Institute of Parallel and Distributed Systems
Department of Machine Learning and Robotics
University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

CSIRO
Autonomous Systems Lab
1 Technology Court
QLD–4069 Pullenvale

Diploma Thesis Nr. 3571

# Real-Time Stabilisation for Hexapod Robots Using Task-Space Constraints

Marcus Hörger

| | |
|---|---|
| **Course of Study:** | Software Engineering |
| **Examiner:** | Prof. Dr. rer. nat. Marc Toussaint |
| **Supervisor:** | Dr. Alberto Elfes |
| **Commenced:** | September 23, 2013 |
| **Completed:** | March 25, 2014 |
| **CR-Classification:** | I.7.8, I.2.9 |

# Abstract

Legged robots such as hexapod platforms are capable of navigating in rough and unstructured terrain. When the terrain model is either known *a priori* or is observed by on-board sensors, motion planners can be used to give desired motion and stability for the robot. However, unexpected leg disturbances could occur due to inaccuracies of the model or sensors or simply due to the dynamic nature of the terrain. This thesis presents a method based on task-space constraints for real-time stabilisation of hexapod robots which keeps the robot inside defined task-space constraints to recover from unexpected events such as leg slip. A ROS-based control system for hexapod robots is developed and implemented, integrating the presented stabilisation method. The approach is experimentally evaluated using two PhantomX hexapod platforms - one with extended tibia segments which significantly reduces its stability. The results show that the proposed method significantly improves the static stability when unexpected events occur during locomotion.

# Contents

# List of Figures

# List of Tables



# List of Algorithms

# 1 Introduction

Hexapod robots are well suited for navigating in rough and uneven terrain that can be challenging to conventional wheeled or tracked vehicles. These robots can efficiently adapt to the complex terrain by adjusting their gait patterns, footfall trajectory or footholds. There are broadly two approaches in attaining hexapod locomotion: when terrain is unknown, the hexapod executes repeated pattern of coupled footfall or a gait with little feedback [LLC88]; and when the terrain model is completely known or is observed by on-board sensors with sufficient accuracy, the footfalls are computed to give desired motion and stability [HBL+08] [BRL03] [Bel11] [BS12] [BS11]. The former approach relies on the stochastic nature of the hexapod's interaction with the terrain to recover from slips and trips. However, there is no guarantee of the approach working in very challenging environments like steep slopes or on slippery surfaces in the event of unexpected disturbances in the leg-terrain interaction such as leg slips or changes in the body orientation due to incomplete and uncertain terrain information. In order to navigate such challenging terrains, a fast reactive approach is necessary which is able to compensate these unforeseen disturbances. The stable footfall generation for a known terrain is computationally expensive due to the high dimensionality of the planning space. When a slip occurs, the planner often has to recompute the footfall from a new post-slip configuration.

In this thesis, a real-time stabilisation system based on state-of-the-art motion planners is proposed to detect and arrest external disturbances in real-time, without knowledge of the local terrain, before the body moves to an unstable configuration, potentially damaging itself, or to a configuration from where recovery is difficult.
To evaluate the proposed method, a software control architecture for legged-locomotion of hexapods has been developed and implemented. This software system integrates the proposed stabilisation method using the ROS-framework, a widely used software-framework in the robotics community.

## Structure of this thesis

The structure used for this thesis is the following:

**Chapter 2 – Mathematical foundations:** Summarises the mathematical foundations an methods used in this thesis.

**Chapter 3 – System description** Overview of the hardware setup and the software system developed during the thesis work.

**Chapter 4 – Hexapod model** Describes the model of the hexapods (including methods for forward- and inverse kinematics).

**Chapter 6 – State-space** Introduces the state space, including important subspaces of the hexapod.

**Chapter 7 – Random sample based planning techniques** Gives an overview of current state-of-the-art random sample based planning techniques.

**Chapter 8 – Stabilisation approach** Describes the stabilisation approach based on random sample based planners.

**Chapter 9 – Experiments and results** Validates the developed stabilisation approach by describing the experimental setup and the achieved results.

**Chapter 10 – Conclusion and future work** Gives a conclusion of the work that has been done for this thesis and summarises ideas for future work.

# 2 Mathematical foundations

This chapter gives an overview of the basic mathematical foundations which are required further in this thesis. It describes basic kinematic methods and, further in this section, more problem specific methods.

## 2.1 Basic kinematic equations

Let point $p$ be a point in the reference frame $M$. The trajectory of p which is traced relative to a fixed frame $F$ is

$$P(t) = [T(t)] \cdot p = \begin{bmatrix} R(t) & d(t) \\ 0 & 1 \end{bmatrix} = \begin{pmatrix} p \\ 1 \end{pmatrix} \tag{2.1}$$

With $[T(t)]$ being a homogeneous transformation assembled by a set of rotations $[R(t)]$ and translations $[d(t)]$.

The velocity $V_p$ of the trajection of point $p$ can be obtained by deriving the transformation with respect to the time $t$:

$$V_p = \left[\dot{T}(t)\right] \cdot p = \begin{bmatrix} \dot{A}(t) & \dot{d}(t) \\ 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} p \\ 1 \end{pmatrix} \tag{2.2}$$

with the dot denoting the derivative with respect to time.

## 2.2 Kinematic chains

A kinematic chain is a set of links $[l_i]$ connected by a set of joints $[j_i]$. The joints play the most important role in kinematic chains as they define the type of motion and constraints of body $A$ in respect to body $B$. Therefore they are also called kinematic pairs. In general there are two classes of kinematic joints:

- Lower-pair joints
- Higher-pair joints

### 2.2.1 Lower-pair joints

A lower-pair joint is an ideal joint which defines contact between a point, line or plane of a body to a point, line or plane of another body. A joint is a lower-pair joint if the two bodies connected by this joint have a surface contact.

### 2.2.2 Higher-pair joints

A joint is called a higher-pair joint when the the two bodies have a point or line contact.

It is clear that the physical joints used for the hexapod platforms are lower-pair joints. However, the hexapod model described in 9.4(a) treast them as lower-pair joints (two links have point contact at the joint).

## 2.3 Denavit-Hartenberg transformation

The Denavit-Hartenberg transformation is a standard procedure based on homogeneous matrices to transform a local coordinate system $T_{n-1}$ into the local coordinate system $T_n$ in a kinematic chain. The Denavit-Hartenberg transformation is widely used for forward-kinematic purposes. To transform the system $T_{n-1}$ to $T_n$ there are three assumptions which have to be met by the local coordinate systems:

- The $z_n$-axis defines the axis of motion of joint $j_n$
- The $x_n$-axis is the cross-product of $z_{n-1}$ and $z_n$

  $$x_n = z_{n-1} \times z_n$$

- The system $(x_n, y_n, z_n)$ forms a right hand system

With those assumptions, the Denavit-Hartenberg transformation consists of the following single transformations:

1. A rotation $\theta_n$ around the $z_{n-1}$-axis till $x_{n-1}$ and $x_n$ are parallel

$$D_{Rot}(z_{n-1}, \theta_n) = \begin{pmatrix} \cos\theta_n & -\sin\theta_n & 0 & 0 \\ \sin\theta_n & \cos\theta_n & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.3}$$

2. A translation $d_m$ along the $z_{n-1}$-axis to the intersection point of $z_{n-1}$ and $x_n$

$$D_{trans_d}(z_{n-1}, d_n) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_n \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.4}$$

3. A translation $a_n$ along the $x_n$-axis till the origins of both systems are congruent

$$D_{trans_a}(x_n, a_n) = \begin{pmatrix} 1 & 0 & 0 & a_n \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.5}$$

4. A rotation $\alpha_n$ around the $x_n$ axis to make $z_{n-1}$ and $z_n$ congruent

$$D_{rot_{\alpha_n}}(x_n, \alpha_n) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha_n & -\sin\alpha_n & 0 \\ 0 & \sin\alpha_n 0 & \cos\alpha_n & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.6}$$

The resulting transformation $D_n$ has the form

$$D_n^{n-1}(\theta_n, d_n, a_n, \alpha_n) = D_{Rot}(z_{n-1}, \theta_n) D_{trans_d}(z_{n-1}, d_n) D_{trans_a}(x_n, a_n) D_{rot_{\alpha_n}}(x_n, \alpha_n) \tag{2.7}$$

$$D_n^{n-1}(\theta_n, d_n, a_n, \alpha_n) = \begin{pmatrix} \cos\theta_n & -\sin\theta_n\cos\alpha_n & \sin\theta_n\sin\alpha_n & a_n\cos\theta_n \\ \sin\theta_n & \cos\theta_n\cos\alpha_n & -\cos\theta_n\sin\alpha_n & a_n\sin\theta_n \\ 0 & \sin\alpha_n & \cos\alpha_n & d_n \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.8}$$

## 2.4 Inverse kinematics

The purpose of inverse kinematics is to calculate the joint angles $\theta_j,\ j \in \{c, f, t\}$ of leg $i$ for a given end-effector pose $\hat{p} = (p, o)$, with $p$ being the position and $o$ being the orientation of the end-effector, relative to a frame $F$. Depending on the kinematic chain, it can be complex to solve the inverse-kinematic problem, since no general approach exists to solve the problem. In practice, several algebraic, geometric and numeric approaches are used. Another difficulty is the fact that in general, multiple solutions for the inverse-kinematic problem exist, given an end-effector pose $\hat{p}$.
However, solving the inverse-kinematic problem for the legs of the hexapod platforms used in this thesis can be achieved by a simple geometrical approach, which is detailed in 4.5.2.

## 2.5 Quaternions

Assume we have a rigid body which has a certain position $p$ and orientation $o$ relative to a reference frame $F_0$. There are several approaches to represent "orientation" in a three-dimensional euclidean space, such a rotation matrices or Tait-Bryan-angles.
Another approach is representing the orientation by a rotation quaternion. A quaternion is an element of a four-dimensional vector-space, consisting of a real and three imaginary components. Every quaternion $q$ can be written as

$$q = x_0 + x_1 i + x_2 j + x_3 k \tag{2.9}$$

with $x_i \in \mathbb{R}$. The elements $\{1, i, j, k\}$ form the standard base of the quaternion space.
In real applications, quaternions are often represented as four dimensional vectors $(x_0, x_1, x_2, x_3)$.
In robotics and other fields, quaternions play an important role since a unit quaternion represent a rotation $x_0$ around the axis $(x_1, x_2, x_3)$ in a three-dimensional space. A quaternion can be normalised like any other vector by dividing it by it's magnitude.

### 2.5.1 Conjugation and Hamilton product

Let $q = x_0 + x_1 i + x_2 j + x_3 k$ be a quaternion. The conjugation of $q$ denoted by $q^{-1}$ is defined as

$$q^{-1} = x_0 - x_1 i - x_2 j - x_3 k \tag{2.10}$$

Two quaternions can be multiplied using the Hamilton product. Let $q_1 = x_0^1 + x_1^1 i + x_2^1 j + x_3^1 k$ and $q_2 = x_0^2 + x_1^2 i + x_2^2 j + x_3^2 k$ be to quaternions. The product of these quaternions is defined as

$$q_1 q_2 = \begin{pmatrix} x_0^1 x_0^2 - x_0^2 x_1^2 - x_2^1 x_2^2 - x_3^1 x_3^2 \\ +(x_0^1 x_1^2 + x_0^2 x_0^2 + x_2^1 x_3^2 - x_3^1 x_2^2) i \\ +(x_0^1 x_2^2 - x_1^1 x_3^2 + x_2^1 x_0^2 + x_3^1 x_1^2) j \\ +(x_0^1 x_3^2 + x_1^1 x_2^2 - x_2^1 x_1^2 + x_3^1 x_0^2) k \end{pmatrix} \tag{2.11}$$

Let $p = p_1 i + p_2 j + p_3 k$ be an arbitrary vector in a three-dimensional euclidean space written as a quaternion. As stated earlier, a quaternion represents a rotation in three-dimensional space. This rotation can be applied to $p$ using

$$p' = qpq^{-1} \tag{2.12}$$

with $p'$ being the rotated vector. Assume now, we first want to rotate $p$ with the unit quaternion $q_1$ and then apply another rotation represented by $q_2$. The two rotations can be represented by a single quaternion $q_{res}$ with

$$q_{res} = q_2 q_1 \tag{2.13}$$

This means when rotations are represented in quaternion form, and arbitrary combination of these rotations can be expressed as a single quaternion, using the Hamilton product. Note that the quaternion multiplication is not commutative, so the order of rotations applied to a vector plays an important role.

## 2.6 Tait-Bryan-angles

Tait-Bryan-angles represent the orientation of a frame $F$ relative to a reference frame $R$ in 3-dimensional Euclidean space. They are usually denoted as $(\varphi, \theta, \psi)$ which is a sequence of three elemental rotations about the axis of a coordinate system.
There are a number of conventions defining the order of the axis the rotation is applied. Here the $z$-$y'$-$x''$ intrinsic rotation is used defined as follows:
Let $R$ be a reference frame with the coordinates $(x, y, z)$ and the origin $o_R = (0, 0, 0))andletF$ be a rotated frame with the coordinates $(\tilde{x}, \tilde{y}, \tilde{z})$ and the origin $o_F = o_R$. Furthermore let $N$ be the intersection between the $x, y$-plane and the $\tilde{y}, \tilde{z}$-plane. The Tait-Bryan-angles $(\varphi, \theta, \psi)$ are defined as follows:

- $\varphi$ is the angle between $y$ and $N$ (yaw)
- $\theta$ is the angle between $\tilde{x}$ and the $x$-$y$-plane (pitch)
- $\psi$ is the angle between $N$ and $\tilde{y}$ (roll)

### 2.6.1 Conversion between Quaternions and Tait-Bryan-angles

Let $q = x_0 + x_1 i + x_2 j + x_3 k$ be a unit quaternion. $q$ can be converted into Tait-Bryan-angles $(\varphi, \theta, \psi)$ with

- $\varphi = arctan2((x_2 x_3 + x_0 x_1), \frac{1}{2} - (x_1^2 + x_2^2)$
- $\theta = arcsin(-2(x_1 x_3 + x_0 x_3))$
- $\psi = arctan2((x_1 x_2 + x_0 x_3), \frac{1}{2} - (x_2^2 + x_3^2))$

## 2.7 Manipulator trajectory planning

The goal of manipulator trajectory planning is to plan a trajectory which brings a manipulator from state $s_{i-1}$ to $s_i$. Generally there are two approaches to plan a trajectory: Workspace planning and - for an $n$-link kinematic manipulator - joint space planning. The first approach is used to plan a desired trajectory for the end-effector, thus requiring knowledge of the exact inverse-kinematic model of the serial chain since the time dependent tip trajectory $p(t)$ has to be mapped into the joint space to perform the trajectory. In general there is no easy solution for the inverse-kinematic problem of an $n$-link serial chain. Therefore joint-space planning approaches are are more interesting for complex kinematic chains.

Joint-space planning approaches aim to plan a joint-trajectory $q(t)$ which describes a path in joint-space from a start configuration $q_0 = q(0)$ to an end configuration $q_n = q(T_n)$. Often intermediate points $q_i = q(T_i)$ are defined which have to be passed during the traversal of the path.
The easiest way to plan a trajectory in joint-space for a given set of points $q_i, 0 \leqslant i \leqslant n$ is to assign a linear path between two successive points $q_{i-1}, q_i$.
However, this approach has some obvious drawbacks. Let $q(t)$ be a function which describes a path with linear segments between the points. $q(t)$ is not differentiable at the intermediate points which leads to non-smooth trajectories. A function $q(t)$ which describes a smooth trajectory has to include the following features:

- Be computational efficient
- $q(t)$ and it's velocity profile $\dot{q}(t)$ (sometimes the acceleration profile $\ddot{q}(t)$ as well) have to be continuous functions of time
- Describe an "optimal" curve under given position, velocity and acceleration constraints.

In the most simple case of a trajectory, a start position $q_0$, and end position $q_1$ and the start and end velocities $\dot{q_0}, \dot{q_1}$ are given. Then the problem is to define a function $q(t)$ under those position and velocity constraints. In general such problems can be solved, using $n-1$-degree polynomial functions

$$q(t) = a_0 + a_1 t + a_2 t^2 + ... + a_n t^n \tag{2.14}$$

where $n$ is the number of constraints of the function which have to be verified at each point of the trajectory to guarantee it's smoothness.
Going back to the simple case where the initial and end position, as well as the initial and end velocity are given. The four constraints lead to a polynomial of degree 3 is required:

$$q(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 \tag{2.15}$$

The four parameters have to be calculated such that the four constraints are satisfied. Having

$$\dot{q}(t) = a_1 + 2a_2 t + 3a_3 t^2 \tag{2.16}$$

we get from the four initial constraints:

$$\begin{aligned} q(0) &= q_0 = a_0 \\ \dot{q}(0) &= \dot{q}_0 = a_1 \\ q(T) &= q_1 = q_0 + \dot{q}_0 t + a_2 T^2 + a_3 T^3 \\ \dot{q}(T) &= \dot{q}_1 = \dot{q}_0 + 2a_2 T + 3a_3 T^2 \end{aligned} \tag{2.17}$$

From $q(T)$ and $\dot{q}(T)$ we get

$$\begin{aligned} a_2 &= \frac{3(q_1 - q_0) - (2\dot{q}_0 + \dot{q}_1)T}{T^2} \\ a_3 &= \frac{2(q_0 - q_1) + (\dot{q}_0 + \dot{q}_1)T}{T^3} \end{aligned} \tag{2.18}$$

The (cubic) polynomial we are looking for is therefore

$$q(t) = q_0 + \dot{q}_0 t + \frac{3(q_1 - q_0) - (2\dot{q}_0 + \dot{q}_1)T}{T^2} t^2 + \frac{2(q_0 - q_1) + (\dot{q}_0 + \dot{q}_1)T}{T^3} t^3 \tag{2.19}$$

In general a trajectory through $n$ points can be defined using a $(n-1)$-degree polynomial. However, this often leads to high oscillations of the curve defined by the polynomial. To avoid this, we are not looking for one polynomial describing the complete trajectory, but $(n-1)$ polynomials with degree $d, d < n-1$ which interpolate the trajectory between two succeeding points $q_{i-1}, q_i$. The degree d is chosen such that the position and velocity constraints in the intermediate points are satisfied. Mathematically we are looking for

$$q(t) = \{q_k(t), t \in [t_k, t_{k+1}], k = 1...n-1\} \tag{2.20}$$

with

$$q_k(\rho) = a_0 + a_1\rho + a_2\rho^2 + ... + a_n\rho^n, \rho \in [0, T_k], T_k = t_{k+1} - t_k \tag{2.21}$$

and the conditions

$$\begin{aligned} q_k(0) &= q_k \\ q_k(T_k) &= q_{k+1} \\ \dot{q}_k(T_k) &= \dot{q}_{k+1}(0) \\ \ddot{q}_k(T_k) &= \ddot{q}_{k+1}(0) \end{aligned} \tag{2.22}$$

The resulting curve is a spline interpolation of the points $q_i$ and has the least amount of oscillation and therefore the lowest "curviness"

## 2.8 Locality sensitive hashing

Locality sensitive hashing [IM98] is a method to perform a approximate or exact nearest-neighbour search in a high dimensional space $M$. The idea is to use a family of hash functions where each hash function maps a set $S \subset M$ into $l$ "buckets", such that similar elements $p \in S$ are hashed into the same bucket. Those hash functions are elements of a *LSH-family* $H(c, R, P_1, P_2)$ with $P_1 > P_2$ defined for a metric space $M = (\mathbb{M}, d)$ with the following properties:

1. $\forall p, q \in M$ : if $d(p,q) \leq R$ then $h_i(p) = h_i(q)$ with probability at least $P_1$
2. $\forall p, q \in M$ : if $d(p,q) \geq cR$ then $h_i(p) = h_i(q)$ with probability at most $P_2$

for any hash function $h_i \in H$. If two elements $p, q \in M$ are close regarding the distance metric $d$, such that $d(p,q) < R$, they are mapped into the same bucket with a higher probability than two elements with a greater distance.
LSH can be used for an approximate nearest neighbour search. Consider a LSH family $F$ with a set of hash functions $\{f_1, f_2, ..., f_n\}$ and a new family of hash functions $H$ where each function $h_i \in H$ is constructed by concatenating $k$ randomly chosen hash functions from $F$ such that

$$h_k(p) = [f_1(p), f_2(p), ..., f_k(p)] \tag{2.23}$$

A common type of functions to use for the $f_i$ is defined as

$$f_i(p) = p_i \tag{2.24}$$

where $p_i$ is the $i$-th bit of $p$. Concatenating randomly chosen $f_i$ to constructs $k$ hash functions $h_i,\ i \in 1, ..., k$ as defined in 2.23, which are called $k$-bit LSH functions as each $f_i$ results in a single bit from the input vector $p$. Using these kind of hash functions compares two input vectors $p, q$ in a $k$-bit Hamming space using the Hamming distance. It can be shown that ...

Figure 2.1 illustrates the hashing of an existing data set.

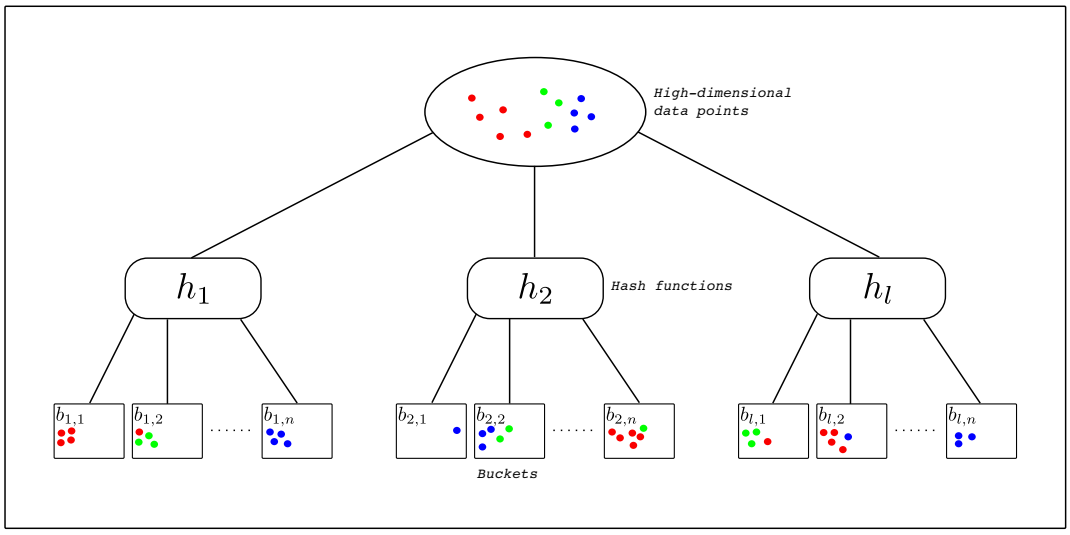



**Figure 2.1:** Illustration of the LSH-preprocessing of the data set (the coloured points in the circle). The data points get mapped into buckets by a set of hash functions $h_1, ..., h_k$. Similar data points get hashed into the same bucket with a higher probability

Given a query point $q$, the algorithm hashes $q$ into each of the $l$ hash tables using the $h_k$ hash functions (as shown in figure 2.2). After that, a brute-force nearest-neighbour search over the points of the data set which were mapped into the same buckets as $q$ is performed. The nearest-neighbour search is stopped as soon as a point $p$ (the approximate nearest-neighbour of $q$) is found such that

(2.25) $d(p, q) \leq cR$

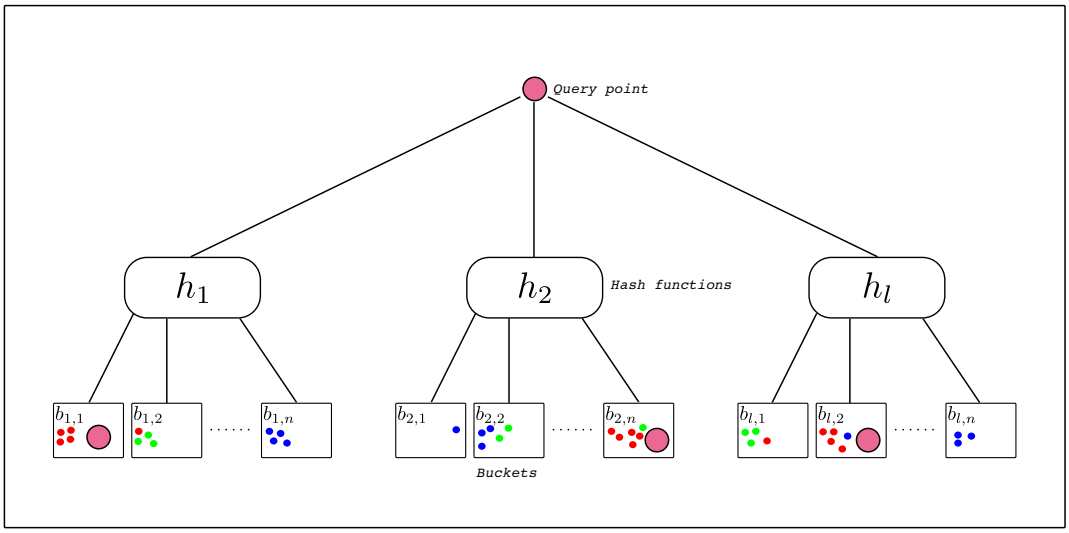



**Figure 2.2:** A point $p$ (red dot) gets hashed into the $l$ hash tables and a brute-force nearest-neighbour search is performed to find the nearest point inside the buckets $p$ has been mapped to.

# 3 System description

## 3.1 Overview

This section gives an overview of the hardware setup used for the hexapod vehicle, as well as an component description of the software architecture and its ROS-implementation developed for this thesis.

## 3.2 Hardware architecture

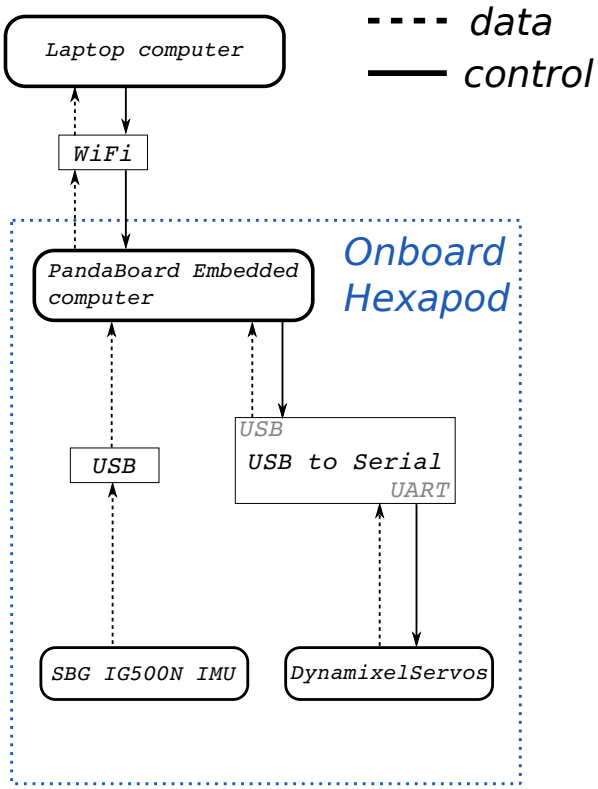



**Figure 3.1:** Overview of the hardware setup for the hexapod platform.

The hardware setup consists of a PandaBoard embedded computer mounted on the hexapod vehicle, an SBG IG-500N internal measurement unit (IMU), 18 Dynamixel AX18 robot actuators, a USB-to-serial interface which connects the PandaBoard to the AX18 servos and a laptop computer connected to the PandaBoard via wireless LAN. The IMU is mounted on top of the PandaBoard, providing orientational information at a 100 Hz rate. The AX18 servos are connected to the PandaBoard via a single serial bus, providing internal information (position, goal position, torque, voltage, temperature) at a rate of 35 Hz. The operating system which runs on the PandaBoard is a compiled Ubuntu 12.04 Desktop version, hosted on a 4 GB SD-Card, the same Ubuntu version which runs on the laptop computer.

For performance purposes, only the ROS drivers for the IMU and the AX18 servos (see 3.3.4) are running on the PandaBoard while the rest of the system runs on the laptop computer.

## 3.3 Software architecture

In this section the developed software control architecture and its ROS implementation is described. A brief introduction to the ROS-framework is given, as well as an overview of the ROS-system used in this paper with a detailed description of its components.

### 3.3.1 Overview

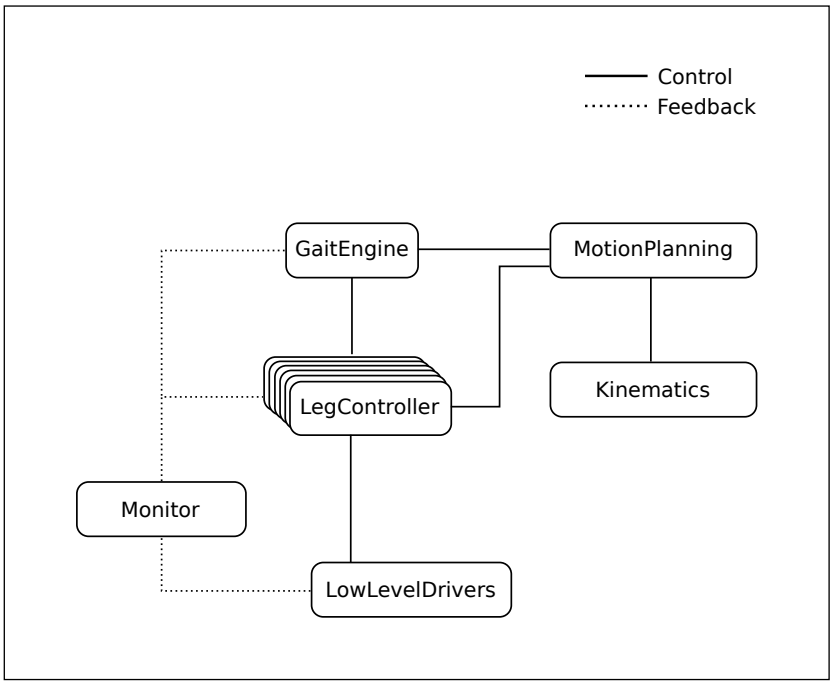



**Figure 3.2:** Software architecture

The core component of the software control architecture developed for this thesis is the *GaitEngine* component. It serves as the main control component, taking inputs from the user and generating gaits using the *MotionPlanning* component. This *MotionPlanning* component is a two-level motion planner. The first level generates trajectories and foothold positions for a given task (here, task refers to world coordinates the hexapod has to move to). The second level generates trajectories for the foot-tips for each gait step.
The *LegController* component servers as an abstraction for each of the single legs. Each LegController controls a single leg. It uses the second level of the *MotionPlanning* component to generate foot-tip trajectories and converts them into servo trajectories using the inverse-kinematic functionality of the *Kinematics module*. These generated servo trajectories are sent to the *LowLevelDriver* component where they are sent to the servos.
In order to be able to monitor the current state of the hexapod in run-time, a *Monitor* component is used. This component collects the raw data coming from the *LowLevelDriver* components and mediates them to the *GaitEngine* and the *LegControllers*.

### 3.3.2 ROS

ROS (Robot Operating system) is an open-source software framework developed and maintained by *WillowGarage* (`http://www.willowgarage.com/`) for developing software for robot

platforms. It is actively used and maintained by the robotics community. ROS consists of a collection of tools, libraries and conventions which aim to support the user in developing robotics software. It consists of a workspace environment, a file system the user can navigate using abstractions of common Unix commands (e.g. *roscd* is the abstraction of *cd*) a build system, and a package manager. The main feature of ROS is it's highly modularised package structure. Functionality for a wide range of existing robots and concepts has been developed by the community and encapsulated into packages. The package system allows for an easy integration of existing packages into a system.
The main features of ROS are

- Highly modularised architecture to encapsulate functionality
- File system which is easy to navigate through using abstractions of common Unix commands
- Integrated build system
- Package manager to easily integrate existing packages into the system

Not only the ROS-framework is modularised. ROS offers a wide range of concepts to develop modularised software. In ROS, functionality is encapsulated in *nodes*. The nodes a software system consists don't communicate directly with each other. Instead, ROS uses an asynchronous publish/subscribe communication infrastructure as well as a synchronous service architecture. The information provided by a node gets published to a topic other nodes can subscribe to. If a node provides functionality (e.g. calculating a trajectory of a robot), this functionality is offered by the node using a service. Other nodes can call that service to access the functionality a node provides. The location of a node is not fixed: All nodes can be moved to a different client in a network. This makes it easy to develop distributed systems with nodes running on different machines.

For a full feature-list and detailed descriptions of ROS, the reader is referred to the ROS website: `http://www.ros.org/`

### 3.3.3 Overview

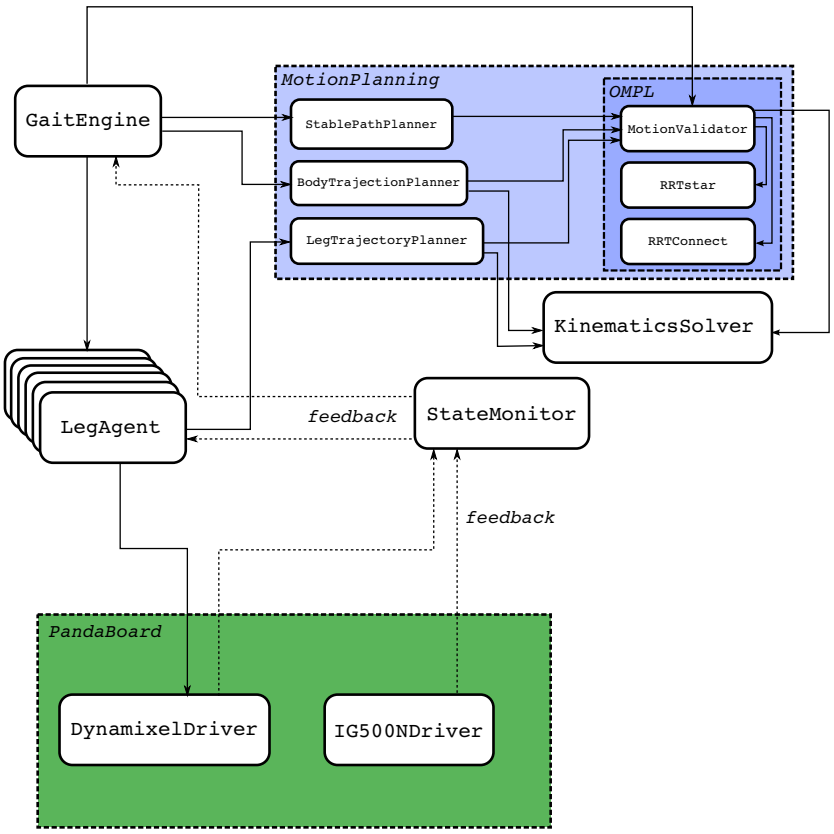



**Figure 3.3:** Overview of the software architecture. The blocks are ROS nodes. The solid lines between components indicate that one node accesses the other using service calls. The dashed line indicates a publish/subscribe relation between two nodes, while the node on the arrow side consumes messages published by the node on the other end of the line using ROS message queues.

### 3.3.4 Components

GaitEngine

The *GaitEngine* component is responsible to take user inputs in the form of goal coordinates for the body in world-coordinates. It generates gaits using the *BodyTrajectionPlanner* and delegates these gaits to the *LegAgents*. The *GaitEngine* keeps track of the current state of the system using a *MotionValidator* provided by the OMPL package. It can interrupt the current execution of a gait and delegate new movement targets to the *LegAgents*

LegAgent

The *LegAgents* are an abstraction of each of the hexapod's leg. They keep track of the current state of each leg by getting feedback from the *StateMonitor* component. The *LegAgents* receive tasks from the *GaitEngine* and use the *LegTrajectionPlanner* component to generate leg movements to perform these tasks.

MotionPlanning

The motion planning component consists of several parts responsible for planning any kind of motion the hexapod platform can perform.

BodyTrajectionPlanner

The *BodyTrajectionPlanner* uses the current position $p_{current} = (x\ y\ z)^T$ of the hexapod in world coordinates and a user-defined goal position $p_{goal} = (x_{goal}\ y_{goal}\ z_{goal})$ to plan a path from $p_{current}$ to $p_{goal}$ the body has to follow. To achieve this, it uses *OMPL* and its *MotionValidator* component to plan a trajectory in the hexapod's environment. This trajectory is then converted into foothold positions and returned to the *GaitEngine*.

LegTrajectionPlanner

The *LegTrajectionPlanner* generates paths for the tip positions of the legs to bring them from a start position $t_{start} = (x\ y\ z)$ to a goal position $t_{goal} = (x_{goal}\ y_{goal}\ z_{goal})$ (expressed in the local leg-frame coordinates, see 4.2). In the simplest case, this tip-path is a cubic polynomial, describing a stride movement from $t_{start}$ to $t_{goal}$. However, it can use local terrain information to plan paths for the tip position to avoid these obstacles. When the leg is in support phase, the planned tip path is simply a straight line from $t_{start}$ to $t_{goal}$ (which keeps the tip positions stationary on the ground, but "pushes" the body towards the desired direction). After the tip-trajectories have been calculated, the *LegTrajectionPlanner* discretises these paths and calculates the joint angles for each sample. These angles are then returned to the *LegAgent*.

OMPL

OMPL (Open Motion Planning Library) is an open-source software package which includes implementations of common motion-planning algorithms. It is not tied to any collision-checking, environmental or visualisation specifications and can therefore be flexibly used in any motion-planning environment. The package includes many state-of-the-art motion-planning algorithms as well as the possibility to define new algorithms due to it's flexible architecture. OMPL is implemented in C++, although Python bindings exists as well. Since OMPL offers ROS bindings, the package can be easily integrated into existing ROS systems.

The ROS system developed for this thesis utilises OMPL and integrates the *RRT*, *RRT*∗ and *RRTConnect* algorithms into the stabilisation framework. It also uses OMPL to define a custom *MotionValidator*.

MotionValidator

The *MotionValidator* is an custom implementation of OMPL's abstract MotionValidator class. It's purpose is to validate motions generated by the *BodyTrajectionPlanner*, *LegTrajectionPlanner* and the *GaitEngine*. Since the navigation problem is not addressed in this thesis, the planned path is always a straight line from $p_{current}$ to $p_{goal}$. However, the *MotionValidator* is implemented in a way which makes it easy to include terrain or vision information to plan a path for the hexapod to avoid obstacles in the environment the hexapod operates in.

StateMonitor

The *StateMonitor* receives the current servo and IMU state at a rate of 35 Hz, converts the servo positions into joint angles and the IMU orientation information into Tait-Bryan-angles and publishes that information to the *GaitEngine* and *LegAgents*. It also uses the *MotionValidator* to determine the current state of the system (*stable*, *critical* and *unstable*).

KinematicSolver

The *KinematicSolver* is a node providing services to solve the forward and inverse kinematic problem. It uses the Denavit-Hartenberg convention (see 2.3) to solve the forward kinematic problem and an analytical solver as described in 4.5.2 for the inverse kinematic problem.

DynamixelDriver

The *DynamixelDriver* is a low-level driver which directly communicates with the servo-controllers using a serial bus. It provides services and message queues to interact with the servo-controllers. This includes setting a goal-position, speed, torque and several other parameters of the servos. Furthermore, the *DynamixelDriver* constantly queries the servos to receive their current state (the query rate is around 35 Hz). This state is then published to expose the servo state to the higher-level components.

IG500NDriver

THe *IG500NDriver* is a low level driver which communicates with the SBG-IG500N IMU. It reads the raw values (orientation, velocity, and acceleration data) of the IMU and publishes it at a rate of 100 Hz. Note that only the orientation data is used by the higher level system.

The orientations gets published in quaternion form which is converted to Tait-Bryan-angles using the method described in 2.6.1.

# 4 Hexapod model

This chapter describes the mathematical model of the hexapod platforms used in this thesis. This includes a full kinematic description as well as a definition of the local frames of the components and their relationship amongst each other.

## 4.1 Body model

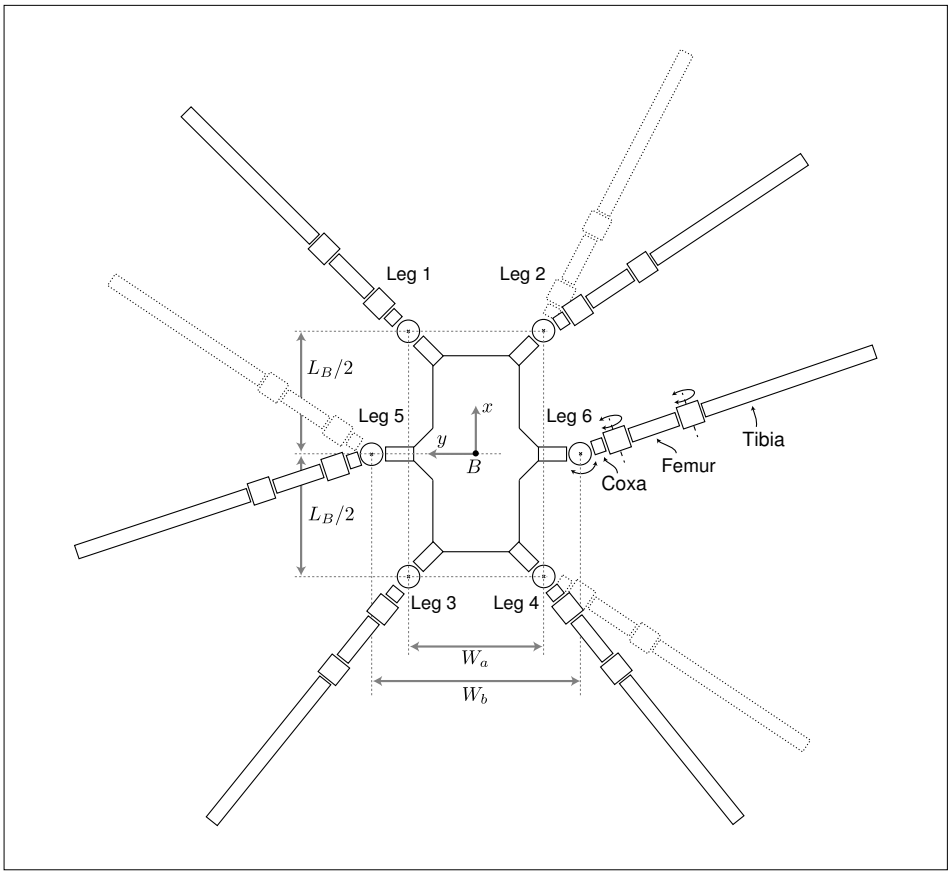



**Figure 4.1:** Hexapod model

The hexapod consists of six legs connected to a rigid body via coxa joints. The total body width is $W_a$. $W_a$ is the distance between the middle-left and middle-right coxa joints. The width $W_b$ describes the distance between the front-left and front-right (or rear-left and rear-right) coxa joint. The total body length is $L_b$, defined by the distance between the front-left and rear-left (front-right and rear-right) coxa joints. The 3-dimensional Cartesian body frame $B$ lies in the centre of the body with its $x$-axis pointing forward, its $y$-axis pointing to the left and its $z$-axis pointing upwards (right-hand rule).

## 4.2 Leg model

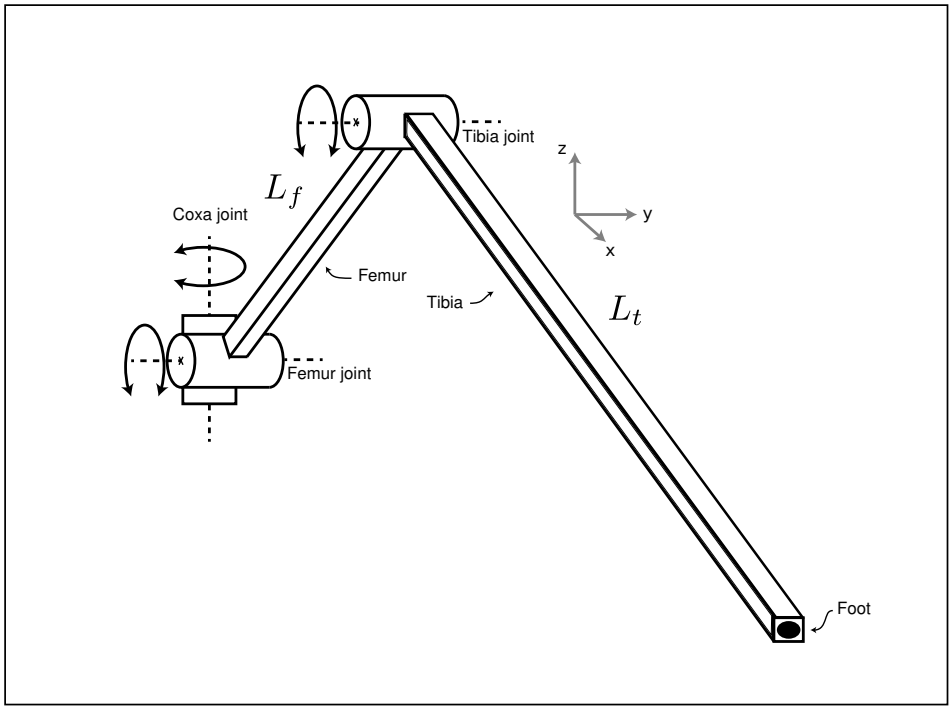



**Figure 4.2:** Hexapod leg model

A leg (4.2) consists of three links. the coxa link $L_c$, the femour link $L_f$ and the tibia link $L_t$ with an end effector $e_t$. $L_c$ is connected to the body via a coxa joint $j_c$, the femour link is connected to the coxa link via a femour joint $j_f$ and the tibia link is connected to the femour link via a tibia joint $j_t$. Each joint has a local reference frame $F_j$, with the $z$-axis being the joint's rotation axis and the $x$ and $y$ axis forming a right-hand-system together with the $z$-axis. The local reference frame of the coxa joint serves also as the leg frame $F_l$ of the corresponding leg. Since the coxa joints for the front legs are rotated by $-45°$ degree (front-right leg) and $45°$ degree (front-right leg), their local frame $F_L$ is rotated relative to $B$ accordingly. The same applies for the rear legs ($135°$ degree for the rear-left and $-135°$ degree for the rear-right leg)

and the middle legs (90° degree for the middle-left and −90° for the middle-right leg).
Each link $L_i,\ i \in c, f, t$ has a fixed length $l_i,\ i \in c, f, t$.

## 4.3 Definition of the joint angles

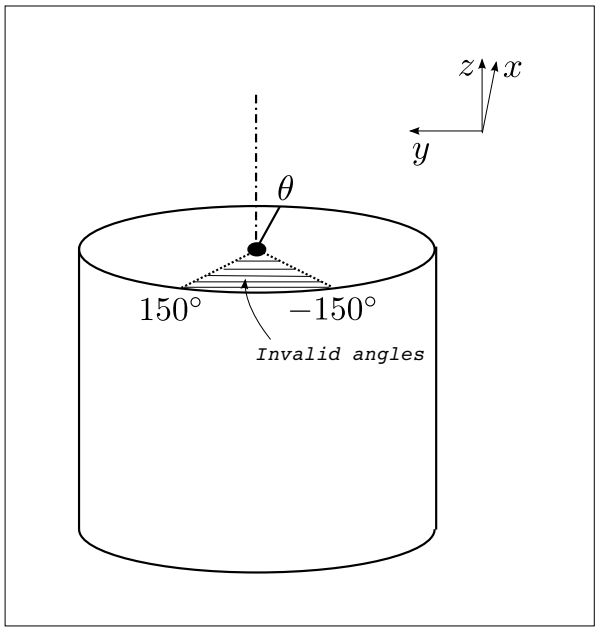



**Figure 4.3:** Model of a joint

The local coordinate frames $F_{j,i},\ i \in \{c, f, t\}$ of the revolute joints are three-dimensional systems following the right-hand rule with the $z$-axis being the rotation axis of the joint. $\tilde{0}$ denotes the the zero position of the joint where the joint angle $\theta = 0$. The rotation angle of a joint is limited to operate within $\{-150°, 150°\}$.

## 4.4 Transformation tree

The model consists of several local frames: The joint frames $F_{j,i}$, the leg frames $F_{L,j}$, as well as the body frame $B$. In order to model the relationship between those frames, they are stored in a tree structure, where each node consists of a local coordinate frame and a link between two nodes represents a transformation between those two nodes. Note that the transformation between two local frames requires homogeneous coordinates.
The root of the tree is a fixed world frame $W$ with the origin being $o_w = (0, 0, 0)^T$.
Figure 4.4 shows the structure of the transformation tree.

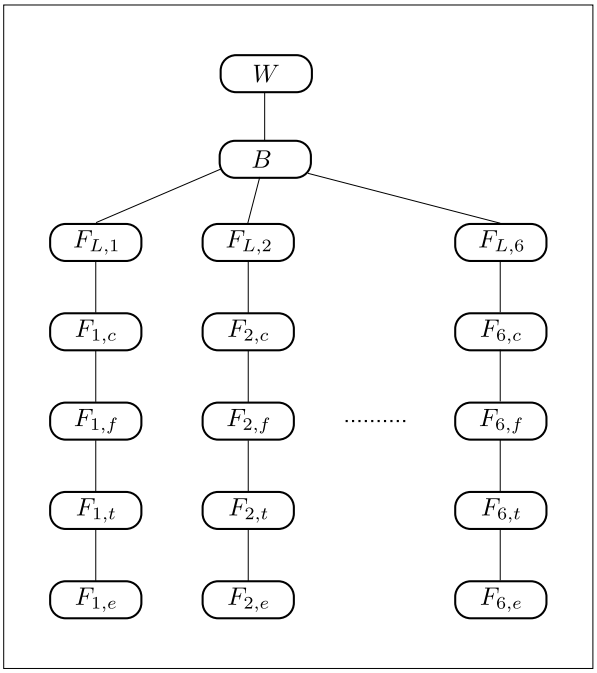



**Figure 4.4:** The tree structure used to represent the relationships between the single local coordinate frames

Since this thesis doesn't cover the problem of navigation of hexapods, the transformation $T_W^B : W \to B$ is a fixed transformation with

$$T_W^B = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.1}$$

The transformations $T_B^{F_{L,i}}$ between the body frame $B$ and the leg frames $F_{L,i}$ uses the geometry of the body and the location of the coxa joints relative to the centre of the body.
The origin of the coxa joint of leg 1 relative to $B$ is achieved by translating $o_B$ (the origin of $B$) $\frac{L_B}{2}$ in $x$ and $\frac{W_a}{2}$ in $y$-direction and rotating it by 45°. Therefore the transformation $T_B^{F_{L,1}}$ from $B$ to $F_{L,1}$ is defined as

$$T_B^{F_{L,1}} = R(\frac{\pi}{4})T(\frac{L_B}{2}, \frac{W_a}{2}) \tag{4.2}$$

$R(\frac{\pi}{4})$ is a rotation a rotation matrix and $T(\frac{L_B}{2}, \frac{W_a}{2})$ a translation matrix.
The transformation between two local joint frames $F_{j,i}$ can be expressed as a Denavit-Hartenberg transformation (see 2.3)

## 4.5 Leg kinematics

### 4.5.1 Forward kinematics

While the rotation axis of the coxa joints $j_c$ is the $z$-axis of $F_{L,i}$ (the frame of the leg, the joint belongs to), the rotation axes of $j_f$ and $j_t$ are orthogonal to the y-z-plane of $F_{L,i}$, in case the joint angles of all three joints are 0. Let $\theta_i$ be the angular displacement of joint $j_i$ and $l_j,\ j \in \{c, f, t\}$ the length of link $L_j$. We can determine the position $p_i$ of the end-effector of leg $i$ relative to the body frame $B$ by applying a combined transformation to the origin $o_B$ of $B$:

$$D_l([l_j], [\theta_i]) = \begin{bmatrix} R & d \\ 0 & 1 \end{bmatrix} = T_B^{F_{L,i}} D_1 D_2 D_3 \tag{4.3}$$

$T_B^{F_{L,i}}$ is the transformation from the origin of $B$ to the leg frame $F_{L,i}$. $D_1$ is the DH-transformation from the coxa joint to the femour-joint $j_i^1$, $D_2$ the DH-transformation from the femour joint to the tibia-joint $j_i^2$ and $D_3$ the DH-transformation from the tibia-joint to the end effector.
$T_B^{F_{L,i}}, D_1, D_2, D_3$ have the form

$$T_B^{F_{L,i}} = \begin{pmatrix} 0 & -1 & 0 & \frac{1}{2} \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.4}$$

$$D_1 = \begin{pmatrix} \cos\theta_1 & 0 & \sin\theta_1 & l_1\cos\theta_1 \\ \sin\theta_1 & 0 & -\cos\theta_1 & l_1\sin\theta_1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.5}$$

$$D_2 = \begin{pmatrix} \cos\theta_2 & -\sin\theta_2 & 0 & l_2\cos\theta_2 \\ \sin\theta_2 & \cos\theta_2 & 0 & l_2\sin\theta_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.6}$$

$$D_3 = \begin{pmatrix} \cos\theta_3 & -\sin\theta_3 & 0 & l_3\cos\theta_3 \\ \sin\theta_3 & \cos\theta_3 & 0 & l_3\sin\theta_3 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.7}$$

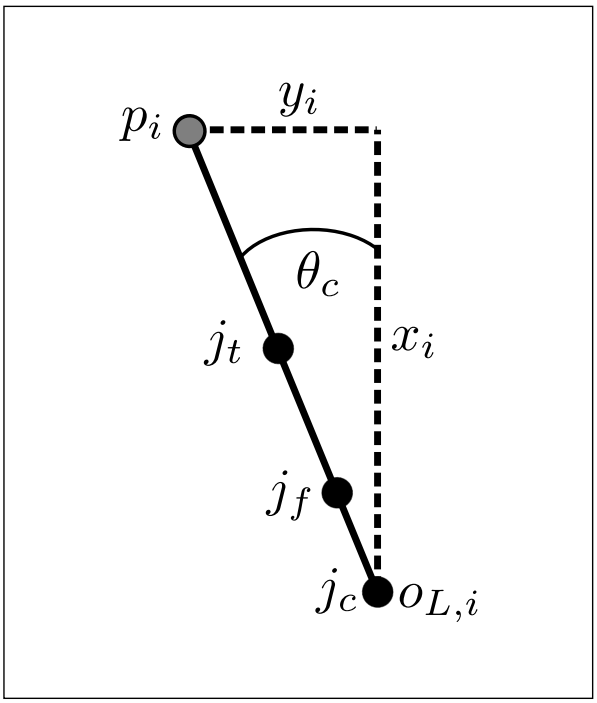
Combining these transformations with

$$D_i = T_B^{F_{L,i}} D_0 D_1 D_2 D3 \tag{4.8}$$

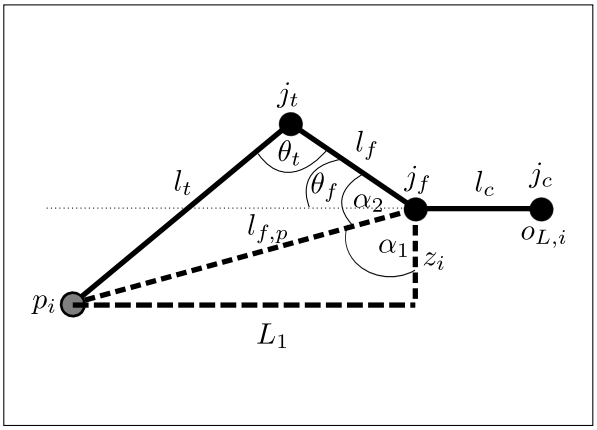
gives the end-effector position $p_i$ of leg $i$ relative to the body frame $B$, by multiplying $D_i$ with the origin $o_B$ of $B$:

$$p_i = D_i o_B \tag{4.9}$$

### 4.5.2 Inverse-kinematic solution




(a)




(b)

**Figure 4.5:** Inverse-kinematic model of the hexapod legs

Since the legs of the hexapod platform used in this thesis consist of only three links connected by three joints, it it possible to find a trivial analytical solution for the inverse kinematic problem.
Let $F_{L,i}$, called the leg-frame of leg $i$, be a frame with it's origin $o_{L,i}$ being in the centre of the coxa joint $j_c$ of the leg we want to find a solution for the inverse kinematics problem for. Furthermore, let $p_i = (x_i, y_i, z_i)$ the coordinates of the end-effector expressed in $F_{L,i}$. To calculate the coxa angle $\theta_c$ we calculate the angle between the $x$-axis of $F_{L,i}$ and the vector from $o_{L,i}$ to $p_i$:

$$\theta_c = arctan\frac{y_i}{x_i} \tag{4.10}$$

This reduces the inverse kinematic problem into a two dimensional problem.
Next we calculate the distance $l_{o,p}$ from $j_f$ to $p_i$:

$$l_{f,p} = \sqrt{z_i^2 + (l_1 - l_c)^2} \tag{4.11}$$

with $l_c$ being the length of the coxa link and $l_1$ being an imaginary line-segment from $j_f$ to $p_i$ in the $x, y$-plane defined as

$$l_1 = \sqrt{x_i^2 + y_i^2} \tag{4.12}$$

This give us $\alpha_1$:

$$\alpha_1 = arccos\frac{|z_i|}{l_{f,p}} \tag{4.13}$$

$\alpha_2$ can be calculated using the cosine rule:

$$\alpha_2 = arccos\frac{{l_t}^2 - {l_f}^2 - l_{f,p}}{2 l_f l_{f,p}} \tag{4.14}$$

With $\alpha_1$ and $\alpha_2$ the joint angle $\theta_f$ can be calculated:

$$\theta_f = \alpha_1 + \alpha_2 - 90^\circ \tag{4.15}$$

The angle $\theta_t$ can be calculated using the cosine rule as well:

$$\theta_t = 180^\circ - arccos\frac{{l_{f,p}}^2 - {l_t}^2 - {l_f}^2}{2 l_t l_f} \tag{4.16}$$

# 5 Stability-margins

In this chapter, an overview over common stability in legged locomotion is given. When a walking system has to be "stable", stability has to be defined in order to be able to distinct a stable state from an unstable state, using a stability function $d$. $d$ can be a mapping of the current state $c$ of the system to the set $\{0, 1\}$ such that

$$S(c) = \begin{cases} 0 & \text{c is unstable} \\ 1 & \text{c is stable} \end{cases} \tag{5.1}$$

In general, there are two categories of stability: Static and dynamic stability

## 5.1 Static stability

A legged vehicle is called statically stable when it is able to stand on the ground without violating the stability constraints. In other words, a hexapod is statically stable, when the projection of the centre of mass (CoM) is inside polygon defined by the tip positions of the support legs (support polygon) [MI79]. Figure 5.1 shows the model of a hexapod during a tripod gait with the projection of the CoM on the support polygon.

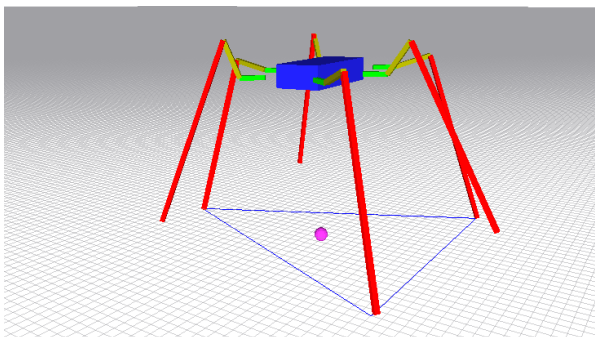


**Figure 5.1:** A model of a hexapod during a tripod locomotion. The blue vertices define the support polygon. The violet sphere is the projection of the CoM on the support polygon

## 5.2 Calculation of the CoM projection

To calculate the coordinates of the CoM projection onto the support polygon, it is sufficient to reduce this problem to a line/plane intersection problem. Let $o_B = (0\ 0\ 0)^T$ being the origin of the CoM and $v_0 = (0\ 0\ -1)^T$ being a vector defined in the body frame $B$. Furthermore, let $e = (r\ p\ y)$ be the orientation of the body relative to the world frame $W$, expressed in Tait-Bryan-angles.
First, $v_0$ is rotated around $o_B$ with $-e$ in order to make it perpendicular to the world frame's $x, y$-plane. The resulting vector is $\tilde{v_0}$. The problem is now to determine if the intersection point $I_p$ of the line which goes through $o_b$ and $\tilde{v_0}$ and the plane $P$ the polygon lies in, is inside the support polygon. Standard algorithms given in [Bad90] or [O'R98] project the support polygon and $I_p$ into a two dimensional plane and test for inclusion. However, these methods are relatively slow, since a projection has to be found which avoids degeneration.
Therefore the method proposed in [MT97] is used, which solves the inclusion problem directly in 3D-space. For further details, the reader is referred to [MT97].

Even though this constraint ensure static stability, it is not enough to ensure that the system is capable of performing it's task. A hexapod might rest on it's belly with all six legs being spread out and in support phase. In this case the above stability margin is still fulfilled, even though the hexapod is not able to perform it's task (performing a gait). Therefore additional stability constraints are necessary. These additional constraints are defined in 6.6.1

## 5.3 Dynamic stability

Dynamic stability margins are not being used in this thesis, so only a brief overview will be given.
Most dynamic stability criteria are moment-based criteria. When a legged vehicle is about to topple-over, it's moments are exceeded about a toppling-axis. [LS01] introduced the concept of a *Dynamics Stability Margin (DSM)* which takes the external, foot, gravitational and inertial forces and moments imparted on the tip-positions into account and calculates the minimum resultant moment about the boundaries of the support polygon. If the resulting moment about one support polygon boundary is negative, the vehicle is about to topple over this boundary. Another dynamic stability criteria is a zero-moment-point-based criteria [TT90].

# 6 State-space

This section gives an overview of the state-space used for the proposed stabilisation approach. It defines the state-space, including its sub-spaces and classifications of different state types inside the state-space (stable, critical and unstable states) and their properties.

## 6.1 State-space representation

The state-space $S$ of a dynamical system is the set of states $\{s_i\}$ the system can take. Each state of the system corresponds to a unique point in $S$. The state space $S$ of the hexapod platform used in this thesis is an 21 dimensional space, containing 18 dimensions for the joint angles (the configuration space $C \subset S$, see 6.2) and 3 dimensions for the body orientation expressed as Tait-Bryan angles.

## 6.2 Configuration space

The configuration space $C$ of a dynamical system is a $n$-dimensional manifold inside $S$. It consists of the set $\{c_i\}$ of all possible configurations of the system. $n$ is the number of degrees of freedom of the system. In the case of the hexapod platform with 18 servo joints, $C$ is an 18-dimensional manifold. Note that $dim(C) \leq dim(S)$ since $S$ can contain arbitrary, non-controllable or indirectly controllable dimensions (e.g. the body orientation, which can be indirectly controller by changing the system's configuration), while $C$ consists of all controllable degrees of freedom. $C$ is a topological space, which allows us to define distance metrics within $C$. $C$ is usually constraint by the constraints of each dimension. (e.g. the minimum/maximum angle of a joint).
Consider a kinematic chain consisting of three links connected by three joints with a minimum joint angle of $-150$ and a maximum joint angle of 150 degrees. Figure 6.1 shows the configuration manifold of that system sampled by 60000 states:

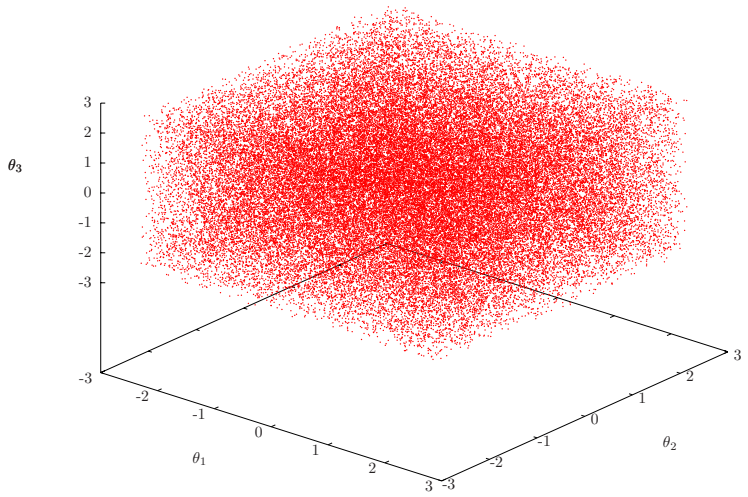



**Figure 6.1:** Configuration manifold of a kinematic chain with three joints. The axes are the joint angles for each joint of the chain (in radians).

The configuration space used in this thesis is the 18-dimensional joint space of the hexapod defined as

$$C = \left\{ c_i : \frac{-150 \cdot \pi}{180} \leq c_i^n \leq \frac{-150 \cdot \pi}{180} \right\} \tag{6.1}$$

where $c_i^n$ is the $n$-th component of $c_i$, and $c_i$ is an 18-dimensional vector. Each $c_i^n$ represents the angle of the $n$-th joint.

## 6.3 Task-space

A task-space (also called *workspace*) of a system is the space, the system operates in. For the three-link kinematic chain defined in 6.2, this could be a 6-dimensional euclidean space $\mathbb{R}^6$ where the 6 dimensions are end-effector position and orientation of the end-effector relative to a reference frame $F$.

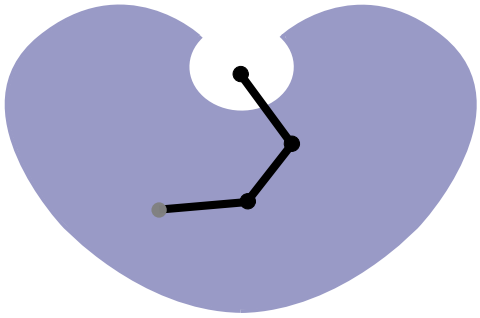


**Figure 6.2:** The task-space of a three-link kinematic chain. The blue area is the set of feasible end-effector positions.

Note that the task-space of a system can be arbitrarily extended in it's dimensionality. It can include dynamics and other task-related properties of the system. For a hexapod, the task-space can consist of the foot-tip positions and orientations of all legs relative to a frame fixed to the centre of the body, as well as the body position and orientation relative to a world-frame $W$, the foot-tip velocities, the friction force imparted on the foot-tips, or (as used in this thesis) stability constraints of the hexapod.

## 6.4 Constraint manifolds

Constraint manifolds are defined in the system's state space and consist of the set of states the system can take. There are some important properties of constraint manifolds which have to be taken into account when planning inside constraint manifolds. Constraint manifolds are usually non-convex. Thus, when planning a path in $S$-space from a start state $s_{start}$ to a goal state $s_{goal}$, a direct connection between those states does not always exist. Therefore, a planner has to search inside the manifold for a valid path from $s_{start}$ to $s_{goal}$. Another important property of constraint manifolds is that it can consist of disjoint parts. In that case, there is no valid path from $s_{start}$ to $s_{goal}$ when both states are inside different disjoint parts of the manifold. A planner, which generates a tree structure inside the constraint manifold (such as RRT, see 7.1) can only generate trees inside one part of the manifold.

Since, in general, it is difficult to define a analytical representation of a constraint manifold, techniques are required to efficiently sample constraint manifolds as well as, for a given state $s$, deciding if $s$ lies within the manifold.

## 6.5 Sampling strategies

There are several sampling strategies being used to sample a constraint manifold. One simple sampling strategy is called *rejection sampling*: This technique samples the entire $S$-space

uniformly and uses an evaluation function $I$ to determine if the sampled state $s$ is inside a manifold $M$ or not:

$$I(s) = \begin{cases} 0 & \text{s is inside M} \\ 1 & \text{s is outside M} \end{cases} \tag{6.2}$$

If $I(s) = 0$, the sample is rejected and a new sample inside $S$ is generated. This is repeated until a configuration $s$ has been found such that $I(s) = 1$.
This sampling technique is efficient when $M$ covers a significant volume of $S$. In order to determine if a sample lies within $M$, the concept of *task space constraints* is used (see 6.6).

Other sampling techniques use a gradient-descent projection methods: First, $S$ is sampled uniformly. If the sampled configuration $s$ lies outside $M$ it is projected on the boundary of $M$ (needs some further details).

## 6.6 Task-space constraints

Constraints of a system defined in its task-space $T$ are called task-space constraints. These constraints can be defined in a more intuitive manner compared to $S$-space constraints. For a hexapod system, task-space constraints can for example include the workspace of the foot-tips, body position, orientation and velocity constraints, or any other task-specific constraint.
More importantly in this scope, task-space constraints can be used to define stability criteria. These stability criteria can then be utilised by a reactive stabilisation system, preventing the platform to get into a state where it is no more able to perform its task (see 8).
These task-space constraints mapped into $S$ define the boundaries of a task-constraint manifold $S_s \subset S$. However, in general there exists no analytical mapping $M_T^S$ from $T$ to $S$, or $M_T^S$ has an infinite number of solutions when mapping a state $t \in T$ to $S$. Therefore it's difficult to map task-space constraints into $S$.
To overcome this issue, in this thesis a concept called *task-space-regions* [BSK11] is used, utilising the mapping $M_S^T$: When checking if a state $s \in S$ is valid according to the task-space constraints (or in other words, lies within the constraint manifold $S_s$), it is mapped into task-space using $M_S^T$ such that

$$M_S^T(s) = t \in T \tag{6.3}$$

and $t$ is evaluated using the task-space constraints. An example of a mapping $M_S^T$ is the DH-transformation defined in 2.3

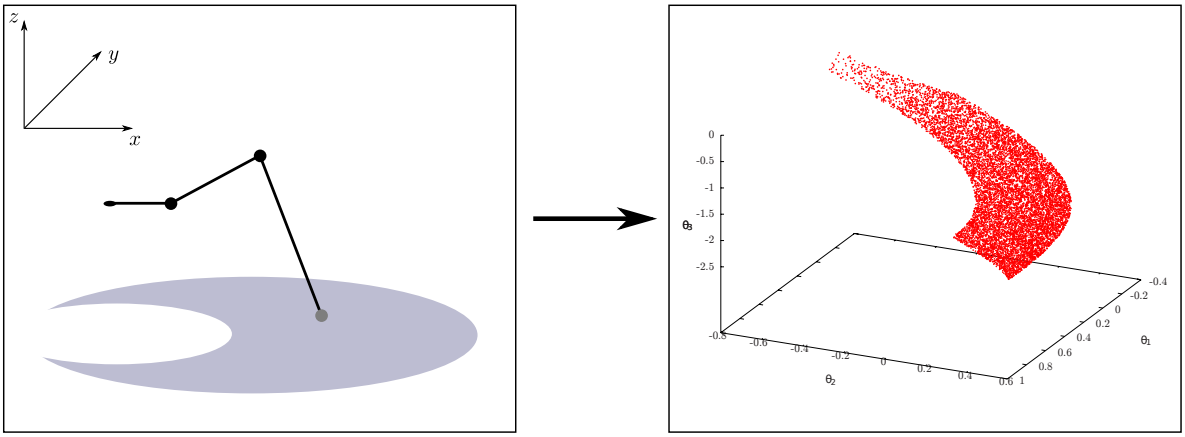



**Figure 6.3:** Projection of task-space constraints into the $C$-space of a three-link manipulator. The left figure shows the constraint for the end-effector position (the end-effector has to move inside the blue area). The right figure shows the related constraint manifold in the manipulator's $S$-space.

### 6.6.1 Definition of the task-space constraints

As mentioned in 5.1, the stability margin defined in [MI79] is not sufficient to keep the vehicle in a manoeuvrable state. In this thesis, additional stability constraints are defined to overcome this issue. These constraints are entirely defined in task-space and include

- The foot-tips have to remain inside a specific area.
- The body has to maintain a certain distance to the support polygon.
- The orientation of the body has to be within certain angular constraints.

The definition of these constraints are as follows:

(a) Workspace of the foot tips

Each tip has an initial Cartesian coordinate $o_i$, expressed in the body frame $B$. The workspace of each tip is defined as

$$w = \begin{pmatrix} x - x_b, & x + x_b \\ y - y_b, & y + y_b \\ z - z_b, & z + z_b \end{pmatrix} = \begin{pmatrix} w_x^-, & w_x^+ \\ w_y^-, & w_y^+ \\ w_z^-, & w_z^+ \end{pmatrix} \tag{6.4}$$

with $\begin{pmatrix} x & y & z \end{pmatrix}^T = o_i$ and $\begin{pmatrix} x_b & y_b & z_b \end{pmatrix}^T$ being the maximum distance in each dimension the tip is allowed to displace from its initial position $o_i$.

(b) Distance of the body to the support polygon

During locomotion, the support polygon is calculated (by calculating the tip-positions of the support legs). Let $d_p = \begin{pmatrix} d_p^x & d_p^y & d_p^z \end{pmatrix}$ be the coordinate vector of the vertical projection of the body's centre of mass onto the support polygon expressed in $B$. Then $\left|d_p^z\right|$ has to be within

(6.5) $d_l \leq d_p^z \leq d_h$

This regulates the distance of the body to the support polygon. $d_l$ and $d_h$ are the lower and upper bounds of $d_p^z$.

(c) Orientation limits of the body

Let $e = \begin{pmatrix} r & p & y \end{pmatrix}$ be the roll, pitch and yaw rotation of the body about its body frame $B$, expressed as Tait-Bryan-angles. $r$ and $p$ have to be within

(6.6) $\sigma_l \leq a \leq \sigma_h$

with $a \in \{r, p\}$. and $\sigma_l, \sigma_h$ are the bounds of the body's orientation. The yaw-angle has no effect on the hexapod's stability and is therefore ignored.

These task-space constraints, along with the static stability margin form a set of constraints in which the hexapod maintains a safe distance to the ground and keeps the system in a manoeuvrable state.

## 6.7 Stable states

A state $s$ is called stable, when it's inside a defined stability manifold $S_s \subset S$. This manifold is bounded by the defined task-space constraints mapped into $S$. As mentioned in 6.6, it is difficult to map these constraints from $T$ into $S$. Instead, the transformation $I_S^T$, which maps a state $s$ into the task-space $T$ is utilised. Thus, a state $s$ is table when the mapping $I_s^T(s)$ fulfils the task-space constraints defined in 6.6.1, and unstable, when at least one of the task-space constraints are violated.

## 6.8 Critical states

The set of critical states $S_c \subset S_s$ is a manifold defined as

(6.7) $S_c = \{s \in S_s : |s - s_{bound}| < \zeta, s_{bound} \in B_c\}$

with $B_c$ being the set of states which lie on the boundaries of $S_s$. In other words, a state is called *critical*, if its distance to the boundaries of the stability manifold $S_s$ is smaller than a

certain threshold $\zeta$. To check, whether the current state of the system is critical, the concept of task-space constraints can be used as well, using the same task-space constraints that define $S_s$, but with different constraint areas. A state $s$ is called *critical* if $s \in S_s$ and at least one of the following conditions are met:

1. Let $w_i = (w_x, w_y, w_z)$ be the workspace of each tip position of leg $i$ as defined in 6.6.1. The critical area $a_i$ of the tip-position of leg $i$ is defined as

   $$a_i = \left\{ \begin{pmatrix} x & y & z \end{pmatrix} \in \mathbb{R}^3 : \begin{pmatrix} x < w_x^- + 20 \vee x > w_x^+ - 20 \\ y < w_y^- + 20 \vee x > w_y^+ - 20 \\ z < w_z^- + 20 \vee x > w_z^+ - 20 \end{pmatrix} \right\} \tag{6.8}$$

   If the tip-position of at least one leg lies within that critical area, the current state of the system is critical.

2. Let $e = \begin{pmatrix} r & p & y \end{pmatrix}$ be the roll, pitch and yaw rotation of the body about its body frame $B$. If

   $$a > 5.0 \vee a < -5.0 \tag{6.9}$$

   with $a \in \{r, p\}$, then the current state of the system is critical

3. Let $d_p = \begin{pmatrix} d_p^x & d_p^y & d_p^z \end{pmatrix}$ be the coordinate vector of the vertical projection body's centre of mass (CoM) onto the support polygon, expressed in $B$. If

   $$d_p^z > -250 \vee d_p^z < -500 \tag{6.10}$$

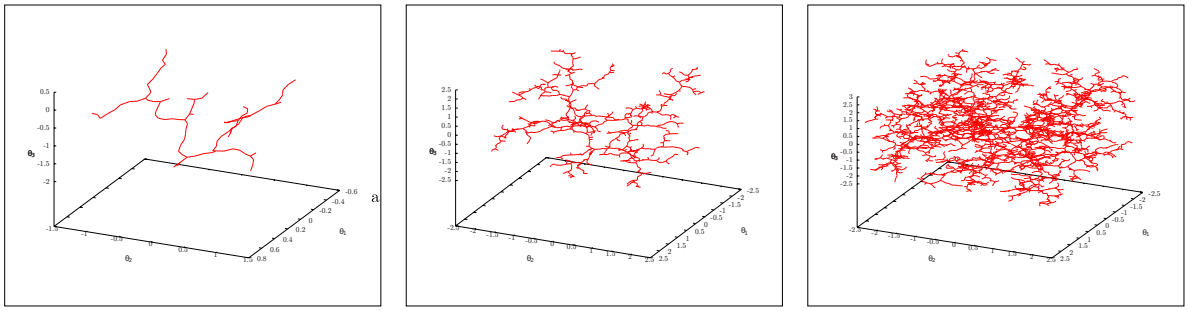
   then the current state of the system is critical (or in other words: $s \in S_c$).

# 7 Random sample based planning techniques

This chapter gives an overview about sample-based planning techniques used for the proposed stabilisation method. It describes the basic RRT-algorithm and discusses state-of-the-art extensions for RRT which significantly improve the performance of the basic RRT-algorithm.

## 7.1 RRT

RRT (Rapid exploring random tree) is an algorithm introduced by Lavalle and Kuffner [Lav98]. RRT has been designed to efficiently search high-dimensional spaces by building a random tree from a start configuration within the search space. The idea is to grow a tree, rooted at a rood node into large, yet unreached areas in order to quickly cover the whole search space by the tree.
Starting from a root node, the algorithm creates a random sample in the search space and finds the closest node in the search tree. If there's a feasible connection between the closest node and the random sample (the connection obeys any constraints), the random sample is added to the tree with the closest node being the parent of the random sample.
The probability of expanding an existing node is proportional to the size of it's Voronoi region. Therefore the tree expands to large, unreached areas with a higher probability than to smaller undiscovered areas, which leads to a vast coverage of the entire search space.
In most cases the length of the connection is limited by a growth factor. This factor can account for the physical constraints of the system the state space the search is performed for, for example the maximum displacement of a joint within a time-step $\Delta t$. If this is the case, a random sample is not directly added to the tree, but the closest node in the three is expanded towards the random sample such that the new sample $q_{new}$ added to the tree lies between the nearest node $q_{near}$ and the random sample and the distance $dist(q_{new}, q_{near})$ satisfies the connection constraints.
Figure 7.1 shows an RRT-tree after 100, 1000 and 5000 iterations:



a b c

(a)

**Figure 7.1:** RRT tree after 100 (a), 1000 (b) and 5000 (c) iterations.

The basic RRT-algorithm is shown in 7.1

---

**Algorithmus 7.1** Build RRT

---

**Require:** *Initial configuration* $q_{init}$, *number of max vertices* $K$, *expansion constraint* $\Delta q$

$T \leftarrow \{q_{init}\}$

**for** i=0 to K **do**

$q_{rand} \leftarrow RANDOM\_SAMPLE()$

$q_{near} \leftarrow NEAREST\_NODE(q_{rand}, T)$

$q_{new} \leftarrow NEW\_NODE(q_{near}, \Delta q)$

$T.add\_node(q_{new})$

$T.add\_edge(q_{near}, q_{new})$

**end for**

**return** $T$

---

### 7.1.1 Properties of RRT

The RRT is an efficient random-sample based search algorithm for high dimensional space. Since the probability of sampling a goal state approaches 1 if the number of iterations goes towards infinity, the algorithm is called probabilistic complete. However, the basic RRT algorithm has some fundamental drawbacks. In general it is very unlikely to sample a defined goal state, or the algorithm takes a large number of iterations until the tree reaches the goal position. The probabilistic completeness ensures that a path from a start state to a goal state will be found, if both states are not within disjoint areas of the manifold. However, even though RRT is guaranteed to find a path if one exists, it is not guaranteed that RRT finds an optimal solution regarding an optimality function $O$ (an example of an optimality function would be $O(path) = l$, with $l$ being the path length).
Another drawback of the basic RRT-algorithm is the fact that it performs poorly if the constraint manifolds in the search space form narrow passages. Therefore a couple of improvements have

been developed to deal with those drawbacks without losing the probabilistic completeness of the basic algorithm.

### 7.1.2 Extensions of the RRT-algorithm

Goal biasing

The performance of the standard RRT can be greatly improved by biasing the growth of the tree towards the goal state. This can be accomplished by choosing the goal state as the random sample with a probability of $p$ and choosing a random sample uniformly within the search space with a probability of $p-1$. $p$ is usually small (between 0.01 and 0.1), which ensures that RRT maintains its explorative behaviour and preventing it to get stuck in traps. Figure 7.2 shows an example of a trapped node, when planning a path from $q_{start}$ to $q_goal$ in a manifold $M$ with an obstacle $M_i$, if $p$ is chosen too greedily.

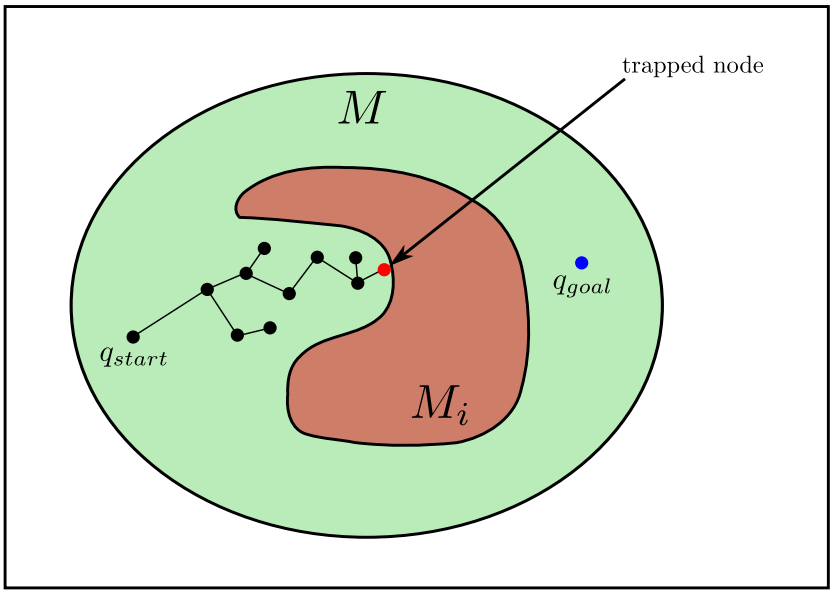



**Figure 7.2:** A RRT-tree gets trapped behind an obstacle $M_i$ when the $p$ is too high.

However, depending on the shape of the manifold RRT tries to find a path from a start state $q_{init}$ to a goal state $s_{goal}$ in, $p$ can be increased to achieve a more greedy behaviour with a better performance, since a fewer number nodes have to be sampled before RRT finds a path from $q_{start}$ to $q_{goal}$. Figure 7.3 shows the path from a start state $q_{start}$ and the generated RRT-tree, while figure 7.4, shows the path for the same start and goal states using goal biasing.

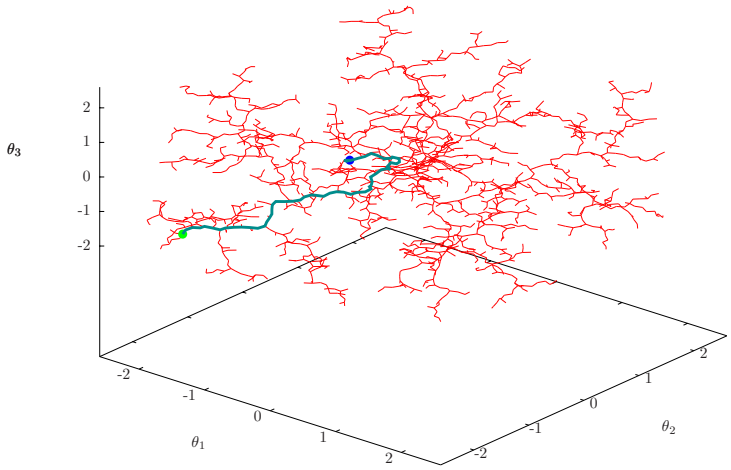



**Figure 7.3:** RRT-tree in a 3 dimensional space with a goal biasing value of 0.001. The path from the start node (green dot) to the end node (blue dot) is shown as the cyan branch.

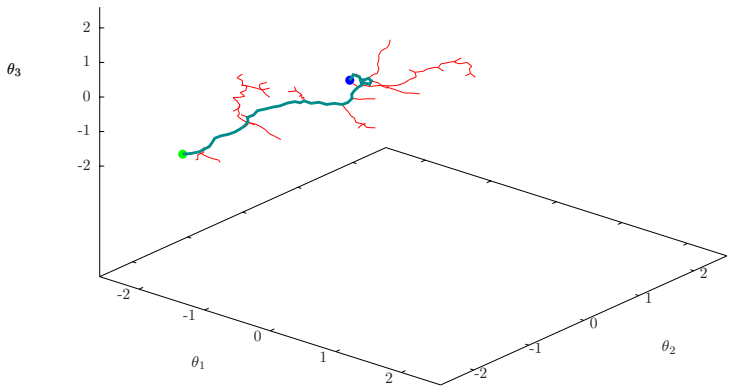



**Figure 7.4:** RRT-tree in a 3 dimensional space with a goal biasing value of 0.05 and the same start end end node as in 7.3

RRTConnect

*RRTConnect* [KL00] is an extension of the the basic RRT-algorithm which rapidly increase the performance when finding a path from a start state $s_{start}$ to a goal state $s_{goal}$. *RRTConnect* uses greedy connect heuristics combined with two trees, starting from $s_{start}$ and $s_{goal}$. In each iteration step, an attempt is made to connect the first tree to the nearest vertex of the other tree. Then the trees are swapped and the second tree attempts to connect to the nearest neighbour of the first tree. When a connection between both trees is found, the algorithm returns the path that has been found.

RRT*

Even though *RRT* is guaranteed to find a path between two states $q_{start}$ and $q_{goal}$ if one exists, it tends to find non-optimal paths (paths which can be much longer than the shortest path). A more recent variation of RRT is called RRT* [KF11] which deals with that issue. Let $T = (V, E)$ be a tree. *RRT** uses a cost function $Cost(q)$ defined as

$$Cost(q) = c,\ q \in V \tag{7.1}$$

with $c$ being the accumulated cost of the path from $q_{start}$ to $q$ and $Cost(q_{start}) = 0$.
The *RANDOM_SAMPLE* function is similar to the one used in the standard *RRT* algorithm, however *RRT** differs from *RRT* in the way the *EXTEND* function is implemented. Given a sample $q_{rand}$ generated by *RANDOM_SAMPLE*, *RRT** first extends the tree towards $q_{rand}$ using the *NEW_NODE* function which returns $q_{new}$. After that, a *NEAR* function is used to find as st of nodes $N$ in the tree for which the distance to $q_{new}$ is smaller than a threshold $\sigma$:

$$NEAR(q_{new}) = \{q \in V :\ d(q, q_{new}) < \sigma\} \tag{7.2}$$

with $d$ being a distance function.

In that set of near nodes a node $q_{min} \in N$ is found such that

$$q_{min} = \operatorname*{argmin}_{q \in N} c(q, q_{new}) \tag{7.3}$$

with

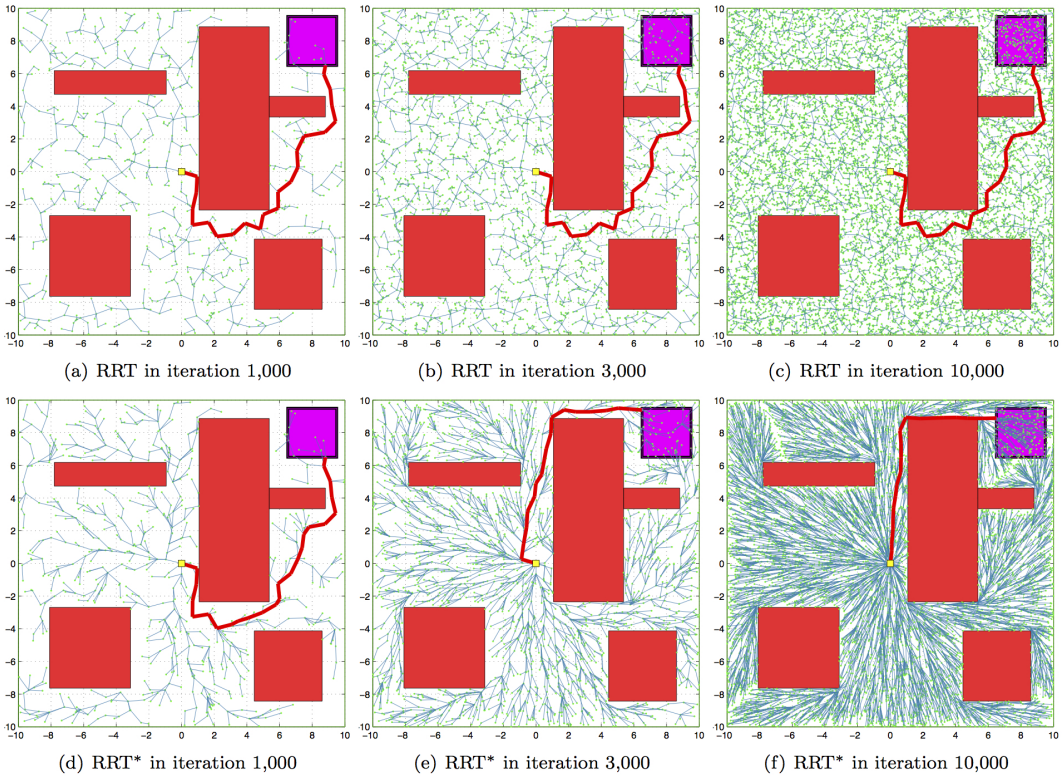
$$c(q_i, q_j) = Cost(q_i) + Cost(E(q_i, q_j)) \tag{7.4}$$

and an edge $E(q_{min}, q_{new})$ from $q_{min}$ to $q_{new}$ is added to the tree.
Then the algorithm performs a r̈ewiringp̈rocedure. For all nodes $q_{near} \in N \setminus q_{min}$ the algorithm checks if these nodes can be reached through $q_{new}$ with a lower cost, such that:

$$Cost(q_{new}) + c(E(q_{new}, q_{near})) < Cost(q_{near}) \tag{7.5}$$

if this is the case, the edge from $E(q_{parent}, q_{near})$ ($q_{parent}$ is the parent node of $q_{near}$) is removed and replaced with $E(q_{new}, q_{near})$. This ensures that the accumulated cost $C(q)$ of the path

from $q_{init}$ to $q \in V$ is minimal.
Figure 7.5 shows a comparison between *RRT* and *RRT** after 1000, 3000 and 10000 iterations. It can be seen that RRT* converges against an optimal solution as the number of iterations increases.



(a) RRT in iteration 1,000 (b) RRT in iteration 3,000 (c) RRT in iteration 10,000

(d) RRT* in iteration 1,000 (e) RRT* in iteration 3,000 (f) RRT* in iteration 10,000

**Figure 7.5:** Comparison between RRT and RRT* (source: http://sertac.scripts.mit.edu/web/?p=502, access date: 20.03.2014)

# 8 Stabilisation approach

In this chapter the main contribution of this thesis, the real-time stabilisation approach is described. An overview of the planning domain is given, as well as a detailed description of the core of the approach which is a planning method based on RRT-trees inside a stability manifold.

## 8.1 Overview

In order to keep a legged vehicle stable, it has to meet defined stability constraints at any time during locomotion. As mentioned in 6.6, these stability constraints can be defined in the robot's task-space. A complete description of the task-space stability constraints is given in 6.6.1.
The main objective of the stabilisation approach presented in this thesis is to prevent the robot's state from falling out of these constraints. In other words, the state of the system has to be inside the manifold (called stability manifold) bounded by the task-space constraints mapped into the robot's state space $S$ at any time. Therefore, a stabilisation system which reacts when the state of the system is outside the stability manifold is not suitable for that task (When a system gets unstable, it might not be possible anymore to get back to a stable state). Instead, the stabilisation algorithm proposed in this reacts **before** the state of the system falls outside the stability manifold.
This is being achieved by separating stable from *critical states* (see 6.8) and reacting, as soon as the system gets into a critical state. It utilises random sample based planners to bring the system back into a defined stable area inside the stability manifold. This is achieved by an offline tree generator, generating RRT* trees inside a stability manifold $C_S$. First, $n$ configurations are sampled from a Gaussian distribution with its mean $\mu$ being the robot's initial configuration $c_{init}$. These samples are the roots of the trees. Starting from these roots, the trees are grown into $C_S$.
During run-time, when the system gets into a critical state $s_c$, $s_c$ is connected to the nearest tree and a path is followed back to the root of the tree. A definition of the stability manifold and the stable area is given in 8.2.

## 8.2 Planning domain

The stabilisation approach proposed in this thesis frames the issue of keeping the system inside defined task-space constraints mapped into $S$ as a motion planning problem in the robot's

configuration space $C \in S$. $S$ consists of 3 subspaces $C_S$, $C_R$, and $S_C$. $C_S \in C$ is the stability manifold in which the motion planning problem is solved. $S_C$ is the set of critical states as defined in 6.8. $C_R$ is a set of configurations considered to be very stable. Here, $C_R$ is sampled by a Gauss distribution with it's mean being $c_{init}$ (the robot's initial configuration) and the variance $\sigma^2 = 0.01$. $C_R$ is the area which is sampled to gain the roots for the stability trees (see 8.3).

## 8.3 Offline generation of RRT* trees

The proposed stabilisation approach consists of an offline tree generator which grows RRT*-trees inside $C_S$, starting from samples around the robot's initial configuration. RRT* is chosen over the standard RRT-trees because of its optimality properties (needs more explanations). First, the algorithm generates $n$ samples $r_i \in C_R$ by sampling a normal distribution with its mean $\mu$ being the robot's initial configuration $c_{init}$. The variance $\sigma$ of this distribution is chosen such that the samples $r_i$ are close (regarding the euclidean distance metric in $C$) to $c_{init}$. Note that $c_{init}$ varies, depending on the inclination angle $\beta$ of the surface the robot walks on (see figure 8.1). Therefore a different $C_R$ has to be defined for each angle of the surface. To reduce the number of $C_R$'s, the inclination angle is discretised (otherwise the number of $C_R$'s would be infinite).

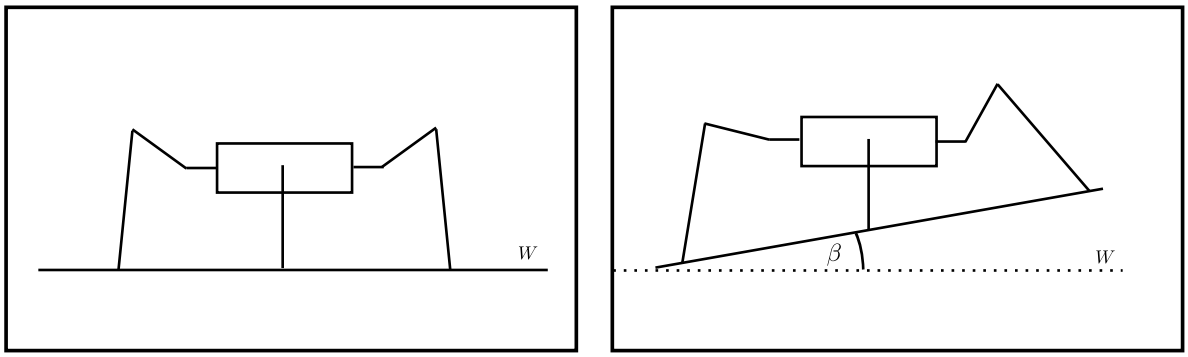



**Figure 8.1:** $c_{inint}$ of a hexapod on different inclined surfaces (side-view)

The samples $r_i$ from the different $C_R$'s serve as the roots of the trees.
Next, for each $r_i$ a goal configuration $c_i$ is sampled with

(8.1) $c_i \in C_C$

This ensures, that the trees are leaving $C_R$ and are grown towards the critical area $C_C$.

Starting from $r_i$ the algorithm grows an RRT*-tree into $C_S$ until a path to $c_i$ has been found. The RRT*-algorithm uses a sampler which uniformly samples $C$ and checks whether newly

generated samples $c_n$ are inside $C_S$. This is done by mapping $c_n$ into $T$ and validating if this mapping fulfils the task-space constraints defined in 6.6.1. If this is the case, the algorithm grows the tree towards that sample. If not, the sample is rejected and a new sample is generated.
After generating a tree for each $r_i$, the trees are saved and loaded during run-time of the system.

## 8.4 Real-time stabilisation

During the locomotion, the current state of the hexapod (the servo positions and the body orientation) is monitored by the *MotionValidator* component with a rate of 35 Hz (which is the maximum servo update rate for the current system design, see 3.3.4). As mentioned earlier, the main approach to stabilise the system is to prevent it from getting into an unstable state by keeping the system inside the defined task-space constraints. In order to find out if the system can potentially get unstable, the *MotionValidator* checks, if it's current state is a *critical state*.

### 8.4.1 Connecting the current state to the offline generated trees

As mentioned in 8.1, the stabilisation algorithm has to react before the current state $s$ of the system becomes unstable. As soon as the system gets into a critical state $s_c$, as defined in 6.8, the stabilisation algorithm attempts to bring the system back from the current critical, to a stable state (which is a configuration inside $C_s$). This is done by using *RRTConnect* to connect the current state to the nearest tree. First, a nearest neighbour search is performed to find the nearest node in the offline generated stable trees.
A brute-force nearest-neighbour search iterates over each node in each tree and finds a node such that

$$g(s_c) = \{c \mid \forall s \in N : \ d(s_c, c) \leq d(s_c, s)\} \tag{8.2}$$

with $N$ being the unification of the nodes of each tree and $d$ being a distance function. The run-time of this brute-force nearest-neighbour is $\mathcal{O}(|N|)$, which can be, depending on the total number of tree nodes $|N|$, too slow for a real-time system. Therefore, instead of performing a brute-force nearest-neighbour search, the algorithm uses the concept of *locality sensitive hashing* (see 2.8).

After the nearest node (or an approximate nearest node) $s_{near}$ has been found using locality sensitive hashing, the algorithm uses *RRTConnect* to find a path $p_c$ in $C_s$ from the current state $s_c$ to $s_{near}$. The time limit for *RRTConnect* to find a solution is $\delta = 0.5$ seconds. Since *RRTConnect* uses two RRT-trees grown from $s_c$ and $s_{near}$ combined with greedy connection heuristics, in most of the cases *RRTConnect* this solve time limit is sufficient. However, depending on the defined task-space constraints and the resulting shape of the stability manifold, *RRTConnect* can take longer than $\delta$. When *RRTConnect* reaches the time limit $\delta$ to solve the path finding problem, the algorithm terminates and the path $p_c$ is a straight

line from $s_c$ to $s_{near}$, even though this could cause $p_c$ to intersect with $C \setminus C_s$, temporarily bringing the system into an unstable state.
A factor which greatly affects the performance of *RRTConnect* is the coverage of $C_S$ by the generated RRT* trees. If the trees cover a large volume of $C_S$, the average distance for each critical state to its nearest neighbour is smaller compared to a set of trees which covers only a small volume of $C_S$. Therefore, *RRTConnect* has to övercomeä smaller distance to find a path from $s_c$ to $s_{near}$.
However, surprisingly the average time *RRTConnect* takes to find a path form $s_c$ to $s_{near}$ increases as the number of nodes in the RRT*-trees increases (and therefore the coverage of $C_S$ by the trees increases as well) and then starts to decrease again as the number of nodes grows further. Figure 8.2 shows the average time *RRTConnect* takes to find a solution, depending on the number of nodes in the RRT*-trees.

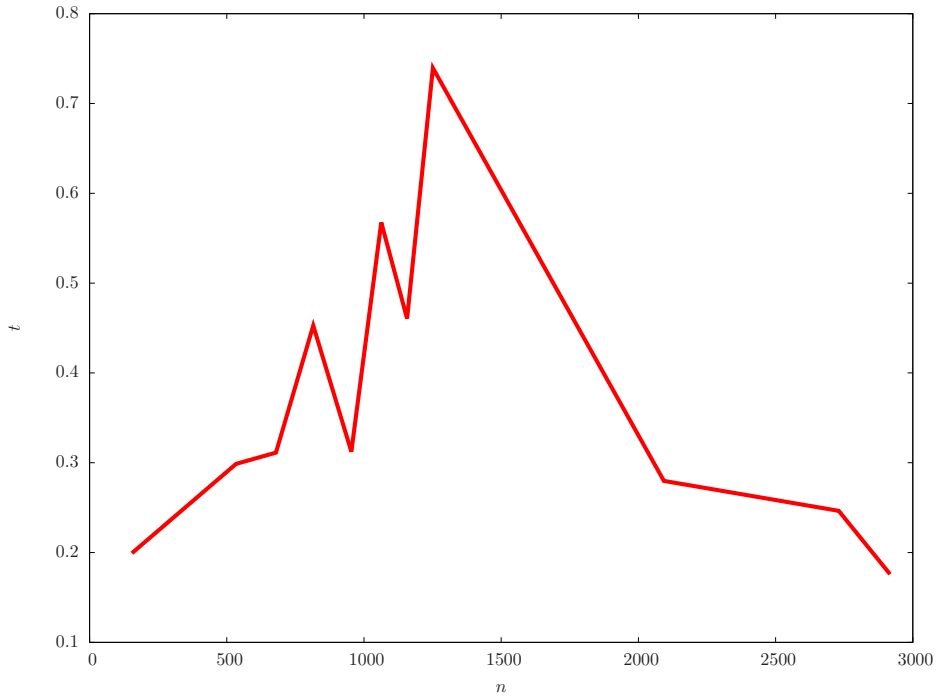



**Figure 8.2:** This plot shows the average time (in seconds) *RRTConnect* takes to find a path from a critical state $s_c$ to it's nearest neighbour $s_{near}$ for 100 critical samples, depending on the number $n$ of nodes in the RRT*-trees.

This result could be explained by the shape of $C_S$. When the number of nodes in the trees is small, for each critical state $s_c$, $s_{near}$ is close to the root of the trees. When the number of nodes is increased, the nearest node $s_{near}$ of a critical state $s_c$ might be inside a region which is, starting from $s_c$, is difficult to reach for *RRTConnect*. However, as the number of nodes is increased, the probability that $s_{near}$ is close enough to be reached from $s_c$ without avoiding regions which are inside $C \setminus C_S$ increases.

The issue of finding the most efficient size of a set of RRT*-trees leads to a more comprehensive analysis of the task-space constraints and their effect on the shape of the constraint manifold $C_S$, which is out of scope for this thesis.
For the experiments in this thesis RRT*-trees are constructed with the combined number of nodes being at least 4000, which turned out to be sufficient for an efficient performance of *RRTConnect*.
After a path $p_c$ from $s_c$ to $s_{near}$ has been found, the final stabilisation path $p_s$ consists of $p_c$ and the path $p_{near,r}$:

$$p_s = p_c \cup p_{near,r} \tag{8.3}$$

where $p_{near,r}$ is the path from $s_{near}$ to the root $r$ of the tree of which $s_{near}$ is a node.
$p_s$ is then executed by the robot such that each joint is consecutively set to the angles defined in the nodes of the path. Figure 8.3 illustrates the stable-path finding process.

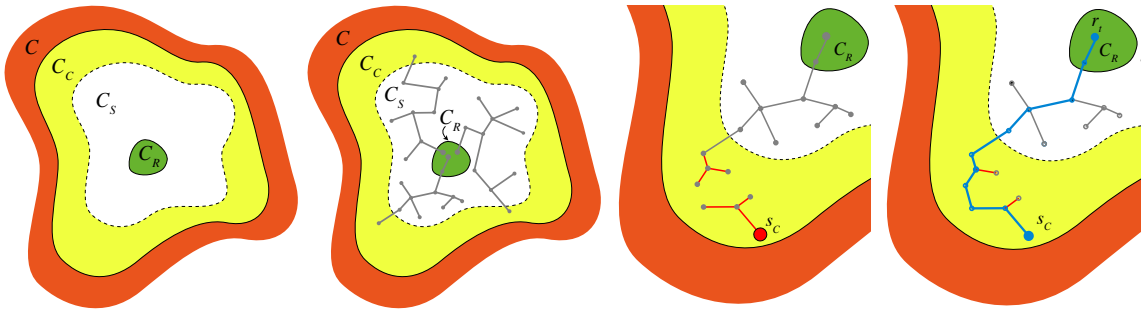



**Figure 8.3:** Overview of the process connecting $s_c$ to the nearest node to find a path back to a stable state.

# 9 Experiments and results

In this chapter the proposed stabilisation approach is evaluated and its effectiveness is shown by conducting several experiments on two real hexapod platforms. First, an overview of the experimental setup is given. The results of the conducted are discussed and analysed.

## 9.1 Experimental setup

The experimental setup consists of two hexapod platforms. The first hexapod platform is a standard commercial PhantomX platform by TrossenRobotics, using AX18 servo actuators. The second platform is a modified PhantomX platform with extended tibia links. This gives the platform a greater flexibility in unknown terrain as is can overcome medium-sized obstacles like rocks. The standard tibia link length is 144 millimetres. The extended tibia link length is 405 millimetres. Figure 9.1 shows the two hexapod platforms.
The platform with extended tibia links is used to validate the stabilisation system's performance in the event of unexpected leg slips, whereas the standard platform is used to validate the stabilisation system when the robot walks on inclined slopes.

(a) (b)

**Figure 9.1:** Modified PhantomX hexapods (a) with additional computing and sensing, (b) with extended tibia segments for reduced stability.

The laptop computer which runs the core part of the developed ROS system is a Intel Core i7 octacore 2.7 Ghz laptop computer with Ubuntu Desktop 12.04 OS, connected to the on-board Pandaboard embedded computer via a WiFi connection. For the outdoor experiments this WiFi connection is replaced by a wired ethernet connection. The complete hardware setup is

described in 3.2.

### 9.1.1 Task-space constraints for the standard hexapod platform

(a) Workspace of the foot tips

The initial tip positions $o_i$ of the standard hexapod platform are

$$\left\{ \begin{pmatrix} 175 \\ 190 \\ -95 \end{pmatrix}, \begin{pmatrix} 175 \\ -190 \\ -95 \end{pmatrix}, \begin{pmatrix} -175 \\ 190 \\ -95 \end{pmatrix}, \begin{pmatrix} -175 \\ -190 \\ -95 \end{pmatrix}, \begin{pmatrix} 0 \\ 230 \\ -95 \end{pmatrix}, \begin{pmatrix} 0 \\ -230 \\ -95 \end{pmatrix} \right\} \tag{9.1}$$

The leg order of these tip-coordinates is front-left, front-right, rear-left, rear-right, middle-left, middle-right.

With these in-ital tip positions, the workspace of each tip is defined as

$$w = \begin{pmatrix} x - 80, & x + 80 \\ y - 40, & y + 40 \\ z - 100, & z + 100 \end{pmatrix} \tag{9.2}$$

with $\begin{pmatrix} x & y & z \end{pmatrix}^T = o_i$

(b) Distance of the body to the support polygon

The constraints for the distance $\left| d_p^z \right|$ of the centre of mass (COM) of the hexapod to the support polygon are defined as

$$-110 \leq d_p^z \leq -50 \tag{9.3}$$

(c) Orientation limits of the body

With $r, p$ being the roll and pitch of the body about its body frame $B$ (expressed as Tait-Bryan-angles), the orientation limits for $r$ and $p$ are

$$-5.0^\circ \leq a \leq 5.0^\circ \tag{9.4}$$

with $a \in \{r, p\}$.

### 9.1.2 Task-space constraints for the hexapod platform with extended tibia links

(a) Workspace of the foot tips

The initial tip positions $o_i$ of the hexapod platform with extended tibia links are

$$\left\{ \begin{pmatrix} 300 \\ 290 \\ -350 \end{pmatrix}, \begin{pmatrix} 300 \\ -290 \\ -350 \end{pmatrix}, \begin{pmatrix} -300 \\ 290 \\ -350 \end{pmatrix}, \begin{pmatrix} -300 \\ -290 \\ -350 \end{pmatrix}, \begin{pmatrix} 0 \\ 380 \\ -350 \end{pmatrix}, \begin{pmatrix} 0 \\ -380 \\ -350 \end{pmatrix} \right\} \tag{9.5}$$

With these initial tip positions, the workspace of each tip is defined as

$$w = \begin{pmatrix} x - 170, & x + 170 \\ y - 160, & y + 160 \\ z - 180, & z + 180 \end{pmatrix} \tag{9.6}$$

with $\begin{pmatrix} x & y & z \end{pmatrix}^T = o_i$

(b) Distance of the body to the support polygon

The constraints for the distance $\left| d_p^z \right|$ of the centre of mass (COM) of the hexapod to the support polygon are defined as

$$-410 \leq d_p^z \leq -300 \tag{9.7}$$

(c) Orientation limits of the body

The constraints for the body roll and pitch $r, p$ are the same as for the standard hexapod platform.

## 9.2 Experiments

In the first series of experiments, the standard hexapod platform is performing a locomotion using a tripod gait, walking outdoors on even terrain towards an inclined slope. The task constraint in this experiment is to maintain an even body orientation. Therefore, the stabilisation system has to automatically adapt the body's pitch angle relative to the ground, when the ground inclination changes.

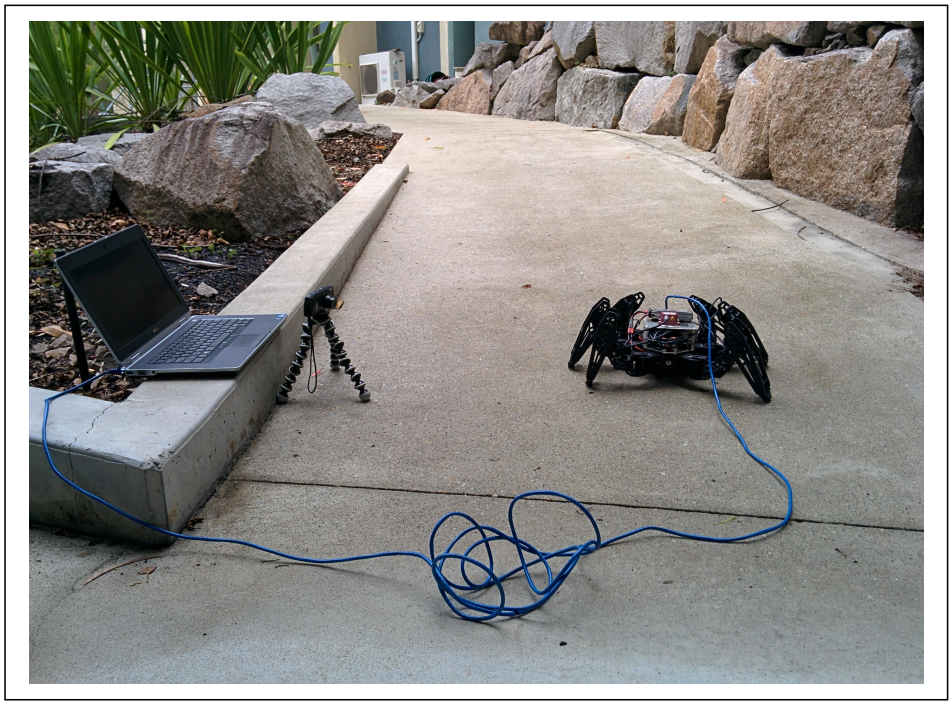


**Figure 9.2:** Outdoor setup

In the second series of experiments, the platform with extended tibia links performs a tripod gait on even indoor terrain. During its locomotion one leg is stepping on a slippery surface - in this case a small metal plate. Due to a low friction coefficient between the metal plate and the terrain, stepping on the metal plate caused it to slip away, along with the leg which stepped on the plate, causing instabilities of the whole system. The goal of that experiment is to show that the stabilisation system is able to keep the legs into their workspace as well as stabilising the body movements which, if not stabilised, can lead to catastrophic events like toppling over of the platform.
The first set of runs were performed without the stabilisation system. Starting with the front-left leg, each leg stepped three times on the slippery metal plate. During the locomotion, the body orientation was measured and recorded. Since the system had no information about the leg slip, it tried to keep performing it's task (walking straight with a tripod gait).
The next set of runs was performed with the stabilisation system turned on, where the stabilisation system actively reacted to the leg slipping event, bringing the system back to a stable state and eventually, after the recovery phase, tried to continue to perform the given task.

## 9.3 Results

In the first experiment, when standard hexapod platform was performing a tripod-gait walking on an inclined surface, the body's pitch angle was automatically regulated by the stabilisation system. Without the stabilisation system, the hexapod was not able to keep an even body orientation. Even though the hexapod was statically stable when not using the stabilisation system, the defined task-space constraints were violated, leading to an inclined pitch angle of the body, parallel to the ground.
Figure **??** shows a time-slice where the hexapod walks towards an inclined surface (picture 1-2), changing it's body orientation according the task-space constraints (picture 3-4) and continues it's gait with the regulated body orientation.

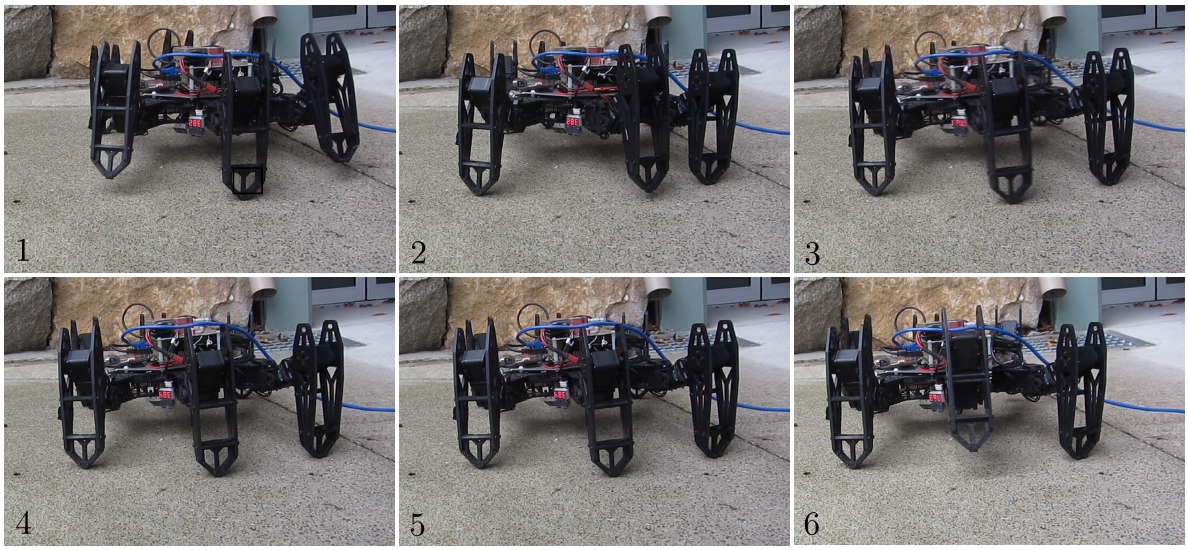



**Figure 9.3:** Inclination experiment

During the leg-slipping experiment, it became clear that even when the hexapod robot maintains static stability when performing an alternating tripod gait, disturbances in foot-ground interactions like leg slippages lead to instabilities of the platform. This impairs the ability to perform locomotion or in worst case, the robot topples over and potentially suffers structural damages due to uncontrolled forces on the joints.
During the whole set of experiments with 3 slip events per leg, 3 servo motors were damaged and had to be replaced. It could be noticed that leg slippages tent to destabilise the system in a way that it eventually topples over when no stabilisation system is used. Table 9.1 shows a summary of damaged servos and loss of static stability during the leg slipping experiments.

| | *broken servos* | *loss of static stability* |
|---|---|---|
| With stabilisation | 0 | 0 |
| Without stabilisation | 3 | 6 |

**Table 9.1:** This table shows the the number of broken servos and the amount of times the hexapod lost static stability during the leg-slipping experiments.

During the experiments it became clear that especially slipping events of the middle legs caused the robot to lose static stability. When the stabilisation system was turned off, 4 out of 6 leg-slipping events for the middle legs caused the robot to topple over. The other two stability losses were induced by slipping events of the front-right leg. It could be seen that a leg-slip didn't immediately cause the robot to topple-over. However, since the system has no information about the leg-slip, it continued to perform the tripod gait, eventually causing the robot to topple over after 2 to 3 gait steps.
By utilising the stabilisation system, the robot was able to react to these leg-slipping events and eventually get back to a stable state in all of the test runs. It could be seen, that the stabilisation system significantly improved the overall static stability of the hexapod after an unexpected external disturbance which caused the hexapod to get into a critical state. As a result of this, the hexapod was able to continue its task after it recovered from these critical states.

Without the stabilisation system the servos were exposed to forces imparted by uncontrolled body movements, causing damage to 3 servos.
As it can be seen in table 9.1, the stabilisation system successfully prevented these structural damages. By reacting quickly to unexpected events, and keeping the system inside the stability manifold.
Figure 9.4 shows one leg-slip event. The front-left leg steps on a small metal plate, causing it to slip on the surface (picture 1-5). The stabilisation system reacts to that by bringing the system back to a stable state (picture 6-8) and eventually continues to perform it's task (picture 9-10).

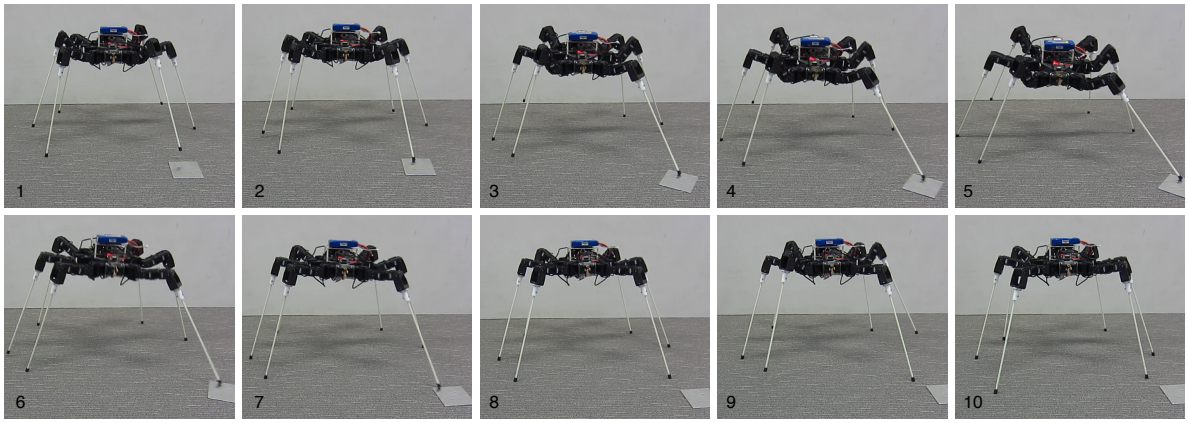



**Figure 9.4:** Leg-slip experiment

In order to gain measurable results while validating the stabilisation system, the orientation information provided by the IMU was measured during the leg-slippages. As seen in figure 9.5, a leg-slip caused unpredictable changes in the orientation of the body (red line). As mentioned earlier, these chaotic behaviours imparted strong forces impacting on the servos, causing three of them to suffer structural damages. Another result of these uncontrolled orientation changes is a loss of static stability eventually causing the hexapod to topple-over.
It can also be seen, that the stabilisation system is able to keep the body within stable orientation constraints (green line).

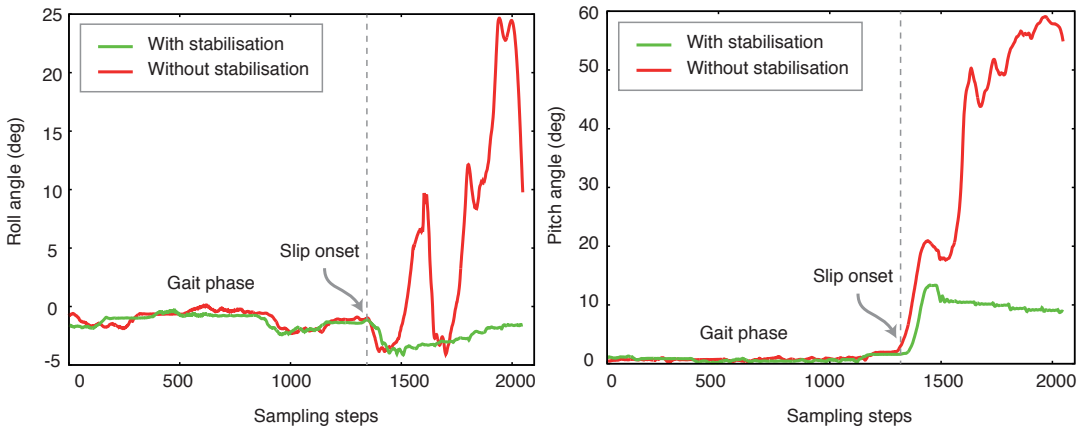



**Figure 9.5:** This graph shows the body roll (a) and pitch (b) during a leg-slipping event with and without stabilisation system.

During the experiments it could be seen that with the proposed stabilisation system, the hexapods were able to keep their body orientation within the defined angular limits. This is most notably when looking at a series of experiments and calculate the deviation of the pitch and roll angles of the body about the zero angle.
Let $\mu_r = 0$ and $\mu_p = 0$ be the expected value for the roll and pitch angle of the body. During the leg-slip and the inclination experiments, the orientation of the body was recorded and the variance from $\mu_r$ and $\mu_p$ was calculated.

Table 9.2 shows the variance of the pitch and roll angle of the body ($\sigma_{roll}$ and $\sigma_{pitch}$) with and without the stabilisation.

| | $\sigma_{roll}$ | $\sigma_{ritch}$ |
|---|---|---|
| With stabilisation | 1.10 deg | 5.78 deg |
| Without stabilisation | 8.91 deg | 21.60 deg |

**Table 9.2:** This table shows the standard deviation of the body's pitch and roll angles from a series a leg-slipping experiments.

It can be seen that the variance of the orientation angles is significantly smaller when the proposed stabilisation system is used. The system is able to keep the body inside the defined orientation constraints, both in the inclination and the leg-slipping experiments, avoiding dangerous orientation angles which occur when the the platform is getting unstable and is about to topple-over.

# 10 Conclusion and future work

In this thesis a hierarchically control software architecture for hexapod robots, which allows to control and monitor the vehicles, has been developed. In order to achieve a high degree of modularisation combined with the possibility to easily extend the system, the hexapod control architecture was implemented in ROS. This implementation has been successfully tested on two real hexapod platforms, the standard PhantomX hexapod robot and a modified platform with extended tibia links.
The main contribution of this thesis however, is the development of a real-time stabilisation system based on state-of-the-art planning algorithms such as RRT, RRT* and RRTConnect. With the proposed stabilisation system, it is possible to define task-space constraints, which are utilised as stability constraints, and keeping the hexapod platforms inside those defined constraints in real-time. The proposed approach requires neither a dynamic model of the hexapod robots, nor information about the local terrain the hexapod performs its task in. However, the implementation of the proposed stabilisation approach is flexible enough to include dynamic models or additional sensor data, like terrain information coming from a stereo-camera.

The proposed stabilisation system has been shown to perform well under unexpected disturbances like leg-slipping or a change in the inclination angle of the terrain.

## Future work

The proposed stabilisation system provides a fast and robust method to stabilise hexapod platforms in real time by keeping the system inside user-defined task-space constraints. Even though doesn't require local terrain information, the system is capable of including terrain information and utilising such additional information to improve the performance of the approach. This could be addressed in future work. As mentioned in 8.4.1, the properties of the stability manifold influence the performance of *RRTConnect*. Analysing the shape of the stability manifold and utilising this information for a more efficient tree-generation algorithm would be another field of work.
The stabilisation system described in this thesis strictly uses a deterministic model of the hexapod (even though real platforms are never deterministic). Modelling the hexapod in a probabilistic fashion leads to a field which is still subject of ongoing research. Another drawback of the models used in this thesis is the fact that they are pure static models. Taking the dynamics of the platform into account could improve the performance of the proposed stabilisation system significantly.

# Bibliography


[Bad90] D. Badouel. An Efficient Ray-Polygon Intersection. In A. S. Glassner, editor, *Graphics Gems*, pp. 390–393. Academic Press, 1990. (Cited on page 38)

[Bel11] P. Belter, Dominik; Skrzypczynski. Integrated Motion Planning for a Hexapod Robot Walking on Rough Terrain. In *Proceedings of the IFAC World Congress*, volume 18, pp. 6918–6923. 2011. (Cited on page 9)

[BRL03] T. Bretl, S. M. Rock, J.-C. Latombe. Motion planning for a three-limbed climbing robot in vertical natural terrain. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 2946–2953. IEEE, 2003. (Cited on page 9)

[BS11] D. Belter, P. Skrzypczyński. Rough terrain mapping and classification for foothold selection in a walking robot. *Journal of Field Robotics*, 28(4):497–528, 2011. (Cited on page 9)

[BS12] D. Belter, P. Skrzypczynski. Posture optimization strategy for a statically stable robot traversing rough terrain. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2012*, pp. 2204–2209. IEEE, 2012. (Cited on page 9)

[BSK11] D. Berenson, S. S. Srinivasa, J. J. Kuffner. Task Space Regions: A framework for pose-constrained manipulation planning. *I. J. Robotic Res.*, 30(12):1435–1460, 2011. (Cited on page 42)

[HBL+08] K. K. Hauser, T. Bretl, J.-C. Latombe, K. Harada, B. Wilcox. Motion Planning for Legged Robots on Varied Terrain. *International Journal of Robotics Research*, 27(11-12):1325–1349, 2008. (Cited on page 9)

[IM98] P. Indyk, R. Motwani. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pp. 604–613. ACM, New York, NY, USA, 1998. doi:10.1145/276698.276876. URL http://doi.acm.org/10.1145/276698.276876. (Cited on page 17)

[KF11] S. Karaman, E. Frazzoli. Sampling-based Algorithms for Optimal Motion Planning. *International Journal of Robotics Research.*, 30(7):846–894, 2011. doi:10.1177/0278364911406761. URL http://dx.doi.org/10.1177/0278364911406761. (Cited on page 51)

[KL00] J. J. Kuffner Jr., S. M. Lavalle. RRT-Connect: An efficient approach to single-query path planning. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 995–1001. 2000. (Cited on page 51)

[Lav98] S. M. Lavalle. Rapidly-exploring random trees: A new tool for path planning. *Technical Report*, 1998. (Cited on page 47)

[LLC88] T.-T. Lee, C.-M. Liao, T. Chen. On the stability properties of hexapod tripod gait. *IEEE Journal of Robotics and Automation*, 4(4):427–434, 1988. (Cited on page 9)

[LS01] B.-S. Lin, S.-M. Song. Dynamic modeling, stability, and energy efficiency of a quadrupedal walking machine. *J. Field Robotics*, 18(11):657–670, 2001. (Cited on page 38)

[MI79] R. B. McGhee, G. I. Iswandhi. Adaptive locomotion of a multilegged robot over rough terrain. *IEEE Transactions on Systems, Man and Cybernetics*, 9(4):633–643, 1979. (Cited on pages 37 and 43)

[MT97] T. Möller, B. Trumbore. Fast, Minimum Storage Ray-Triangle Intersection. *J. Graphics, GPU and Game Tools*, 2(1):21–28, 1997. (Cited on page 38)

[O'R98] J. O'Rourke. *Computational Geometry in C (2Nd Ed.)*. Cambridge University Press, New York, NY, USA, 1998. (Cited on page 38)

[TT90] A. Takanishi, T. Takeya. Dynamic modeling, stability, and energy efficiency of a quadrupedal walking machine. *IROS*, 2:795–801, 1990. (Cited on page 38)

All links were last followed on March 21, 2014.

**Decleration**

I declar that this thesis is the solely effort of the author. I did not use any other sources and references than the listed ones. I have marked all contained direct or indirect statements from other sources as such. Neither this work nor significant parts of it were part of another review process. I did not publish this work partially or completely yet. The electronic copy is consistent with all submitted copies.

---

place, date, signature