2019-04-29

An Omni-Directional Kick Engine for NAO Humanoid Robot

Pedro Pena
University of Miami, pedro@cs.miami.edu

Follow this and additional works at: https://scholarlyrepository.miami.edu/oa_theses

Recommended Citation
https://scholarlyrepository.miami.edu/oa_theses/751
UNIVERSITY OF MIAMI

AN OMNI-DIRECTIONAL KICK ENGINE FOR NAO HUMANOID ROBOT

By

Pedro Peña

A THESIS

Submitted to the Faculty
of the University of Miami
in partial fulfillment of the requirements for
the degree of Master of Science

Coral Gables, Florida

May 2019
AN OMNI-DIRECTIONAL KICK ENGINE FOR NAO HUMANOID ROBOT

Pedro Peña

Approved:

Ubo Visser, Ph.D.
Associate Professor of Computer Science

Geoff Sutcliffe, Ph.D.
Professor of Computer Science

Christine Lisetti, Ph.D.
Associate Professor of Computer Science
Florida International University

Guillermo Prado, Ph.D.
Dean of the Graduate School
Incorporating a dynamic kick engine that is both fast and effective is pivotal to be competitive in one of the world’s biggest AI and robotics initiative: RoboCup [1]. Using the NAO robot [2] as a testbed, we developed a dynamic kick engine that can generate a kick trajectory with an arbitrary direction without prior input or knowledge of the parameters of the kick. The trajectories are generated using cubic splines, sextic polynomials, and cubic Hermite splines, and the trajectories are executed while the robot is dynamically balancing on one foot. When the robot swings the leg for the kick motion, unprecedented forces might be applied on the robot, and to compensate for these forces, we developed a Zero Moment Point (ZMP) based preview controller that minimizes the ZMP error. Although a variety of kick engines have been implemented by others, there are only a few papers on how kick engine parameters have been optimized to give an effective kick or even a kick that minimizes energy consumption and time. Parameters such as kick configuration, limit of the robot, or shape of the polynomial can be optimized. We propose an optimization framework based on the Webots simulator [3] to optimize these parameters. We also integrated a state-of-the-art walk engine from Seekircher and Visser [4] and kick controller from Pena, Masterjohn, and Visser [5] to generate a kick while walking. Experiments of the physical robot show promising results.
To my family and RoboCanes
Table of Contents

LIST OF FIGURES vii

LIST OF TABLES x

1 INTRODUCTION 1

1.1 Related Work ................................................. 4
1.2 Motivation .................................................. 7

2 BACKGROUND 9

2.1 Kinematics of Humanoid Robots ............................. 9
2.1.1 Rotation Representations ................................. 14
2.1.1.1 Exponential Coordinates .............................. 14
2.1.1.2 Versors ................................................ 16
2.1.1.3 Euler Angles .......................................... 17
2.1.2 Kinematic Parameterization .............................. 19
2.1.2.1 Twists ............................................... 19
2.1.2.2 Screws ............................................... 20
2.1.2.3 Denavit-Hartenberg Parameters .................................. 22
2.1.3 Forward Kinematics ................................................... 23
2.1.4 Inverse Kinematics ................................................... 24
2.2 Stability ...................................................................... 25
  2.2.1 Static Stability ...................................................... 25
  2.2.2 Dynamic Stability ................................................... 27
2.3 Motions .................................................................. 28
  2.3.1 Static Motions ........................................................ 28
  2.3.2 Dynamic Motion ..................................................... 29
  2.3.3 Centroidal Dynamics ................................................. 32
  2.3.4 Interaction Dynamics ................................................. 33
    2.3.4.1 Simple Impedance Control ................................. 35
3 APPROACH ................................................................. 39
  3.1 ZMP based Preview Controller ....................................... 39
    3.1.1 Zero Moment Point ............................................... 39
    3.1.2 Preview Controller ................................................. 42
  3.2 Simulated CoM Trajectory ............................................. 43
  3.3 Analytical Approximation for Inverse Kinematics ................. 46
  3.4 Kick Trajectory .......................................................... 48
    3.4.1 Kick Trajectory using Cubic Spline .......................... 48
    3.4.2 Kick Trajectory using Cubic Hermite Spline ................. 51
3.4.3 Kick Trajectory using Sextic Polynomial .......................... 52
3.5 Rigid-Body Simulators ....................................................... 53
  3.5.1 GazeboSim ................................................................. 56
  3.5.2 Webots ................................................................. 57
  3.5.3 V-Rep ................................................................. 57

4 IMPLEMENTATION ................................................................. 61
  4.1 Dynamic Kick Engine ....................................................... 61
  4.2 Dynamic Walk Kick ......................................................... 62

5 RESULTS AND SUMMARY ......................................................... 66
  5.1 Dynamic Kick ............................................................... 66
  5.2 Model Optimization .......................................................... 68
    5.2.1 Kick Trajectory Optimization ........................................ 71
  5.3 Summary ................................................................. 74

BIBLIOGRAPHY ................................................................. 78
## List of Figures

1.1 Knight Robot built by Leornado Da Vinci .............................. 2
1.2 Robot dynamically generating motions to kick a ball [5] ............... 3
1.3 Overview of the control framework for the kick engine [5,6] ......... 4

2.1 Kinematics of NAO Robot ................................................. 10
2.2 Coordinate Axis of Robot [2] ............................................. 11
2.3 Coordinate Axis of Robot .................................................. 13
2.4 Unit angular vector with scalar angle ................................... 16
2.5 ZYX Euler Angles .......................................................... 19
2.6 Screw Motion ................................................................. 21
2.7 Pepper robot from SoftBank Robotics [7] ................................ 26
2.8 Keyframe Animation from Aldebran’s Choreographe [8] ............. 29
2.9 Dynamic Motion Framework ............................................... 30
2.10 Passive Dynamic Walker from Collins et al. [9] ....................... 32
2.11 The dynamics of the robot are based on the center of mass .......... 33
2.12 Interaction Control ........................................................ 38

3.1 Support Polygon ............................................................. 39
3.2 ZMP location ................................................................. 39
3.3 The acceleration of the center of mass while the robot is kicking . . . 45
3.4 An approximation of the center of mass kinematic .................... 46
3.5 Kick trajectory interpolators ............................................ 48
3.6 Kick trajectory cubic interpolator ...................................... 50
3.7 communication between the simulator and the RoboCanes agent . . 54
3.8 An overview of the communication between the simulator and the RoboCanes agent .................................................. 55
3.9 Gazebo Graphical Interface [10] ........................................ 56
3.10 Webots Graphical Interface [3] ........................................... 57
3.13 Head yaw and pitch is controlled by the RoboCanes agent in V-Rep . 60
4.1 Overview of the dynamic kick engine framework [5] .................... 62
4.2 A side kick in the physical world and a front kick in the simulation . 63
4.3 Walk kick framework ....................................................... 64
4.4 Example of walk kick trajectory ........................................... 65
5.1 The ZMP error between the reference point and the actual ZMP, and the position of the CoM in x and y. ................................. 68
5.2 The CoM velocity, and the kick limit of the NAO in 3D space. ...... 69
5.3 The front kick trajectory in 3D space, and the back kick trajectory in 3D space. ............................................................... 70
5.4 Optimization framework for the kick engine and the simulation control framework .......................................................... 72
5.5 Results from optimization on Webots simulator ....................... 73
5.6 The beginning of the optimization where the robot cannot kick the ball without the system failing to recover 75

5.7 The optimization after many iterations finds stronger kicks that allow the system to be in balance 76
List of Tables

5.1 Optimization results .............................................. 67
CHAPTER 1

Introduction

Building humanoid robots and integrating them into society has always been a motivation for scientists and engineers to solve the problem. One of the earliest accounts of humanoid robots is from Leonardo Da Vinci in 1495 who had designs of a robot that looked like a knight that can stand up on its feet and wave its arm [12]. Figure 1.1. shows the knight robot possibly built by Leonardo Da Vinci.

Humanoid robots are very promising for the future because we might be able to understand more clearly how the human body works which will help with technologies such as exoskeletons for human advancement. They can also help us in the future at home to wash our clothes and clean our house. The question is: is it necessary to have humanoid robots doing these tasks? The answer is not clear but what is clear is humanoid robots will be able to do anything a human might be able to do, and the environment we live in was built with the human anatomy in mind, therefore, these humanoid robots will be able to integrate into our environments without having to make expensive modifications to our structures. They will be useful in environments that are dangerous to humans such as Nuclear decommissioning or Mars exploration. National Aeronautical Space Administration (N.A.S.A) has recently built a humanoid robot called Valkyrie [13] that was built for space exploration in mind. For example
if humanoid robots are sent to Mars first, they can build our colonies and go to places that wheeled robots are not able to. They are also very efficient such as going over barriers or walking over gaps which might be impossible for other robots. Human shaped robots can also be very efficient at walking since dynamic walking can conserve energy. Also, robots that are human-like are also easier to integrate into our lives. If robots were completely different from us, we might feel scared or even threatened by them. This has motivated engineers and scientists to solve this problem.

RoboCup, *Robot World Cup Initiative*, “is an initiative to foster AI and intelligent robotics research by providing a standard problem where wide range of technologies can be integrated and examined” [1]. RoboCup has designed interesting competitions such as robot soccer that utilizes humanoid robots to help with this initiative. The mission of this competition is to beat the world’s best human FIFA soccer team by 2050. An important task, therefore, is to be able to perceive in real-time and act accordingly. In order to act in the physical world while internally handling all the controls to deal with the dynamics of the physical world, dynamic motions need to be considered. An essential motion in soccer is kicking the ball, and an example of a
robot dynamically generating motions to kick a ball is shown in figure 1.2. Generating dynamic motions on a robot without explicitly programming the motion is a difficult task and a fairly new research area. Kick engines such as [14–19] have been developed for the NAO robot and although they are dynamic, there are static values incorporated into the kicks such as retraction points, foot position from floor, hip/ankle pitch and hip/ankle roll ratio, and shapes of trajectories. These values are usually derived from empirical observations and are not guaranteed to be optimal values. The difficulty of the task is to optimize parameters on the physical robot because the robot is limited by hardware, energy consumption, and most importantly real time execution. Hence, the robot cannot run for thousands or millions of iterations to get a good set of parameters. Other dynamic kick engines such as from Yi et al. developed a kick engine for the THOR-MANG from Robotis that generates static kick motions [20]. Lengagneua et al. have developed a kick engine that incorporates optimization offline and a re-planning step when the kick is executed but do not optimize on the physical robot or use a simulation software that takes into account hardware properties of the robot [21]. We propose a new kick engine for the NAO robot that can dynamically
balance on one foot using a Zero Moment Point (ZMP) based preview controller while also generating kick trajectories using cubic splines, cubic Hermite splines, and sextic polynomials [5, 6]. The values of the kick trajectories are optimized in the Webots Simulator [3] to get a good set of parameter values. The overview of the control framework is presented in 1.3. The major components of the control framework are preview controller, spline trajectory, center of mass (CoM), inverse kinematics, and parameter optimization.

The thesis is organized as follows: I will discuss relevant work in the next section and describe the ZMP based preview controller in section 3.1. In section 3.3, I will describe how to move the CoM to the desired position, and in section 3.4.1, I will describe how the kick trajectory is generated. The model optimization is discussed in section 5.2, and the experiments and results are explained in section 5, followed by the conclusion in section 5.3.

1.1 Related Work

Wenk et al. [18] implemented a kick engine that generates online kick motions using trajectories generated by Bézier curves, but in order to create such motions, the humanoid robot has to dynamically balance on one foot so it can handle any
force generated by the kick. In order to find these forces, Wenk et al. use inverse
dynamics to calculate the ZMP; inverse dynamics can be computed such that given
the acceleration, forces are calculated. The inverse dynamics problem is solved using
a variation of the Recursive Newton Euler Algorithm (RNEA) [22]. For the kick
trajectories, the authors use Bézier curves that are continuously differentiable to
generate a kick.

Böckmann et al. [19] provided a mass spring damper model to model motor be-
havior and modified the ZMP equation to account for this behavior to get the actual
motor position rather than a believed state. The authors also adapted Dynamic Mo-
tion Primitives (DMP) to generate kick trajectories. Kick trajectories are usually
generated from Bézier curves or via-point kick trajectories, but the authors use a PD
controller with a forcing term in the transformation system to control the shape of
the trajectory.

Sung et al. [23] use full body motion planning and via-point representation to
generate joint angle trajectories. These trajectories are generated using five degree
polynomials and via-points are specified to constrain the swing trajectory. For the
balancing the authors also use ZMP. In order to create efficient full body motion tra-
jectories, the author use optimization techniques such as Semi-Infinite Programming
(SIP) to specify constraints such as minimal energy and torque. The optimization
also deals with joint redundancy.

Yi et al. [20] use THOR-OP (Tactical Hazardous Operations Robot - Open Plat-
form) full sized humanoid robot to generate kick and walk motions for the RoboCup
2014 Soccer - Humanoid AdultSize sub-league. The robot has a hybrid walking con-
troller that uses two types of controllers: ZMP preview controller and a ZMP based
reactive controller. The ZMP based reactive controller uses techniques such as Central Pattern Generators to create motions that require less computation than the ZMP preview controller. The kick motions generated are handled by the hybrid walking controller to create smooth transitions between the dynamic walk and strong kick.

The kick engine of Xu et al. [15] is separated into four phases: preparation, retraction, execution, and wrap-up phase. The authors use a grid space to find the kick that maximizes the distance in the retraction point and minimizes the angle between the direction of the foot and the direction of the ball. The maximum distance of the retraction point of the foot is assumed to create the greatest impulse. The stabilization of the robot is done with a Body Inclination Control that controls the torso angle to maintain the CoM in the support polygon. The authors also find the reachable space through experimentation. The kick engine was experimented on the NAO robot. Wenk and Röfer developed a Linear Quadratic Regulator and Cart-Table Preview Controller for balance estimation and Bézier curves to generate kick trajectories [18].

Becht et al. [14] use a proportional controller to minimize the distance of the center of mass with regards to the support polygon of the supporting foot. The center of the polygon is found empirically by determining the center of the support polygon is three centimeters from the center. The authors also use the data from the foot sensors to recognize external disturbances. To move the center of mass to a desired position, the authors use a numerical inverse kinematics solution. The kick trajectory is determined by the retraction point and the contact point. The contact point of the ball is determined by the center of mass of the ball and the force destination. The retraction point is the point that is farthest away from the ball. The rest of the
points is solved using the inverse kinematics solution. Wang et al. [24] use machine learning algorithms such as Q-learning to optimize parameters in their kick motion in simulation. Learning kick motions for the NAO robot in simulation has also been investigated by other researchers [24–26].

Müller et al. [17] implemented a kick engine that generates kick motions using Bézier curves and the balancing is done using a PID controller that controls the center of mass using gyroscope data. Kim et al. use Whole-Body Operational Space Control to generate stable dynamic motions and use a virtual model as an interface for a real robot using an extended Kalman-filter [27]; the controller was tested on a NAO robot generating dynamic kick motions. A motion optimization for a 30 degrees of freedom humanoid robot, HRP-2, is also done for kick motions where the minimization of energy and joint friction is considered [28]. Choi et al. use an impact-based trajectory planner for a kicking robot where the initial velocity and launch angle of the ball is determined by modeling the external and aerodynamic forces [29].

1.2 Motivation

Although these approaches use a dynamic kick engine that utilizes a controller for balancing and generates kicks online or offline, they all lack a framework that uses model optimization to find kicks under certain constrains such as strongest kick or fastest kick. In our approach, we use a controller for balancing and cubic splines, Hermite cubic splines, or sextic splines for kick trajectory, and then use optimization to find a good set of parameters for the kick. Also, we developed a walk-kick controller which integrates a kick controller that generates kick trajectories derived from Pena et al. [5, 6] and the walk engine from Seekircher and Visser [4] which makes
no assumption of the walk model and allows the robot to use the model as a black box to plan steps while also utilizing a torso angle controller to keep the robot from falling. Since the balance controller used for the walk engine uses the walk model as a black box, essentially enabling modularity in the controller to add features such as the kick controller, it grants the kick trajectories to integrate with the walk engine without causing the robot to destabilize. This notion of kicking in any direction dynamically while the robot is walking is not considered in the literature as of yet. In the next section, we will discuss the mathematical background used to implement the controllers.
CHAPTER 2

Background

2.1 Kinematics of Humanoid Robots

In order to describe the motions of a humanoid robot, one needs a way to describe the system in hand. One way to describe the robot is using kinematics which is a description or a property of the motion without integrating the forces of the system. Therefore by using relative information such as position and rotation, we can obtain the configuration of the robot in a single frame. In the case of the NAO robot, the robot’s motion is described using a system of links and joints. The links of the system serves to describe the offsets of the joints from the parent. It also describes the rotation relative to the parent. This information is used in a tree where a root is selected which is usually the head or the torso. The NAO’s root joint is the torso. Illustrated in fig 2.1 is the kinematic tree of the NAO robot. The links of the robots are the joints of the robots, and the joints of the robots are all revolute joints which means that the joint rotates around an axis. The axis that describe the NAO robot is shown in figure 2.2. The coordinates are a right hand coordinate system where the Z axis points upright and the rotations are described using Euler’s angles. Therefore if a revolute joint rotates about the Z axis, this is defined as a yaw angle. If the
Figure 2.1: Kinematics of NAO Robot [30]
revolute joint rotates about the X axis, the angle is a roll, and a pitch angle if it rotates about the Y axis. In figure 2.1, yaw revolute joints are green, roll revolute joints are orange, and pitch revolute joints are yellow. By the hip, there is a diagonal revolute joint, which is called a Yaw-Pitch because it is a combination of the yaw and pitch revolute joints. These blue joints move in unison and are considered one joint. Therefore there is a total of 25 joints which means the degree of freedoms of a NAO robot is 25. It is also important to notice that the position of the robot can be described with position which belongs to $\mathbb{R}^3$ and with changes in rotation which belongs to $\mathbb{R}^3$. Therefore a robot that resides in three dimensional space has a total of six degrees of freedom (DOF) which is referred to as total degree of freedom (TDOF). The robot is able to reach any point in space using three degrees of freedom but it is constrained in one orientation. Otherwise by having at least six DOF, the robot can reach a point in space in different orientations. A robot where DOF is less than
TDOF is said to be nonholonomic and if DOF greater than or equal to TDOF is said to be holonomic [31]. If the DOF is strictly greater than TDOF, the robot is said to be redundant [31]. In the case of a humanoid robot that has at least 24 DOF such as the NAO robot, it is said to be kinematically redundant. Kinematic redundancy complicates the kinematics of the robot because many configurations can reach a point in space, hence, an optimal configuration needs to be computed. To describe the links of the robot using the joints, a homogeneous transformation is used. A homogeneous transformation describes relative information using a Special Euclidean Group in three dimensional space, better known as $SE(3)$ where,

$$SE(3) = \{(p, R) : p \in \mathbb{R}^3, R \in SO(3)\} = \mathbb{R}^3 \times SO(3)$$

and,

$$SO(3) = \{(R : R \in \mathbb{R}^{3 \times 3}, RR^T = I, det(R) = +1\}.$$

Henceforth $R$, the rotation matrix, has the appearance of

$$R = \begin{bmatrix}
\cos(\theta) & -\sin(\theta) & 0 \\
\sin(\theta) & \cos(\theta) & 0 \\
0 & 0 & 1
\end{bmatrix}.$$

It can be verified that $RR^T$ is the identity matrix and the determinant is positive one. This rotation matrix is a yaw rotation and it describes the relative rotation with respect to the parent frame, an illustration is depicted in figure 2.3. The reader might also notice that the distance between the origins of the frame is zero, therefore the
homogeneous transformation matrix is the following,

\[
\begin{bmatrix}
R & p \\
0 & 0 & 0 & 1
\end{bmatrix}
= \begin{bmatrix}
cos(\theta) & -sin(\theta) & 0 & 0 \\
sin(\theta) & cos(\theta) & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}.
\]

In the case where the distance between the origins is not zero such as the robot’s kinematic tree, then \( p \) is non zero and describes the distance in \( \mathbb{R}^3 \). It is also important to notice that by concatenating these transformation matrices, it is easy to obtain the relative information of the whole kinematic tree. For example, if it is important to know the homogeneous transformation between the torso and knee. Then, it can be computed as follows,

\[
torsoT_{knee} = torsoT_{hip}^\text{hip}T_{knee}.
\]

The final transformation computed is the relative information of torso and knee. Hence, any point in the knee frame can be transformed into the torso frame. In order to achieve the opposite, the inverse of the transformation can be computed. Therefore in order to transform any point in the torso frame to the knee frame, the following matrix is needed, \( (torsoT_{knee})^{-1} \). By having these powerful tools in our possession, we
can now compute the forward kinematics and inverse kinematics of the robot. The full explanation and proofs of section 2.1 can be found in Murray, Li, & Sastry [32].

2.1.1 Rotation Representations

Kinematic representations help to represent each joint as a special coordinate. These coordinates can then be used to compute the kinematic tree of the robot. The reason the representations are used and not the full $\mathbb{R}^6$ which describes a joint in three-dimensional space is due to minimization of parameters. For example, Denavit-Hartenberg parameters are used to reduce the parameter space from $\mathbb{R}^6$ to $\mathbb{R}^4$. In the case of exponential coordinates, it helps us obtain a representation of the rotation matrix where the rotation matrix does not need to be constructed, but rather it uses the angular velocity vector of the joint and the amount of rotation at a particular time to compute the rotation matrix using the Rodrigues’ Formula [32]. The quaternions are also used to avoid the singularities of the exponential coordinates, and it helps the exponential coordinates avoid surjectivity by representing the parameters in quaternions.

2.1.1.1 Exponential Coordinates

$R$ which belongs to SO(3) needs to be a function of $\theta$ and $\omega$ where $\omega$ is a unit vector that describes the angular velocity vector of the revolute joint. Figure 2.4 shows an example of the angular velocity vector, $\omega$, which is the rotational axis of the revolute joint. The magnitude of rotation is shown as $\theta$. Now, consider a point $q(t)$ in a rotating body. The velocity of this point is:

$$\dot{q}(t) = \omega \times q(t) = \hat{\omega}q(t),$$

(1)
since the velocity of any point about a rotating body is always the cross product of
the angular velocity vector with the radius of that point from the angular velocity
vector. Therefore equation 1 shows the amount that point is rotating over time. The
integration of this differential equation is trivial because it is separable. Hence if we
move $dt$ to the right and $q(t)$ to the left hand side:

$$ \frac{dq}{q(t)} = \hat{\omega} dt $$

and we integrate this equation, the equation becomes,

$$ q(t) = e^{\hat{\omega} t} q(0). $$

If the initial value, $q(0)$, is the initial position of the point at $t = 0$, then the
rotation matrix is defined as:

$$ R(\omega, \theta) = e^{\hat{\omega} \theta}. $$

$e^{\hat{\omega} \theta}$ is a matrix exponential and the Taylor’s expansion of the matrix exponential is,

$$ e^{\hat{\omega} \theta} = I + \hat{\omega} \theta + \frac{(\hat{\omega} \theta)^2}{2!} + \frac{(\hat{\omega} \theta)^3}{3!} + ... $$

$\hat{\omega}$ has the following properties,

$$ \hat{\omega}^{2k} = -1^{k-1} \hat{\omega}^2 $$

$$ \hat{\omega}^{2k-1} = -1^{k-1} \hat{\omega} $$

where $k \geq 2$ and $k \in \mathbb{Z}$ and the higher power terms follow recursively, and we get the
following equation using the Taylor series of $\sin(\theta)$ and $\cos(\theta)$,

$$ e^{\hat{\omega} \theta} = I + \hat{\omega} \sin(\theta) + \hat{\omega}^2 (1 - \cos(\theta)). $$

This equation is called the Rodrigues’ formula and it is used to obtain the rotational
matrix from an angular velocity vector and angle. The issue with this exponential
representation is that it is surjective onto $SO(3)$. Therefore many exponential coor-
dinates map to one rotation matrix.
2.1.1.2 Versors

Unit quaternions provide a simpler way to represent rotations in three-dimensional space. Unit quaternions are also referred to as versors [33], and they provide a more efficient representation than rotation matrices. Exponential coordinates can be represented as a corresponding versor. A versor has two components:

\[ q_0 \in \mathbb{R} \text{ and } q \in \mathbb{R}^3, \]

which means a versor is a 4-dimensional vector, \( Q = (\mathbb{R}, \mathbb{R}^3) \) where \( ||Q|| = 1 \) such that,

\[ ||Q|| = \sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2} = 1 \text{ where } q = (q_1, q_2, q_3) \]

Given a rotational matrix represented as exponential coordinates, \( e^{\omega \theta} \), a versor can be calculated as:

\[ Q = (\cos(\frac{\theta}{2}), \omega \sin(\frac{\theta}{2})), \]

and given a versor, the rotational matrix in exponential coordinates is:

\[ \theta = 2\cos^{-1}q_0 \quad , \quad \omega = \begin{cases} \frac{q}{\sin(\frac{\theta}{2})} & \text{if } \theta \neq 0, \\ 0 & \text{otherwise}, \end{cases} \]

The strength of versors is that they do not deal with singularities where, for example, a singularity is faced when the rotation matrix is equal to the identity matrix because
any $\omega$ with $\theta = 0$ is equivalent to $R = I$. Therefore by representing orientations with versors, there is a smooth dependency to the rotation matrix $R = I$. The disadvantage of this representation is it deals with four parameters compared to two parameters from the exponential coordinates, but the advantage of this representation far outweighs the disadvantage.

2.1.1.3 Euler Angles

Euler Angles consist of three angles named yaw, pitch, and roll where the three angles are denoted as $(\phi, \theta, \psi)$. The roll angle is related with the x-axis, the pitch angle is related with the y-axis, and the yaw angle is related with the z-axis. These angles describe the orientation of a rigid body in three dimensional space. When rotation is done on each of the axes, in the correct order and after subsequent rotations, it can be described as a unique orientation that can be represented by $R_{AB}(\phi, \theta, \psi)$ where $R_{AB}$ represents the rotation matrix for frame B with respect to frame A. The three angles are denoted as ZYX Euler angles as shown in figure 2.5 and the following matrices represent the elementary rotation matrix for each corresponding
rotation angle:

\[
R_\phi = \begin{bmatrix}
1 & 0 & 0 \\
0 & cos\phi & -sin\phi \\
0 & sin\phi & cos\phi \\
\end{bmatrix}
\]

\[
R_\theta = \begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & -sin\theta & cos\theta \\
\end{bmatrix}
\]

\[
R_\psi = \begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1 \\
\end{bmatrix}
\]

These three elementary rotations describe the rotation with respect to a fixed frame for each corresponding axis. In order to describe the final orientation when all three axes have been rotated, the three elementary rotations can be multiplied, \( R_\phi R_\theta R_\psi \).

The final rotation matrix looks like the following:

\[
R_{\phi\theta\psi} = \begin{bmatrix}
c_{\phi\theta\psi} - s_{\phi\psi} & -c_{\phi\theta} s_{\psi} - s_{\phi} c_{\psi} & c_{\phi} s_{\theta} \\
s_{\phi\theta\psi} + c_{\phi\theta} s_{\psi} & -s_{\phi} c_{\theta} s_{\psi} + c_{\phi} \psi & s_{\phi} c_{\theta} \\
-s_{\phi} c_{\theta} & s_{\theta} s_{\psi} & c_{\theta} \\
\end{bmatrix}
\]

where \( c_a \) represents \( cos(a) \) and any concatenation of notation such as \( c_{ab} \) represents multiplication of cosines such as \( cos(a)cos(b) \) and the same goes for sine, \( s \). Other Euler angles exist such as Tait-Bryan angles, Helmholtz angles, and Fick angles. The difference between these angles are the ordering of the elementary rotations. Euler angles also suffer from singularities depending on the order of the rotations. In the case of \( ZYX \text{ Euler angles} \), the singularity exists in the identity rotation where any parameter satisfies the rotation; thus, losing the dependence as a function of the rotation matrix, \( R = I \).
2.1.2 Kinematic Parameterization

2.1.2.1 Twists

As rotation matrices, $R \in SO(3)$, can be represented as exponential coordinates, so can the homogeneous transformation matrix. The homogeneous transformation matrix, $T \in SE(3)$, can be represented with exponential coordinates, $e^{\hat{\xi}\theta}$. The variable $\xi$ is the twist coordinate $(v, \omega)$ for $\hat{\xi}$ which is a skew symmetric matrix such as $\hat{\omega}$ from section 2.1.1.1. To derive the twist coordinates, $\xi$, rigid motion needs to be defined. Rigid motion satisfies two properties: the distance between two particles never change which means the object in motion does not deform. Note that soft robots violate this first assumption since it has to go through deformation to actuate. The second assumption of rigid motions is that it preserves orientation between points. Therefore,

$$g(v \times w) = g(v) \times g(w) \forall v, w \in \mathbb{R}^3 \text{ and } g \in SE(3).$$

This means that orientation between particles never change. Now that rigid motion is defined, assume the tip point undergoes a rotation along an angular velocity vector,
\( \omega \) as shown in figure 2.4. Therefore the velocity of the tip point is,

\[
\dot{p}(t) = \omega \times (p(t) - q) \quad \text{where} \quad q \in \mathbb{R}^3.
\]

\( q \) is a point on the axis of the angular velocity vector \( \omega \). Since the cross product is distributive over subtraction. The equation then becomes,

\[
\dot{p}(t) = \omega \times p(t) - \omega \times q = \dot{\omega} p(t) - \omega \times q.
\]

Hence in matrix form, the same equation is

\[
\begin{bmatrix}
\dot{p} \\
0
\end{bmatrix} = \begin{bmatrix}
\dot{\omega} & -\omega \times q \\
0 & 0
\end{bmatrix} \begin{bmatrix}
p \\
1
\end{bmatrix} = \hat{\xi} \begin{bmatrix}
p \\
1
\end{bmatrix}.
\]

It is important to notice that \(-\omega \times q\) is the velocity of \( q \) in the translation axis of the vector \( \omega \),

\[
\hat{\xi} = \begin{bmatrix}
\dot{\omega} & v \\
0 & 0
\end{bmatrix}.
\]

As was done with the exponential coordinates in section 2.1.1.1, the equation above can be differentiated because it is separable. \( \xi = (v, \omega) \) is defined as the twists coordinate of \( \hat{\xi} \), and there is an exponential map, \( \hat{\xi} \mapsto SE(3) \).

2.1.2.2 Screws

Screws are a geometric interpretation of twists and provide additional information to facilitate the parameterization of rigid body motions. A general screw motion consists of three key components: an axis, \( l \), a pitch, \( h \), and a magnitude, \( M \). The pitch, \( h \), is defined as the ratio between the translation, \( d \), and rotation along an axis, \( \theta = M \). As can be seen in figure 2.6, \( q \) is a point on the axis \( l \), and any point along this axis can be described as,

\[
l = q + \lambda \omega : \lambda \in \mathbb{R},
\]
where \( \omega \) is the angular velocity vector as stated above. Therefore if there is a translation along this axis, then the translation can be described as,

\[
f(p) = q + h\theta \omega,
\]

since, \( h\theta = d = \lambda \). In pursuance of describing the entire screw motion, the rotation can be described as the rotation, \( e^{\hat{\omega} \theta} \), of vector, \( p - q \), where \( p \) is the resulting point after the rotation, \( e^{\hat{\omega} \theta} \),

\[
h(p) = e^{\hat{\omega} \theta}(p - q).
\]

Therefore the entire screw motion is the concatenation of the above equations,

\[
g(p) = f(p) + h(p).
\]

According to Chasles Theorem [32], all rigid body motions can be described by a translation along an axis followed by a rotation which can be described as the screw motion show in figure 2.6. Figure 2.6 also shows the two motors that produce these motions, prismatic and revolute joints; these two joints can describe any rigid body motion.
2.1.2.3 Denavit-Hartenberg Parameters

Denavit-Hartenberg parameters [34] are widely used in the robotic community because it reduces the SE(3) parameter space which includes \( p \in \mathbb{R}^3 \) and \((\phi, \theta, \psi)\) to only four parameters \( d, \theta, r, \) and \( \alpha \). These four parameters completely explain the SE(3) of a particular link. The following four parameters are described as follows:

- \( d \): is the distance between the current joint and the previous joint.
- \( \theta \): is the joint angle of the current joint.
- \( r \): is the radius of the link.
- \( \alpha \): is the link twist of the current link.

These parameters where a joint is denoted as \( j \), x-axis of the coordinate frame is denoted as \( X \), and z-axis of the coordinate frame is denoted as \( Z \) can also be described as,

- \( d_i \): is the distance between \( X_i \) and \( X_{i-1} \) along \( Z_i \). \( d_i \) is constant if the joint is revolute, and it is variant if the joint is prismatic,
- \( \theta_i \): is the angle between \( X_i \) and \( X_{i-1} \) about \( Z_i \). \( \theta_i \) is constant if it is a prismatic joint and variant if it is revolute joint,
- \( r_i \): is the distance between \( Z_i \) and \( Z_{i-1} \) along \( X_i \),
- \( \alpha_i \): is the angle between \( Z_i \) and \( Z_{i-1} \) about \( X_i \).

There are also constraints to satisfy DH parameters:

- \( X_i \) has to be perpendicular to both \( Z_i \) and \( Z_{i-1} \),
• $X_i$ has to intersect both $Z_i$ and $Z_{i-1}$,

• the origin of $j_i$ is at the intersection of $X_i$ and $Z_i$,

• the coordinate frame is right handed.

Any robot model that violates these constraints will not be able to use this parameterization. It is also important to note that there is a relationship between exponential coordinates and Denavit-Hartenberg coordinates. Details about how this relationship can be derived can be found in [32].

Now that we have defined some kinematic parameterizations that can help us model the motion of the robot, we can go ahead and discuss how we can use these parameterizations to generate motions based off the position or angle of the robot’s links.

### 2.1.3 Forward Kinematics

Forward kinematics describes the relationship between joint rotation and the position of the robot in three dimensional space. It is termed forwards because it is a mapping from joint angles to a new position of the robot in space. Therefore the map is as follows,

$$f : Q \rightarrow SE(3) \quad \text{where} \quad Q \in \{\theta_1, \theta_2, \ldots, \theta_{dof}\},$$

where $f$ is the forward kinematics map. Therefore given a joint position, the robot can compute the resulting position of its end-effector ($eef_i$) by using the forward kinematics map. A general formula that satisfies this map is

$$eef_{origin} = eef_i T_i(\theta_1)^t T_{i-1}(\theta_2) \ldots T_{-n+1} T_{origin}(\theta_n).$$
Murray et al. [32] state that an equivalent map is the product of exponentials formula, which is the product of exponential twists since $\text{eef}_T_{\text{origin}} = e^{\hat{\mathbf{k}}\mathbf{\theta}} \text{eef}_T_{\text{origin}}(0)$.

Forward kinematics is easy to compute because it is a straightforward product of the joints that have changed with the initial configuration of the robot. The complexity of the final transformation can be simplified by the initial configuration of the robot. The simplest initial configuration of the robot is zero value for all joints.

Forward kinematics is very useful if given the rotation of the joints, but normally, we want the robot to move its joints to a desired position. Hence if the robot needs to move to a desired position, how do we calculate the joint values? This will be discussed in the next section.

2.1.4 Inverse Kinematics

Inverse kinematics is used when the robot has the position of its end-effector given and the joints values need to be computed. There are various of ways to do this and the complexity increases as the design of the robot increases. There are two ways to compute the joint values given the desired position: analytically and numerically. Analytic solutions can be as easy as using trigonometric properties of the robot, and other methods such as Paden-Kahan subproblems [32] that use a geometric algorithm based of the product of exponentials formula can be used. In order to solve the system numerically, approaches such as Newton’s method [35] can be used to minimize the error of the current position with respect to the desired position.
2.2 Stability

Stability in robotics is very important to do any task. Stability refers to a robot being able to stand in its leg without falling or wobbling. Since the robot lives in a physical world, it is constrained by the laws of physics. Therefore in order to conduct any task, in this case a kick, the robot needs to first know the equilibrium position in needs to be in. An equilibrium position is a position in the configuration of the robot where the robot executes a desired motion without any undesired external forces. There are two stability type: static and dynamic stability [31].

2.2.1 Static Stability

In order for a humanoid robot to balance, it has to maintain contact with the ground. When the robot is at rest, the Center of Mass (CoM) has to be inside the support polygon. The CoM is defined by

\[ CoM = \frac{m_j c_j}{M}, \]

where \((m_j, c_j)\) is the weight and position vector respectfully of the \(j^{th}\) link, and \(M = \sum m_j\). The support polygon is the area touched by the robot and the ground. When the robot is kicking, the area touched by the robot in the ground is the supporting foot. The support polygon is the convex hull of the foot touching the ground and it is defined by

\[ CoS = \left\{ \sum \alpha_j p_j | \alpha_j \geq 0, \sum \alpha_j = 1, p_j \in S(j = 1, \ldots, N) \right\}, \]

where \(S(j = 1, \ldots, N)\) is the set of edges of the supporting foot. When a robot is standing upright and does not actively balance on its legs, then the robot is statically
stable. Usually these robots have more than two feet since their support polygon is bigger than a humanoid. In order for a robot to walk while being statically stable, the center of mass needs to be inside the polygon of support and the legs of the robot may not lose contact with the ground. Usually wheeled robots can achieve this, but other robots may need to lose contact with the ground. These robots can change to dynamic stability to walk if a leg needs to be lifted. Figure 2.7 shows an example of a robot that is always statically stable. Although the locomotion of this robot might be easier, other problems are still present such as Natural Language Processing and Human-Robot Interaction. RoboCup Nagoya 2017 introduced a new league for home scenarios using the Pepper platform [7].
2.2.2 Dynamic Stability

Dynamic stability refers to a robot actively balancing while standing upright. For example, humans are always actively balancing while standing by using their muscles and bones. Humanoid robotics have a small support polygon relative to their body’s height which makes it hard to balance on two feet. Dynamic stability requires a model of the physical world to predict how the system should react to external forces to balance. In the case of humanoid robotics or any biped robot, an inverted pendulum model is the closest approximation to the dynamics of an upright biped robot because the CoM is relatively high. For dynamic stability, the CoM is not guaranteed to stay within the support polygon because any motion can accelerate the CoM from the support polygon. Hence, controllers are needed to maintain the robot in an upright position while standing. In order to maintain contact with the ground, the external forces of the robot needs to be counteracted to avoid forces such as the force of gravity to pull the robot down. Therefore, the normal force of the robot which is at the support polygon to the center of mass needs to be within the polygon of support when projected down. The gravity vector can be denoted as \( V_G \) and the normal force can be denoted as \( V_N \). Hence if their cross product is equal to zero or close to an amount \( \epsilon \), then these two forces cancel out and the robot is in equilibrium, but if the cross product of these two forces is higher than an amount \( x \), then the robot can lose contact with the ground and fall. Therefore the stability of the robot can be defined by

\[
\psi_R = \begin{cases} 
\|V_G \times V_N\| \leq \epsilon, & \text{statically stable} \\
\text{else unstable} 
\end{cases}
\]
where $\epsilon$ depends on the radius of the support polygon. Stability is pivotal to execute motions successfully without the robot falling. In the next section, we will discuss different types of motions and how they are executed.

2.3 Motions

In robotics, motion is the process of moving a link or links of a robot from one place to another in order to achieve an objective. When moving the links of a robot, the motion of the robot might shift the weight distribution to an unstable position. When the robot compensates for the instability and shifts the weight to a stable configuration, then the robot generates dynamic motions. Otherwise if the robot has a predefined set of motions, then the robot generates static motions.

2.3.1 Static Motions

Static motions are any type of motions that have been generated in advanced and executed in the environment multiple times to execute a task. An example of a static motion might be an AI character in a video game. Another example of a static motion is a robot producing a get-up motion to stand up after falling down. When these robots do not consider the environment around them, it is very hard to be interactive with the environment and it is also unstable. For example if a robot is walking with a static motion and does not consider an applied force from the outside, the robot is likely to fall down. Usually static motions are produced with key frame animations such as the software from Aldebaran named Choreographe [8] illustrated in figure 2.8. These animation software slices time into time intervals and at each interval, the robot is programmed to produce a frame of motion designed by a robot.
designer. Static walks can also be produced by carefully programming each step. The issue with this is that the walk needs to be slow to decrease the acceleration of the center of mass as much as possible to avoid the center of mass leaving the support polygon so it can be stable. Therefore a statically stable walk is energy deficient because it does not take advantage of any applied force. A static walk also cancels out any force applied to the robot and creates a walk motion that keeps the CoM inside the support polygon. Dynamic walks are much faster and also perceives the environment around to adjust the robot accordingly.

2.3.2 Dynamic Motion

Dynamic motions are motions that have not been preprogrammed and are interactive with the environment. As illustrated in figure 2.9, a dynamic motion generator receives feedback from perceptors. The information from the perceptors can then be filtered, and a model of the robot and the physical world is needed to understand how the robot is affected. These models do not have all the nonlinear equations of motions to understand how the robot is behaving in the world because it is not feasible, but rather these models are simplified to a reasonable degree. In the case of humanoid
robots, the model is the inverted pendulum. Therefore, the data from the sensors and the model is integrated and an error is calculated. The error is determined by the task being performed. For example if a robot is walking, the error is the model’s reference CoM and the perceived CoM, or if a robot is trying to grasp an object, the error will be the perceived robot arm and the object’s position in the world. When the error is calculated, then the controller tries to adjust the system by reducing the error and bring the actual output closer to the reference point. The system then receives this control input and communicates with the motion generator to produce a motion that satifies this control input. The motion is then executed using the actuators of the system, and the whole process starts all over again. This type of controller is a closed-feedback loop and it is essential for dynamic motion generation. Other dynamic motion generators use the structure of the robot and the external forces of the system to produce motion. One example of this is dynamic passive walkers.

By allowing the CoM to leave the support polygon, the robot can use the force given by gravity to generate motions. Passive Dynamic Walkers, as shown in figure 2.10, use gravity to actuate the links of the robot and generates a gait to walk forward.
by smart design of the structure. Without the help of any actuation or motors, the robot is able to move down an incline with natural walk movements.

With dynamic motions, the robot does not have to be careful to leave the CoM inside the support polygon and can let the CoM leave the support polygon. Humans always do this in sports and even while walking. By allowing gravity to push us forward and catching ourselves with the swinging leg, we use energy applied by external forces to produce motions. If we used static motions in the other hand, then we will cancel out all the external forces and walk with a low CoM which will waste energy and get people very tired. For the omnidirectional kick engine discussed in this paper, the balancing is done dynamically. The kick engine has limited knowledge of the motions is going to produce during the game, and when a kick is requested, the kick engine has to compute how the robot will balance and how much to shift the weight. The kick engine also executes many kicks, hence it also has to compensate for moments produced about the kick that might make the robot fall down. At RoboCup Nagoya 2017, a new artificial turf was introduced. The artificial turf has grass blades of 8mm which means air pockets are bound to form under the foot of the robot when standing upright. These air pockets cause nonlinear forces and makes the robot unstable when standing on one foot. For this reason, dynamic motions for kicks were nonexistent because of the difficulty. The difficulty of dynamic motions is controlling the CoM outside the support polygon and to achieve this task, we need another criterion that can stay within the support polygon: Zero Moment Point.
2.3.3 Centroidal Dynamics

Due to the complexity of the robot due to components such as non-linear forces applied on the robot’s links, the dynamics of the robot is very complicated. In order to simplify the dynamics and generate a model to help us understand how forces from the outside affect the overall motion trajectory of the robot, the structure of the robot is simplified to a 3D inverted pendulum [36] as shown in figure 2.11. The pendulum itself is considered to be the center of mass of the robot and it consists of a rod from the pendulum to the ground which is the overall motion of the robot. A system that is simplified to a clustered mass to study its dynamics is referred to as Centroidal Dynamics. Centroidal dynamics is a good approximation of the overall system, and it helps us derive an important equation to compute the Zero Moment Point discussed in section 3.1.1.
2.3.4 Interaction Dynamics

The most essential part of robotics is how the robot behaves in the physical world. It is also crucial to integrate the physical world in the design of a robot. For example if a robot needs to lift load, the robot designer needs to consider the torque output of the motors to be able to lift these loads in the physical world. Although the robot might be stable on its own, it can lose stability as soon as lifts a load if the mass is heavy. Also if a robot is grasping an object, the material of the object influences how an object should be grasped. Therefore without integrating the physical world, the robot is not able to interact with it. Hence, interaction dynamics helps with the analysis of interactions between a system and the physical world. The future of robotic applications will be environments where humans and robots cohabitate together. Therefore, mechanical interaction is pivotal to avoid danger being done to humans. Many controllers only consider the robot in isolation because the integration
of environment is difficult since the environment is dynamic. Especially if a human
is in the environment, it will be unpredictable. Mechanical impedance [37] can be
used to control interaction dynamics. Mechanical impedance comes from electrical
impedance where force is analogous to voltage and velocity is analogous to current.
Mechanical Impedance is the ratio of the input velocity and the output force, and it
is defined by
\[ Z = \frac{F}{v}, \]
where \( F \) is the output force and \( v \) is the input velocity. The impedance is considered
a dynamic extension of stiffness where stiffness is defined by
\[ k = \frac{F}{\delta}, \]
where \( F \) is the output force and \( \delta \) is the displacement of the object and the inverse of
stiffness is compliance. As more force is required to deform the object, the stiffness,
\( k \), is larger. Therefore by controlling the mechanical impedance, the output force
towards the environment will be controlled. When a robot manipulates a robot or
encounters an unknown object in the enviroment, the mechanical impedance should
be lower. In the case, where the robot needs to grab an object and stabilize it such as
a hand tool, the mechanical impedance needs to increase. Contrarily, when the robot
changes configuration while also handling an object, it has to do in such a way where
the mechanical impedance is not volatile. The inverse of mechanical impedance is
mechanical admittance and it is defined by
\[ Z^{-1} = \frac{v}{F}. \]
Henceforth, the environment acts as an admittance because the environment contains
inertial objects that output a force. For this reason, the robot’s output is treated as
a mechanical impedance because the environment acts like an admittance. Therefore as the environment outputs a force, the robot outputs a force back. Therefore to control the mechanical impedance and not damage the environment, the impedance should be stabilized and equal to the environment’s admittance.

2.3.4.1 Simple Impedance Control

A simple impedance controller [37] can be used to control the output impedance of the robot. This impedance controller does not control the actual physical impedance of the robot such as the impedance generated by mass or friction. Therefore the total impedance is the output impedance of the controller and the physical impedance. In order to derive the impedance of the controller, the dynamics of the robot will be explained first. If a multi-freedom robot is modeled, then the dynamics of the robot is modeled as,

\[ T_a + T_e = I(\Theta)\ddot{\Theta} + C(\dot{\Theta}, \Theta)\dot{\Theta} + N(\dot{\Theta}, \Theta), \]  

where \( T_a \) is the actuator torques given to the robot and \( T_e \) are the external torques given by the environment. \( I(\Theta) \) is the inertial matrix which is dependent on the robot configuration. \( C(\dot{\Theta}, \Theta) \) are the centrifugal and coriolis accelerations on the robot and \( N(\dot{\Theta}, \Theta) \) is the gravity vector. Hence, a simple control law that can control the output impedance of the robot is,

\[ T_a(\dot{\Theta}, \Theta) = K(\Theta_{ref} - \Theta) + B(\dot{\Theta}_{ref} - \dot{\Theta}). \]  

\( K \) is the stiffness coefficient and \( B \) is the damping coefficient. By combining equation 4 and 3, we get:

\[ K(\Theta_{ref}) + B(\dot{\Theta}_{ref} + T_e = I(\Theta)\ddot{\Theta} + C(\dot{\Theta}, \Theta)\dot{\Theta} + N(\dot{\Theta}, \Theta) + K(\Theta) + B(\dot{\Theta}). \]
As shown in figure 2.12, the applied torque resembles a damp-spring model where the impedance will drive the robot to a desired state. This controller is defined in the joint space and can be used to regulate the output impedance of the end-effector. The issue with the joint space, constant stiffness, and damping coefficients do not consider the configuration of the robot. Therefore, it can generate different impedance in varying configuration. Consequently if we define the controller in configuration space, then we can guarantee the desired impedance of the robot. In pursuance of controlling the configuration space, the kinematics of the robot needs to be defined. The robot’s actuators are defined by generalized coordinates, \( q = q_1, q_2, ..., q_k \), and the relationship between the actuators and the end-effector is

\[
\dot{x} = J\dot{q},
\]

where \( J \) is the manipulator Jacobian \( x \) is the end effector’s coordinates. With this in mind we can now define the work done by a system where

\[
W = F \cdot s,
\]

such that \( s \) is the displacement of the object and \( F \) is a force vector. Therefore:

\[
W = F \cdot s = \int \dot{x} \cdot F dt, 
\]

and since the work done by the end effector is the same work done by the actuators then,

\[
\int \dot{q} \cdot \tau dt = \int \dot{x} \cdot F dt, 
\]

but the integrals are equal and using equation 6, therefore,

\[
\dot{q} \cdot \tau = J\dot{q} \cdot F
\]
and eliminating $\dot{q}$ on both sides we get,

$$\tau = J^T F.$$  \hspace{1cm} (8)

According, the relationship between the force produced by the end effector and the actuator torques is stated in equation 8. By using 8, a control law in the configuration space can be introduced,

$$T_a(\dot{\Theta}, \Theta, x, \dot{x}) = K(x_{ref} - L_{map}(\Theta)) + B(\dot{x}_{ref} - L_{map}(\dot{\Theta})).$$  \hspace{1cm} (9)

This simple impedance controller does not account for the environmental torques imposed on the environment by the robot since we are only considering the output impedance and not the mechanical impedance. Henceforth, to control the environmental torque, we need a force feedback gain integrated to the controller such as the following,

$$T_a(\dot{\Theta}, \Theta, x, \dot{x}) = K(x_{ref} - L_{map}(\Theta)) + B(\dot{x}_{ref} - L_{map}(\dot{\Theta})) + K_f(T_e + K(x_{ref} - L_{map}(\Theta)) + B(\dot{x}_{ref} - L_{map}(\dot{\Theta}))).$$  \hspace{1cm} (10)

where $K_f$ and $T_e$ are the force feedback gain and the environmental torque, respectively. By verification of equation 10, if the environmental torque is zero, the equation then becomes equation 9. This is the case when the robot is standing still. In other words, when the robot is standing still, equation 9 does a good job in keeping the robot stable. If equation 10 and 3 are combined, then the result is

$$T_e = \frac{l(\Theta)}{1+K_f} + \frac{C(\dot{\Theta}, \Theta)\ddot{\Theta}}{1+K_f} + \frac{N(\dot{\Theta}, \Theta)}{1+K_f} + K(L_{map}(\Theta) - x_{ref}) + B(L_{map}(\dot{\Theta}) - \dot{x}_{ref}).$$  \hspace{1cm} (11)

The controlled motion equation 11 controls the forces on the robot by $1 + K_f$ to get the desired torque. The full explanation and proofs of section 2.3.4.1 can be found in Hogan & Buerger [38]. This controller can be very useful for the application discussed.
in this paper: omnidirectional kick engine. The reason for its usefulness is the force control that can be imposed on the supporting foot. By being able to control the environmental forces, the robot can control the amount friction under the sole of the supporting foot to avoid falling about the supporting foot. For example if a robot is generating moment about the x axis, i.e. the robot is falling sideways, the robot can generate the same moment in the opposite direction to cancel the force. This can be done by solving for $x_{ref}$ in equation 11.

Interaction control is very useful because it does not treat the system in isolation from the environment. Although the environment is unpredictable, the interaction control can account for present forces and moments and generate motion trajectories that will be safe and stable.
CHAPTER 3

Approach

3.1 ZMP based Preview Controller

This section describes how the robot stabilizes on the supporting foot. A ZMP based Preview Controller is used to keep the Zero Moment Point (ZMP) in the support polygon. The first section describes the definition of ZMP and how it is derived. The following section will illustrate how the Preview Controller uses the ZMP to stabilize the robot. The full explanation and proofs of section 3.1 can be found in Kajita, Hirukawa, Harada, & Yokoi [35].

3.1.1 Zero Moment Point

When the robot is at rest, the CoM criterion is enough to stabilize the robot, but if the robot is in motion, the CoM might leave the support polygon. In that case,
we can use another criterion to verify the supporting foot still has contact with the ground.

The dynamic criterion for the robot to maintain contact with the ground is the ZMP [39]. In order to correctly balance the robot, roboticists use ZMP to keep the robot from falling. The ZMP is the point where the robot’s contact point with the ground does not generate a moment about the y and x axis. This is defined by

\[ ZMP = \{ p_o \times f + \tau = 0 | \tau_x, \tau_y = 0 \} \]  

(12)

Therefore the robot does not rotate about these axes and tip over. Notice that the ZMP criterion does not say anything about the moment about the z axis. This rotation is allowed in the ZMP but it does not fall but rotate about the z axis. In order to derive the ZMP, the x and y components of (12) need to be solved and approximated by discretized points:

\[ ZMP_{approx.} = \{ (p_i - p_o) \times f_i + \tau_i = 0 | \tau_{ix}, \tau_{iy} = 0 \} \]

Therefore,

\[ p_x = \frac{\sum_{i=1}^{N} \tau_{iy} - (p_{iz} - p_z)f_{ix} + p_{ix}f_{iz}}{\sum_{i=1}^{N} f_{iz}} \]  

and

(13)

\[ p_y = \frac{\sum_{i=1}^{N} \tau_{ix} - (p_{iz} - p_z)f_{iy} + p_{iy}f_{iz}}{\sum_{i=1}^{N} f_{iz}} \]

(14)

This is a very simple model and the full dynamics of the physical world cannot be captured. Hence, a Kalman filter [40] was used to add noise to the model. The measurement model used for the Kalman filter was the foot sensors from the NAO robot. The data from the foot sensors were used to solve for (13) and (14). The
foot sensors give us values for the normal ground force exerted at the contact points. Therefore all the terms in (13) and (14) except $f_{iz}$ are zero, resulting in

$$p_x = \frac{\sum_{i=1}^{N} p_{ix} f_{iz}}{\sum_{i=1}^{N} f_{iz}}, \quad p_y = \frac{\sum_{i=1}^{N} p_{iy} f_{iz}}{\sum_{i=1}^{N} f_{iz}}.$$  \hspace{1cm} (15)

The NAO robot has four sensors in the each foot, and it can be used to measure the magnitude of the force. Using (15) and the magnitude and position of each sensor, the empirical ZMP can be computed. For the belief model of the Kalman filter, we use (12) and we approximate the ZMP using a linear inverted pendulum [35,36]. The dynamics of the inverted pendulum is well understood in physics and it is used to model the dynamics of balancing for the humanoid robot. The top of the pendulum is assumed to be the CoM of the robot and the bottom part is assumed to be negligible. Therefore, we need to assume that the legs of the robot do not weigh much compared to the center of mass of the robot. Therefore, the linear and angular momentum of the center of mass is

$$\mathcal{P} = M \dot{\mathbf{c}}, \quad \mathcal{L} = \mathbf{c} \times M \dot{\mathbf{c}}.$$  \hspace{1cm} (16)

To compute the approximated ZMP, we use (12) and (16) and get,

$$p_x = \frac{(z - p_z) \ddot{x}}{\ddot{z} + g}, \quad p_y = \frac{(z - p_z) \ddot{y}}{\ddot{z} + g}.$$  \hspace{1cm} (17)

With a belief and measurement model, we can get a close approximation to the ZMP. We can now use a controller that will control the ZMP to maintain it in the polygon of the supporting foot. This will be discussed in the next section.


3.1.2 Preview Controller

The ZMP Preview Controller is based on the cart table model and it is a closed-loop system [35, 41, 42]. This model assumes that given the cart on the table, the Center of Pressure (CoP) (foot of the table) is affected, and in turn, affects the stability of the table (ZMP is affected). The ZMP Preview Controller is based on future ZMP positions of the robot. Since the robot is not walking, it is assumed that the robot will have the same future preview gains. This is an important simplification because it helps with the computation time. The Preview Controller uses a Linear Quadratic Regulator (LQR) to find the optimal values of the gains,

\[
J = \sum_{j=1}^{\infty} \left\{ Q (p_{j}^{ref} - p_{j})^2 + Ru^2 j \right\} .
\]

where R and Q are weights. The LQR tries to solve a Riccati differential equation for the optimal values,

\[
P = A^T PA + c^T Q c - A^T Pb (R + b^T Pb)^{-1} b^T PA
\]

where,

\[
A = \begin{bmatrix} 1 & \delta t & \delta t^2 / 2 \\ 0 & 1 & \delta t \\ 0 & 0 & 1 \end{bmatrix}, \quad b = \begin{bmatrix} \delta t^3 / 6 \\ \delta t^2 / 2 \\ \delta t \end{bmatrix}, \quad c = \begin{bmatrix} 1 & 0 & -z/g \end{bmatrix},
\]

and tries to minimize the cost function J by using the input and K such that,

\[
u_k = -K x_k + \begin{bmatrix} f_1 & f_2 & \ldots & f_N \end{bmatrix} \begin{bmatrix} p_{k+1}^{ref} \\ \vdots \\ p_{k+N}^{ref} \end{bmatrix}
\]

where

\[
K = (R + b^T Pb)^{-1} b^T PA
\]

\[
f_i = (R + b^T Pb)^{-1} b^T (A - bK)^T s(i-1) c^T Q
\]
This calculation is only done once and it is used for the Preview Controller. When all the optimal values have been calculated, the state transition equation can be used to compute the next CoM position, velocity and acceleration,

\[ x_{k+1} = Ax_k + bu_k. \]

It is important to note that \( p^{ref} = \{ p^{ref}_{k+1}, \ldots, p^{ref}_{k+N} \} \) will always be the same because \( p^{ref} \) will always be in the supporting foot of the kick and it will not move. Therefore, it will save computation time since \( p^{ref} \) is a constant when computing \( u_k \). When \( u_k \) is computed and we find the next CoM, then we need to move the CoM to next position. This will be discussed in chapter 3.3.

### 3.2 Simulated CoM Trajectory

When the robot is about to generate a kick trajectory, we can assume the center of mass of the robot does not generate any acceleration in the kicking trajectory. The 3D linear inverted pendulum has an equation defined as [35],

\[ \ddot{x} = \left( \frac{g}{z_c} \right) (x - p), \]

where \( x \) is the center of mass position of the robot and \( p \) is the ZMP. The height of the robot is constrained to \( z_c \), and \( g \) is the gravity vector. If the center of mass does not generate an acceleration in any axis then, the equation becomes

\[ \frac{g}{z_c} x = \frac{g}{z_c} p, \]

and therefore \( x = p \), which means the center of mass equals to the ZMP. As can be observed in figure 3.3, the acceleration of the center of mass is very close to zero.
bounded by $-1.5 \times 10^{-5} \leq \ddot{x} \leq 2 \times 10^{-5}$. Therefore, it is fair to assume the center of mass of the robot is not accelerating. With this assumption, we can use the center of mass to generate a trajectory that does not use the Zero Moment Point. This has advantages such as less numerical errors by not having to compute the ZMP equation every cycle. We can also use the CoM to simulate the trajectory of the robot before the kick trajectory is generated and quickly move to the side while kicking. This is done by moving the center of mass towards the sole of the support foot. Therefore the equation to minimize the error of the current position of the center of mass and the optimal center of mass position is,

$$\text{com} - \text{com}_{\text{opt}} = 0.$$

We can then use Newton-Rhapson method [43] to minimize the error. It is important to notice that this process is simulated and the robot only moves when the optimal center of mass position is found. Algorithm 1 is an iterative algorithm that uses Newton-Rhapson method to find the optimal center of mass. It starts with an initial

\begin{algorithm}
\begin{algorithmic}
\Procedure{SimulateCoMTrajectory}{q, \text{com}_{\text{opt}}}
\State $\text{iterations} \leftarrow 0.$
\State $\text{com}_{\text{cur}} \leftarrow \text{ComputeCoM}(q)$.
\State $\text{error} \leftarrow \text{com}_{\text{cur}} - \text{com}_{\text{opt}}$.
\Loop
\If{$\text{error} > \epsilon$ and $\text{iterations} < \text{limit}$}
\State $\text{J}^{-1} \leftarrow \text{ComputePseudoInverse}()$.
\State $q \leftarrow q - \text{J}^{-1} \text{com}_{\text{cur}}$.
\State $\text{com}_{\text{cur}} \leftarrow \text{ComputeCoM}(q)$.
\State $\text{error} \leftarrow \text{com}_{\text{cur}} - \text{com}_{\text{opt}}$.
\State $\text{iterations} \leftarrow \text{iterations} + 1$.
\EndIf
\EndLoop
\State $\text{Stop}$.
\EndProcedure
\end{algorithmic}
\end{algorithm}
set of joint angles and an optimal set of joint angles. In the loop of the algorithm, the pseudoinverse of the Jacobian matrix is needed to find the gradient of the function assuming it is multivariate, which in this case it is, and it is used to get closer to the solution. This is done until we are close enough to the roots of the equation or we hit the upper limit of the iterations allowed. When the loop is terminated, we assume we are close to the optimal center of mass position, and the controller moves the robot to this configuration where the robot reaches a stable state. We can also calculate the joint angle positions analytically to reach the optimal center of mass position which will be discussed next in section 3.3.
Figure 3.4: An approximation of the center of mass kinematic for inverse kinematic model. $l_1$ is the length from the ankle to the hip and $l_2$ is the length from the hip to the center of mass. This is approximation is done to the frontal plane and sagittal plane. The hip pitch ($\theta_{2,pitch}$), hip roll ($\theta_{2,roll}$), ankle pitch ($\theta_{1,pitch}$) and ankle roll ($\theta_{1,roll}$) are adjusted according to the inverse kinematic model.

3.3 Analytical Approximation for Inverse Kinematics

In order to move the CoM to the position that was given by the Preview Controller, an analytical approximation using inverse kinematics can be used to move the joints to the correct position. Since the CoM is not directly actuated by a specific joint, an analytical approximation can be used. The position of the CoM can be affected by the leg joints. Let’s assume the CoM is above the leg joints, then we can approximate the CoM using the hip joints and ankle joints. In the frontal plane, the CoM can be affected by the hip and ankle roll. If we approximate the joints and the CoM using a two-link arm \cite{32}, then we can use an analytical solution to move the CoM. The first link of the arm is the ankle roll joint to the hip roll joint. The second arm is the hip roll joint to the CoM. The link of the first and second arm are revolute joints, and the end effector which is the CoM does not have a joint. The length of the first arm is the distance from the ankle roll to the hip roll, $l_1$ and the length of the second arm, $l_2$, is the distance from the hip roll to the CoM. In the frontal plane, we want our end-effector to get to $(y, z)$. $Y$ is the difference between the next CoM’s $y$ position.
and the current CoM’s $y$ position. $Z$ is the plane that CoM is constrained on. Now, that the problem is defined, we can use these values to get the two angles we need, but we first need to get the range from the first link to the next CoM position in the frontal plane, $(y, z)$:

$$r = \sqrt{(y - \alpha_{\text{ankle},y}) + (z - \alpha_{\text{ankle},z})}. \quad (18)$$

With $r$, the second angle, $\theta_2$, can be found. But before computing $\theta_2$, we need to find the angle $\alpha$ that corresponds to the angle formed by $l_1$, $l_2$, and $r$. Therefore, we can use the cosine law to get the following:

$$\alpha = \cos^{-1} \frac{l_1^2 + l_2^2 - r^2}{2l_1l_2}, \quad \theta_2 = \alpha \pm \pi. \quad (19)$$

Therefore there are two solutions for $\theta_2$ but only one can give us the correct solution because of the balancing constraint placed on the robot. Hence the correct solution is as follows

$$\theta_2 = \begin{cases} 
\alpha - \pi, & \text{if right kick} \\
\alpha + \pi, & \text{else left kick} 
\end{cases}.$$ 

To compute $\theta_1$, the angle opposite of $l_2$, $\beta$, needs to be calculated first. To calculate $\beta$, the law of cosines can be reused and we end up with

$$\beta = \cos^{-1} \frac{r^2 + l_1^2 - l_2^2}{2l_1r}. \quad (20)$$

$\beta$ gives the angle in the triangle, but the $\theta_1$ is the angle between $y$-axis and $l_1$. To get this angle, the bearing of the first link to the the next CoM position is needed,

$$\arctan 2(z, y), \quad \theta_1 = \arctan 2(z, y) \pm \beta. \quad (21)$$
Figure 3.5: The left polynomials are the cubic splines. As shown in the figure, two polynomials need to be defined with a knot referred to as the via-point. The middle polynomial is referred to as a sextic (six degree polynomial). The six degree polynomial allows definition of a via-point without adding another polynomial. The right polynomial is a cubic Hermite spline which guarantees the control points to be tangent to the tangent vectors creating smooth trajectories for the kick. In this case, no via-point is defined unless a cubic Hermite spline is chained with another.

There are also two solutions for $\theta_1$, and a condition is given to decide which one to use

$$\theta_1 = \begin{cases} 
\arctan 2(z, y) - \beta, & \text{if right kick} \\
\arctan 2(z, y) + \beta, & \text{else left kick}
\end{cases}$$

The same needs to be done in the sagittal plane. The only difference in the sagittal plane, is that instead of hip and ankle roll, hip and ankle pitch will be used, and the trigonometric solution is done for point $(x, z)$. When both calculations are done for the frontal and sagittal plane, the current CoM position can now be inferred from these four joint angles.

### 3.4 Kick Trajectory

#### 3.4.1 Kick Trajectory using Cubic Spline

In order to generate kick trajectory, the most trivial case is to change the leg joint angles until a desired kick configuration has been reached. This is tedious work and will be suboptimal since the joint space is very large. Another approach is to use optimization to find key-frame values, but this only works for a set of kicks and it is
not dynamic enough to create any kick trajectory. Therefore, a more efficient solution is to generate motions using polynomials [44]. Polynomials are a great solution in robot motion because they can be configured to generate smooth curves. To generate a polynomial for the kick motion requires constraints such that the motion generated by the polynomial does not conflict with any unwanted configuration. The polynomials will not be used in the joint space, but rather will be used to determine the next position of the swinging foot. When the position of the foot is determined by the cubic polynomial, the inverse kinematic module will provide the angles for the foot position requested. For the kick motion, two cubic polynomials are generated. The point where the two polynomials meet is called the via-point. The purpose of this point will be discussed later. The cubic polynomial is as follows,

\[ \alpha_1(t) = a_{13}t^3 + a_{12}t^2 + a_{11}t + a_{10}, \quad \alpha_2(t) = a_{23}t^3 + a_{22}t^2 + a_{21}t + a_{20}. \]

In order to generate an arbitrary motion, specific constraints need to be put upon the polynomials. Since there are two cubic polynomials (i.e. eight coefficients / DOF), there are eight constraints. The first constraint is the point of the first polynomial at \( t = 0 \). At \( t = 0 \), the kick motion will swing the leg back. This is called the retraction point. This is the point farthest from the ball. The second constraint is that the velocity of the first point at \( t = 0 \) which is zero. The third constraint is the the position of the via-point where both cubic polynomials meet. The via-point is used to determine the height of the kick trajectory. It is also a very important point because it is where both polynomials meet. Hence, both polynomials need to have the same position at this point and their velocities need to match. Moreover, the acceleration at the via-point for both polynomials need to also match. This guarantees a smooth trajectory with \( C^2 \) continuity. The last two constraints define
Figure 3.6: These two polynomials are generated for a front kick. Since the front kick does not vary on the y-plane, it is two dimensional in the x and z plane. The white circles are the beginning and end points of the kick respectively. The black circle is the via-point that constrains the height of the polynomial. The first polynomial is from the first white circle to the black circle, and the second polynomial is from the black circle to the last white circle.

the position and velocity of the second cubic spline at $t = t_f$. The constraints are summarized as follows,

$$
\begin{align*}
\alpha_1(0) &= [x_0, y_0, z_0] & \text{(retraction point)} \\
\dot{\alpha}_1(0) &= [0, 0, 0] \\
\alpha_1(t_{\text{via}}) &= [x_{t_{\text{via}}}, y_{t_{\text{via}}}, h_{\text{floor}}] & \text{(constraint on foot from floor)} \\
\alpha_2(0) &= [x_{t_{\text{via}}}, y_{t_{\text{via}}}, h_{\text{floor}}] & \text{(constraint on foot from floor)} \\
\dot{\alpha}_1(t_{\text{via}}) &= \dot{\alpha}_2(0) \\
\ddot{\alpha}_1(t_{\text{via}}) &= \ddot{\alpha}_2(0) \\
\alpha_2(t_f) &= [x_f, y_f, z_f] & \text{(contact point)} \\
\dot{\alpha}_2(t_f) &= [0, 0, 0]
\end{align*}
$$

Moreover, as shown above, the constraints are defined as vectors because there needs to be polynomials for the $x$, $y$, and $z$ plane. The six cubic polynomials (two for each plane) will form a parametric curve in $\mathbb{R}^3$. To solve the coefficients of the poly-
nominals we solve the following system which was created by inputting the constraint in the polynomial and rearranging terms:

\[
\begin{bmatrix}
1 & t_{\text{via}} & 0 & 0 & 0 \\
0 & 0 & 1 & t_f & t_f^2 \\
0 & 0 & 1 & 2t_f & 3t_f^2 \\
2t_{\text{via}} & 3t_{\text{via}}^2 & -1 & 0 & 0 \\
2 & 6t_{\text{via}} & 0 & -2 & 0
\end{bmatrix}
\begin{bmatrix}
a_{12} \\
a_{13} \\
a_{21} \\
a_{22} \\
a_{23}
\end{bmatrix}
= 
\begin{bmatrix}
\frac{a_1(t_{\text{via}}) - a_1(0)}{t_{\text{via}}} \\
\frac{a_2(t_f) - a_2(t_{\text{via}})}{t_{\text{via}}^2} \\
0 \\
0 \\
0
\end{bmatrix}.
\tag{22}
\]

As seen in (22), \(a_{10}, a_{11},\) and \(a_{20}\) do not need to be a part of the system of equations because their answer is trivial:

\[
\alpha_1(0) = a_{10}, \quad \dot{\alpha}_1(0) = a_{11}, \quad \alpha_2(0) = a_{20}.
\]

Solving (22), gives us the coefficients for our polynomials, but we still are missing one step. Before we begin solving (22), we need to determine the optimal via-point position. This is discussed in section 5.2.

### 3.4.2 Kick Trajectory using Cubic Hermite Spline

The cubic Hermite spline was also used to generate kick trajectories. Every piece of the cubic Hermite spline is a polynomial of three degree. The cubic Hermite spline is specified by its end points and the tangents of those points (i.e. derivatives at the end points). The cubic Hermite spline is a smooth continuous function and to interpolate the Hermite polynomial, the following Hermite polynomial is defined as

\[
p(t) = (2t^3 - 3t^2 + 1)p_0 + (t^3 - 2t^2 + t)m_0 + (-2t^3 + 3t^2)p_1 + (t^3 - t^2)m_1.
\]

This equation can be used to interpolate the foot of the robot through the kick trajectory. The above interpolator is for unit intervals. Therefore, \(0 \leq t \leq 1\) and
\( p(t) \) is the position of the foot at time \( t \). \( p_0 \) and \( p_1 \) are the end points of the kick trajectory, and \( m_0 \) and \( m_1 \) are the derivatives of the end points. The velocity of the foot at time \( t \) can be defined as the derivative of \( p(t) \):

\[
\dot{p}(t) = (6t^2 - 6t)p_0 + (3t^2 - 4t + 1)m_0 + (-6t^2 + 6t)p_1 + (3t^2 - 2t)m_1.
\]

### 3.4.3 Kick Trajectory using Sextic Polynomial

Using sextic polynomials as a kick interpolator lets us define seven constraints due to the seven coefficients in the polynomial. The sextic polynomial is defined as:

\[
\alpha(t) = a_6 t^6 + a_5 t^5 + a_4 t^4 + a_3 t^3 + a_2 t^2 + a_1 t + a_0.
\]

The constraints for the kick trajectory are the following,

\[
\begin{align*}
\alpha(0) &= [x_0, y_0, z_0] \quad \text{(retraction point)} \\
\dot{\alpha}(0) &= [\dot{x}_0, \dot{y}_0, \dot{z}_0] \\
\ddot{\alpha}(0) &= [\ddot{x}_0, \ddot{y}_0, \ddot{z}_0] \\
\alpha(t_{via}) &= [x_{t_{via}}, y_{t_{via}}, h_{floor}] \quad \text{(constraint on foot from floor).} \\
\alpha(t_f) &= [x_f, y_f, z_f] \quad \text{(contact point)} \\
\dot{\alpha}(t_f) &= [\dot{x}_f, \dot{y}_f, \dot{z}_f] \\
\ddot{\alpha}(t_f) &= [\ddot{x}_f, \ddot{y}_f, \ddot{z}_f]
\end{align*}
\]

The seven constraints of the kick trajectory are used to control the position, velocity, and acceleration of the kick. It is also used to control the height of the foot from the ground at the via-point. As seen, there is still a notion of the via-point in the sextic polynomial but a knot where two polynomials meet does not exist. The
via-point is only used to control the swing shape of the kick trajectory. With these seven constraints, the linear system can now be solved to obtain a polynomial that is able to generate the kick trajectory desired,

$$
\begin{bmatrix}
1 & t_{\text{via}} & t_{\text{via}}^2 & t_{\text{via}}^3 \\
1 & t_f & t_f^2 & t_f^3 \\
3 & 4t_f & 5t_f^2 & 6t_f^3 \\
6 & 12t_f & 20t_f^2 & 30t_f^3
\end{bmatrix}
\begin{bmatrix}
a_3 \\
a_4 \\
a_5 \\
a_6
\end{bmatrix} = 
\begin{bmatrix}
\frac{\alpha(t_{\text{via}}) - \alpha(0)}{t_{\text{via}}^3} \\
\frac{\alpha(t_f) - \alpha(t_{\text{via}})}{t_f^3} \\
0 \\
0
\end{bmatrix}.
$$

(23)

The velocity of the position of the foot is defined as the derivative of the polynomial,

$$
\dot{\alpha}(t) = 6a_6t^5 + 5a_5t^4 + 4a_4t^3 + 3a_3t^2 + 2a_2t + a_1.
$$

All three trajectory polynomials can be seen in figure 3.5.

### 3.5 Rigid-Body Simulators

Rigid-body simulators simulate rigid-body dynamics which are the movement of interconnected rigid links under an external force. Robotics naturally fall under rigid-body dynamics, and there are simulators that help with the simulation of robots. Simulation of robotics is very important because it helps with the design process of robots by allowing robot designers to design parts and test them before fabrication. It also allows for quick prototyping of robots, and most importantly, simulators help with designs of motion controllers. Due to the constraints of the physical world, physical robots can not be used for long before it starts to deteriorate performance due to batteries, motor temperatures, and plastic stress under motion. Due to these factors, it is hard to test any motions on the physical robot for long periods of time.
Therefore, by having simulators that can help with robot motions is very essential to do any type of learning that requires large amounts of data. The only issue with simulators is the complexity of rigid-body dynamics. Due to nonlinear forces on the robot applied by ground frictional forces, it is very hard to simulate the real world. Robots that are attached to the floor are easier to simulate, but robots that are free-floating (robots that do not have a fixed base) such as humanoid robots are very hard to simulate. Simulating physical robots accurately is still an open research problem and requires heavy computation. Therefore using the simulator is only to have a rough idea of how the motion will execute in the physical world. Simulators use many rigid-body dynamic engines such as ODE [45], Bullet [46], Simbody [47], and Dart [48] for their simulations. Although making the simulator as accurate as possible is still an open question, engineers and scientists still use simulators to train their controllers. NAOs from SoftBank Robotics provide a simulator sdk in order to communicate with NAOqi which is the middleware between the simulated agent and
Figure 3.8: An overview of the communication between the simulator and the RoboCanes agent

the RoboCanes agent. HAL [2] is the program that simulates the physical hardware for the simulated agent. An overview of the framework is provided in figure 3.7. In order to simulate the correct NAO, HAL module provides various XML files describing the structure of the robot, sensors, and actuators of the robot. Therefore in order to use the correct robot in the simulation, the simulator needs to provide an XML file of the robot when instantiating HAL. HAL is then able to simulate the physical hardware of the robot and communicates with NAOqi. It is important to note that NAOqi does not notice the difference between a physical NAO and a virtual NAO because of simulation of HAL. In order to communicate with NAOqi, the simulator shares memory with HAL, and then HAL communicates with NAOqi. NAOqi shares memory with the RoboCanes agent and it updates the joint angles given by NAOqi. When the RoboCanes agent finishes updating joint angles, it sends a request back to NAOqi and the cycle begins again. The RoboCanes group uses various simulators and some are still under construction. An overview of how RoboCanes communicates with the simulator is shown in figure 3.8.
3.5.1 GazeboSim

GazeboSim [10] was developed by Nate Koenig and is now part of the Open Source Robotics Foundation, OSRF, which develops Robot Operating System, ROS, [49] used worldwide and the most widely used framework for robotics applications. The GazeboSim represents close to accurate results, but it is also very expensive and requires expensive hardware to run on. The RoboCup community is trying to adapt GazeboSim into the simulation league because of its accurate Rigid-body dynamics simulation, but has not been successful because the simulation league consists of two teams with 11 robots on each team (22 robots) where they have to be simulated correctly. The contact forces alone is very expensive for all these robots and so GazeboSim has some issues with scalability that needs to be fixed in order for the simulation to be used with its full potential. Figure 3.9 shows an example of many robots running on the simulator in a RoboCup environment.

Figure 3.9: Gazebo Graphical Interface [10]
3.5.2 Webots

Webots [3] was developed by Cyberbotics, and it is widely used in the NAO community because it has official support from Aldebaran Robotics [2] for the NAO firmware which seamlessly connects to and allows users to develop applications for the NAO robot on the simulator. Webots, shown in figure 3.10, is also widely used by our research group to test motion controllers and was mainly used for optimization schemes that will be discussed in section 5.2.

![Webots Graphical Interface](image)

Figure 3.10: Webots Graphical Interface [3]

3.5.3 V-Rep

V-Rep [11] is a rigid-body simulator developed by Coppelia Robotics. The advantage of V-Rep is the amount of functionality provided by the simulator in the interface. Unlike other interfaces that only allow scripts to control the functionality
of the simulator, V-Rep contains an easy-to-use interface that allows the developer to work in the interface and create quick scripts to control a robot. An example of V-Rep’s interface is shown in figure 3.11. In order to control a robot in V-Rep, there are various ways to develop a plugin. The first way to communicate your agent with V-Rep is to write an internal application that communicates and lives inside V-Rep. These applications are called child scripts and are located in the working tree of the robot in the simulator. A child script is a lua script and it uses the regular API to call functions from V-Rep. These child scripts can communicate with external applications by communicating through the V-Rep server. Another way of communication is writing a plugin that launches when V-Rep launches. These applications are C/C++ programs that provide specialized behavior for the simulator. These plugins are also written using the regular API. The third way of communicating with V-Rep is an external application that communicates with V-Rep through a server. External applications use a remote API that has functions that communicate through a client id. External applications have the advantage of running independently from
V-Rep. Another way of communication is through a ROS (Robotic Operating System) interface. There is also an auxiliary API that provides helper functions to any application to use libraries provided by the simulator. The general V-Rep framework is provided in figure 3.12. Figure 3.13 shows a sequence of head motions controlled by our RoboCanes agent in the V-Rep framework using V-Rep’s remote API.
Figure 3.13: Head yaw and pitch is controlled by the RoboCanes agent in V-Rep
CHAPTER 4

Implementation

4.1 Dynamic Kick Engine

The dynamic kick engine was developed with the RoboCanes framework developed by prior members from the RoboCanes team. The NAO robot was used for testing and validation of the dynamic kick engine. The kick engine was written in C++. The initial parameters used for the kick engine were computed empirically but then were optimized which will be discussed in section 5.2. The kick trajectories were tested with different direction vectors and kick speeds which are the inputs into the dynamic kick engine. From the direction kick vector and speed, the dynamic kick engine is able to produce kick trajectories by generating end points and via-points that are kinematically feasible that reach the points the lie in this direction vector. The time it takes to get to the end point is computed from the speed given by the agent. The dynamic kick engine is made of many components as shown in figure 4.1. When the input of the agent is received, the robot starts leaning on one side of the foot depending the foot chosen for the kick. The leaning side motion is done with static values determined empirically. When the robot is on one foot, the robot automatically transitions into the preview controller so it can balance one foot. If the
Figure 4.1: Overview of the dynamic kick engine framework [5]

preview controller is not turned on, the robot will fall on its side. When the preview controller is running, it calculates the ZMP from section 3.1.2 and CoM from section 2.3.3 so it can then find a CoM trajectory that will keep the robot balanced in the next frame. Meanwhile, a spline trajectory is generated that satisfies the requirements of the kick. When a trajectory is found, the robot executes the trajectory by sampling the kick trajectory at different times, $\delta t$. Of course as $\delta t$ goes to zero, the trajectories will be smoother but it is also computationally impossible to do so. Therefore, we choose a $\delta t$ that generates feasible trajectories and it is computationally feasible. When the kick trajectory is generated, the robot goes back to its two feet through a static motion. This completes the whole state machine and the robot goes back to its initial configuration. A kick in the simulator and the physical world can be seen in figure 4.2a and 4.2b.

4.2 Dynamic Walk Kick

Seekircher et al. [4] developed a dynamic walk engine that uses a 3D linear inverted pendulum model (LIPM) to generate a gait. Since there is an analytical solution
for the LIPM, the LIPM in [4] is used to compute the change of the center of mass trajectory with respect to the state of the robot which in this case is the foot positions of the robot.

The actual ZMP and equations for step positions were not derived from the LIPM such as [35], but rather Seekircher el al. did an optimization on the parameters of the step value that satisfies the stable center of mass trajectory on the given surface. This relieves the engineer from having to calibrate the robot on different surfaces which is the case in RoboCup preparation. The reason for these manual calibration is due to the simplicity of LIPM which does not capture the complete dynamics of the robot such as Coriolis effects from the ground friction forces and nonlinear forces applied to the robot. There are more elegant techniques to find stable motions such as linear and angular momentum control [50] but requires more computation; this is not feasible on the NAO robots.

To generate a kick motion, the robot is required to execute a stable stop motion when the robot is walking which means the robot has to decrease linear momentum and both foot positions need to be in an initial configuration. When this is done, the robot will only then execute the kick motion. Henceforth, it is inefficient and
Figure 4.3: The kick controller resides in the dynamic kick engine. When the agent sends a request for a walk kick, the kick controller sends a request to the step planner which resides in the walk engine. The kick controller sends the kicking foot and the kick direction of the foot. The step planner sends this information to the swing phase control which keeps track of the current swing foot. When the swing foot phase of the kick starts, the swing phase control sends a request to the swing control which generates the sextic polynomial, in collaboration with the kick controller, that corresponds to the direction given. When the swing phase ends, the swing control sends a finish signal to the kick controller. The kick controller now waits for another request and the cycle starts again.

expensive to generate a kick motion. Thus, a kick that can be executed while the robot is walking will be efficient because it will allow the robot to use the linear momentum of the robot and generate a kick motion. In pursuance of a kick motion, the kick controller has to have a notion of the step planning of the foot positions of the robot. Therefore if the agent requests a kick while the robot is walking, the step planner of Seekircher et al. has to plan for the kick motion while guaranteeing a stable walk. In order to do this, the step planner waits until the current foot for the kick motion is in the swinging phase. When this is the case, the step planner sends a request to the swing control for a kick motion trajectory. The kick motion trajectory is then generated using a trajectory generated from one of the interpolators which
Figure 4.4: An example of the walk kick trajectory is executed in real time. When the agent sends a command to kick, the walk kick trajectory generates a starting position, via position, and the end position. The start position is the first point in the curve from the left. The second point is the via-point and it is the contact point to the ball. The end position is the next step position for the support phase. Note that the first and second position are the same positions from the step planner. The only difference is the via-point position which is the contact point to the ball. This via-point is what makes the trajectory a kick motion.

in this case is the sextic interpolator. The starting position (the retraction point) of the kick trajectory is the same starting position of the step planner. In fact, the only difference between a regular walk trajectory and a kick trajectory will be the via-point. Therefore, the final position of the swinging leg is the same as the step planner, and the via-point is the actual contact point of the foot and the ball. The whole walk kick trajectory synthesis is shown in figure 4.3 and an example of how the walk kick trajectory is generated is shown in figure 4.4.
CHAPTER 5

Results and Summary

5.1 Dynamic Kick

For the validation of our approach, various kicks were generated and data for the balancing and spline trajectory generation was recorded. Figure 5.1a shows the ZMP reference point for a right front kick was 0.03 for x and 0.052 for y for all preview reference points in the preview controller. Although the x and y ZMP values varies around the reference points, the error is less than 1 centimeter which is less than the radius of the foot or support polygon. Furthermore, the small error accumulated is due to the simplification of the inverse kinematic model, but it still gives a good approximation to the CoM position. The CoM position in 5.1b is very close to the ZMP value shown in figure 5.1a because when the robot is balancing on one foot, it does not generate a force in the x or y axis. This can be verified in figure 5.2a, which is very close to zero. The graph in 5.2a is bounded between $-1.5 \times 10^{-4}$ and $2 \times 10^{-4}$. Hence, the difference between the estimated ZMP and the estimated CoM is negligible within working levels of precision. Although we might be tempted to control the CoM alone and disregard ZMP because they coincide, we need to notice that if a force is applied to the robot by another robot, it will generate a force in
Table 5.1: Optimization results from Webots Simulator for front kick trajectories using cubic splines as the interpolator. The Time column is the duration of the kick. The Percent column shows the percent of the time spent on the first polynomial. X, Y, and Z Via is the via-point in 3D space.

<table>
<thead>
<tr>
<th>Dimensions</th>
<th>Distance (m)</th>
<th>Time (ms)</th>
<th>Percent</th>
<th>X Via</th>
<th>Y Via</th>
<th>Z Via</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2.008</td>
<td>1180</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>2</td>
<td>2.124</td>
<td>853</td>
<td>0.46</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>3</td>
<td>2.230</td>
<td>1316</td>
<td>0.53</td>
<td>0.49</td>
<td>0.49</td>
<td>0.49</td>
</tr>
<tr>
<td>5</td>
<td>2.696</td>
<td>2273</td>
<td>0.77</td>
<td>0.1</td>
<td>0.42</td>
<td>0.44</td>
</tr>
</tbody>
</table>

the x or y axis, which in turn might make the CoM leave the support polygon, but we can still use the ZMP criterion to stabilize the robot. In figure 5.3a - 5.3b, the polynomials in the x, y, and z plane have been generated and projected in 3D space. The blue polynomial is the first polynomial and the orange polynomial is the second polynomial generated, and the union is the via-point. As can be observed, the polynomials are very close to a straight line. This is due to the configuration limit of the robot. Since the robot operates in a very small space, the coefficients of the polynomials are very small (less than 1). Therefore, the quadratic and cubic terms are negligible and the linear term is dominant. In order to get better shapes for the kick trajectories, a Covariance Matrix Adaptation Evolution Strategy (CMA-ES) optimization [51] in Webots simulator was done to find front kick trajectories that generated kicks where the ball traveled the farthest, the objective function. The actual parameters tuned during the optimization were the trajectory properties discussed in section 3.4.1. Four optimizations were done. For the first optimization, only two properties of the polynomial were optimized: time and the via-point location on the trajectory, percent. For the second experiment, time, percent, and uniform via-point position for the three planes were optimized, and the last optimization optimized all properties of the polynomial. Dimension zero in table 5.1, shows the empirical values for comparison and all the results were based on kick trajectories generated from the
Figure 5.1: The ZMP error between the reference point and the actual ZMP, and the position of the CoM in x and y.

empirical values. The optimization was done with the cubic spline generator and the configuration limit of the kick trajectory can be visualized in 5.2b. A video of the kick engine can be found at: https://www.youtube.com/watch?v=4fmuqI_CpQw.

5.2 Model Optimization

Generating a good parameter set is important to attain a good kick. Although these values can be found empirically, it is a tedious task. We therefore used CMA-ES for model optimization for different kick trajectory generators. The values were optimized and visualized in Webots, which can be seen as the standard simulation software for NAOqi. The first parameter set optimized was the time length of the
Figure 5.2: The CoM velocity, and the kick limit of the NAO in 3D space.
Figure 5.3: The dashed polynomial is generated first. The front kick trajectory in 3D space, and the back kick trajectory in 3D space.
kick and the speed towards the via-point. The second run on the model optimization was done on the time length of the kick, the speed of the via-point, and the via-point location in the x, y, z plane. Although this gave good results, the via-point was moved equally proportionally in 3D space, which resulted in a polynomial where the cubic and quadratic components were negligible in the specified interval of $t$. Therefore for the last optimization run, the via-point was moved disproportional in the x, y, and z plane to generate curves in 3D space. An overview of the optimization framework can be seen in figure 5.4a and the whole architecture and each component can be seen in 5.4b. Although we got promising results from the optimization, we believe we can get better results for the optimization by increasing the dimension of the optimization. As you can see in table 5.1, as we include more parameters, there is an increase in the distance of the optimal kick as well as a decrease in the number of generations until convergence. The additional parameters that can be added are: kick limit configuration (exploring a set of limits on the polynomials that can generate a good kick), end points of the polynomial, and the time it takes to get to the retraction point.

### 5.2.1 Kick Trajectory Optimization

For the validation of our approach, various kicks (forward, side, diagonal, backward) were generated and data for kick trajectory generation were recorded. The first graph in figure 5.5 shows the fitness function value and it demonstrates that the optimization was able to learn new kicks that can kick the ball further than the original kicks built from observation. It is also important to notice that sextic had a harder time to minimize the function value as well as Hermite. The cubic spline
(a) Simulation Framework [6]

(b) Optimization Framework [5, 6]

Figure 5.4: Overview of the optimization framework for the kick engine and the simulation control framework
Figure 5.5: Results from optimization on Webots simulator
interpolator was able to converge faster as well as more stable at every generation. This can be due to the initial seed of the other interpolators being inferior to the seed of cubic spline making it harder to explore. On the right of the fitness function graph in figure 5.5, the distance traveled of the ball was plotted. Initially, it can be seen that the ball traveled about three meters with a viable seed. After 300 iterations, the distance of the ball increased by a factor of about two. It is important to note that the distance traveled of the ball in each iteration is the maximum distance traveled for the current generation. Hence within each generation, the distance traveled for the population must have been less than or equal to the maximum distance traveled in the current generation. The dynamics of the parameter values for the three kick interpolators can also be seen in figure 5.5. As can be seen, the optimization explores the parameter space vastly for the first 200 iterations. After exploring the parameter space for 200 iterations, the optimization stops searching and rather exploits the current parameters. In general, the optimization converges quickly because the initial seed of the optimization is a viable kick. Therefore this shows how essential it is to start with a good seed. The kick trajectories were also generated on the physical robots. Figure 5.6 shows the first iterations of the optimization on Webot and figure 5.7 shows the final iterations of the optimization on Webot.

5.3 Summary

We have developed and implemented a dynamic kick engine that dynamically balances using a ZMP based preview controller. In order to move the CoM to the correct position, we use a simplified model that consists of a two link arm. The end effector of the arm is denoted as the CoM. The kick trajectory is generated using
Figure 5.6: The beginning of the optimization where the robot cannot kick the ball without the system failing to recover
Figure 5.7: The optimization after many iterations finds stronger kicks that allow the system to be in balance
cubic splines. The knot in the cubic splines is called the via-point, and it is used to control the shape of the polynomial. The kick was optimized using a simulation software called Webots. On Webots, we optimized the via-point position, the speed to the via-point, and the time duration of the kick. The optimization attained optimal set of parameters without having to experiment on the robot to find them. These values were used on the physical robot to verify results.

We have also compared results of kick trajectories generated by different joint interpolators. The results have shown that kicks can be optimized using these interpolators. The joint interpolators used for these experiments were cubic splines which are two three degree polynomials that meet at a via point, a sextic polynomial which is a six degree polynomial, and a cubic Hermite spline which needs to have the end points and their tangent vectors defined. These results demonstrated that the joint interpolators can be optimized to get powerful kicks rather than optimizing on the full joint space; drastically reducing the dimension of the problem. The cubic spline resulted the best joint interpolator in terms of minimizing the function value at a faster pace. The results have also exhibited that with a good initial seed, the optimization can converge at a rather fast pace; in contrast with optimizations done on the joint space which can take longer training times. Lastly, we have integrated the kick engine with a state-of-the-art walk engine to generate kick trajectories while walking.
Bibliography


