Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Motion Planning for a Six-Legged Robot

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Bernhard Wimmer
Matrikelnummer 0928776

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Asst.-Prof. Dr. Ezio Bartocci

Wien, 28. April 2016

Bernhard Wimmer   Ezio Bartocci

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Motion Planning for a Six-Legged Robot

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Bernhard Wimmer
Registration Number 0928776

to the Faculty of Informatics
at the TU Wien
Advisor: Asst.-Prof. Dr. Ezio Bartocci

Vienna, 28th April, 2016

Bernhard Wimmer
Ezio Bartocci

Erklärung zur Verfassung der Arbeit

Bernhard Wimmer
Aßmayergasse 23/8

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 28. April 2016

Bernhard Wimmer

Acknowledgements

I would like to express my gratitude toward my advisor, Ezio Bartocci, for giving me the opportunity to work on a long-time passion of mine and for encouraging me to be ambitious.

Furthermore I would like to thank my friends for always providing welcome diversions and my sister in particular for putting up with me during frustrating times.

Finally, I wish to thank my parents for their mental as well as financial support during my time as a student and throughout my life.

Abstract

Motion planning refers to the process of translating high-level specifications of tasks into low-level sequences of control inputs for a robot's actuators.

Legged robots, although more flexible with respect to wheeled robots in uneven and cluttered environments, are a very challenging application domain for motion planning. Such systems may benefit from the use of a multi-modal planner that is able to switch between discrete modes corresponding to the set of contact points of the robot with the ground. Widely available single-mode planners can then used to find a continuous trajectory through a given mode. Kinematic constraints usually require such paths to be constrained to a submanifold of the configuration space, which limits the efficacy of common single-mode planners and requires them to be adapted for this environment.

In this thesis I provide an introduction to modeling and simulating robotic systems under the influence of dynamics, evaluate a multi-modal planner using different single-mode planners for controlling a six-legged robot and develop a prototype hardware platform for future development.

Two different scenarios are considered for the planner's evaluation: walking on a flat surface and climbing a step. For each scenario several metrics to compare the planners' performance are collected, including the execution time of each algorithm, the required number of single-mode planner instances and the error in the final positions of the footfalls after the generated motions are executed by a closed-loop controller in a simulated environment.

I believe this comparison is useful in helping others make informed decisions about which of the common single-mode planners are effective in this context. Furthermore, I hope to provide some insights on the changes necessary to adapt other planners for this environment and future challenges that need to be solved for autonomous motion planning.

Contents

Abstract **ix**

Contents **xi**

1 Introduction **1**

2 Mathematical/Physical Background **3**
2.1 Differential Equations 3
2.2 Newtonian Particles 8
2.3 Rigid bodies 9
2.4 Constraints 19
2.5 Articulated Rigid Body Dynamics 25

3 Direct Control **31**
3.1 Constraints 31

4 Motion Planning **37**
4.1 Motion Planning 37

5 Implementation Details **53**
5.1 Inverse Kinematics (IK) 53
5.2 Collision Detection and Response 56

6 Robot Platform **61**

7 Results **63**
7.1 Flat Surface 63
7.2 Step 66

8 Conclusion and Future Work **69**
8.1 Future Work 69

A IROS 2017 Submission **71**

List of Figures **79**

List of Tables 80

List of Algorithms 81

Bibliography 83

CHAPTER 1

Introduction

Legged robot locomotion is a topic that has been fascinating researchers for a long time. The fact that legs are a common trait in many land animals indicates their advantage in unknown and varying environments. Legs allow biological creatures to navigate in cluttered terrain by stepping over or on top of obstacles. Consequently robots with legs have been proposed for applications in mining, building inspection, fire-fighting and planetary-exploration among others [MI79] [BBS94].

Autonomous behavior is an integral part of increasing the usefulness of such robots. Hand-made motion sequences can be useful in many cases [PMS07] [RBN$^+$08] but for true autonomy these motions need to be generated automatically. Despite the long history, motion planning for legged robots is still a challenging problem. The high number of actuators and changing dimensionality of the problem require powerful computers and sophisticated algorithms to perform reliable planning in acceptable time frames.

The configuration space of legged robots is unlike that of many, more conventional robots. The robot can have a varying number of contact points with the ground (footfalls). These contacts introduce constraints that limit the motion space to a sub manifold of the original space. Additional footfalls decrease the dimension of the motion space by adding more constraints. Removing a footfall (e.g. by raising a leg) removes constraints and increases the dimension of the valid motion space. Planners for such systems need to be able to traverse a discrete set of modes and plan motions in the continuous space connecting those modes. This problem is referred to as multi-modal planning and has been investigated in the context of grasping and re-grasping operations [ALS94] [FB97] [NK00] [SCS02] [HNTHGB07], walking [CKNK03] [Hau08] as well as re-configuring robots [CY99].

1.0.1 Overview

The following listing gives a short overview of the contents of each chapter.

- *Mathematical Background*

  This chapter serves as an introduction to the basics of simulating physical systems and modeling robots as trees of links and joints.

- *Direct Control*

  This chapter builds on the previous one by creating a closed-loop controller that is able to return joint torques (or other controlling variables) from a specification that provides a combination of target positions(/velocities/accelerations) for any links or joints in the system.

- *Motion Planning*

  In this chapter the general approach of the multi-modal planner is outlined along with a selection of commonly used single-mode planners.

- *Implementation and Platform*

  Those two chapters describe and justify choices made during the implementation of the previously described planner. The latter focuses on the physical implementation and a description of the employed hardware.

- *Results*

  This chapter aims to compare and discuss the results of the implemented motion planner. Particular focus lies on the performance of different single-mode planners in the context of the overall approach.

CHAPTER 2

Mathematical/Physical Background

2.1 Differential Equations

Differential equations are used to describe the relationship between some unknown function and its derivatives. Solving a differential equations equates to finding a function that satisfies this relation. Often this needs to be done while satisfying some additional constraints. One class of such problems are the so-called *initial value problems.*

2.1.1 Initial Value Problems

A canonical initial value problem is an ordinary differential equation (ODE) of the form

$$\dot{\mathbf{x}} = f(\mathbf{x}, t) \tag{2.1}$$

where $\dot{\mathbf{x}}$ denotes the derivative of $\mathbf{x}$ with respect to the time t.

As the name indicates the goal is to find an expression for $\mathbf{x}$ such that $\mathbf{x}(t_0) = \mathbf{x_0}$. Many physical phenomena can be formulated as initial value problems and consequently finding solutions to them is an important part of designing and building reliable systems.

2.1.2 Analytical Solutions

Analytical solutions are a way to describe the function $\mathbf{x}(t)$ in an explicit manner, i.e. the solution can be written in the form of an integral. It is important to note that not all ODEs have such a solution. The most common first order ODEs with analytical solutions are:

$$\text{Directly integrable: } \dot{\mathbf{x}} = f(\mathbf{x}(t), t) = g(t) \tag{2.2}$$

$$\text{Linear: } \dot{\mathbf{x}} = f(\mathbf{x}(t), t) = g(t)\mathbf{x}(t) + h(t) \tag{2.3}$$

$$\text{Separable: } \dot{\mathbf{x}} = f(\mathbf{x}(t), t) = g(t)h(\mathbf{x}(t)) \tag{2.4}$$

$$\text{Homogeneous: } \dot{\mathbf{x}} = f(\mathbf{x}(t), t) = g(\mathbf{x}(t)/t) \tag{2.5}$$

$$\begin{aligned} \text{Exact: } f(\mathbf{x}, t)dt + g(\mathbf{x}, t)dx = 0 \\ \text{where the partial derivatives satisfy } f_{\mathbf{x}} = g_t \end{aligned} \tag{2.6}$$

2.1.3 Numerical Solutions

Numerical solutions on the other hand evaluate the function f iteratively to obtain the solution. For each step i an approximate change for $\mathbf{x}$ is calculated as $\boldsymbol{\Delta}\mathbf{x} = f(\mathbf{x}_i, \Delta t)$. The value of $\mathbf{x}_{i+1}$ is then given by $\mathbf{x}_i + \boldsymbol{\Delta}\mathbf{x}$. In this scenario the function f can be seen as a black box - it is given some values for $\mathbf{x}$ and t and returns a numerical value for $\dot{\mathbf{x}}$.

Euler's method

The simplest numerical method is Euler's method. With an initial value $\mathbf{x_0} = \mathbf{x}(t_0)$ the goal is to find a value for $\mathbf{x}(t_0 + h)$. Euler's method simply calculates this value by taking a step in the direction of the derivative:

$$\mathbf{x}(t_0 + h) = \mathbf{x_0} + h\dot{\mathbf{x}}(t_0) \tag{2.7}$$

This method is very simple but also inaccurate. Fig. 2.1 shows what happens to $\mathbf{x}$ when the integral curves of f are concentric circles. The correct solution has $\mathbf{x}$ orbit the origin along a circle with the same radius. Due to the error of the Euler integration this circle turns into a polygonal spiral. Smaller step sizes can slow the rate at which $\mathbf{x}$ distances itself from the correct solution, but the error can not be eliminated.

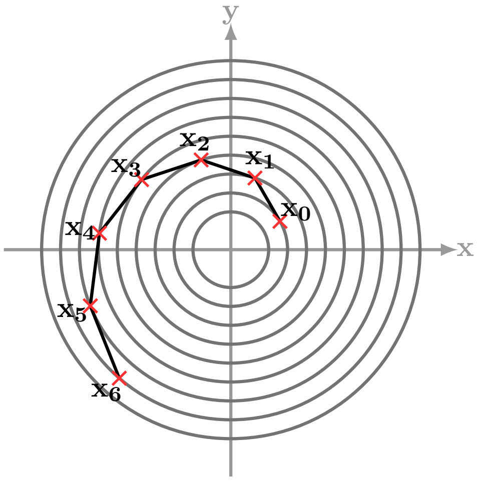


Figure 2.1: Example of error due to Euler integration ($x^2 + y^2 = r^2$, $\dot{\mathbf{x}} = [-y, x]^T$)

In some cases Euler's method can also be completely unstable. Consider $\dot{\mathbf{x}} = -k\mathbf{x}$ whose solution has the form $\mathbf{x}(t) = e^{-kt}$: This function is clearly supposed to decay toward zero. For small enough step sizes we get the desired behavior. If $h > \frac{1}{k}$ then $|\mathbf{\Delta x}| > |\mathbf{x}|$ and the result of the integration oscillates about 0. For larger values the oscillation diverges and the results are useless.

To get an idea on how to improve this method it is useful to look at what this method actually is. For this reason let's compare it to the Taylor series. It is defined for infinitely differentiable (i.e. smooth) functions and can be written as:

$$\mathbf{x}(t_0 + h) = \sum_{n=0}^{\infty} \frac{h^n}{n!} \mathbf{x}^n(t_0)$$

$$or$$

$$\mathbf{x}(t_0 + h) = \mathbf{x}(t_0) + h\dot{\mathbf{x}}(t_0) + \frac{h^2}{2!}\ddot{\mathbf{x}}(t_0) + \frac{h^3}{3!}\dddot{\mathbf{x}}(t_0) + \cdots$$

The difference between this and Eq. 2.7 are the higher order derivatives. This means that the Euler integration is only correct if the factors for the second, third, etc. derivatives are all zero, i.e. if $\mathbf{x}$ is a linear function. Consequently the error consists of the remaining terms. The dominating error term is $\frac{h^2}{2!}\ddot{\mathbf{x}}(t_0)$ as the contributions of the higher order derivatives decrease. The Euler integration's local truncation error is thus often written as $\mathcal{O}(h^2)$. The local truncation error is the error incurred after each iteration. An easy way to illustrate this, is to look at what happens when the step size (h) is changed. If h is halved then the resulting error is around a quarter of the previous one. However, it also means that twice as many steps need to be calculated to arrive at the original stepsize's value. Another consequence of this is that, in theory, the accuracy can be improved as desired by reducing the step size at the cost of computation time.

Midpoint method

One way to improve on the results of the Euler integration is to add an additional term of the Taylor series. Adding $\frac{h^2}{2!}\ddot{\mathbf{x}}(t_0)$ to Eq. 2.7 gives

$$\mathbf{x}(t_0 + h) = \mathbf{x}(t_0) + h\dot{\mathbf{x}}(t_0) + \frac{h^2}{2!}\ddot{\mathbf{x}}(t_0) \tag{2.8}$$

If one additionally assumes that $\dot{\mathbf{x}}$ does not depend on t directly then it can be rewritten as

$$\dot{\mathbf{x}} = f(\mathbf{x}(t)) \tag{2.9}$$

using the chain-rule to calculate $\ddot{\mathbf{x}}$ gives

$$\ddot{\mathbf{x}} = \frac{\partial f}{\partial \mathbf{x}}\dot{\mathbf{x}} = f'f \tag{2.10}$$

Calculating f' is often expensive and can be approximated by performing another Taylor expansion. This time on f:

$$f(\mathbf{x_0} + \mathbf{\Delta x}) = f(\mathbf{x_0}) + \mathbf{\Delta x} f'(\mathbf{x_0}) + \mathcal{O}(\mathbf{\Delta x^2}) \tag{2.11}$$

Choosing $\mathbf{\Delta x} = \frac{h}{2} f(\mathbf{x_0})$ introduces $\ddot{\mathbf{x}}$ to this expression:

$$f(\mathbf{x_0}) + \frac{h}{2} f(\mathbf{x_0}) = f(\mathbf{x_0}) + \frac{h}{2} f(\mathbf{x_0}) f'(\mathbf{x_0}) + \mathcal{O}(h^2) = f(\mathbf{x_0}) + \frac{h}{2}\ddot{\mathbf{x}}(t_0) + \mathcal{O}(h^2) \tag{2.12}$$

multiplying by h reduces the error term from $\mathcal{O}(h^2)$ to $\mathcal{O}(h^3)$ and gives

$$\frac{h^2}{2}\ddot{\mathbf{x}} + \mathcal{O}(h^3) = h f(\mathbf{x_0} + \frac{h}{2} f(\mathbf{x_0})) - f(\mathbf{x_0}) \tag{2.13}$$

Finally the right hand side can be substituted into Eq. 2.8

$$\mathbf{x}(t_0 + h) = \mathbf{x}(t_0) + h f(\mathbf{x_0} + \frac{h}{2} f(\mathbf{x_0})) \tag{2.14}$$

Essentially this evaluates $f(\mathbf{x_0})$ to calculate some point between x_0 and the next value to evaluate the final result for $\mathbf{x}(t_0 + h)$ (hence the name midpoint method).

Taking this approach even further to reduce the error is also possible. One popular version of this method is called Runge-Kutta of order 4 (RK4) with a local truncation error of $\mathcal{O}(h^5)$. The derivation is similar to the one above (which could also be called Runge-Kutta of order 2) and is not done here. The calculation is performed in the following way:

$$\begin{aligned}
k_1 &= h f(\mathbf{x_0}, t_0) \\
k_2 &= h f(\mathbf{x_0} + \frac{k_1}{2}, t_0 + \frac{h}{2}) \\
k_3 &= h f(\mathbf{x_0} + \frac{k_2}{2}, t_0 + \frac{h}{2}) \\
k_4 &= h f(\mathbf{x_0} + k_3, t_0 + h) \\
\mathbf{x}(t_0 + h) &= \mathbf{x_0} + \frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4
\end{aligned}$$

Adaptive stepsizes

Picking an appropriate stepsize is one issue that the above methods have in common. On the one hand a bigger step size means that the result can be calculated more quickly as fewer steps need to be taken and on the other this also equates to less precision. For this reason adaptive stepsizes can be introduced. Their goal is to maximize the step size while retaining the desired precision of the calculations.

A way to calculate such an adaptive stepsize is step doubling - We calculate two different results for $\mathbf{x}(t_0 + h)$. The first one ($\mathbf{x_a}$) by taking one step of size h and the second one($\mathbf{x_b}$) by taking two steps of size $\frac{h}{2}$. For the Euler method both of these results have an error of $\mathcal{O}(h^2)$. They also differ from each other by $\mathcal{O}(h^2)$. This gives a convenient way to estimate the current error:

$$e = |\mathbf{x_a} - \mathbf{x_b}| \tag{2.15}$$

The new stepsize can then be calculated with the allowed tolerance for the error (tol) as

$$h_{adaptive} = \left(\frac{tol}{e}\right)^{\frac{1}{2}} h \tag{2.16}$$

The same approach can be used for the Runge-Kutta method [PTVF96]. When RK4 is used this approach requires f to be evaluated eleven times - three times for each step while k_1 for the first step with $\frac{h}{2}$ and h are identical. One property of Runge-Kutta formulas is that for orders (M) higher than four more than M evaluations of f are required (but not more than $M + 2$). This is one reason why RK4 is so popular. It requires a small number of evaluations but still gives a reasonably accurate result.

A different method was developed when Fehlberg discovered that the factors of a fifth order method with six function evaluations could be recombined to give a fourth order method as well. The difference between those two methods can then be used to adjust the stepsize accordingly.

A fifth order Runge-Kutta formula has the form:

$$\begin{aligned}
k_1 &= hf(\mathbf{x_0}, t_0) \\
k_2 &= hf(\mathbf{x_0} + b_{21}k_1, t_0 + a_2h) \\
k_3 &= hf(\mathbf{x_0} + b_{31}k_1 + b_{32}k_2, t_0 + a_3h) \\
k_4 &= hf(\mathbf{x_0} + b_{41}k_1 + b_{42}k_2 + b_{43}k_3, t_0 + a_4h) \\
k_5 &= hf(\mathbf{x_0} + b_{51}k_1 + b_{52}k_2 + b_{53}k_3 + b_{54}k_4, t_0 + a_5h) \\
k_6 &= hf(\mathbf{x_0} + b_{61}k_1 + b_{62}k_2 + b_{63}k_3 + b_{64}k_4 + b_{65}k_5, t_0 + a_6h) \\
\mathbf{x}(t_0 + h) &= \mathbf{x_0} + c_1k_1 + c_2k_2 + c_3k_3 + c_4k_4 + c_5k_5 + c_6k_6 + \mathcal{O}(h^6)
\end{aligned}$$

And the embedded fourth order method

$$\mathbf{x}^*(t_0 + h) = \mathbf{x_0} + c_1^*k_1 + c_2^*k_2 + c_3^*k_3 + c_4^*k_4 + c_5^*k_5 + c_6^*k_6 + \mathcal{O}(h^5) \tag{2.17}$$

For the values of the factors the ones found by Cash and Karp [CK90] in table 2.1 are usually used as they have slightly better error properties than Fehlberg's.

Table 2.1: Cash-Karp factors for Runge-Kutta method

i	a_i	b_{ij}					c_i	c_i^*
1							$\frac{37}{378}$	$\frac{2825}{27648}$
2	$\frac{1}{5}$	$\frac{1}{5}$					0	0
3	$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$				$\frac{250}{621}$	$\frac{18575}{48384}$
4	$\frac{3}{5}$	$\frac{3}{10}$	$-\frac{9}{10}$	$\frac{6}{5}$			$\frac{125}{594}$	$\frac{13525}{55296}$
5	1	$-\frac{11}{54}$	$\frac{5}{2}$	$-\frac{70}{27}$	$\frac{35}{27}$		0	$\frac{277}{14336}$
6	$\frac{7}{8}$	$\frac{1631}{55296}$	$\frac{175}{512}$	$\frac{575}{13824}$	$\frac{44275}{110592}$	$\frac{253}{4096}$	$\frac{512}{1771}$	$\frac{1}{4}$
	$j =$	1	2	3	4	5		

2.2 Newtonian Particles

The motion of Newtonian objects is described by $\mathbf{f} = m * \mathbf{a}$ or "force equals mass times acceleration" in words. This can easily be rewritten in the canonical form:

$$\ddot{\mathbf{x}} = \frac{\mathbf{f}}{m} \tag{2.18}$$

To apply the above methods, which operate on first order equations, we split this expression into two coupled first order equations by introducing a variable v for the velocity.

$$\dot{\mathbf{v}} = \frac{\mathbf{f}}{m} \tag{2.19}$$

$$\dot{\mathbf{x}} = \mathbf{v} \tag{2.20}$$

This can be used to form a six-vector in the so called phase space

$$\begin{pmatrix} \dot{x_1} & \dot{x_2} & \dot{x_3} & \dot{v_1} & \dot{v_2} & \dot{v_3} \end{pmatrix}^T = \begin{pmatrix} v_1 & v_2 & v_3 & \frac{f_1}{m} & \frac{f_2}{m} & \frac{f_2}{m} \end{pmatrix}^T \tag{2.21}$$

Further assuming that the force f is a function of $\mathbf{x}$ and t leads us back to the canonical form in Eq. 2.1. The concatenated vector of a single object can also be imagined as a point traveling in the 6D phase space. Each Newtonian body adds another 6 elements to this vector and the system in its entirety can be viewed as a point traveling in $6n$-space.

Related nomenclature

In order to operate on a robot using mathematical tools, we will represent a robot as single point in a so called *configuration space* $\mathcal{C}$ in later sections (4.1). This configuration space is sometimes described as "R^m"-like, i.e. it is (locally) similar to an Euclidean space of dimension m.
The set of tangent vectors at any $\mathbf{q} \in \mathcal{C}$ forms a tangent space $T_{\mathbf{q}}(\mathcal{C})$. Since these tangents are the derivatives at a point $\mathbf{q}$ it is also sometimes called *velocity space*. Often it also makes sense to combine multiple such tangent spaces into a single structure, called *tangent bundle*: $T(\mathcal{C}) = \cup_{\mathbf{q}\in\mathcal{C}} T_{\mathbf{q}}(\mathcal{C})$.
A pair (configuration, velocity) is called *phase* in Physics and *state* in Control Theory. Accordingly the tangent bundle of $\mathcal{C}$ is referred to as *phase space* and *state space* depending on the context [Lat03].

From here on out we will use $\mathbf{x}(t)$ to denote the state of a system, i.e. the collection of all parameters of the system that influence the "position" in this space. This is not limited to the position and velocity of objects, but can also include their angles and angular velocities.

Rewriting Eq. 2.18 using this vector and denoting the spatial position using $\mathbf{p}$ instead of $\mathbf{x}$ gives us

$$\dot{\mathbf{x}} = \begin{pmatrix} \dot{\mathbf{p}}(t) \\ \dot{\mathbf{v}}(t) \end{pmatrix} = \begin{pmatrix} \mathbf{v}(t) \\ \frac{\mathbf{F}(t)}{m} \end{pmatrix} \tag{2.22}$$

2.3 Rigid bodies

Rigid bodies, unlike Newtonian particles, are not just points in space that have mass attached to them. It is not sufficient to describe them using just their position and velocity. Their orientation is an essential part that needs to be considered. As the shapes of rigid bodies can vary wildly we are going to describe the geometric properties in a local coordinate system called the body-space. Additionally we assume that the origin of the body space coincides with the center of mass of that body. This simplifies several calculations and can be done without loss of generality.

If the rotation about the center of mass is denoted by $R(t)$ and the translation by $\mathbf{p}(t)$ then the transformation from the local coordinate system to the world coordinates can be written as:

$$\mathbf{r_i} = R(t)\mathbf{r_{i_b}} + \mathbf{p}(t) \tag{2.23}$$

where $\mathbf{r_{i_b}}$ is a point in body coordinates and $\mathbf{r_i}$ the corresponding point in world coordinates.

2.3.1 Linear Velocity

In order for a rigid body to move it needs a velocity. As the velocity is the rate of change of the position we will write it as

$$\mathbf{v}(t) = \dot{\mathbf{p}}(t) \tag{2.24}$$

Where $\mathbf{p}(t)$ denotes the center of mass of the body in world coordinates. Consequently $\mathbf{v}$ is the velocity of the center of mass.

2.3.2 Angular Velocity

One of the big differences to Newtonian particles is that the orientation of rigid bodies is important. Similarly to the Linear Velocity the rate of change of the rotation also needs to be modeled. The angular velocity is usually denoted by $\omega(t)$. The direction of this vector is the axis of the rotation and the magnitude corresponds to the speed of the rotation. The next thing that needs to be determined is how $\mathbf{R}(t)$ and $w(t)$ are related. If we assume that $\mathbf{R}(t)$ is matrix then $\omega(t)$ can not be $\dot{\mathbf{R}}(t)$ as $\omega(t)$ is a vector.

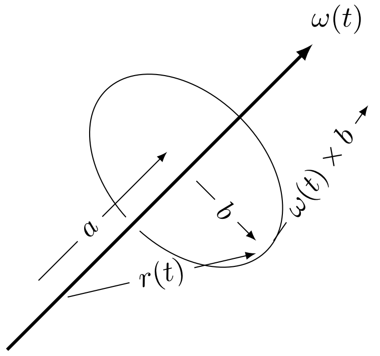


Figure 2.2: Rate of change of a rotating vector

Fig. 2.2 shows how an arbitrary vector $\mathbf{r}(t)$ is changed by the angular velocity. $\mathbf{r}(t)$ is decomposed into $\mathbf{a}$ and $\mathbf{b}$ to calculate $\dot{\mathbf{r}}(t)$. $\mathbf{a}$ is parallel to $\omega(t)$ and $\mathbf{b}$ is perpendicular to it. The resulting motion corresponds to $\mathbf{r}(t)$ tracing a circle that is perpendicular to $\omega(t)$ and whose center lies on that same vector. As the tip of $\mathbf{r}(t)$ is moving along a circle the instantaneous velocity has a magnitude of $|\mathbf{b}||\omega(t)|$ and due to the fact that $\mathbf{b}$ and $\omega(t)$ are perpendicular the magnitude of their cross product is also given by

$$|\omega(t) \times \mathbf{b}| = |\omega(t)||\mathbf{b}| \tag{2.25}$$

This results in $\dot{\mathbf{r}}(t) = \omega(t) \times \mathbf{b}$ and since $\mathbf{r}(t) = \mathbf{a} + \mathbf{b}$ with $\mathbf{a}$ being parallel to $\omega(t)$ and thus $\omega(t) \times \mathbf{a} = 0$ we arrive at

$$\dot{\mathbf{r}}(t) = \omega(t) \times \mathbf{b} = \omega(t) \times \mathbf{b} + \omega(t) \times \mathbf{a} = \omega(t) \times (\mathbf{b} + \mathbf{a}) \tag{2.26}$$

or simply

$$\dot{\mathbf{r}}(t) = \omega(t) \times \mathbf{r} \tag{2.27}$$

Knowing that the columns of the rotation matrix $\mathbf{R}(t)$ correspond to the axes of the rigid body in world coordinates we can now write the change in rotation as

$$\dot{\mathbf{R}} = \begin{pmatrix} \omega(t) \times \begin{pmatrix} r_{xx} \\ r_{xy} \\ r_{xz} \end{pmatrix} & \omega(t) \times \begin{pmatrix} r_{yx} \\ r_{yy} \\ r_{yz} \end{pmatrix} & \omega(t) \times \begin{pmatrix} r_{zx} \\ r_{zy} \\ r_{zz} \end{pmatrix} \end{pmatrix} \tag{2.28}$$

As that is a quite cumbersome way to express this we will rewrite the cross product of two vectors in the following way

$$\mathbf{a} \times \mathbf{b} = \begin{pmatrix} a_y b_z - b_y a_z \\ -a_x b_z + b_x a_z \\ a_x b_y - b_x a_y \end{pmatrix} = \begin{pmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{pmatrix} \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix} = \mathbf{a}^* \mathbf{b} \tag{2.29}$$

This allows us to write $\dot{\mathbf{R}}(t)$ as

$$\dot{\mathbf{R}} = \begin{pmatrix} \omega^*(t) \begin{pmatrix} r_{xx} \\ r_{xy} \\ r_{xz} \end{pmatrix} & \omega^*(t) \begin{pmatrix} r_{yx} \\ r_{yy} \\ r_{yz} \end{pmatrix} & \omega^*(t) \begin{pmatrix} r_{zx} \\ r_{zy} \\ r_{zz} \end{pmatrix} \end{pmatrix} = \omega^*(t)\mathbf{R} \tag{2.30}$$

which is gives us an expression for $\dot{\mathbf{R}}$ in canonical form.

2.3.3 Splitting a rigid body into particles

To make some of the next calculations easier to visualize we are going to pretend that a rigid body consists of many distinct particles. These particles are enumerated using m_i and have a fixed position in body coordinates r_{i_b}. The movement of such a particle in world coordinates can be expressed using Eq. 2.23. The total mass of the rigid body is:

$$M = \sum_{n=0}^{N} m_i \tag{2.31}$$

Such a particle also needs a velocity.

It can be calculated using Eqs. 2.23 and 2.30

$$\dot{r_i}(t) = \omega^*(t)\mathbf{R}(t) r_{i_b} + v(t) \tag{2.32}$$

This allows us to rewrite $\dot{r_i}(t)$ using the position $p(t)$ of the rigid body:

$$\dot{r_i}(t) = \omega^*(t)\mathbf{R}(t)r_{i_b} + v(t) \tag{2.33}$$
$$= \omega^*(t)(\mathbf{R}(t)r_{i_b} + p(t) - p(t)) + v(t) \tag{2.34}$$
$$= \omega^*(t)(r_{i_b}(t) - p(t)) + v(t) \tag{2.35}$$

with $\omega^*(t)a = \omega(t) \times a$ for any vector a this gives

$$\dot{r_i}(t) = \omega(t) \times (r_i - p(t)) + v(t) \tag{2.36}$$

It is important to note here that this allows us to split the velocity into an angular $\omega(t) \times (r_i - p(t))$ and linear $v(t)$ part.

2.3.4 Force and Torque

Another important aspect is the force acting on a rigid body. Here we will use the previously defined concept of particles again. Let us denote the forces acting on an arbitrary point (particle) in this rigid body with $F_i(t)$. Additionally the torque acting on this particle is defined as

$$\tau_i(t) = (r_i(t) - p(t)) \times F_i(t) \tag{2.37}$$

Intuitively this tells us that the torque depends on the distance of this particle from the center of mass. The direction of $\tau_i(t)$ can be thought of as the axis the body would spin about due to $F_i(t)$.

The total force acting on a body is

$$F(t) = \sum_{n=0}^{N} F_i(t) \tag{2.38}$$

and the total torque

$$\tau(t) = \sum_{n=0}^{N} \tau_i(t) = \sum_{n=0}^{N} (r_i(t) - p_i(t)) \times F_i(t) \tag{2.39}$$

It is worth mentioning that the total force contains no information about the distribution of forces over the body, while the total torque does.

2.3.5 Linear Momentum

A particle's linear momentum is usually given as

$$p = mv \tag{2.40}$$

with a mass m and the velocity v.

Similarly to the previous sections the total momentum of a rigid body can be written as

$$P(t) = \sum_{n=0}^{N} m_i \dot{r_i} \tag{2.41}$$

Using Eq. 2.36 this can be rewritten as

$$\begin{aligned} P(t) &= \sum_{n=0}^{N} m_i \dot{r_i} \qquad (2.42) \\ &= \sum_{n=0}^{N} (m_i v(t) + m_i \omega(t) \times (r_i(t) - p(t))) \qquad (2.43) \\ &= \sum_{n=0}^{N} m_i v(t) + \omega(t) \times \sum_{n=0}^{N} m_i (r_i(t) - p(t)) \qquad (2.44) \end{aligned}$$

Here we will use the fact that we chose the origin of the rigid body as the center of mass and its property given in Eq. 2.31. Thus the second sum equals

$$\sum_{n=0}^{N} m_i (r_i(t) - p(t)) = \sum_{n=0}^{N} m_i (R(t) r_{i_b} + p(t) - p(t)) = R(t) \sum_{n=0}^{N} m_{i_b} = 0 \tag{2.45}$$

and

$$P(t) = \sum_{n=0}^{N} m_i v(t) = M v(t) \tag{2.46}$$

This tells us that the total linear momentum of a rigid body is identical to that of a single particle with mass M and velocity $v(t)$.

2.3.6 Angular Momentum

Angular Momentum is the analogue to the linear momentum for rotations. A very important property of the angular momentum is that it is conserved in nature. A body floating in space on which no torque acts has a constant angular momentum. This is not the case for the angular velocity, however. Even if the angular momentum is constant the corresponding velocity may not be - it can vary even if no force acts on the body. This is a reason why it is chosen as a state variable over the angular velocity.

Eq. 2.46 shows the definition for the linear momentum. The angular momentum can be written in a similar fashion

$$L(t) = I(t) \omega(t) \tag{2.47}$$

where $I(t)$ is a rank 2 tensor called the inertia tensor or sometimes inertia matrix. It describes the distribution of mass in a rigid body and depends on its orientation but not its translation.

The linear momentum as well as the angular momentum are linear functions of the velocity. The difference is that the angular momentum is scaled by a matrix while the linear momentum is using a scalar. Additionally $L(t)$ does not depend on translational effects while $P(t)$ is independent of rotational effects.

2.3.7 Inertia Tensor

The inertia tensor represents the transformation to get from the angular velocity to the angular momentum. It can be calculated as

$$I(t) = \sum_{n=0}^{N} \begin{pmatrix} m_i(r'^2_{iy} + r'^2_{iz}) & -m_i r'_{ix} + r'_{iy} & -m_i r'_{ix} + r'_{iz} \\ -m_i r'_{iy} + r'_{ix} & m_i(r'^2_{ix} + r'^2_{iz}) & -m_i r'_{iy} + r'_{iz} \\ -m_i r'_{iz} + r'_{ix} & -m_i r'_{iz} + r'_{iy} & m_i(r'^2_{ix} + r'^2_{iy}) \end{pmatrix} \tag{2.48}$$

where $r' = r_i(t) - p(t)$, i.e. the distance of a particle from the center of mass. Generally the above sum is replaced by an integral to calculate the inertia tensor. Calculating this at each point in time can be very time consuming so rewriting it in terms of body coordinates is a better approach. As $r'^T_i r'_i = r'^2_{ix} + r'^2_{iy} + r'^2_{iz}$, $I(t)$ becomes

$$I(t) = \sum_{n=0}^{N} m_i r'^T_i r'_i \mathcal{I} - \begin{pmatrix} m_i r'^2_{ix} & m_i r_{ix} r'_{iy} & m_i r_{ix} r'_{iz} \\ m_i r_{iy} r'_{ix} & m_i r'^2_{iy} & m_i r_{iy} r'_{iz} \\ m_i r_{iz} r'_{ix} & m_i r_{iz} r'_{iy} & m_i r'^2_{iz} \end{pmatrix} \tag{2.49}$$

and further

$$I(t) = \sum_{n=0}^{N} m_i (r'^T_i r'_i \mathcal{I} - r'_i r'^T_i) \tag{2.50}$$

Using Eq. 2.23 and the fact that rotation matrices are orthogonal ($RR^T = \mathcal{I}$) we can rewrite this as

$$\begin{aligned}
I(t) &= \sum_{n=0}^{N} m_i(r_i'^T r_i' \mathcal{I} - r_i' r_i'^T) && (2.51)\\
&= \sum_{n=0}^{N} m_i((\mathbf{R}(t) r_{i_b})^T (\mathbf{R}(t) r_{i_b}) \mathcal{I} - (\mathbf{R}(t) r_{i_b})(\mathbf{R}(t) r_{i_b})^T) && (2.52)\\
&= \sum_{n=0}^{N} m_i(r_{i_b}^T \mathbf{R}(t)^T \mathbf{R}(t) r_{i_b} \mathcal{I} - \mathbf{R}(t) r_{i_b} r_{i_b}^T \mathbf{R}(t)^T) && (2.53)\\
&= \sum_{n=0}^{N} m_i(r_{i_b}^T r_{i_b} \mathcal{I} - \mathbf{R}(t) r_{i_b} r_{i_b}^T \mathbf{R}(t)^T) && (2.54)\\
&= \sum_{n=0}^{N} m_i(\mathbf{R}(t) r_{i_b}^T r_{i_b} \mathbf{R}(t)^T \mathcal{I} - \mathbf{R}(t) r_{i_b} r_{i_b}^T \mathbf{R}(t)^T) && (2.55)\\
&= \mathbf{R}(t)(\sum_{n=0}^{N} m_i(r_{i_b}^T r_{i_b} \mathcal{I} - r_{i_b} r_{i_b}^T))\mathbf{R}(t)^T && (2.56)
\end{aligned}$$

Defining I_b as the matrix

$$I_b = \sum_{n=0}^{N} m_i(r_{i_b}^T r_{i_b} \mathcal{I} - r_{i_b} r_{i_b}^T) \tag{2.57}$$

gives us a nice way to rewrite the inertia tensor

$$I(t) = R(t) I_b R(t)^T \tag{2.58}$$

I_b only needs to be calculated once for each body and can then be transformed using the rotation of the rigid body to get the required matrix for each point in time.

The inverse of $I(t)$ can also easily be given as

$$I(t)^{-1} = (R(t) I_b R(t)^T)^{-1} = R(t)^T I_b^{-1} R(t) \tag{2.59}$$

2.3.8 The state of a rigid body

All of the above can now be combined to get the state vector for a rigid body, which consists of the body's pose (position and orientation) as well as its linear and angular momentum.

$$\mathbf{X}(t) = \begin{pmatrix} \mathbf{p}(t) \\ \mathbf{R}(t) \\ \mathbf{P}(t) \\ \mathbf{L}(t) \end{pmatrix} \tag{2.60}$$

However what we also need is the derivative of the state.

Eq. 2.46 gives us an expression for the velocity (i.e. $\dot{\mathbf{p}}(t)$) and Eq. 2.30 the counterpart for the angular velocity. This leaves us with the derivative of the linear and angular momentum.

Calculating $\dot{P}$

Previously we conceptualized how forces act on a rigid body by splitting it into a large number of small particles. However, this is not enough any more. In order for the body itself to not deform each of those external forces $F_i(t)$ needs some internal constraint force that counteracts it. For the next part we will assume that these internal forces act passively and do not perform any net work. If we denote the internal constraint force with $F_{ci}(t)$ then the work performed by each particle is

$$\int_{t_0}^{t_1} F_{ci}(t) \cdot \dot{r_i}(t) dt \tag{2.61}$$

with the velocity of that particle being $\dot{r_t}(t)$ and two points in time t_0 and t_1. The net work of that body is zero

$$\sum_{n=0}^{N} \int_{t_0}^{t_1} F_{ci}(t) \cdot \dot{r_i}(t) dt = \int_{t_0}^{t_1} \sum_{n=0}^{N} F_{ci}(t) \cdot \dot{r_i}(t) dt = 0 \tag{2.62}$$

and thus the sum over all particles at each point in time must also be zero. Using the expression in Eq. 2.36 with $r_i'(t) = r_i - p(t)$ leads to

$$\sum_{n=0}^{N} F_{ci}(t) \cdot (v(t) - r_i'(t) \times \omega(t)) = 0 \tag{2.63}$$

which must be true for arbitrary values of $v(t)$ and $\omega(t)$ as they are independent. Choosing $\omega(t) = 0$ gives

$$\sum_{n=0}^{N} F_{ci}(t) \cdot v(t) = 0 \tag{2.64}$$

and similarly letting $v(t) = 0$ shows

$$\sum_{n=0}^{N} -F_{ci}(t) \cdot (r_i'(t) \times \omega(t)) = 0 \tag{2.65}$$

The net force on each particle is the sum of the external force $F_i(t)$ and the internal constraint force $F_{ci}(t)$. This means that we can write the acceleration of such a particle as

$$\ddot{r_i}(t) = \frac{d}{dt}\dot{r_i}(t) = \frac{d}{dt}(v(t) - r_i'(t) \times \omega(t)) = \dot{v}(t) - \dot{r_i'}(t) \times \omega(t) - r_i'(t) \times \dot{\omega}(t) \tag{2.66}$$

Each particle must also obey Newton's law ($f = ma$) so

$$m_i\ddot{r}_i(t) - F_i(t) - F_{ci}(t) = 0 \tag{2.67}$$

$$m_i\Big(\dot{v}(t) - \dot{r}'_i(t) \times \omega(t) - r'_i(t) \times \dot{\omega}(t)\Big) - F_i(t) - F_{ci}(t) = 0 \tag{2.68}$$

Summing over all particles gives

$$\begin{aligned}&\sum_{n=0}^{N} m_i\Big(\dot{v}(t) - \dot{r}'_i(t) \times \omega(t) - r'_i(t) \times \dot{\omega}(t)\Big) - F_i(t) - F_{ci}(t) = \\ &\sum_{n=0}^{N} m_i\dot{v}(t) - \sum_{n=0}^{N} m_i\dot{r}'_i(t) \times \omega(t) - \sum_{n=0}^{N} m_i r'_i(t) \times \dot{\omega}(t) - \sum_{n=0}^{N} F_i(t) - \sum_{n=0}^{N} F_{ci}(t) = \\ &\sum_{n=0}^{N} m_i\dot{v}(t) - \Big(\frac{d}{dt}\sum_{n=0}^{N} m_i r'_i(t)\Big) \times \omega(t) - \sum_{n=0}^{N} m_i r'_i(t) \times \dot{\omega}(t) - \sum_{n=0}^{N} F_i(t) - \sum_{n=0}^{N} F_{ci}(t) = \\ &0\end{aligned} \tag{2.69}$$

$\sum_{n=0}^{N} m_i r'_i(t)$ and $\frac{d}{dt}\sum_{n=0}^{N} m_i r'_i(t)$ are both 0 because we chose a coordinate system with the center of mass at the origin.
Finally we get

$$\sum_{n=0}^{N} m_i\dot{v}(t) - \sum_{n=0}^{N} F_i(t) = 0 \tag{2.70}$$

or

$$M\dot{v}(t) = \dot{P} = F(t) \tag{2.71}$$

Calculating $\dot{L}$

To get an expression for $\dot{L}(t)$ we will start with Eq. 2.68. Applying the $*$ operator to both sides gives:

$$r'^*_i(t)m_i\Big(\dot{v}(t) - \dot{r}'^*_i(t)\omega(t) - r'^*_i(t)\dot{\omega}(t)\Big) - r'^*_i(t)\Big(F_i(t) + F_{ci}(t)\Big) = r'^*_i(t)\mathbf{0} = 0 \tag{2.72}$$

Calculating the sum over all particles results in

$$\begin{aligned}&\sum_{n=0}^{N} r'^*_i(t)m_i\dot{v}(t) - \sum_{n=0}^{N} r'^*_i(t)m_i\dot{r}'^*_i(t)\omega(t) - \sum_{n=0}^{N} m_i r'^*_i(t)r'^*_i(t)\dot{\omega}(t) \\ &- \sum_{n=0}^{N} r'^*_i(t)F_i(t) - \sum_{n=0}^{N} r'^*_i(t)F_{ci}(t) = 0\end{aligned} \tag{2.73}$$

As $\sum_{n=0}^{N} r_i^{\prime *}(t) F_{ci}(t) = 0$ and $\sum_{n=0}^{N} m_i r_i'(t) = 0$ we get

$$-\Big(\sum_{n=0}^{N} r_i^{\prime *}(t) m_i \dot{r}_i^{\prime *}(t)\Big)\omega(t) - \Big(\sum_{n=0}^{N} m_i r_i^{\prime *}(t) r_i^{\prime *}(t)\Big)\dot{\omega}(t) - \sum_{n=0}^{N} r_i^{\prime *}(t) F_i(t) = 0 \tag{2.74}$$

Using the definition of r_i' and Eq. 2.39 leads to

$$-\Big(\sum_{n=0}^{N} r_i^{\prime *}(t) m_i \dot{r}_i^{\prime *}(t)\Big)\omega(t) - \Big(\sum_{n=0}^{N} m_i r_i^{\prime *}(t) r_i^{\prime *}(t)\Big)\dot{\omega}(t) = \tau(t) \tag{2.75}$$

Remembering the definition of the * operator shows that $-a^* a^*$ is equivalent to $(a^T a)\mathcal{I} - a a^T$. Thus

$$\sum_{n=0}^{N} -m_i r_i^{\prime *}(t) r_i^{\prime *}(t) = \sum_{n=0}^{N} m_i \Big((r_i^{\prime T}(t) r_i'(t))\mathcal{I} - r_i'(t) r_i^{\prime T}(t)\Big) = I(t) \tag{2.76}$$

The last part we want to avoid is having to calculate $\sum_{n=0}^{N} r_i^{\prime *}(t) m_i \dot{r}_i^{\prime *}(t)$ as that is just as expensive as calculating the inertia tensor in the first place. To that end we will use the two equations $\dot{r}_i'(t) = \omega(t) \times r_i'(t)$ and $r_i^{\prime *}(t)\omega(t) = -\omega(t) \times r_i'(t)$

$$\begin{aligned}
\sum_{n=0}^{N} m_i \dot{r}_i^{\prime *}(t) r_i^{\prime *}(t) \omega(t) &= 0 \\
\sum_{n=0}^{N} m_i (\omega(t) \times r_i'(t))^* (-\omega(t) \times r_i'(t) &= 0 \\
\sum_{n=0}^{N} -m_i (\omega(t) \times r_i'(t)) \times (\omega(t) \times r_i'(t)) &= 0
\end{aligned} \tag{2.77}$$

Adding this to Eq. 2.75 and substituting Eq. 2.76 gives

$$\Big(\sum_{n=0}^{N} -m_i r_i^{\prime *}(t) \dot{r}_i^{\prime *}(t) - m_i \dot{r}_i^{\prime *}(t) r_i^{\prime *}(t)\Big)\omega(t) + I(t)\dot{\omega}(t) = \tau(t) \tag{2.78}$$

Which can be simplified further by using the derivative of the inertia tensor

$$\dot{I}(t) = \frac{d}{dt}\sum_{n=0}^{N} -m_i r_i^{\prime *}(t) r_i^{\prime *}(t) = \sum_{n=0}^{N} -m_i r_i^{\prime *}(t) \dot{r}_i^{\prime *}(t) - m_i \dot{r}_i^{\prime *}(t) r_i^{\prime *}(t) \tag{2.79}$$

to

$$\boxed{\dot{I}(t)\omega(t) + I(t)\dot{\omega}(t) = \frac{d}{dt}(I(t)\omega(t)) = \tau(t)} \tag{2.80}$$

where the definition of the angular momentum from Eq. 2.47 is $L(t) = I(t)\omega(t)$

Another way this is often written is without the derivative of the inertia tensor:

$$\dot{I} = \frac{d}{dt}RI_bR^T \tag{2.81}$$

$$= \dot{R}I_bR^T + RI_b\dot{R}^T \tag{2.82}$$

$$= \omega^* RI_bR^T + RI_b(\omega^* R)^T \tag{2.83}$$

$$= \omega^* I + I\omega^* \tag{2.84}$$

which can be used with the identity $\omega^{*T} = -\omega^*$ to write τ as

$$\tau(t) = I(t)\dot{\omega}(t) + \omega(t) \times I(t)\omega(t) \tag{2.85}$$

Finally using Eqs. 2.46 2.71, 2.47, 2.58 gives us

$$\dot{\mathbf{X}}(t) = \frac{d}{dt}\begin{pmatrix}\mathbf{p}(t)\\ \mathbf{R}(t)\\ \mathbf{P}(t)\\ \mathbf{L}(t)\end{pmatrix} = \begin{pmatrix}\mathbf{v}(t)\\ \omega(t)^*\mathbf{R}(t)\\ \mathbf{F}(t)\\ \tau(t)\end{pmatrix} \tag{2.86}$$

2.4 Constraints

2.4.1 Single Particle

So far we have looked at Newtonian particles and rigid bodies and how they behave individually under the influence of external forces. To get more interesting and useful results we will need to look at how they interact with each other and the environment. Take a simple pendulum for example - so far we have described how a particle would behave under the influence of gravity, for example, but no way of specifying that the weight at the end of the pendulum is a fixed distance from some pivoting point.

This is where constraints come in. They are used to describe the legal states of a system. They are of the form:

$$C(\mathbf{x}) = 0 \tag{2.87}$$

where $\mathbf{x}$ is the state of the system. $C(\mathbf{x})$ is an implicit function, i.e. it relates all elements of the vector $\mathbf{x}$ to each other. An implicit function that describes a unit circle could be given by $x^2 + y^2 - 1 = 0$ for example.

If $C(\mathbf{x})$ describes all allowed states of the system, then $\dot{C}(\mathbf{x})$ and $\ddot{C}(\mathbf{x})$ describe the valid velocities and accelerations, respectively. Or

$$\dot{C}(\mathbf{x}) = \mathbf{x} \cdot \dot{\mathbf{x}} = 0 \tag{2.88}$$

and

$$\ddot{C}(\mathbf{x}) = \ddot{\mathbf{x}} \cdot \mathbf{x} + \dot{\mathbf{x}} \cdot \dot{\mathbf{x}} = 0 \tag{2.89}$$

This means that if the initial position and velocity satisfy the constraints then, in principle all that needs to be done is to ensure that Eq. 2.89 is satisfied from that point onward. To deal with constrains a new virtual force is introduced that counteracts the external applied force in order to satisfy those constraints.

Newton's law with this new force can be written as

$$\ddot{C}(\mathbf{x}) = \frac{\mathbf{f} + \hat{\mathbf{f}}}{m} \tag{2.90}$$

where $\mathbf{f}$ is a given applied force and $\hat{\mathbf{f}}$ is the unknown constraint force.

substituting the above equation in 2.89 leads to

$$\ddot{C}(\mathbf{x}) = \frac{\mathbf{f} + \hat{\mathbf{f}}}{m} \cdot \mathbf{x} + \dot{\mathbf{x}} \cdot \dot{\mathbf{x}} = 0 \tag{2.91}$$

$$\hat{\mathbf{f}} \cdot \mathbf{x} = -\mathbf{f} \cdot \mathbf{x} - m\dot{\mathbf{x}} \cdot \dot{\mathbf{x}} \tag{2.92}$$

This is an equation with two unknowns so we will need some other information to solve it. For this reason we require that the constraint does not add or remove energy to the system, i.e. it is passive and lossless. The kinetic energy can then be written as

$$T = \frac{m}{2} \dot{\mathbf{x}} \cdot \mathbf{x} \tag{2.93}$$

with its derivative

$$\dot{T} = m\ddot{\mathbf{x}} \cdot \dot{\mathbf{x}} = m\mathbf{f} \cdot \dot{\mathbf{x}} + m\hat{\mathbf{f}} \cdot \dot{\mathbf{x}} \tag{2.94}$$

The previously mentioned requirement implies that the second part of the equation is 0 for every valid $\dot{\mathbf{x}}$ (Eq. 2.88). This means that $\hat{\mathbf{f}}$ must point in the direction of $\mathbf{x}$

$$\hat{\mathbf{f}} = \lambda \mathbf{x} \tag{2.95}$$

substituting in Eq. 2.92 allows us to solve for λ

$$\boxed{\lambda = \frac{-\mathbf{f} \cdot \mathbf{x} - m\dot{\mathbf{x}} \cdot \dot{\mathbf{x}}}{\mathbf{x} \cdot \mathbf{x}}} \tag{2.96}$$

Finally $\hat{\mathbf{f}}$ can be calculated and the system can be iterated with $\ddot{\mathbf{x}} = \frac{\hat{\mathbf{f}} + \mathbf{f}}{m}$ instead of Eq. 2.18.

2.4.2 Multiple Particles

Usually one is interested in simulating more than a single particle. For this reason we will extend the above method to handle an arbitrary number of such particles. We start by combining the positions of all particles into a single state vector $\mathbf{q}$. Its length is $3n$ for n particles in 3D. Similarly the masses are put into a diagonal matrix ($\mathbf{M}$) where, again for 3D, the first three diagonal elements are the mass of the first particle, the next three the mass of the second and so on. The inverse of $\mathbf{M}$ can easily be calculated by taking the reciprocal of each element to get another matrix $\mathbf{W}$. Similarly to the state vector all forces acting on the particles are collected in a vector $\mathbf{Q}$. This allows us to write Newton's equation of motion as

$$\ddot{\mathbf{q}} = \mathbf{W}\mathbf{Q} \tag{2.97}$$

All constraints on these particles are collected in a vector function $\mathbf{C}(\mathbf{q})$ which takes a vector of length $3n$ and returns a vector of length m which is the number of constraints in the system. Like in the previous section we require that $\mathbf{C} = \dot{\mathbf{C}} = 0$ and then attempt to find a constraint force $\hat{\mathbf{Q}}$ that, when added to the applied force $\mathbf{Q}$, guarantees $\ddot{\mathbf{C}} = 0$.

Differentiating $\mathbf{C}$ gives

$$\dot{\mathbf{C}} = \frac{\partial \mathbf{C}}{\partial \mathbf{q}} \dot{\mathbf{q}} \tag{2.98}$$

The matrix $\frac{\partial \mathbf{C}}{\partial \mathbf{q}}$ is usually referred to as the Jacobian and denoted with $\mathbf{J}$. The sparsity of this matrix depends on how the constraints interact with multiple particles. If, for example, there are n constraints that act on each particle independently then the Jacobian is a diagonal block matrix with each block describing the relation between x, y and z coordinate of a particle.

Taking the derivative of $\dot{\mathbf{C}}$ gives

$$\ddot{\mathbf{C}} = \dot{\mathbf{J}}\dot{\mathbf{q}} + \mathbf{J}\ddot{\mathbf{q}} \tag{2.99}$$

where $\dot{J}$ could be written as $\frac{\partial \mathbf{J}}{\partial \mathbf{q}} \dot{\mathbf{q}}$. This would involve taking the derivative of a matrix with respect to a vector which results in a rank 3 tensor (3D array) and is generally cumbersome. In many cases an expression for $\dot{\mathbf{C}}$ is available and so $\dot{J}$ becomes $\frac{\partial \dot{\mathbf{C}}}{\partial \mathbf{q}}$ instead which is more manageable. Next we replace $\ddot{\mathbf{q}}$ with the sum of constraint force and applied force times the inverse mass matrix and set $\ddot{\mathbf{C}} = 0$

$$\ddot{\mathbf{C}} = \dot{\mathbf{J}}\dot{\mathbf{q}} + \mathbf{J}\mathbf{W}(\mathbf{Q} + \hat{\mathbf{Q}}) \tag{2.100}$$

$$\mathbf{J}\mathbf{W}\hat{\mathbf{Q}} = -\dot{\mathbf{J}}\dot{\mathbf{q}} - \mathbf{J}\mathbf{W}\mathbf{Q} \tag{2.101}$$

Like in the previous case with only a single particle we have too many unknown variables to solve the equation by itself. Again we make use of the principle of virtual work. All legal velocities must satisfy $\mathbf{J}\dot{\mathbf{x}} = 0$ and consequently

$$\hat{\mathbf{Q}} \cdot \dot{\mathbf{x}} = 0, \forall \dot{\mathbf{x}} | \mathbf{J}\dot{\mathbf{x}} = 0 \tag{2.102}$$

This allows us to express $\hat{\mathbf{Q}}$ as

$$\hat{\mathbf{Q}} = \mathbf{J}^T \lambda \tag{2.103}$$

where λ is a vector with the dimensions of $\mathbf{C}$. A way of imagining this is to view the matrix $\mathbf{J}$ as a collection of gradients for the constraint functions. Since $\mathbf{C} = 0$ those gradients are normals to the constraint hyper-surfaces and represent the directions the system is not allowed to move in. Vectors of the form $\mathbf{J}^T\lambda$ are the linear combinations of those gradients and consequently span the set of prohibited directions. The principle of virtual work then requires that the dot product with any allowed direction is zero.

Finally this allows us to rewrite Eq. 2.101 in the following way

$$\boxed{\mathbf{J}\mathbf{W}\mathbf{J}^{\mathbf{T}}\lambda = -\dot{\mathbf{J}}\dot{\mathbf{q}} - \mathbf{J}\mathbf{W}\mathbf{Q}} \tag{2.104}$$

which can be solved for λ and used to calculate $\hat{\mathbf{Q}}$.

Compensating for numerical drift

As with all systems with limited precision the numerical calculations of the above matrices will accumulate drift over time. This can be compensated by changing the requirement that $\ddot{\mathbf{C}} = 0$ slightly. Two new terms are introduced to pull the system back toward legal states

$$\ddot{\mathbf{C}} = -k_s\mathbf{C} - k_d\dot{\mathbf{C}} \tag{2.105}$$

k_s and k_d can be viewed as spring and damping constants, respectively. The final constraint force equation that can be solved for λ is then given by

$$\mathbf{J}\mathbf{W}\mathbf{J}^{\mathbf{T}}\lambda = -\dot{\mathbf{J}}\dot{\mathbf{q}} - \mathbf{J}\mathbf{W}\mathbf{Q} - k_s\mathbf{C} - k_d\dot{\mathbf{C}} \tag{2.106}$$

2.4.3 Lagrangian Dynamics

Constrained particle systems may be sufficient to describe a number of scenarios, but we do not want to have to split rigid bodies into a large number of particles that are connected by constraints. What we want is to describe constraints that operate on these rigid bodies themselves. Previously each constraint described a hyper-surface in our state space and the intersections of these hyper-surfaces described our valid states. This time the constraints are described using parametric functions, i.e. a function $\mathbf{q}(\mathbf{u})$ with

$\dim \mathbf{u} < \dim \mathbf{q}$ which specifies all legal states. Using a particle on a unit circle again such a function could be $\mathbf{q}(\mathbf{u}) = \mathbf{q}([\theta]) = [\cos\theta, \sin\theta]^T$. This has the advantage of reducing the degrees of freedom. Instead of depending on x and y, now the constraint only depends on θ.

The constrained system's equations of motion need to be rewritten in terms of the new constrained representation rather than the unconstrained one we used previously. The advantage of that method is that it removes redundant information. This improves the stability of the simulation, but automatically combining constraints becomes a lot harder. These new equations are known as Lagrange's equations of motion.

We will start with the global state vector $\mathbf{q}$, the diagonal mass matrix $\mathbf{M}$ and the forces $\mathbf{Q}$ and $\hat{\mathbf{Q}}$. This time the elements of $\mathbf{q}$ are not independent variables but are given by a function $\mathbf{q}(\mathbf{u})$ instead. The goal is to solve for $\ddot{\mathbf{u}}$ while keeping the constraints satisfied.

The Jacobian of the constraint function is used again

$$\mathbf{J} = \frac{\partial \mathbf{q}}{\partial \mathbf{u}} \tag{2.107}$$

The valid velocities and accelerations can again be calculated using the chain rule

$$\dot{\mathbf{q}} = \mathbf{J}\dot{\mathbf{u}} \tag{2.108}$$

$$\ddot{\mathbf{q}} = \mathbf{J}\ddot{\mathbf{u}} + \dot{\mathbf{J}}\dot{\mathbf{u}} \tag{2.109}$$

The principle of virtual work requires

$$\hat{\mathbf{Q}}^T \mathbf{J}\dot{\mathbf{u}} = 0, \forall \dot{\mathbf{u}} \tag{2.110}$$

Similarly to the previous description this means that $\mathbf{J}^T \hat{\mathbf{Q}} = 0$ and the unconstrained equation of motion is given by

$$\mathbf{M}\ddot{\mathbf{q}} = \mathbf{Q} + \hat{\mathbf{Q}} \tag{2.111}$$

This time instead of solving for the constraint force we remove it completely by multiplying both sides with $\mathbf{J}^T$

$$\mathbf{J^T M}\ddot{\mathbf{q}} - \mathbf{J^T Q} = 0 \tag{2.112}$$

Substituting Eq. 2.109 for $\ddot{\mathbf{q}}$ gives us the classical Lagrangian equation of motion

$$\boxed{\mathbf{J^T M J}\ddot{\mathbf{u}} + \mathbf{J^T M}\dot{\mathbf{J}}\dot{\mathbf{u}} - \mathbf{J^T Q} = 0} \tag{2.113}$$

Dynamics Equation

In the literature one will often find the equation

$$M(\mathbf{q})\ddot{\mathbf{q}} + C(\mathbf{q}, \dot{\mathbf{q}}) = \mathbf{Q} \tag{2.114}$$

instead. The equation contains the mass matrix M, the Coriolis and centrifugal term C and the vector of generalized forces Q. The similarity to Eq. 2.113 should be obvious, but in the hopes of making it entirely clear we will derive the above equation using the Newton-Euler equations. We start by rewriting Eq. 2.85 and $f = ma$ in vector form:

$$\begin{pmatrix} m\mathcal{I}_\mathbf{3} & 0 \\ 0 & I \end{pmatrix} \begin{pmatrix} \dot{\mathbf{v}} \\ \dot{\omega} \end{pmatrix} + \begin{pmatrix} \mathbf{0} \\ \omega \times \mathbf{I}\omega \end{pmatrix} = \begin{pmatrix} \mathbf{f} \\ \tau \end{pmatrix} \tag{2.115}$$

Then the Jacobian is split into terms for $\mathbf{v}$ and ω

$$\begin{pmatrix} \mathbf{v} \\ \omega \end{pmatrix} = J(\mathbf{q})\dot{\mathbf{q}} = \begin{pmatrix} J_v \\ J_\omega \end{pmatrix} \dot{\mathbf{q}} \tag{2.116}$$

and the matrix M_c defined as

$$M_c = \begin{pmatrix} m\mathcal{I}_\mathbf{3} & 0 \\ 0 & I \end{pmatrix} \tag{2.117}$$

With Eq. 2.109 in mind this allows us to rewrite Eq. 2.115 as

$$M_c(\dot{J}\dot{q}) + \begin{pmatrix} 0 \\ (J_\omega\dot{\mathbf{q}}) \times IJ_\omega\dot{\mathbf{q}} \end{pmatrix} = \begin{pmatrix} \mathbf{f} \\ \tau \end{pmatrix} \tag{2.118}$$

$$M_cJ\ddot{\mathbf{q}} + M_c\dot{J}\dot{\mathbf{q}} + \begin{pmatrix} 0 & 0 \\ 0 & (J_\omega\dot{\mathbf{q}})^* \end{pmatrix} M_cJ\dot{\mathbf{q}} = \begin{pmatrix} \mathbf{f} \\ \tau \end{pmatrix} \tag{2.119}$$

As a last step the Cartesian forces are converted to the generalized space by multiplying with J^T to get

$$(J^TM_cJ)\ddot{\mathbf{q}} + (J^TM_c\dot{J} + J^T \begin{pmatrix} 0 & 0 \\ 0 & (J_\omega\dot{\mathbf{q}})^* \end{pmatrix} M_cJ)\dot{\mathbf{q}} = J_v^T\mathbf{f} + J_\omega^T\tau \tag{2.120}$$

This allows us to list the factors in Eq. 2.114

$$M(\mathbf{q}) = J^TM_cJ \tag{2.121}$$

$$C(\mathbf{q}, \dot{\mathbf{q}}) = (J^TM_c\dot{J} + \begin{pmatrix} 0 & 0 \\ 0 & (J_\omega\dot{\mathbf{q}})^* \end{pmatrix} M_cJ)\dot{\mathbf{q}} \tag{2.122}$$

$$Q = J_v^T\mathbf{f} + J_\omega^T\tau \tag{2.123}$$

2.5 Articulated Rigid Body Dynamics

Next we want to expand on the previous sections by introducing articulated rigid bodies. Such an articulated rigid body system is represented by a tree-like structure of rigid links connected by joints. An important point here is that each body has exactly one parent, but there is no restriction on the number of children.

The state of the system will be expressed in terms of the generalized coordinates. As the root of the tree does not have any parents the generalized coordinates of the system are the degrees of freedom (DOF) of that link.

The state of this system can be expressed as $(\mathbf{x_k}, R_k, \mathbf{v_k}, \omega_\mathbf{k})$ where $k = 1, .., m$ is the number of links in the system. $\mathbf{x_k}$ and R_k denote the position of the COM and the rotation of the link, respectively. $\mathbf{v_k}$ and $\omega_\mathbf{k}$ are the linear and angular velocity in the world frame. The Cartesian force and torque applied to each link in the world frame are given by $(\mathbf{f_k}, \tau_\mathbf{k})$.

The system can also be expressed in generalized coordinates where the state is given by $(\mathbf{q}, \dot{\mathbf{q}})$, with $\mathbf{q} = (\mathbf{q_1}, ..., \mathbf{q_i}, ..., \mathbf{q_m})$. Each $\mathbf{q_i}$ is the set of DOFs of the joint connecting the link i to its parent.

In order to make the following sections more concise we will introduce several notations:

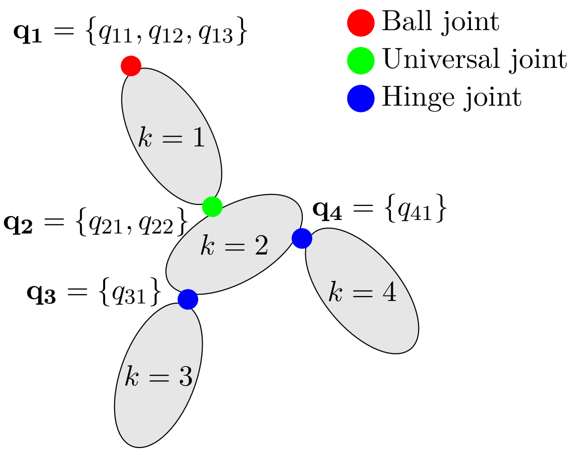

$\mathbf{q_1} = \{q_{11}, q_{12}, q_{13}\}$
$\mathbf{q_2} = \{q_{21}, q_{22}\}$
$\mathbf{q_3} = \{q_{31}\}$
$\mathbf{q_4} = \{q_{41}\}$
$k = 1$
$k = 2$
$k = 3$
$k = 4$
● Ball joint
● Universal joint
● Hinge joint

Figure 2.3: Example of an articulated rigid body system

- $p(k)$ gives the index of the parent link. E.g. $p(4) = 2$ in Fig. 2.3. Additionally $p(1, k)$ returns the set of rigid links from the root to the link denoted by k. E.g. $p(1, 3) = \{1, 2, 3\}$
- $n(k)$ returns the number of degrees of freedom of the joint connecting link k to its parent. E.g. $n(1) = 3$, $n(3) = 1$. The total number of DOFs is denoted by n ($n = 7$ in Fig. 2.3)
- R_k denotes the rotation matrix for the link k from its parent. It only depends on the DOFs $\mathbf{q_k}$. R_k^0 is the product of the rotations from the world frame to the local frame. A recursive definition is $R_k^0 = R_{p(k)}^0 R_k$ with $R_{p(1)}^0 = \mathcal{I}_3$

Eq. 2.116 shows the relationship between the generalized coordinates and Cartesian coordinates for a single rigid body. We will need to produce a similar expression for an articulated rigid body system. Such an expression for the angular velocity (in skew symmetric matrix form) can be derived in the following way:

$$\begin{aligned}
\omega_{\mathbf{k}}^{\mathbf{0}} &= \dot{R}_k^0 {R_k^0}^T && (2.124)\\
&= (\dot{R^0_{p(k)} R_k})(R^0_{p(k)} R_k)^T && (2.125)\\
&= (\dot{R}^0_{p(k)} R_k + R^0_{p(k)} \dot{R}_k) R_k^T {R^0_{p(k)}}^T && (2.126)\\
&= \dot{R}^0_{p(k)} {R^0_{p(k)}}^T + R^0_{p(k)} (\dot{R}_k R_k^T) {R^0_{p(k)}}^T && (2.127)\\
&\equiv \omega^0_{p(k)} + R^0_{p(k)} \omega_{\mathbf{k}} {R^0_{p(k)}}^T && (2.128)
\end{aligned}$$

$\omega_{\mathbf{k}}^{\mathbf{0}}$ denotes the angular velocity of link k in the global frame and $\omega_{\mathbf{k}}$ is used for the angular velocity in the frame of the link's parent.

Eq. 2.116 allows us to write $\omega_{\mathbf{k}} = \hat{J}_{\omega k} \dot{\mathbf{q}}_{\mathbf{k}}$ where $\hat{J}_{\omega k}$ is the local $3 \times n(k)$ Jacobian relating the joint velocity of the link k to the angular velocity in the parent frame.

The second part of Eq. 2.128 can be rewritten using the a property of skew symmetric matrices - $R \omega R^T = R\omega$:

$$\begin{aligned}
\omega_k^0 &= \omega^0_{p(k)} + R^0_{p(k)} \hat{J}_{\omega k} \dot{\mathbf{q}}_{\mathbf{k}} && (2.129)\\
&= \sum_{l \in p(1,k)} R^0_{p(l)} \hat{J}_{\omega l} \dot{\mathbf{q}}_{\mathbf{l}} && (2.130)\\
&\equiv J_{\omega k} \dot{\mathbf{q}} && (2.131)
\end{aligned}$$

where $J_{\omega k}$ is

$$J_{\omega k} = (\hat{J}_{\omega 1} \;\cdots\; R^0_{p(l)} \hat{J}_{\omega l} \;\cdots\; \mathbf{0} \;\cdots) \tag{2.132}$$

The $3 \times n(l)$ zero matrices in the above equation are the DOFs that are not part of the chain from the root to the link k.

Examples for Fig. 2.3

To illustrate what typical local Jacobians and the corresponding angular velocities look like we will give a few example for the ones in Fig. 2.3.

$$\begin{aligned}
\omega_{\mathbf{1}} &= (\hat{J}_{\omega 1} \; \mathbf{0} \; \mathbf{0} \; \mathbf{0}) \dot{\mathbf{q}} && (2.133)\\
\omega_{\mathbf{3}} &= (\hat{J}_{\omega 1} \; R_1^0 \hat{J}_{\omega 2} \; R_2^0 \hat{J}_{\omega 3} \; \mathbf{0}) \dot{\mathbf{q}} && (2.134)
\end{aligned}$$

with $\hat{J}_{\omega 1} \in \mathcal{R}^{3\times 3}$, $\hat{J}_{\omega 2} \in \mathcal{R}^{3\times 2}$ and $\hat{J}_{\omega 3} \in \mathcal{R}^{3\times 1}$
If we further assume that the ball joint at the link $k = 1$ is represented by three Euler angles R_x, R_y and R_z with $R_1(\mathbf{q_1}) = R_x(q_{11}) R_y(q_{12}) R_z(q_{13})$ then

$$\hat{J}_{\omega 1} = \left(\begin{pmatrix} 1\\0\\0 \end{pmatrix} \quad R_x \begin{pmatrix} 0\\1\\0 \end{pmatrix} \quad R_x R_y \begin{pmatrix} 0\\0\\1 \end{pmatrix} \right) \tag{2.135}$$

$\mathbf{v}_k$ can be written in a similar fashion:

$$\mathbf{v}_k = J_{vk}\dot{\mathbf{q}} \tag{2.136}$$

J_{vk} are the partial derivatives of the position with respect to the generalized coordinates:

$$J_{vk} = \frac{\partial \mathbf{x}_k}{\partial \mathbf{q}} = \frac{\partial W_k^0 \mathbf{c}_k}{\partial \mathbf{q}} \tag{2.137}$$

where W_k^0 denotes the homogeneous transformation from the world frame to the local frame, i.e. W_k^0 includes the translations between links. $\mathbf{c}_k$ is the center of mass of the rigid link in its local frame.

This allows us to write the Cartesian velocities in this way:

$$\begin{pmatrix} \mathbf{v}_k \\ \omega_k \end{pmatrix} = \begin{pmatrix} J_{vk} \\ J_{\omega k} \end{pmatrix} \dot{\mathbf{q}} \tag{2.138}$$

$$\mathbf{V}_k = J_k \dot{\mathbf{q}} \tag{2.139}$$

Since the kinetic energy of the whole system is the sum of the kinetic energies of its parts we can write the equations of motion of our system as (cmp. Eq. 2.114)

$$\boxed{\sum_k J_k^T Q = \sum_k (J_k^T M_{ck} J_k)\ddot{\mathbf{q}} + \sum_k (J_k^T M_{ck} \dot{J}_k + J_k^T \omega_{\mathbf{k}} M_{ck} J_k)\dot{\mathbf{q}}} \tag{2.140}$$

2.5.1 Inverse Dynamics

In the previous section we looked at how a system behaves when all forces are given and we want to calculate the resulting motion of that system. In many cases, however, we want to specify the motion of a system and calculate the required forces at certain points of interest (e.g. joints). This problem is called *inverse dynamics*.

The sum of forces for each link k must be zero. If the forces acting on a link from the parent are given by (f_k, τ_k) and the set of the children by $c(k)$ then the total force acting on the body is:

$$m_k \dot{\mathbf{v}}^l = \mathbf{f}_k^l - \sum_{i \in c(k)} \mathbf{R}_i \mathbf{f}_i^l \tag{2.141}$$

where the superscript l denotes quantities in the local frame. A similar expression for the angular velocity is:

$$\mathbf{I}_{ck}\dot{\omega}_k^l + \omega_k^l \times \mathbf{I}_{ck}\omega_k^l = \tau_k^l - \mathbf{c}_k \times \mathbf{f}_k^l - \sum_{i \in c(k)} (\mathbf{R}_i \tau_i^l + (\mathbf{d}_i - \mathbf{c}_k) \times (\mathbf{R}_i \mathbf{f}_i^l)) \tag{2.142}$$

where $\mathbf{d}_i$ is the vector from the parent joint to the child joint and $\mathbf{c}_k$ the vector from the parent joint to the center of mass of the link.

The actual calculation is usually done in two passes: first the velocities and accelerations are calculated and then these are used with the Newton-Euler equations to compute the forces and torques transmitted between links.

Velocity and Acceleration

Since we are using a tree structure to represent the rigid body system we can calculate the required values recursively. That is, if the corresponding values of the parent joint ($\mathbf{v}^l_{p(k)}$, $\omega^l_{p(k)}$, $(\dot{\mathbf{v}}_{p(k)})^l$, $(\dot{\omega}_{p(k)})^l$) are known then we can calculate the values for the child link. The linear velocity of the child with the center of mass in the global frame $\mathbf{W}^0_k\mathbf{c}_k$ (a 4 × 4 matrix) is

$$\begin{aligned}
\mathbf{v}_k &= \dot{\mathbf{W}}^0_k\mathbf{c}_k = \dot{\mathbf{W}}^0_{p(k)}\mathbf{W}_k\mathbf{c}_k + \mathbf{W}^0_{p(k)}\dot{\mathbf{W}}_k\mathbf{c}_k && (2.143)\\
&= \dot{\mathbf{W}}^0_{p(k)}(\mathbf{c}_{p(k)} + \mathbf{W}_k\mathbf{c}_{\mathbf{k}} - \mathbf{c}_{p(k)}) + \mathbf{W}^0_{p(k)}\dot{\mathbf{W}}_k\mathbf{c}_k && (2.144)\\
&= \mathbf{v}_{p(k)} + \dot{\mathbf{W}}^0_{p(k)}(\mathbf{W}_k\mathbf{c}_{\mathbf{k}} - \mathbf{c}_{p(k)}) + \mathbf{W}^0_{p(k)}\dot{\mathbf{W}}_k\mathbf{c}_k && (2.145)
\end{aligned}$$

Note that the fourth column of $\mathbf{W}_k\mathbf{c}_{\mathbf{k}} - \mathbf{c}_{p(k)}$ as well as $\dot{\mathbf{W}}\mathbf{c}_k$ is zero to remove the translational part of the transformation. This allows us to express $\mathbf{v}_k$ in Cartesian space as:

$$\begin{aligned}
\mathbf{v}_k &= \mathbf{v}_{p(k)} + \dot{\mathbf{R}}^0_{p(k)}(\mathbf{R}_k\mathbf{c}_k + \mathbf{d}_k - \mathbf{c}_{p(k)}) + \mathbf{R}^0_{p(k)}\dot{\mathbf{R}}\mathbf{c}_k && (2.146)\\
&= \mathbf{v}_{p(k)} + \omega^*_{p(k)}\mathbf{R}^0_{p(k)}(\mathbf{R}_k\mathbf{c}_k + \mathbf{d}_k - \mathbf{c}_{p(k)}) + \mathbf{R}^0_{p(k)}\hat{\omega}^*\mathbf{R}_k\mathbf{c}_k && (2.147)
\end{aligned}$$

To transform the velocity into the local frame we need to multiply the previous result with the transpose (inverse) of the rotation matrix.

$$\begin{aligned}
\mathbf{v}^l_k &= \mathbf{R}^{0^T}_k\mathbf{v}_k = \mathbf{R}^T_k{\mathbf{R}^0_{p(k)}}^T\mathbf{v}_k && (2.148)\\
&= \mathbf{R}^T_k(\mathbf{v}_{p(k)} + \omega^*_{p(k)}(\mathbf{R}_k\mathbf{c}_k + \mathbf{d}_k - \mathbf{c}_{p(k)}) + \hat{\omega}^*\mathbf{R}_k\mathbf{c}_k) && (2.149)\\
&= \mathbf{R}^T_k(\mathbf{v}_{p(k)} + \omega^*_{p(k)}(\mathbf{d}_k - \mathbf{c}_{p(k)})) + \mathbf{R}^T_k(\omega^*_{p(k)} + \hat{\omega}^*)\mathbf{R}_k\mathbf{c}_k && (2.150)\\
&= \mathbf{R}^T_k(\mathbf{v}_{p(k)} + \omega_{p(k)} \times (\mathbf{d}_k - \mathbf{c}_{p(k)})) + \omega^l_k \times \mathbf{c}_k && (2.151)
\end{aligned}$$

The angular velocity can be calculated in a similar fashion:

$$\begin{aligned}
\omega^*_k &= \dot{\mathbf{R}}^0_k\mathbf{R}^{0^T}_k && (2.152)\\
&= \dot{(\mathbf{R}^0_{p(k)}\mathbf{R}_k)}\mathbf{R}^T_k\mathbf{R}^T_k{\mathbf{R}^0_{p(k)}}^T && (2.153)\\
&= \dot{\mathbf{R}}^0_{p(k)}\mathbf{R}_k\mathbf{R}^T_k{\mathbf{R}^0_{p(k)}}^T + \mathbf{R}^0_{p(k)}\dot{\mathbf{R}}_k\mathbf{R}^T_k{\mathbf{R}^0_{p(k)}}^T && (2.154)\\
&= \omega_{p(k)} + \mathbf{R}^0_{p(k)}\hat{\omega}^*{\mathbf{R}^0_{p(k)}}^T && (2.155)
\end{aligned}$$

This can be transformed into the local coordinate frame again:

$$\begin{aligned}
{\omega_k^l}^* &= ({\mathbf{R}_k^0}^T \omega_k)^* && (2.156)\\
&= {\mathbf{R}_k^0}^T (\omega_k)^* \mathbf{R}_k^0 && (2.157)\\
&= \mathbf{R}_k^T ({\mathbf{R}_{p(k)}^0}^T \omega_{p(k)}^* \mathbf{R}_{p(k)}^0 + \hat{\omega}^*) \mathbf{R}_k && (2.158)\\
&= \mathbf{R}_k^T (\omega_{p(k)}^* + \hat{\omega}^*) \mathbf{R}_k && (2.159)
\end{aligned}$$

which gives

$$\omega_k^l = \mathbf{R}_k^T (\omega_{p(k)} + \hat{\omega}) \tag{2.160}$$

The next step is to calculate the linear and angular acceleration for each link. It is important to note that $\dot{\mathbf{v}}_k^l \neq (\dot{\mathbf{v}}_k)^l$ as the former does not take into account the Coriolis forces caused by the moving reference frame.

$$\begin{aligned}
(\dot{\mathbf{v}}_k)^l =& {\mathbf{R}_k^0}^T \dot{\mathbf{R}}_{p(k)}^0 (\mathbf{v}_{p(k)}^l + \omega_{p(k)}^l \times (\mathbf{d}_k - \mathbf{c}_{p(k)}))+ && (2.161)\\
& \mathbf{R}_k^T (\dot{\mathbf{v}}_{p(k)}^l + \dot{\omega}_{p(k)}^l \times (\mathbf{d}_k - \mathbf{c}_{p(k)}))+ && (2.162)\\
& {\mathbf{R}_k^0}^T \dot{\mathbf{R}}_k^0 (\omega_k^l \times \mathbf{c}_k) + \dot{\omega}_k^l \times \mathbf{c}_k && (2.163)\\
=& \mathbf{R}_k^T (\dot{\mathbf{v}}_{p(k)}^l + \omega_{p(k)}^l \times \mathbf{v}_{p(k)}^l + \dot{\omega}_{p(k)}^l \times (\mathbf{d}_k - \mathbf{c}_{p(k)})+ && (2.164)\\
& \omega_{p(k)}^l \times (\omega_{p(k)}^l \times (\mathbf{d}_k - \mathbf{c}_{p(k)})))+ && (2.165)\\
& \omega_k^l \times (\omega_k^l \times \mathbf{c}_k) + \dot{\omega}_k^l \times \mathbf{c}_k && (2.166)\\
=& \mathbf{R}_k^T ((\dot{\mathbf{v}}_{p(k)})^l + \dot{\omega}_{p(k)}^l \times (\mathbf{d}_k - \mathbf{c}_{p(k)}) + \omega_{p(k)}^l \times (\omega_{p(k)}^l \times (\mathbf{d}_k - \mathbf{c}_{p(k)})))+ && (2.167)\\
& \omega_{p(k)}^l \times (\omega_{p(k)}^l \times \mathbf{c}_k) + \dot{\omega}_{p(k)}^l \times \mathbf{c}_k && (2.168)
\end{aligned}$$

The angular acceleration $(\dot{\omega}_k)^l$ on the other hand is identical to $\dot{\omega}_k^l$.

$$(\dot{\omega}_k)^l = \mathbf{R}_k^T ((\dot{\omega}_{p(k)})^l + \dot{\hat{\omega}}_k + \omega_{p(k)}^l \times \hat{\omega}_k) \tag{2.169}$$

Force and Torque

This pass traverses the tree from the leaves of the tree to the root node and uses the previously calculated values to solve for $\mathbf{f}_i$ and τ_i in Eqs. 2.141 and 2.142. Since the leaf nodes have no children the force and torque at those links is zero. Similarly they are also zero for the root link.

Gravity can easily be modeled by setting the linear acceleration of the root link to $-\mathbf{g}$ which is equivalent to a fictitious force $-m_k\mathbf{g}$ at each link.

CHAPTER 3

Direct Control

One of the most important aspects of controlling a system is to find a way to express the goals and constraints in a usable manner. This is often done in the form of matrices, representing equality and inequality constraints.

$$\mathbf{A}\mathbf{u} = \mathbf{b} \quad \text{and} \quad \mathbf{C}\mathbf{u} \leq \mathbf{d} \tag{3.1}$$

where u denotes the joint commands necessary to control the system. One such joint command that lends itself well to that task is the joint acceleration, which can then be used to calculate the required torque for motors for example.

3.1 Constraints

3.1.1 Position and Orientation of links

In many cases a system should be controlled such that a link reaches some specified position (x_d) and orientation (R_d). In order to achieve that, these goals need to be expressed in terms of the joint accelerations. Since the links have a linear relationship with the control commands we can express the linear and angular accelerations ($\ddot{X} \in \mathcal{R}^6$) of a link as

$$\ddot{\mathbf{X}} = \frac{\partial \ddot{\mathbf{X}}}{\partial \ddot{\mathbf{q}}_a} \ddot{\mathbf{q}}_a + \ddot{\mathbf{X}}_0 \tag{3.2}$$

where $\frac{\partial \ddot{\mathbf{X}}}{\partial \ddot{\mathbf{q}}_a}$ is the derivative of the link acceleration with respect to the joint command and $\ddot{\mathbf{X}}_0$ is the link acceleration when $\ddot{\mathbf{q}}_a = 0$. This is used to build an equality constraint

$$\frac{\partial \ddot{\mathbf{X}}}{\partial \ddot{\mathbf{q}}_a} \ddot{\mathbf{q}}_a = \ddot{\mathbf{X}}_d - \ddot{\mathbf{X}}_0 \tag{3.3}$$

with the desired link acceleration $\ddot{\mathbf{X}}_d = (\dot{\omega}_d, \ddot{\mathbf{x}}_d)$. The linear and angular parts can be calculated as

$$\ddot{\mathbf{x}}_d = k_p(\mathbf{x}_d - \mathbf{x}) - k_v\dot{\mathbf{x}} \tag{3.4}$$

$$\dot{\omega}_d = k'_p R \log(R^{-1}R_d) - k'_v\omega \tag{3.5}$$

where log() is the matrix logarithm, $\mathbf{x}_d$ and R_d the desired position and orientation and $\mathbf{x}$ and R the current position and rotation of the link. The factors k_p, k_v, k'_p, k'_v can be used to adjust how aggressively the link should move toward the desired state.

The matrix logarithm of an orthogonal $3x3$ matrix is a 3-element vector and can be calculated using its skew-symmetric part [Eng01]. $\mathbf{R}$ and $\mathbf{R_d}$ in Eq. 3.5 refer to rotation matrices, so R^{-1} can be replaced with R^T. However, rotation matrices are not the only way to represent rotations in 3D space. The theory of Lie groups provides a unified way to deal with such rotations in the form of the special orthogonal group in 3 dimensions ($SO(3)$). This allows us to treat 3D rotations as black boxes and the logarithm returns the corresponding element of the Lie algebra ($so(3)$, a 3-vector). A library implementing operations in this group may internally represent the rotation using matrices in which case Eq. 3.5 becomes

$$\dot{\omega}_d = k'_p R \log(R^T R_d) - k'_v\omega \tag{3.6}$$

Alternatively if the rotation is internally represented using a quaternion then the multiplication of $R * x$ is defined as

$$\mathbf{R} * \mathbf{x_q} * \mathbf{R^{-1}} \tag{3.7}$$

where x_q is a quaternion containing the elements of x and 0 as its real part.

3.1.2 Joints

Joint constraints can be set up in a similar fashion. Depending on what the goal is the equations change slightly. If the joint should reach a desired position then the constraint can be expressed as:

$$\ddot{q}^i_a = k_p(q^i_d - q^i) - k_v\dot{q}^i \tag{3.8}$$

where the superscript i is used to indicate the index of the joint coordinate that needs to be controlled. If the goal is a specified velocity then

$$\ddot{q}^i_a = k_v(\dot{q}^i_d - \dot{q}^i) \tag{3.9}$$

where k_p and k_v can be used to control the speed with which the desired state is reached.

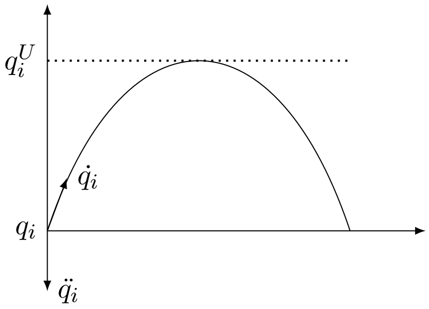


Figure 3.1: Joint position limit parabola

The joint position limits can be handled in a graceful manner (i.e. without introducing sudden impulses) by defining a region close to the limits where the allowed acceleration is reduced. This reduction is only activated when the joint is moving toward that limit.

$$\ddot{q}_i \leq \ddot{q}_i^U \quad \text{if} \quad q_i \in [q_i^U - \Delta_i, q_i^U] \quad \text{and} \quad \dot{q}_i > 0 \tag{3.10}$$

$$\ddot{q}_i \geq \ddot{q}_i^L \quad \text{if} \quad q_i \in [q_i^L, q_i^L + \Delta_i] \quad \text{and} \quad \dot{q}_i < 0 \tag{3.11}$$

$$\tag{3.12}$$

This can be done by setting up a parabolic arc that passes through the current position q_i with the current velocity $\dot{q}_i$ and has its maximum (or minimum) at the joint limit (see 3.1). The maximum value of this parabola can be calculated as

$$|q_i^U - q_i| = \frac{\dot{q}_i^2 sin^2(\theta)}{2\ddot{q}_i} \tag{3.13}$$

and since the angle θ is 90° the maximum allowed acceleration is given by

$$\ddot{q}_i^U = -\frac{\dot{q}_i^2}{2|q_i^U - q_i|} \tag{3.14}$$

Torque limits for joints can similarly be written as:

$$\frac{\partial \tau_u}{\partial \ddot{q}_a}\ddot{q}_a \leq \tau_a^U - \tau_{a,0} \tag{3.15}$$

$$-\frac{\partial \tau_u}{\partial \ddot{q}_a}\ddot{q}_a \leq \tau_{a,0} - \tau_a^L \tag{3.16}$$

where $\tau_{a,0}$ denotes the torque at the joint when there is no acceleration. The partial derivatives are calculated as part of the hybrid dynamics algorithm [KP11].

3.1.3 Constraint Solving

In many cases the system is under-determined, which enables us to optimize the control input for some goal and in other cases there is no (exact) solution but we still want to attempt to get as close as possible. In the previous sections we formulated the constraints in terms of the accelerations of the articulated joints. Consequently trying to minimize these accelerations is a good choice.

An important part of picking a solution is deciding which constraints must not be violated and which ones are less important. We can, for example, have a robot that cannot be represented as a tree structure (which is required for recursively solving the dynamics). In such a case a virtual cut in the structure is made and then treated like a robot that can be represented by a tree structure. Since the robot cannot physically be cut the solver must respect that. This is usually done by introducing an equality constraint that guarantees that the point where the robot was cut in the global frame is identical for both ends of the loop. That is if the subscript l denotes one side of the cut and r the other then $\mathbf{x}_l - \mathbf{x}_r = 0$ and $\mathbf{R}_l - \mathbf{R}_r = 0$, where $\mathbf{x}$ and $\mathbf{R}$ are the position and rotation in the global frame.

Goal positions can also be expressed in this manner. In that case $\mathbf{x}_r$ and $\mathbf{R}_r$ (or $\mathbf{x}_l$ and $\mathbf{R}_l$) are fixed in the global frame so $\mathbf{x}_l = x_G$ and $\mathbf{R}_l = R_G$. Often these constraints are not intended to be inviolable, but rather used to find a solution that moves an end effector as close to some target as possible. Naturally this presents a hierarchy of constraints and is taken into account by solving the system for the more important constraints first and then building a subspace in which the less important constraints are solved.

To illustrate we build three different sets of constraints:

$$\mathbf{A}_b\mathbf{x} = \mathbf{b}_b \tag{3.17}$$

$$\mathbf{A}_p\mathbf{x} = \mathbf{b}_p \tag{3.18}$$

$$\mathbf{A}_s\mathbf{x} = \mathbf{b}_s \tag{3.19}$$

$$\tag{3.20}$$

The constraints denoted with the subscript b are ones that must not be violated such as the loop constraints mentioned before. In this case the solution can be obtained directly by solving the linear system (using a Pseudoinverse to get a minimal solution in the least-square sense if necessary). Note that these equality constraints must not conflict with the inequality constraints in order to use common solvers. The solution $\mathbf{x}_0$ and the nullspace of $\mathbf{A}_b$ can then be used to build a subspace that can be explored further

$$\mathbf{x} = \mathbf{x}_b + \mathbf{N}_b\mathbf{y} \tag{3.21}$$

The system can then be optimized for the primary (denoted by subscript p) constraints by using the above equation and then solving for y, i.e. by solving the new system

$$\mathbf{A}_p(\mathbf{x}_b + \mathbf{N}_b\mathbf{y}) = \mathbf{b}_p \quad \text{and} \quad \mathbf{C}(\mathbf{x}_b + \mathbf{N}_b\mathbf{y}) \leq \mathbf{d} \tag{3.22}$$

$$\mathbf{A}_p\mathbf{N}_b\mathbf{y} = \mathbf{b}_p - \mathbf{A}_p\mathbf{x}_b \quad \text{and} \quad \mathbf{C}\mathbf{N}_b\mathbf{y} \leq \mathbf{d} - \mathbf{C}\mathbf{x}_b \tag{3.23}$$

The final solution can be obtained by plugging the solution for y in Eq. 3.21.

After the base constraints are solved we can optimize for the primary constraints by building a Quadratic programming (QP) problem:

$$\min ||\mathbf{A}\ddot{\mathbf{q}}_{\mathbf{a}} - \mathbf{b}||^2 \quad \text{s.t.} \quad \mathbf{C}\ddot{\mathbf{q}}_{\mathbf{a}} \leq \mathbf{d} \tag{3.24}$$

Finally the secondary constraints are also formulated as a QP problem but on a subspace of the primary constraints (like the above subspace of the base constraints), which ensures that all secondary solutions are equally good solutions of the primary constraints.

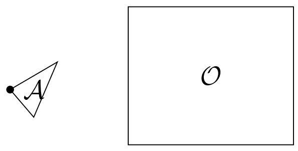
CHAPTER 4

Motion Planning

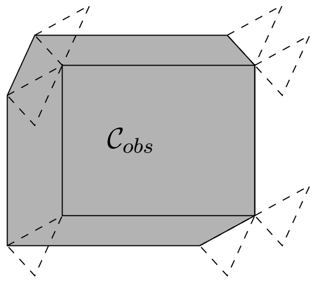
4.1 Motion Planning

The term Motion Planning is used to describe the automatic generation of robot motions from some higher level description. These generated motions generally need to satisfy different constraints, while aiming to be as efficient as possible with regards to criteria like speed, energy efficiency or safety.

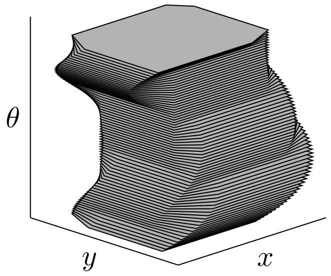
4.1.1 The configuration space



(a) An abstract robot $\mathcal{A}$ and obstacle $\mathcal{O}$. The dot marks the origin in the robot's frame



(b) The robot's configuration space without rotation



(c) Slices of the 3D configuration space if the robot can rotate

An important part of solving a problem is simplifying it or transforming it into another easier or better understood problem. This is no different for motion planning tasks. Consider the problem of an irregularly shaped robot navigating among arbitrary planar obstacles. While many may be aware of graph search algorithms such as A* it is not immediately obvious how those can be applied to this problem. Capturing collisions in the (familiar) Euclidean space is bothersome and potentially computationally expensive. Reducing the robot to a single point and changing the world this robot lives in to the *configuration space* simplifies the path planning considerably. Fig. 4.1a shows an abstract example of such a robot. The triangle $\mathcal{A}$ represents the robot and its origin is marked using a dot. The rectangle $\mathcal{O}$ represents an obstacle. For this first example the robot is not allowed to rotate. This obstacle is transformed into the configurations space by computing the Minkowski sum of the robot and the obstacle. The Minkowski sum of two sets of vectors A and B is given by

$$A + B = \mathbf{a} + \mathbf{b} | \mathbf{a} \in A, \mathbf{b} \in B \tag{4.1}$$

In non-mathematical terms this sum is created by sliding one object around the other and tracing the outline. The resulting obstacle in the configuration space, $\mathcal{C}_{obs}$, is shown in fig. 4.1b.

If one additionally allows the robot to rotate around its origin the configuration space gains an additional dimension (fig. 4.1c). This example also shows that while the obstacle is convex for a fixed θ this is not necessarily the case for the corresponding obstacle in the configuration space.

Many of the early motion planning approaches then try to capture the free space in its entirety, by representing it as graph and reducing the problem to finding the shortest path between the start and goal nodes (see. [Lat03]).

4.1.2 State space basics

Additionally robots operating in the real world come with certain physical limits such as motor torque or physical stops for rotating parts. Traditionally the nature of such constraints is used to categorize robots as holonomic or nonholonomic [Lat03]. Holonomic constraints are equality relations among the parameters that define the configuration of the robot. These equations can be solved for one parameter and (assuming minimal cardinality of the parameters) reduces the configuration space by one dimension for each such equation. An example of such a constraint is a revolute joint (or hinge). The configuration space of two freely moving planar objects has a dimension of 6 (x,y coordinates and angle of each object). If these objects are connected by a revolute joint, however, the dimension of the configuration space is reduced to 4 (x,y coordinates and angle of one object and the hinge angle). Nonholonomic constraints are equality relations that involve not only the parameters of the configuration space but their derivatives as well. These constraints do not reduce the dimensionality of the configuration space but rather limit the possible differential motions and are usually much harder to deal with. A car-like robot with the parameters (x, y, θ), where θ is the angle between the x-axis and the main axis of the car is an example of a system with nonholonomic constraints. At each instant the velocity $(\dot{x}, \dot{y})$ points along the main axis. The motion is thus constrained by

$-\dot{x}sin(\theta) + \dot{y}cos(\theta) = 0$ which is non-integrable and thus does not reduce the dimension of the configuration space (see [Lat03] for a more detailed description).

Historically this distinction between holonomic and nonholonomic robots was an important one as *complete* planners were developed first. One of the most important parts of these algorithms was to reduce the configuration space into as simple data structures as possible. This led to acceptable planning times, but these planners had a hard time dealing with differential motions and a such were mostly applied to quasi-static robots. The resulting motions were often slow and rough.

More recently with the increasing popularity of randomized planners (RRT, PRM) the importance of creating simple representations of the configuration space has diminished, while more effort was put into generating highly dynamic motions.

These movements are usually modeled as transition equations of the form $\ddot{q} = h(q, \dot{q}, u)$ with the generalized coordinates q (as well as $\dot{q} = \frac{dq}{dt}$ and $\ddot{q} = \frac{d\dot{q}}{dt}$) and the control input u. Since higher order differential equations are often difficult to handle a simple "trick" is employed to turn them into several first-order equations at the cost of introducing more variables. The simplest such system is the so-called *double-integrator*. The configuration space $\mathcal{C} = \mathbb{R}$ and the transition function is given by $\ddot{q} = h(q, \dot{q}, u) = u$. This corresponds to a Newtonian particle accelerated by a force u. This representation is then converted into the state space $\mathcal{X} = \mathbb{R}^2$ by setting $x_1 = q$ and $x_2 = \dot{q}$ with $(x_1, x_2) \in \mathcal{X}$. It is important to note that $\dot{x_1} = x_2$ and $\dot{x_2} = u$. In vector form this results in $x = (x_1, x_2) = (q, \dot{q})$ and $\dot{x} = (\dot{x_1}, \dot{x_2}) = (x_2, u)$. Finally the state transition function has been reduced to $\dot{x} = f(x, u)$ [LaV11].

One can see that moving from the configuration space requires twice the number of variables in the state space (the dimension of the planning problem is increased accordingly).

However this gives a unified way of dealing with a whole number of problems. Namely, holonomic, nonholonomic and kinodynamic problems can now be solved using the same algorithms.

Nonholonomic planning often arises from underactuated systems. This can happen if the dimension of the configuration space is greater than the number of action variables. *Kinodynamic* planning is used when the differential constraints on the robot are of second order. This can arise, for example, if drift needs to be modeled or the state of the robot changes, regardless of the control input [LaV11].

4.1.3 Configuration-Time Space

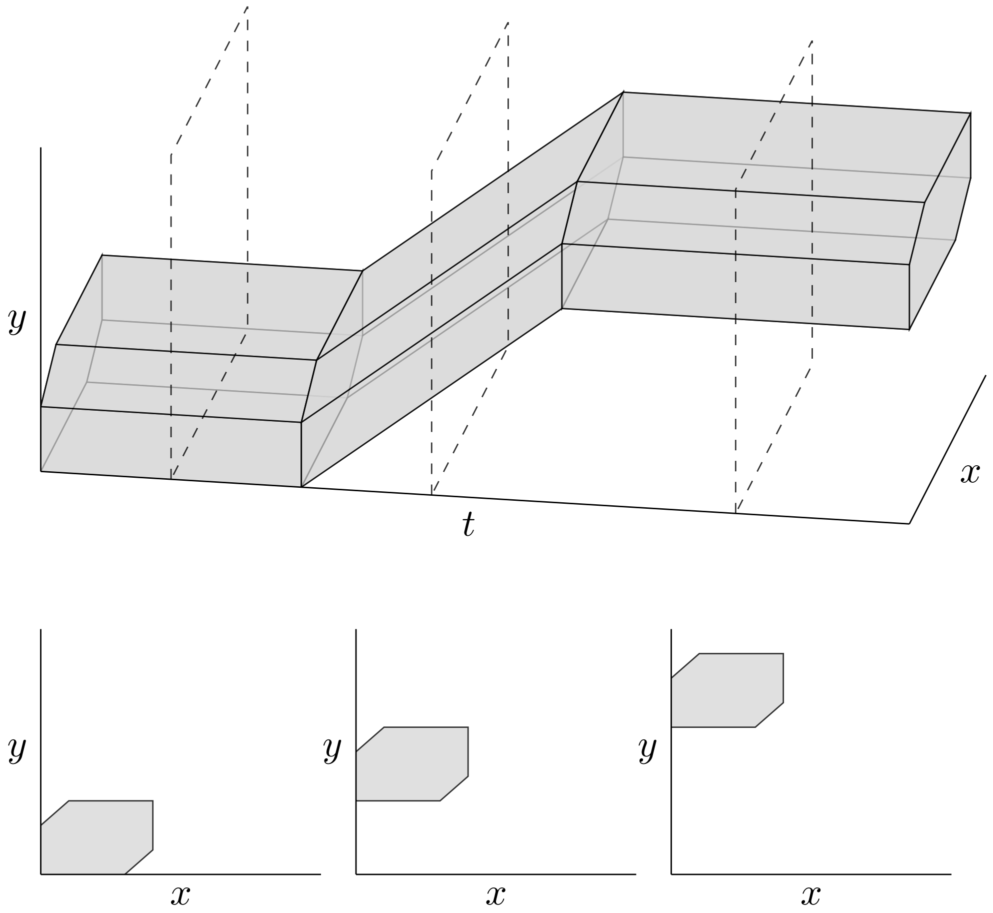


Figure 4.2: Configuration-time space

In many real world cases the obstacles are not always fixed. They can move or be moved over time which adds additional complexity. A straightforward way to extend planning to these problems is by introducing an additional dimension to the configuration or state space. This additional dimension, time, has one particular feature that makes it trickier to deal with - the robot can only move forward in time. The resulting space is usually referred to as *configuration-time space*. Sampling based and incremental planners are generally easily modified to deal with this space by only connecting nodes if the trajectory between them moves forward in time [LaV11]. Limits on the maximum velocity of the robot make these problems yet more challenging by constraining the slope of paths along the time axis.

4.1.4 Multi-Modal Planning

The configuration space of legged robots has additional features that more traditional robots, such as fixed base manipulators, do not. When such a robot touches the ground with its legs, the possible motions lie in a subspace of the configuration space. If such a contact is removed (i.e. by raising a leg) the dimension of the motion space is increased. Making another step reduces the dimensionality again by introducing more kinematic constraints. Motion planning for such robots requires the planner to be able to switch between a discrete sets of modes. Such systems combining discrete and continuous behavior are called hybrid systems and motion planning for such systems is usually referred to as multi-modal planning.

There have been several attempts at multi-modal motion planning for robots. Many of these approaches focus on developing planners for specific systems. Hybrid systems are required to switch between modes which each have their own constraints. A planner for such systems needs to be able to choose a discrete series of modes and traverse them in a continuous manner. In many cases hard problems in both, the continuous and discrete domains may need to be solved. A complete way to generate a collision free path through the configuration space is exponential in the number of degrees of freedom [Lat03]. In [Wil88] the author proves that navigation among moving 2D obstacles is NP-hard when the number of obstacles is not fixed.

Multi-modal planning has been investigated in the context of grasping and re-grasping operations [ALS94, FB97, NK00, SCS02, HNTHGB07], walking [CKNK03, Hau08] as well as re-configuring robots [CY99].

The difficulty in exploring the configuration space for motion planning tasks has spawned a popular class of motion planners - Probabilistic roadmap planners [KSLO96] as well as Rapidly-exploring Random Trees (RRT) [KL00]. They build a graph that approximates the connectivity of the configurations space by randomly sampling points in this space and connecting them with straight line paths. They are limited to a single mode but are still a very useful tool for multi-modal planners. Implementations of variations of these algorithms are available in the open source Open Motion Planning Library [SMK12].

One common way of applying them across modes is to find an intermediate state that is feasible in both modes and then attempting to find a path from the initial state (in the first mode) to the intermediate state (in both modes) and finally to the goal state (in the second mode). This approach has been successfully applied to manipulator planning [NK00] [SCS02]. A drawback of this approach is that the random nature of these algorithms means that they cannot answer that no path exists in finite time. In [Hau08] a multi-modal planner has been proposed that addresses some of those problems by exploiting the fact that the number of mode switches to achieve a goal is usually relatively small. Differential constraints and under-actuated poses still present difficult problems. More recently developed planners allow for more advanced motions like jumping [Shk10] [DS12].

Mode transitions

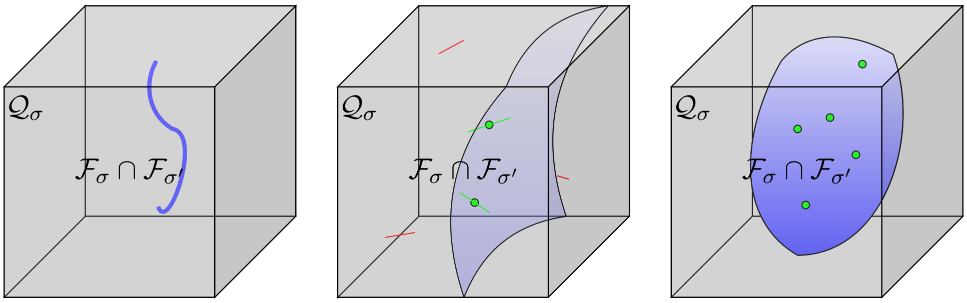

$\mathcal{Q}_\sigma$
$\mathcal{F}_\sigma \cap \mathcal{F}_{\sigma'}$
$\mathcal{Q}_\sigma$
$\mathcal{F}_\sigma \cap \mathcal{F}_{\sigma'}$
$\mathcal{Q}_\sigma$
$\mathcal{F}_\sigma \cap \mathcal{F}_{\sigma'}$

Figure 4.3: Three abstract examples that show different possible intersection regions between modes

An important aspect of multi-modal planners is their ability to plan paths across modes. This necessarily requires them to find states that are part of multiple modes to be considered for transitions. The generation of such states can be implicit or explicit [Hau08]. Implicit methods follow a generate-and-test approach whereby states are normally sampled and if they are part of the transition region a mode switch is executed. Explicit methods sample directly from the transition region and then generate paths to and from them using single mode planners. Consequently, mode switches are planned before sampling and then connected while implicit methods need to check if any given state can be used to connect multiple modes.

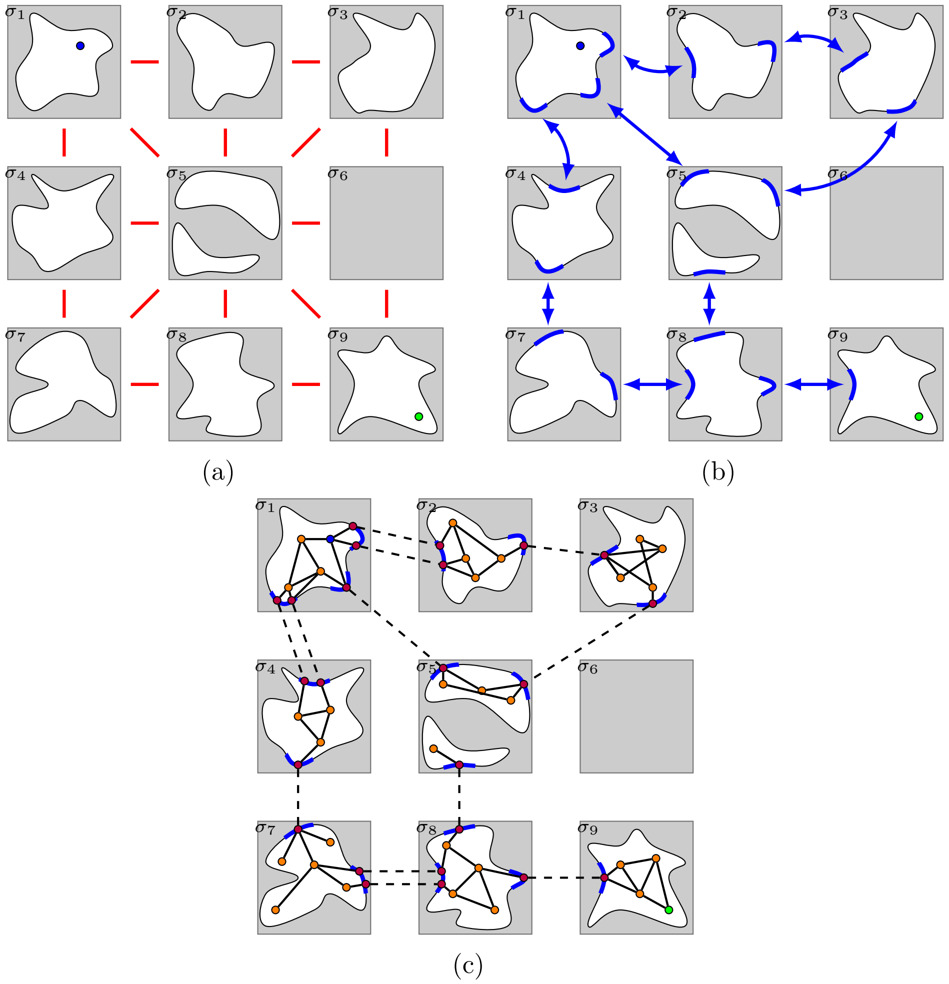
While implicit methods are often easier to implement they are not applicable in all situations. Fig. 4.3 shows an abstract example, where the configuration space $\mathcal{Q}$ is represented by a cube. In fig. 4.3 the dimension of the intersection (or the transition) region is less that $dim(\mathcal{Q}) - 1$ and it has zero volume in relation to $\mathcal{Q}$. It follows that the chance of sampling from this region is zero as well. Implicit methods cannot be used in this case. Similarly (fig. 4.3), if the dimension of $\mathcal{F}_\sigma \cap \mathcal{F}_{\sigma'}$ equals $dim(\mathcal{Q}) - 1$ the chance sampling from this region is zero. However a path between two states may intersect the transition region and implicit boundary-scan methods can be used. Lastly if $dim(\mathcal{F}_\sigma \cap \mathcal{F}_{\sigma'}) = dim(\mathcal{Q})$ then sampling from $\mathcal{Q}$ has a non-zero chance of also being in the intersection region and implicit methods can be used.

Some planners employ a mixture of implicit and explicit methods where some mode switches, such as making contact with a ground plane are explicit and others, like breaking contact are implicit [HNTHGB07] [eCG98].

If both, explicit and implicit methods, are viable then choosing an appropriate one depends on the shape and size of the intersection region as well as their implementation details. Highly constrained systems may find solutions more quickly by using explicit methods while systems with few constraints may be faster by using implicit methods [Hau08].

Especially in legged locomotion the number of constraints is usually large and like described above choosing suitable footfalls at random is often very unlikely. For this reason it makes sense in many instances to augment this procedure with additional methods, such as inverse kinematics (IK) solvers. Analytical methods are sometimes available for robots with few joints. The advantage is that the solution can be found quickly (if one exists) which enables the planner to try more possibilities in the same amount of time. The most flexible methods, however, find solutions numerically (e.g. using a Newton-Raphson iteration), but have the disadvantage of needing to find the pseudo-inverse of the Jacobian (see Ch. 3 and Sec. 5.1). The basic approach is to generate the loop closure conditions ($C(q) = 0$) for the contacts and solve the resulting system of equations. By using this method additional constraints can be included and solved for simultaneously. A common constraint for legged robots is that they remain statically stable and consequently inequalities that force the (projected) center of mass inside a support polygon are added. Analytical solutions often only consider solutions for a single leg, while numerical methods can easily adjust all joints to meet the requirements. It may, for example be possible to reach a certain footfall by shifting the robots body, which is generally not considered by analytical methods.

Basic operation



(a)

(b)

(c)

Figure 4.4: A basic multi-modal planning approach

The basic multi-modal motion planner (Multi-Modal-PRM) presented in [Hau08] builds Probabilistic Roadmaps across several modes and connects them via explicitly sampled states from the transition regions. Each mode has a corresponding roadmap. Initially the start and goal configurations are added to the appropriate modes. Then the following steps are repeated:

1. Sample a configuration for each mode, if it succeeds add it to the roadmap and connect it to existing waypoints.
2. For all pairs of adjacent modes sample a configuration from the transition region. Add it to the roadmaps for both modes if it succeeds.

If there exists a path from the start to the goal configurations the algorithm can be terminated.

Fig. 4.4 shows an abstract example to illustrate the general approach. The blue and green dots in fig. 4.4a are the start and goal configurations, respectively. Each square represents a mode and the white regions represent valid regions in the configuration space. The red lines between the squares indicate which modes are adjacent to each other. This does, however, not mean that a transition from one mode to an adjacent one must exist. Fig. 4.4b highlights the transition regions in blue and the blue arrows indicate which modes they connect. The orange dots in fig. 4.4c are sampled according to the first step above. Red points are configurations from the transition region (Step 2). The dashed lines indicate that connected points are not distinct configurations but the same sample in different modes.

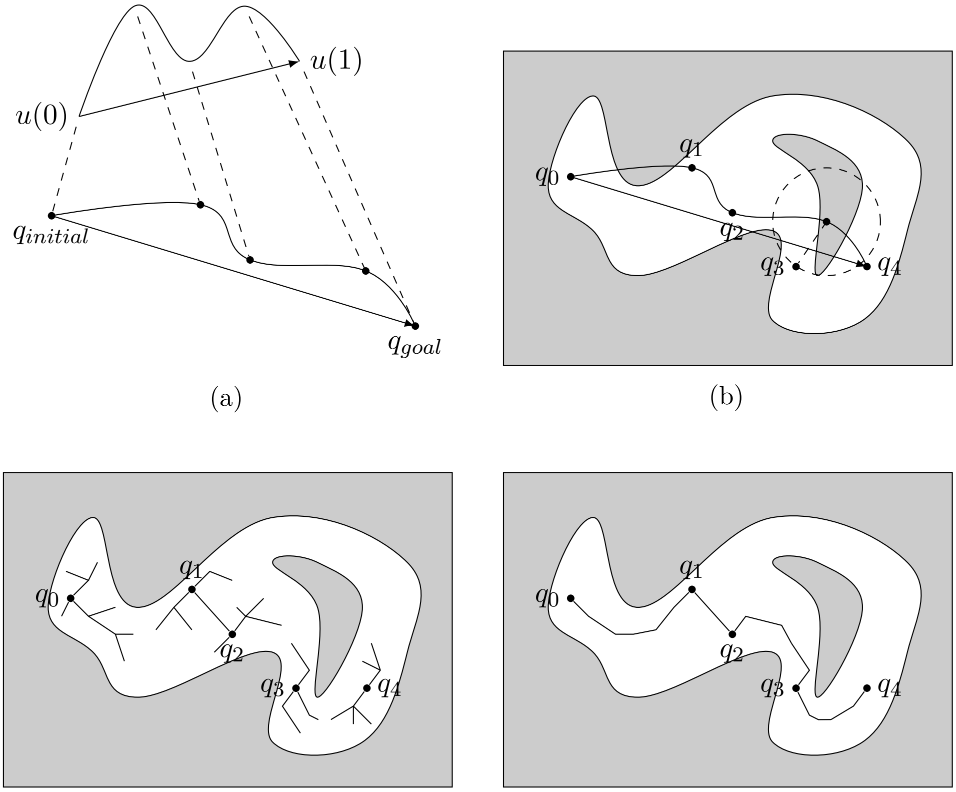
Finally the separate roadmaps built in each mode can be merged into a single aggregate roadmap.

Using Domain Knowledge

The above approach has the disadvantage that it may result in very unnatural-looking motions which could be distracting to humans operating alongside such robots. In order to mitigate this there have been several different suggestions. One way is to take the solutions generated by the above algorithm and apply some smoothing operations to blend the transitions. The authors of [GO04] reduce the jerkiness of the generated paths by picking two random points along adjacent path segments and attempting to connect them. Penalty based shortcutting techniques, whereby successful shortcuts are rewarded and failed shortcuts are penalized, have been found to converge faster than randomly picking points along the paths [Hau08].

Post-processing steps can only do so much, however. They cannot fix all unnatural motions generated by the underlying method. Other approaches don't use straight line segments to connect states and instead use fixed-final-state-free-final-time controllers to find optimal trajectories between states [WvdB13]. In this case the jerkiness of the motion is entirely dependent on the number of sampled states.

There has also been some effort and success in the film and video game industry to generate natural-looking motion from predefined motion primitives [RCB98], [WP95]. The goal of these methods is less to create physically accurate (or even possible) motions but rather to create physically plausible (or good looking) motions.



(a) (b)

(c) (d)

Figure 4.5: Path warping using motion primitives

Similar work has been done in [HBHL08] where the authors use a library of predefined motion primitives to generate feasible paths for the HRP-2 robot. They split the motion primitives into several parts and transform it (see 4.5) so the initial stance of the robot coincides with the one of the motion primitive and the goal stance matches the final stance of the primitive. These milestones (where the motion primitive has been split) are then used as root nodes for a sampling based planner. The algorithm terminates when a path connecting the initial and final configurations has been found. The motion primitives can easily be included in previously described multi-modal planner by assigning them to the appropriate mode. This naive approach may waste unnecessary processing time by greatly increasing the number of modes and consequently spending much time on infeasible primitives and so care should be taken when using it.

4.1.5 Single Mode Planning

In order for the previously presented planner to work well it is important to pick a good single mode planner. This sections aims to provide an overview of several approaches and highlight their strengths and weaknesses.

Over the years there have been several attempts at finding solutions to this problem. The most popular approaches can be categorized as

- Roadmaps
- Rapidly Exploring Random Trees (RRT)
- Cell Decomposition
- Potential Field
- Reward Based

Roadmaps

These approaches attempt to capture the connectivity of the free configurations space as a network of simple curves, also called the *roadmap*. This roadmap is often generated in advance. The execution stage only needs to connect the initial and goal configurations to this roadmap and then a use graph search algorithm (e.g. A*) to find a suitable path. Examples of such an approach include the visibility graph, freeway nets, retractions methods (in particular Voronoi diagrams in 2D) and Probabilistic roadmaps. An advantage of roadmaps is their reusability. Once a connectivity graph between a decent number of states has been established the roadmap can be reused for differing start and goal configurations. Such *multi-query* planners are popular for online planning since the heavy computation can be done offline (provided the environment is known). Generating a good roadmap can be difficult as it may end up turning into a high-dimensional grid which reduces its usefulness. In order to deal with such problems good pruning strategies are extremely valuable. The authors of [SLN00] for example split the configuration space into visibility domains, which greatly reduces the size of the generated roadmap.

Rapidly Exploring Random Trees

As the name suggests RRTs build a tree structure from the initial configuration toward the goal by randomly sampling the configuration space and attempting to connect the new states with existing ones in the tree until the goal configuration can be reached from an existing node in the tree. The sampling can be biased toward unsearched areas of the configurations space (hence the Rapidly Exploring). They are well suited for high dimensional problems with nonholonomic or kinodynamic constraints [LaV98]. One disadvantage of this approach is that, unlike roadmaps, the tree needs to be rebuilt if the initial (and to a lesser extent the goal) configuration changes. Such methods fall under the so called *single-query* category which makes them less attractive for online planning. There have been attempts to alleviate this by rewiring the paths inside the tree if the initial state changes [NRH15]. The very simple concept and high versatility of this approach has caused many researches to develop RRT variations with varying advantages (including provable asymptotic optimality [KF11]) and disadvantages. This approach has been successfully applied to autonomous driving in urban environments [KTF+09].

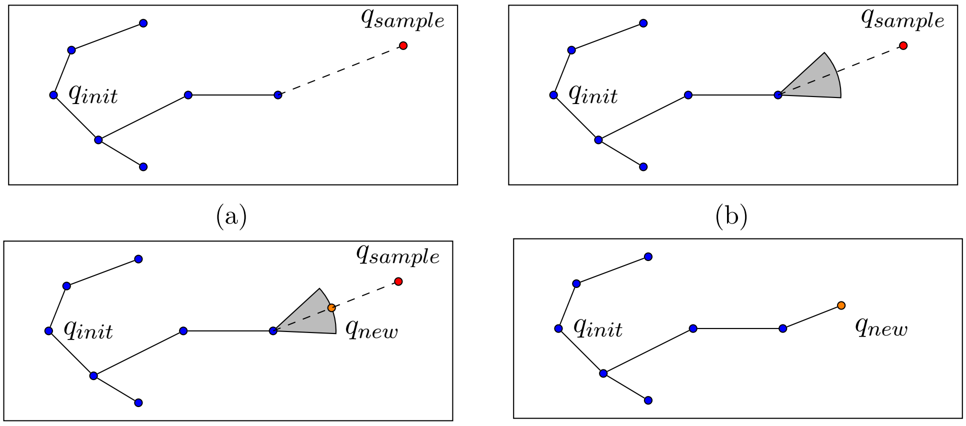
The basic RRT approach is as follows:

1. Initialize the tree with the starting configuration q_{init}
2. Sample a random configuration q_{rand}
3. Discard if it is not in the free space (C_{free})
4. Find the nearest neighbor in the tree according to some metric
5. Use a local planner to attempt to connect the nearest node in the tree to the new node
6. Repeat from step 2 until the goal configuration can be connected to a node in the tree

There are several important details that need to be considered. Initially new configurations were sampled using a uniform distribution [LaV98]. This approach has the disadvantage of favoring large empty regions of the configuration space. Connecting states in these regions is often straightforward and so unnecessary time may be wasted here. Other implementations force sampling near obstacles [AW96] or use adaptive Gaussian sampling based on obstacle and collision data [BOvdS99]. The authors of [SLN00] sample the state space based on visibility regions which drastically reduces the number of nodes in the graph.

Next, picking an appropriate metric for the nearest neighbor search is crucial. The goal here is to quickly determine the *cost* of moving from one state to another. Common cost metrics are the time required to change from one state to the other or the mechanical energy required. This metric is calculated frequently so ideally it should be fast to determine. A theoretical overview of several metrics is given in [WVDBH08]. Alternatively or in conjunction different expansion strategies can be chosen to reduce the dependence on metrics. *Expansive Space Trees* (EST) and *Guided Expansive Space Trees* (GEST) [PBK04] select which nodes to expand based on their neighborhood. *Path Directed Subdivision Trees* (PDST) [LK04] and the more recent *Kinodynamic Motion Planning by Interior-Exterior Cell Exploration* algorithm (KPIECE) [ŞK12] choose the expanding node based on its coverage in order to reduce the time spent in well-explored areas.

Since a large part of the total CPU time is spent checking for collisions [SLN00] there have been several approaches to alleviate this problem. Some planners use the collision checker to guide the search ([PBK04] [CL01]), adapt the sampling strategy ([KM12]) or improve the connectivity of the resulting graph ([SLN00]) to achieve this. Others ([NK00], [BK00]) delay the collision checking until a potential path has been constructed (Lazy collision checking).



(a) (b)

(c) (d)

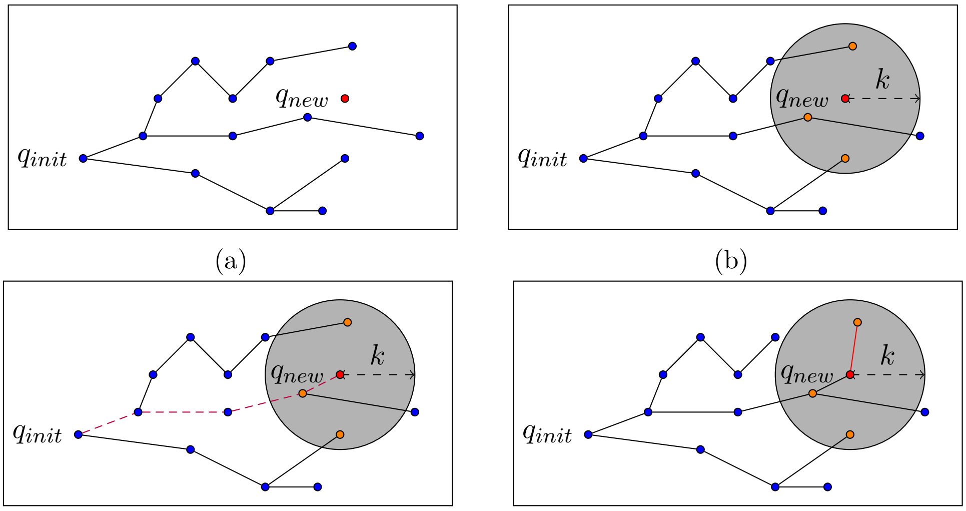
Figure 4.6: Expansion using a reachability criterion

Connecting two states is another area that is highly dependent on the problem. For quasi-static robots it is often sufficient to use straight line segments to connect states (in this case the Euclidean distance can be used as a metric). This is often not enough for nonholonomic and kinodynamic robots (shifting a car a few centimeters to the left or right may require a number of control inputs to achieve if it is at all possible). In these cases it is often useful to change the expansion of the graph slightly. The new (sampled) configuration is not immediately discarded if it is unreachable. Instead the set of reachable states is calculated for each node in the tree. If a reachable state is closer to the sample it is added to the graph. If an existing node is closest, the sample is discarded [SWT09]. While the set of reachable states can be hard to calculate explicitly it is often sufficient to find its boundary by forward integrating the system at its action limits. Fig. 4.6 illustrates the basic approach. First a new configuration is sampled (red dot in fig. 4.6a), then the set of reachable states (gray circular sector in fig. 4.6b) is used to find a state closer to the goal. Finally this state is added to the tree instead of the original sample (figs. 4.6c and 4.6d).

Another modification of the basic RRT and PRM algorithms was presented in [KF11]. These variations of *Rapidly Exploring Random Graphs* (RRG) called *RRT** and *PRM** guarantee asymptotic optimality. The nearest neighbor search is adapted to return all nodes within a certain radius k of the new sample. This radius is calculated as

$$k = \gamma\left(\frac{\log n}{n}\right)^{\frac{1}{d}} \tag{4.2}$$

where d is the dimension of the $\mathcal{C}$-Space, n the number of nodes in the graph and γ, a factor based on the environment characteristics. The new sample is then connected to the node that provides the lowest cost to the root. Lastly all nodes in the neighborhood are connected to the new node if it decreases their cost to the root. Fig. 4.7 gives a simple example. After sampling a new node (fig. 4.7a) all nodes inside the hypersphere



(a) (b) (c) (d)

Figure 4.7: RRT* algorithm

with radius k are found (orange dots in fig. 4.7b). Then the sample is connected to the node with the lowest cumulative path cost (fig. 4.7c) and finally the neighboring nodes are rewired if necessary (fig. 4.7d).

It is very important to note that this optimality is guaranteed in regards to the used metric, so defining a metric that expresses the true cost of connections is vital.

Single-Query Bi-Directional Probabilistic Roadmap Planner with Lazy Collision Checking (SBL)

SBL [SL03] builds a network of milestones between the start and goal and performs no collision checking unless absolutely necessary. During each step either the tree rooted at the start or goal nodes is chosen along with an existing milestone in that tree. This milestone is then used to generate a (valid) nearby state which is inserted into the appropriate tree. Following that an attempt to connect the two trees is made and if it succeeds a path from the start to the goal is returned. The main difference to a lazily evaluating bi-directional RRT planner is that an existing node in the tree is chosen first and then a new sample is generated nearby, while RRTs sample first and find the closest existing node later. Projections of the state space have been successfully used to guide the planner toward exploring new areas.

Search Tree with Resolution Independent Density Estimation (STRIDE)

STRIDE [GMK13] is a planner for high-dimensional problems that maintains a *Geometric Nearest Neighbor Access Tree* (GNAT) to estimate the sampling density in the configuration space. This facilitates rapid exploration of all dimensions of the state space.

Unlike SBL or KPIECE the density estimates are computed in the full dimensional space directly rather than on a lower dimensional projection of it.

Kinodynamic Motion Planning by Interior-Exterior Cell Exploration (KPIECE)

KPIECE [ŞK09] is a randomized planner that is suited to dealing with kinodynamic problems (i.e. problems involving the velocity and acceleration of a system, [DXCR93]). The resulting solution does not only provide a path in the state space but also the necessary control input to achieve that path. It builds and uses a (hierarchical) discretization of the state space to bias the search towards less explored regions. Boundary cells (cells that have less than $2n$ explored neighboring cells, with n being the dimension of the discretization) are given a higher priority when sampling. To further accelerate the exploration a projection of the state space into a lower dimensional one is often used for the discretization.

Cell Decomposition

Cell Decomposition methods divide the free space into cells, such that a path between any two states in these cells can be easily generated. Methods in this category can further be classified as *exact* or *approximate* cell decompositions, depending on whether the free space is covered by the decomposition in its entirety or not. Common methods are quadtrees (octees) or decompositions into convex objects. An advantage of approximate decompositions is that in many cases the search time can easily be traded for accuracy. Less accurate decompositions provide fast results while more accurate ones can find very difficult paths until an exact decomposition is reached at which point the planner is guaranteed to find a free path if one exists [Lat03].

Potential Field

This method is heavily inspired by physical phenomena. The robot is regarded as a particle in the configuration space wherein obstacles induce a repulsive force on the robot and the goal region an attractive one. The motion planning is then reduced to a fastest descent search. Great care needs to be taken that the robot does not get stuck in local minima, however.

Reward Based

Reward Based methods assume that the robot can take one of several actions at each state. The result of such an action is, however, not definite. This means that even if the robot chooses an action that results in state A it might end up in state B by chance (or due to the uncertainty of its motion). Similarly to potential field methods some states attract the robot (rewards) while others repulse it (punishments). Markov Decision Processes provide a nice mathematical framework for these approaches and are often used to find optimal solutions. A disadvantage of this approach is that there is only a discrete set of actions, which may unnecessarily limit the free space.

CHAPTER 5

Implementation Details

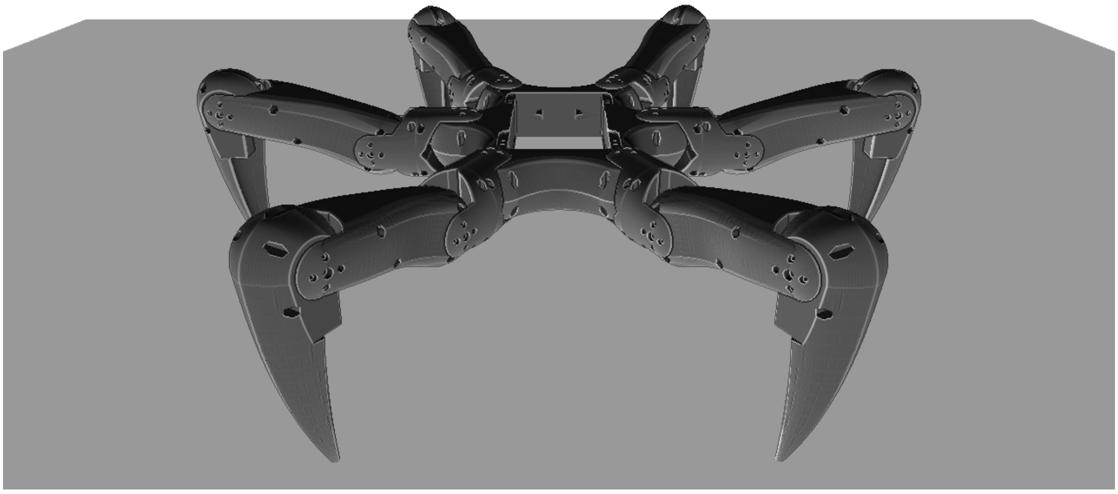


Figure 5.1

5.1 Inverse Kinematics (IK)

The planning algorithm in Sec. 4.1.4 first explores a stance graph where each stance is defined by its contact constraints. In order to generate motions that satisfy these constraints the joint positions of the robot need to be adjusted accordingly. However, these constraints lie in a sub manifold of the configuration space ($\mathcal{Q}$) that has a lower dimension than $\mathcal{Q}$. This means that a purely sampling based approach is not likely to generate joint positions that satisfy the contact constraints (See Fig. 4.3).

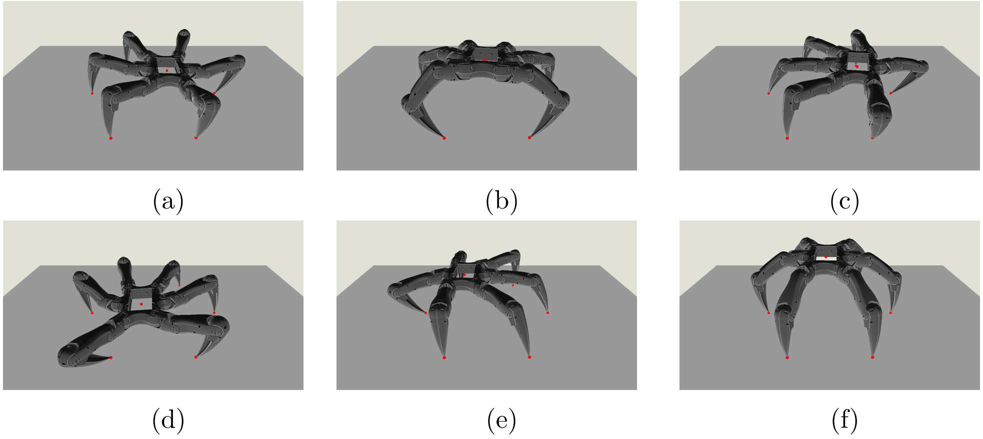

(a) (b) (c) (d) (e) (f)

Figure 5.2: IK solutions for random poses of the root link

The problem of finding the joint positions based on the target poses of the end-effectors is called *Inverse Kinematics*. Over the years several techniques to solve this problem have been developed. Some of the most common ones are [AL09]:

- Jacobian-based: These numerical methods iteratively compute the joint positions (θ) in this manner: $\theta_{i+1} = \theta_i + \mathbf{\Delta}\theta$. The change in the end-effector poses can be estimated as $\mathbf{\Delta s} \approx J\mathbf{\Delta}\theta$, with the Jacobian J. Setting $\mathbf{\Delta s}$ to the error term ($\mathbf{e}$) gives us $\mathbf{\Delta}\theta = J^{-1}\mathbf{e}$. However, the Jacobian is not necessarily square or even invertible and even if it is it may work poorly in near singular configurations. Alternatives to inverting J directly include:
  - The Pseudoinverse (or Moore-Penrose Pseudoinverse):

    The Pseudoinverse minimizes the least square error of $J\mathbf{\Delta}\theta = \mathbf{e}$. Unlike the real inverse of the Jacobian this method is guaranteed to find a solution, but it also performs poorly in near singular configurations.
  - The Jacobian Transpose:

    This method calculates the change in joint positions as $\mathbf{\Delta}\theta = \alpha J^T\mathbf{e}$ for some scalar α. Calculating the transpose is obviously very different from calculating the inverse, however this method can be justified using virtual forces [WE84]. In practice it is much quicker to calculate than the others in this category but frequently suffers from convergence issues (e.g. oscillation).
  - Dampened Least Squares (DLS):

    One way to calculate the Pseudoinverse of the Jacobian is with the help of its Singular Value Decomposition (SVD) $J = \mathbf{U\Sigma V}^T$ with the unitary matrices $\mathbf{U}$ and $\mathbf{V}$ as well as the diagonal matrix $\mathbf{\Sigma}$ consisting of the eigenvalues of J. The Pseudoinverse can then be calculated as $\mathbf{V\Sigma}^\dagger\mathbf{U}^T$. $\mathbf{\Sigma}^\dagger$ is the diagonal matrix that is generated from $\mathbf{\Sigma}$ by replacing its elements with their inverse, i.e.:

$$\sigma_i^\dagger = \begin{cases} \frac{1}{\sigma_i} & \sigma_i \neq 0 \\ 0 & \sigma_i = 0 \end{cases} \tag{5.1}$$

The dampened least squares method (or singularity-robust inverse [NH86]) changes $\sigma_i^\dagger$ to

$$\sigma_i^\dagger = \frac{\sigma_i}{\sigma_i^2 + \kappa} \tag{5.2}$$

with an appropriately chosen κ. This solution minimizes the equation $\|J\boldsymbol{\Delta}\theta - \mathbf{e}\|^2 + \kappa\|\boldsymbol{\Delta}\theta\|^2$. The result is that the changes in joint positions in near singular configurations are reduced. Picking an appropriate value for κ can be cumbersome and strongly depends on the system.

– Selectively Dampened Least Squares [BK04]:

Similarly to DLS the goal is to avoid large changes in the joint positions due to singularities. This is achieved by only dampening the effects of singular values that induce canceling motions (e.g. the change in one joint moves the end effector in one direction and a change in another joint moves it in the opposite direction). Compared to DLS it converges faster and there are no magic numbers to tweak [BK04].

All of the above methods (save for the Jacobian Transpose) are costly to compute, but provide a nice way to encode additional constraints and priorities (by making use of the null space of J).

- Cyclic Coordinate Descent (CCD, [WC91])

  CCD is a fast, commonly used, heuristic that is often found in games, animations and has even been employed in protein folding applications [CD03]. Starting from the end-effector and moving toward the base it fixes all joint positions except for the one it is currently operating on and attempts to reduce the error using that joint only. Depending on the problem it may generate many different solutions and choosing a good one is not always easy. Local constraints (e.g. joint limits) can easily be incorporated in this method while global constraints often pose difficulties. Additionally this method operates on a single chain only. The authors of [MD04] extend this method to more complex systems by finding sub-chains and solving them separately.

- Analytical Methods

  Analytical solutions to the IK problem are exclusive to specific robot configurations and often come with certain restrictions, such as only considering parts of the robot (e.g. a single leg). They do, however, provide extremely fast solutions (if they can be found).

For this implementation the DLS method was chosen as it a very general approach and additional constraints can easily be incorporated. The SDLS method was also implemented, but not used as it seemed to suffer from poor convergence issues (possibly due to bugs). Neither of those approaches are particularly fast, so first using a fast solver (such as CCD) followed by a refinement step using a Jacobian based method was considered. Ultimately this was not implemented due to the fact that in the context of the multi-modal planner nearby valid configurations, that can be used as seeds for DLS, are readily available.

5.2 Collision Detection and Response

An important part of any planning algorithm is a procedure to decide whether a particular state of the robot is allowed or not. One aspect of such a procedure is often a check if the robot's geometry intersects with the environment (or itself) in an undesired way. Most of the approaches mentioned in section 4.1.5 require many samples until a path from the start to the goal is found so it is important to be able to quickly decide if collisions are present. For this implementation LibCCD's (available at `https://github.com/danfis/libccd`) Minkowski Portal Refinement (MPR) algorithm was used. It is based on the Gilbert-Johnson-Keerthi distance algorithm (GJK), which uses the Minkowski-sum to iteratively build intersection simplices near the colliding vertices.

An advantage of this approach is that it only requires a *support*-function that takes a vector and returns the farthest vertex of the object in that direction. This allows for specialized implementations of certain objects without having to turn them into meshes first. Additionally, model translations and rotations do not require the object itself to be transformed; instead the input direction of the support function is rotated using the inverse of the model rotation first and the output is transformed using the full model transformation (including translation) later. For example, listing 5.1 shows the support function for a cylinder (with its center at the origin and its flat sides in the xy-plane) whose transformation from the local to the global coordinate frame is given by $\mathcal{T}$.

For general convex meshes a hill-climbing approach can be used. Due to the low number of vertices in the collision meshes used here this approach was, however, slower than simply checking every vertex.

The GJK algorithm (and thus MPR) requires objects to be convex, so meshes from CAD programs usually need to be pre-processed. The simplest form of preprocessing could be done by calculating the convex hull of the mesh and using that as the input for the collision detection algorithm. This potentially introduces many false positives for concave objects and was not used here. Instead an approximate convex decomposition was applied (`https://github.com/kmammou/v-hacd`) to generate several "near" convex meshes. The meshes are preprocessed once during the loading process but if necessary this could be done in advance as well. For the used robot model each of the seven collision meshes has between 150000 and 700000 vertices and the decomposition takes less than ten seconds in total on a mid- to low-end desktop graphics card (using the OpenCL back end of the library). The decimated meshes contain less than 100 vertices

Algorithm 5.1: Support function for a cylinder

```
Function sign(v)
    if $v < 0$ then
        return -1
    else if $v > 0$ then
        return 1
    else
        return 0
    end
Function support(direction)
    d ← $\mathrm{Rot}(\mathcal{T})^T *$ direction
    z-dist ← $\sqrt{d.x^2 + d.y^2}$
    if $z\text{-}dist < \epsilon$ then
        return $\mathcal{T} * [0, 0, \frac{\mathrm{sign}(d.z)*length}{2}, 1]^T$
    else
        return $\mathcal{T} * [\frac{radius*dx}{\text{z-dist}}, \frac{radius*dy}{\text{z-dist}}, \frac{\mathrm{sign}(d.z)*length}{2}, 1]^T$
    end
```

(per convex object), which indicates that the original ones may be excessively detailed. Nonetheless, it is more convenient for the user to not have to worry about potential vertex count limits.

5.2.1 Collision Response

Collisions can be incorporated into the control approach described in chapter 3 in several different ways.

One technique is to include them in the base constraints (eq. 3.20) so the primary and secondary solutions can not violate the contact constraints. The controller, however, can only affect powered joints, so the non-prescribed joints (e.g. the free 6-dof joint connecting the robot to the ground) need to be adjusted in a second step. During this second step the joint coordinates generated by the QP solver are fixed while the others can be changed (in many cases this is only the root joint).

Alternatively the root joint can be changed to one with fewer degrees of freedom, depending on the number of contacts and their locations. For example if the tip of one of the robot's legs touches the ground then the original root joint can be removed and a new 3-dof joint, connecting the ground to the robot at the contact point, added that only allows rotations and no translations. Similarly, if two legs touch the ground one of them can be connected to the ground using a hinge joint (1-dof) with the vector between the contact points as its axis. If there are three or more (non-collinear) contacts then the root joint becomes a 0-dof fixed joint. The other contact points still need to be kept which can be done by adding their loop closure equations to the QP problem. This method requires the robot kinematic tree to be rebuilt every time a contact is added or removed. Additionally if the root joint is only adjusted for the controller (i.e. the simulation uses the original 6-dof joint) then the integration step still needs to ensure that the collisions are dealt with.

Instead of the above approach the collisions are handled by applying forces at the contact points before the controller attempts to solve the system. This is done as in [Dru08] where the contact forces are calculated in a similar fashion to how PID controllers operate. The restorative force is calculated as

$$\|f(p)\| = \max(\frac{k_p d(p,t) - k_v v(p) + k_i I}{r}, 0) \tag{5.3}$$

where $d(p,t)$ is the penetration depth at the simulation step t for the contact point p, $v(p)$ the linear velocity at the contact, I the integral (sum) of the previous penetrations and r the number of contacts. The gains k_p,k_v and k_i control how quickly the error (penetration) is reduced. The integral I is calculated for each link separately as $\sum_{i=0}^{t-1} \lambda^{t-i-1} \max_y(d(y,i))$ with the exponential decay factor λ (set at 0.9). The amount of additional storage for the penetration history is reduced by removing old values of $d(y,i)$ if λ^{t-i-1} becomes less than 0.00001. k_p,k_v and k_i were set at 100mass_{robot}, 50mass_{robot} and 20mass_{robot} respectively. In the implementation the deepest point method is used (see [Dru08] for a comparison), however since MPR calculates the penetration normal and depth from a simplex this is essentially equivalent to the presented multiple point method.

5.2.2 Single Mode Planners

All of the used single-mode planners have certain features and parameters that need to be adjusted to work within the framework. The next sections aim to justify and explain the choices made during the implementation.

5.2.3 SBL

A projection matrix from the state space to the grid is chosen randomly (see [AK$^+$09] for more refined alternatives).

As SBL samples new states in a neighborhood of existing ones we make use of the local smoothness of the Jacobian (like [Hau08] and [YLK01]). Let the rank of the $m \times n$ Jacobian (J) be r. This allows us to choose $n - r$ independent random variables to displace the configuration while the remaining k parameters are used to satisfy $J\mathbf{q} = \mathbf{0}$. The SVD of J can be written as

$$J = \left[\begin{array}{ccc|ccc} u_1 & \cdots & u_k & u_{k+1} & \cdots & u_m \end{array}\right] \left[\begin{array}{ccc|c} \sigma_1 & & & \\ & \ddots & & 0 \\ & & \sigma_k & \\ \hline & 0 & & 0 \end{array}\right] \left[\begin{array}{c} v_1^T \\ \vdots \\ v_k^T \\ \hline v_{k+1}^T \\ \vdots \\ v_n^T \end{array}\right] \tag{5.4}$$

The last $n-k$ rows of V represent the null space of J and are used to calculate the random displacements of the seed state $\mathbf{q_s}$ as $\mathbf{q} = \mathbf{q_s} + \begin{bmatrix} v_{k+1} & \cdots & v_n \end{bmatrix} \text{rand}_{n-k\times 1}()$. After that the displaced seed state is repaired using using the IK-solver from section 5.1.

This new state (or milestone for the planner) is then added as a child of the existing one until the trees originating at the start and goal configurations can be connected. At this point the edges between milestones are checked for their validity. However, a state generated by simply interpolating between two milestones will most likely not satisfy the contact constraints and so it is transformed into a nearby state that does first. Listing 5.2 shows pseudo-code for a recursive implementation that checks if a path between two states is valid. The algorithm essentially bisects the straight line path from the initial to the final state and attempts to move the mid point onto the valid sub manifold. The drawback of this approach is that the midpoint is most likely the point farthest from being valid and consequently requires the most iterations to repair. This implementation was chosen in order to be comparable to [Hau08] and because it provides a nice way to specify the tolerance of the line segments.

Alternatively starting from q_0 and moving by a small δ each step could have been implemented but this approach would have the difficulty of choosing an appropriate δ. For RRT in particular this approach can be even more useful as the goal state is not necessarily valid and the path validity checker can simply return a state in the direction of q_1 (see fig. 4.6) [BS10]. However as only the KPIECE implementation can use this feature currently this has not been further considered.

Algorithm 5.2: Path validity checker used by the local planners

```
Function is_path_valid(stance, $q_0$, $q_1$)
    if distance($q_0$, $q_1$) $< \epsilon$ then
        return true
    $q_{\text{mid}} \leftarrow$ interpolate($q_0, q_1, 0.5$)
    /*apply Newton-Raphson iteration mentioned in sec.  5.1 */
    $q_{\text{mid}} \leftarrow$ transform $q_{\text{mid}}$ s.t. it satisfies the constraints in stance
    if $q_{mid}$ was successfully transformed then
        /*check if $q_{\text{mid}}$ is collision-free and satisfies all
          constraints                                             */
        if $q_{mid}$ is valid then
            return is_path_valid(stance,$q_0$,$q_{\text{mid}}$) and is_path_valid(stance,$q_{\text{mid}}$,$q_1$)
        else
            return false
        end
    else
        return false
    end
```

5.2.4 STRIDE

An advantage of the STRIDE implementation is that there are very few parameters to adjust so the default values were kept. Nonetheless a projection was supplied to make the results comparable. Like SBL this planner also samples near existing states and the same approach was used. The main difference is that the implementation immediately checks any potential paths connecting states for collisions (more on that in chapter 7).

5.2.5 RRT

Unlike SBL, RRT and its derivatives do not sample near existing states. This presents a problem as sampling randomly is unlikely to generate valid states. One possible solution is to attempt to repair the random states like the sampler used for SBL does, but without the advantage of having a nearby state to use as a seed. Another option is to allow for errors in the contact constraints. This means the volume of the sub manifold the constraints span is no longer zero and can consequently be sampled.

The first approach tended to turn the sampling process into a costly operation that hindered the progress of the planner more than it helped. The second led to some limited success (see chapter 7).

To check the validity of paths the same algorithm as in the previous section was used (5.2). Since the path validity checking is also a fairly costly operation (compared to sampling), a RRT version using lazy path checking was utilized ([BK00]).

5.2.6 KPIECE

Similarly to SBL this implementation uses a projection to reduce the number of dimensions for the discretization. To keep the results comparable a random projection matrix was chosen in this case as well. The discretization hierarchy has a single level and collision checking is performed lazily.

CHAPTER 6

Robot Platform

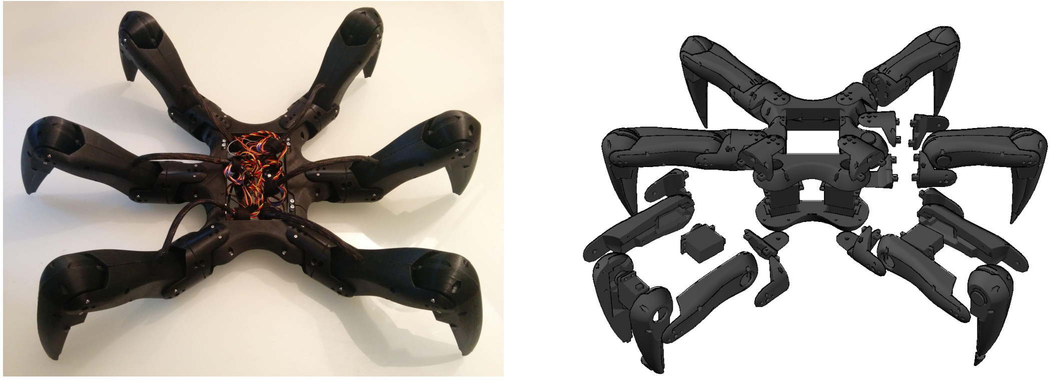
A six-legged robot was built with the intention of testing the results of the multi-modal planner on a physical platform and to verify the simulation results.

One of the main goals was to make it as easily manufacturable as possible. The design was done in a way that makes it easy to 3D print, without the need to use glue. The chassis can be printed using widely available printers that employ fused filament fabrication techniques. Not needing to glue parts together facilitates its expandability as many internal components can be changed without having to reprint parts. Besides a few short pieces of aluminum, screws and ball bearings the entire robot can be printed. The aluminum is used to connect the separate pieces of the main body and to provide mounting points for future expansions. The main body needed to be cut into several pieces for printing because the available 3D printer's printing bed was too small to fit the entire construction.

Each of the legs has three degrees of freedom and consists of three main sections - a connector that links the leg to the main body, a femur (thigh bone) and a tibia (shin bone). The tibia contains a 9DOF IMU (accelerometer, gyroscope and magnetometer) intended to detect when a leg makes contact with an object and to be used for closed loop controllers.

To keep the costs low, the initial version (shown in 6.1a) uses conventional hobby servos to actuate its joints. Unfortunately this makes executing highly dynamic motions difficult as the servos are moved by providing a target position directly rather than the motor acceleration. Hitec HS-645MG servos with up to 9.6kg-cm of torque are used for the joints farthest from the body and Hitec HS-485HB servos (with up to 6kg-cm torque) for the ones connecting the legs to the body.

The IMUs (MPU-9250) can be connected using either an Inter-Integrated Circuit (I^2C) or Serial Peripheral Interface (SPI) bus. The advantage of I^2C is that it only requires four wires (VCC,GND,SCL and SDA) to be connected to a central controller that configures



(a) Assembled robot (b) Assembly of the robot

Figure 6.1: The Robot

the chips and reads their data. Additional devices employing this protocol could be built into the legs without introducing extra wiring. The maximum frequency of the bus is 400kHz. The accelerometer and gyroscope are capable of sampling at 4kHz while the magnetometer is only able to do so at 8Hz. The values are sampled using 16-bit ADCs so the overall sustainable data rate is almost 385kbit/s. As this is very close to the I^2C bus' maximum data rate SPI was chosen instead. It is capable of dealing with bus speeds of up to 1MHz. The main disadvantage is that a total of six wires are necessary - VCC, GND, SCLK, MOSI, MISO and $\overline{\text{SS}}$. Additional devices can share most of these, but each needs its own $\overline{\text{SS}}$ pin. This reduces the flexibility of this approach slightly. More sensors could be added with the help of an intermediate device that handles the sensors and makes them available to the main controller via the existing wiring. This would also allow one to reverse the master-slave relationship with the central device. Rather than constantly asking for new data the central processor is free to do other work until it is notified of new data.

The servos are connected to the system via two 12-bit PWM controllers (PCA9685) that are controlled using I^2C. These chips are capable of operating even at 3.3V which simplifies the overall power management of the system as the IMUs require 3.3V, while the servos' nominal operating voltage is between 4.8V and 6.0V

An ESP8266 WiFi capable processor was chosen to provide the main interface for external commands (such as the ones provided by the controller in chapter 3). Its sole ADC pin is used to monitor the battery voltage and warn the user if necessary. Currently this device is only used as to interface with a program running on a desktop computer but future versions may use more sophisticated soft- and hardware to increase the robot's autonomy. FPGAs, in particular offer a number of advantages and opportunities for improvement.

CHAPTER 7

Results

All experiments were performed on a machine with a 3.06Ghz i7-950 processor and 24GB of RAM (the application itself used less than 1.5GB in all tests). Since not all the considered single-mode planners have a multi-threaded version available, the single-threaded version for all the planners was used.

The result of each single-mode planner is given to a path-simplification routine that attempts to shortcut and smooth the generated paths.

7.1 Flat Surface

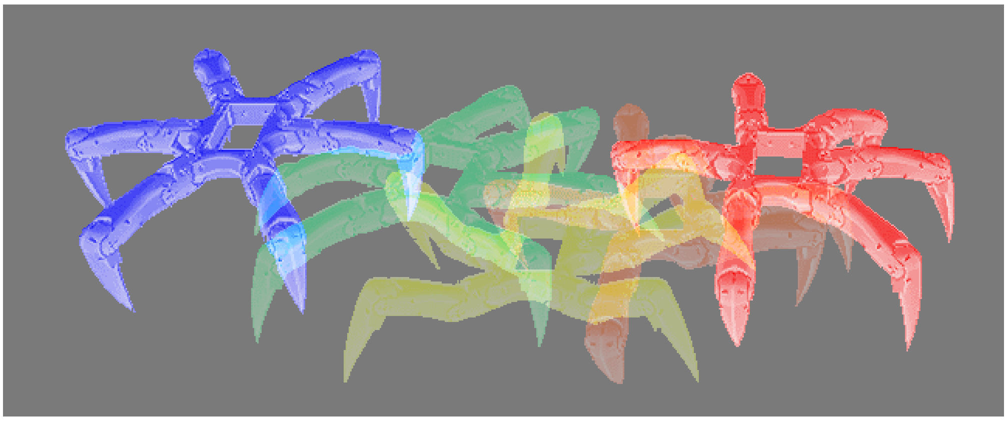


Figure 7.1: States while walking on a flat surface

At first the planner was tested on a flat surface. The objective was to find a path that moves the robot one meter to the side. Five single-mode planners were tested - SBL, STRIDE, KPIECE, RRT and BFMT* [SGS+15](as a bidirectional extension of FMT*

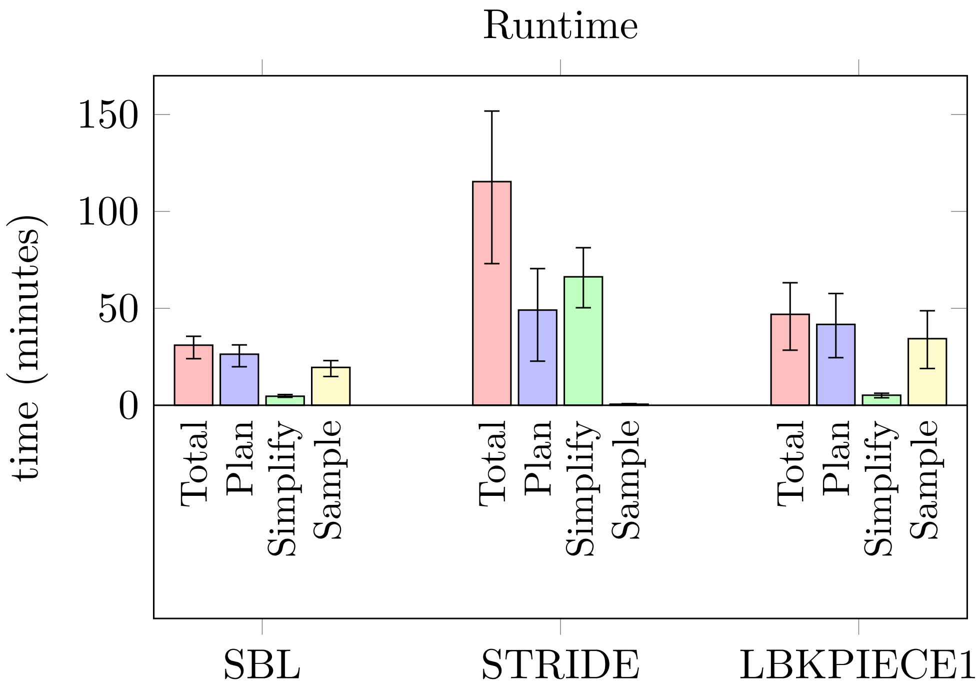


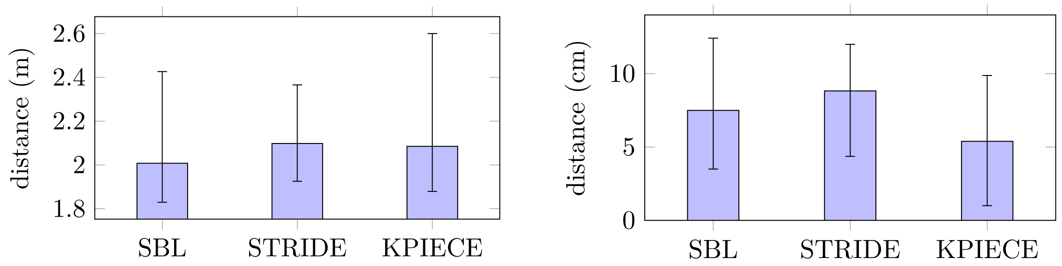
Figure 7.2: Planning time for movement on a flat surface

[JSCP15]). Three planners, SBL, STRIDE and KPIECE, managed to find a path in reasonable time (less than 3 hours). Fig. 7.2 shows the average time each planner spent on certain parts of the algorithms over several attempts using the same sequence of stances. Error bars indicate the fastest and slowest results of all attempts using that particular planner. The total time, shown in red, includes all operations to generate a smooth path between two stances. Right of that the planning time (in blue) is only the time spent by the respective single mode planner. This does not include any smoothing or path simplification (in green). Planning time and simplification time do not add up to the total exactly as that is the average of the sums of planning and simplification time and not the sum of the averages. The rightmost bar indicates how much time each planner spent sampling new states. In the case of SBL and STRIDE all of that time is spent sampling nearby states (as described in 5.2.3). KPIECE is able to use nearby samples as well as arbitrary ones so this number is the sum of both. However, the total time spent on arbitrary samples was less than one second so all three bars show essentially the same thing.

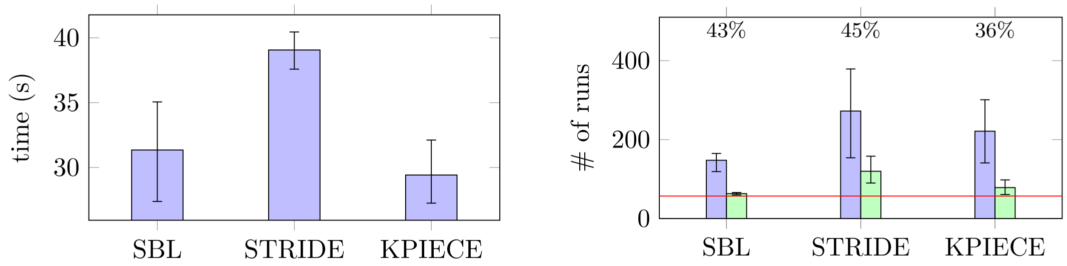
The extreme difference in planning time between STRIDE and the others (SBL in particular as STRIDE was designed as an improvement to it) can be attributed to how they handle connecting new states. In the case of SBL collision checking is done lazily, so more time is spent sampling new states until a potential path is available at which point the edges are checked for validity. Similarly a variant of KPIECE with lazy collision checking was chosen. At the time of writing, the STRIDE implementation in the Open Motion Planning Library was still considered experimental and did not have a version with lazy collision checking so all edges are checked for their validity immediately which is a costly operation that includes matrix inversions (see listing 5.2). Despite this disadvantage the planner managed to find reasonable paths. The long duration of the path simplification process can also be attributed to this disadvantage, since the planner generates fewer samples (resulting in more jagged paths) and so more time is spent on transforming invalid paths to valid ones.

Two variations of RRT were also tried. RRT-Connect, which builds two trees starting from the start and goal states, respectively and LazyRRT which only builds a single tree. Similarly to STRIDE RRT-Connect spends most of its time connecting states while LazyRRT only builds a single tree which has trouble connecting the goal configuration.

If collision checking is relaxed such that intersections with the ground outside the defined footfalls is permissible (and only joint limits remain) then RRT variations often find paths in reasonable time as in many of these cases the start and goal states can be connected directly. Most solution paths only slightly intersect with the ground. Additionally BFMT* ([SGS+15]) was also tested as an alternative. This algorithm is bi-directional as well as lazy and generally performs better than plain RRT. However, it was also unable to find paths within the allotted time.



(a) Average distance moved by each leg (b) Error in the final position of the legs



(c) Execution time by the controller (d) Successful planning instance

Figure 7.3: Movement on a flat surface

Fig. 7.3 compares the results after simulating the movement using the control approach presented in chapter 3. The first graph shows how far the tip of each leg moved in order to reach the goal. This can be used as an indication of how successful the path simplification was. Interestingly there is quite a bit of variation between different solutions. Additionally, when compared with the third plot the execution speed does not seem to correlate much with the path length for this particular problem.

The second plot shows the average error in the final position of the leg tips after simulating the movement. The main cause of these errors is the dynamic friction introduced by solutions that move the leg tip close to the ground. Dynamic friction also makes the paths generated by RRT variants virtually unusable, so they have not been included. To reduce these errors the area of allowed collisions near potential footfalls could be reduced at the cost of increased planning time.

Lastly, the bottom right plot lists how many of the single-mode planner instances were created and how many of those could be solved successfully. The red horizontal bar indicates the lower bound on the number of planner instances. It corresponds to the number of node transitions and is 57 for this particular set of solutions.

For this experiment each planner was given a maximum of 15 seconds per transition (i.e. moving from one stance to the next). The influence of this value is two-fold - on the one hand increasing it gives the planner more time to find a path while on the other hand this also means that more time is wasted on impossible or needlessly difficult transitions.

Another interesting observation here is that despite STRIDE managing to solve slightly more transitions (fig. 7.3d) than the others, it still ends up performing more planning steps overall. Since the goal states are randomly chosen, some of these samples may result in the robot ending up in unfavorable positions. The better performance of STRIDE leads to it still being able to solve the problem in many cases. However, this poor goal state is the start for the next, more difficult planning problem. The increased difficulty means it is more likely to fail and consequently requires backtracking to a previous path segment which wastes time.

SBL finds fewer transitions (especially with poor start and goal states) but these paths often end up providing a better starting point for the following runs of the planner, resulting in less backtracking and faster solutions overall.

7.2 Step

The aim of the second experiment was to find a path in a more difficult environment. Similarly to the previous one the robot needed to move one meter to the side, however this time the goal position could only be reached after climbing a 10cm step. Since only SBL, STRIDE and KPIECE found solutions previously only those planners were tested. Fig. 7.5 shows the time spent by each planner. The ratios are very similar to the ones in fig. 7.2, with STRIDE in particular spending very little time sampling (less than one minute) and an even greater portion simplifying the path. The main difference to the movement on a flat surface is that the planning time was increased to 60 seconds per transition.

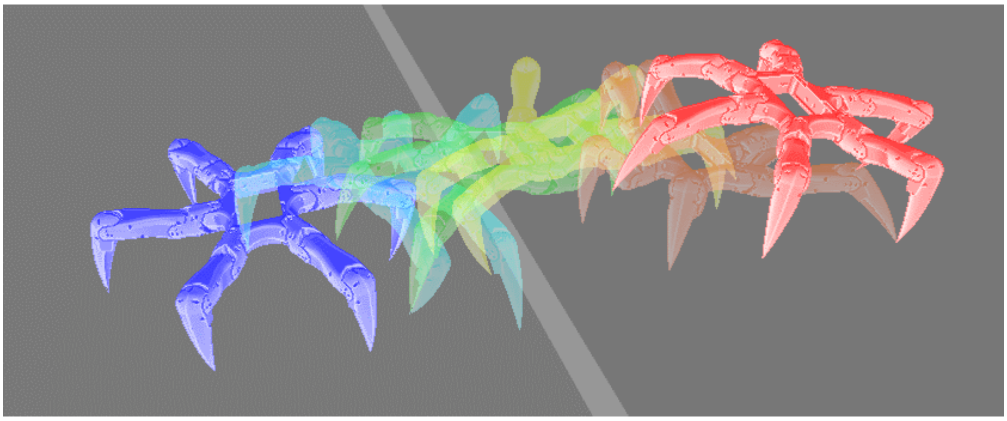


Figure 7.4: Step in the simulation

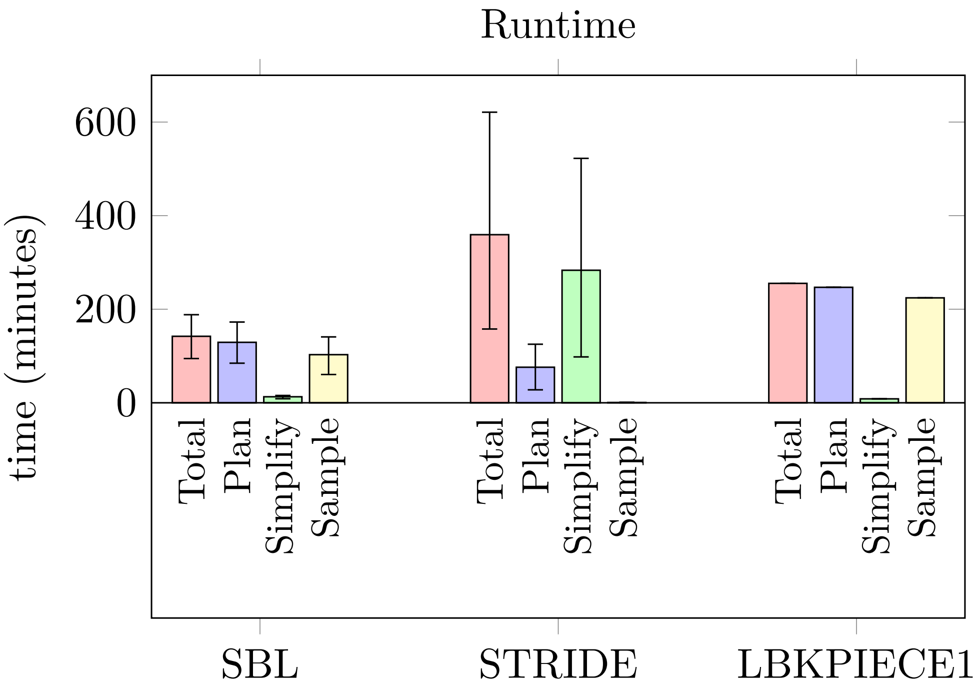
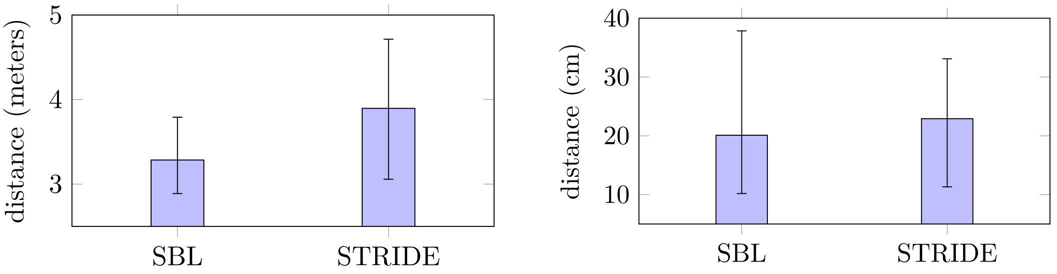


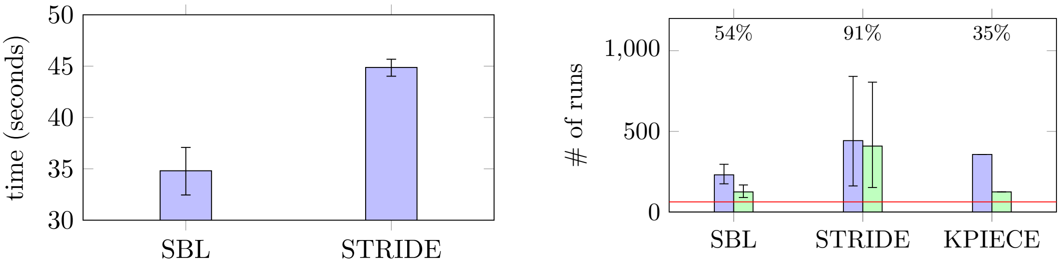
Figure 7.5: Planning time for climbing a step

Fig. 7.6 compares various metrics of the solutions. An important note here is that while SBL and STRIDE eventually solved the problem, KPIECE only did so once during more than ten attempts (cut off after a day). This is the reason why there are no error bars for KPIECE in figs. 7.5 and 7.6. The single solution calculated by the KPIECE based planner also failed to climb the step and consequently no meaningful data could be derived.



(a) Average distance moved by each leg

(b) Error in the final position of the legs



(c) Execution time by the controller

(d) Successful planning instance

Figure 7.6: Climbing a step

The paths found by SBL and STRIDE were usable by the controller most of the time. The ones that failed usually clipped the step in some way, which resulted in the robot pushing itself away from the ledge such that the following stances were nowhere close to their intended positions. These results have not been included in the graphs. However solutions where a majority (but not necessarily all) of the legs reached the upper part of the step were considered successful and have been included.

One observation is that the error in the final position increased noticeably. This indicates that dynamic friction has more of an impact than initially assumed. This is exacerbated by planning the entire path ahead of time and relying on the robot being where it is assumed to be. More local planning approaches (such as only planning the next step) may be more successful here. Control based planning can also help while keeping the global nature of this approach.

The number of single-mode runs gives an idea of how much time each planner wastes on infeasible paths. As the planning time per transition was increased we see an impressive 91% solve rate for STRIDE. Unfortunately, as discussed previously, back-tracking means that many of those paths are discarded.

CHAPTER 8

Conclusion and Future Work

A multi-modal planner has been implemented to automatically derive sequences of footfalls and connect them using various single-mode planners. Motion planning is an essential part of robotics and particularly, as robots become more sophisticated, necessary for them to efficiently interact with their environment.

Legged robots are challenging to work with as they need to be able to move through different submanifolds of their often high-dimensional configuration space. These manifolds are of varying dimensionality themselves and come with their own sets of constraints. The high number of probabilistic planners that are in use today makes it difficult to gauge which ones are applicable to this problem domain and how they need to be adapted in order to be useful.

The work in this thesis can be summarized as follows:

- The implementation of a generic planner that is able to determine a sequence of steps necessary to reach a goal as well as the motions connecting them.
- A comparison of different popular single-mode planners along with adaptations to make them work in the context of legged-locomotion.
- A physical platform to facilitate further development in this domain.

8.1 Future Work

8.1.1 Planning with Dynamics

Dynamics are an important part of smooth and efficient movement. Most of the work presented in this thesis can be adapted to include dynamics at the cost of further increasing the dimension, and thus planning time, of the problem. The closed-loop controller and simulation are already capable of operating with dynamics. Modifications to the hardware may be necessary as conventional position controlled servos are used, rather than directly controlling the motor current.

8.1.2 Planning with Uncertainty

Currently the planner assumes that the robot is where the planner wants it to be. From the simulation alone it has become clear that in many cases this is not enough to arrive at the desired goal. Further work may investigate how these uncertainties can be incorporated to find safer footfalls or compensate for errors during the execution.

8.1.3 Reducing the Planning Time

It is clear that minutes or even hours of planning time are not adequate to operate in an ever changing environment like the real world. Combining purely reactionary control ([RBN+08] [PMS07]) with higher level motion planning like the one presented here might drastically improve the overall performance. Learning efficient gaits ahead of time [Hau08] or generating and gradually improving frequently used sequences of steps may also lead to significant speed and quality gains.

APPENDIX A

IROS 2017 Submission

An Evaluation of Probabilistic Motion Planners in the Setting of a Multi-Modal Planner for a Six-legged Robot

Bernhard Wimmer[1] and Ezio Bartocci[1]

***Abstract*— *Motion planning* consists in translating high-level specifications of tasks into low-level sequences of control inputs for the robot's actuators.**

Legged robots, although more flexible with respect to wheeled robots in uneven and cluttered environments, are a very challenging application domain for motion planning. Such systems may benefit from the use of a multi-modal planner that is able to switch between discrete modes corresponding to the set of contact points between the legged robot and the ground. A single-mode planner then resolves the continuous trajectory that is constrained to a submanifold of the configuration space.

In this paper we evaluate a multi-modal planner using different single-mode planners for controlling a six-legged robot. Two different scenarios are considered: walking on a flat surface and climbing a step. For each scenario we collect several metrics to compare the planners' performance, including the execution time of each algorithm, the required number of single-mode planner instances and the error in the final positions of the footfalls. We believe this comparison is useful in helping others to make informed decisions about which of the common single-mode planners is effective in this context. Furthermore, we also provide some insights on the changes necessary to adapt other planners for this environment.

I. INTRODUCTION

Motion planning is the process of automatically translating a high level specification of a task into a low-level sequence of control inputs for the robot's actuators. The generated trajectories generally need to satisfy different constraints, while aiming to be as efficient as possible with respect to criteria like speed, energy efficiency or safety.

Legged locomotion has been an active area of research for many years due to its obvious advantage in uneven and cluttered environments over wheeled robots. However, besides the difficulty of finding suitable motion sequences, legged robots often have a large number of actuators with varying constraints resulting in a more complex configuration space to explore, turning motion planning into a very challenging task. For example, when a legged robot touches the ground, the possible motions lie in a subspace of the original configuration space. If such a contact is removed (e.g. by raising a leg) the dimension of the motion space is increased. Taking another step reduces the dimensionality again by introducing more kinematic constraints. Motion planning for such robots may benefit from the use of a *multi-modal planner* that is able to switch between a discrete set of modes defined by the contact points between the legged robot and the terrain. A single-mode planner then generates a continuous trajectory through the corresponding submanifold of the configuration space.

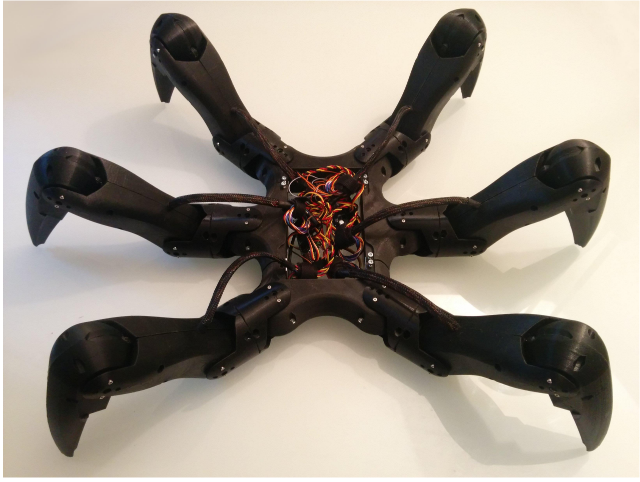


Fig. 1: Six-legged robot.

Multi-modal planning has been investigated in the context of grasping and re-grasping operations [ALS94], [FB97], [NK00], [SCS02], [HNTHGB07], walking [CKNK03], [Hau08] as well as re-configuring robots [CY99].

A complete way to generate a collision free path through the configuration space is exponential in the number of degrees of freedom [Lat03].

The difficulty in exploring the configuration space of such motion planning tasks has spawned a popular class of planners that forgo capturing the entirety of the configuration space in favor of randomly sampling representative portions of it. Probabilistic roadmap planners [KSLO96] and Rapidly-exploring Random Trees (RRT) [LaV98], [KL00] were among the first, with new variations and improvements still being developed. They build a graph that approximates the connectivity of the configurations space by randomly sampling points in this space and connecting them with (usually) straight line paths. They are limited to a single mode and cannot decide if no path exists in finite time. Such algorithms are still a very useful tool for multi-modal planners. Implementations of variations of these algorithms are available in the open source Open Motion Planning Library [SMK12]. A common way to extend their application across modes is to find an intermediate state that is feasible in both modes and then attempting to find a path from the initial state (in the first mode) to the intermediate state (in both modes) and finally to the goal state (in the second

[1] Bernhard Wimmer and Ezio Bartocci are with the Faculty of Informatics, Institute of Computer Engineering, Vienna University of Technology, Vienna, Austria `bernhard.wimmer@student.tuwien.ac.at`, `ezio.bartocci@tuwien.ac.at`

mode). Differential constraints and under-actuated poses still present difficult problems. More recently developed planners enable robots to perform more advanced motions like jumping [Shk10], [DS12].

Over the years a large number of planning algorithms have been proposed. They are usually tested and developed for spaces that do not change or are easily sampleable. This work attempts to test their applicability on a problem where contact constraints drastically alter their typical environment. In particular we provide a performance evaluation of a multi-modal planner controlling a six-legged robot (see Figure 1) by using different single-mode planners.

Two different scenarios are considered: *walking on a flat-ground* and *climbing a step*. In all of the experiments the planner is given the initial and final footfalls as well as the pose of the root link in the initial stance. After that a sequence of stances to reach the goal is generated. This sequence is used to give all planners the same starting conditions. During normal operations this sequence is generated automatically and may change based on the feasibility of randomly generated transition states. If the algorithm terminates successfully, then the generated motions are executed by a closed-loop controller in order to see how useful these results are in a simulated environment. For each scenario we measure the execution time of each algorithm, the number of single-mode planner instances required and the error in the final position of the footfalls.

Organization of the paper. In Section II we provide a background on multi-modal planning, followed by a short overview of the employed single mode planners. In Section III we present a description of the experimental setup. Section IV compares and discusses the results of different approaches in two different scenarios. Finally, we draw our conclusion in Section V.

II. MOTION PLANNING

A. MULTI-MODAL PLANNING

Formally the robot moves through a set of modes (referred to as stances in the rest of this paper) $\Sigma = \sigma_1, \sigma_2, \cdots, \sigma_n$. Each mode may constrain the configuration space $\mathcal{Q}$ of the robot to a set of feasible states $\mathcal{F}_\sigma$. These constraints fall into one of two categories:

- dimensionality-reducing constraints: lower the dimension of the feasible space (in relation to $\mathcal{Q}$). This includes loop-closure constraints (e.g. contacts). They are commonly written in the form $C_\sigma(q) = 0$.
- volume-reducing constraints: reduce the volume of the feasible space but not its dimension. Collisions and joint limits fall into this category. These constraints are expressed as inequalities $D_\sigma(q) > 0$.

In order to switch modes, it is necessary to find the states from a *transition region* between stances. Planning between modes is performed from a state q_σ in $\mathcal{F}_\sigma$ to a transition state $q_{\sigma'}$ in $\mathcal{F}_\sigma \cap \mathcal{F}_{\sigma'}$. This transition state can then be used to find a path through $\mathcal{F}_{\sigma'}$.

The planner terminates once a path from the initial state to any transitions state in $\mathcal{F}_{\sigma_{n-1}} \cap \mathcal{F}_{\sigma_n}$ through the mode-graph is found. It can be trivially extended to terminate in an arbitrary state in $\mathcal{F}_{\sigma_n}$.

Fig. 2 shows an abstract example of the planning approach. A path from the initial (blue node in σ_1) to the goal state (green node in σ_9) needs to be found. The set of feasible states $\mathcal{F}_\sigma$ is represented by the outlined white blobs. Note that each mode can have a number of non-connected components, so failure to find a path through a single mode does not imply that no such path exists in a different component (even if one assumes that the used single mode planner finds a path through a component if it exists). The transition regions between modes are highlighted in blue, and dashed lines indicate which transition states (in red) are identical. A network of straight lines connecting the nodes is used to represent potential paths generated by a single-mode planner.

In this work we implement a hierarchical planner solving the planning problem by interleaving stance exploration with trajectory generation. Initially it is given a *stance* (defined by the contact locations of the legs that touch the ground) and iteratively explores neighboring stances until the goal can be reached. This neighborhood is represented as a *stance-graph* where nodes correspond to stances and edges indicate possible transitions. Each node has an associated cost that aims to encode the quality of a particular distribution of footfalls. Like [Hau08] we have chosen a weighted sum of several factors including: similarity to the nominal stance, distance to the goal and number of iterations required to sample a valid state in the stance. Additionally a penalty for all stances that do not share a footfall with the goal is added. This helps to reduce the number of nodes that need to be explored when the distance to the goal (δ) is very low. In such stances the corresponding factor of the cost function ($\alpha\delta$, with some weight α) is not sufficient to guide the stance exploration toward the goal. This can, for example, lead to cases where only a single footfall needs to be changed to reach the final stance. However, instead of exploring the neighborhood of this stance the planner picks a different stance with fewer correct footfalls due to them having a slightly shorter distance to the goal (and a shorter path from the root, which results in the planner exploring stances close to the goal in a breadth-first manner).

Edge costs are initialized to one and are increased if a single-mode planner fails to find a path between stances connected by an edge. During each iteration an existing stance with the lowest cost (i.e. the sum of the edge costs leading to this node and the cost of the stance itself) is chosen. If that stance is not the goal then the algorithm attempts to find transition states in the neighboring stances and adds them to the graph alongside their corresponding cost. Due to the relatively high computational effort required to fully explore the neighborhood of a given stance the footfalls are placed on a grid (this can easily be changed to allow for different contact locations depending on the environment). If a particular distribution of footfalls is already part of the graph, then its cost is updated with the minimum of the

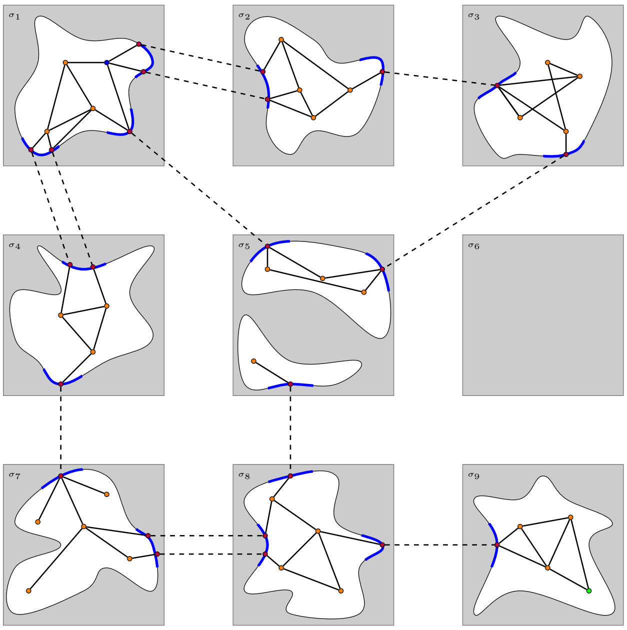


Fig. 2: Multi-modal planning

current and old value (the cost depends on how many tries were necessary to find a valid configuration). Transitions to well connected nodes are sampled more frequently (at least once per explored neighboring stance), which increases the probability of finding *good* states in this stance. Good transition states increase the reusability of partial paths and reduce the likelihood of having to backtrack (see the discussion about STRIDE in Section IV).

Finally the stance is added to a blacklist so it is not examined repeatedly. If the chosen stance is the goal then the shortest path through the stance graph is determined and explored by single-mode planners. Failure to generate a valid trajectory results in an increase of the edge cost corresponding to the failed planner instance followed by the start of the next iteration. Note that the goal stance is not added to the blacklist.

B. *SINGLE MODE PLANNING*

In the following we provide a short overview of the single-mode planners we have used for our test.

1) Rapidly-exploring Random Trees: RRTs [LaV98], [KL00] build one or more tree-structures by sampling random (feasible) configurations and connecting them to the nearest node in the tree. Narrow passages and dead ends initially led to long planning times but extensions, such as building two separate trees from the start and goal configurations help to alleviate some of these problems.

The nature of the configuration space of walking robots adds some challenges to this approach - directly sampling from the sub manifold induced by the contact constraints is generally not possible, and random samples from the configuration space have zero probability of satisfying dimension-reducing constraints (or a very low chance if the constraints are rewritten as inequality constraints $|C(q)| < \epsilon$). Alternatively one can use a sample from the configuration space, along with its nearest valid neighbor in the tree to transform the valid state toward the sampled state along the constraint manifold [BS10].

2) Single-Query Bi-Directional Probabilistic Roadmap Planner with Lazy Collision Checking (SBL): SBL [SL03] builds a network of milestones between the start and goal and performs no collision checking unless absolutely necessary. During each step either the tree rooted at the start or goal nodes is chosen along with an existing milestone in that tree. This milestone is then used to generate a (valid) nearby state which is inserted into the appropriate tree. Following that an attempt to connect the two trees is made and if it succeeds a path from the start to the goal is returned. The main difference to a lazily evaluating bi-directional RRT planner is that an existing node in the tree is chosen first and then a new sample is generated nearby, while RRTs sample first and find the closest existing node later. Projections of the state space have been successfully used to guide the planner toward exploring new areas.

3) Search Tree with Resolution Independent Density Estimation (STRIDE): STRIDE [GMK13] is a motion planner designed for high-dimensional, highly constrained systems. It uses a geometric nearest neighbor tree to estimate the sampling density of the configuration space. Unlike SBL or KPIECE the density estimates are computed in the full dimensional space directly rather than on a lower dimensional projection of it.

4) Kinodynamic Motion Planning by Interior-Exterior Cell Exploration (KPIECE): KPIECE [ŞK09] is a planner for kinodynamic systems (that has later been adapted for non-kinodynamic ones). In addition to the kinematic constraints (joint position limits and collisions) of geometric planners, kinodynamic ones need to be able to operate while respecting additional dynamic constraints like torque, velocity or acceleration limits [DXCR93]. It explores the state space with the help of a hierarchical, grid-based discretization. Sampling of new states is biased toward the boundary between the explored and unexplored space.

5) Bi-directional Fast Marching Tree (BFMT):* BFMT* [SGS15] is an extension of the original Fast Marching Tree [JSCP15] implementation to bi-directional search aiming to provide similar guarantees on its runtime, probabilistic completeness and asymptotic optimality.

A dynamic programming recursion is performed on random samples to grow trees toward regions that minimize the *cost-to-come* (for our system without dynamics this was chosen to be the distance to the goal). This can be imagined as two wavefronts expanding from the initial states. The trees are grown by expanding nodes on the boundary of this wave-front (*frontier-nodes*) according to some expansion strategy. The *alternating tree* strategy operates similarly to SBL [SL03] and RRT-Connect [KL00] by picking one of the two trees at each iteration and extending it. The *balanced tree* strategy picks the tree with the lowest cost to the frontier. A hard upper limit on the runtime is guaranteed by fixing the number of probabilistic samples before the planner is executed. The main reason for including this planner was its ability to sample near existing states like SBL, STRIDE and KPIECE. No serious attempts to improve the planner's performance on this problem were made (doing so might lead to more success in the future).

III. EXPERIMENTAL SETUP

We test our implementation in two different scenarios. First, we establish a baseline for our evaluation of the planning performance by considering a test case where the six-legged robot needs to find a path to move across a flat surface. Second, we evaluate a more challenging scenario where the six-legged robot needs to climb a ledge.

In all of the experiments the planner is given the initial and final footfalls as well as the pose of the root link in the initial stance. After that a sequence of stances to reach the goal is generated. This sequence is used to give all planners the same starting conditions. Additional stances and transitions may be generated automatically during the run if the planner fails to pass certain transitions repeatedly.

If the algorithm successfully terminates a closed loop controller is employed in a simulated environment to follow the trajectory from the start state to a state in the goal stance. The average difference between the achieved and desired positions of the leg tips is reported as well as the number of single-mode planner instances and the corresponding solve rate.

Using this setting we test several different single-mode planners implemented in the Open Motion Planning Library (OMPL,[SMK12]) including: SBL [SL03], STRIDE [GMK13], KPIECE [ŞK09], RRT variants [KL00] [LaV98] and BFMT* [SGS15].

Path validity checking is performed by recursively bisecting the path and moving the midpoint onto the constraint manifold until the distance between neighboring states is small enough. The path is valid if all transformed states are collision free and within the joint limits.

Among the aforementioned planners, SBL, STRIDE and KPIECE provide a direct interface for sampling from the constraint manifold so similarly to [Hau08] and [YLK01] we find an orthonormal basis for the tangent space of the constraint manifold using the Singular Value Decomposition (SVD). New valid samples near a provided one can then be generated by following the tangent space and repairing the resulting state using a (small) number of Newton-Raphson iteration steps.

Since the other planners have no such ability a different approach can be used. Instead of assuming that the final state of a potential path is valid, the starting state can be iteratively moved toward the goal (along the constraint manifold) until a state close enough to the goal, projected onto the manifold, is reached. This intermediate point is then used instead of the random sample [BS10]. While OMPL provides a way to achieve that during the motion-validation step, only KPIECE currently uses this feature (edges connecting states are discarded completely if the path is not valid in its entirety in the other cases).

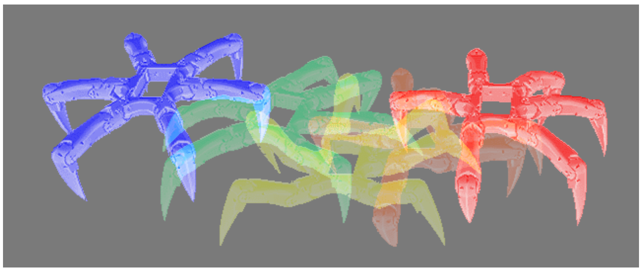


Fig. 3: Snapshots of motion on a flat surface

To be able to use these planners the contact constraints have been slightly relaxed (i.e. turned into inequality constraints $|C(q)| < \epsilon$, with a small ϵ), by considering a contact valid if it is with a certain radius of the desired footfall. This makes the transition regions sampleable and as a result some paths could be found. In the context of the multi-modal planner, however, sampling this region is still very unlikely and finding a trajectory through the entire stance sequence takes an impractical amount of time.

IV. RESULTS

We have performed all the experiments on a machine with a 3.06Ghz i7-950 processor and 24GB of RAM (the application itself used less than 1.5GB in all tests). Since not all the considered single-mode planners have a multi-threaded version available, we used the single-threaded version for all the planners.

The results of the single-mode planners are given to a path-simplification routine that attempts to shortcut and smooth the generated paths.

A. *Planning on a flat surface*

We first test our implementation on a flat terrain with the goal to move the robot one meter to its side.

Fig. 4 shows how much time was spent on various parts of the algorithm. The total time (in red) is the sum of the time spent planning (in blue) followed by a path simplification step (in green).

The yellow bars indicate how much of the planning time was spent sampling new states. The experiment was repeated several times (using the same sequence of stances) and the best and worst cases are marked using error bars. STRIDE is at a slight disadvantage as the planning time was limited to 15 seconds per stance-transition and, unlike the other two, it checks if any potential path segment is valid immediately. SBL and the chosen KPIECE variant both use lazy collision checking, so only a small part of the overall runtime is taken up by path validity checking. In STRIDE's case a significant amount of time is also spent simplifying the resulting path, due to the often jerky paths between states caused by poor coverage of the planning space.

After a valid trajectory has been generated it is given to a closed loop controller that generates the motor commands

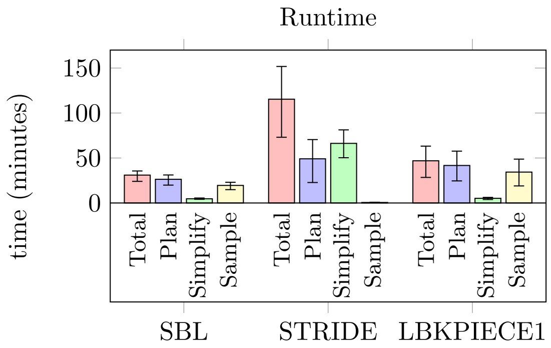
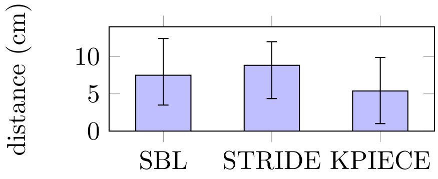
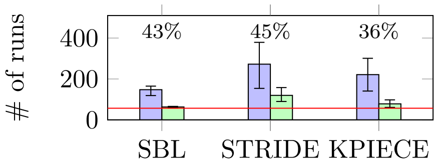


Fig. 4: Planning time on a flat surface

during a simulation. Fig. 5a shows the error in the final position of the footfalls. The main cause of the inaccuracies here are the effects of dynamic friction caused by quickly shifting a significant amount of mass as well as trajectories that touch the ground slightly. Fig. 5b gives an idea of how effective the single mode planners are in this context. The left bars (in blue) list how many single mode planning instances were launched in total. The green bars indicate how many of those were successfully solved. The percentages at the top show the success rate for a given planner. The red line sets a lower bound for the best possible solution (one where each stance transition requires only a single planner instance). Note here that there are more solved transitions than necessary due to the occasional need to backtrack. It may, for example, be possible to find a path between two stances but the resulting goal state could end up being a poor starting state for the next transition, which means the planner needs to find a better goal for the previous planning problem to continue.



(a) Error in final position



(b) Number of single-mode planner instances

Fig. 5: Walking on a flat surface

B. Climbing a step

In this test case the robot needs to move one meter to the side, similarly to the experiment on a flat surface. However, the final position is now on top of a 10 cm high step. In this environment we tested only the three planners that were able to solve the previous problem. We ran each instance several times, however the KPIECE based planner could solve the problem only once (the other attempts were terminated after 12 hours each), so the runtime shown in fig. 6 should not be viewed as being representative. The other two were successful in all runs.

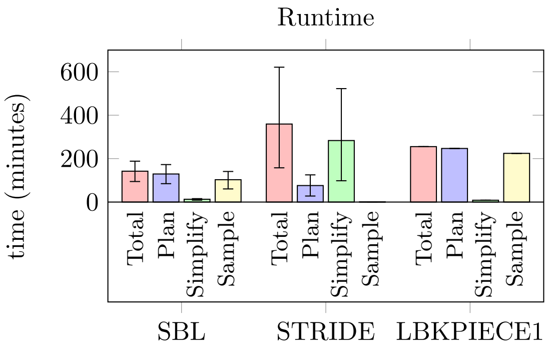


Fig. 6: Planning time to climb a ledge

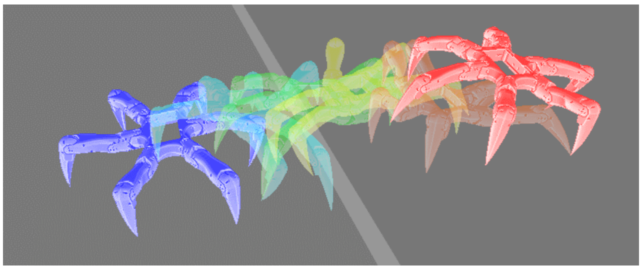
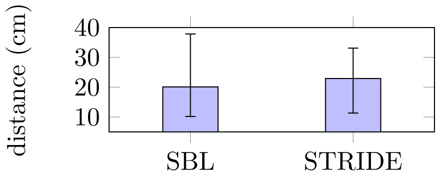


Fig. 7: Snapshots of climbing a step

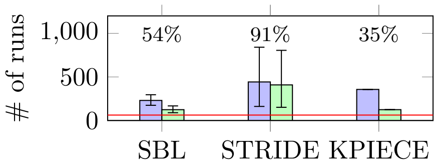
Fig. 6 lists the time spent on various parts of the algorithm. Like in the previous case STRIDE spends very little time sampling new states (less than one minute total) and ends up simplifying the path for a long time instead.

Unlike in the case of walking on a flat surface not all solutions provided by the planner could be executed by the controller. In some cases the accumulated error was enough for the robot to push itself away from the ledge rather than climb it. Only results where a majority of the legs reached the top of the stair were included in fig. 8 (this includes solutions that could be considered stuck, e.g. if two legs are still touching the lower portion of the step). The single valid solution returned with KPIECE as the single-mode planner failed to climb the ledge and has not been included in the graph.

Fig. 8b lists the number of planner instances for this problem. The maximum allowable time for each instance was increased to 60 seconds (from 15 seconds). The result is that slightly more problems were solved by SBL and that a majority of the STRIDE instances were successful. This highlights an important issue - longer planning times and thus more solutions do not necessarily imply a better performance overall. In particular, the high solve rate of STRIDE meant that paths to many poor goal states were found that needed to be revised after a successive goal could not be found.



(a) Error in final position



(b) Number of single-mode planner instances

Fig. 8: Climbing a ledge

V. CONCLUSIONS

In this work we present an implementation of a multi-modal planner in the context of legged-locomotion. In particular our research focuses on the comparison of different planning strategies that can be employed in the discrete modes of our planner. In order to verify and compare the computed solutions, we have performed two experiments in a simulated environment - moving on a flat surface and climbing a step. Additionally a robot was developed to further test the results on a physical platform.

The results indicate that many of the commonly used motion planning methods are only applicable with modifications that often end up nullifying their advantages. Many of the assumptions made during the implementation, such as the ability to follow a given trajectory precisely, do not hold up in the real world or even a simulated one in some cases.

During the simulation we have observed that friction and other dynamic effects play a critical role in the overall performance of legged robots. The closed loop controller can compensate for some of these effects but global position changes and uncertainties need to be accounted for during the planning phase.

Currently the dynamics of the system are only respected during the simulation step, but in future work the general planning approach can be adopted to use optimizing and control-based single-mode planners. Another point of interest lies in uncertainties that are inherent to systems in the real world. For such problems the planner could be improved to adapt to local changes with the help of sensor input.

REFERENCES

[GMK13] Bryant Gipson, Mark Moll, and Lydia E Kavraki. Resolution independent density estimation for motion planning in high-dimensional spaces. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 2437–2443. IEEE, 2013.

[SL03] Gildardo Sánchez and Jean-Claude Latombe. A single-query bi-directional probabilistic roadmap planner with lazy collision checking. In *Robotics Research*, pages 403–417. Springer, 2003.

[DS12] Christopher M Dellin and Siddhartha S Srinivasa. A framework for extreme locomotion planning. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 989–996. IEEE, 2012.

[ŞK09] Ioan A Şucan and Lydia E Kavraki. Kinodynamic motion planning by interior-exterior cell exploration. In *Algorithmic Foundation of Robotics VIII*, pages 449–464. Springer, 2009.

[YLK01] Jeffery Howard Yakey, Steven M LaValle, and Lydia E Kavraki. Randomized path planning for linkages with closed kinematic chains. *IEEE Transactions on Robotics and Automation*, 17(6):951–958, 2001.

[KSLO96] Lydia E Kavraki, Petr Svestka, J-C Latombe, and Mark H Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE transactions on Robotics and Automation*, 12(4):566–580, 1996.

[Hau08] Kris Hauser. *Motion planning for legged and humanoid robots*. ProQuest, 2008.

[LaV98] Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning. 1998.

[Lat03] Jean-Claude Latombe. *Robot motion planning*. The Kluwer international series in engineering and computer science ; 124. Kluwer, Boston [u.a.], 7. print. edition, 2003.

[NK00] Christian L Nielsen and Lydia E Kavraki. A two level fuzzy prm for manipulation planning. In *Intelligent Robots and Systems, 2000.(IROS 2000). Proceedings. 2000 IEEE/RSJ International Conference on*, volume 3, pages 1716–1721. IEEE, 2000.

[KL00] James J Kuffner and Steven M LaValle. Rrt-connect: An efficient approach to single-query path planning. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 2, pages 995–1001. IEEE, 2000.

[SGS15] Joseph A Starek, Javier V Gomez, Edward Schmerling, Lucas Janson, Luis Moreno, and Marco Pavone. An asymptotically-optimal sampling-based algorithm for bi-directional motion planning. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pages 2072–2078. IEEE, 2015.

[BS10] Dmitry Berenson and Siddhartha S Srinivasaz. Probabilistically complete planning with end-effector pose constraints. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 2724–2730. IEEE, 2010.

[SCS02] Anis Sahbani, Juan Cortés, and Thierry Siméon. A probabilistic algorithm for manipulation planning under continuous grasps and placements. In *Intelligent Robots and Systems, 2002. IEEE/RSJ International Conference on*, volume 2, pages 1560–1565. IEEE, 2002.

[Shk10] Alexander C Shkolnik. *Sample-based motion planning in high-dimensional and differentially-constrained systems*. PhD thesis, Massachusetts Institute of Technology, 2010.

[SMK12] Ioan A Sucan, Mark Moll, and Lydia E Kavraki. The open motion planning library. *IEEE Robotics & Automation Magazine*, 19(4):72–82, 2012.

[Wil88] G. Wilfong. Motion planning in the presence of movable obstacles. In *Proceedings of the Fourth Annual Symposium on Computational Geometry*, SCG '88, pages 279–288, New York, NY, USA, 1988. ACM.

[ALS94] Rachid Alami, Jean-Paul Laumond, and Thierry Siméon. Two manipulation planning algorithms. In *WAFR Proceedings of the workshop on Algorithmic foundations of robotics*, pages 109–125. AK Peters, Ltd. Natick, MA, USA, 1994.

[FB97] Pierre Ferbach and Jérôme Barraquand. A method of progressive constraints for manipulation planning. *IEEE Transactions on Robotics and Automation*, 13(4):473–485, 1997.

[HNTHGB07] Kris Hauser, Victor Ng-Thow-Hing, and H Gonzales-Banos. Multi-modal planning for a humanoid manipulation task. In *Intl. Symposium on Robotics Research, Hiroshima, Japan*, 2007.

[CKNK03] Joel Chestnutt, James Kuffner, Koichi Nishiwaki, and Satoshi Kagami. Planning biped navigation strategies in complex environments. In *IEEE Int. Conf. Hum. Rob., Munich, Germany*, 2003.

[CY99] Arancha Casal and Mark H Yim. Self-reconfiguration planning for a class of modular robots. In *Photonics East'99*, pages 246–257. International Society for Optics and Photonics, 1999.

[DXCR93] Bruce Donald, Patrick Xavier, John Canny, and John Reif. Kinodynamic motion planning. *Journal of the ACM (JACM)*, 40(5):1048–1066, 1993.

[JSCP15] Lucas Janson, Edward Schmerling, Ashley Clark, and Marco Pavone. Fast marching tree: A fast marching sampling-based method for optimal motion planning in many dimensions. *The International journal of robotics research*, 34(7):883–921, 2015.

List of Figures

2.1 Example of error due to Euler integration ($x^2 + y^2 = r^2$, $\dot{\mathbf{x}} = [-y, x]^T$) . . . 4
2.2 Rate of change of a rotating vector 10
2.3 Example of an articulated rigid body system 25

3.1 Joint position limit parabola 33

4.2 Configuration-time space 40
4.3 Three abstract examples that show different possible intersection regions between modes 42
4.4 A basic multi-modal planning approach 44
4.5 Path warping using motion primitives 46
4.6 Expansion using a reachability criterion 49
4.7 RRT* algorithm 50

5.1 53
5.2 IK solutions for random poses of the root link 54

6.1 The Robot 62

7.1 States while walking on a flat surface 63
7.2 Planning time for movement on a flat surface 64
7.3 Movement on a flat surface 65
7.4 Step in the simulation 67
7.5 Planning time for climbing a step 67
7.6 Climbing a step 68

List of Tables

2.1 Cash-Karp factors for Runge-Kutta method 8

List of Algorithms

5.1 Support function for a cylinder 57
5.2 Path validity checker used by the local planners 59

Bibliography

[AK$^+$09] Ioan Alexandru, Lydia E Kavraki, et al. On the performance of random linear projections for sampling-based motion planning. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference On*, pages 2434–2439. IEEE, 2009.

[AL09] Andreas Aristidou and Joan Lasenby. *Inverse kinematics: a review of existing techniques and introduction of a new fast iterative solver.* University of Cambridge, Department of Engineering, 2009.

[ALS94] Rachid Alami, Jean-Paul Laumond, and Thierry Siméon. Two manipulation planning algorithms. In *WAFR Proceedings of the workshop on Algorithmic foundations of robotics*, pages 109–125. AK Peters, Ltd. Natick, MA, USA, 1994.

[AW96] Nancy M Amato and Yan Wu. A randomized roadmap method for path and manipulation planning. In *Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on*, volume 1, pages 113–120. IEEE, 1996.

[BBS94] Leoncio Briones, Paul Bustamante, and Miguel A Serna. Wall-climbing robot for inspection in nuclear power plants. In *Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on*, pages 1409–1414. IEEE, 1994.

[BK00] Robert Bohlin and Lydia E Kavraki. Path planning using lazy prm. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 1, pages 521–528. IEEE, 2000.

[BK04] Samuel R Buss and Jin-Su Kim. Selectively damped least squares for inverse kinematics. 2004.

[BOvdS99] Valérie Boor, Mark H Overmars, and A Frank van der Stappen. The gaussian sampling strategy for probabilistic roadmap planners. In *Robotics and automation, 1999. proceedings. 1999 ieee international conference on*, volume 2, pages 1018–1023. IEEE, 1999.

[BS10] Dmitry Berenson and Siddhartha S Srinivasaz. Probabilistically complete planning with end-effector pose constraints. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 2724–2730. IEEE, 2010.

[CD03] Adrian A Canutescu and Roland L Dunbrack. Cyclic coordinate descent: A robotics algorithm for protein loop closure. *Protein science*, 12(5):963–972, 2003.

[CK90] Jeff R Cash and Alan H Karp. A variable order runge-kutta method for initial value problems with rapidly varying right-hand sides. *ACM Transactions on Mathematical Software (TOMS)*, 16(3):201–222, 1990.

[CKNK03] Joel Chestnutt, James Kuffner, Koichi Nishiwaki, and Satoshi Kagami. Planning biped navigation strategies in complex environments. In *IEEE Int. Conf. Hum. Rob., Munich, Germany*, 2003.

[CL01] Peng Cheng and Steven M LaValle. Reducing metric sensitivity in randomized trajectory design. In *Intelligent Robots and Systems, 2001. Proceedings. 2001 IEEE/RSJ International Conference on*, volume 1, pages 43–48. IEEE, 2001.

[CY99] Arancha Casal and Mark H Yim. Self-reconfiguration planning for a class of modular robots. In *Photonics East'99*, pages 246–257. International Society for Optics and Photonics, 1999.

[Dru08] Evan Drumwright. A fast and stable penalty method for rigid body simulation. *IEEE Transactions on Visualization and Computer Graphics*, 14(1):231–240, 2008.

[DS12] Christopher M Dellin and Siddhartha S Srinivasa. A framework for extreme locomotion planning. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 989–996. IEEE, 2012.

[DXCR93] Bruce Donald, Patrick Xavier, John Canny, and John Reif. Kinodynamic motion planning. *Journal of the ACM (JACM)*, 40(5):1048–1066, 1993.

[eCG98] Mo ez Cherif and Kamal K Gupta. Planning for in-hand dextrous manipulation. 1998.

[Eng01] Kenth Engø. On the bch-formula in so (3). *BIT Numerical Mathematics*, 41(3):629–632, 2001.

[FB97] Pierre Ferbach and Jérôme Barraquand. A method of progressive constraints for manipulation planning. *IEEE Transactions on Robotics and Automation*, 13(4):473–485, 1997.

[GMK13] Bryant Gipson, Mark Moll, and Lydia E Kavraki. Resolution independent density estimation for motion planning in high-dimensional spaces. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 2437–2443. IEEE, 2013.

[GO04] Roland Geraerts and Mark H Overmars. Clearance based path optimization for motion planning. In *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, volume 3, pages 2386–2392. IEEE, 2004.

[Hau08] Kris Hauser. *Motion planning for legged and humanoid robots.* ProQuest, 2008.

[HBHL08] Kris Hauser, Timothy Bretl, Kensuke Harada, and Jean-Claude Latombe. Using motion primitives in probabilistic sample-based planning for humanoid robots. In *Algorithmic foundation of robotics VII*, pages 507–522. Springer, 2008.

[HNTHGB07] Kris Hauser, Victor Ng-Thow-Hing, and H Gonzales-Banos. Multi-modal planning for a humanoid manipulation task. In *Intl. Symposium on Robotics Research, Hiroshima, Japan*, 2007.

[JSCP15] Lucas Janson, Edward Schmerling, Ashley Clark, and Marco Pavone. Fast marching tree: A fast marching sampling-based method for optimal motion planning in many dimensions. *The International journal of robotics research*, 34(7):883–921, 2015.

[KF11] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The International Journal of Robotics Research*, 30(7):846–894, 2011.

[KL00] James J Kuffner and Steven M LaValle. Rrt-connect: An efficient approach to single-query path planning. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 2, pages 995–1001. IEEE, 2000.

[KM12] Ross A Knepper and Matthew T Mason. Real-time informed path sampling for motion planning search. *The International Journal of Robotics Research*, page 0278364912456444, 2012.

[KP11] Junggon Kim and Nancy S. Pollard. Direct control of simulated nonhuman characters. *IEEE Comput. Graph. Appl.*, 31(4):56–65, July 2011.

[KSLO96] Lydia E Kavraki, Petr Svestka, J-C Latombe, and Mark H Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE transactions on Robotics and Automation*, 12(4):566–580, 1996.

[KTF+09] Yoshiaki Kuwata, Justin Teo, Gaston Fiore, Sertac Karaman, Emilio Frazzoli, and Jonathan P How. Real-time motion planning with applications to autonomous urban driving. *IEEE Transactions on Control Systems Technology*, 17(5):1105–1118, 2009.

[Lat03] Jean-Claude Latombe. *Robot motion planning.* The Kluwer international series in engineering and computer science ; 124. Kluwer, Boston [u.a.], 7. print. edition, 2003.

[LaV98] Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning. 1998.

[LaV11] Steven M LaValle. Motion planning. *IEEE Robotics & Automation Magazine*, 18(1):79–89, 2011.

[LK04] Andrew M Ladd and Lydia E Kavraki. Fast tree-based exploration of state space for robots with dynamics. In *Algorithmic Foundations of Robotics VI*, pages 297–312. Springer, 2004.

[MD04] Damian Merrick and Tim Dwyer. Skeletal animation for the exploration of graphs. In *Proceedings of the 2004 Australasian symposium on Information Visualisation-Volume 35*, pages 61–70. Australian Computer Society, Inc., 2004.

[MI79] Robert B McGhee and Geoffrey I Iswandhi. Adaptive locomotion of a multilegged robot over rough terrain. *IEEE transactions on systems, man, and cybernetics*, 9(4):176–182, 1979.

[NH86] Yoshihiko Nakamura and Hideo Hanafusa. Inverse kinematic solutions with singularity robustness for robot manipulator control. *Journal of dynamic systems, measurement, and control*, 108(3):163–171, 1986.

[NK00] Christian L Nielsen and Lydia E Kavraki. A two level fuzzy prm for manipulation planning. In *Intelligent Robots and Systems, 2000.(IROS 2000). Proceedings. 2000 IEEE/RSJ International Conference on*, volume 3, pages 1716–1721. IEEE, 2000.

[NRH15] Kourosh Naderi, Joose Rajamäki, and Perttu Hämäläinen. Rt-rrt*: a real-time path planning algorithm based on rrt. In *Proceedings of the 8th ACM SIGGRAPH Conference on Motion in Games*, pages 113–118. ACM, 2015.

[PBK04] Jeff M Phillips, Nazareth Bedrossian, and Lydia E Kavraki. Guided expansive spaces trees: A search strategy for motion-and cost-constrained state spaces. In *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, volume 4, pages 3968–3973. IEEE, 2004.

[PMS07] Dimitris Pongas, Michael Mistry, and Stefan Schaal. A robust quadruped walking gait for traversing rough terrain. In *Robotics and Automation, 2007 IEEE International Conference on*, pages 1474–1479. IEEE, 2007.

[PTVF96] William H Press, Saul A Teukolsky, William T Vetterling, and Brian P Flannery. *Numerical recipes in C*, volume 2. Citeseer, 1996.

[RBN+08] Marc Raibert, Kevin Blankespoor, Gabriel Nelson, Rob Playter, and T Bigdog Team. Bigdog, the rough-terrain quadruped robot. In *Proceedings of the 17th world congress*, volume 17, pages 10822–10825. Proceedings Seoul, Korea, 2008.

[RCB98] Charles Rose, Michael F Cohen, and Bobby Bodenheimer. Verbs and adverbs: Multidimensional motion interpolation. *IEEE Computer Graphics and Applications*, 18(5):32–40, 1998.

[SCS02] Anis Sahbani, Juan Cortés, and Thierry Siméon. A probabilistic algorithm for manipulation planning under continuous grasps and placements. In *Intelligent Robots and Systems, 2002. IEEE/RSJ International Conference on*, volume 2, pages 1560–1565. IEEE, 2002.

[SGS+15] Joseph A Starek, Javier V Gomez, Edward Schmerling, Lucas Janson, Luis Moreno, and Marco Pavone. An asymptotically-optimal sampling-based algorithm for bi-directional motion planning. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pages 2072–2078. IEEE, 2015.

[Shk10] Alexander C Shkolnik. *Sample-based motion planning in high-dimensional and differentially-constrained systems*. PhD thesis, Massachusetts Institute of Technology, 2010.

[ŞK09] Ioan A Şucan and Lydia E Kavraki. Kinodynamic motion planning by interior-exterior cell exploration. In *Algorithmic Foundation of Robotics VIII*, pages 449–464. Springer, 2009.

[ŞK12] Ioan A Şucan and Lydia E Kavraki. A sampling-based tree planner for systems with complex dynamics. *IEEE Transactions on Robotics*, 28(1):116–131, 2012.

[SL03] Gildardo Sánchez and Jean-Claude Latombe. A single-query bi-directional probabilistic roadmap planner with lazy collision checking. In *Robotics Research*, pages 403–417. Springer, 2003.

[SLN00] Thierry Siméon, J-P Laumond, and Carole Nissoux. Visibility-based probabilistic roadmaps for motion planning. *Advanced Robotics*, 14(6):477–493, 2000.

[SMK12] Ioan A Sucan, Mark Moll, and Lydia E Kavraki. The open motion planning library. *IEEE Robotics & Automation Magazine*, 19(4):72–82, 2012.

[SWT09] Alexander Shkolnik, Matthew Walter, and Russ Tedrake. Reachability-guided sampling for planning under differential constraints. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 2859–2865. IEEE, 2009.

[WC91] L-CT Wang and Chih-Cheng Chen. A combined optimization method for solving the inverse kinematics problems of mechanical manipulators. *IEEE Transactions on Robotics and Automation*, 7(4):489–499, 1991.

[WE84] William A Wolovich and H Elliott. A computational technique for inverse kinematics. In *Decision and Control, 1984. The 23rd IEEE Conference on*, pages 1359–1363. IEEE, 1984.

[Wil88] G. Wilfong. Motion planning in the presence of movable obstacles. In *Proceedings of the Fourth Annual Symposium on Computational Geometry*, SCG '88, pages 279–288, New York, NY, USA, 1988. ACM.

[WP95] Andrew Witkin and Zoran Popovic. Motion warping. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 105–108. ACM, 1995.

[WvdB13] Dustin J Webb and Jur van den Berg. Kinodynamic rrt*: Asymptotically optimal motion planning for robots with linear dynamics. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 5054–5061. IEEE, 2013.

[WVDBH08] Ron Wein, Jur Van Den Berg, and Dan Halperin. Planning high-quality paths and corridors amidst obstacles. *The International Journal of Robotics Research*, 27(11-12):1213–1231, 2008.

[YLK01] Jeffery Howard Yakey, Steven M LaValle, and Lydia E Kavraki. Randomized path planning for linkages with closed kinematic chains. *IEEE Transactions on Robotics and Automation*, 17(6):951–958, 2001.