Learning to Walk
A Self Optimizing Gait for the Nao

Daniel Mescheder

July 10, 2011

Abstract

In this paper, a self optimizing walk for the Nao humanoid robot is presented. The approach uses Reinforcement Learning with the Linear Inverted Pendulum Mode as an initial bias. A controller based on a linear dynamical system is derived as a means to calculate the inverse kinematics which are needed for an implementation of the Reinforcement Learning trajectory. Furthermore, it is shown, how the number of parameters to be learned can be reduced by applying a technique similar to Taylor approximation.

1 Introduction

Biped walking is a natural skill for humans. Attempts to let an artificial entity walk, however, have proven it to be a challenge from an engineering perspective. Clearly, biped walking constitutes a very flexible means of locomotion: Legged entities can generally maneuver easily in rugged terrain. Another reason to study biped walking, especially in relation with machine learning, is to find a model that explains the human gait and can therefore be used for physiological analysis.

In this research the Nao robot was used as a platform. This humanoid, developed by Aldebaran robotics is (amongst others) used in the RoboCup standard platform league [6]. Several walks have been developed for the Nao robot. Kulk and Welsh for instance presented an improved version of the walk available on the Nao by default in which they lowered the joint stiffness [7]. Yet, the Nao's effective mobility is still far from humanoid.

Kajita et al. derived the Linear Inverted Pendulum Mode (LIPM), a model for biped walking based on the assumption that the robot can be approximated by a pointmass on a stick [8]. It was shown that the LIPM can be implemented on the Nao [4]. Nevertheless, this model is very simplistic and does not capture the complexity of the real dynamics involved as it neglects friction and other real world constraints.

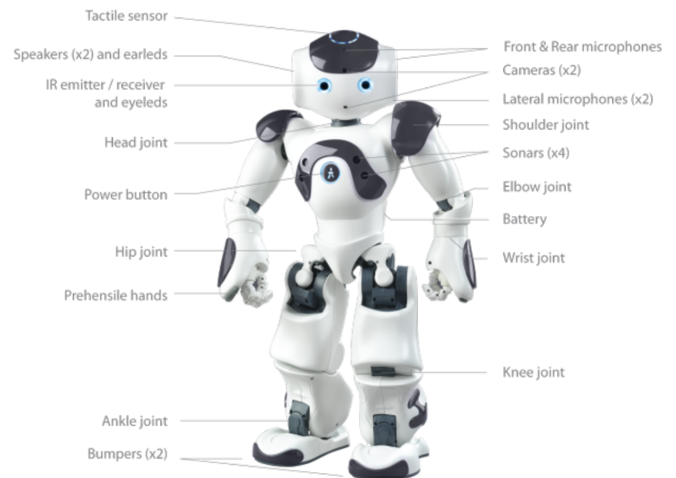


Figure 1: The Nao humanoid. Source: Aldebaran [1]

This paper will discuss how the LIPM can serve as a basis for a self optimizing gait based on Reinforcement Learning (RL). The LIPM will be used as a starting point. It will be especially pointed out how the high dimensionality of this problem can be approached and how supervised learning can be used to transfer the LIPM to a setting in which RL is applicable.

The next section will sketch the general structure of our approach. It will be pointed out, which subproblems have to be solved in order to arrive at a stable walk. Section 2 provides the preliminaries which are necessary to implement a self optimizing walk on the Nao. For this purpose, the concept of forward- and inverse-kinematics is introduced, and an overview of the LIPM model is given. In the following section, Reinforcement Learning is presented. Furthermore it is described, how gradient descent can be used to overcome the curse of dimensionality using the example of an approximation technique based on Taylor series. Section 3 shows how to translate the LIPM to a Q-function which can be improved using RL. Experimental results are given in Section 4 and finally the conclusion will be drawn in Section 5.

1.1 Problem Analysis

To control the Nao in a walking pattern, there are several steps involved. The Nao's operating system runs in a cycle of 20ms. In every such step, new joint positions can be set. The Nao's internal controller will then generate the necessary motor torques to reach the desired position.

We want to describe the desired trajectory as a sequence of euclidean space positions, i.e. x, y and z coordinates. Therefore a procedure is needed which calculates joint space positions from euclidean space coordinates. This problem is referred to as *inverse kinematics*. For most systems, there exists no analytical solution to this problem. Depending on the structure of the robot at hand, there might be no joint space configuration for some euclidean space positions. Other systems in turn are redundant such that there exists an infinite number of joint configurations for the same euclidean space position.

Finally, a trajectory needs to be found which describes a stable walk. Ideally it should take into account feedback from the robot's sensors and to modify itself such that it will get more stable over time. This is the point were Reinforcement Learning comes in.

It is possible to describe and to learn a motion in joint-space i.e. as a sequence of joint position vectors. However, this has several disadvantages: Firstly, joint space trajectories are difficult to interpret. Secondly, joint chains are often redundant, thus a joint space trajectory would have more dimensions and is thus more difficult to handle by learning algorithms. Finally, it is possible to create a model which allows us to analytically derive a trajectory in euclidean space as the seciton about the LIPM will show.

2 Background

This section will provide the necessary background to implement a walk on the Nao robot. First, we consider coordinate frames and homogeneous transformations. Subsequently, we apply these concepts to robotics and derive the forward-kinematics. It will be shown how joint space velocities can be translated to euclidean space velocities using Jacobians. It is then skteched how these concepts can be used in a controller that translates euclidean space coordinates to joint space coordinates. In Section 2.4 the LIPM will be introduced and it is shown how the above mentioned controller can be used to implement it on the Nao.

2.1 Kinematics

This section will introduce the concept of forward- and inverse-kinematics which is necessary for constructing a controller which lets the Nao follow an euclidean space trajectory. For more information about kinematics in general, refer to Craig's introduction to robotics [5].

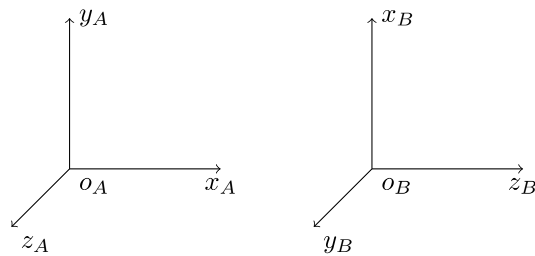


Figure 2: The coordinate frames A (left) and B (right)

Frames Consider the two coordinate frames A and B shown in Figure 2. Each frame consists of three orthonormal axes, x, y and z, and an origin o. We denote the coordinates of a given vector v in a frame F by v^F. We now consider how, knowing the coordinates in frame A of the unit vectors x_B, y_B, z_B of frame B and the origin o_B of frame B, we can use v^B to find v^A. Notice that $v^B = (o_B v)^B = x_B^B v_x^B + y_B^B v_y^B + z_B^B v_z^B$ where, v_x^B denotes the x-coordinate of v in frame B, and v_y^B and v_z^B are defined similarly. Furthermore, $(ab)^F$ denotes the vector from point a to point b, expressed in frame F. Thus,

$$\begin{aligned} (o_B v)^A &= x_B^A v_x^B + y_B^A v_y^B + z_B^A v_z^B \\ &= \begin{bmatrix} x_B^A & y_B^A & z_B^A \end{bmatrix} \begin{bmatrix} v_x^B \\ v_y^B \\ v_z^B \end{bmatrix} \\ &= R_B^A v_B \end{aligned}$$

where $R_B^A = \begin{bmatrix} x_B^A & y_B^A & z_B^A \end{bmatrix}$. Using this equation and the fact that $v^A = (o_A v)^A$, we can derive Equation 1, given below. The matrix R_B^A that occurs in this equation is called the rotation matrix from frame B to frame A.

$$\begin{aligned} v^A &= (o_A v)^A \\ &= (o_A o_B)^A + (o_B v)^A \\ &= o_B^A + R_B^A v^B \end{aligned} \tag{1}$$

If we transform v^C to v^B and then to v^A using Equation 1, the expression we find can become rather large, especially if many frames are involved. To avoid this, we introduce the homogeneous coordinates of a vector in a given frame. These coordinates are obtained by adding an additional 1 to the ordinary coordinates of the vector. Using these homogeneous coordinates, we can rewrite Equation 1 to Equation 2. The matrix T_B^A that appears in this equation is called the (homogeneous) transformation matrix from B to A. The composition of

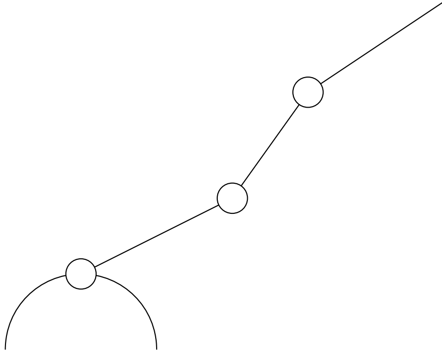


Figure 3: A robot arm. The base is indicated by the large semicircle and joints are indicated by circles.

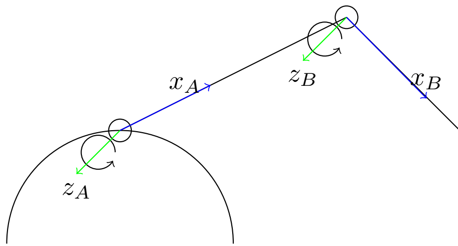


Figure 4: Joints and their associated frames. The direction of rotation of each joint is indicated by an arc.

such transformation matrices is equivalent to their matrix product, as indicated in Equation 3.

$$\begin{bmatrix} v^A \\ 1 \end{bmatrix} = \begin{bmatrix} R_B^A & o_B^A \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v^B \\ 1 \end{bmatrix} = T_B^A \begin{bmatrix} v^B \\ 1 \end{bmatrix} \tag{2}$$

$$T_C^A = T_B^A T_C^B \tag{3}$$

Chains, end effectors and forward kinematics We now apply the concepts of frames and transformations between them to robot arms. For our purposes, a robot arm is a series of joints connected to a static base, as shown in Figure 3. This will also be referred to as a *chain*. Each joint in a chain is assumed to move only by rotating around a particular axis, called its *joint axis*. We assign a frame to each joint, whose z-axis points along the respective joint axis. The x-axis of the i-th frame is taken to be a common normal of the z-axes of the i-th and $(i+1)$-th frame.[1] The y-axis is then determined by the additional assumption that each frame should be right-handed, that is[2], that $x \times y = z$. An example of these frames is shown in Figure 4.

Given two frames, we can now define 4 parameters that determine the transformation from the $(i-1)$-th frame to the i-th frame. These numbers, a_i, α_i, d_i and θ_i, have a geometrical interpretation, as shown in Figure 5. The parameter α_i is the rotation angle between z_{i-1} and z_i, along the axis x_{i-1}. The distance between these vectors along x_{i-1} is named a_i. The third parameter, d_i, gives the distance along z_i between x_{i-1} and x_i. The final parameter, θ_i, is the rotation angle about z_i between x_{i-1} and x_i. With these parameters, we can calculate the transformation matrix from the frame i to the frame $i-1$, using Equation 4. Of the four DH-parameters, a_i, α_i and d_i are determined by the robot's geometry. The remaining parameter, θ_i, is a variable giving the position of the i'th joint. Thus, the transformation matrix as defined by Equation 4 can be viewed as a function from the joint's angle, that is, its parameter θ_i, to its transformation matrix.

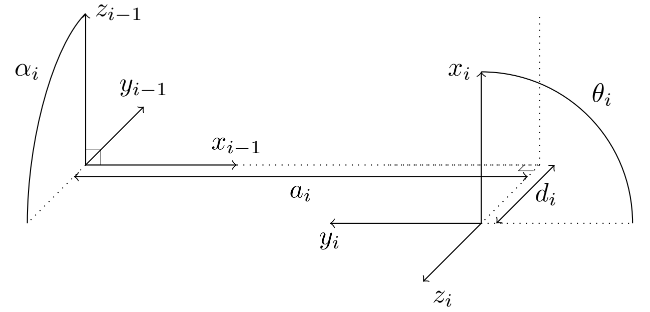


Figure 5: The frames $i-1$ and i and the DH-parameters of the transformation between them

$$T_i^{i-1}(\theta_i) = \begin{bmatrix} \cos\theta_i & -\sin\theta_i & 0 & a_i \\ \sin\theta_i\cos\alpha_i & \cos\theta_i\cos\alpha_i & -\sin\alpha_i & -d_i\sin\alpha_i \\ \sin\theta_i\sin\alpha_i & \cos\theta_i\sin\alpha_i & \cos\alpha_i & d_i\cos\alpha_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4}$$

Usually, we are not interested in the positions of individual joints in the chain, but rather in the final one. The final joint of a chain is commonly called its end-effector. The transformation from the end-effector frame to the base is called the forward kinematics of the chain. Using Equation 3, we can define the forward kinematics for an n-joint chain as the matrix F defined in equation 5. F is given as a function of q, the vector of joint angles of the chain.

$$F(q) = T_n^0 = \prod_{i=1}^{n} T_i^{i-1}(q_i) \tag{5}$$

Though the forward kinematics completely specify the position and orientation, this is a redundant representation. In particular, the rotation matrix uses 9

[1] A vector w is a common normal of u and v if w is orthogonal to both u and v

[2] $x \times y$ is the cross product of the vectors x and y

parameters to encode orientation, while smaller representations using 3 or 4 parameters exist. A smaller representation will lead to a better performance. Additionally, in what follows it will become clear that we need a representation whose time derivative can be easily found from an angular velocity vector. To satisfy both these requirements, we will represent orientations by quaternions, which require 4 parameters. For a rotation about an axis v over an angle ϕ, the corresponding quaternion[3] $\begin{bmatrix}\mathbf{r} & r_0\end{bmatrix}^T$ is given below in Equation 6.

$$\begin{bmatrix}\mathbf{r} \\ r0\end{bmatrix} = \begin{bmatrix}v\sin(\frac{\phi}{2}) \\ \cos(\frac{\phi}{2})\end{bmatrix} \tag{6}$$

Though Equation 6 allows us to find the quaternion corresponding to a given rotation if we know the rotation axis and the rotation angle, it does not tell us how to find the quaternion corresponding to a given rotation matrix. For more information about this subject refer to [4]. We will not need the opposite calculation, that is, finding a rotation matrix for a given quaternion. A formula for this matrix can be found in Angeles' textbook on robotics [2, Section 2.3.6].

Using quaternions as we have defined them above, we can define the position of an end-effector as a seven-dimensional vector $p(q)$. Since the forward kinematics F is a transformation matrix, it has the form given in Equation 2. Thus, we can extract the linear position of the end-effector by taking the first three elements of the last column of $F(q)$. We will denote this 3-element vector by $o(q)$. The rotation matrix R_n^0 consisting of the first 3 rows and first 3 columns of $F(q)$ can then be used to find the quaternion $r(q)$ corresponding to the end-effector's orientation. We can then define $p(q)$ as the vector consisting of $o(q)$ and $r(q)$, as given by Equation 7.

$$p(q) = \begin{bmatrix}o(q) \\ r(q)\end{bmatrix} \tag{7}$$

Jacobians Previously, we showed how to calculate the position of an end-effector given the joint angles of a chain. We will now consider how to calculate the linear and angular velocity of the end-effector.[4] To do so, let $\dot{q}$ be the vector of time derivatives of the chain's joint angles. The linear and angular velocity of the end-effector can be found using Equation 8.[5] The matrix $J(q)$ that appears in this equation is called the Jacobian of the system.

$$\begin{bmatrix}v \\ \omega\end{bmatrix} = J(q)\dot{q} \tag{8}$$

To compute the Jacobian, we shall use its explicit form, as discussed in the Stanford introductory course in robotics [9, Lecture 7]. The essential idea of this form is that each column of the Jacobian specifies the contribution of the corresponding joint to the total linear and angular velocity. Using this idea, one can derive the form given in Equation 9[6] , where J_i is the i-th column of J. Furthermore, p_i is the position of the end-effector relative to the origin of frame i and z_i is the z-axis of frame i.

$$J_i = \begin{bmatrix}z_i^0 \times p_i^0 \\ z_i^0\end{bmatrix} \tag{9}$$

We can now compute the Jacobian of a robot arm and therefore its linear and angular velocity. This is not yet sufficient for inverse kinematics. In what follows we need the matrix of partial derivatives of $p(q)$ with respect to q. To find this matrix, we need to transform the Jacobian to the matrix $J_r(q)$ that satisfies Equation 10, where $\dot{r}$ is the time derivative of the quaternion corresponding to the end-effector's orientation.

$$\begin{bmatrix}v \\ \dot{r}\end{bmatrix} = J_r(q)\dot{q} \tag{10}$$

As shown by Angeles [2, Section 3.4.2], $\dot{r} = H(r)\omega$, where $H(r)$ is a matrix that depends on the quaternion r. In order to derive the formula for H, we will first define the cross product matrix (cpm) operator. Given a vector v, the cross product matrix $\text{cpm}(v)$ is the matrix such that for any vector x, $v \times x = \text{cpm}(v)x$. The formula for this matrix given by Angeles [2, Section 2.3.1] is given below as Equation 11[7] .

$$\text{cpm}(v) = \begin{bmatrix}0 & -v_z & v_y \\ v_z & 0 & -v_x \\ -v_y & v_x & 0\end{bmatrix} \tag{11}$$

Using this equation, we can give the formula for the matrix $H(r)$. To write this formula, we decompose the 4-element quaternion r into two parts, the vector $\mathbf{r}$ consisting of the first 3 elements and the last element r_0. With this notation, the formula given by Angeles [2, Section 3.4.2] for $H(r)$ is given below as Equation 12 where I_3 denotes the 3×3 identity matrix. We conclude this section with the formula for $J_r(q)$ in terms of $J(q)$. As mentioned previously, $J_r(q)$ is obtained from

[3] The superscript T denotes the matrix transpose. Thus, a quaternion is a column vector of 4 elements.

[4] Despite its name, the angular velocity is not the derivative of orientation. As we shall see below, the derivative of the quaternion encoding orientation and the angular velocity are related by a linear transformation.

[5] v denotes the linear velocity of the end-effector. The angular velocity is denoted by ω.

[6] Here, the base frame is referred to as frame 0. The vector $z_i^0 \times p_i^0$ is the cross-product of z_i and p_i, when both vectors are expressed in the base frame.

[7] As before, v_x is the x-coordinate of v and v_y and v_z are defined accordingly.

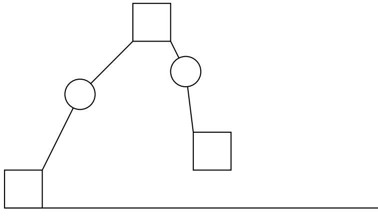


Figure 6: The nao's lower body

$J(q)$ by means of a linear transformation, as given in Equation 13.

$$H(r) = \frac{1}{2}\begin{bmatrix} r_0 I_3 - \text{cpm}(\mathbf{r}) \\ \mathbf{r}^T \end{bmatrix} \tag{12}$$

$$J_r(q) = \begin{bmatrix} I_3 & 0 \\ 0 & H(r(q)) \end{bmatrix} J(q) \tag{13}$$

2.2 A model of the Nao

We will now apply the concepts we have previously reviewed to the Nao. In doing so, we will show how to handle the simultaneous motion of the Nao's torso and foot relative to a stationary foot. We will also handle a number of peculiarities involved in applying the Nao's DH-parameter model. The first of these is that each chain of the Nao has an additional base frame and an end-effector frame, specified by constant transformation matrices. The second is that all of the Nao's chains are specified with the torso as the base. As we would like to use a foot as the base, we will need to calculate the parameters and transformations for the chain from the foot to the torso.

Before we define the position and velocity of the torso and foot moving simultaneously, we consider how to model the Nao's lower body. The structure of this lower body is drawn in Figure 6. The stationary foot that we take to be the base of the system is shown on the left. In the middle, we have the torso, which is one of the two end-effectors we wish to control. The other end-effector is the moving foot, which is indicated on the right. The joints connecting the base to the torso and the foot are indicated by circles.

Comparing Figure 6 to Figure 3, we notice that the Nao's lower body is in some sense a union of two chains. That is, the chain from the stationary foot to the torso forms part of the larger chain from the stationary foot to the other foot. Letting q be the vector of joint angles of the larger chain, we can calculate the positions of both the torso and the moving foot relative to the base using q. We will consider the combination of these two positions, stacked vertically, to be the position of the Nao's lower body. This is shown in Equation 14 where the subscript T refers to the torso and the subscript F to the moving foot.

$$p(q) = \begin{bmatrix} p_T(q) \\ p_F(q) \end{bmatrix} \tag{14}$$

Our next problem is how to define the Jacobian of the Nao's lower body. As before, we will define this Jacobian $J_r(q)$ to be the matrix of first-order partial derivatives of $p(q)$ with respect to q, that is, $J_r(q) = \frac{\partial p(q)}{\partial q}$. Since $p(q)$ has the structure shown in Equation 14, $J_r(q)$ has a similar structure, shown in Equation 15. Here, the matrix $J_{T,r}(q)$ is the Jacobian of the torso with respect to all the joints in the larger chain. Since the torso's position depends on only a limited number of these joints, the columns corresponding to the other joints should be set to zero. Each of the Jacobians $J_{T,r}(q)$ and $J_{F,r}(q)$ can be found using the explicit form of Equation 9.

$$J_r(q) = \begin{bmatrix} J_{T,r}(q) \\ J_{F,r}(q) \end{bmatrix} \tag{15}$$

There are four more practical issues that we encounter when we apply the above to the Nao: Firstly, it is not necessarily the case that the reference frame of the support foot coincides with the standard base frame. It is also possible that the frame associated with the end of a chain (end effector frame) is different from the last joint frame. Thus, there needs to be a technique to include this information into both the forward kinematics and the Jacobian.

Secondly, the DH-parameters for the Nao can be found in the documentation provided by Aldebaran [1]. However, the documentation lists the parameters of a chain from the torso to the support foot. The model we presented in this section requires a chain from the support foot to the torso. Hence, we need to compute the DH-parameters of this chain using the parameters given in the documentation.

Thirdly, the model described earlier in this section requires a chain from the support foot to the moving foot. Using the documentation [1] and the inversion procedure mentioned above, we can find a chain between the support foot and the torso and a chain between the torso and the moving foot. We can then combine these two chains to find the desired chain from the support foot to the moving foot. To do so, we have to take into account individual base- and end-transforms.

Finally, we have thus far assumed that all the joints of a chain can be moved independently. This, however, is not the case for the Nao's legs: The LHipYawPitch and the RHipYawPitch joints are controlled by the same physical motor which means that their joint angles are always equal. This needs to be taken into account during the construction of the Jacobian.

We will present two equivalent techniques to solve the problems described above. One is to explicitly model

the change of kinematics caused by each of them and to derive additional transformations that can be applied to the model presented above. The other technique relies on introducing special joints. Using these joints we need to make only small changes to the calculation of the forward kinematics and the Jacobian.

Base- and End Transforms As described above, there needs to be a way to represent the fact that the base frame of a chain does not coincide with the first joint frame and the end-effector-frame is not the same as the last joint frame.

This section will first describe the explicit approach. Let b be the new base frame and T_0^b as given below be the transformation between b and the standard base frame. Let accordingly e denote the end-effector frame and T_e^n be the corresponding transformation to the last joint frame.

$$T_0^b = \begin{bmatrix} R_0^b & p_0^b \\ 0 & 1 \end{bmatrix} \qquad T_e^n = \begin{bmatrix} R_e^n & p_e^n \\ 0 & 1 \end{bmatrix}$$

The new forward kinematics including base- and end-transforms are then given by the matrix $F_{be}(q)$ as specified below.

$$F_{be}(q) = T_0^b \cdot F(q) \cdot T_e^n \tag{16}$$

To transform a velocity vector to a new base frame it suffices to apply the rotation to the explicit form of the Jacobian (i.e. before the quaternion operator is applied):

$$J_b = \begin{bmatrix} R_0^b & 0 \\ 0 & R_0^b \end{bmatrix} \cdot J \tag{17}$$

If there is an additional end transform to the end effector, the following holds for the velocity vector:

$$\begin{aligned} v_e &= v_n + \omega_n \times p_e^n \\ \omega_e &= \omega_n \end{aligned}$$

This is equivalent to the following notation that uses the matrix form of the cross product defined in Equation 11:[8]

$$\begin{bmatrix} v_e \\ \omega_e \end{bmatrix} = \begin{bmatrix} I_3 & -\operatorname{cpm}(p_e^n) \\ 0 & I_3 \end{bmatrix} \begin{bmatrix} v_n \\ \omega_n \end{bmatrix} \tag{18}$$

Equation 18 only holds if v_n and ω_n are expressed in the same frame as p_e^n. Usually, however, we find that v_n and ω_n are expressed in the base frame, while p_e^n is located in frame n. We can use the rotation matrix $(R_n^b)^T$ to first transform v_n and ω_n to frame n. This ensures that the multiplication takes place in the right frame before the result is transformed back to the base frame using the rotation matrix R_n^b.

[8] Recall that I_3 denotes the 3×3 identity matrix

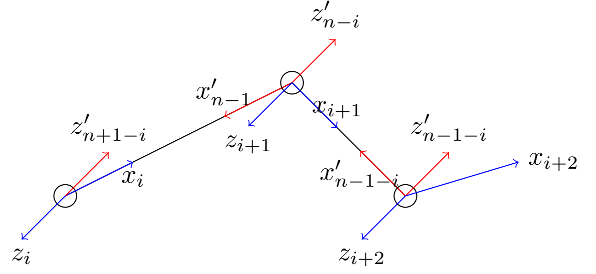


Figure 7: Original and reversed frames for one chain of joints.

$$\begin{bmatrix} v_e \\ \omega_e \end{bmatrix} = \begin{bmatrix} I_3 & -R_n^b \operatorname{cpm}(p_e^n) \left(R_n^b\right)^T \\ 0 & I_3 \end{bmatrix} \begin{bmatrix} v_n \\ \omega_n \end{bmatrix} \tag{19}$$

Premultiplying the Jacobian by the transformations shown in Equations 17 and 19 yields a new Jacobian matrix which takes into account both base- and end transforms. Also taking into account the quaternion transform shown in Equation 12 leads to Equation 20 below.

$$J_{be}(q) = \begin{bmatrix} R_0^b & -R_n^b \operatorname{cpm}(p_e^n) R_0^n \\ 0 & H(r(q)) R_0^b \end{bmatrix} J(q) \tag{20}$$

The second way to approach this problem is to add static joints at the beginning and end of the chain. Unlike regular joints, which are defined by their DH-parameters, static joints are defined by given transformation matrices. Thus, the transformation matrix of such a joint is a constant. With these joints, we can represent the base- and end-transform as a static joint at the beginning and end of a chain, respectively. The forward kinematics of the resulting chain can be calculated using Equation 5, where T_i^{i-1} is equal to the given transformation matrix if joint i is static. Similarly, we can use the explicit form of Equation 9 to calculate the columns of the Jacobian corresponding to the non-static joints. In doing so, we take into account the effect of the static joints' transformations on the vectors p_i^0 and z_i^0. Since static joints do not represent a moving mechanism, their contribution to the linear and angular velocity of a chain is zero. Thus, we do not add columns to the Jacobian corresponding to the motion of these joints.

Reverse DH-Parameters This section presents a procedure which can be used to invert the DH-parameters of a chain given in the Nao documentation.

Let x_i and z_i be the respective x and z axis of the i'th frame in the original chain, counting from the hip.

Furthermore, let x'_i and z'_i be the respective x and z axis of the i'th frame in the new chain, counting from the foot. Then we can define the axes of the reversed frames to correspond to the original frames according to the following rule which is depicted in Figure 7:

$$x'_i = -x_{(n-i)}$$
$$z'_i = -z_{(n+1-i)}$$

With this convention the new set of parameters a'_i, α'_i, θ'_i and d'_i can be determined in terms of the old parameters a_i, α_i, θ_i and d_i. For Equations 21 and 22 let $2 \leq i \leq n$ and for Equations 23 and 24 let $1 \leq i \leq n$.

$$a'_i = a_{(n+2-i)} \quad (21)$$
$$\alpha'_i = \alpha_{(n+2-i)} \quad (22)$$
$$\theta'_i = \theta_{(n+1-i)} \quad (23)$$
$$d'_i = d_{(n+1-i)} \quad (24)$$

In accordance with the DH convention, a'_1 and α'_1 are set to zero. It remains to find the reverse counterparts for x'_n and x'_0. When $\theta_i = 0$ the DH convention requires the angle between x_i and x_{i+1} to be zero. Therefore the following is a reasonable choice:

$$x'_n = x'_{(n-1)} = -x_1$$
$$x'_0 = x'_1 = -x_{n-1}$$

Whether static joints or explicit transforms are used to represent the base- and end transforms, these transformations need to be reversed as well in the process of reversing a chain:

$$(T^n_e)' = R_z(\pi)\, R_x(\pi)\, (T^n_e)^{-1} \quad (25)$$
$$\left(T^b_0\right)' = \left(T^b_0\right)^{-1} R_z(\pi)\, R_x(\pi) \quad (26)$$

Unfortunately, the DH parameters from the Nao documentation furthermore define $a_1 \neq 0$ and $\alpha_1 \neq 0$. Following the original DH convention, these values should be a part of the base transform. Let $T_x(d)$ denote the translation transform of d units along the x-axis and $R_x(\phi)$ be the rotation of ϕ around the x-axis. Let furthermore T^b_0 be the old base-transform. Then an equivalent formulation can be given by setting $a^{new}_1 = 0$, $\alpha^{new}_1 = 0$ and using the matrix $(T^b_0)^{new}$ derived below as a new base-transform:

$$\left(T^b_0\right)^{new} = T^b_0 T_x(\alpha_1)\, R_x(a_1) \quad (27)$$

Combined Chain for Nao-Legs For walking the control of the moving foot from the support foot frame is required. For this reason it is necessary to define a chain leading from the base frame to the desired end effector. Using the reversed chain derived in section 2.2 this is mostly a concatenation of the inverse of the support leg chain and the original chain of the moving leg.

However, one has to be careful with the DH-parameters of the HipYawPitch joints: If the original chains used base- and end-transforms, the DH parameters of the hip joint have to incorporate these transforms. One solution again is to use static joints instead of explicit base- and end-transforms. Using this method a simple concatenation is indeed sufficient. If explicit transforms are being used, the changed parameters of the chain from the left to the right foot are given below where HipOffsetY is defined in the documentation [1].

$$a_7 = 0$$
$$\alpha_7 = \frac{1}{2}\pi$$
$$d_6 = \sqrt{2} \cdot \text{HipOffsetY}$$
$$\theta_6 = -\frac{1}{2}\pi$$
$$d_7 = -\sqrt{2} \cdot \text{HipOffsetY}$$
$$\theta_7 = \frac{1}{2}\pi$$

The chain from the right foot to the left foot can be calculated using the chain reversal technique derived above.

Aliased Joints In regular robot control problems one normally assumes all joints of a chain to be independent from one another. This, however, is not the case for the Nao legs: The LHipYawPitch and the RLHipYawPitch joints are controlled by the same physical motor which means that their associated θ values are always equal.

This does not have any impact on the forward kinematics, but the Jacobian matrix has to be modified to represent this information. In the case of the chain from the support foot to the moving foot $\theta_6 = \theta_7$, i.e.

$$v = J \begin{bmatrix} \theta_1 \\ ... \\ \theta_6 \\ \theta_6 \\ ... \\ \theta_{12} \end{bmatrix}$$

Let J' be a new matrix which is nearly equal to J except that column 6 in J' is the sum of column 6 and 7 in J and that column 7 is removed from J'. From the

interpretation of matrix multiplication as a product sum the following holds:

$$v = J' \begin{bmatrix} \theta_1 \\ \dots \\ \theta_6 \\ \theta_8 \\ \dots \\ \theta_{12} \end{bmatrix}$$

A second way to approach this problem is to introduce alias joints. Conceptually, an alias joint is like a regular joint, except that its joint variable coincides with the joint variable of another joint. To clarify how this affects the calculations of the forward kinematics and the Jacobian, suppose joint i is an alias joint and its joint variable coincides with that of joint j. Then, to calculate the transformation matrix T_i^{i-1} of joint i, we use Equation 4, filling in the parameters a_i, α_i and d_i of joint i and the joint angle θ_k of joint j. With this transformation matrix, we can use Equation 5 to calculate the forward kinematics in the usual way. When we calculate the Jacobian, we calculate the column J_i as usual. However, instead of appending this column to the Jacobian, we add it to the k-th column J_k corresponding to joint j. Using alias joints, we can represent the fact that the LHipYawPitch and RHipYawPitch joints share the same physical motor by making one of these joints an alias joint whose joint variable coincides with that of the other.

Deriving all the particularities of the Nao kinematics was cumbersome. Nevertheless, it gives us the tools we need to construct an inverse kinematics controller which will be presented in the next section.

2.3 Inverse Kinematics

In this section, the concepts presented above will be used to derive the inverse kinematics. Inverse kinematics attempt to solve the problem of finding joint angles q such that $p(q)$ takes on some desired value p_d.

One straight forward method arises from the interpretation of the jacobian as a mapping of velocities from Joint space to euclidean space (Equation 10). Let $A^\dagger$ denote the pseudo-inverse of A. Then

$$\dot{q} = J_r^\dagger(q) \begin{bmatrix} v \\ \dot{r} \end{bmatrix} \tag{28}$$

gives us a joint space velocity corresponding to the euclidean space velocity. If we assume that the inverse kinematics function p was linear with respect to time and that the system is controlled in discrete time steps of δt, then moving by a displacement of δx within a time step corresponds to moving by δq in joint space where

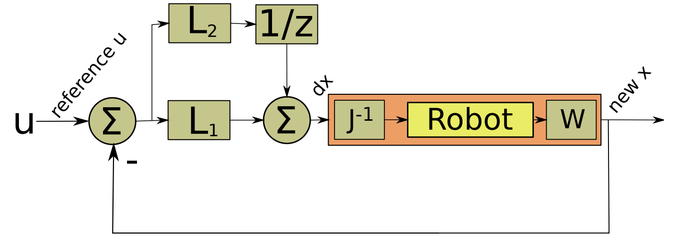


Figure 8: Controller for inverse kinematics. The two gains L_1 and L_2 have to be chosen such that the poles of the complete system lie within the unit circle in order to make the actual output trajectory converge to the reference trajectory u. The J^{-1} subsystem denotes the approximation of Equation 29. The output of the physical robot (yellow) is a vector of joint angles. W is the forward kinematics; the output of the red box is thus the euclidean space position of the controlled robot arm.

δq is calculated as shown in Equation 29:

$$\delta q \approx J_r^\dagger(q)\frac{\delta x}{\delta t} \tag{29}$$

In practice, the system is not linear. Equation 29 rather constitutes a first order Taylor approximation. If we wish to use this approximation for reliable trajectory tracking, a control system as shown in Figure 8 can be constructed to compensate for the error [12]. This controller takes the reference trajectory as its input. The current error is calculated as the difference between the reference trajectory and the position of the mechanical system. This error is then split into two parts, one is fed into a proportional gain subsystem L_1, one is fed into the delayed subsystem L_2. These two signals are summed and used as input to the approximation shown in Equation 29. The result of the J^{-1} block is an approximation of δq which is then used in the mechanical system (yellow) to control the joints. The output of the mechanical system will be the new joint positions. Using the forward kinematics presented in Section 2.1 (denoted by the W-block in Figure 8) this is translated to a new euclidean space position which in turn serves as the feedback to the controller.

Results from control theory show that the error of a discrete time system like the one shown in Figure 8 converges to zero if the poles of the system lie within the unit cycle of the complex plane [12]. The poles of a system are the eigenvalues of the corresponding system matrix.

In what follows, we will derive the system matrix of the system in Figure 8 and find its eigenvalues as a function of L_1 and L_2. Then it is shown for which values of L_1 and L_2 the system becomes stable. For this purpose we will analyze the individual subsystems and use them to calculate the overall system matrix.

The delay subsystem (denoted by $\frac{1}{z}$ in Figure 8) can be described in state-space form as shown below. Equation 30 shows the structure of the state vector, Equation 31 is the state update rule and Equation 32 describes the output of the delay-subsystem where $s(i)$, $y(i)$ and $u(i)$ denote the state, the output and the input, respectively, at time step i. The value n is the number of dimensions of the position vector. If for example a position vector as derived in Section 2.2 is used, this amounts to $n = 14$.

$$s(i) = \begin{pmatrix} \delta x_i \\ \delta x_{i-1} \end{pmatrix} \quad (30)$$

$$s(i+1) = \begin{bmatrix} I_n & 0 \\ I_n & 0 \end{bmatrix} s(i) + \begin{bmatrix} I_n \\ 0 \end{bmatrix} u(i) \quad (31)$$

$$y(i) = \begin{bmatrix} 0 & I_n \end{bmatrix} s(i) \quad (32)$$

This boils down to storing the current state and using it as the output in the next step. The constant gain is given by Equation 33 below:

$$y(i) = L_2 u(i) \quad (33)$$

Thus the combination of delay and constant gain L_2 is given by:

$$s(i+1) = \begin{bmatrix} I_n & 0 \\ I_n & 0 \end{bmatrix} s(i) + \begin{bmatrix} L_2 \\ 0 \end{bmatrix} u(i) \quad (34)$$

$$y(i) = \begin{bmatrix} 0 & I_n \end{bmatrix} s(i) \quad (35)$$

The output of the delay needs to be added to the constant gain L_1. The resulting Equation 36 is thus the state update rule of the controller and $dx(i)$ in Equation 37 describes the controller output at time step i.

$$s(i+1) = \begin{bmatrix} I_n & 0 \\ I_n & 0 \end{bmatrix} s(i) + \begin{bmatrix} L_2 \\ 0 \end{bmatrix} u(i) \quad (36)$$

$$dx(i) = \begin{bmatrix} 0 & I_n \end{bmatrix} s(i) + L_1 u(i) \quad (37)$$

At the beginning of this section we presented the assumption that the system consisting of inverse Jacobian and robot behave linearly with respect to time. Formally this means that we assume that the system behaves according to Equations 38 and 39. Here, the state consists of nothing but the current euclidean space position.

$$s(i+1) = I_n s(i) + I_n dx(i) \quad (38)$$

$$x(i) = I_n s(i) \quad (39)$$

Combining the description of the controller (Equations 36 and 37) with the description of the plant (38 and 39), the following forward system is found with $x(i)$ denoting the system output (i.e. the robots euclidean space position) at time i:

$$s(i) = \begin{pmatrix} \delta x_i \\ \delta x_{i-1} \\ x_i \end{pmatrix} \quad (40)$$

$$s(i+1) = \begin{bmatrix} I_n & 0 & 0 \\ I_n & 0 & 0 \\ 0 & I_n & I_n \end{bmatrix} s(i) + \begin{bmatrix} L_2 \\ 0 \\ L_1 \end{bmatrix} u(i) \quad (41)$$

$$x(i) = \begin{bmatrix} 0 & 0 & I_n \end{bmatrix} s(i) \quad (42)$$

This can be interpreted as follows: In the state given in Equation 40 we store the new displacement (with a gain), the previous displacement and the absolute position. The absolute position is calculated by adding the current absolute position to the delayed displacement to the input with a proportional gain.

Now that the forward system is known, the feedback can be taken into account. As there is no gain involved in the feedback, we get the following result:

$$s(i+1) = \begin{bmatrix} I_n & 0 & -L_2 \\ I_n & 0 & 0 \\ 0 & I_n & I_n - L_1 \end{bmatrix} s(i) + \begin{bmatrix} L_2 \\ 0 \\ L_1 \end{bmatrix} u(i) \quad (43)$$

$$x(i) = \begin{bmatrix} 0 & 0 & I_n \end{bmatrix} s(i) \quad (44)$$

Equation 43 is the state update rule for the overall system, including feedback. This is all we need to know to derive a controller by pole setting as we will show in the next section.

Setting Poles Setting the poles of this system amounts to choosing L_1 and L_2 such that the eigenvalues of the state-update matrix are the desired pole locations. For simplicity it is assumed that both L_1 and L_2 are diagonal matrices. The symbols l_{1j} and l_{2j} will be used to refer to the j-th entry on the diagonal of L_1 and L_2 respectively. Setting the eigenvalues of the state-update matrix means to chose values for λ and to solve the following equation for $L_1$and L_2:

$$\left(\lambda I_n - \begin{bmatrix} I_n & 0 & -L_2 \\ I_n & 0 & 0 \\ 0 & I_n & I_n - L_1 \end{bmatrix} \right) x = 0$$

$$\det \begin{bmatrix} (\lambda - 1) I_n & 0 & L_2 \\ -I_n & \lambda I_n & 0 \\ 0 & -I_n & (\lambda - 1) I_n + L_1 \end{bmatrix} = 0$$

$$\det \begin{bmatrix} M_{1,1} & 0 & M_{1,3} \\ M_{2,1} & M_{2,2} & 0 \\ 0 & M_{3,2} & M_{3,3} \end{bmatrix} = 0$$

First of all, we require that $\lambda \neq 0$ and $\lambda \neq 1$. This ensures that the inverse matrices of $M_{1,1}$ and $M_{2,2}$ exist and a Gaussian elimination can thus be performed:

$$\det \begin{bmatrix} M_{1,1} & 0 & M_{1,3} \\ M_{2,1} & M_{2,2} & 0 \\ 0 & M_{3,2} & M_{3,3} \end{bmatrix} = 0$$
$$\Leftrightarrow \det L \cdot \det U = 0$$

where

$$L = \begin{bmatrix} M_{1,1} & 0 & 0 \\ 0 & M_{2,2} & 0 \\ 0 & 0 & M_{3,3} + M_{3,2} M_{2,2}^{-1} M_{2,1} M_{1,1}^{-1} M_{1,3} \end{bmatrix}$$
$$U = \begin{bmatrix} I_n & 0 & M_{1,1}^{-1} M_{1,3} \\ 0 & I_n & -M_{2,2}^{-1} M_{2,1} M_{1,1}^{-1} M_{1,3} \\ 0 & 0 & I_n \end{bmatrix}$$

The determinant of an upper triangular matrix is just the product of its diagonal entries. Therefore $\det U = 1$ and it remains to solve $\det L = 0$ for L_2 and L_1. L is a diagonal block matrix. Therefore we can reduce the equation to:

$$\det M_{1,1} M_{2,2} \det \left(M_{3,3} + M_{3,2} M_{2,2}^{-1} M_{2,1} M_{1,1}^{-1} M_{1,3} \right) = 0$$

Recall that $\lambda \neq 1$ and $\lambda \neq 0$. That means that $\det M_{1,1} \neq 0$ and $\det M_{2,2} \neq 0$. It thus remains to solve the following:

$$\det \left(M_{3,3} + M_{3,2} M_{2,2}^{-1} M_{2,1} M_{1,1}^{-1} M_{1,3} \right) = 0$$
$$\Leftrightarrow \det \left((\lambda - 1) I_3 + L_1 + \frac{1}{\lambda(\lambda - 1)} L_2 \right) = 0$$
$$\Leftrightarrow \prod_{j=0}^{n-1} \left(\lambda_j^3 - 2\lambda_j^2 + \lambda_j + l_{1j}\lambda_j^2 - l_{1j}\lambda_j + l_{2j} \right) = 0 \quad (45)$$

Thus we can chose n poles by setting the diagonal entries of L_1 as shown in Equation 46 and the diagonal entries of L_2 as shown in Equation 47.

$$l_{1j} = a \quad (46)$$
$$l_{2j} = -\lambda_j^3 + (2 - a)\lambda_j^2 + (a - 1)\lambda_j \quad (47)$$

The variable a in Equation 46 is a not quite arbitrary: The values for l_{1j} and l_{2j} have to be real, i.e. $\text{imagpart}(l_{2i}) = 0$. Let $\lambda_j = x + yi$ then this requirement can be reduced as follows.

$$y^3 + \left(-3x^2 + (4 - 2a)x + a - 1 \right) y = 0$$
$$\Leftrightarrow a = \frac{y^2 - 3x^2 + 4x - 1}{2x - 1} \quad (48)$$

In conclusion, if we want to place a pole at $\lambda_j = x + y \cdot i$, we choose:

$$l_{1j} = \frac{y^2 - 3x^2 + 4x - 1}{2x - 1}$$
$$l_{2j} = \frac{(y^4 + (2x^2 - 2x + 1)y^2 + x^4 - 2x^3 + x^2)}{(2x - 1)}$$

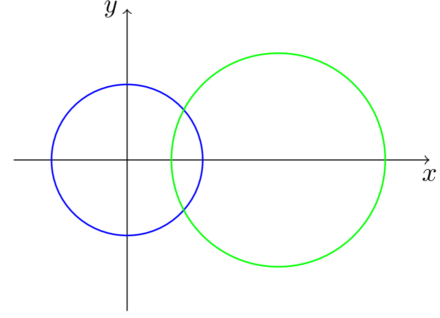


Figure 9: The range in which the pole of the control system can be chosen is the intersection of the two circles.

As the third order polynomial in Equation 45 has three (complex) roots, we get two more poles automatically every time we set one. As for optimal control, all of them have to lie within the unit circle, these poles impose further constraints on the choice of λ_j. Equations 49 and 50 below give the two additional poles we get for each λ_j:

$$\lambda_j' = x - i \cdot y \quad (49)$$
$$\lambda_j'' = -\frac{(y^2 + x^2 - 2x + 1)}{(2x - 1)} \quad (50)$$

λ_j' clearly is the complex conjugate of λ_j, thus if λ_j is within the unit circle, clearly so is λ_j'. λ_j'' is located on the real axis - we are interested in the extra constraint it poses unto x and y. Firstly, consider the upper bound:

$$\frac{-y^2 - x^2 + 2x - 1}{2x - 1} < 1$$
$$\Leftrightarrow \sqrt{y^2 + x^2} > 0$$

This simply means, that the radius should be greater than 0 - a constraint that already has been established before by demanding $\lambda_j \neq 0$. Now, consider the lower bound:

$$\frac{-y^2 - x^2 + 2x - 1}{2x - 1} < 1$$
$$\Leftrightarrow y^2 + (x - 2)^2 < 2$$

In summary, this means that the feasible region in which we can choose λ_j is the intersection between the two circles described by Equations 51 and 52 respectively. This is depicted in Figure 9.

$$x^2 + y^2 < 1 \quad (51)$$
$$(x - 2)^2 + y^2 < 2 \quad (52)$$

This equips us with a controller that approximately solves the inverse kinematics in each step. Related

research showed that inverse kinematics based on a Newton-Raphson or Levenberg-Marquardt procedure yields a better tracking and in the case of the latter a better resistance against kinematic singularities at the expense of a higher computational complexity [4].

The inverse kinematics can be used to make the Nao follow an euclidean space trajectory. An open question remains, how such a trajectory must look like to form a stable walking pattern. The LIPM presented in the next section is an attempt to analytically answer this question using a simplified model of a biped robot.

2.4 The LIPM

In the previous section, we have derived a method that computes the joint angles for the Nao lower body to make the system follow an euclidean space input trajectory.

The remaining task is to define such an input trajectory. Kajita et al. proposed a set of differential equation based on the model of an inverted pendulum whose solution is a walking trajectory. In a related research those differential equations were solved to functions of time [4] (See the corresponding report for further details on the LIPM). Furthermore, a foot trajectory was derived as a polynomial satisfying a set of position constraints.

The position vector generated by the LIPM at each time step therefore consists of two sub-positions, one for the torso and one for the foot. Each of these encompasses a three dimensional position in euclidean space and a four dimensional quaternion for orientation. In what follows, this will be referred to as the LIPM-trajectory.

Even though it has been shown, that the LIPM can be used on the Nao to make the robot walk, it is most likely not optimal as the model of a point mass on a stick neglects a lot of real world constraints as for example friction. The subsequent section will present Reinforcement Learning, a technique which can be used to optimize a trajectory with respect to a set of learning goals. The aim will be to eventually apply RL to a model based on a LIPM-trajectory in order to derive a better suited walking trajectory.

2.5 Reinforcement Learning

Reinforcement Learning (RL) is a means for achieving optimal control in *Markov Decision Processes* (MDPs). An MDP consists of a finite set of states $\mathbb{S}$, a set of input actions $\mathbb{A}$ and a transition function $\delta : \mathbb{S} \times \mathbb{A} \times \mathbb{S} \to [0, 1]$ which maps from a state and an input action to a probability distribution over all states. In particular, $\delta(s_0, a, s_1)$ is the probability of transferring to state s_1 given that the current state is s_0 and a was the input to the system. Such a system is said to be *markov* if it has the property that this transfer probability only depends on the current state and the current input and is independent from what has happened before. Typically in RL we consider MDPs with a reward function $r : \mathbb{S} \times \mathbb{A} \to \mathbb{R}$ which maps every state/action pair to a reward value that describes the utility of performing a particular action in a particular state.

A strategy π in such a MDP is a function $\mathbb{S} \to \mathbb{A}$ which associates every state with an action. Generally, the aim is to find a strategy $\pi^\star$ which maximizes the expected reward in a given MDP.

There are multiple ways to interpret the problem of walking as a MDP. Most of these interpretations are conceptually difficult as they require infinite state spaces or violate the markov property. The easiest interpretation is to assume time as a state and let the euclidean space movement during the next time step be the action. As we will see below, this interpretation has the particular advantage that it is easy to translate the LIPM into this framework. Furthermore, it is minimalistic which reduces the number of parameters which have to be learned. However, taking time as state clearly violates the markov property: Whether or not the robot falls over at a point in time t depends on what has happened in the past. A way to overcome this is to use the current position as a state. This can strictly speaking still not guarantee that the markov property holds, as a robot still might or might not fall in one and the same position, depending on its current velocity. This can be compensated by adding the velocity to the state. This shows, that by adding more higher derivatives of the position to the state makes the system more and more markov. At the same time it will also increase the parameter space and make it more difficult to learn.

No matter which interpretation of the walking problem is assumed: The common denominator is the strategy π which is a description of a trajectory in euclidean space. We will now introduce techniques that can be used to optimize π.

Q-Learning A popular RL algorithm is Q-Learning [13, 10]. Q-learning seeks to approximate the utility of every pure action in each state. It uses the concept of temporal differences between the last best estimate of a state/action pair and a the new information.

Let α denote the learning rate, γ the discount factor and r_t be the reward observed in the t-th step. then the following is called the *Q-learning update rule* for a state s and an action a:

$$\begin{aligned} Q_{t+1}(s,a) \leftarrow & Q_t(s,a) \\ & + \alpha \left(r_t + \gamma \max_{a'} Q_t(s,a') - Q_t(s,a) \right) \end{aligned} \tag{53}$$

We shall write Q^π to denote the Q-function of a strategy π. The aim is to learn $Q^\star$, the Q-function corre-

sponding to the optimal strategy $\pi^\star$. Generally, once $Q^\star$ is known it is easy to construct $\pi^\star$.

Regular Q-learning can be very inefficient in some learning scenarios. Consider the case in which a decision taken in a state s at time t results in a very negative reward which does not occur before time $t+1000$. Then regular Q-learning needs at least 1000 learning episodes before this reward has any impact on the Q-values of s. Monte Carlo (MC) techniques are a variant of RL in which complete episodes following a state/action pair (s,a) are sampled and $Q(s,a)$ is updated with the actual reward observed during such a sample rather than with the current temporal difference. By using MC sampling in the above example, the negative reward that was observed after 1000 steps would influence the Q-function estimate immediately. At the same time, it discards valuable information about neighboring states which has already been gathered. Temporal difference learning in turn bases on exactly this information and will be more efficient if such reusable data is available.

Eligibility Traces Eligibility traces are a technique to improve the behavior of RL by providing a compromise between temporal difference based RL such as Q-learning and MC sampling[9].

The idea is to keep a record of how "eligible" or responsible a state is for the observed effect. The algorithm is parametrized by a new variable λ which controls the "decay" of eligibility traces. For high values of λ, the algorithm will exhibit a behavior similar to Monte Carlo methods [10, Chapter 7]. In contrast to MC sampling, it will perform online updating instead of waiting for the end of an episode. For low values of λ, the behavior resembles the temporal difference updates of typical Q-learning.

A practical implementation typically involves a vector e_t each entry of which contains a value reflecting the eligibility of a state/action pair (s,a) at time t, denoted $e_t(s,a)$. If an action a_t is performed in a state s_t, then e_{t+1} is updated by

$$e_{t+1}(s,a) = \begin{cases} \gamma\lambda e_t(s,a)+1 & \text{if } (s,a)=(s_t,a_t) \\ \gamma\lambda e_t(s,a) & \text{otherwise} \end{cases} \tag{54}$$

After that, instead of updating just $Q(s_t,a_t)$, the Q-value for all pairs (s,a) is updated proportionally to their eligibility:

$$\begin{aligned} Q_{t+1}(s,a) \leftarrow & Q_t(s,a) + \alpha e_{t+1}(s,a) \\ & \cdot \left(r_t + \gamma \max_{a'} Q_t(s_t,a') - Q_t(s_t,a_t) \right) \end{aligned}$$

[9] Refer to [10, Chapter 7] for details.

The concept of eligibility traces equips us with a means to apply RL efficiently in domains where the implications of an action become apparent very late. At the same time, in contrast to pure MC learning, we reuse previously gathered information from neighboring states. It is expected that this improves the efficiency of learning in the domain of walking trajectories where an undesirable displacement in the beginning can lead to a latter instability. Yet, also with eligibility traces, the trajectory generation problem is a difficult one due to its high dimensionality. The next section will highlight this problem and present a possible solution based on gradient descent.

The Curse of Dimensionality Robot control and trajectory generation are hard to model as a MDP. In most interpretations of the problem, we will face a continuous action- and state-space. Consider the case in which the current euclidean space position of the system is assumed as its state and the euclidean space displacements are the actions. Assume furthermore that we follow the convention of the LIPM trajectory to keep both moving foot and torso upright all the time and the torso's z coordinate will be kept constant. This leaves us with a state- and action-space consisting of 5 variable dimensions.

One solution to approach the continuity of these dimensions is to discretize them into chunks of width h each, i.e. each dimension is discretized into $\frac{1}{h}$ discrete intervals. That however means, that the Q-function table will consist of $\left(\frac{1}{h}\right)^5$ entries. Even for large values of h that yield a very coarse discretization, this will lead to a large parameter space which is unfeasible for Reinforcement Learning. If we decide to add more information to the state space, it will get even worse as the size of the parameter vector will then grow exponentially.

Nevertheless, there are more techniques different from discretization which can be used as an approximation technique. We can introduce a function $\tilde{Q}^\pi(s,a,\Theta)$ that will replace the standard lookup table commonly applied to store the quality function for a given strategy π. This function is parametrized by a finite parameter vector Θ. The goal is to find a parameter vector $\Theta^\star$ for which the error E defined in equation 55 is minimal[10].

$$E = \left(Q^\pi(s,a) - \tilde{Q}^\pi(s,a,\Theta^\star) \right)^2 \tag{55}$$

Note that discretization is a special case of this approach in which Θ contains the entries of the lookup table. If there exists a gradient of $\tilde{Q}^\pi$ with respect to Θ, which we will denote by $\nabla_\Theta \tilde{Q}^\pi$, then this parameter vector can be improved according to the gradient descent

[10] See [10, Chapter 8] for details.

update rule $\Theta_{t+1} = \Theta_t - \frac{1}{2}\alpha\nabla E(s,\Theta_t)$ where α is the learning rate and E is the error defined in Equation 55. Thus

$$\Theta_{t+1} = \Theta_t + \alpha(Q^\pi(s_t,a_t) - \tilde{Q}^\pi(s_t,a_t,\Theta_t))\nabla_\Theta \tilde{Q}(s_t,a_t,\Theta_t)$$

Of course, generally we do not know $Q^\pi(s_t,a_t)$ but we can replace it with the estimate $r_t + \max_a \tilde{Q}(s_{t+1},a)$; just as it is done in a Q-learning update.

If a linear gradient is used, then on-policy temporal difference learning is guaranteed to converge [11]. Unfortunately, this is not the case for Q-learning. In fact it has been shown to diverge for certain examples.

Let (s,a) be a point to consider. A linear approximation can be created by mapping this pair into a feature space such that a linear combination of these features corresponds to the approximate Q-value at (s,a). This means we need a function

$$\phi : \mathbb{S} \times \mathbb{A} \to \mathbb{R}^m$$

where m depends on the desired feature space. This function ϕ should have the following properties:

- The approximation bias must be low meaning that it must be possible to approximate a large class of functions to ensure that the optimum is learnable.
- The feature space should not be too large as otherwise the curse of dimensionality occurs.
- It must be computationally efficient to determine $\operatorname{argmax}_a \phi(s,a)$ for any given s.
- Additionally, we want ϕ to well approximate the LIPM.

Clearly these requirements are not independent: The more functions ϕ can approximate, the larger the feature space is expected to get.

Taylor-Approximation The approach used in this paper is to discretize the action space and to use a polynomial to approximate the Q-value with respect to the state variable s. Thanks to the discritization, $\operatorname{argmax}_a \phi(s,a)$ can easily be found.

Consider the generalized Taylor series for n-dimensional functions. Taylor's theorem states, that by choosing d high enough, every sufficiently differentiable function can be approximated arbitrarily accurately by Equation 56 given the correct weights Θ_v. This in turn is a linear combination of the product terms. Those terms can thus be seen as the new features corresponding to each state and the weights Θ_v are determined by gradient descent.

$$\mathcal{Q}(s,\Theta) = \sum_{\mathbf{v}\in\mathbb{V}} \Theta_v \prod_{i=0}^{n_s} s_i^{v_i} \tag{56}$$

$$\mathbb{V} = \left\{ \mathbf{v} \;\middle|\; |\mathbf{v}| = n_s \wedge \sum_{x\in\mathbf{v}} x \leq d \right\} \tag{57}$$

For both the gradient descent performed in Section 3 and the Reinforcement Learning procedure used for online optimization, the gradient of $Q(s,a)$ with respect to the parameter vector Θ is needed.

Let an *n-composition*[11] of a number b be a tuple of n non-negative integers whose sum equals b. Then, the set $\mathbb{V}$ shown in Equation 57 denotes the set of all n_s compositions of all numbers smaller or equal to d. These compositions $\mathbf{v} \in \mathbb{V}$ are used to form the exponents of the polynomial constructed in Equation 56. A parameter Θ_v is associated with each composition $\mathbf{v}$.

Furthermore, in Equation 58 we define the set of discretization ranges as all the intervals $[x, x+h]$ of length h between a minimum $\underline{x}$ and a maximum $\overline{x}$ such that x is divisible by h.

$$\mathbb{D} = \{[x,x+h] \mid x \geq \underline{x}, x+h \leq \overline{x}, x = yh\} \tag{58}$$

Algorithm 1 Q-learning update with Taylor approximation and eligibility traces. s is the state we were before performing action a. s' is the state we land in after having performed action a. r is the reward observed after having performed a in s. Θ is the parameter vector and e is the vector of eligibility factors.

1: **function** UPDATE-Q(s, s', a, r, Θ, e)
2: $\quad a^\star \leftarrow \operatorname{argmax}_{a'} Q(s', a', \Theta)$
3: $\quad \delta \leftarrow r + \gamma Q(s', a^\star, \Theta) - Q(s, a, \Theta)$
4: $\quad e \leftarrow \gamma\lambda e + \nabla_\Theta Q(s, a, \Theta)$
5: $\quad \Theta \leftarrow \Theta + \alpha\delta e$
6: $\quad$ **return** Θ, e

This equips us with all we need to construct the entire Q-function (Equation 59). The new parameter vector Θ has size $|\mathbb{D} \times \mathbb{V}|$ and we will refer to its entries by $\Theta_{d,v}$. The notation Θ_d shall refer to the subvector of Θ which contains all the entries associated with the parameter d. Equation 60 is the partial derivative of this Q-function with respect to a parameter $\Theta_{x,v}$. The gradient $\nabla_\Theta Q(s,a,\Theta)$ is the vector of all those partial derivatives.

[11] Note that this is not quite equal to the notion of a composition used in number theory: We adapted this idea to suit our needs.

$$Q(s, a, \Theta) = \begin{cases} \mathcal{Q}(s, \Theta_x) & \text{if } \exists x \in \mathbb{D} : a \in x \\ 0 & \text{otherwise} \end{cases} \quad (59)$$

$$\frac{\partial Q(s, a, \Theta)}{\partial \Theta_{x,v}} = \begin{cases} \prod_{i=0}^{n_s} s_i^{v_i} & \text{if } a \in x \\ 0 & \text{otherwise} \end{cases} \quad (60)$$

Inserting the functions Q and $\nabla_\Theta Q$ into Algorithm 1 yields the final Q-function update rule. In the next section, we will be explain how the same techniques can be applied to gradient descent supervised learning. This will enable us begin Reinforcement Learning with a biased Q-function.

3 Learning the LIPM

No matter what interpretation of the world we use to model it as a MDP, it is generally not realistic to learn a walk from scratch. We suggest using the LIPM as an initial bias for RL. Below we will present a technique to achieve this using the interpretation of time as state. The aim is to derive a Q-function such that the greedy strategy according to this function is to follow the LIPM trajectory. This means that for every state s, the maximum $\text{argmax}_a Q(s, a)$ must be the action which would have been executed in state s if we followed the LIPM trajectory. In what follows $\mathcal{L}(t)$ shall denote the position that the robot should assume at time t according to the LIPM.

If time is used as the state, this requirement can be satisfied easily: We will generate a large sample om points, such that for each point

1. a random time t within the duration of a step is chosen
2. $a^\star = \mathcal{L}(t) - \mathcal{L}(t - 20ms)$ is calculated
3. a random action a with in the range of feasible actions is generated
4. the reward of a in t is calculated as $r = -c \cdot |a - a^\star|^2$ for some constant c

This procedure ensures, that within a state the optimal action always gets the value 0 whilst all the other actions receive negative values. The sample shown above can be used for supervised learning by stochastic gradient descent [3]: For each point (t, a, r), the parameter vector is updated

$$\Theta \leftarrow \Theta + \alpha \left(r - Q(t, a, \Theta)\right) \nabla_\Theta Q(t, a, \Theta)$$

where the functions Q and $\nabla_\Theta Q$ are the ones we derived in Section 2.5 for the Reinforcement Learning update. This process is then repeated until convergence is achieved. Figure 10 shows an example how the resulting Q-function looks like at state $t = 0$.

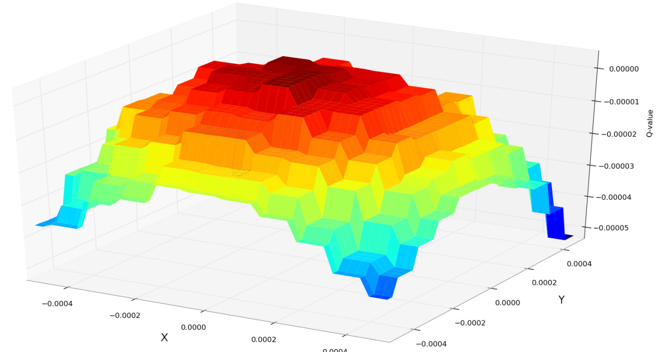


Figure 10: The Q-function for $s = 0$.

4 Experiments and Results

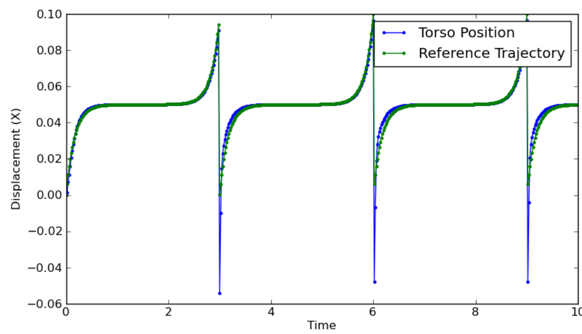
In the preceding sections, all the techniques required for implementing a self optimizing walk have been assembled: A controller, which allows us to solve the inverse kinematics on the Nao, a Reinforcement Learning algorithm to optimize a trajectory and a technique to cast the LIPM trajectory into a form that can be used by Q-learning. This section will present the results of several experiments that have been conducted in order to assess the performance of the proposed techniques on each of these subproblems.

Controller First, we will assess the performance of the controller presented in Section 2.3. This measurement was done using the Webots simulator[12] which provides a model of the Nao robot. The trajectory fed into the system is a LIPM based trajectory with a steptime $t_s = 3$ and a foot offset of $s_x = 0.05$, $s_y = 0.05$. The torso z-position was set to $z = 0.25$ and the maximal z-displacement of the foot was set to $z_m = 0.02$.

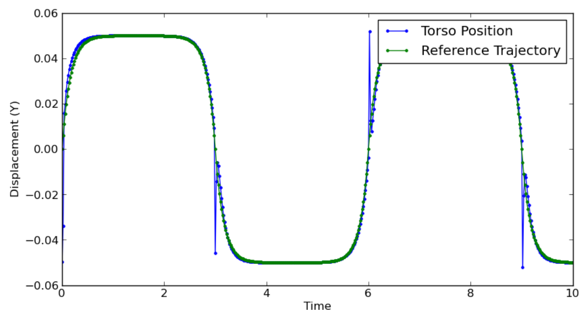
Figure shows the trajectory of the Nao together with the respective reference prescribed by the LIPM. The overall tracking seems to be fine except for the point between two steps where a significant deviation from the reference trajectory can be observed. The tracking error plotted in Figure confirms this impression. Furthermore, in this graph, multiple pole settings are compared with each other. The results confirm the theoretical predictions from Section 2.3 as the system indeed turns out to be stable if the poles are chosen in the predicted range.

The peaks in the tracking error can be explained by the fact that the LIPM is not continuous between two steps whilst the controller relies on the assumption that the system does not change rapidly between two subsequent states. Possibly, this problem can be solved by manually adjusting the state vector between two steps.

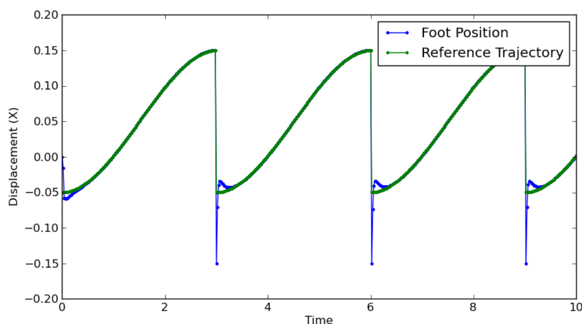
[12] http://www.cyberbotics.com/overview



(a) displacement of torso along the x-axis



(b) displacement of torso along the y-axis



(c) displacement of foot along the x-axis

Figure 11: Reference trajectories and the result of applying the linear controller to various body parts of the Nao.

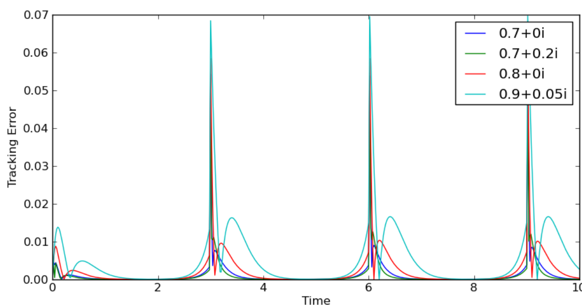


Figure 12: Tracking error for various pole settings.

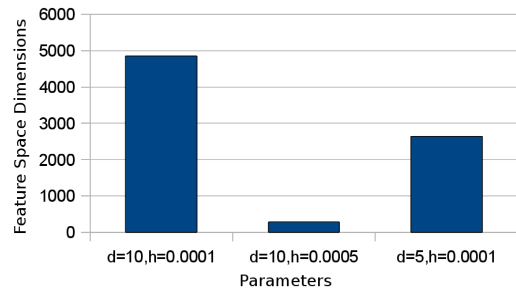


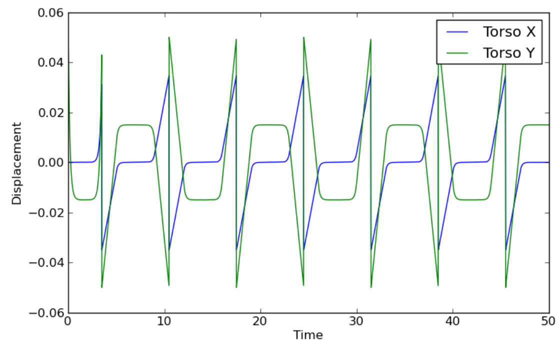
Figure 13: Comparison of feature space sizes for different parameter settings. The feature space size was measured as the number of parameters in the vector Θ.

Learning the LIPM with Gradient Descent In Section 3 we devised a procedure for creating a Q-function which reflects the LIPM trajectory. However, it could not be shown theoretically, which hyper-parameters are necessary to achieve a good representation of the LIPM at minimal size of the state space. For the purpose of this project, only the x and y movement of the torso has been taken into account. Yet, for the foot-displacement or the joint motion, the learning would work accordingly.

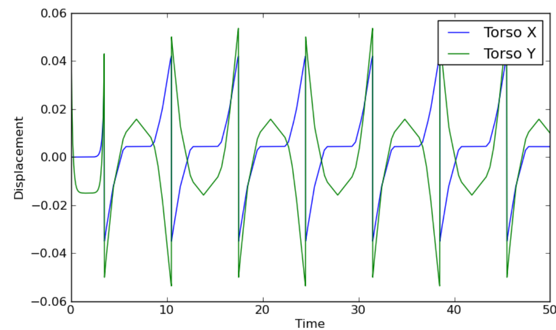
In Figure 14 the resulting LIPM based body trajectory for different parameter settings is compared to the original LIPM trajectory which uses a single support time $t_s = 4s$, a double support time $t_d = 3s$ and a foot offset of $s_x = 0.035$, $s_y = 0.2$. The torso z-position was set to $z = 0.3$ and the maximal z-displacement of the foot was set to $z_m = 0.05$. Figure 10 shows an exemplary slice of the resulting Q-function at state $t = 0$.

A curious result is, that a coarse discretization as the one used in Figure 14c seems to yield better results than attempts with a finer granularity. One possible explanation is, that the feature space of finer discretizations is so big that convergence only takes place very slowly. This would mean that Figures 14b and 14d are actually not fully converged. This suspicion is supported by Figure 13 which compares the feature space size of the various parameter settings tested in Figure 14. At the same time it is apparent, that the trajectory in Figure 14c does not capture the smoothness of the LIPM trajectory. Figure 14b was superior in capturing the movement along the X-axis due to the finer discretization.

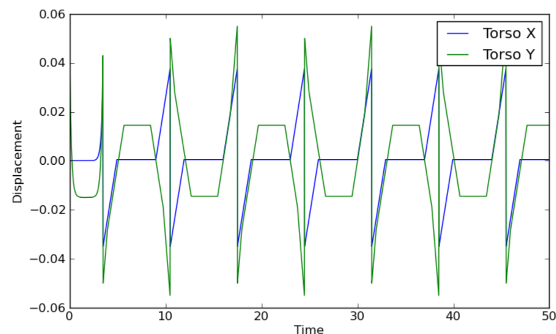
Reinforcement Learning The RL procedure presented in section 2.5 was applied on the Q-function derived above with $h = 0.0001$ and $d = 10$. The robot's initial trajectory was hence the one shown in Figure 14b. For testing purposes, the state feedback only consisted in a negative reward of -100 for falling over. The trajectory resulting from the learning is shown in Figure 15. It is



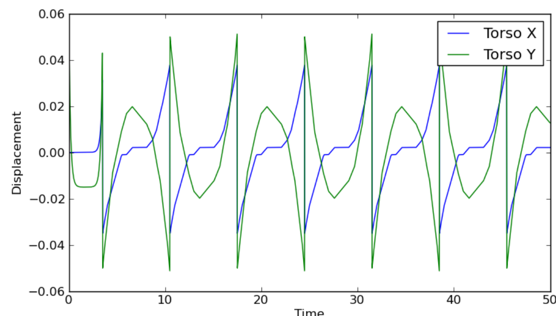
(a) Original



(b) $h = 0.0001$, $d = 10$



(c) $h = 0.0005$, $d = 10$



(d) $h = 0.0001$, $d = 5$

Figure 14: The original LIPM and the result of gradient descent with different parameters compared. The parameter h is the size of one discretized window, d is the degree of the polynomial used for the Taylor approximation.

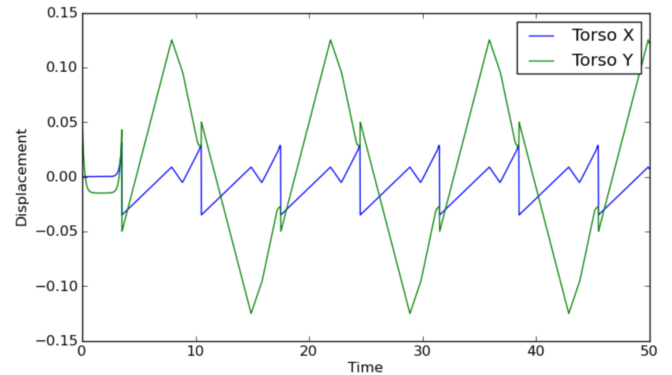


Figure 15: Trajectory learned by RL.

clearly visible, that the displacement of the body along the y axis is wider than in the original LIPM. In fact the resulting trajectory does not at all look like a walk. Yet, it does satisfy the only criterion imposed which was not to fall.

5 Conclusion and Discussion

This paper investigated an approach on biped robot walking based on solving three subproblems. The first one is the control of the lower body of the Nao robot in euclidean space including inverse kinematics. The second step is generation of a euclidean space trajectory in a representation which can be improved by learning algorithms. The final step is to use feedback from the environment to actually improve the trajectory.

We were able to give a detailed analysis of the mechanics involved in the control of the Nao's lower body and derived a Jacobian which can be used to control multiple overlapping chains at once. A technique which is not restricted to the Nao robot. It was furthermore shown that the control system devised in Section 2.3 can efficiently track a LIPM based trajectory. Unfortunately, the controller needs to be improved concerning its behavior between two subsequent steps. Furthermore, Remco Bras tested the Levenberg-Marquardt method on the Nao which provides a significantly better tracking performance and less problems with kinematic singularities at the expense of slightly higher computational complexity [4]. It might be fruitful to investigate the possibility of a hybrid solution between Levenberg-Marquardt and a linear controller as presented in this paper to achieve a better tradeoff between computational complexity and tracking quality.

We presented a technique based on Taylor approximation that is able to learn the characteristics of a LIPM trajectory with a comparably low state-space. However, it must be noted, that the Taylor approach is just one approximation technique amongst many. It would be

interesting to see, how well the LIPM can be approximated if for instance a Neural Network is applied instead. Also, it was thus far necessary to use a discretization approach to approximate the action space because of the need to efficiently calculate $\text{argmax}_a Q(s,a)$. With techniques like non-linear programming, however, there are tools available that allow performing such a calculation in more sophisticated functions. This will allow us to use a more powerful class of approximation schemes in action-space. It is likely that such an approach will lead to smaller action-spaces and therefore to more efficient learning.

We implemented a proof of concept walk based on the Reinforcement Learning procedure. For simplicity we assumed the state to be time and the only feedback to be a negative reward when the Nao falls over. It turned out, that the procedure was indeed able to modify the walk to something that successfully avoided falling over, however, the result did not at all look like a gait anymore. Two improvements have been proposed which are beyond the scope of this project: First of all the Q-learning procedure was set up such that we can use multidimensional states. It should therefore not be difficult to use the current position of the Nao's lower body as its state instead of time. This will make the representation "more markov" and we would expect a better RL performance. At the same time, we should use a more sophisticated feedback. A possible extension would be to use the distance of the center of pressure from the edges of the support area as a measure of stability.

Despite these shortcomings, the technique presented above provides a viable basis for implementing walking schemes like the LIPM on the Nao and adding feedback to optimize this trajectory online.

References

[1] Aldebaran. Nao Documentation, 2009.

[2] J. Angeles. *Fundamentals of Robotic Mechanical Systems.* Springer Science+Business Media,LLC, 2007.

[3] C. M. Bishop. *Pattern Recognition and Machine Learning.* 2006.

[4] R. Bras. A Walk for the Nao. Technical report, Maastricht University, 2011.

[5] J. J. Craig. *Introduction to Robotics - Mechanics & Control.* Addison-Wesley, 2nd edition, 1989.

[6] D. Gouaillier, V. Hugel, P. Blazevic, C. Kilner, J. Monceaux, P. Lafourcade, B. Marnier, J. Serre, and B. Maisonnier. The NAO humanoid: a combination of performance and affordability. *CoRR*, abs/0807.3, 2008.

[7] J. W. Jason Kulk. A Low Power Walk for the NAO Robot. In *Australasian Conference on Robotics and Automation*, 2008.

[8] S. Kajita, F. Kanehiro, K. Kaneko, K. Yokoi, and H. Hirukawa. The 3D linear inverted pendulum mode: a simple modeling for a biped walking pattern generation. In *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 1, pages 239–246, Oct. 2001.

[9] O. Khatib. CS223A: Introduction to Robotics, 2008.

[10] R. S. Sutton and A. G. Barto. *Reinforcement Learning : An Introduction*, volume 3. MIT Press, Sept. 1998.

[11] J. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, May 1997.

[12] R. J. Vaccaro. Digital Control: A State-Space Approach, 1995.

[13] C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, May 1992.