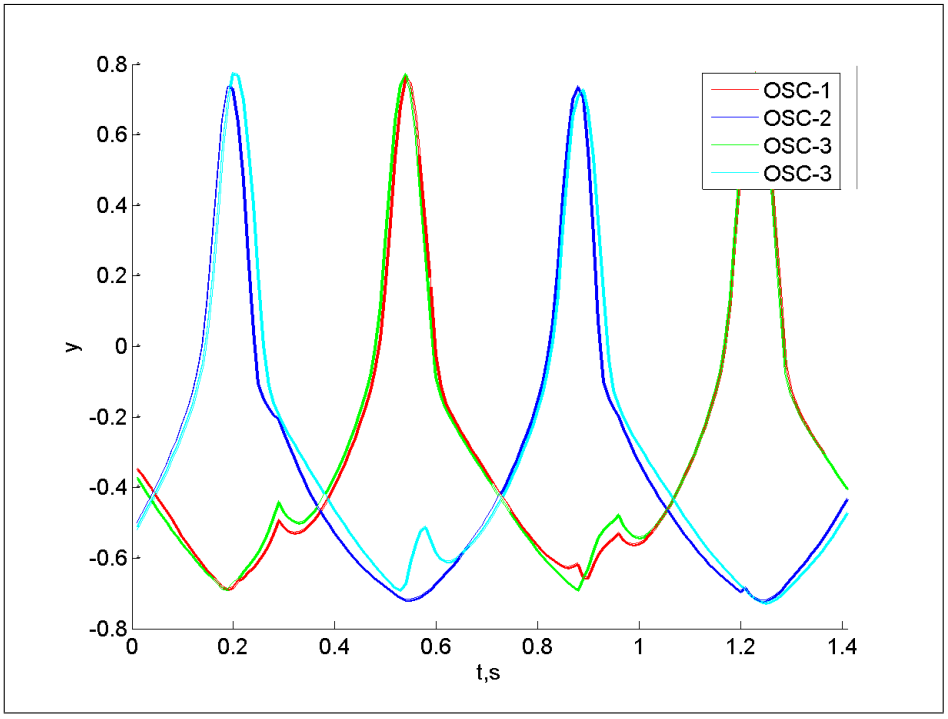

Figure 14: CPG output state, y_2, over two gait cycles.

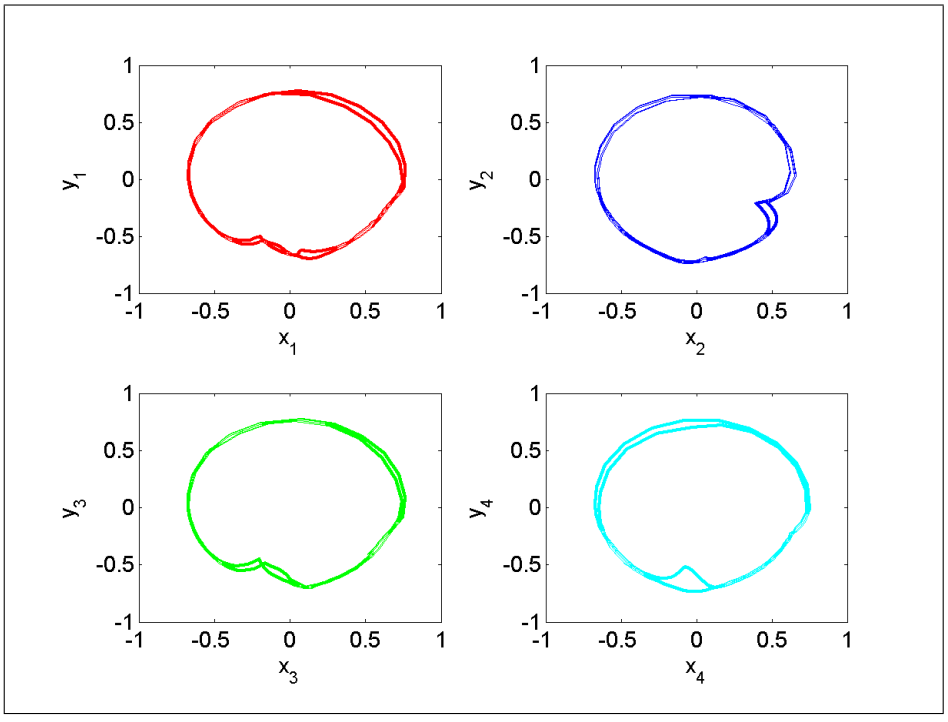

Figure 15: CPG state phase portrait over two gait cycles; depicts the effects of feedback-modulation on the CPG's limit cycles.

$\omega_c \in R$. Each virtual point is updated by:

$$\begin{aligned} \dot{\tilde{\theta}}_c &= \omega_c \\ \dot{\tilde{p}}_c &= v_c \\ \dot{\tilde{p}}_{i,e} &= v_c + S\left(\omega_c \vec{h}_P\right) \tilde{R}_P \left(\tilde{p}_{i,e} - \tilde{p}_c\right) \end{aligned} \tag{5.7}$$

where $\vec{h}_P$ is a unit vector that is orthogonal and pointed outward from the surface beneath the robot and $S\left(\omega_c \vec{h}_P\right) \in R^{3\times 3}$ forms a skew-symmetric matrix from the vector argument $\omega_c \vec{h}_P$. The virtual foothold dynamics progress the target footholds at a commanded translational (forward) and rotational velocity, v_c and ω_c, respectively. The robot follows the virtual model at nearly the same velocities with minimal lag so long as system bandwidth is not exceeded. Using this virtual-foothold method is convenient as it allows for foot-placement planning to be independent of foot-trajectory planning. For example, terrain adaptation can be incorporated by modifying the location of virtual foot positions such that they conform to an upcoming surface. The robot's gait will then track these adaptations without any explicit modification of foot trajectories. In the event of contact with unperceived terrain, virtual-foothold positions will reset with respect to the position of the contact. The full foothold controller represented by $\dot{p}_{i,e} =$

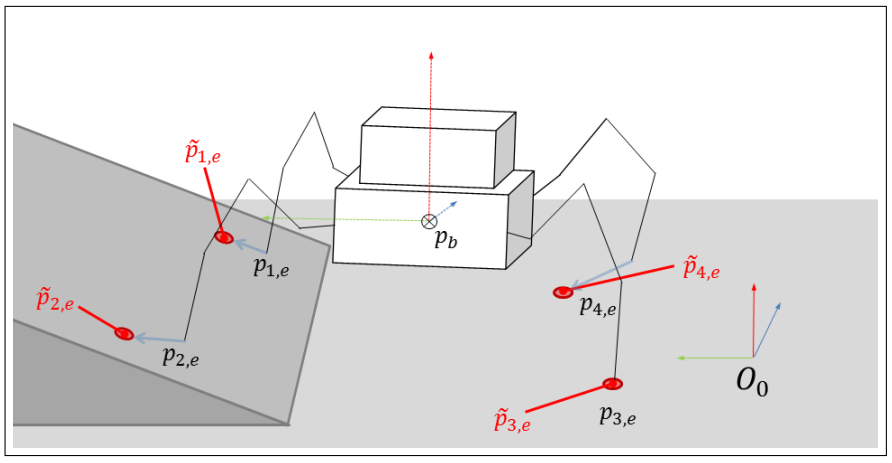

Figure 16: Virtual foothold representaion. Blue arrows represent an attractive "force" between the feet and their corresponding virtual foothold.

$\dot{\tilde{p}}^v_{i,e} + \dot{\tilde{p}}^y_i$, $i = 1, ..., 4$, is a composite of a foot-repositioning and step-height controller which are formulated in terms of the dynamics of the signals $\dot{\tilde{p}}^v_{i,e}$ and $\dot{\tilde{p}}^y_i$, respectively. Each foot is treated as a point mass attracted to their corresponding virtual foothold

by an over-damped mass-attractor system. $\dot{\tilde{p}}^{v}_{i,e}$ is defined by

$$\dot{\tilde{p}}^{v}_{i,e} = \frac{1}{m_i} \int^{t} P_{\vec{h}_i} \left(F_{s,i} - b_c \dot{p}_{i,e} + F_{\epsilon,i} \right) d\tau \tag{5.8}$$

where

$$\begin{aligned} F_{s,i} &= k_c \mu_i \left(\tilde{p}_i - p_{i,e} \right) U(y_{2,i}) \\ F_{\epsilon,i} &= a_\epsilon \dot{\epsilon}_\theta \, U(y_{2,i}) \end{aligned}$$

with $k_c, b_c, a_\epsilon, m_i \in R$. k_c and b_c represent attraction and viscous damping constants for the virtual force system. $P_{\vec{h}_i}(*)$ projects the sum of forces, $(*)$, onto the surface beneath the i^{th} foot orthogonal to $\vec{h}_i$. $U(y_{2,i})$ is a standard unit step function. The force $F_{\epsilon,i}$ is a compensatory force to adjust for the inertial disturbance, $\dot{\epsilon}_\theta$. For instance, if the robot was pushed from the side of its body, this force would induce a side stepping motion in attempt to compensate for the disturbance. Because of $U(y_{2,i})$, $F_{s,i}$ and $F_{\epsilon,i}$ are nonzero only when $y_{2,i}$ is positive.

The controller component $\dot{\tilde{p}}^{y}_{i,e}$ is defined from the CPG output signal $\dot{y}_{2,i}$. The height of each step is made proportional to the magnitude of desired platform velocity.

$$\dot{\tilde{p}}^{y}_{i,e} = \left(\| v_c \| \, a_v \dot{y}_{2,i} \right) \vec{h}_i \;\; ; \;\; a_v \in R. \tag{5.9}$$

Scaling step height relative to v_c has been seen in experimental studies to be more effective than using a fixed step-height for all gait configurations.

5.4 ZMP Based Trunk-Placement Control

A modified Zero-Moment-Point (ZMP) based controller is utilized in positioning BlueFoot's body during gaiting. In this context, the ZMP of the system is a set of state values for which the net-torque exerted upon the system, about the COG, is zero [23], [24]. Unlike [25, 26], which address ZMP-based control by considering the robot's body as point-mass with massless limbs, this method takes into account the torque contribution of each non-supporting leg. Each leg in flight is considered as a series of point masses whose locations are computed using joint position feedback and trunk orientation estimates, in concert with robot's forward kinematic model.

The ZMP approach is used to first calculate static ZMP configurations (*i.e.*, $\ddot{p}_{COG} \approx 0$) at each time instance. The distance between the robot's COG and associated ZMP is incrementally minimized on-line by treating the ZMP as a mass-attractor and "pulling"

a reference body position, p_b^r, towards the ZMP point at each update. This differs from approaches similar to [26] because these routines feature off-line, trunk trajectory design which aim to minimize deviations between the robot's COG and ZMP by creating an appropriate limit-cycle motion for the robot's trunk. These approaches typically utilize simplified, linearized models of the actual system about marginally-stable equilibrium points. To realize an adjustment of the robot's COG, towards the ZMP, BlueFoot's trunk position is modified during the execution of a gait. The trunk is controlled (and not individual feet) as it contributes most of the system's total mass and, thus, has the greatest influence over the location of the platform's COG. Restricting adjustments to the trunks translational states allow pre-planned foot trajectories to remain unmodified by the ZMP-control module during gait execution, thus decoupling foot-placement and body-placement control.

To incrementally compute the static ZMP of the platform, a measure of net-moment on the body of the platform must be known. The net-moment due to gravitational forces, τ_{net}, is approximated using the locations of each link and joints as point-mass loads. Hence, τ_{net} is calculated as follows:

$$\tau_{net} = \vec{g} m_P \left(\bar{p}_b - \hat{p}_{COG}\right) + \tau_{legs} \tag{5.10}$$

where

$$\begin{aligned}
\tau_{legs} &= \sum_{i=1}^{4}(1-\mu_i)\sum_{i=1}^{4}\vec{g}\times\left(m_{i,j}^J d_{i,j}^J + m_{i,j}^L d_{i,j}^L\right) \\
d_{i,j}^J &= p_{i,j} - \hat{p}_{COG}. \\
d_{i,j}^L &= \begin{cases} 0.5\left(p_{i,j+1} - p_{i,j}\right) + p_{i,j} - \hat{p}_{COG} & \text{if } j < 4 \\ 0.5\left(p_{i,e} - p_{i,j}\right) + p_{i,j} - \hat{p}_{COG} & \text{if } j = 4 \end{cases}
\end{aligned}$$

with m_P, $m_{i,j}^J$ and $m_{i,j}^L$ represent the mass of the main body; the mass of each joint; and the mass of each link, respectively. μ_i is included in the above formulation such that only stepping legs contribute to τ_{legs}. The estimated center of gravity, $\hat{p}_{COG}$, is generated as follows:

$$\begin{aligned}
\hat{p}_{COG} &= \frac{1}{m_T}\left(m_b p_b + \sum_{i=1}^{4}\left(m_{i,e}p_{i,e} + \sum_{j=1}^{4}\left(m_{i,j}^J p_{i,j} + m_{i,j}^L p_{i,j}^L\right)\right)\right) \\
p_{i,j}^L &= \begin{cases} 0.5\left(p_{i,j+1} - p_{i,j}\right) + p_{i,j} & \text{if } j < 4 \\ 0.5\left(p_{i,e} - p_{i,j}\right) + p_{i,j} & \text{if } j = 4 \end{cases}
\end{aligned}$$

where

$$m_T = m_b + \sum_{i=1}^{4} \left(m_{i,e} + \sum_{j=1}^{4} \left(m_{i,j}^J + m_{i,j}^L \right) \right) \tag{5.11}$$

Setting $\tau_{net} = 0$, (**??**) is manipulated to a derived solution for the ZMP as follows:

$$p_{ZMP} = R_{z_P} \left(\frac{\pi}{2} \right) \left(\|g\| / m_P \right) \tau_{legs} + \hat{p}_{COG}. \tag{5.12}$$

Using p_{ZMP}, the platform's posture is then updated using a virtual-force controller. Moreover, p_b is to be controlled through p_b^r such that it is smoothly attracted to p_{ZMP}. The controller is given by

$$\dot{p}_b^r = \frac{1}{m_P} \int^t P_{\vec{h}_i} \left(K_{ZMP}(p_{ZMP} - p_b) + F_r \right) d\alpha \tag{5.13}$$

where

$$\begin{aligned} F_r &= \sum_{i=1}^{4} \exp\left(k_l \left(r_{min} - \|r_i\| \right) \right) + \exp\left(k_l \left(\|r_i\| - r_{max} \right) \right) \\ r_i &= p_{i,e} - p_b \end{aligned}$$

and $K_{ZMP}, K_c, K_\epsilon, k_l > 0$ and $\dot{p}_b^r$ is the reference body velocity. F_r is a boundary force added to ensure that the workspace of each manipulator, defined by the radii r_{min} and r_{max}, is not exceeded when the body is repositioned. k_l is picked to be adequately large such that this force is nearly zero when the body and foot positions comply with the local workspace of each leg, and large when the workspace is nearly compromised, forcing the placement of the body to comply with each leg workspace.

5.5 Trunk Leveling NARX-Network Learning Approach

Disturbance rejection from the trunk sub-system of a legged platform has practical significance when carrying a payload (such as cameras, optical systems, armaments, etc.) rigidly fixed to their main body. Disturbances are imparted upon the trunk during gaiting in two main ways:

1. instantaneous changes in force distribution when feet make and break contact with the ground

2. under-actuation that occurs during certain dynamic gaits. During dynamic gaits, such as trot gaits, the state of contact between the feet and the ground is changed often so as to prevent the walking robot from tipping past a recoverable configuration.

Additionally, these gaits feature the utilization of two or fewer legs to support the trunk at any given time, causing the system to enter an under-actuated state where the body is free to rock about the planted feet, as shown in Figure 17.

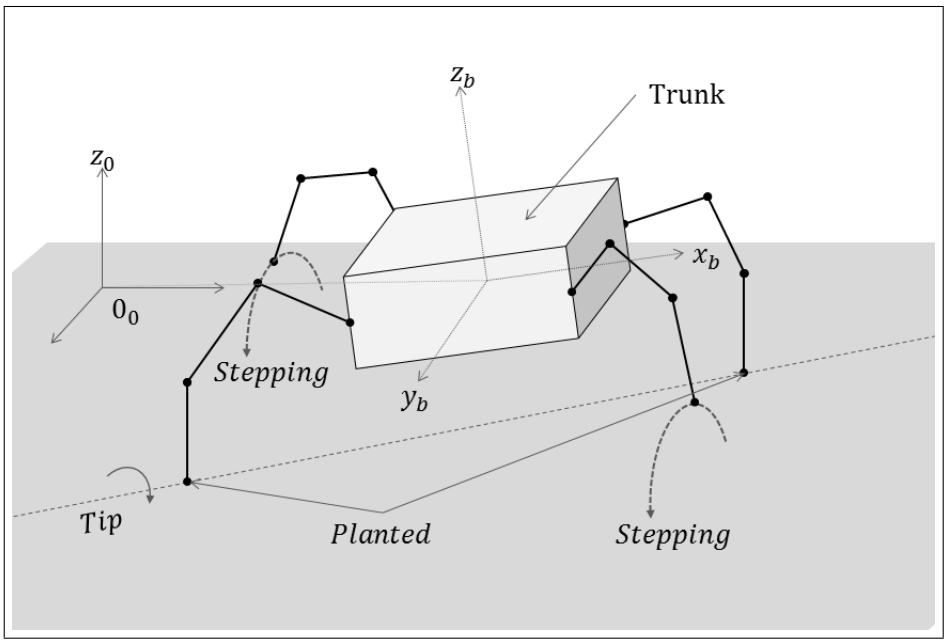

Figure 17: Quadruped tipping about planted feet.

To achieve disturbance rejection on the trunk orientation and to attain a fixed orientation, experimentation has been performed using a control methodology which utilizes a Nonlinear Autoregressive Neural Network with Exogenous inputs (NARX-NN) as part of an active compensation mechanism. The network is used to estimate the system dynamics and, further, predict periodic disturbances in an on-line fashion. The compensator is utilized to modify referential joint trajectories by way of a weighted sum between the original joint trajectories generated by the gaiting mechanism and a reference correction signal generated by the compensator.

5.5.1 NARX-Neural Network

A NARX-NN architecture is used in this controller because of its known effectiveness in approximating nonlinear difference systems and making multivariate time-series predictions [27–30]. Moreover, the NARX-NNs is a natural fit for a problem of this nature where the dynamics being considered are both periodic and of a high enough complexity where a nonlinear approximation method is warranted. The parallel NARX-NN model, shown in Figure 18, is comprised of a feed-forward neural network whose

input layer accepts a series of time-delayed system state values and network-output histories. The NARX-NN is trained to predict system states in the next time-instant from these inputs. Conveniently, NARX-NN training can be performed using standard BP because recurrence occurs between network inp uts and outputs, and not within the hidden layers [31].

The NARX-NN is trained to capture the effects of forces and moments and dynamical couplings that act on the trunk so that an appropriate torque input to the joints is computed to reduce such effects on trunk orientation while performing the gate. This is achieved by considering the inverse dynamics corresponding to joint motion.

Disturbances imparted upon the trunk during gaiting manifest in the term Φ, largely as a result of variations in f_{ext} and associated effects due to dynamical coupling. Because of this, the NARX-NN will learn an estimate for Φ, denoted $\hat{\Phi}$. The network is trained on-line using the standard incremental back-propagation (BP) algorithm with an adapted learning rate, γ^{lr} and momentum term, μ [32, 33]. This error BP algorithm is a gradient-descent based method used to train a feed-foward nueral network with n-layers and layer-connection matrices $\{W^1, W^2, ..., W^{n-1}\} \in W$. The BP algorithm, as used in this control approach, is summarized in matrix-vector form in (adapted from [**?**]) as follows:

$$\Delta W^i = -\gamma^{lr}\left(\frac{\partial o^i}{\partial W^i} o^{i-1}\right)^T + \mu \Delta W^i = -\gamma^{lr}\delta^i \left(o^{i-1}\right)^T + \mu \Delta W^i \tag{5.14}$$

where

$$\delta^i = \left(\nabla_y \sigma^i \left(y^i\right)\right) e^i$$

$$y^i = W^i o^{i-1}$$

$$e^i = \left(W^i\right)^T \delta^{i+1} \ \forall \ i \neq n,$$

$\gamma^{lr} \in [0, 1]$, the learning and $\mu \in [0, 1]$, the learning momentum; $W^i \in \Re^{N_O^i \times N_I^i}$, which represents the weighting matrix between the i^{th} layer (of size N_I^i nodes) and $(i+1)^{th}$ layer (of size $N_O^i = N_I^{i-1}$); ΔW^i, which represents the corresponding weight update to W^i; and e^i is the ouput error for each i^{th} layer. For the output (n^{th}) layer, e^n is equal to the difference between the network output and the network output target, which will be defined later. For all other layers, e^i represents a *back-propogated* error. from the $(i+1)^{th}$ layer.

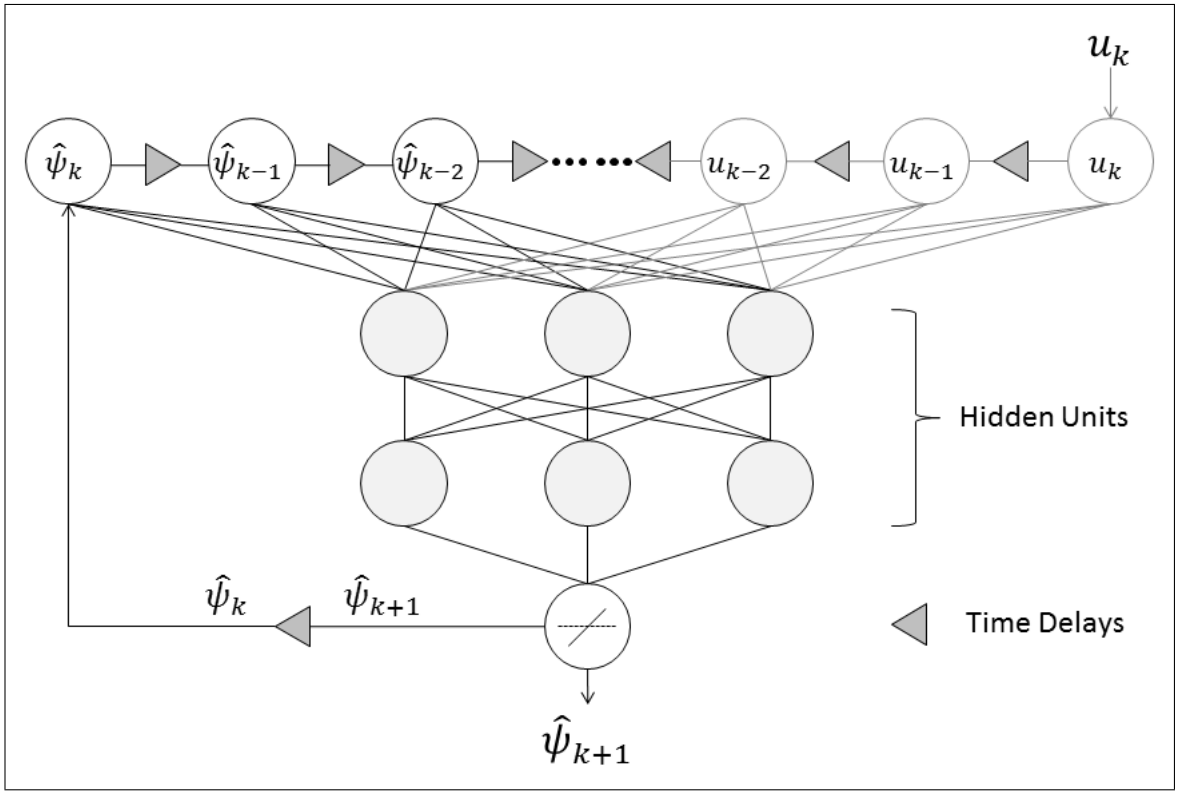

Figure 18: Parallel NARX-network model with a linear output layer.

$\sigma^i(y^i)$ is a layer-wise activation function which outputs a vector of scalar activation outputs, $\sigma^i_j(y^i_j)$ for each j^{th}, weighted input, y^i_j, defined as follows:

$$\left\{\sigma^i(y^i) = \left[\sigma^i_1(y^i_1), ..., \sigma^i_{N^i_I}(y^i_{N^i_I})\right]^T \;\; : \;\; \Re^{N^i_I} \rightarrow \Re^{N^i_I}\right\}$$

For the control approach being described, each $\sigma^i_j(y^i_j) \equiv \tanh(y^i_j) \in [-1, 1]$. Hence the gradient $\nabla_y \sigma^i(y^i)$ is defined as follows:

$$\nabla_y \sigma^i(y^i) = \text{diag}\left(\sigma^i(y^i)\right)\left(\vec{1}_{N^i_I \times 1} - \sigma^i\left(y^i\right)\right) \tag{5.15}$$

given the derivative properties of the $\tanh(*)$ function.

The success of this learning mechanism, as it applies to the presented controller, is predicated on the periodicity of the system dynamics during gaiting. Like any BP-trained neural network, repetition of similar input and output sets is paramount for successful network training and, by extension, prediction accuracy. It is assumed that this specification can be met given the inherently cyclic nature of the dynamics being estimated during gaited locomotion. Using the NARX-NN, the dynamical estimate $\hat{\Phi}_k$ is generated as a prediction of the sampled system dynamics, $\hat{\psi}_{k+1}$. The relationship

between $\hat{\Phi}_k$ and the network prediction $\hat{\psi}_{k+1}$will be made clear in the description of the network training signal given in (5.17). The general input-output relationship of the NARX-NN predictor, N, is described as follows:

$$\begin{aligned} \hat{\psi}_{k+1} &= N(\hat{\Psi}_k^N, U_k^N) \\ \hat{\Psi}_k^N &= [\hat{\psi}_k, \hat{\psi}_{k-1}, ..., \hat{\psi}_{k-N+1}] \\ U_k^N &= [u_k, u_{k-1}, ..., u_{k-N+1}] \end{aligned} \tag{5.16}$$

where U_k^N and $\hat{\Psi}_k^N$ are collections of N most recent samples of the network inputs, u_k, and the network output, $\hat{\psi}_k$, respectively. The NARX-NN input, u_k, represents a tuple $u_k = (z_{1,k}, z_{2,k}, f_{ext,k})$ whose components are the arguments of Φ at time instant k.

5.5.2 NARX-NN Training Regimen

The NARX-NN training signal is formulated to estimate Φ_k from the system dynamics. By (4.18), it can be seen that Φ_k can be estimated if $z_{2,k+1}$ can be predicted. We consider the following target network prediction output, ψ_{k+1}, defined by:

$$\psi_{k+1} = \tau_k - \hat{M}_{1,k}(z_{2,k+1} - z_{2,k})\Delta_s^{-1} = \Phi_k - e_{2,k}^{\Delta_s} \ . \tag{5.17}$$

This training signal formulation assumes that $\hat{M}_{1,k}$ represents $M_{1,k}$ exactly, which is likely not the case given the system's complexity. In the absence of a well-modeled $\hat{M}_{1,k}$, a constant symmetric $\hat{M}_{nom}$ will be picked such that $\hat{M}_{1,k} = \hat{M}_{nom} \forall k$. $\hat{M}_{nom}$ has the following structure:

$$\hat{M}_{nom} = \begin{bmatrix} \hat{M}_{bb} & \hat{M}_{bq} \\ \hat{M}_{qb} & \hat{M}_{qq} \end{bmatrix} \tag{5.18}$$

where $\hat{M}_{bb} \in R^{6\times6}$, $\hat{M}_{bq} = \hat{M}_{qb}^T \in R^{6\times16}$, and $\hat{M}_{qq} \in R^{16\times16}$. It is particularly important that $\hat{M}_{bq} \neq 0$ to reflect some degree of coupling between the joint states q and the trunk states p_b and θ_b. In general, if $\hat{M}_{nom}$ should be selected to reflect the *average* system mass matrix over the range of configurations, z_1, seen during gaiting. This approximation has shown to be adequate from our results, and depends on the assumption that changes in $\hat{M}_{1,k}$ are small over the subset of state z_1 experiences during a periodic gaiting sequence. Future improvements of this controller involve the formulation of a separate estimator for $M(z_1)$, or a control/learning scheme with no direct dependence on $M(z_1)$.

Since $\hat{\psi}_{k+1}$ is non-causal, training is performed one time-step after a prediction is made using the input-output pair $\hat{\psi}_k$ and $\{\Psi_{k-1}^N,\ U_{k-1}^N\}$. Note that $\hat{\psi}_k$ can calculated

directly using (5.17) where all component signals are time-delayed by one time-step. Training can then be described by:

$$\psi_k \xrightarrow{BP(\gamma^{lr})} N(\Psi_{k-1}^N, U_{k-1}^N) \tag{5.19}$$

where $\gamma_{min}^{lr} < \gamma^{lr} < \gamma_{max}^{lr}$ is a learning rate adapted using a *bold-driver* update routine. Bold-driver learning-rate adaptation is a heuristic method for speeding up the rate of convergence of back-propagation training regimes [34, 35]. This γ^{lr} update law is parameterized by $\beta \in (0,1)$ and $\zeta \in (0,1)$ which are selected to specify the amount by which γ^{lr} increases or decreases per update, and γ_{min}^{lr} and γ_{max}^{lr} which are used to saturate γ^{lr}. The bold-driver scheme utilizes the current and previous mean-squared network output error values (MSE_k and MSE_{k-1}, respectively) to adjust γ^{lr} as follows:

$$\gamma^{lr} \leftarrow \begin{cases} \gamma^{lr}(1-\beta) & \text{if } MSE_k > MSE_{k-1} \\ \gamma^{lr}(1+\zeta\beta), & \text{otherwise.} \end{cases} \tag{5.20}$$

Since network training is being performed on-line as an incremental routine, the effective mean-squared NARX-NN output error is low-passed by a factor $\lambda \in (0,1)$. This update technique has been selected to ensure that outliers presented during training do not affect network learning updates as significantly as "nominal" training pairs. Network output error, $e_{N,k}$, and its associated MSE values are calculated after each prediction by:

$$\begin{aligned} e_{N,k} &= \hat{\psi}_k - \psi_k \\ MSE_k &\leftarrow \lambda \|e_{N,k}\|_2^2 + MSE_{k-1}(1-\lambda). \end{aligned} \tag{5.21}$$

5.5.3 Compensator Output

The control scheme is first presented with respect to the servo input torques, $\tau_{q,k}$, and formulated to achieve a level trunk characterized by $\theta_b = 0$, $\dot{\theta}_b = 0$. To formulate this controller, the dynamical sub-system which corresponds to the un-actuated trunk orientation states is isolated by:

$$\ddot{\theta}_b = \Gamma_1 M^{-1}(z_1)(\Gamma_2 \tau_q + \Phi) \tag{5.22}$$

where

$$\begin{aligned} \Gamma_1 &= [0_{3\times3}, I_{3\times3}, 0_{3\times16}] \\ \Gamma_2 &= [0_{16\times6}, I_{16\times16}]^T \end{aligned}$$

and $\Gamma_2\tau_q$ is equivalent to the original system input, τ. In order to enforce a level platform with zero angular velocity, we seek a τ_q which emulates the proportional-derivative (P.D.) control law:

$$\ddot{\theta}_b = -K_b\theta_b - K_d\dot{\theta}_b \tag{5.23}$$

where K_b and K_d are constant gain matrices. Using this P.D. law and (5.22), we propose a least-squares solution for τ_q by:

$$\tau_q \approx -\left[\Gamma_1 M^{-1}(z_1)\Gamma_2\right]^\dagger \Gamma_1 M^{-1}(z_1)(\Phi + K_b\theta_b + K_d\dot{\theta}_b) \tag{5.24}$$

where $[*]^\dagger$ denotes the Penrose-Moore pseudo-inverse of $[*]$. Replacing all dynamical terms with their associated discrete-time equivalents, and Φ by the NARX-NN output $\hat{\Phi}_k = \hat{\psi}_{k+1}$, we apply ((5.24)) to arrive at the following required joint torque estimate:

$$\hat{\tau}_{q,k} = -\left[\Gamma_1 \hat{M}_{1,k}^{-1}\Gamma_2\right]^\dagger \Gamma_1 \hat{M}_{1,k}^{-1}(\hat{\psi}_{k+1} + K_b\theta_{b,k} + K_d\dot{\theta}_{b,k}) \tag{5.25}$$

where $\theta_{b,k}$ and $\dot{\theta}_{b,k}$ are samples of angular trunk position and rate, respectively.

Using the joint controller dynamics presented in **??** and the estimate $\hat{\tau}_{q,k}$, we can formulate a reference-trajectory correction which is used to alter the joint reference positions, q_k^r. Moreover, the corrected reference position, $q_{1,k}^{r,*}$ is generated such that the estimated output torque $\hat{\tau}_{q,k}$ is attained by each joint controller. This joint-reference compensator output is defined using (5.25) as follows:

$$q_{1,k}^{r,*} = k_s^{-1}\left(\hat{\tau}_{q,k}\right) + q_{1,k}. \tag{5.26}$$

The correction signal, $q_k^{r,*}$, is combined with the original gaiting trajectory signal, q_k^r, as a weighted sum to form a compensated joint control reference signal, $\tilde{q}_k^r$, defined by:

$$\tilde{q}_k^r \leftarrow (1-\alpha)q_k^r + \alpha(q_k^{r,*}) \tag{5.27}$$

where $\alpha \in (0,1)$ is a uniform mixing parameter. The parameter α must be tuned with respect to the stability margins of the gait being compensated. The resultant $\tilde{q}_k^r$ is then applied to each joint controller in place of the original reference signal, q_k^r, generated by the gait controller. Selection of the parameter α is crucial for achieving good performance.

5.5.4 NARX-NN Compensator Results

The NARX-NN compensator has been tested, exclusively, in simulation and has been applied to applied to the quadruped as it executes a stable CPG-driven trot gait

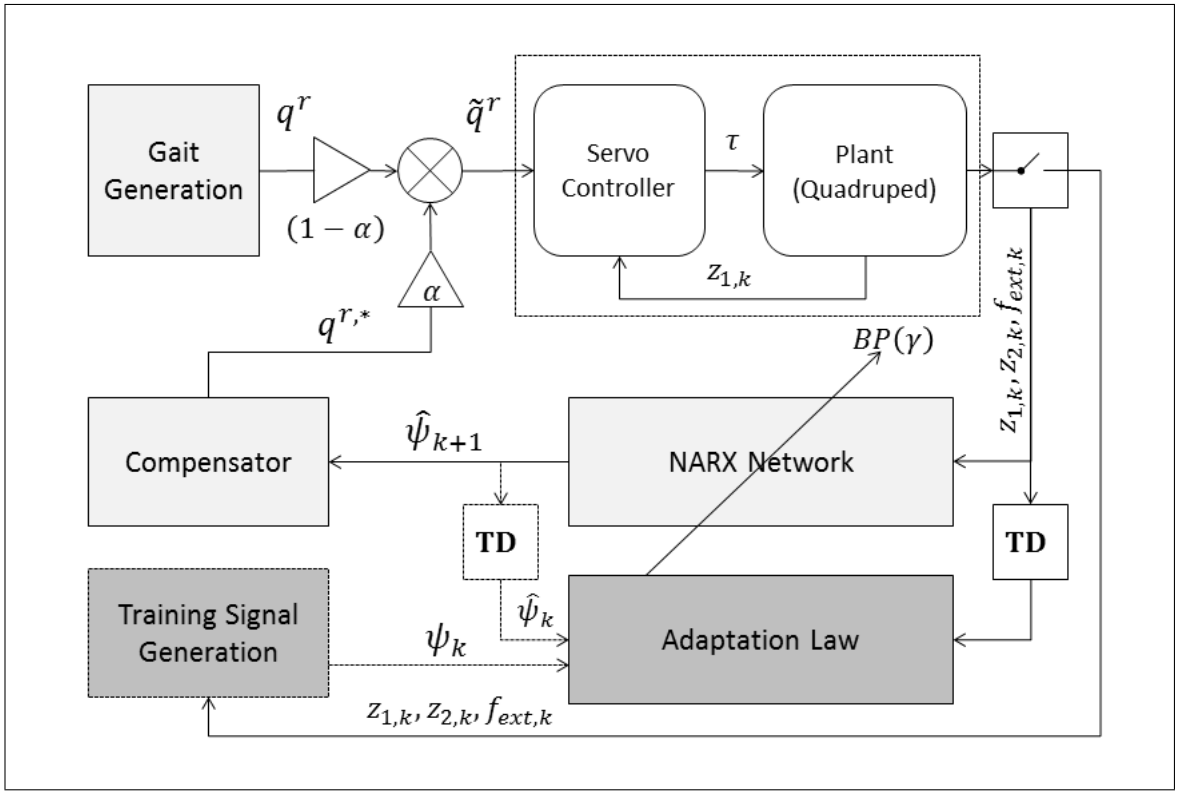

Figure 19: Full system diagram with NARX-NN compensator mechanism.

depicted in Figure 17. In these trials, gaiting frequency is adjusted accordingly to achieve particular forward speeds. NARX-NN parameters are fixed for all trials with learning-rate parameters set to $\beta = 0.0001$, $\zeta = 0.0005$ and $\lambda = 0.01$. The NARX-Network is configured with two hidden layers containing 50 neurons each. Each input and hidden-layer neuron is modeled using a symmetric sigmoid activation function. Output layer neurons are modeled using linear activation functions to avoid output-scaling saturation issues. Figure 20 exemplifies the convergence of the NARX-NN prediction error when the platform executes a gait at 100 $\frac{mm}{s}$ with $\alpha = 0.35$.

All simulated trials are performed over a period of 60 seconds each. During the first 10 seconds of each simulation, the robot moves from sitting position to a standing position and initiates walking. During each simulation period, the NARX-NN compensator is activated (not training) and deactivated (training) every 10 seconds. Figures 21, 22 and 23 depict an initial set of simulation results showing the effect of varying the mixing parameter $\alpha \in \{0.125, 0.25, 0.35\}$. For all such trials, the robot performs a trot-gait which achieves a forward speed of 60 $\frac{mm}{s}$. It is expected that as α increases, the compensator will have greater authority over trunk stabilization. From these results, we

observe that for all α, disturbance magnitude is decreased to some extent. However, for smaller α, the compensator is less effectual due to the fact that it has less authority over joint reference signals. From the results in Figure 22, it is shown that the compensator improves pitch stability by more than roughly 50% and roll stability by more than 60%. Figure 24 shows the compensator's performance at higher gaiting speeds of 80 $\frac{mm}{s}$ and 100 $\frac{mm}{s}$. Here the controller improves both pitch and roll by nearly 50% and 40% of the uncompensated signal magnitude, respectively.

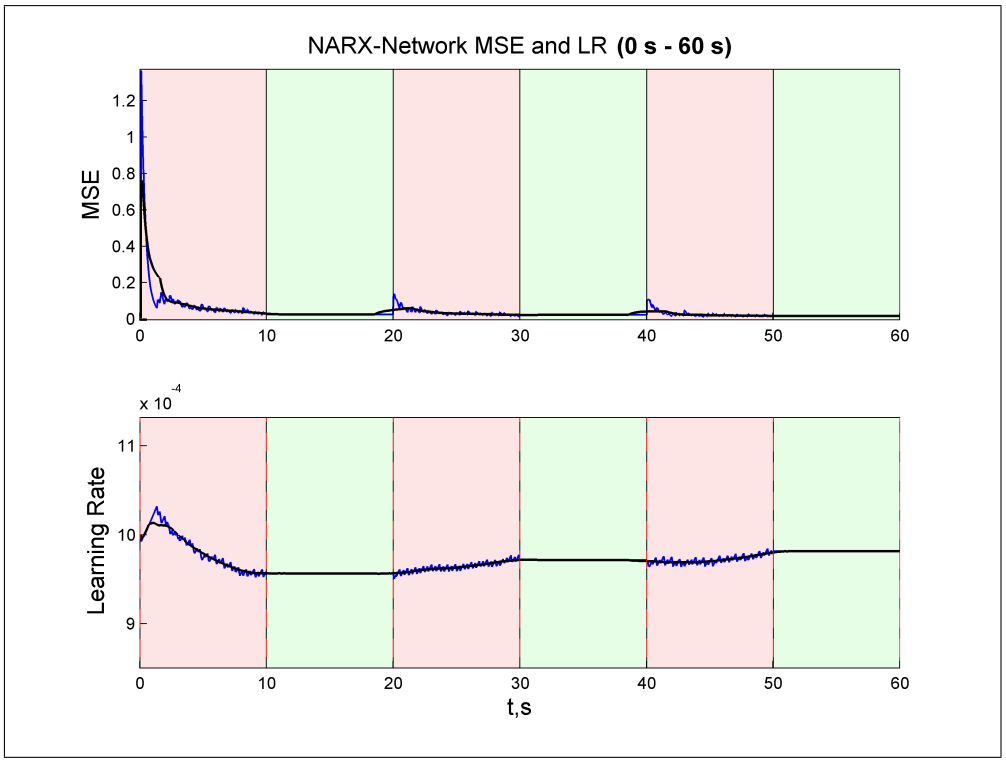

Figure 20: NARX Network MSE convergence for trial shown in Figure 25

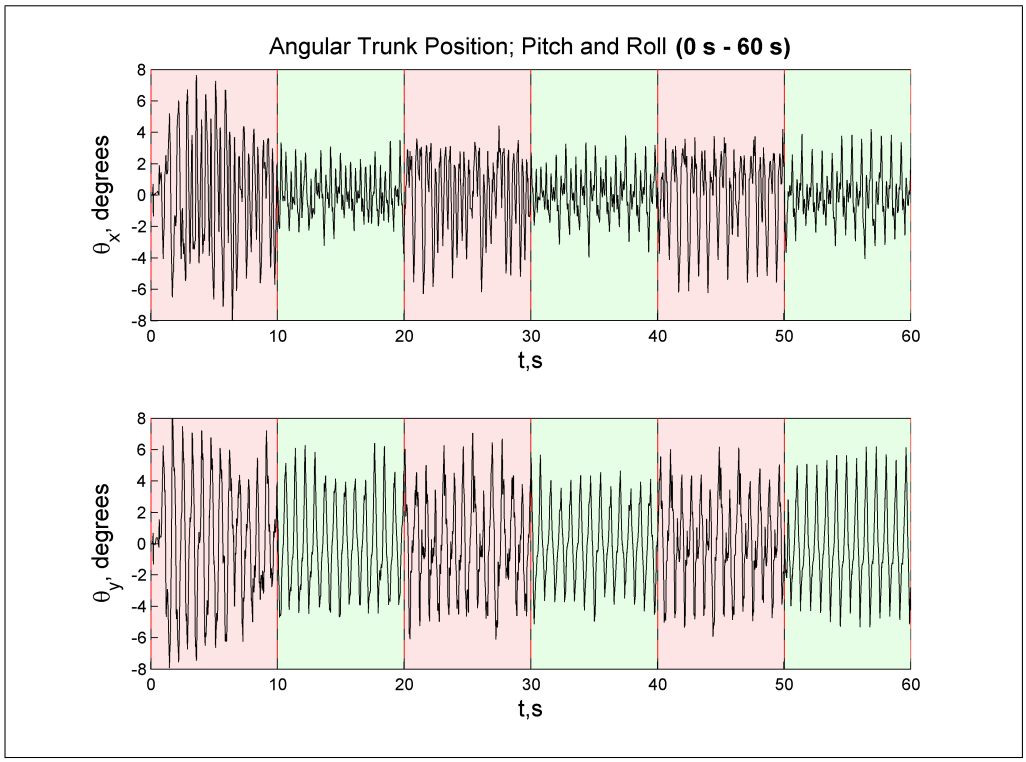

Figure 21: Trunk orientation during 60 $\frac{mm}{s}$ gait with mixing parameter set to $\alpha = 0.125$.

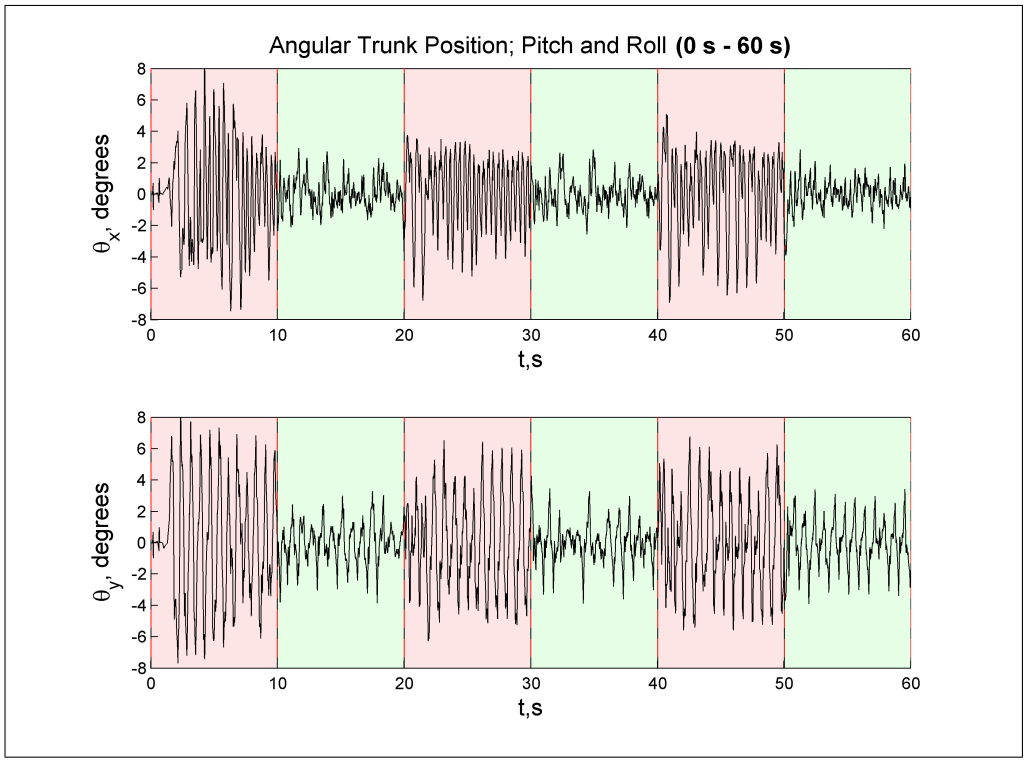

Figure 22: Trunk orientation during 60 $\frac{mm}{s}$ gait with mixing parameter set to $\alpha = 0.250$.

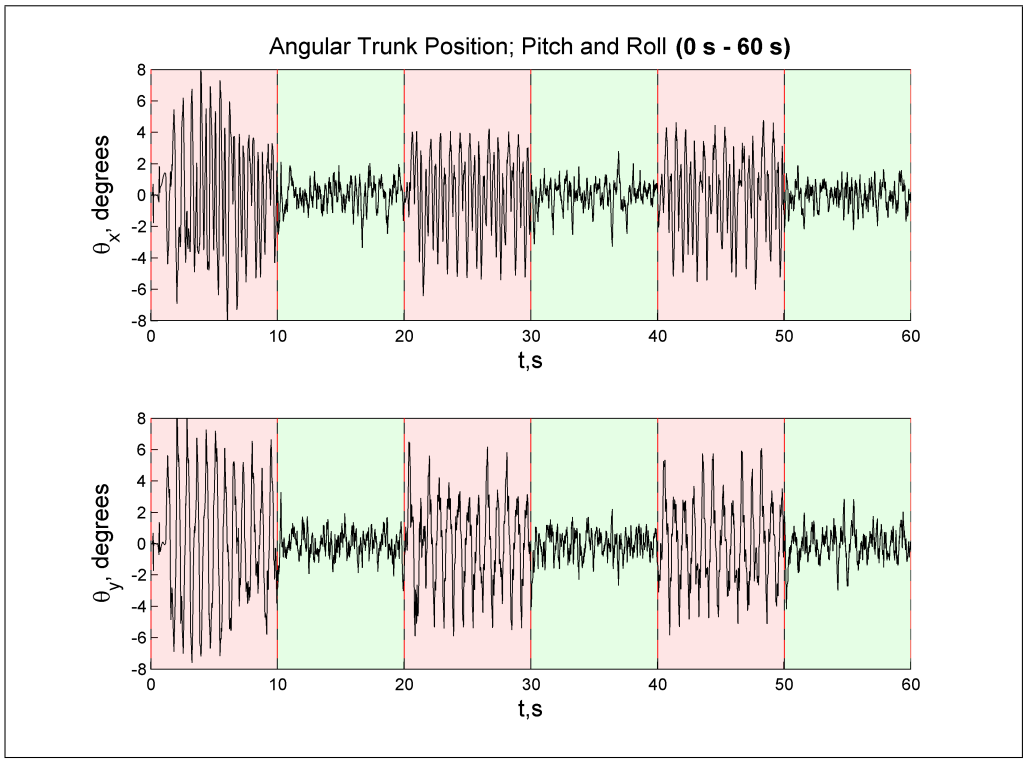

Figure 23: Trunk orientation during 60 $\frac{mm}{s}$ gait with mixing parameter set to $\alpha = 0.350$.

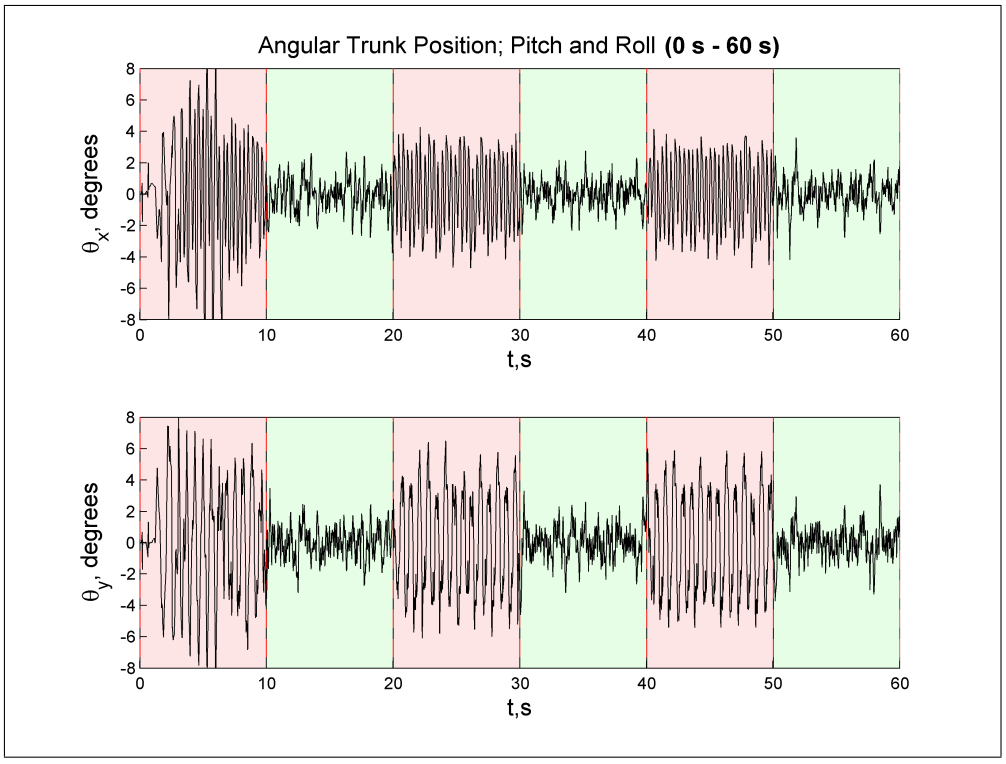

Figure 24: Trunk orientation during 80 $\frac{mm}{s}$ gait with mixing parameter set to $\alpha = 0.35$

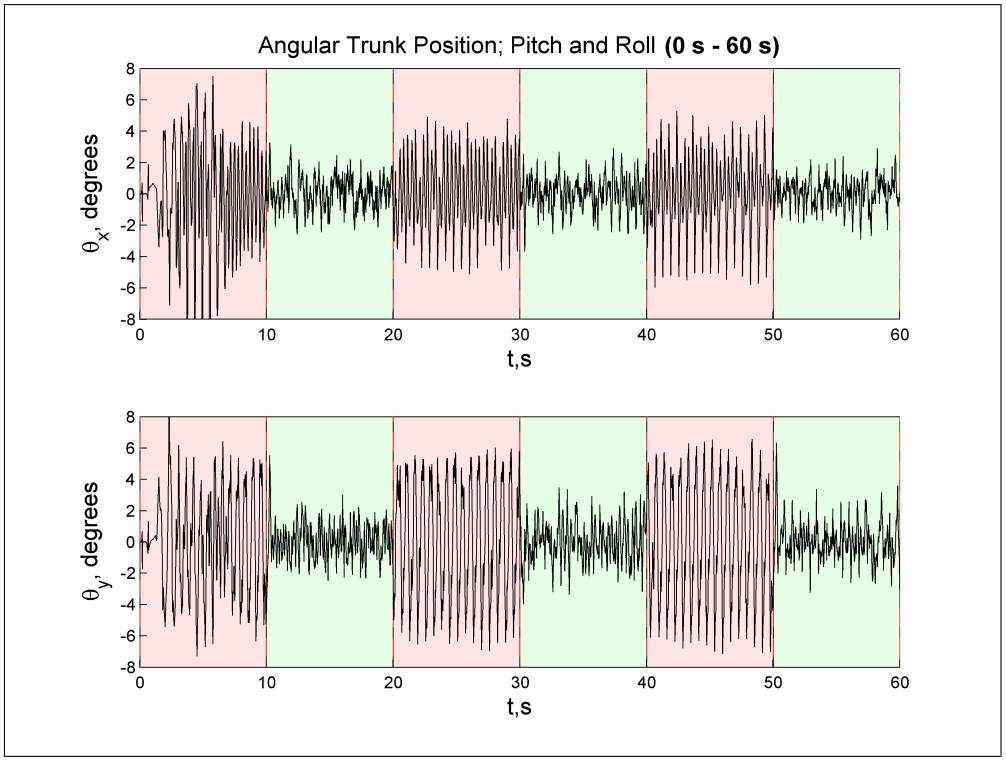

Figure 25: Trunk orientation during 100 $\frac{mm}{s}$ gait with mixing parameter set to $\alpha = 0.35$

CHAPTER VI

Perception and Navigation

6.1 Flatland navigation and Goal Tracking

The BlueFoot has been outfitted with flatland navigation controls for obstacle avoidance and camera-based goal-tracking over flatland. Platform navigation is governed by two main command parameters, v^r and ω^r, which represent forward and turning rate of the platform with respect to the robot's trunk frame O_b. During navigation, the pitch of the trunk, $\theta_{b,x}$ and $\theta_{b,z}$ is also controlled via its respective reference signal, $\theta^r_{b,x}$. This control over trunk pose allows for additional articulation of the LIDAR and camera sensors mounted to BlueFoot's head.

6.1.1 LIDAR-based Potential-Fields Algorithm

BlueFoot's base navigation and obstacle avoidance algorithm involves a potential fields-based approach which fuses LIDAR data and features from processed camera images. This algorithm is essentially used as a "wandering" mechanism which would normally be employed as a first-level navigation measure in an unknown environment when no information (*e.g.*, map data) is known about the environment *apriori*. This navigation approach uses only 2D LIDAR data (does not take into account features of terrain) and is thus most fit for navigation over flatland.

The LIDAR-base portion of this algorithm takes sequential, planar LIDAR scans as an input and generates an output in the form of a vectored navigation command, $\vec{V}^r_L \in \Re^3$, relative to the world frame, O_0, as defined in Section 4.1.1. Each point from a LIDAR scan is mapped to a corresponding scalar *potential* which is used to influence the direction of the newly generated command. Given a LIDAR scan S with 2D scan points, $x^L_i \in S^L$, relative to the LIDARs local coordinate frame O_L, an output command vector is generated using a potential function $\{f(x) : \Re^3 \rightarrow \Re^1\}$, and a biasing function,

$\{g(x,\psi) : \Re^3 \rightarrow \Re^1\}$, as follows:

$$\begin{aligned} x_i &= P_{\vec{z}}\left(H_b^0 H_L^b \Gamma_L x_i^L\right) \\ \vec{V}_c &= \alpha_c \left(\sum_{x_i \in S} |f(x_i - p_b)|_1\right)^{-1} \left(\sum_{x_i \in S} g(x_i - p_b, \psi) f(x_i - p_b) \frac{x_i - p_b}{\|x_i - p_b\|}\right) \end{aligned} \quad (6.1)$$

where H_b^0 defines a homogeneous transformation between O_0 and O_b; H_L^b defines a homogeneous transformation which relates the LIDAR sensors pose to the frame O_b; $S \subset \Re^3$ is the set of newly transformed points, $x_i \in S$ which respect the current LIDAR scan in the world coordinate system; α_c is a scalar tuning parameter; and

$$\Gamma_L = [I_{2\times 2}, 0_{2\times 1}]^T .$$

Since we are assuming navigation over flat ground, the operator $P_{\vec{z}}(*)$ is used to project each transformed LIDAR scan-point onto the plane defined by the z-axis unit vector in the frame O_0.

The piecewise-continuous potential function, defined in (6.2), used in BlueFoot's navigation scheme is dsigned to "repel" the platform from objects which are at some minimum distance, d_{min} from the trunk, and attract toward objects that are further away. As a result, this function tends to draw the robot towards long apertures, such as corridors or openings, and away from close-by obstructions. It is written as follows:

$$f(x) = \begin{cases} (\|x\| - d_{min})\,\lambda_{c,1} & \text{if } \|x\| - d_{min} < 0 \\ (\|x\| - d_{min})\left(1 - e^{-\lambda_{c,2}(\|x\| - d_{min})^2}\right) & \text{else} \end{cases} \quad (6.2)$$

where $\lambda_{c,1} > 0$ and $\lambda_{c,2} > 0$ are tuning parameters sued to specify the output range and sensitivity of the potential function output with respect to $(\|x\| - d_{min})$, respectively. It can be observed that this potential function exhibits $f(x) < 0$ when $\|x\| < d_{min}$ and vice-verse, thus achieving the desired attractive/repulsive characteristics.

The biasing function $g(x,\psi)$ is used to weight the effect of individual scan-point potentials, $f(x_i - p_b)$, on the final navigation output, with respect to the angular-window parameter $\psi > 0$. A simple masking-type biasing function is used in BlueFoot's navigation function, which is formally defined as follows:

$$g(x,\psi) = \begin{cases} 1 & \tan^{-1}\left(\frac{x_2}{x_1}\right) < \psi \\ 0 & \text{otherwise.} \end{cases} \quad (6.3)$$

where x_1 and x_2 are the first and second elements of the vector argument $x \in \Re^3$. This function is used to give priority to "foward" points within an angular window ψ and serves to reduce the potential for getting stuck in local minima. In the event that the robot does get stuck in a potential minima, a "stuck" detection algorithm has been implemented. As the name suggests, this algorithm first detects if the robot is stuck in a local minima of the potential field based on a window of command histories.

Finally, the command vector $\vec{V}_L^r$ is transformed into scalar forward and turning rate commands, v_L^r and ω_L^r, respectively, by the following proportional control scheme:

$$\begin{aligned} \dot{v}_L^r &= \beta_v \left(\left\| \vec{V}_L^r \right\| - v_L^r \right) \\ \dot{\omega}_L^r &= \beta_\omega \left(\omega_L^{r,max} \left(\tan^{-1} \left(\frac{\vec{V}_{c,2}^L}{\vec{V}_{c,1}^L} \right) - \theta_{b,z} \right) \pi^{-1} - \omega_L^r \right) \end{aligned} \tag{6.4}$$

where β_v and β_ω are proportional-gain tuning parameters; and $\theta_{b,z}$ is the platforms yaw in O_0. The parameters β_v and β_ω can be viewed as "update-inertias," as they directly effect the influence of the effect of instantaneous commands on the forward velocity and turning rate of the robot. Furthermore, these gains can be used to low-pass navigation updates to remove jitter cause by command outliers generated from degenerate sensor readings.

6.1.2 Results of LIDAR-based potential field navigation from simulation

6.1.3 Incorporation of Camera-based Feature-Tracking

Camera-based goal tracking is used in conjunction with the aforementioned potential fields navigation scheme to move the platform through an environment while seeking or tracking a particular target. In this case, targets take the the form of features extracted from processed camera data. The robot is guided towards these features using a simple visual-serving approach.

Trackable camera features can be generated in a variety of ways. For the purpose of BlueFoot's navigation, objects with distinct shapes or color have been chosen for tracking so that shape and blob detection algorithms can be employed to detect their positions relative to BlueFoot's camera view range. Namely, Bluefoot's image processing routines make use of the Hough Transform-based shape detection and standard color-blob detection algorithms available from the Open Computer Vision (OpenCV) libraries [Need Hough Ref, Need OpenCV Ref]. Once detected, center-positions of each feature, represented with respect to the 2D camera-viewing frame, are mapped into forward rate

and turning commands to control the robot towards the relative location of the feature. These camera-based commands are then mixed with the outputs of the potential-fields navigation controller to form a hybrid navigation control law.

In this visual-servoing approach, features are used to control the robot in a way that is agnostic of the type of feature being tracked. Namely, this approach relies on the relative position of the center of each features, represented as a pixel-position, $p_{Im} = [u, v]^T \in Z^2$ in the 2D image frame, O_{Im}, and a relative size, r, measured in pixels. In the case of circular features, for example, r is represent the radius of the detected circle. For color-blob features, r represents the radius of a circle which fully inscribes the colored object.

For the purpose of target tracking, it is desired that the robot's forward speed be controlled such that is proportional to r. Namely, it is desired that the robot stop when it becomes close to the target object, and move faster when the feature is in sight but the robot is further away. The position of the feature's center us used to control the robot's turning rate, as well as the commanded pitch of the robot's trunk, $\theta^r_{b,x}$. Trunk articulation important during feature tracking routing as it aids in keeping the tracked-target objects centered in the image frame. Provided that the target is moving slower than the what the system can achieve, this will ensure the target remains in sight at all times.

A separate set of navigation commands, v^r_C and ω^r_C; and an additional body-pitching command, $\theta^r_{b,x}$, are generated from an extracted feature location, $p_{Im} = [u, v]^T$ as follows:

$$\begin{aligned} v^r_C &= v^{r,max}_C \left(1 - e^{-c_r(r-r_{min})^2}\right) \\ \omega^r_C &= \omega^{r,max}_C \left(\frac{w_{Im} - 2u}{w_{Im}}\right) \\ \theta^r_{b,x} &= \theta^{r,max}_{b,x} \left(\frac{2v - h_{Im}}{h_{Im}}\right) \end{aligned} \tag{6.5}$$

where $v^{r,max}_C$, $\omega^{r,max}_C$ and $\theta^{r,max}_{b,x}$ are the maximum magnitude of forward velocity, turning rate, and body-pitching commands, respectively; c_r is a sensitivity parameter; r_{min} defines a minmum feature size which will result in the administration of a zero velocity command to the platform (and thus the distance from the feature at which to halt forward motion); and w_{Im} and h_{Im} define the width and height, respectively, of the image being processed. Having now established a formulation for how a single, distinct, feature is used to guide the platform towards a target, a means of fusing the LIDAR-based command signals and the camera-based command signals will be defined.

The composition of this hybrid command technique is motivated by two key sub-tasks: to use the potential fields algorithm during a wandering phase, when a target object is not in sight; and to guide the robot towards the goal once in sight, allowing the camera-based tracking commands to have a greater influence on system navigation (through the variables v^r and ω^r) as the platform becomes closer to the desired target. A straight-forward way to achieve this is to use the relative size. r, of tracked target features in the image frame. With this in mind, a simple command mixture scheme has been defined as follows:

$$\begin{aligned} v(r) &\equiv \begin{cases} e^{-c_{mix}(r-r_{mix})^2} & \text{if } r < r_{mix} \\ 1 & \text{else} \end{cases} \\ \begin{bmatrix} v^r \\ \omega^r \end{bmatrix} &= v(r) \begin{bmatrix} v_C^r \\ \omega_C^r \end{bmatrix} + (1 - v(r)) \begin{bmatrix} v_L^r \\ \omega_L^r \end{bmatrix} \end{aligned} \tag{6.6}$$

where c_{mix} is a sensitivity parameter; and r_{mix} defines the processed-feature size which will grant full navigation control to the camera-based navigation scheme. The reasoning for such a scheme involves a heuristic approach to obstacle aversion, which assumes that when the platform is further away from a goal, there is a higher probability that it will encounter an obstacle. Conversely, when the goal becomes closer to the platform, it assumed that the number of obstacles between the robot and the target decreases, making it safe to shift full priority to reaching the goal from the current position.

6.1.4 Hybrid Potential-Fields Navigation Results

NEED RESULTS

6.2 Rough Terrain Navigation

6.2.1 Terrain Mapping with 3D Point-clouds

3D Point Cloud generation from 2D scans

Figure 26: Need sensor sweep diagram.

The BlueFoot platform has the ability to compose 3D point clouds from a series of swept 2D LIDAR scans in conjunction with a trunk orientation estimates, $\hat{\theta}_b$, which is generated using an EKF. LIDAR articulation is achieved by slowly pitching the trunk over some angular range while keeping the platform's feet ridgedly planted. Sweeping

range is limited by the kinematic-feasibility of each trunk pose that must be reached during a sweep. Given a particular set of foot and body location, kinematic feasibility is validated using the inverse kinematics solution described in Section 4.1.2. A single 2D scan is taken at each pose within the body-sweep trajectory. The newly acquired scan is transformed from the LIDAR sensor frame, O_L, to the world frame by a homogeneous transformation H_0^L which is defined as follows

$$H_0^L = H_b^L H_0^b \tag{6.7}$$

where H_0^b is a transformation from O_0 to the trunk frame O_b, as defined in Chapter IV; and H_b^L defines a transformation from the frame O_b to the LIDAR frame, H_b^L. H_b^L is necessary for knowing the position of the LIDAR head with respect world frame, as the sensor itself has some offset and rotation relative to the robot's body. Each 2D point from $x_i \in S$ from the initial scan, $S \subset \Re^2$, can then be transformed into a 3D scan segment, $\bar{S}_j$, in O_0 by:

$$\begin{bmatrix} y_{i,j} \\ 1 \end{bmatrix} = H_b^L H_0^b \begin{bmatrix} x_i \\ 0 \\ 1 \end{bmatrix} \forall x_i \in S \tag{6.8}$$

where $y_{i,j} \in \bar{S}_j$ is a point withing the 3D j^{th} scan segment $\bar{S}_j \subset \Re^3$. After the sweeping routine is complete, 3D scan segements are composed into a final point cloud, $\bar{S}$ by:

$$\bar{S} = \cup_{j=1}^{N_s} \bar{S}_j \tag{6.9}$$

where N_s defines the number of scans taken during the sweeping routine. For the sake of simplicity, it is assumed that the trunk's position, p_b, is fixed (system is completely ridged) during a swept-scan routine. In the results to be presented, this seems to be a reasonable assumption given that the platform is at rest and the trunk is pitched sufficiently slowly over the angular sweeping range. A slow sweep rate ensures that perturbations caused by vibrations incurred by trunk rotation and foot-slip are small, and thus does not cause for significant deviations in LIDAR scan points.

Height-map from 3D point cloud

3D point clouds can be transformed into height-map representations. These representations are convenient for use in planning as they can be converted into discrete cost-maps in a straightforward way. These height-maps are generated by populating the elements of a matrix $M \in \Re^{n \times m}$ with the z components of points which exist within

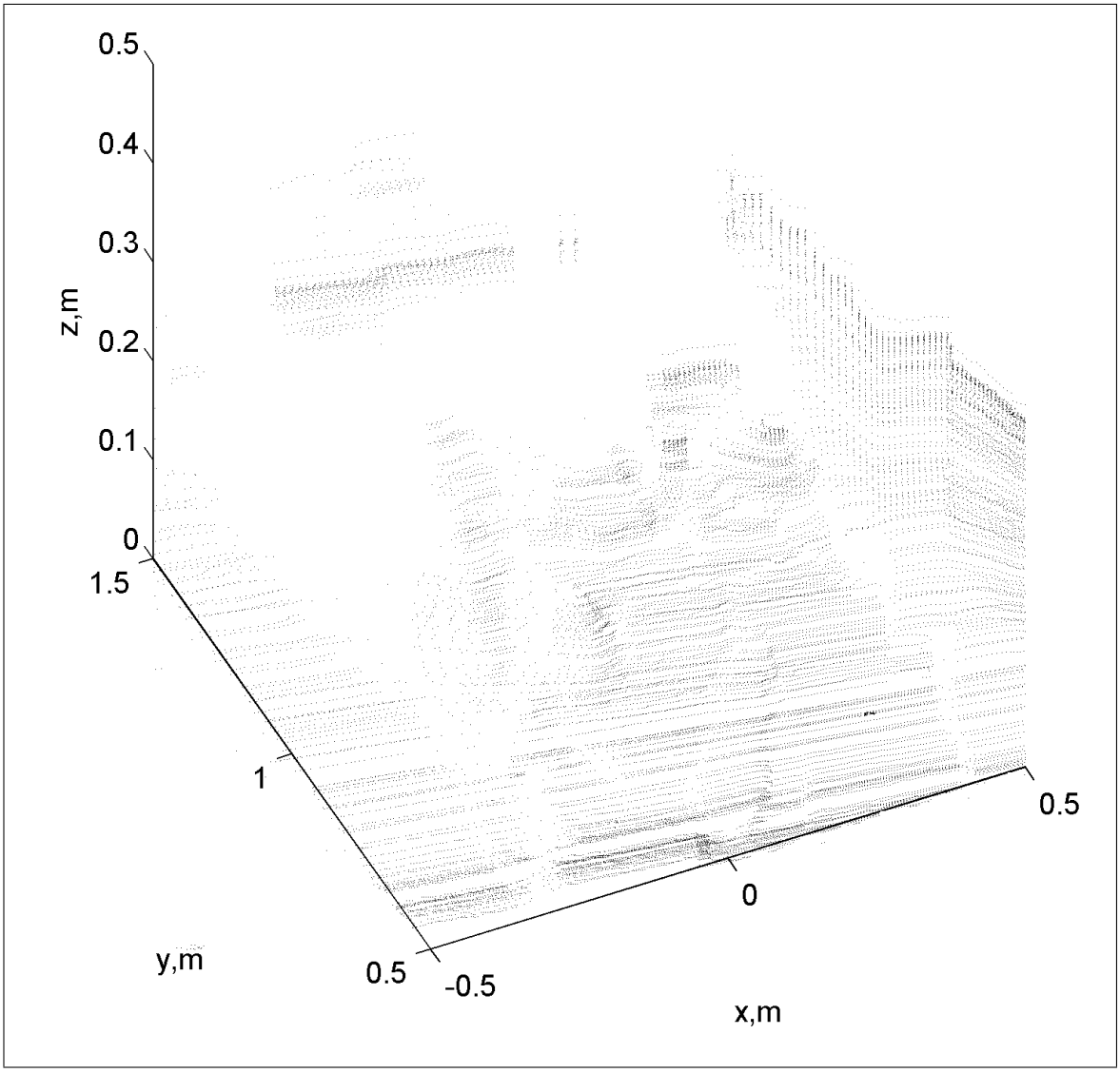

Figure 27: Original 3D point-cloud of terrain patch.

discretized $W \times D$ region within the original 3D point cloud. The location and size of this regions would have to be determined with some auxillary detection process which 1) determines that the area has high terrain variation, and 2) determines the bounding region where this patch of rough-terrain exits. Once this regions is determined, the 3D cloud is divided into $n \times m$ sub-divisions, each of which covers a $(W/m) \times (D/n)$ area. Each i^{th}, j^{th} element of M is then populated with the highest point (larger z component) within each i^{th}, j^{th} subdivision. Depending on the density of the 3D point cloud, this process has the potential to produce relatively sparse heighmap. To deal with this, a dialation and smoothing routine is used to complete the hieghtmap conversion process.

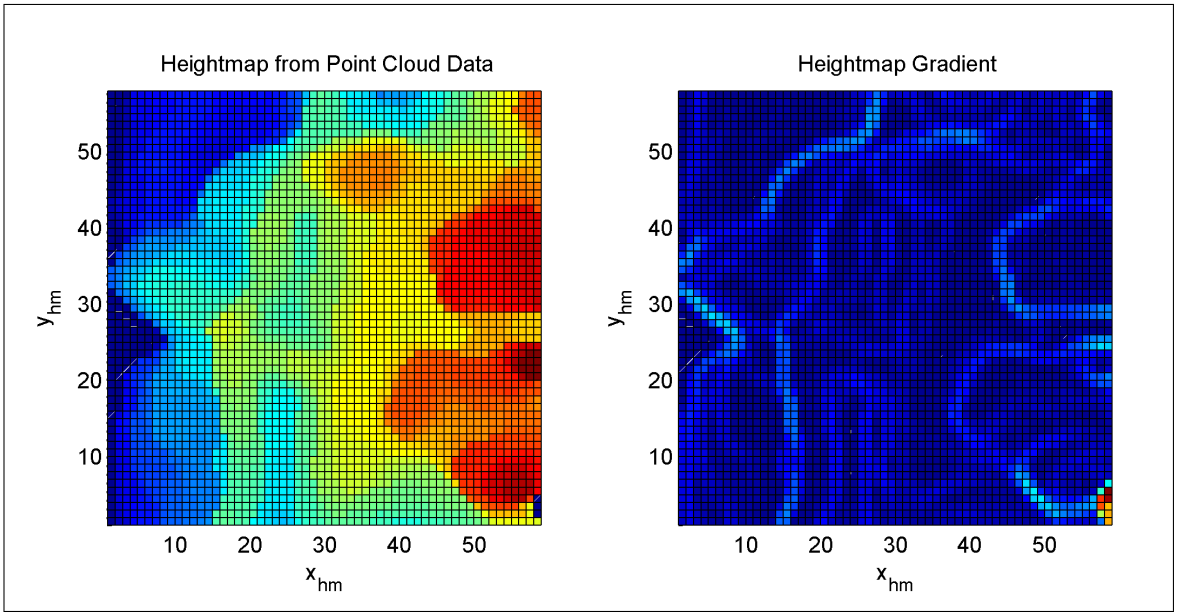

Figure 28: Relative height-map *(left)* and its corresponding gradient *(right)*.

During dilation, each non-zero height element within the map is expanded into a region around the exiting element until. This process is perfomed until a semi-uniform map is produced with minimal gaps betweenn non-zero height elements. Finally, a median filter is applied to the dilated heighmap to produce smooth transitions between height elements.

The full 3D point cloud-to-heightmap conversion algorithm is summarized as follows:

Surface Reconstruction

6.2.2 Proposed Methods for determining the existence of Rough Terrain

6.2.3 Foot-placement planning

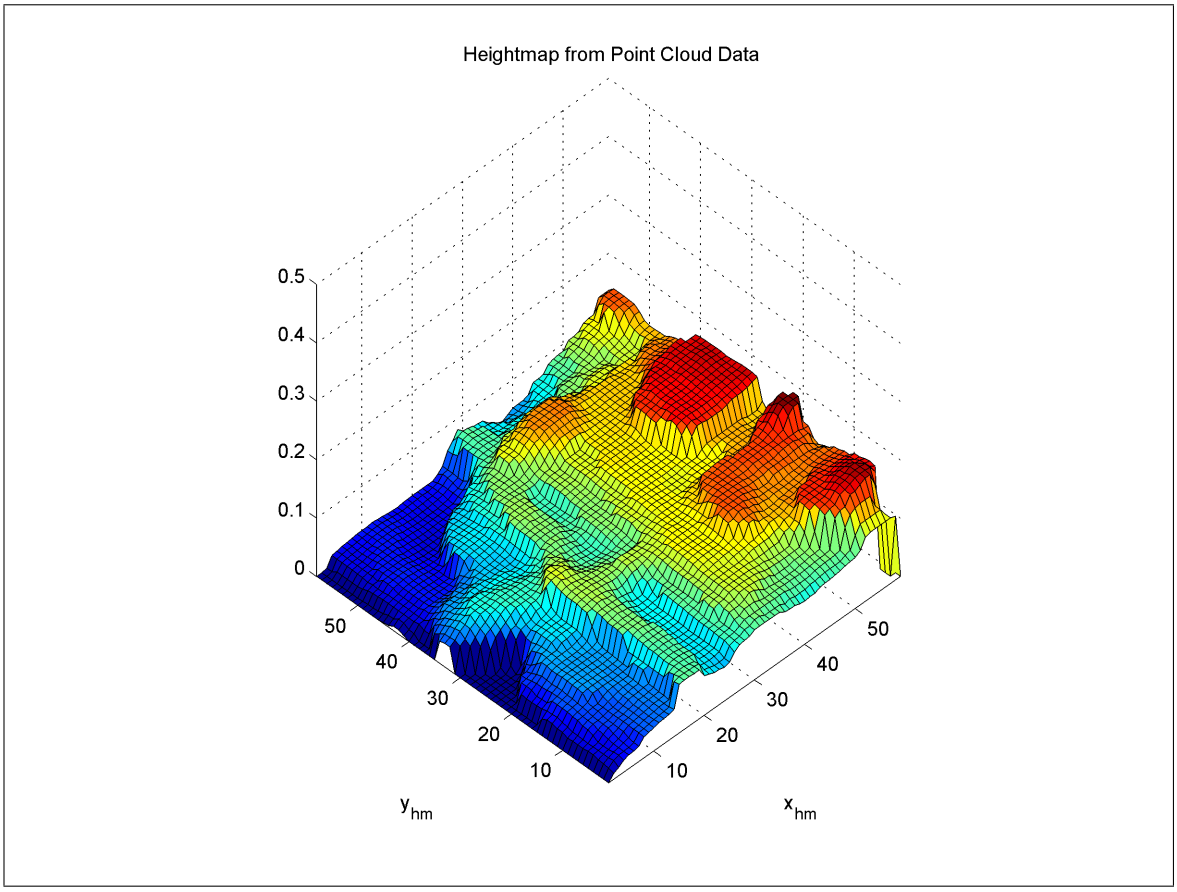

Figure 29: Height-map showing terrain variation in the z_{hm} direction.

CHAPTER VII

Concluding Remarks

[1] R.B. McGhee and A.A. Frank. On the stability properties of quadruped creeping gaits. *Mathematical Biosciences*, 3(0):331 – 351, 1968.

[2] J.K. Hodgins and M. Raibert. Adjusting step length for rough terrain locomotion. *IEEE Transactions on Robotics and Automation*, 7(3):289–298, Jun 1991.

[3] R. Altendorfer, N. Moore, H. Komsuoglu, M. Buehler, Jr. Brown, H.B., D. McMordie, U. Saranli, R. Full, and D.E. Koditschek. RHex: A Biologically Inspired Hexapod Runner. *Autonomous Robots*, 11(3):207–213, 2001.

[4] Y. Fukuoka, H. Kimura, Y. Hada, and K. Takase. Adaptive dynamic walking of a quadruped robot 'Tekken' on irregular terrain using a neural system model. In *IEEE International Conference on Robotics and Automation*, volume 2, pages 2037–2042 vol.2, Sept 2003.

[5] Joaquin Estremera and Kenneth J. Waldron. Leg Thrust Control for Stabilization of Dynamic Gaits in a Quadruped Robot. In Teresa Zieliska and Cezary Zieliski, editors, *Romansy 16*, volume 487 of *CISM Courses and Lectures*, pages 213–220. Springer Vienna, 2006.

[6] Marc Raibert, Kevin Blankespoor, Gabriel Nelson, and Rob Playter. BigDog, the Rough-Terrain Quaduped Robot, 2008.

[7] C. Semini. *HyQ Design and Development of a Hydraulically Actuated Quadruped Robot*. PhD thesis, Apr 2010.

[8] J.R. Rebula, P.D. Neuhaus, B.V. Bonnlander, M.J. Johnson, and J.E. Pratt. A Controller for the Littledog quadruped walking on Rough Terrain. In *IEEE International Conference on Robotics and Automation*, pages 1467–1473, April 2007.

[9] M. Ajallooeian, S. Pouya, A Sproewitz, and A.J. Ijspeert. Central Pattern Generators augmented with virtual model control for quadruped rough terrain locomotion. In *IEEE International Conference on Robotics and Automation*, pages 3321–3328, May 2013.

[10] Simon Rutishauser, A Sprowitz, L. Righetti, and A.J. Ijspeert. Passive compliant quadruped robot using Central Pattern Generators for locomotion control. pages 710–715, Oct 2008.

[11] P.-B. Wieber. Holonomy and nonholonomy in the dynamics of articulated motion. In Moritz Diehl and Katja Mombaur, editors, *Fast Motions in Biomechanics and Robotics*, volume 340 of *Lecture Notes in Control and Information Sciences*, pages 411–425. Springer Berlin Heidelberg, 2006.

[12] Russel Smith. *Open Dynamics Engine, http://www.ode.org/.*

[13] Kiyotoshi Matsuoka. Sustained oscillations generated by mutually inhibiting neurons with adaptation. *Biological Cybernetics*, 52(6):367–376, 1985.

[14] J.J. Collins and Ian Stewart. Hexapodal gaits and coupled nonlinear oscillator models. *Biological Cybernetics*, 68(4):287–298, 1993.

[15] G. Endo, J. Morimoto, J. Nakanishi, and G. Cheng. An empirical exploration of a neural oscillator for biped locomotion control. In *Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE Int. Conf. on*, volume 3, pages 3036–3042 Vol.3, April 2004.

[16] L. Righetti and A.J. Ijspeert. Programmable central pattern generators: an application to biped locomotion control. pages 1585–1590, May 2006.

[17] Auke Jan Ijspeert. Central pattern generators for locomotion control in animals and robots: A review. *Neural networks*, 21(4), 2008.

[18] Vitor Matos and Cristina P. Santos. Omnidirectional locomotion in a quadruped robot: A CPG-based approach. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3392–3397, Oct 2010.

[19] Meng Yee (Michael) Chuah Park, Hae-Won and Sangbae Kim. Quadruped bounding control with variable duty cycle via vertical impulse scaling. 2014.

[20] Y. Fukuoka, H. Yasushi, and F. Takahiro. A simple rule for quadrupedal gait generation determined by leg loading feedback: a modeling study. *Sci. Rep.*, 5, 2015.

[21] Tedrake R. Kuindersma S. Wieber, P.-B. Modeling and control of legged robots. In Siciliano and Khatib, editors, *Springer Handbook of Robotics, 2nd Ed*, Lecture Notes in Control and Information Sciences, chapter 48. Springer Berlin Heidelberg, 2015.

[22] Luiz Castro, Cristina P. Santos, Miguel Oliveira, and Auke Ijspeert. Postural Control on a Quadruped Robot Using Lateral Tilt: A Dynamical System Approach. In Herman Bruyninckx, Libor Peuil, and Miroslav Kulich, editors, *European Robotics Symposium 2008*, volume 44 of *Springer Tracts in Advanced Robotics*, pages 205–214. Springer Berlin Heidelberg, 2008.

[23] S. Kajita, F. Kanehiro, K. Kaneko, K. Fujiwara, K. Harada, K. Yokoi, and H. Hirukawa. Biped walking pattern generation by using preview control of zero-moment point. In *IEEE International Conference on Robotics and Automation*, volume 2, pages 1620–1626 vol.2, Sept 2003.

[24] Katie Byl, Alec Shkolnik, Sam Prentice, Nick Roy, and Russ Tedrake. Reliable Dynamic Motions for a Stiff Quadruped. *International Symposium on Experimental Robotics*, pages 319–328, 2009.

[25] A. Takanishi, M. Tochizawa, T. Takeya, H. Karaki, and I. Kato. Realization of dynamic biped walking stabilized with trunk motion under known external force. In KennethJ. Waldron, editor, *Advanced Robotics: 1989*, pages 299–310. Springer Berlin Heidelberg, 1989.

[26] R. Kurazume, T. Hasegawa, and K. Yoneda. The sway compensation trajectory for a biped robot. In *Robotics and Automation, 2003. Proceedings. ICRA '03. IEEE International Conference on*, volume 1, pages 925–931 vol.1, Sept 2003.

[27] Tsungnan Lin, B.G. Horne, P. Tino, and C.L. Giles. Learning long-term dependencies in NARX recurrent neural networks. *Neural Networks, IEEE Transactions on*, 7(6):1329–1338, Nov 1996.

[28] Grant P. M. Chen S., Billings S. A. Non-linear system identification using neural networks. In *Int. Journal of Control*, volume 51 of *Lecture Notes in Control and Information Sciences*, pages 1191–1214. Taylor and Francis, 1990.

[29] Salah El Hihi and Yoshua Bengio. Hierarchical recurrent neural networks for long-term dependencies, 1996.

[30] S. A Billings. *Nonlinear system identification : NARMAX methods in the time, frequency, and spatio-temporal domains*. John Wiley and Sons Ltd, 2013.

[31] Oliver Nelles. Neural networks with internal dynamics. In *Nonlinear System Identification*, pages 645–651. Springer Berlin Heidelberg, 2001.

[32] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Neurocomputing: Foundations of research. chapter Learning Representations by Back-propagating Errors, pages 696–699. MIT Press, Cambridge, MA, USA, 1988.

[33] David E. Rumelhart, Richard Durbin, Richard Golden, and Yves Chauvin. Backpropagation. chapter Backpropagation: The Basic Theory, pages 1–34. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1995.

[34] Roberto Battiti. First and second-order methods for learning: between steepest descent and newton's method. *Neural Computation*, 4:141–166, 1992.

[35] G. D. Magoulas, M. N. Vrahatis, and G. S. Androulakis. Improving the convergence of the backpropagation algorithm using learning rate adaptation methods. *Neural Comput.*, 11(7):1769–1796, oct 1999.