Imperial College London

Final Report for MSc Individual Project

Gait generation for quadrupedal walking robot

Author name:
David Jedeikin

Supervisor:
Dr. Petar Kormushev

Submitted in partial fulfilment of the requirements for the award of MSc in Human and Biological Robotics from Imperial College London

September 14, 2018

Word Count: 5992

Abstract

Over the past few decades we have seen significant advances in multi-pedal robotics. Legged robots are able to traverse terrain and environments where wheeled or tracked robots would otherwise fail. This report documents the design, implementation and testing of a bespoke high-level controller, implemented using ROS and Python, for DogBot: a quadrupedal robot developed by React Robotics. A gait graph method was used to generate multiple gaits, while a mapping function allowing the robot to be commanded in any direction was designed. A step interpolation algorithm has been constructed allowing online, step parameter changes, which has radically improved the response of the controller. An IMU was embedded in both the physical and simulated model of the robot allowing inference of the robots' orientation. Using this information, a yaw controller capable of operating in both manual and autonomous mode was developed, while a roll and pitch controller was designed to improve locomotion over uneven and inclined terrains. In order to defend against unanticipated forces or pushes, a push detector and defence strategies were designed. Finally, a Bayesian optimization hyper-parameter search was carried out in an attempt to *learn* the best parameters for certain scenarios. The majority of the testing was conducted in simulation as the physical robot experienced a number of mechanical failures. In simulation, the robot performed consistently and well, while although the physical robot showed significant potential it required additional testing and parameter tuning. This controller provides a strong and modular base upon which further research may be conducted.

Acknowledgements

First and foremost, I would like to thank my supervisor, Dr. Petar Kormushev, for guiding me through this incredible project. Petar's wisdom and experience has been invaluable, he has taught me how to efficiently simplify and break down complex problems into achievable tasks, starting with simple solutions and increasing complexity where necessary. I would also like to thank Ke Wang and Roni Saputra for their constant support, assistance and advice. Lastly I would like to thank Gor Nersisyan, Kawin Larppichet, Shloak Mehta, Marion Tormento and Louis Rouillard for their support and great company throughout this brilliant experience.

Contents

List of Figures

Chapter 1

Introduction

Over the past few decades, a significant amount of research regarding developing and improving quadrupedal robotic platforms has been conducted. This research is being driven by the potential implications and applications of such technology. In particular, stable and well-controlled legged robots will be able to access locations that are inaccessible to wheeled or tracked robotic platforms. Among other things, this will allow autonomous systems to complete tasks that currently lead people into dangerous or potentially life-threatening situations. Examples of quadrupedal robots that have been - and are being - developed include Boston Dynamics Big Dog[2], Spot, Spot mini, Wildcat and LS3, for which there have been very few scientific publications. Other such robots include MIT's cheetah [3], IIT's HyQ [4], HyQmini [5] and HyQ2max [6] and ETH Zurich's starlETH [7], and ANYmal [8].

In addition to research focusing on physical robot implementations, significant simulation and animation-based research has successfully been conducted. Such an example is the work conducted by van de Panne et al. [9] in the development of locomotion skills for simulated quadrupeds. Gait graphs and motion capture comparisons were used to optimize and generate exceptionally realistic quadrupedal gaits and skills. This research clearly shows that gait generation has been successfully implemented in a number of different robots, by a number of different research groups. However, this project focuses on developing a high-level controller for DogBot (Figure 1): a unique quadrupedal robot, developed by React Robotics, with different dynamics, physical properties, sensory capabilities and low-level joint controllers. Although inspiration, principals and theories from previous publications may be utilized, a novel and bespoke high-level controller is required.

Figure 1.1: Side view photograph of DogBot

Figure 1.2: Oblique angle photograph of DogBot

The robot's current design allows for position control of the 12 joints and a ROS controller has been developed by React Robotics, thus the output of the high-level controller should be the joints required to drive the robot. The aim of this project is therefore to develop a novel and bespoke high-level gait controller for DogBot that is able to:

1. Generate multiple gaits
2. Manoeuvre in all directions
3. Be autonomously and/or manually controlled
4. Be robust, perturbation-resistant, and able to traverse uneven terrain.

Since it is likely that this controller will be improved upon in the future and used to develop, implement and test machine/reinforcement learning algorithms, additional major requirements include the usability, simplicity, and modularity of the controller. The controller must be treated as a black-box model, passing and changing parameters as required.

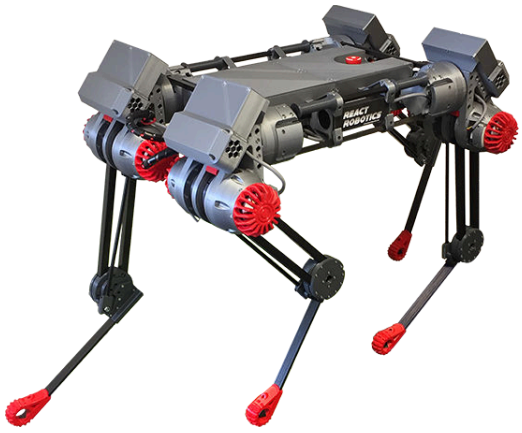


Figure 1.3: React Robotics photograph of DogBot [1]

Chapter 2

Methods

This section will include the detailed design of the high-level controller. It will explain the controller conceptionally, as well as the software design involved in implementing the controller. The controller was written using Python and ROS, and Gazebo was used to visualize and test the algorithms. Figure 2 below shows a flow diagram of the final high-level controller. In all flow diagrams, red parallelograms will represent global variables, green rectangles will represent callback functions and orange rectangles will represent separate threads. This section will explain the controller using a bottom-up approach, meaning that it will begin by explaining the control of a single leg and a two-dimensional step trajectory, thereafter increasing the complexity and building up the various components of the high-level controller shown in Figure 2.

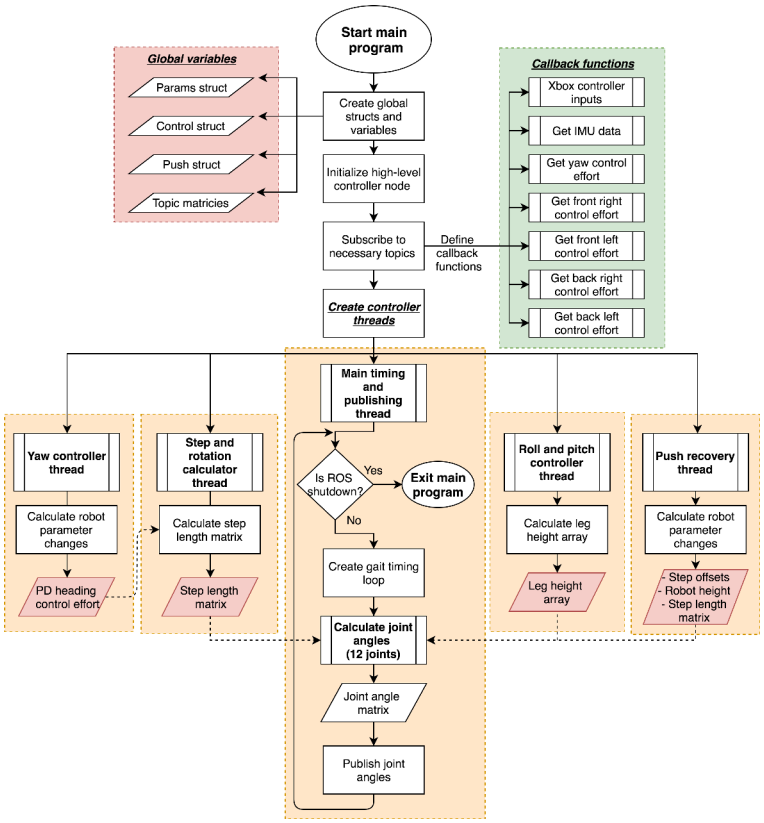


Figure 2.1: Flow diagram of the final high-level controller

2.1 Individual leg control

2.1.1 Two-dimensional piecewise cycloid step trajectory

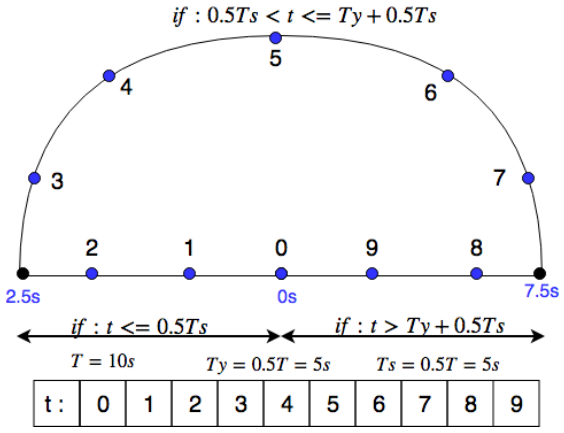
A simple step trajectory generator, based on the composite cycloid trajectory initially proposed in [10], was developed. The trajectory generator depends on a number of parameters that affect the shape, size, and direction of the step. The step consists of a separate stance and a swing phase. The trajectory generator was modified such that the origin corresponds to the *hip* of the robot, and the starting position is in the middle of the stance phase. The piecewise trajectory generator was defined as follows:

$$\begin{array}{ccc} \text{Rh} = \text{Robot height} & \text{H} = \text{Step height} & \text{Sx} = \text{Step length} \\ \text{T} = \text{Total gait period} & \text{Ts} = \text{Stance period} & \text{Ty} = \text{Swing period} \\ t \in [0, T] & \alpha = t - \frac{Ts}{2} & \tau = \frac{\alpha}{Ty} \end{array}$$

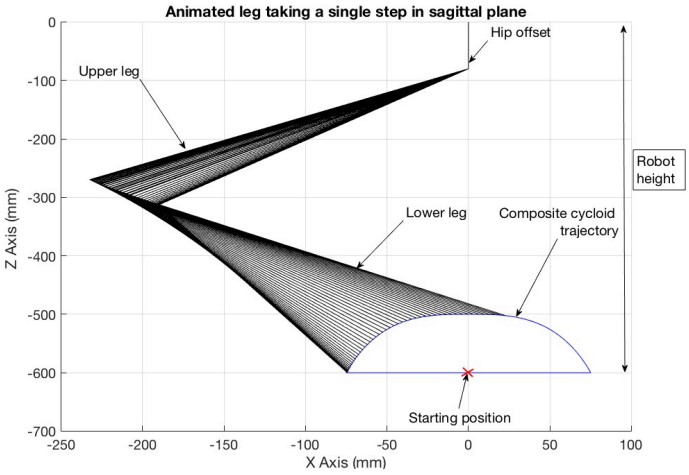
$$z(t) = \begin{cases} -Rh & \text{if } t <= \frac{Ts}{2} \\ \begin{cases} 2H(\tau - \frac{1}{4\pi} sin(4\pi\tau) - 1) - Rh & \text{if } (\alpha <= \frac{Ty}{2} \\ -2H(\tau - \frac{1}{4\pi} sin(4\pi\tau) - 1) - Rh & \text{if } (\alpha > \frac{Ty}{2} \end{cases} & \text{if } \frac{Ts}{2} < t <= Ty + \frac{Ts}{2} \\ -Rh & \text{if } t > Ty + \frac{Ts}{2} \end{cases} \tag{2.1}$$

$$x(t) = sign(Sx) \begin{cases} -\frac{Sx}{Ts} t & \text{if } t <= \frac{Ts}{2} \\ Sx(\tau - \frac{sin(2\pi\tau)}{2\pi}) - \frac{Sx}{2} & \text{if } \frac{Ts}{2} < t <= Ty + \frac{Ts}{2} \\ \frac{0.5Sx}{Ty+0.5Ts-T} + 0.5Sx(1 - \frac{Ty+0.5Ts}{Ty+0.5Ts-T}) & \text{if } t > Ty + \frac{Ts}{2} \end{cases} \tag{2.2}$$

Figure 2.2(a) below is a visual example of this piecewise trajectory generator. It's important to observe how the position in space is dependent on the timing vector, **t**. This property was exploited in order to generate various gaits. Figure 2.2(b) shows a labeled animation of a step.



(a) Visual example of the piecewise step generator with a gait period of 10 s a time increment of 1 s and a swing/stance ratio of 0.5



(b) Labelled animated step diagram

Figure 2.2: Step trajectory diagrams

2.1.2 Three-dimensional trajectory generator and inverse kinematics

The two-dimensional trajectory generator was expanded into a three-dimensional trajectory generator, shown in Figure 2.3, by including a transverse step defined by Equation 2.3. Figure 2.4 shows how the cycloid is maintained in both step directions.

$$y(t) = sign(Sy) \begin{cases} -\frac{Sy}{Ts}t & \text{if } t <= \frac{Ts}{2} \\ Sy(\tau - \frac{sin(2\pi\tau)}{2\pi}) - \frac{Sy}{2} & \text{if } \frac{Ts}{2} < t <= Ty + \frac{Ts}{2} \\ \frac{0.5Sy}{Ty+0.5Ts-T} + 0.5Sx(1 - \frac{Ty+0.5Ts}{Ty+0.5Ts-T}) & \text{if } t > Ty + \frac{Ts}{2} \end{cases} \tag{2.3}$$

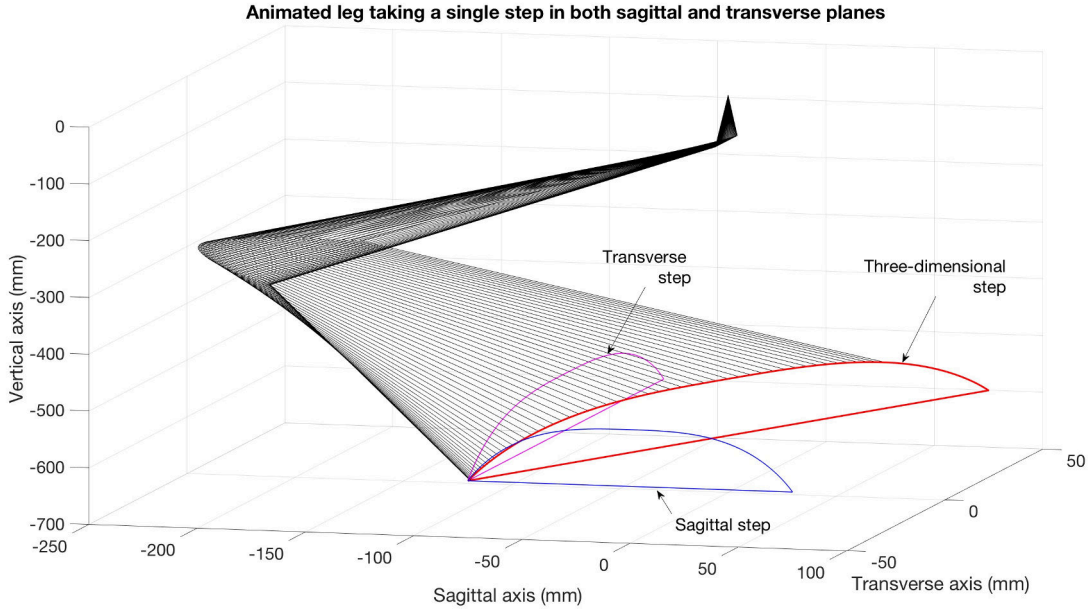
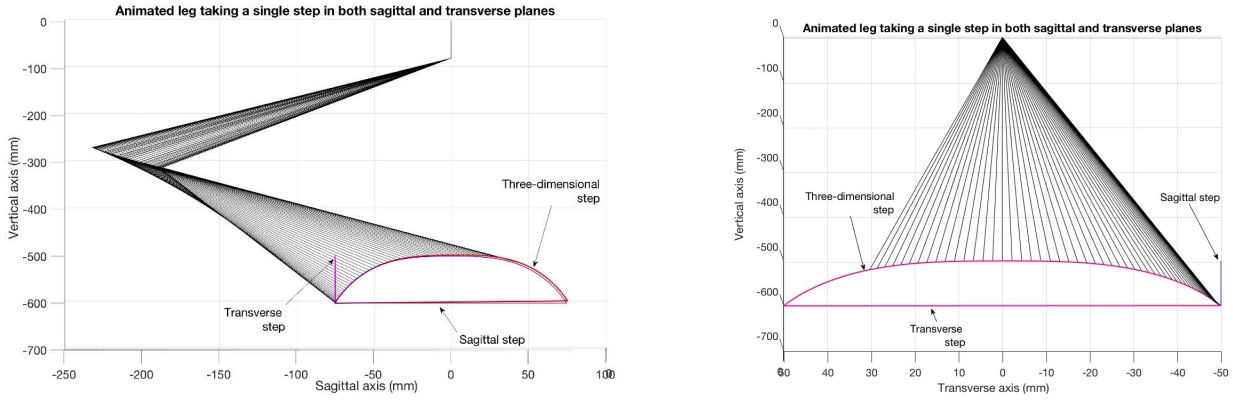


Figure 2.3: Three-dimensional step trajectory



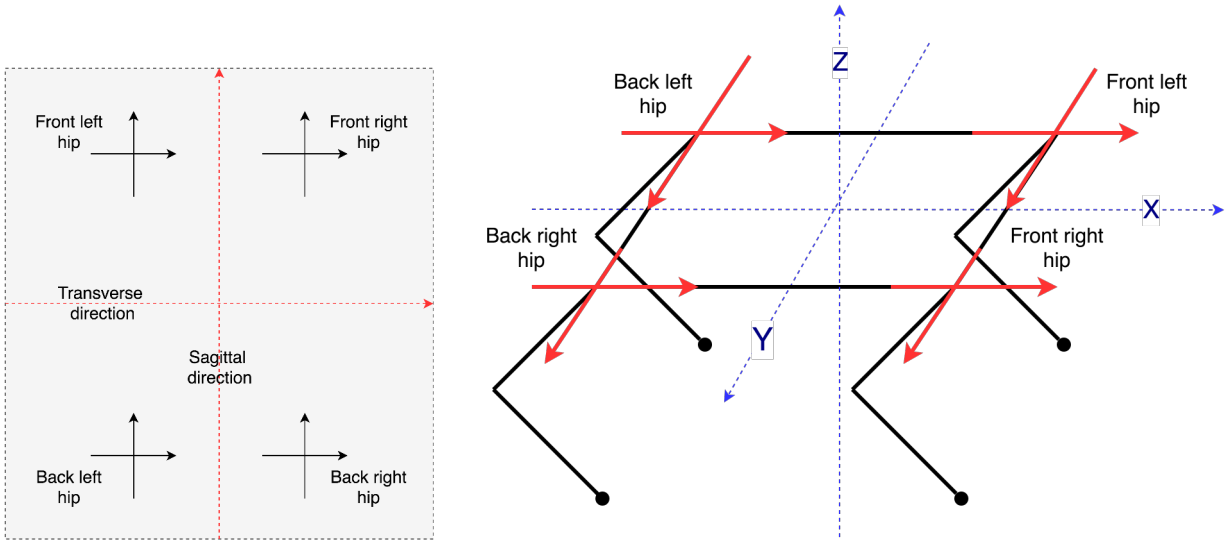
(a) Sagittal plane view

(b) Transverse plane view

Figure 2.4: Side views of a three-dimensional step trajectory

Given a time vector, robot parameters, and step parameters, Equations 2.1, 2.2 & 2.3 now define the position (x, y, z) of a *foot* during a step, relative to the respective hip joint of the leg. This allows a step in any direction to be generated, however, a convention was required. Figure 2.5 shows the convention

used through the design. The advantage of such a convention is that a signed step vector results in the same step for each leg.



(a) (b)

Figure 2.5: Sign convention showing how the origin of each leg lies at the hip.

Three-dimensional inverse kinematics equations were then used to determine the angles required to follow a trajectory. Figure 2.6 shows the trigonometric diagrams used to calculate the inverse kinematic equations which start at Equation 4.

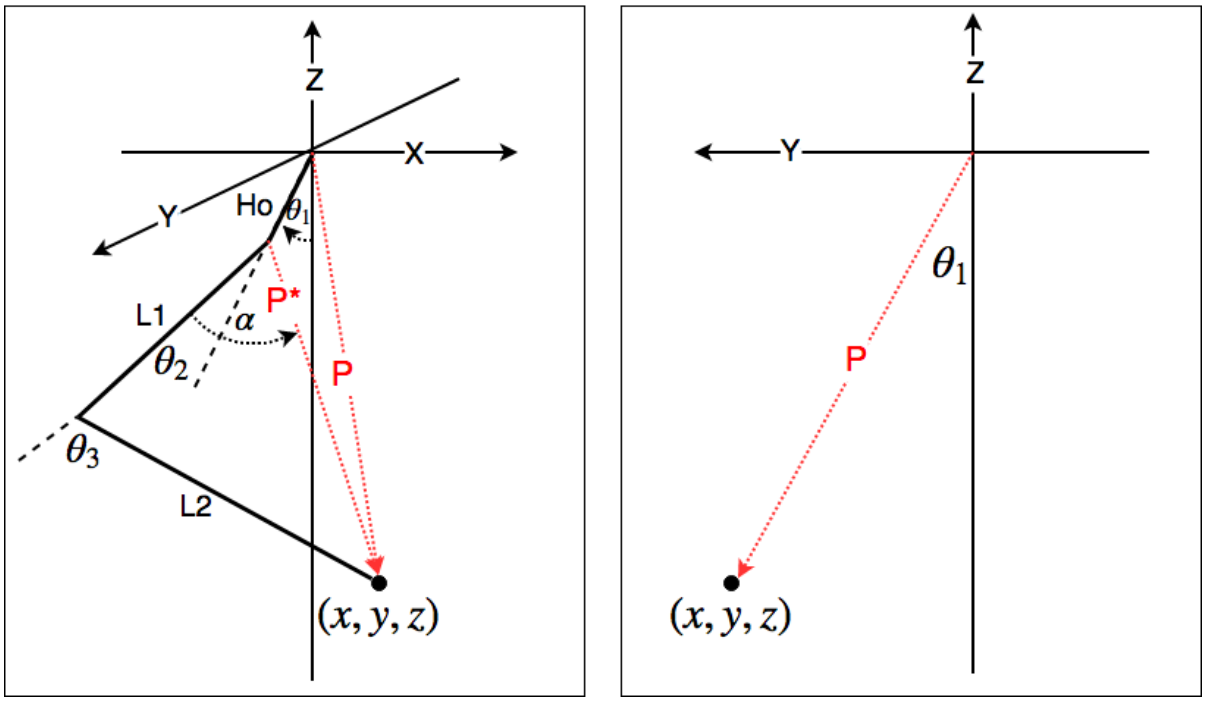


Figure 2.6: Trigonometric diagrams used to calculate inverse kinematics equations

$$P^* = \sqrt{x^2 + (y - Ho(sin(\theta_1)))^2 + (z + Ho(cos(\theta_1)))^2} \tag{2.4}$$

$$P = \sqrt{x^2 + y^2 + z^2} \tag{2.5}$$

$$\alpha = cos^{-1}\left(\frac{(P^*)^2 + L_1^2 - L_2^2}{2L_1P^*}\right) \tag{2.6}$$

$$\boldsymbol{\theta_1} = sin^{-1}\left(\frac{Y}{P}\right) \tag{2.7}$$

$$\boldsymbol{\theta_2} = tan^{-1}\left(\frac{-P^*}{x}\right) - \alpha \tag{2.8}$$

$$\boldsymbol{\theta_3} = cos^{-1}\left(\frac{(P^*)^2 - L_1^2 - L_2^2}{2L_1L_2}\right) \tag{2.9}$$

2.2 Gait generation

Using the step trajectory generator and inverse kinematics solutions, a gait generator was developed. A gait graph method, similar to the one proposed in [9], was used. The gait graph method assumes all four legs are simultaneously cycling through step trajectories. The various gaits are then defined by timing offsets between the four legs. Figure 2.7 shows the gait graphs used for generating the trotting and walking gaits. The timing offsets are shown for the walking gait, these offsets are used to generate individual timing vectors for each leg, which determine the position of the respective foot at a given time.

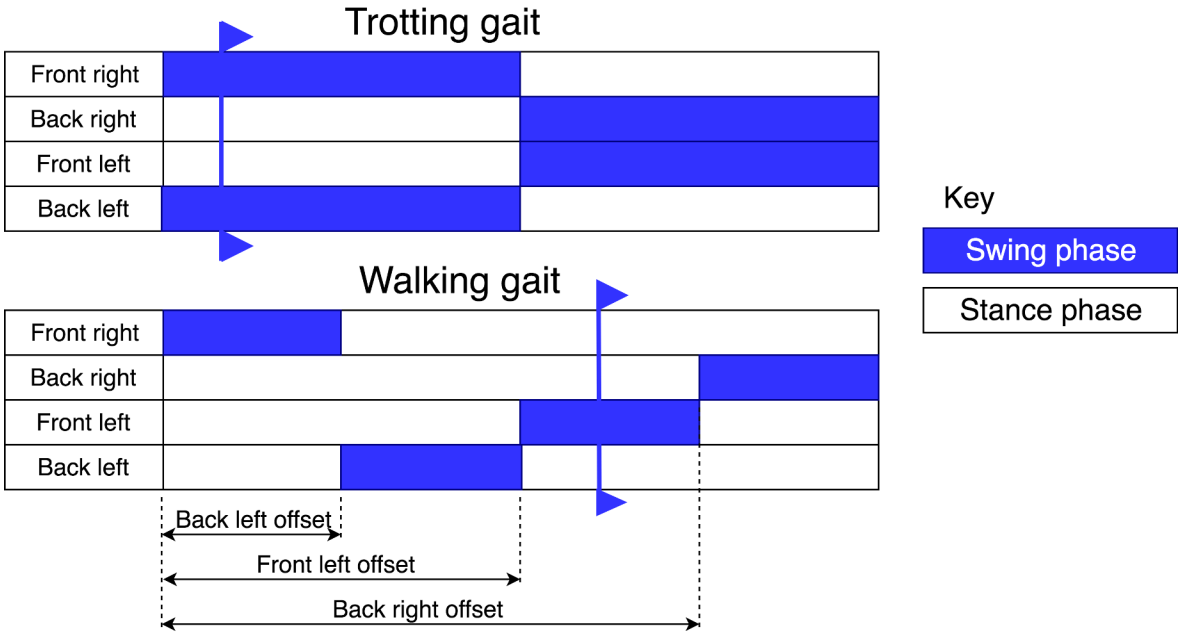
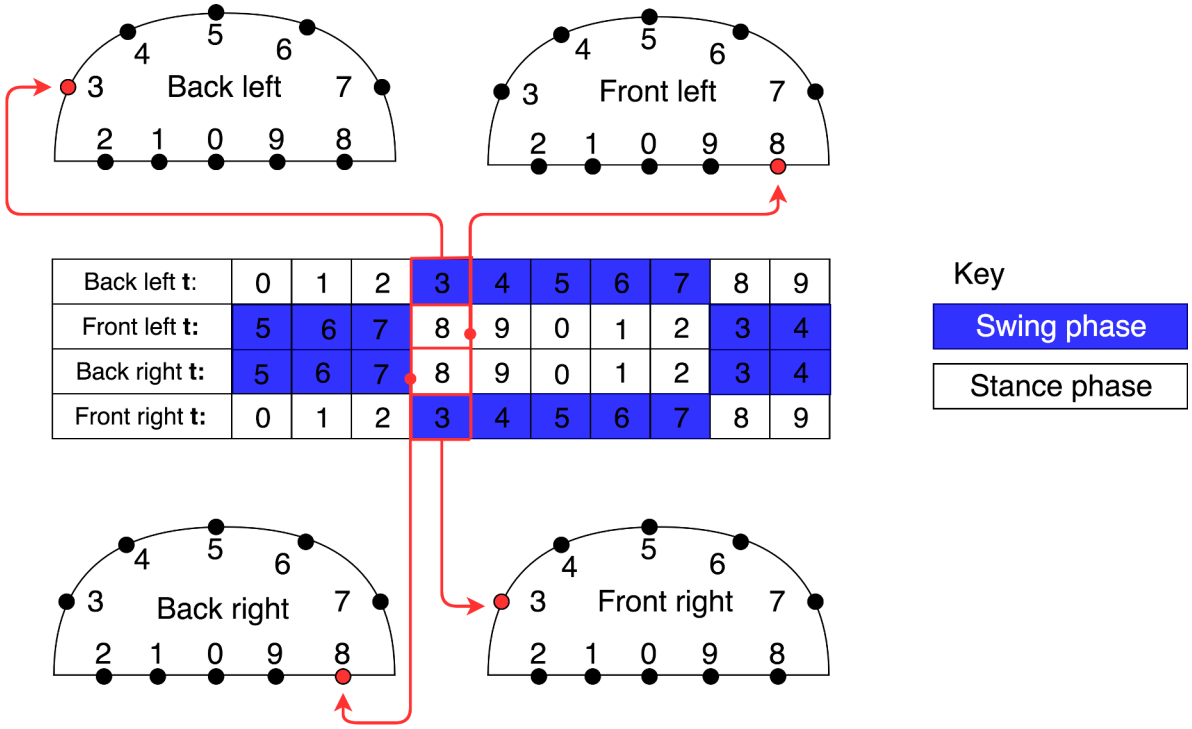


Figure 2.7: Timing-based gait graphs for walking and trotting gaits

Figure 2.8 is an example of how a gait graph could be used to generate a trotting gait. Superimposing correctly-offset timing vectors onto the gait graph allows the simple case to be observed. Unrealistic timing parameters, the same as those chosen in Figure 2.2(a), have been used to simplify the example. At the time of interest, (indicated with a red box), both the front right and back left legs have a $\boldsymbol{t}$ value of 3 s, and are about to initiate the swing phase. The front left and back right legs have a $\boldsymbol{t}$ value of 8 s, and are about to initiate the stance phase.



Back left **t:**	0	1	2	3	4	5	6	7	8	9
Front left **t:**	5	6	7	8	9	0	1	2	3	4
Back right **t:**	5	6	7	8	9	0	1	2	3	4
Front right **t:**	0	1	2	3	4	5	6	7	8	9

Figure 2.8: Simplified example of a trotting gait graph and timing diagram superposition

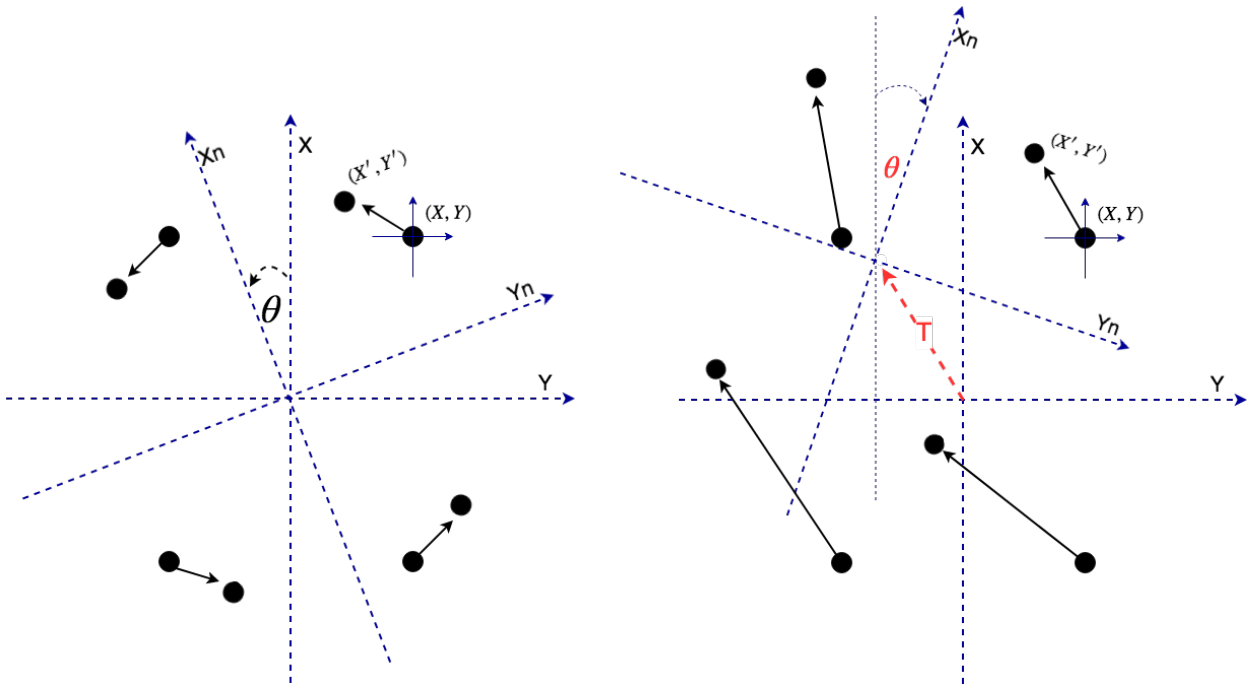
2.3 Rotation and translation step vector mapping

At this stage, although the robot could step in any direction, the motion and direction of the actual robot was not yet controlled. In order to do so, a function mapping body rotations(θ) and translations (T) to individual footstep vectors was developed. Figure 2.9 shows the diagrams used to develop this function. The black circles represent the respective feet of the robot, before and after the steps have been taken, with their positions defined relative to the center of the robot. The large dashed arrows represent the axis of the robot, before (X, Y) and after (Xn, Yn) a body rotation and/or translation. Finally, the black arrows represent the step vectors from the current to the future feet position, required to perform the body rotation and/or translation. The calculation mapping a translation and rotation to a footstep vector for a single foot is as follows:

$$\begin{bmatrix} Step_y \\ Step_x \end{bmatrix} = \left(\begin{bmatrix} y \\ x \end{bmatrix} \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} + \begin{bmatrix} T_y \\ T_x \end{bmatrix} \right) - \begin{bmatrix} y \\ x \end{bmatrix} \tag{2.10}$$

Figure 2.10 shows the data flow from the step and rotation calculator through the trajectory generator, and inverse kinematics calculator to the final angles required to move the robot.

In order to test the developed functions, various parameters were mapped to an Xbox remote control. The parameters can be seen in Table 2.1. A labeled diagram of the Xbox remote control can be seen



(a) Pure rotation

(b) Rotation and translation

Figure 2.9: Rotation and step translation diagrams

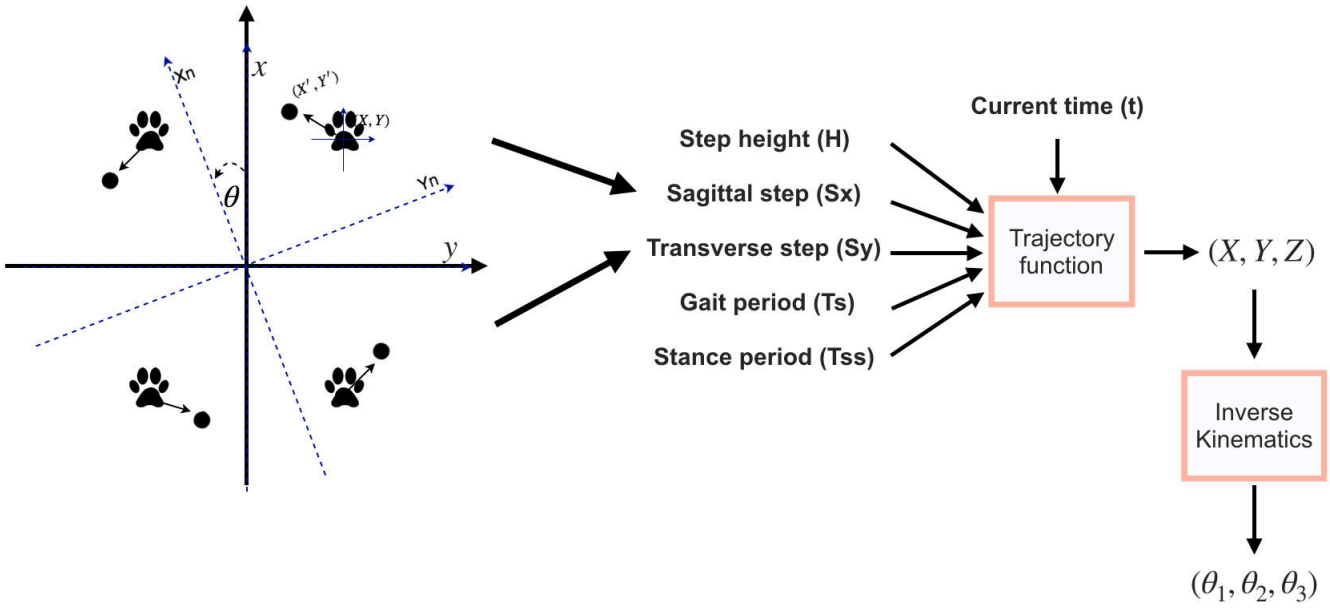


Figure 2.10: Step vector information flow

in Figure 2.11. Please see Figure B.1, a flow diagram, in the Appendix for a further explanation of the software implementation of this function.

Xbox command	Robot parameter
Sagittal step	T_x
Transverse step	T_y
Body yaw rotation	θ
Robot height control	Rh
Break control	$Sx, Sy, H = 0$

Table 2.1: Table showing robot parameter Xbox mapping

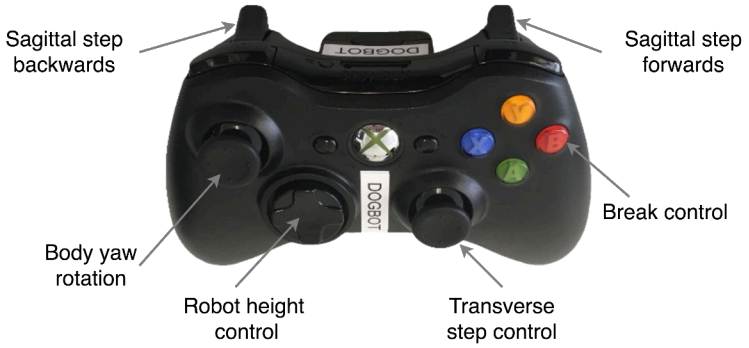
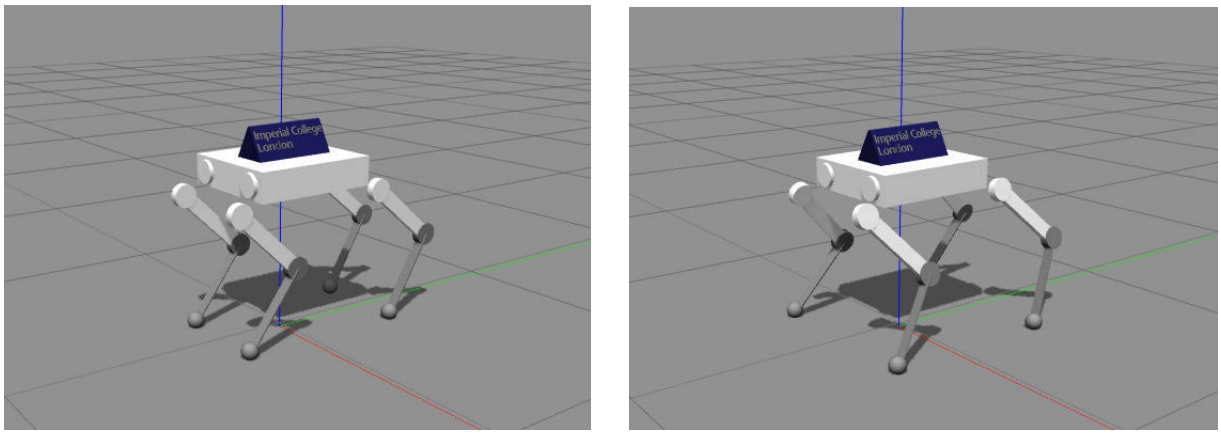


Figure 2.11: Labelled diagram of Xbox remote control mapping

2.4 Yaw feed forward term

In order to improve the stability of the robot, a transverse step offset was added: Figure 2.12. Although this improved the stability, it caused yaw to oscillate between gait cycles. Each time the trotting leg pairs made contact with the ground, a slight moment was generated causing the robot to rotate.



(a) Zero transverse step offset

(b) Transverse step offset of 130 mm

Figure 2.12: Gazebo model with various transverse step offsets

These oscillations, generated as the trotting leg pairs transition from swing to stance phase, can be seen in Figure 2.13. In order to solve this problem, a yaw feedforward term - acting in opposite directions during the two stance phases - was included in the step and rotation calculator (Figure 2.14).

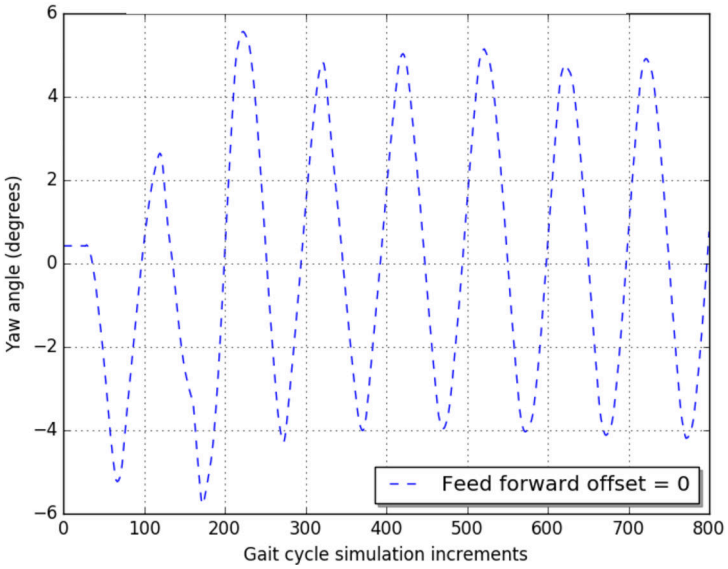


Figure 2.13: Yaw angle oscillations over eight trotting gait cycles, with no feed forward correction term

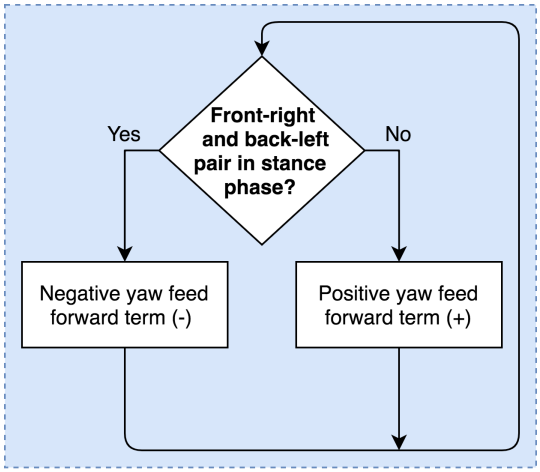
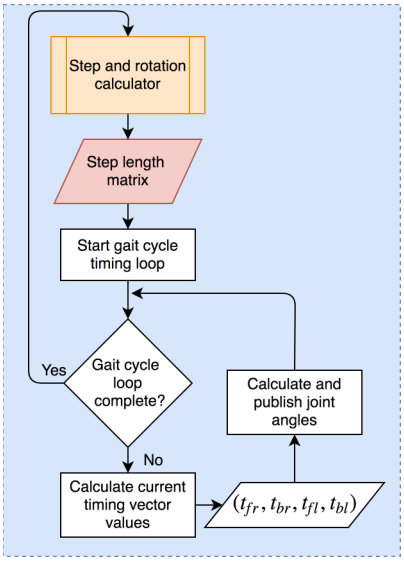


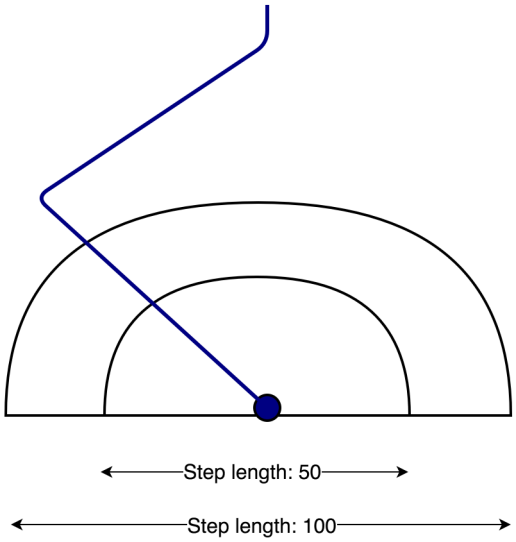
Figure 2.14: Feed forward yaw term algorithm

2.5 Step interpolation

During the initial testing of the controller, gait and step parameters - such as step vectors (which were contained in the step length matrix), step height, and robot height - were only changed between gait cycles. This ensured continuity between gait cycles and step trajectories, however, reduced the responsiveness of the controller. This algorithm can be seen in Figure 2.15. Simply changing step and gait parameters within gait cycles, without interpolation, resulted in large discontinuities between step trajectories, and very jerky motion. This can be seen in Figure 2.16.

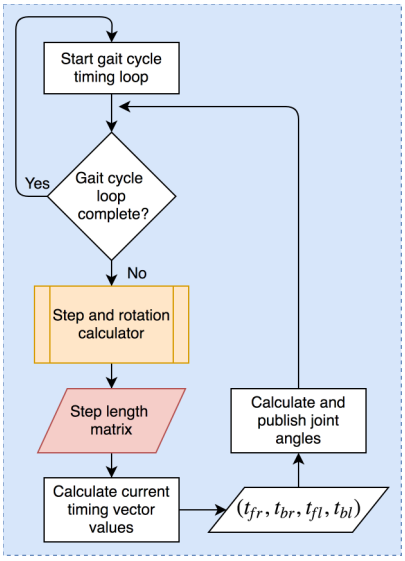


(a) Algorithm changing gait and step parameters between gait cycles

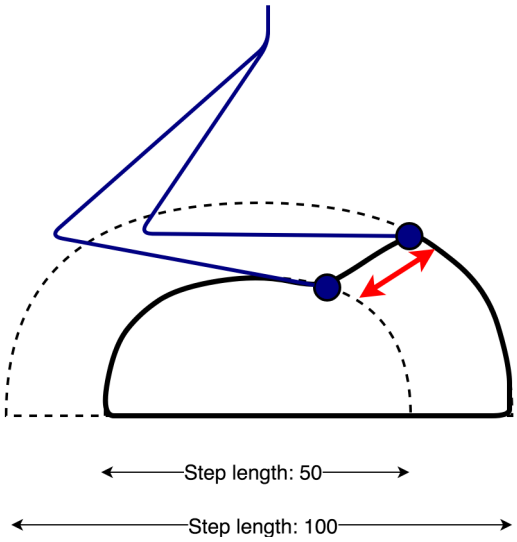


(b) Visual diagram of increasing the step length and height between gait cycles

Figure 2.15: Step and gait parameter changes between gait cycles



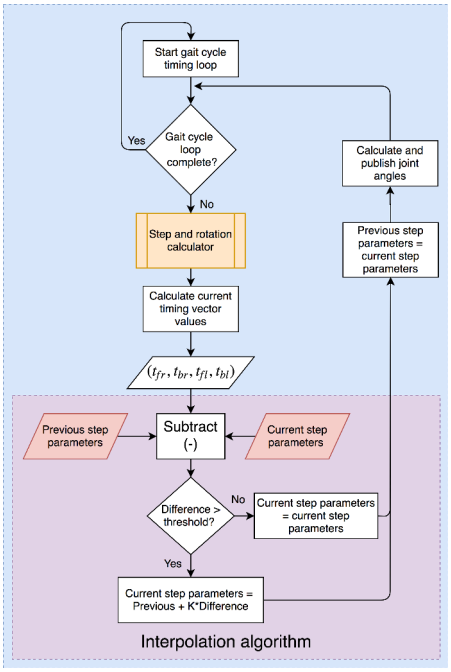
(a) Algorithm changing gait and step parameters within gait cycles



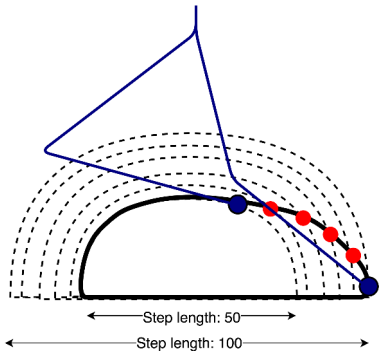
(b) Visual diagram of step length and height increase within gait cycles

Figure 2.16: Step and parameter changes within gait cycles

In order to solve this problem, a simple step interpolation algorithm was developed. The algorithm (Figure 2.17(a)) compares the modulus of the difference between the previous and current step parameters. If the difference of any parameter is greater than a defined threshold, the new parameter is incremented towards the desired step parameter within the gait cycle. This works by creating multiple small step parameter changes that do not cause jerky motion. This results in a smooth trajectory transition profile and increased responsiveness of the controller. An example diagram can be seen in Figure 2.17(b).



(a) Step parameter interpolation solution algorithm



(b) Step parameter interpolation solution example

Figure 2.17: Step parameter interpolation solution

2.6 Attitude control

In order to obtain feedback information, allowing the inference of the robot's orientation, an IMU (inertial measurement unit) was added to both the simulated model and the physical robot. The IMU was used to implement two separate general controllers, as well as a push recovery method. Standard ROS PID controller nodes were used, these controllers required set-points and states as inputs, and output the necessary control effort.

2.6.1 Yaw controller

The PD yaw controller was designed to operate in two different modes: manual and autonomous. In manual mode, when using the Xbox remote control, the robot is rotated until the required angle is reached. Once the position is reached, the controller is activated, the current angle is saved as the set-point, and the yaw is used as the robot state. This holds the robot in position until a new command is received. This algorithm can be seen in Figure 2.18. In autonomous mode, the set-point angle is predefined and control effort is supplied until the set-point is reached.

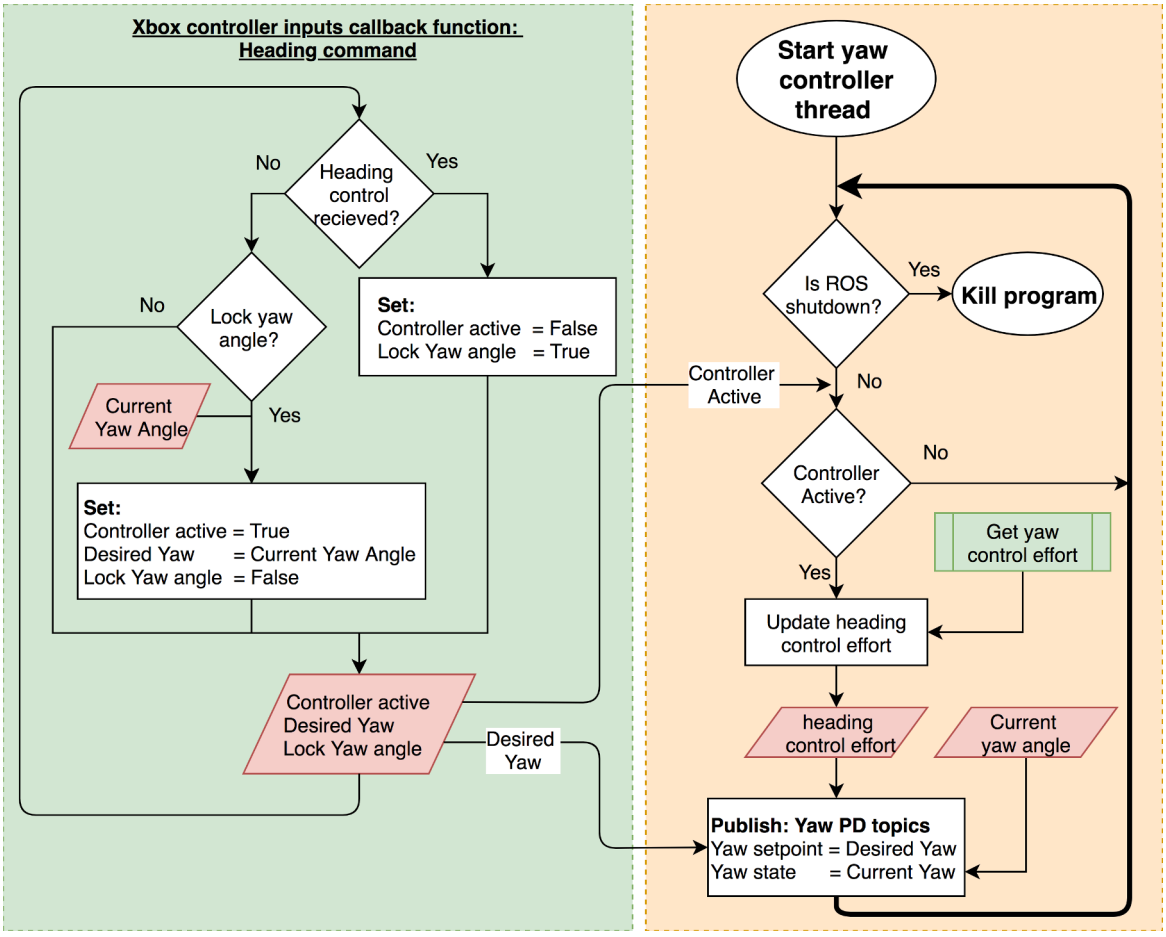


Figure 2.18: Flow diagram of the yaw controller

2.6.2 Roll and pitch controller

The roll and pitch controller was initially designed to improve stability when traversing inclined, declined, rough and uneven terrain. Traversing uneven terrain with all legs the same height causes the robot to roll and pitch. This shifts the center of mass, which destabilizes the robot, often resulting in it falling over.

By individually controlling the height of each leg, the roll and pitch of the robot was minimized, resulting in improved stability. In order to achieve this, each leg was assigned a control node and the height of the hip relative to a horizontal reference (initially proposed in [11]) was controlled. This reference frame can be seen in Figure 2.19. The set-point was set to zero, the state was defined as (ΔZ) (the relative height of the hip) and the controlled effort was used to adjust the leg height. Figure 2.20(a) shows how ΔZ is defined under a pure roll case, and Figure 2.20(b) shows the robot after it has been stabilized.

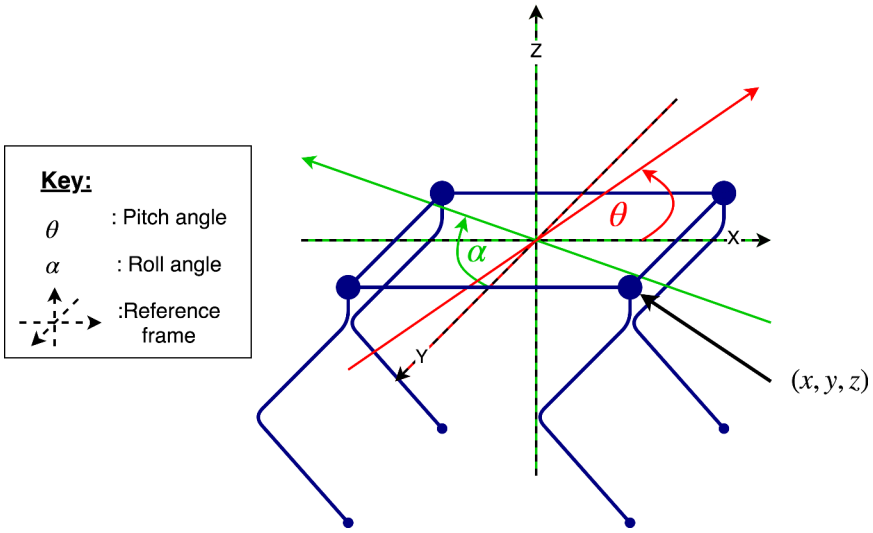
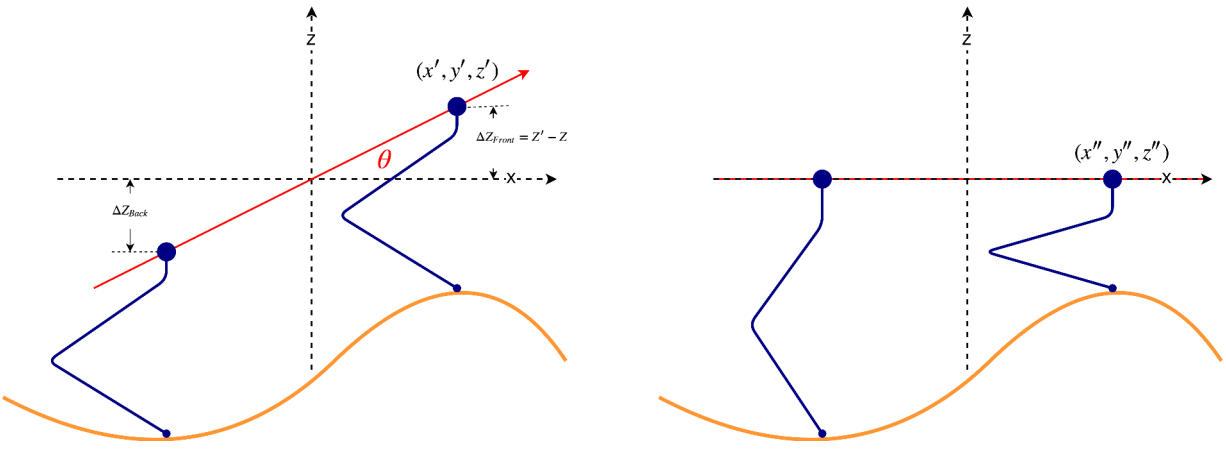


Figure 2.19: Diagram showing the roll, pitch and horizontal reference frame



(a) DogBot experiencing a pure pitch due to uneven terrain

(b) DogBot, after the roll and pitch controller has been used to stabilize the robot

Figure 2.20: Side view diagrams of DogBot traversing uneven terrain

The ΔZ for a single leg was calculated as follows, where (x, y, z) represents the coordinates of the hip with zero roll or pitch.

$$\begin{bmatrix} y' \\ x' \\ z' \end{bmatrix} = \begin{bmatrix} cos(\theta) & 0 & -sin(\theta) \\ 0 & 1 & 0 \\ sin(\theta) & 0 & cos(\theta) \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\alpha) & sin(\alpha) \\ 0 & -sin(\alpha) & cos(\alpha) \end{bmatrix} \begin{bmatrix} y \\ x \\ z \end{bmatrix} \tag{2.11}$$

$$\Delta Z = z' \tag{2.12}$$

Figure 2.21 shows how the roll and pitch controller thread is implemented in software. One final addition to the thread was the ability to detect flat ground or constant slopes/steps. These two states were detected by comparing the individual legs height as well as the average leg height. If flat ground was detected, all leg heights were reset to the robot height. If a constant slope or step was detected, the front and back legs were set to the respective front and back average heights.

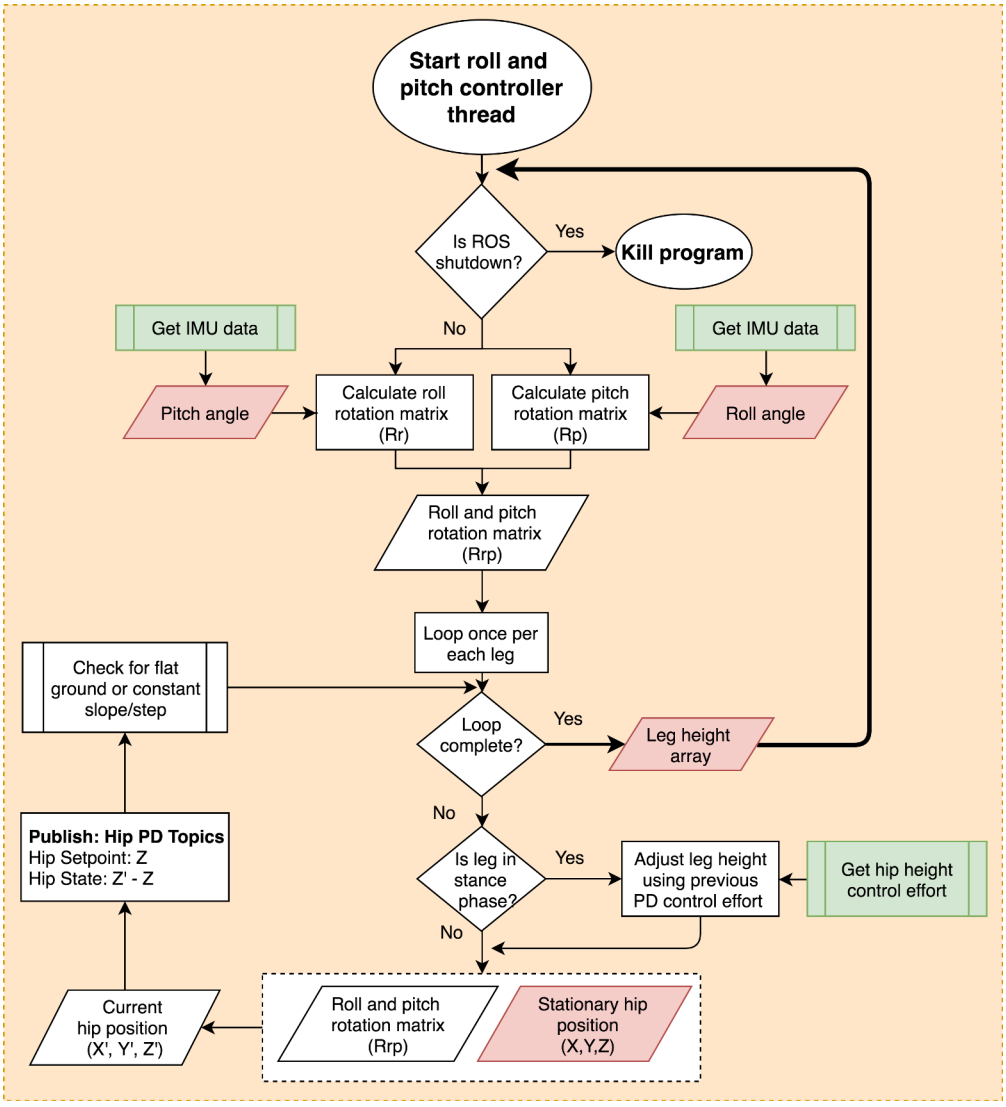
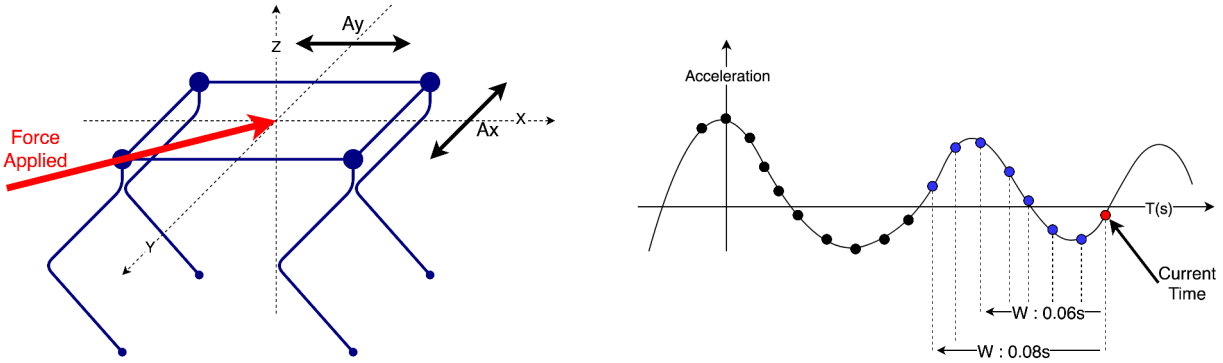


Figure 2.21: Software implementation of the roll and pitch controller thread

2.7 Push recovery

In order to improve the robustness and perturbation resistance of the controller, a push/force detector was developed using the linear accelerations linear accelerations (Ax, Ay) from the IMU (Figure 2.22(a)). Simply analysing instantaneous linear acceleration proved unsuccessful as due to the dynamic nature of the system - the signals contained significant noise, making it difficult to define threshold detection values.

Sliding window averages of the accelerations were instead computed and used for detection. The sliding window average works by computing the average acceleration over a time window period. Figure 2.22(b) is a simplified visual example of how the average acceleration windows of 0.06 s and 0.08 s would be calculated. Assuming an IMU publish rate of 100 hz, for a window width of 0.06 s the previous 6 accelerations would be averaged, while for a window width of 0.08 s the previous 8 acceleration values would be averaged.



(a) Digram showing theoretical force application and lateral accelerations

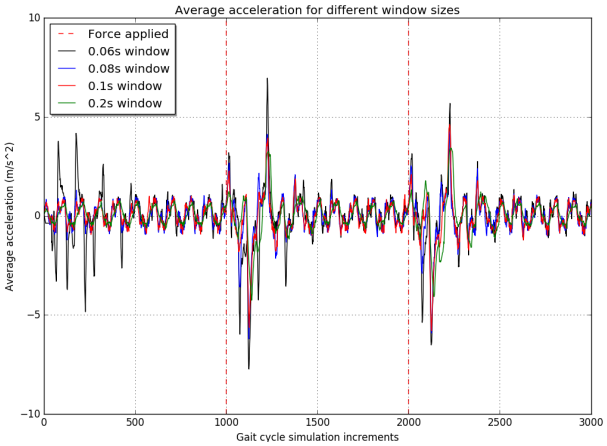
(b) Example of the sliding window average acceleration detection

Figure 2.22: Force detector diagrams

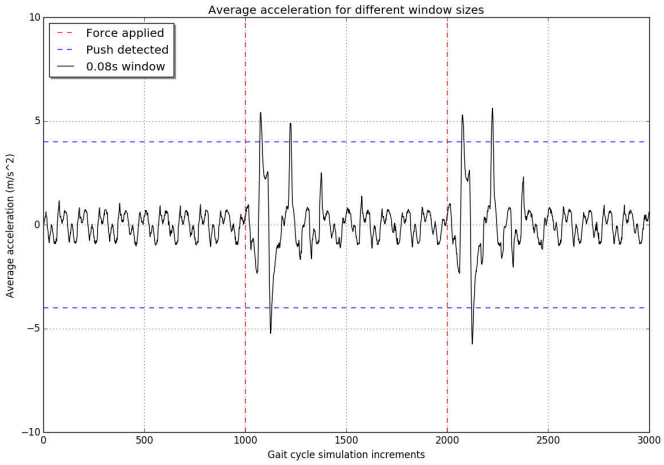
The force detectability for a number of different window sizes were compared in order to determine the most suitable size. Figure 2.23(a) shows how the window sizes were compared. The horizontal axis shows the gait cycle simulation increments, where a full gait cycle takes 100 increments. The vertical axis shows the average window acceleration.

In Figure 2.23(a), both 0.06 s and 0.08 s windows clearly showed detectable acceleration spikes. However, 0.06 s seemed slightly too sensitive, with a larger average window amplitude. Thus, 0.08 s was chosen as the most suitable window size, and a force detection acceleration threshold was defined as $4\,m/s^2$. This is shown in Figure 2.23(b).

With the push/force detection in place, different defence tactics were experimented with. One of the more successful strategies involved stepping in the direction opposing the force - by an amount proportional to the force - increasing the transverse step offset and reducing the height of the robot. These robot parameters were maintained for 0.5 s, thereafter resetting the default parameters (Figure 2.24).



(a) Graph showing average acceleration for different window sizes for 30 gait cycles, with applied forces at 10 and 20 gait cycles.



(b) Graph showing average acceleration for a window size of 0.08 s, with force detection threshold defined at $4\,m/s^2$. In this example, both forces are clearly detected.

Figure 2.23: Average acceleration window graphs

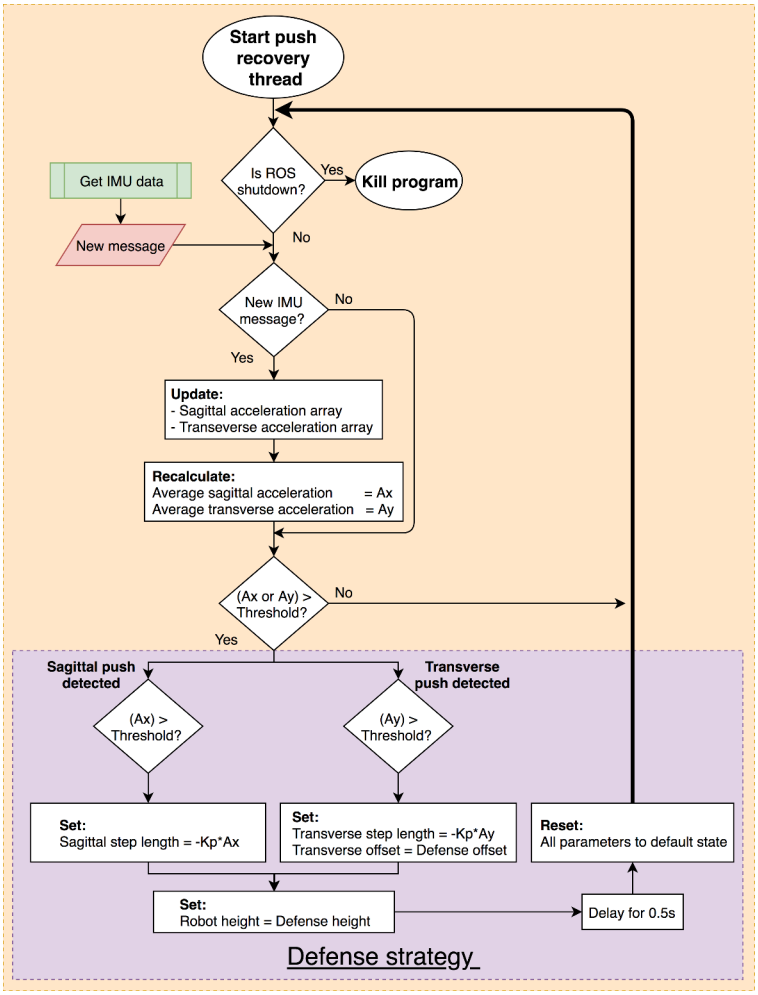


Figure 2.24: Push recovery algorithm including the force detection and the defence strategy

2.8 Bayesian optimization hyper-parameter search

Increasing the complexity of the controller resulted in a severe increase in the number of hyper-parameters. It was noticed how slight changes of certain parameters significantly affected the performance of the controller. Due to the number of hyper-parameters and possible combinations, manual tuning is simply not feasible. This scenario is well suited to Bayesian optimization hyper-parameters search, as extracting data points is highly expensive. The hyper-parameters to be optimized included:

Step length	Step height	Robot Height	Gait period
Yaw Kp	Yaw Kd	RollPitch Kp	Roll Pitch Kd
Transverse step offset	Sagittal step offset	Gait	Swing period

In order to implement a Bayesian optimization hyper-parameter search, a *function* was required. The function was defined as a particular simulated task to be optimized. The inputs to the function were defined as the hyper-parameters to be optimized, while the output of the function was a custom-define reward, based on how successfully the robot completed the task. In this way, the robot could *learn* the best parameters for a particular task. Figure 2.25 shows the basic algorithm used to optimize the hyper-parameters for a particular task.

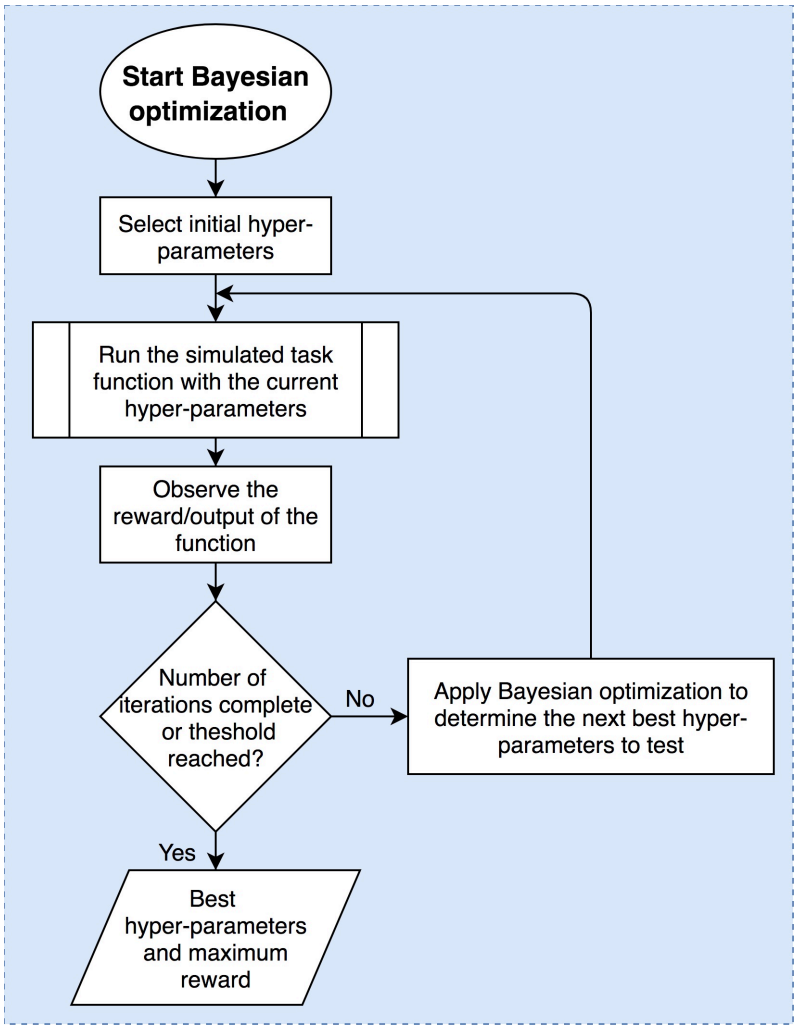


Figure 2.25: Bayesian optimization algorithm

2.9 Overview of complete controller

Referring back to the overall controller flow diagram shown in Figure 2, it is now clear how each process has been designed and how the various processes interact with one another. The main controller elements have been assigned individual threads that allow them to be turned on and off as required.

Figure 2.26 is an example of a ROS rqt graph for the entire controller. In this example, the robot is being controlled manually, using the Xbox remote control. Here it can be seen how the DogBot name-space contains one topic per joint-angle of the robot. Thus, in order to control the robot - using the ROS controller developed by React Robotics - the calculated joint angles must be published to the respective joint-angle topics.

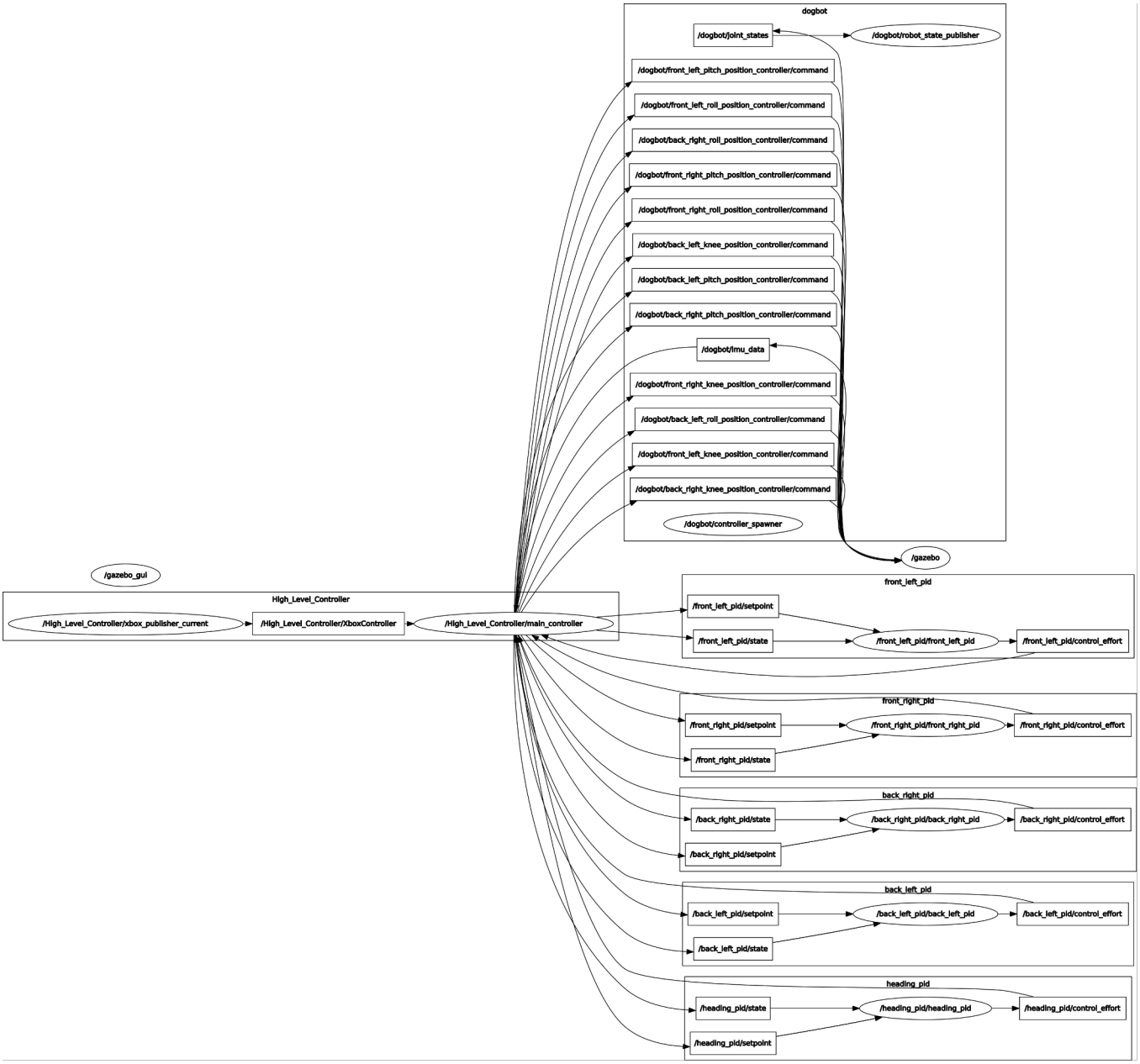


Figure 2.26: ROS rqt node/topic graph

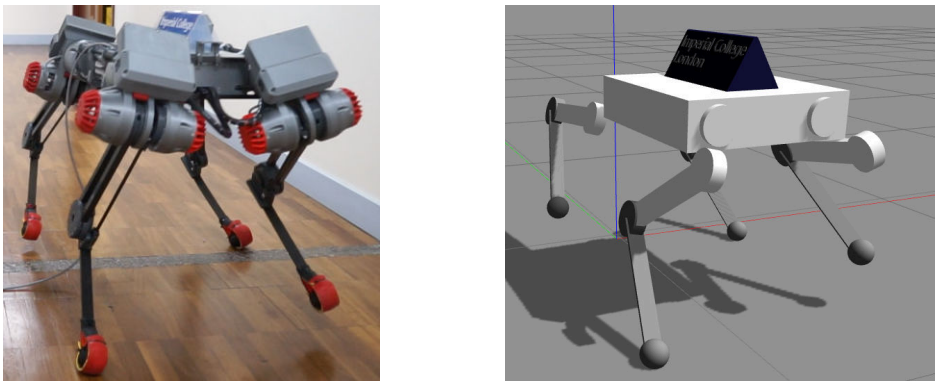
Chapter 3

Results

Testing was extensively conducted on the simulated model of the robot, throughout the project. Unfortunately, as the physical robot was an earlier iteration of the design, some mechanical problems were experienced, limiting the physical robot testing time. Only the open loop controller (Without the IMU) was briefly tested on the physical robot.

3.1 Gait generation

The controller was able to generate both walking and trotting gaits in the simulated robot. These two gaits were also tested on the physical robot, while suspended in the air, and while unsupported. While suspended, the physical robot elegantly demonstrated both walking and trotting gaits. While unsupported, the robot was able to generate both gaits, however, it was unstable and required further parameter tuning. Figures 3.1 and 3.2 compare the trotting gait of the simulated and physical robots. Although it appears both the simulated and physical robot are successfully trotting, this was not always the case. The simulated robot was far more consistent and stable, often the physical robot became unstable and therefore struggled to maintain synchronicity. Weight shifting during the dynamic trotting gait prevented swing pairs from being able to lift simultaneously (Figure 3.3(a)). This problem was improved by adding a saggital step offset however. However, defining the correct offset required more testing time.

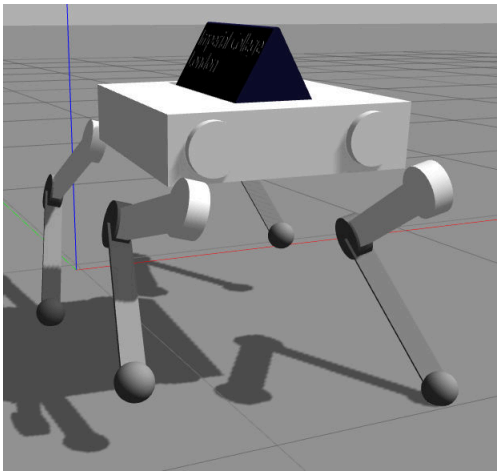


(a) Physical robot

(b) Simulation of the robot

Figure 3.1: Front left and back right legs in stance phase during trotting gait

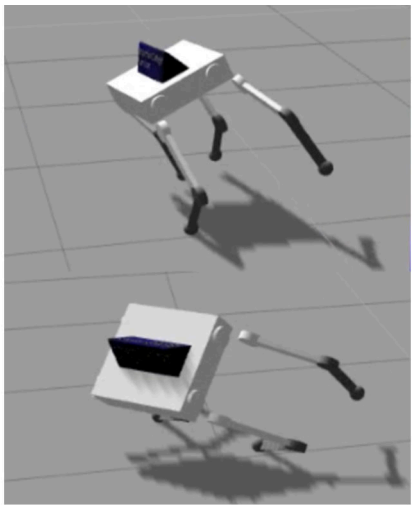
(a) Physical robot



(b) Simulation of the robot

Figure 3.2: Front right and back left legs in stance phase during trotting gait

(a) Example loss of gait synchronicity



(b) Example of the simulated robot falling while rotating at high speeds

Figure 3.3: Robot failure instances

3.2 Rotation, translation and manoeuvrability

In simulation, the controller was efficacious: both rotation and translation were tested autonomously and with the Xbox remote control. Although the robot was very manoeuvrable, without feedback control it occasionally became unstable and fell, especially when turning at relatively high speeds, Figure 3.3(b). Figure 3.4 shows the simulated robot successfully rotating anti-clockwise.

The physical robot performed in a similar manner, it was able to rotate and translate in any direction, however - as with the gait testing - it was not completely stable and required significant further parameter tuning. Figure 3.5 shows the physical robot rotating clockwise.

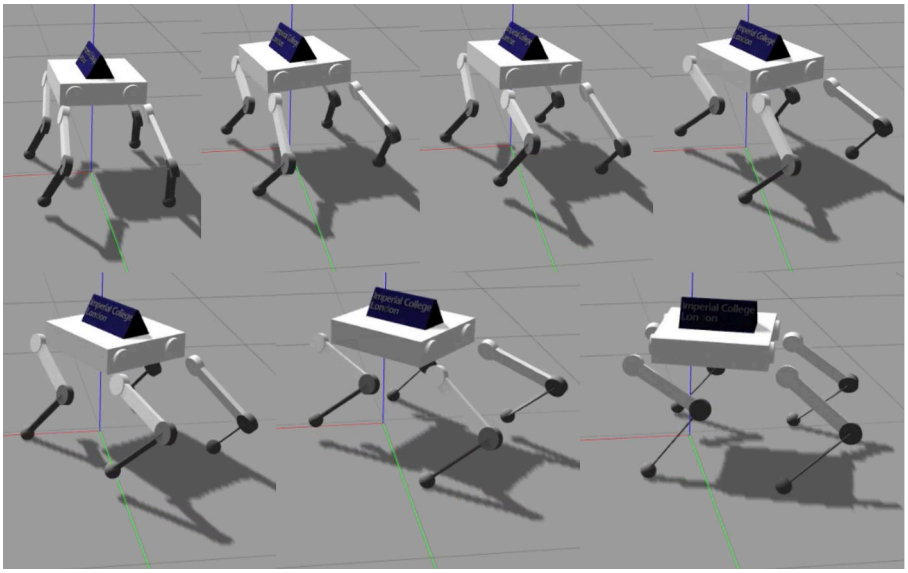


Figure 3.4: Anti-clockwise rotation sequence, starting at the top left, of the simulated robot

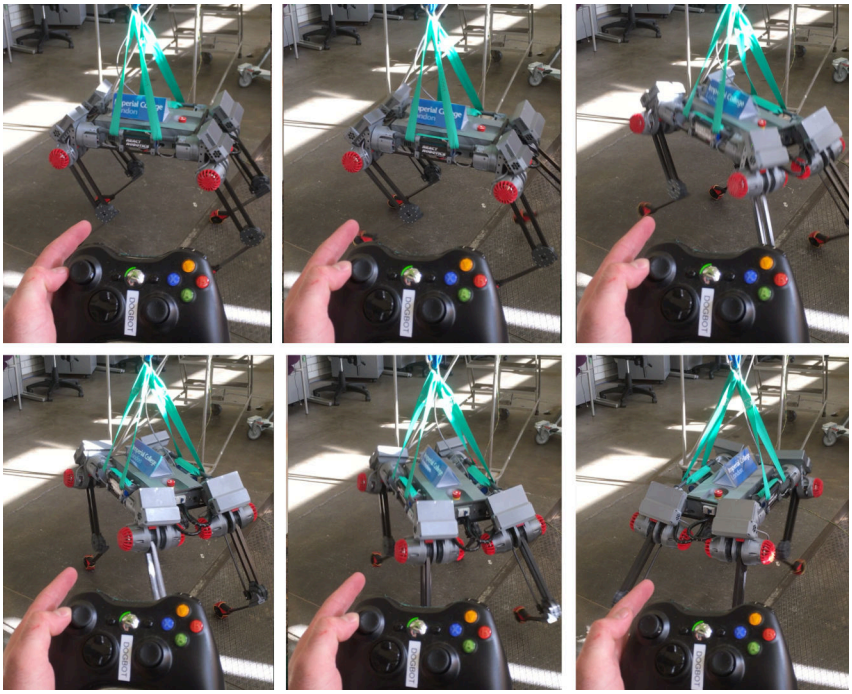


Figure 3.5: Clockwise rotation sequence, starting at the top left, of the real robot

3.3 Step interpolation

The step interpolation method was implemented as jerky and loud noises were noticed when changing step parameters of the physical robot within gait cycles. The step interpolation radically improved this. Once it was implemented, the robot's motion was smooth and step parameter changes caused no additional noise. Figure 3.6 shows an example of a major sagittal step change.

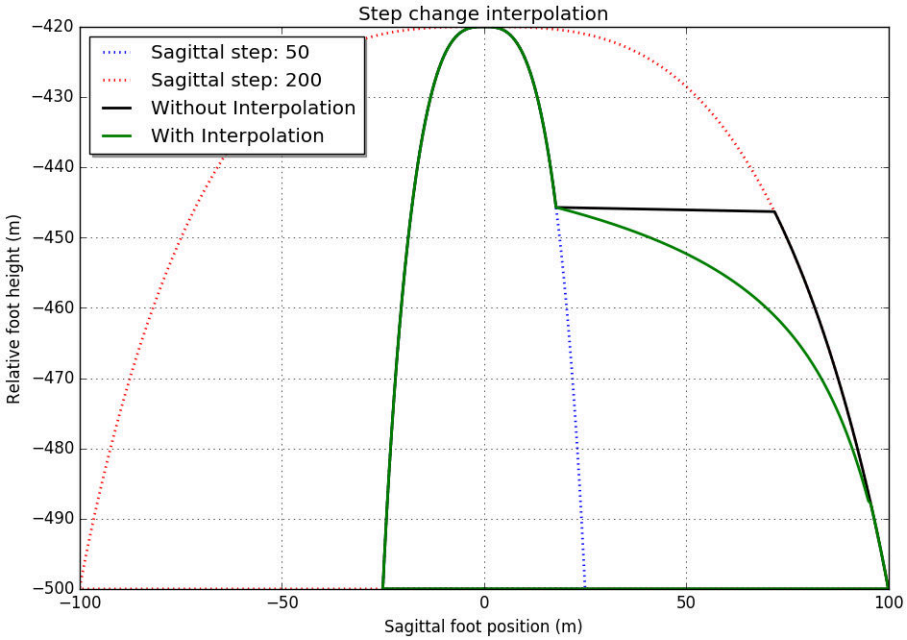
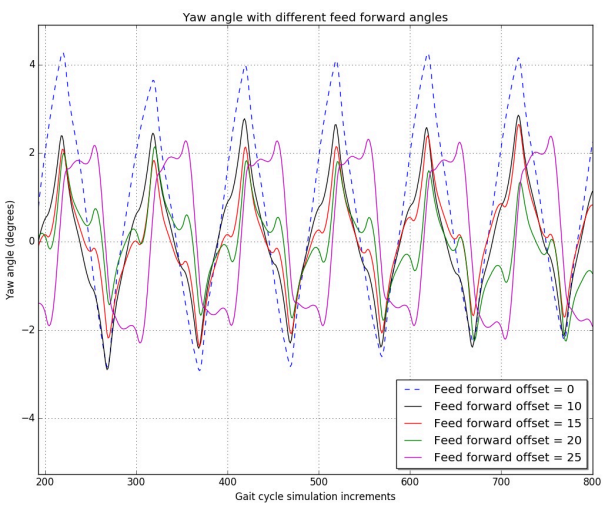


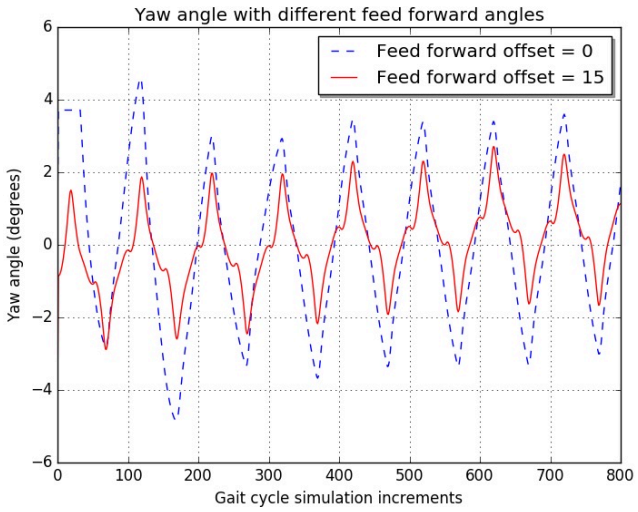
Figure 3.6: Graph showing sagittal step length change from 50 cm to 200 cm within a gait cycle

3.4 Yaw controller

The yaw controller consisted of a feed forward term, intended to reduce yaw oscillations, as well as a feedback controller to maintain position. Figure 3.7 shows the yaw oscillations with various different feed forward term values.



(a) Yaw oscillations with various different feed forward terms

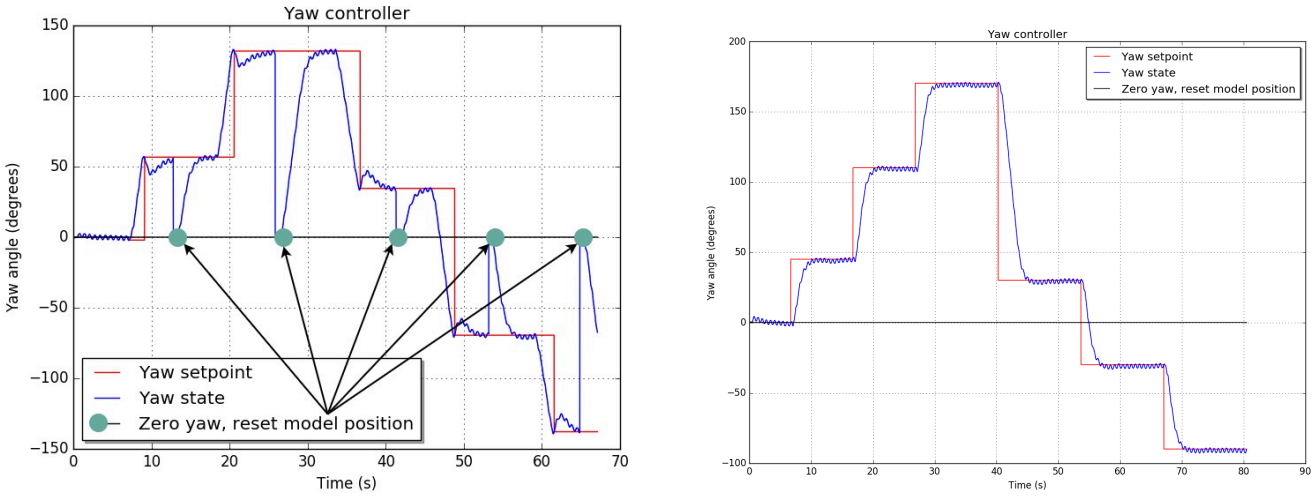


(b) Yaw oscillations with the best feed forward term of 15°, which reduced the oscillations by 2°

Figure 3.7: Resulting effect of various different yaw feed forward terms

Figure 3.8 shows the two different modes of the yaw feedback controller. Figure 3.8 (a) is an example of the manual feedback controller: as expected, the yaw state leads the set-point. The feedback controller is turned off while an Xbox yaw command is being received, the desired position is then saved and the controller is turned on, holding the robot in-place. This was tested by resetting the model position

to 0°, using Gazebo, during each saved position. The controller successfully returns the robot to the desired, position demonstrating its effectiveness. Figure 3.8 (b) is a conventional example of the controller in autonomous mode: desired positions are defined and the controller provides effort to reach these positions.



(a) Manual, Xbox yaw feedback controller mode

(b) Autonomous yaw feedback controller mode

Figure 3.8: Resulting effect of various different yaw feed forward terms

3.5 Roll and pitch controller

The roll and pitch controller was tested on a number of different terrains and uneven surfaces. Figure 3.9 shows a model that was designed specifically to test the controller in simulation.

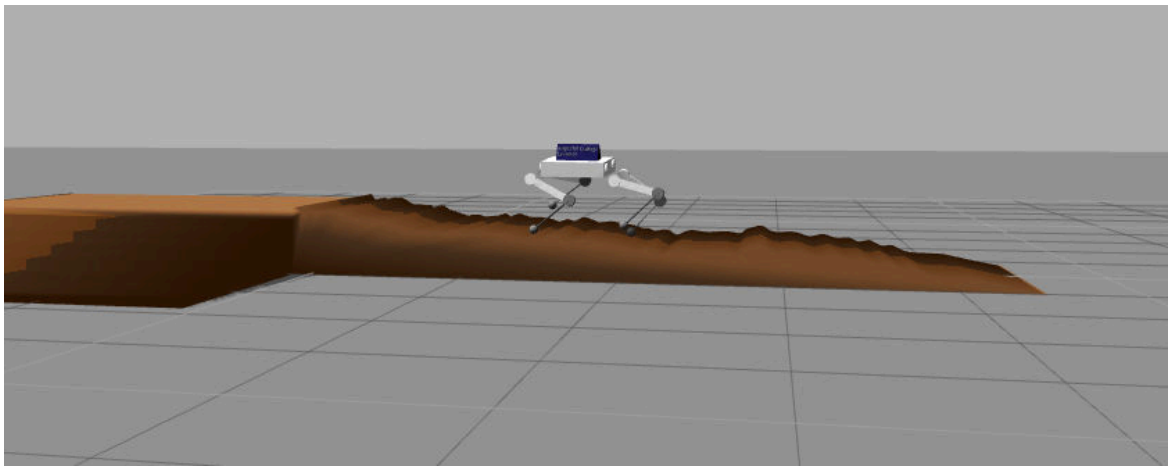
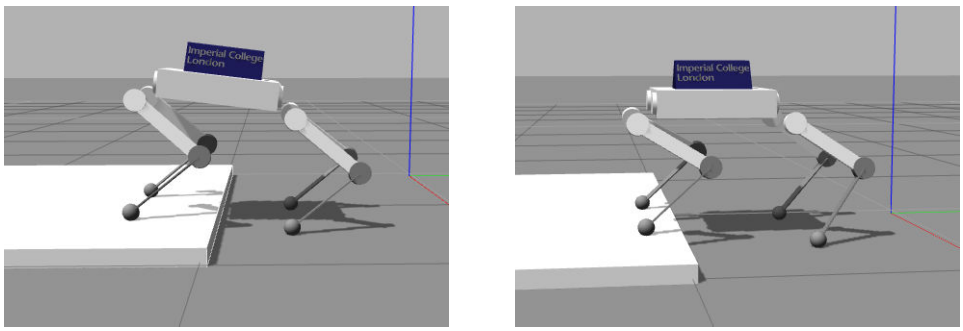


Figure 3.9: Rough terrain model

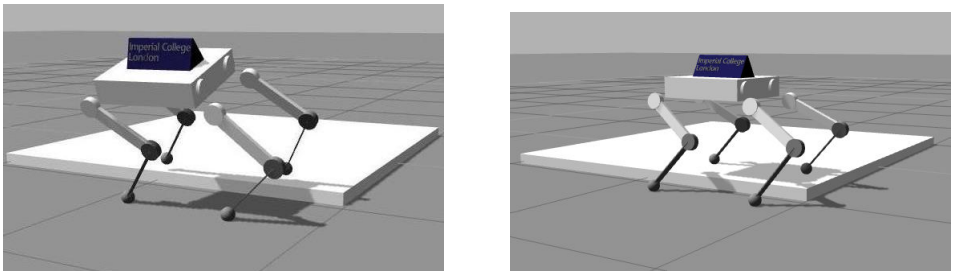
Figure 3.10 shows the robot's front two legs stepping onto a box. In Figure 3.10 (a), the roll and pitch controller is inactivated, causing the robot to experience a pitch, which then leads to instability. In figure 3.10 (b), the roll and pitch controller is active. Here, the robot adjusts the height of the front legs to reducing the pitch and regaining stability. Figure 3.11 shows a similar experiment, however, in this case, the robot experiences a roll by side stepping onto the box. With an inactive controller, Figure 3.11 (a),

the robot is not able to compensate for this and falls over. In Figure 3.11 (b) the robot uses the controller to autonomously regain stability.



(a) Controller inactive (b) Controller active

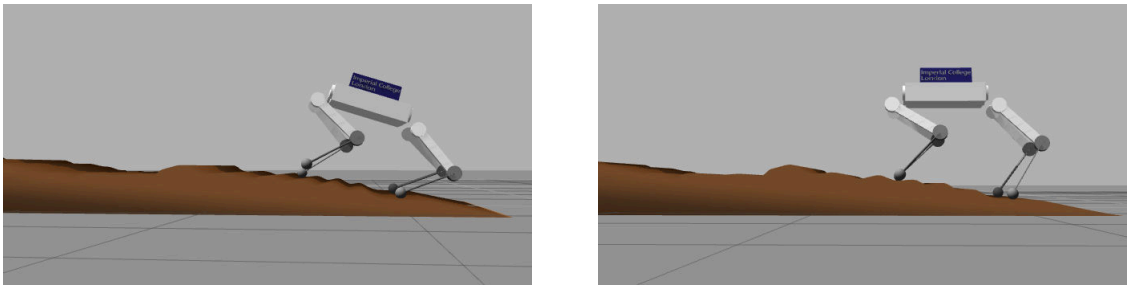
Figure 3.10: Simulated robot stepping onto a box



(a) Controller inactive (b) Controller active

Figure 3.11: Simulated robot side stepping onto a box

Figure 3.12 shows the robot attempting to traverse an uneven inclined terrain. Without the controller, Figure 3.12 (a), the robot experiences a significant pitch and in most cases falls over. With the controller active, Figure 3.12 (b), the robot adjusts the height of the legs accordingly and improves the stability. The controller improved the stability of the robot in this scenario, however, it did still tend to fall on some occasions. It is however, worth mentioning that, using the roll and pitch controller, the robot was able to elegantly descend stairs very consistently.



(a) Controller inactive (b) Controller active

Figure 3.12: Simulated robot attempting to traverse an uneven inclined terrain

3.6 Push recovery

Before different push recovery defence strategies were tested, the push detector was tested by applying a force between 0 - 100 N for a duration of between 0 - 1 s. Figure 3.13 shows how the four different force and duration combinations were successfully detected.

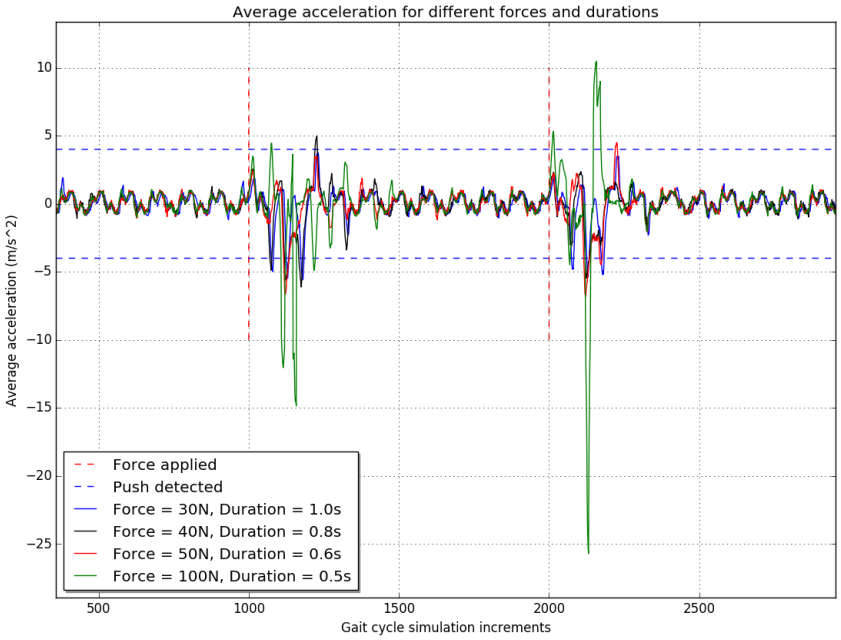
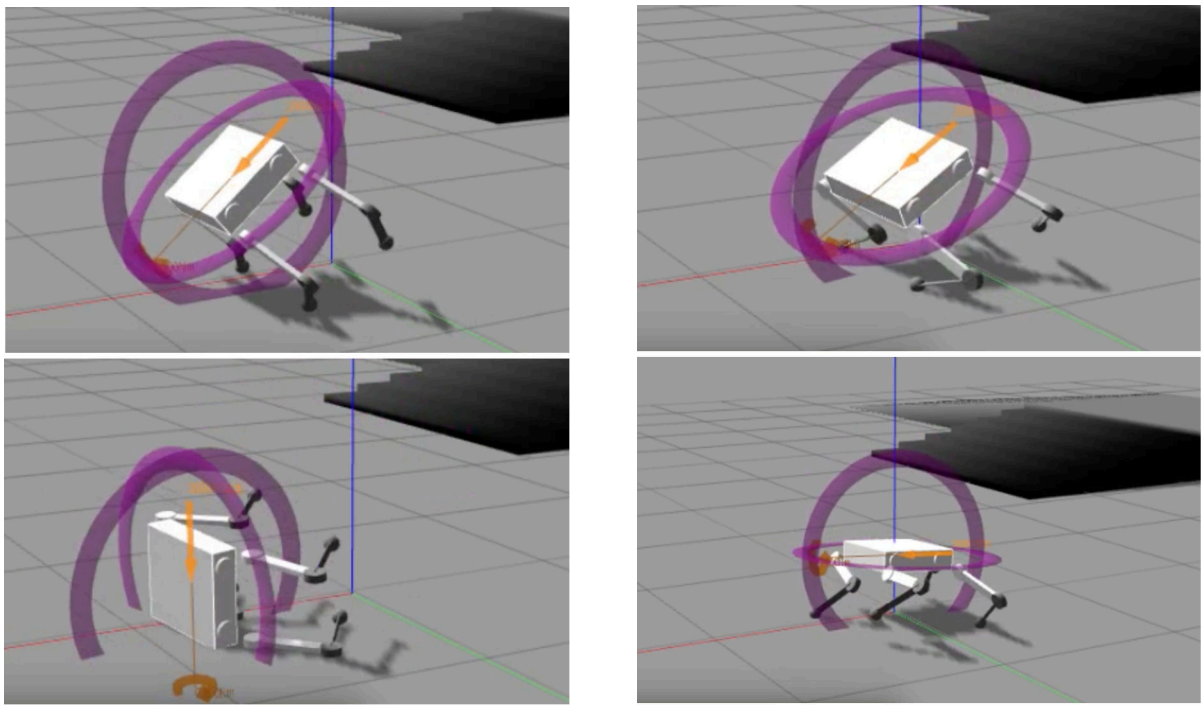


Figure 3.13: Force detector tested with combinations of different forces and durations

Figure 3.14 shows the simulated robot experiencing a transverse force, both with and without the push recovery in place. In this example, the defence strategy involved stepping in the direction opposing the force, increasing the transverse step offset and reducing the height of the robot. This strategy was very effective and dramatically increased the resistance to perturbations of the robot, and prevented it from falling over.

3.7 Bayesian Optimization

The Bayesian optimization hyper-parameters search was used in an attempt to optimize the parameters for two different scenarios. In each case, a simulation function, including a task-specific reward function, was developed. The first case involved walking up uneven terrain, the reward function was dependent on distance, speed and stability. The second case involved defending against random forces, acting in all different directions and for different durations. The reward function was dependent on time and cumulative force magnitudes experienced. An open-source Bayesian Optimization framework was then used to perform the *learning*. Unfortunately, due to time constraints, neither of the scenarios were correctly optimized to completion and required additional training.



(a) Push recovery inactive

(b) Push recovery active

Figure 3.14: Simulated robot experiencing a transverse force

Chapter 4

Discussion

4.1 Gait generation

The gait generation results - in particular, the simulated and supported robot results - suggest that the controller is able to successfully generate both the walking and trotting gaits using the simple and adaptable gait graph method. When these locomotion gaits were implemented on the unsupported physical robot, it showed significant potential, however, it was unstable and difficult to control. Significant time is required to perform the sim to real conversion, which can be a tedious parameter-tuning process. Without this critical testing time, it is impossible to know whether the instability is caused by incorrectly-tuned parameters, the hardware or the controller itself.

The addition of tactile sensors on the feet of the robot could radically improve the stability. The current position controller assumes that all feet make contact with the ground during the transition from swing to stance phase. However, this is not always the case, as a small roll or pitch angle cause the feet to make contact before or after the swing to stance phase transition. If contact is made before the transition, the robot continues the trajectory, which forces it away from the ground and further disturbs the roll/pitch of the robot. A similar process occurs if contact is made after the transition. Tactile sensors could be used as a binary switch, indicating a required transition from swing to stance phase, which would improve the stability.

Another major problem experienced was the lack of friction on the feet of the robot. This cause the robot to slip, and further prevented elegant locomotion. Various different high-friction materials, such as bicycle tubing, rubber, and tennis table bat padding were appended in an attempt to increase the friction. These materials only slightly reduced the slippage. A possible solution could be an integrated rubber compressible spherical foot.

4.2 Manoeuvrability

A function mapping a translation or rotation to individual footstep vectors was used to maneuver the robot. Additionally, these translations, rotations and other locomotion parameters could be updated within gait cycles, resulting in a highly responsive controller. Trajectory discontinuities were eliminated by implementing a step interpolator. The results suggest that the controller was able to successfully maneuver the robot in any direction.

Although the simulated robot demonstrated precise control, the physical robot was far more erratic. Additionally, the physical robot tended to move backwards for sagittal step lengths less than 50 mm. In

order to solve this, a feed forwards offset of 50 mm was added. This, however, introduced a problem with the step interpolation when transitioning from stationary to non-stationary gaits. The step interpolator prevented an instantaneous step length of 50 mm, resulting in the robot moving backward initially before continuing.

This brought to light an obvious trade-off of the step interpolator between response time and smooth transitioning. A possible solution would be a variable step interpolator, depending on the desired action of the robot.

4.3 Yaw controller

The yaw controller consisted of a feed forward term and a feedback controller, and was specially designed to be used in either autonomous or manual mode (using the Xbox controller). Based on the results, the yaw controller was successfully able to maintain or achieve a desired position. One problem that remained was the yaw oscillations. The feedforward term reduced the oscillations, however, the robot continued to yaw back and forth by about 2 degrees. In order to completely solve this problem, all feet must make perfectly normal contact with the ground when transitioning from swing to stance phase - this will prevent the contact forces from generating undesired moments.

4.4 Roll and pitch controller

The roll and pitch controller was designed to maintain the roll and pitch angles of the robot at zero. The testing showed that the design was successfully implemented, and in most cases it improved the stability of the robot. The controller worked well on both constant and irregular gradients between 0 and 20 degrees, and on step of heights between 0 mm and 200 mm. However, maintaining zero roll and pitch on steeper inclines became very impractical as either the front or the back legs were forced to inoperable heights. This problem could be solved by using the IMU to detect the gravity vector, pitching the legs appropriately, such that the body of the robot remained parallel to the slope, while the legs remained vertical. In doing so, the center of mass would shift, preventing the robot from toppling over. This solution was attempted, however, further tuning and testing is required.

4.5 Push recovery

Linear accelerations, were used to detect forces applied to the robot. This force detector successfully detected a range of different forces applied for different durations. A defence strategy - which included reducing the height of the robot, increasing the transverse step offset and stepping in the direction opposing the applied force - was then implemented. This strategy significantly improved the resistance to applied forces and perturbations. Although this strategy was successful in simulation, it was not tested on the physical robot and could prove to be impractical or ineffective. Many different research groups have successfully solved this problem using slightly more calculated and complex strategies, such as the push recovery based on capture points presented in [11]. Thus, in the future, it may be necessary to use the existing and effective push detector with a different defence strategy.

4.6 Bayesian optimization

A Bayesian optimization algorithm was implemented, in an attempt to perform hyper-parameter optimization for various tasks. Although the implementation didn't necessarily yield successful results, it was a sound demonstration of the modularity, and proved that this controller can be integrated and used in machine learning applications.

One major problem was the occasional case in which seemingly good parameters yielded poor results. This dramatically distorted the optimization function, and disrupted the process. This could be corrected by running each set of parameters more than once, only using the best outcome. The underlying premise is that it is possible for good parameters to yield a poor result, however, it is highly unlikely that poor parameters will yield a good result.

4.7 Overall controller

The controller successfully demonstrated the ability to generate multiple gaits, manoeuvre in all directions, be autonomously or manually controlled, and - to some extent - was perpetuation resistant. Therefore, it met the primary aims of the project. The results suggest that the gait generation method, manoeuvrability and yaw control require little improvement and are suitable to be used as they are, in future work. However, the other feedback controller methods could require improvement and additional development. With that said, before any improvements are made, the current controller must be fully tested on the physical robot - with appropriate parameter tuning time - in order to legitimately identify the successes and shortcomings of the controller.

Chapter 5

Conclusion

A bespoke high-level controller for DogBot - a quadrupedal robot - was designed, implemented and tested. The controller is capable of generating multiple gaits using a gait graph method and maneuver in all directions. An IMU (inertial measurement unit) was added to the robot, which enabled the development of various attitude controllers as well as a push detection and recovery method. The controller was successfully implemented and tested in simulation, however, the physical robot was unstable and erratic. Due to a number of minor mechanical failures, the physical robot testing time was limited, making it impossible to accurately identify the true source of the instability. The modular design of the controller allows for application and implantation within learning algorithms, this was demonstrated using a Bayesian optimization hype-parameter search. Moving forward, the addition of tactile and other sensors could be used to further improve the controller. Thus, this controller provides a strong and modular base upon which further research and development can be conducted.

Bibliography

[1] R. Robotics, "Dogbot," 2018, photograph taken from React Robotics website. [Online]. Available: https://reactrobotics.com/

[2] M. Raibert, K. Blankespoor, G. Nelson, and R. Playter, "Bigdog, the rough-terrain quadruped robot," *IFAC Proceedings Volumes; 17th IFAC World Congress*, vol. 41, no. 2, pp. 10 822–10 825, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1474667016407020

[3] S. Seok, A. Wang, Y. C. Meng, D. Otten, J. Lang, and S. Kim, "Design principles for highly efficient quadrupeds and implementation on the mit cheetah robot," 2013, pp. 3307–3312.

[4] C. Semini, G. T. N, E. Guglielmino, M. Focchi, F. Cannella, and G. C. D, "Design of hyq a hydraulically and electrically actuated quadruped robot," *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering*, vol. 225, no. 6, pp. 831–849, 2011, doi: 10.1177/0959651811402275; 31. [Online]. Available: https://doi.org/10.1177/0959651811402275

[5] H. Khan, S. Kitano, M. Frigerio, M. Camurri, V. Barasuol, R. Featherstone, D. G. Caldwell, and C. Semini, "Development of the lightweight hydraulic quadruped robot minihyq," 2015, pp. 1–6.

[6] C. Semini, V. Barasuol, J. Goldsmith, M. Frigerio, M. Focchi, Y. Gao, and D. G. Caldwell, "Design of the hydraulically actuated, torque-controlled quadruped robot hyq2max," *IEEE/ASME Transactions on Mechatronics*, vol. 22, no. 2, pp. 635–646, 2017.

[7] M. Hutter, C. Gehring, M. Bloesch, M. A. Hoepflinger, C. D. Remy, and R. Siegwart, *StarlETH: A compliant quadrupedal robot for fast, efficient, and versatile locomotion*, ser. Adaptive Mobile Robotics. World Scientific, 2012, pp. 483–490.

[8] M. Hutter, C. Gehring, D. Jud, A. Lauber, C. D. Bellicoso, V. Tsounis, J. Hwangbo, K. Bodie, P. Fankhauser, M. Bloesch, R. Diethelm, S. Bachmann, A. Melzer, and M. Hoepflinger, "Anymal - a highly mobile and dynamic quadrupedal robot," 2016, pp. 38–44.

[9] S. Coros, A. Karpathy, B. Jones, L. Reveret, and M. van de Panne, "Locomotion skills for simulated quadrupeds," *ACM Trans. Graph.*, vol. 30, no. 4, pp. 59:1–59:12, Jul. 2011. [Online]. Available: http://doi.acm.org/10.1145/2010324.1964954

[10] Y. Sakakibara, K. Kan, Y. Hosoda, M. Hattori, and M. Fujie, "Foot trajectory for a quadruped walking machine," 1990, p. 322 vol.1.

[11] V. Barasuol, J. Buchli, C. Semini, M. Frigerio, E. R. D. Pieri, and D. G. Caldwell, "A reactive controller framework for quadrupedal locomotion on challenging terrain," 2013, pp. 2554–2561.

[12] C. Gehring, S. Coros, M. Hutter, M. Bloesch, M. A. Hoepflinger, and R. Siegwart, "Control of dynamic gaits for a quadrupedal robot," 2013, pp. 3287–3292.

[13] L. Righetti and A. J. Ijspeert, "Pattern generators with sensory feedback for the control of quadruped locomotion," 2008, pp. 819–824.

[14] A. J. Ijspeert, "Central pattern generators for locomotion control in animals and robots: A review," *Neural Networks; Robotics and Neuroscience*, vol. 21, no. 4, pp. 642–653, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0893608008000804

[15] C. Semini, V. Barasuol, T. Boaventura, M. Frigerio, M. Focchi, D. G. Caldwell, and J. Buchli, "Towards versatile legged robots through active impedance control," *The International Journal of Robotics Research*, vol. 34, no. 7, pp. 1003–1020, 2015, doi: 10.1177/0278364915578839; 12. [Online]. Available: https://doi.org/10.1177/0278364915578839

[16] M. Hutter, C. Gehring, M. Bloesch, M. Hoepflinger, and R. Siegwart, "Walking and running with starleth," 2013, iD: 189758. [Online]. Available: https://infoscience.epfl.ch/record/189758/files/Hutter2013AMAM.pdf

[17] M. Hutter, C. Gehring, M. A. Hpflinger, M. Blsch, and R. Siegwart, "Toward combining speed, efficiency, versatility, and robustness in an autonomous quadruped," *IEEE Transactions on Robotics*, vol. 30, no. 6, pp. 1427–1440, 2014.

[18] J. Liu, J. Pu, and J. Gu, "Central pattern generator based crawl gait control for quadruped robot," 2016, pp. 956–962.

[19] R. F. Souto, G. A. Borges, and A. R. S. Romariz, "Gait generation for a quadruped robot using kalman filter as optimizer," 2009, pp. 1037–1042.

[20] C. Gehring, S. Coros, M. Hutler, C. D. Bellicoso, H. Heijnen, R. Diethelm, M. Bloesch, P. Fankhauser, J. Hwangbo, M. Hoepflinger, and R. Siegwart, "Practice makes perfect: An optimization-based approach to controlling agile motions for a quadruped robot," *IEEE Robotics & Automation Magazine*, vol. 23, no. 1, pp. 34–43, 2016.

[21] S. Zhang, H. Ma, Y. Yang, and J. Wang, "The quadruped robot adaptive control in trotting gait walking on slopes," in *2nd International Conference on Materials Science, Resource and Environmental Engineering (MSREE 2017)*, vol. 1890, Coll. of Mechatron. Eng. Autom., Nat. Univ. of Defense Technol. Changsha, Changsha, China. USA: AIP - American Institute of Physics, 10/05 2017, p. 020004 (14 pp.), t3: AIP Conf. Proc. (USA); undefined; undefined; undefined; undefined. [Online]. Available: http://dx.doi.org/10.1063/1.5005182

Appendices

Appendix A

Operational instructions

This section includes the operation instructions for getting the physical robot, as well as the Gazebo model of the robot up and running. It includes common problems experiences at the various stages solutions to them. It also includes tips regarding the controller scripts and how to work your way around them. Please note that the ROS controller developed by React Robotics is designed such that controller scripts, which publish desired motor positions to the respective topics, work identically on both the physical robot and the Gazebo simulation. For more information and a detailed set-up guide, please see: React Robotics DogBot Repository. Please note, Dr. Petar Kormushev has been provided with screen and video recording of the following processes.

A.1 Getting started with the Gazebo simulation

1. Open up a terminal and navigate to the following directory: ∼/src/RR/DogBot/ROS
2. Run: catkin build
3. Run: source devel/setup.bash
4. Run: roslaunch dogbot_gazebo gztest.launch
5. At this point Gazebo with the model of DogBot should be open
6. If Gazebo crashes, close the terminal, wait a minute and go back to step 4.
7. You now need to run a controller script in a **new terminal**, using a launch file
8. Run: roslaunch dogbot_control current_controller.launch
9. Have a look at this launch file, it launches a number of nodes including the main controller node, the xbox controller node, a PID node for the yaw controller and 4 PID nodes (one per leg) for the roll and pitch controller
10. Make sure the xbox remote control is connected, once it is, you should be able to use it to control the robot.
11. If you could like to view or edit the main scripts, you can find them in the following directory: ∼/src/RR/DogBot/ROS/src/dogbot_control/scripts

A.2 Getting started with the physical robot

A.2.1 Connecting the batteries of the physical robot

1. Please see: Official DogBot Powering-up and Battery Guide for a detailed description of the powering up processes. I have provided an exact of their battery connection process below.
2. Switch the breaker to the green position
3. Move the power switch to the horizontal position
4. Ensure the power supply is disconnected
5. Connect the batteries on each side
6. Hold the power switch in the up position for 5 seconds
7. Switch the breaker on (to the red position)
8. Switch the power switch to the down position
9. Make sure that the red E-Stop is deactivated and in the up position.

A.2.2 Homing the motors using the GUI

Before you can control the robot using ROS, you must home the twelve motors using the supplied GUI. In order to do so, you must first ensure that the robot is completely supported in the air with enough space to freely move it's legs in all directions. You can then complete the following steps:

1. Make sure that the ROS dogbot hardware controller is not running, if it is, the following process with fail and it will take you a long time to figure out why!
2. Connect the physical robot to the communication box using the grey communications cable.
3. Connect the communication box to the computer using the USB cable
4. Open up a terminal and navigate to the following directory: ∼/src/RR/DogBot/API/build
5. If the build directory does not exist, you will need to create it using the Setup Steps Manual.
6. Run ./dogBotServer
7. Open Qt Creator and run DogBotUI
8. Click the connect button
9. Navigate to the **Overview** tab at the top of the GUI
10. Click the **Standby** tab at the bottom of the GUI, the status of all items should be green and read "ok"
11. Click the **Power On** tab at the bottom of the GUI
12. You will now need to click **Home all** tab, several times, until all of the motors have been homed.
13. In order to ensure the motors are correctly homed, navigate to the **Animation** tab at the top of the GUI and select **Run Animation**
14. If you experience any problems with any of the aforementioned steps, complete a full power cycle, disconnect connections where possible and wait about a minute.

A.2.3 Controlling the physical robot using ROS

Once you have homed the twelve motors and have the DogBot server running you will be able to control the robot using ROS.

1. Open up a terminal and navigate to the following directory: ∼/src/RR/DogBot/ROS
2. Run: catkin build
3. Run: source devel/setup.bash
4. Run: roslaunch dogbot_control dogbot_hardware.launch
5. You now need to run a controller script in a **new terminal**, using a launch file
6. Run: roslaunch dogbot_control current_controller.launch

Appendix B

Additional information

B.1 State estimator

With the future improvement of the controller in mind, a state estimator was developed. The state estimator uses forward kinematics to calculate the current Cartesian coordinates of each foot relative to the hip of the respective leg. The state estimator outputs the feet positions in a 3x4 array in the following way:

$$\begin{bmatrix} FRx & BRx & FLx & BLx \\ FRy & BRy & FLy & BLy \\ FRz & BRz & FLz & BLz \end{bmatrix}$$

This information can be combined with the information from the IMU to predict a number of scenarios. For example, the height of the four legs could be used to calculate theoretical roll and pitch angles which could be compared to the roll and pitch angle from the IMU in order to determine whether the robot is stepping on an object. The state estimator can be found in the following directory:
~/src/RR/DogBot/ROS/src/dogbot_control/scripts/StateEstimator

B.2 Additional flow diagrams

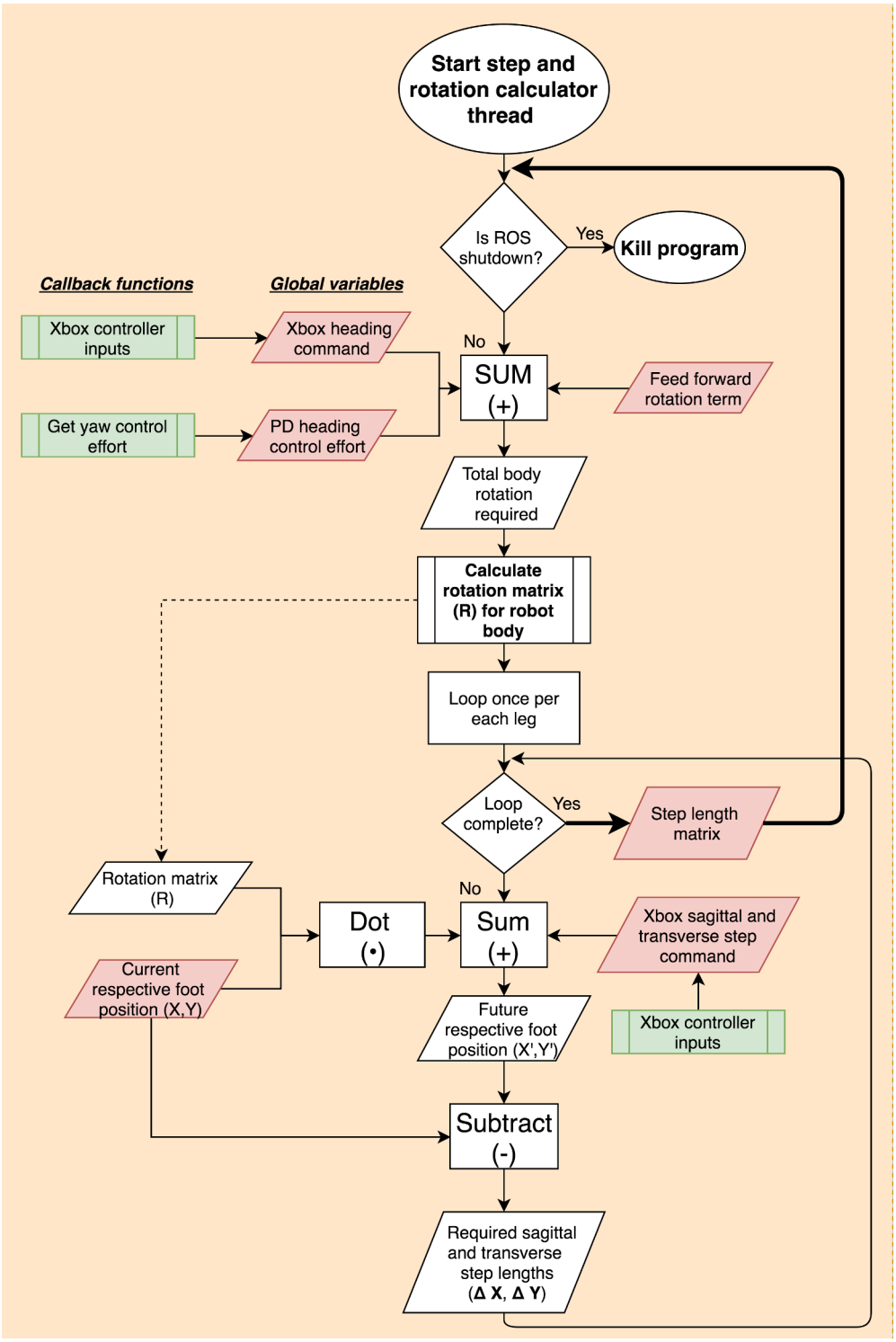


Figure B.1: Software implementation of the Step and Rotation Function