A walk for the Nao

Remco Bras

July 4, 2011

Contents

1 Introduction

Many papers on biped walking have been published. In some of these, researchers describe an entire walking system. For instance, the paper by Hirai et al. [7] on Honda's P2 robot includes not only a description of its walking-related features and systems but also a description of the robot hardware. Gouallier et al. [6] have written a similar paper on the Nao robot.

In other papers, researchers have addressed specific aspects of a walking system. We will consider three of these aspects. The first aspect, trajectory generation, is the process of finding a trajectory that specifies how a robot should move while walking. The second, trajectory execution, considers how a robot can be made to execute a given trajectory. The third and final aspect we consider is stability measurement, which considers how the stability or balance of a robot can be quantified and measured.

In a number of papers, researchers have proposed methods to generate the trajectory for a walking robot. Many of these methods use a variation of the following simplified model. In this model, the robot is represented as an inverted pendulum. The base of this pendulum, i.e. the point where it touches the ground, represents the robot's stationary foot. The other end of the pendulum, moving through the air, represents the rest of the robot. In some applications of this idea, such as Kajita et al's Linear Inverted Pendulum Mode(LIPM)[8, 9], the base of the pendulum is stationary. In others, such as Kajita et al.'s work on using preview control with the LIPM[10] and Mitobe et al.'s work on zero-moment point-based control[11], the position of the pendulum's base is used as a control input to influence the motion of the rest of the robot.

In order to make a robot execute a trajectory generated by one of the methods we described above, a controller needs to move the robot's motors correctly. On some robots, such as the Nao, a user's control system does not directly control the motors, but specifies for each joint the angle it should move to. The hardware then controls the motors in order to reach this given angle. In these cases, the control system's primary responsibility is to provide the hardware with the correct joint angles. To do so, it needs to convert the trajectory specified in terms of Euclidean positions to a trajectory specified in terms of joint angles. One method to do this is to calculate for each position given by the trajectory the corresponding joint angles. This process is called Inverse Kinematics. Analytical solutions to this problem are possible only in special cases. These cases are described in textbooks on robotics and a number of papers. General-purpose methods, such as those described by Buss [3] are based on numerical optimization techniques.

A number of methods to measure the stability or balance of a robot have been published. The most influential of these is the method using the zero-moment point, as described by Vukobratović et al. [16]. In this method, one first finds the zero-moment point. Once this point has been found, its position relative to the area of support, the convex hull of the robot's feet, indicates whether or not the robot is stable, and if so, how stable it is. Many methods to measure stability are similar to the method described above, but differ in how they define the zero-moment point. Vukobratović et al. define this point as the point where the ground reaction force on the robot's foot would have to act to keep the robot's foot stationary. Other authors, such as Goswami [5], refer to this point as the FRI point. According to these authors, the term "zero-moment point" is synonymous with "center of pressure", a name for the point where the ground reaction force actually acts. Yet other researchers, such as Poskriakov [13] define the zero-moment point in terms of static equilibrium of the entire robot. Defined this way, the zero-moment point is similar to the ZRAM (zero rate of change of angular momentum) point defined by Goswami [4].

In this paper, we consider how a walking system can be implemented on the Nao. To do so, we will consider each of the three aspects we have described above in turn. Thus, we will begin by considering how a trajectory for the Nao's walking motion may be generated. To do so, we will use the three-dimensional version of the LIPM described by Kajita et al.[9]. Then, we will consider how this trajectory may be executed. As we noted above, the Nao has internal controllers that move the Nao's joints to specified angles. Because of this, we will primarily be concerned with Inverse Kinematics, using some of the methods given by Buss[3]. Finally, we will consider how we can measure the stability of the walking motion we generate and other aspects of the performance of our walking system. To do so, we will be using the scheme described above, where we will use the center of pressure instead of the zero-moment point as the center of pressure is easier to measure on the Nao. We conclude this paper by briefly discussing the performance of our system and reconsidering each of the three aspects we have introduced here.

2 Trajectory Generation

2.1 Introduction

Trajectory generation is finding a trajectory specifying how the robot should move. The trajectory is a function over time that gives the positions in Cartesian space of all independently controlled parts of the robot. For biped walking, the independently controlled parts are the torso and the moving foot.

One can distinguish two phases in walking by considering the moving foot. When the foot is at rest on the ground, the motion is in the double-support phase. When the foot is moving through the air, the motion is in the single-support phase. One can generate most of the trajectory for each phase without considering the other. The only requirement is that the two trajectories together form a smooth function, that is, that the combined function and its derivative are continuous.

To generate the trajectory for each phase, one needs a suitable model. For the single-support phase, the literature provides many models. This work uses the Linear Inverted Pendulum Mode ("LIPM") of Kajita et al. [8, 9]. The LIPM is based on an inverted pendulum. The robot's non-moving foot is considered the base of the pendulum, with the robot's torso at the top. For the double-support phase, this model would not apply, since we would have a pendulum with two bases. Following Kajita et al., we move the robot's torso with constant velocity during the double-support phase.

The next subsection explains in more detail how the LIPM is applied to generate steps.

2.2 Walk structure and parameters

Before using the LIPM, one must consider how a robot moves while walking. At every moment, one of the robot's feet is on the ground. This foot alternates with each step. That is, one step is done with the left leg, then the next with the right leg and so on. A natural boundary between steps is when this leg changes, that is, when the leg that was previously supporting the robot leaves the ground. The double-support phase, where the robot's feet are both on the ground, can be assigned to any step, as either leg could be viewed as supporting the robot. We use the convention that the supporting leg is changed at the midpoint of the double-support phase. Due to this convention, the robot's torso always starts and ends each step exactly in the middle between the legs. Thus, the step consists of first moving the torso towards the support leg, then moving through the single-support phase and finally moving the torso away from the support leg.

To determine how the robot should move in detail, we use a set of parameters that define the robot's trajectory. As previously mentioned, the robot's torso starts a step midway between the robot's legs. Two parameters are given by the position of the robot's support leg at the start of a step. This position, (s_x, s_y), is taken in a coordinate system where the initial position of the robot's torso is at the origin, the x-axis points in the direction of walking, and the y-axis points horizontally to the left of the robot. These parameters govern the robot's step length, that is, the distance it travels in a given step. Since the torso starts and ends exactly between the two legs and the support leg is initially s_x in front of the torso, the other leg must be s_x in front of the torso when the step ends. Similarly, the support leg must then be s_x behind the torso. As such, the torso moves forward from 0 to $2s_x$, while the moving leg moves from $-s_x$ to $3s_x$. The initial and final horizontal positions of the torso are the same, both being zero. The same holds for the moving leg, with a value of $-s_y$.

While s_x and s_y govern step length, they do not influence how long a step takes.

The two parameters t_s and t_d specify the time a step takes. Of these parameters, t_s is the time spent in the single-support phase, while t_d is the time spent in the double-support phase. From these parameters and the fact that a step consists of two double-support parts and one single-support part, it follows that from 0 to $\frac{t_d}{2}$ after starting the step, the robot is in double-support. From $\frac{t_d}{2}$ to $\frac{t_d}{2} + t_s$, it is in single-support. Finally, from $\frac{t_d}{2} + t_s$ to $t_d + t_s$, it is in double-support. The step as a whole takes $t_s + t_d$.

The previously mentioned parameters govern motion in the (x,y)-plane and step time. In the vertical (z-) direction, the LIPM moves the torso through a plane. Since we assume that the ground is horizontal, we constrain this further to move the torso through a horizontal plane. Hence, the z-coordinate of the torso is a constant, the parameter z_c. This is measured with 0 on the ground and positive z pointing up. The height of the moving foot is not constant, but its maximum height is. This maximum is given by the parameter z_m, which is measured similarly to z_c.

The parameters described in this subsection imply a set of constraints on the generated motion. Before considering how a motion satisfying these constraints can be generated, it is instructive to reiterate the constraints. The entire set of constraints is given below in 1. The position functions are given in a coordinate system with the ground projection of the torso's initial position at the origin, positive x forward, positive y to the left and positive z upward. The subscript "t" refers to the robot's torso, the subscript "f" to the robot's moving foot.

$$\begin{aligned}
x_t(0) &= 0 \\
x_t(t_s + t_d) &= 2s_x \\
x_f(\tau) &= -s_x \text{ for } 0 \leq \tau \leq \frac{t_d}{2} \\
x_f(\tau) &= 3s_x \text{ for } t_s + \frac{t_d}{2} \leq \tau \leq t_s + t_d \\
y_t(0) &= y_t(t_s + t_d) = 0 \\
y_f(0) &= y_f(t_s + t_d) = -s_y \\
z_t(\tau) &= z_c \\
z_f(\tau) &= 0 \text{ for } 0 \leq \tau \leq \frac{t_d}{2} \\
z_f(\tau) &= 0 \text{ for } t_s + \frac{t_d}{2} \leq \tau \leq t_s + t_d \\
max_{\frac{t_d}{2} < \tau < \frac{t_d}{2} + t_s} \{z_f(\tau)\} &= z_m
\end{aligned} \tag{1}$$

2.3 Generating the trajectory

This subsection describes how a trajectory satisfying the constraints of the previous subsection can be generated. The motion of the robot's torso and of the robot's foot are considered in separate subsubsections. The torso motion is determined from the LIPM's differential equations, while the foot motion is generated by fitting polynomials to the constraints.

2.3.1 Torso motion

Before considering the single-support phase, let us consider what we already know about the torso motion. First, we have derived a set of constraints in the previous subsection, which is given in the set of equations 1. Second, we know that the torso moves with constant velocity in the double-support phase. We do not yet know the value of this velocity. Hence, let us refer to the velocity of the torso during the first double-support part as $(\alpha_x, \alpha_y, 0)$. By considering the next step, we can find the

velocity of the torso during the final double-support part of the current step. In the forward direction, the torso always moves with a non-negative velocity. Hence, we know that the forward velocity of the torso at the start of the next step will have the same sign, even if the supporting leg is different. Therefore, the forward velocity during the final double-support part of this step must also be α_x. Horizontally, the torso must move towards the other leg at the start of the next step. Since this leg is on the other side of the origin, the horizontal velocity at the end of the step must be the negative of the horizontal velocity during the first double-support part, that is, $-\alpha_y$.

Now that we have expressions for the velocity during the double-support phase, we can derive the torso's position at the moments when the motion switches from the double-support phase to the single-support phase. This position is given by (β_x, β_y, z_c), where $\beta_x = \frac{\alpha_x t_d}{2}$ and $\beta_y = \frac{\alpha_y t_d}{2}$. [1] The torso position at the end of the single-support phase can be derived from the torso's final position at the end of the step and the velocities we derived previously for the final double-support part. In the x-direction, since the torso ends the step at $2s_x$ and moves with a velocity of α_x during the final double-support part, we find that the torso ends the single-support phase at $2s_x - \beta_x$. In the y-direction, we find the torso ends the single-support phase at β_y, from similar considerations.

The constraints derived in the previous two paragraphs, as well as those given in the last subsection, are summarized below in equations 2 through 11. Notice that the subscript "t" has been dropped, as we only consider the torso here. Also note that these constraints, particularly equations 3, 4, 6, 8, 9 and 10, specify initial and final position and velocity conditions for the single-support phase.

$$x(0) = 0 \tag{2}$$

$$x(\frac{t_d}{2}) = \beta_x = \frac{\alpha_x t_d}{2} \tag{3}$$

$$x(\frac{t_d}{2} + t_s) = 2s_x - \beta_x \tag{4}$$

$$x(t_s + t_d) = 2s_x \tag{5}$$

$$\dot{x}(\tau) = \alpha_x \text{ for } 0 \leq \tau \leq \frac{t_d}{2} \vee \frac{t_d}{2} + t_s \leq \tau \leq t_d + t_s \tag{6}$$

$$y(0) = y(t_d + t_s) = 0 \tag{7}$$

$$y(\frac{t_d}{2}) = y(\frac{t_d}{2} + t_s) = \beta_y \tag{8}$$

$$\dot{y}(\tau) = \alpha_y \text{ for } 0 \leq \tau \leq \frac{t_d}{2} \tag{9}$$

$$\dot{y}(\tau) = -\alpha_y \text{ for } \frac{t_d}{2} + t_s \leq \tau \leq t_d + t_s \tag{10}$$

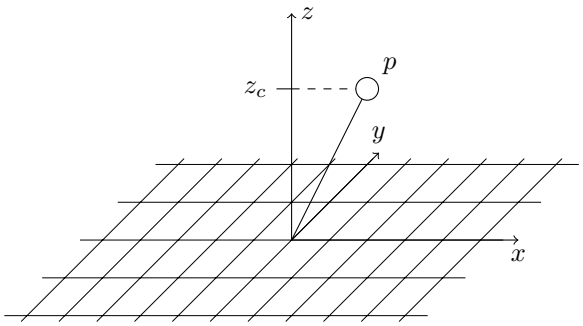
$$z(\tau) = z_c \tag{11}$$

As we now have initial and final conditions for the single-support phase, we can describe the trajectory in this phase. We will first consider the differential equations implied by the LIPM. Thereafter, we will show how the conditions we derived earlier can be satisfied by the solution of these differential equations.

To derive the LIPM, Kajita et al. [9], use the dynamics of an inverted pendulum, such as the one shown in Figure 1. The base of this pendulum, which is shown at the origin, represents the stationary foot of the robot. The legs of the robot are assumed to be massless and the remaining mass of the robot is concentrated at point P. To simplify the dynamics, Kajita et al. constrain the point P to move through

[1] Recall that the torso moves with constant velocity during the double-support phase

Figure 1: An inverted pendulum, with its base at the origin and the top at p



a plane. With the additional assumption that this plane is horizontal, that is, that the height of the point P is a constant, the dynamics take the form of Equations 12 and 13. In these equations, g is the acceleration due to gravity and m is the mass of the robot. The variables u_p and u_r are derived from torques. We ignore u_p and u_r since their omission simplifies the equations and implies that the dynamic balance of the model is never lost. We also modify the equations slightly by moving the base of the LIPM's inverted pendulum from the origin to $(s_x, s_y, 0)$, as required by the parameters we have discussed in the previous subsection. The equations then reduce to equations 14 and 15.

$$\ddot{x} = \frac{g}{z_c}x + \frac{1}{mz_c}u_p \tag{12}$$

$$\ddot{y} = \frac{g}{z_c}y - \frac{1}{mz_c}u_r \tag{13}$$

$$\ddot{x} = \frac{g}{z_c}(x - s_x) \tag{14}$$

$$\ddot{y} = \frac{g}{z_c}(y - s_y) \tag{15}$$

Equations 14 and 15 are identical modulo a substitution of variable names. This implies that their solution will also take the same general form. The differences between the two position functions will then be due to differences in the initial and final conditions we have described previously. Since the general solution to the equations 14 and 15 is identical, let us first consider this general solution. One can verify using calculus that the solution $p(t)$ to $\ddot{p} = \frac{g}{z_c}(p - s_p)$, for $t \geq \frac{t_d}{2}$, is given by equation 16, with A_p and B_p constant. The initial conditions for this equation are again identical in both the x- and y-direction, being given by $p(\frac{t_d}{2}) = \beta_p$ and $\dot{p}(\frac{t_d}{2}) = \alpha_p$. The solution for the constants A_p and B_p , given these conditions , is given by equations 17 and 18.

$$p(t) = A_p e^{\sqrt{\frac{g}{z_c}}(t - \frac{t_d}{2})} + B_p e^{-\sqrt{\frac{g}{z_c}}(t - \frac{t_d}{2})} + s_p \tag{16}$$

$$A_p = \frac{\beta_p - s_p + \alpha_p\sqrt{\frac{z_c}{g}}}{2} \tag{17}$$

$$B_p = \frac{\beta_p - s_p - \alpha_p\sqrt{\frac{z_c}{g}}}{2} \tag{18}$$

To complete our considerations of the LIPM, we have to determine α_p and β_p in both the x- and y-directions using the final conditions of equations 4,6,8 and

10. Before doing so, let us define $\gamma_p(t)$ as $e^{\sqrt{\frac{g}{z_c}}(t-\frac{t_d}{2})}$, so equation 16 simplifies to equation 19. In the x-direction, the final condition of equation 4 then reads $A_x\gamma_x(\frac{t_d}{2}+t_s)+\frac{B_x}{\gamma_x(\frac{t_d}{2}+t_s)}+s_x = 2s_x-\beta_x$. Solving for gamma, one obtains equation 20. Substituting the values given by equations 17 and 18 for A_x and B_x, one obtains equation 21. The first solution clearly does not correspond to any real instant in time, while the second is what we require. Solving for α_x in the second solution, one obtains equation 22. To complete the derivation, one can substitute this result in $\beta_x = \frac{\alpha_x t_d}{2}$ to obtain 23. If one wishes, one can obtain equation 24 by substituting 23 back into 22. A similar procedure yields the solutions 25 and 26 for α_y and β_y. One can verify that the function of equation 16, using the values of A_p, B_p, α_p and β_p given in the other equations, satisfies all the initial and final conditions derived previously.

$$p(t) = A_p\gamma_p(t) + \frac{B_p}{\gamma_p(t)} + s_p \tag{19}$$

$$\gamma_x(\frac{t_d}{2}+t_s) = \frac{s_x-\beta_x \pm \sqrt{(s_x-\beta_x)^2-4A_xB_x}}{2A_x} \tag{20}$$

$$\gamma_x(\frac{t_d}{2}+t_s) = -1 \vee \gamma_x(\frac{t_d}{2}+t_s) = \frac{\alpha_x\sqrt{\frac{z_c}{g}}+s_x-\beta_x}{\alpha_x\sqrt{\frac{z_c}{g}}-s_x+\beta_x} \tag{21}$$

$$\alpha_x = \frac{(s_x-\beta_x)(\gamma_x(\frac{t_d}{2}+t_s)+1)}{\sqrt{\frac{z_c}{g}}(\gamma_x(\frac{t_d}{2}+t_s)-1)} \tag{22}$$

$$\beta_x = \frac{s_x t_d(\gamma_x(\frac{t_d}{2}+t_s)+1)}{2\sqrt{\frac{z_c}{g}}(\gamma_x(\frac{t_d}{2}+t_s)-1)+t_d(\gamma_x(\frac{t_d}{2}+t_s)+1)} \tag{23}$$

$$\alpha_x = \frac{2s_x(\gamma_x(\frac{t_d}{2}+t_s)+1)}{2\sqrt{\frac{z_g}{g}}(\gamma_x(\frac{t_d}{2}+t_s)-1)+t_d(\gamma_x(\frac{t_d}{2}+t_s)+1)} \tag{24}$$

$$\alpha_y = \frac{2s_y(\gamma_y(\frac{t_d}{2}+t_s)-1)}{2\sqrt{\frac{z_c}{g}}(\gamma_y(\frac{t_d}{2}+t_s)+1)+t_d(\gamma_y(\frac{t_d}{2}+t_s)-1)} \tag{25}$$

$$\beta_y = \frac{s_y t_d(\gamma_y(\frac{t_d}{2}+t_s)-1)}{2\sqrt{\frac{z_c}{g}}(\gamma_y(\frac{t_d}{2}+t_s)+1)+t_d(\gamma_y(\frac{t_d}{2}+t_s)-1)} \tag{26}$$

Though the above discussion is sufficient to obtain an algebraic expression for the position functions of the robot's torso, these expressions were found to be unsuited to numerical evaluation. In an appendix, we discuss these numerical problems and how we avoided them. In the main text, we now continue with the motion of the robot's foot.

2.3.2 Foot motion

As we mentioned in the introduction, we generate the foot trajectory by fitting polynomials to the constraints it should satisfy. These constraints, taken from the set of equations 1 used earlier, are given by equations 27 through 31. The constraints give initial and final positions for the foot motion in each direction. The constraints also imply that the foot starts and ends at rest, that is, with zero velocity. We will use the forward direction to illustrate how polynomials can be fit to these constraints. The other directions will be left as an exercise to the reader, though we will give solutions for the position functions.

$$x(\tau) = -s_x \text{ for } 0 \leq \tau \leq \frac{t_d}{2} \tag{27}$$

$$x(\tau) = 3s_x \text{ for } t_s + \frac{t_d}{2} \leq \tau \leq t_s + t_d \tag{28}$$

$$z(\tau) = 0 \text{ for } 0 \leq \tau \leq \frac{t_d}{2} \tag{29}$$

$$z(\tau) = 0 \text{ for } t_s + \frac{t_d}{2} \leq \tau \leq t_s + t_d \tag{30}$$

$$max_{\frac{t_d}{2} < \tau < \frac{t_d}{2} + t_s} \{z(\tau)\} = z_m \tag{31}$$

For the forward direction, we have two position constraints, given by 27 and 28, which each imply a velocity constraint. Since we have 4 constraints and each constraint is a linear equation in the coefficients of our polynomial, we will use a polynomial of the form $x(t) = at^3 + bt^2 + ct + d$, where t runs from 0 at the start of the foot's actual motion to t_s at the end of that motion. This polynomial's derivative is clearly given by $\dot{x}(t) = 3at^2 + 2bt + c$. Filling in the constraints, we find the system of equations 32 through 35. The equations 32 and 33 give a solution for the coefficients c and d. The system then reduces to the two equations 34 and 35, which could be solved either numerically or analytically. Equation 36 gives the analytical solution for the position function. Equations 37 and 38 were obtained by applying a similar procedure to the horizontal and vertical constraints. For the vertical direction, we used the additional constraint that the maximum required by equation 31 occurs at $t = \frac{t_s}{2}$.

$$x(0) = d = -s_x \tag{32}$$

$$\dot{x}(0) = c = 0 \tag{33}$$

$$x(t_s) = at_s^3 + bt_s^2 + ct_s + d = 3s_x \tag{34}$$

$$\dot{x}(t_s) = 3at_s^2 + 2bt_s + c = 0 \tag{35}$$

$$x(t) = -\frac{8s_x}{t_s^3}t^3 + \frac{12s_x}{t_s^2}t^2 - s_x \tag{36}$$

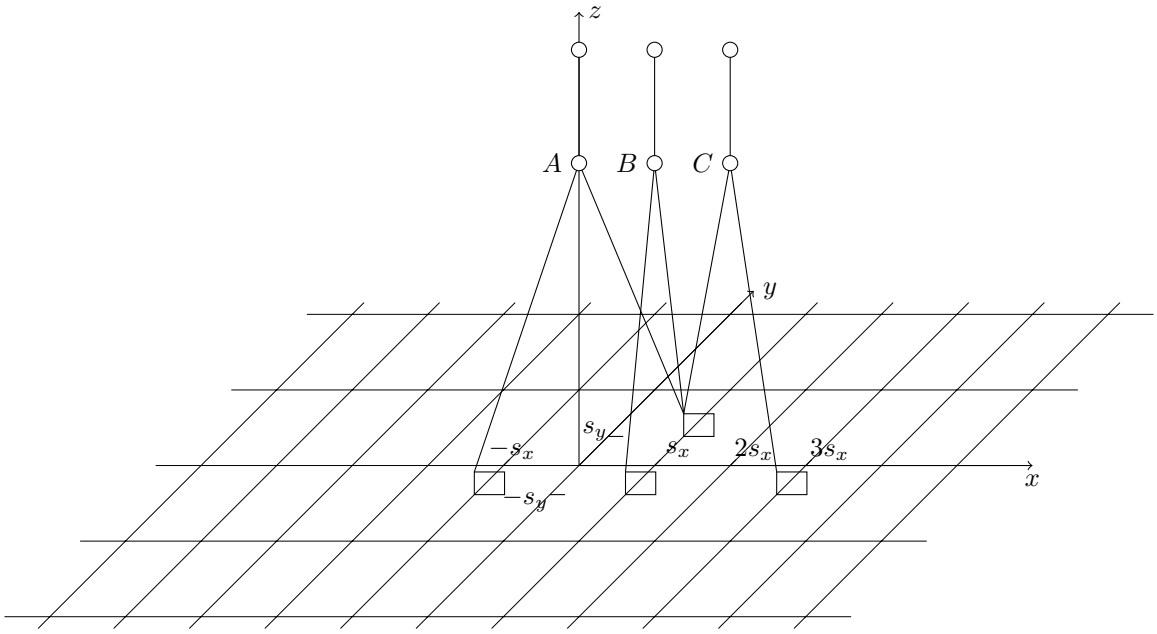
$$y(t) = -s_y \tag{37}$$

$$z(t) = \frac{16z_m}{t_s^4}t^4 - \frac{32z_m}{t_s^3}t^3 + \frac{16z_m}{t_s^2}t^2 \tag{38}$$

2.4 Handling special steps and tweaking the walk

In this subsection, we consider a number of minor changes to the trajectories described earlier. First, we consider how the trajectories change if we require the robot to start and end a sequence of steps with its feet next to each other. Thus, the robot should start and end a series of steps in the pose B of Figure 2. Therefore, in the first step, the robot moves from pose B to pose C. In the last step, it moves from pose A to pose B. All the steps in between are ordinary steps, in which the robot moves from pose A to pose C. Using Figure 2, we can find the initial and final conditions for the first and last step. For historical reasons, these steps are executed with no double-support phase, that is, $t_d = 0$. The constraints we find from the figure and this assumption for the first step are given by Equations 39 through 42. Similarly, the constraints for the final step are given by Equations 43

Figure 2: The poses A, B and C



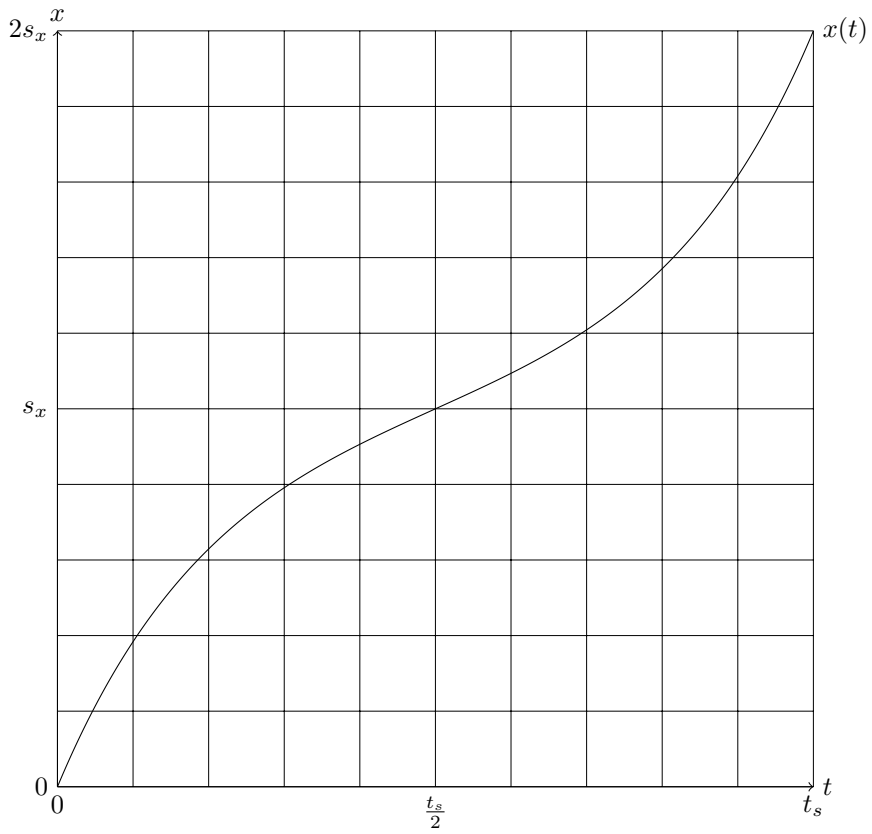
through 46.

$$x_t(0) = s_x \tag{39}$$
$$x_t(t_s) = 2s_x \tag{40}$$
$$x_f(0) = s_x \tag{41}$$
$$x_f(t_s) = 3s_x \tag{42}$$
$$x_t(0) = 0 \tag{43}$$
$$x_t(t_s) = s_x \tag{44}$$
$$x_f(0) = -s_x \tag{45}$$
$$x_f(t_s) = s_x \tag{46}$$

Starting with the torso motion, we notice that the trajectory y_t we derived earlier satisfies the constraints for both initial and final steps. For the trajectory x_t, we consider the trajectory we defined earlier for $t_d = 0$, as shown in Figure 3. We notice that $x_t(\frac{t_s}{2}) = s_x$, and $x_t(t_s) = 2s_x$. Therefore, the function $x_{t,i}(t) = x_t(\frac{t+t_s}{2})$ satisfies the constraints of Equations 39 and 40. Similarly, the function $x_{t,l}(t) = x_t(\frac{t}{2})$ satisfies Equations 43 and 44. Thus, for an initial step, the torso trajectory in the x-direction is given by $x_{t,i}$, and for the last step, it is given by $x_{t,l}$. The total trajectory for the torso can then be generated by combining the entire horizontal and vertical trajectory with the appropriate function defined above.

The foot motion can be generated using the same approach we used for the original trajectory. The solutions thus derived are given in equations 47 and 48, for the initial and final step respectively.

Figure 3: The x-coordinate of the torso in the LIPM, with $t_d = 0$.



$$x_i(t) = -\frac{4s_x}{t_s^3}t^3 + \frac{6s_x}{t_s^2}t^2 + s_x \tag{47}$$

$$x_f(t) = -\frac{4s_x}{t_s^3}t^3 + \frac{6s_x}{t_s^2}t^2 - s_x \tag{48}$$

Though we have now described how to generate reasonable sequences of a finite number of steps, we found that the robot fell over while executing them. In order to avoid this, we introduced an additional parameter y_{off}. To plan the torso motion, we use $s'_y = s_y + y_{off}$ instead of s_y. Intuitively, this is equivalent to planning the motion with respect to a support leg that is slightly further away. Before executing the motion, we apply the translation $y'_t(\tau) = y_t(\tau) + y_{off}$ to derive the coordinates with respect to the usual coordinate frame. The effect of using non-zero y_{off} is that the torso moves further towards the support leg. As we shall see later, this was sufficient to keep the robot from falling.

3 Trajectory Execution

3.1 Introduction

In this section, we describe how the trajectories we have generated earlier can be executed on the Nao. This execution is performed by an open-loop controller. The controller runs every 20 milliseconds. Every time it runs, it queries the Nao to find the current reference joint angles for the Nao's controllers. Having done this, it uses the LIPM to calculate a new reference position in Cartesian space. The controller then applies inverse kinematics to this reference position, using the angles it found earlier as an initial guess. The reference angles of the Nao's controllers are then set to the solution found by inverse kinematics.

In the remainder of this section, we consider a number of steps the controller executes in more detail. First, we will recall a number of preliminaries and consider how to use the LIPM to find reference positions for inverse kinematics. Second, we consider the problem of inverse kinematics and how it can be solved. The subsection on preliminaries was co-written by Daniel Mescheder.

3.2 Preliminaries

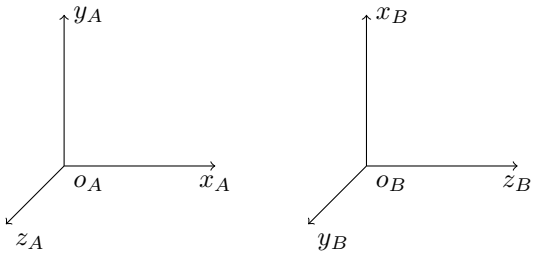
3.2.1 Kinematics

We now recall a number of concepts we will need to perform inverse kinematics. First, we consider coordinate frames and homogeneous transformations between them. Then, we apply these concepts to robotics, using them to derive the forward kinematics. We conclude by considering Jacobians, which are used to find the velocity of a robotic system.

Frames Consider the two coordinate frames A and B shown in Figure 4. 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 $v^B = (o_B v)^B = x_B^B v_x^B + y_B^B v_y^B + z_B^B v_z^B$. [2] Thus, $(o_B v)^A = x_B^A v_x^B + y_B^A v_y^B + z_B^A v_z^B = \begin{bmatrix} x_B^A & y_B^A & z_B^A \end{bmatrix} \begin{bmatrix} v_x^B \\ v_y^B \\ v_z^B \end{bmatrix} = R_B^A v_B$,

[2] Here, 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.

Figure 4: The coordinate frames A (left) and B (right)



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 49, given below. The matrix R_B^A that occurs in this equation is called the rotation matrix from frame B to frame A.

$$v^A = (o_A v)^A = (o_A o_B)^A + (o_B v)^A = o_B^A + R_B^A v^B \tag{49}$$

If we transform v^C to v^B and then to v^A using Equation 49, 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 49 to Equation 50. The matrix T_B^A that appears in this equation is called the (homogeneous) transformation matrix from B to A. The composition of such transformation matrices is equivalent to their matrix product, as indicated in Equation 51.

$$\begin{bmatrix} v^A \\ 1 \end{bmatrix} = \begin{bmatrix} R_B^A & o_B^A \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v^B \\ 1 \end{bmatrix} = T_B^A \begin{bmatrix} v^B \\ 1 \end{bmatrix} \tag{50}$$

$$T_C^A = T_B^A T_C^B \tag{51}$$

Chains, end effectors and forward kinematics We now apply the concepts of frames and transformations between them to robotic systems. For our purposes, a robotic system is a series of joints connected to a static base, as shown in Figure 5. We will refer to such a system 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 this 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.[3] The y-axis is then determined by the additional assumption that each frame should be right-handed, that is[4], that $x \times y = z$. An example of these frames is shown in Figure 6.

Given two frames, we can now define 4 parameters, the Denavit-Hartenberg (DH) parameters, that determine the transformation from one frame to the next. These numbers, a, α, d and θ, have a geometrical interpretation, as shown in Figure 7. The number α is the rotation angle between z_A and z_B, along the axis x_A. The distance between these vectors along x_A is named a. The third parameter, d, gives the distance along z_B between x_A and x_B. The final parameter, θ, is the rotation angle over z_B between x_A and x_B. With these parameters, we can calculate the transformation matrix from the frame B to the frame A, using Equation 52. Of the four DH-parameters, a, α and d are determined by the robot's geometry. The

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

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

Figure 5: A robotic system. The base is indicated by the large semicircle and joints are indicated by circles.

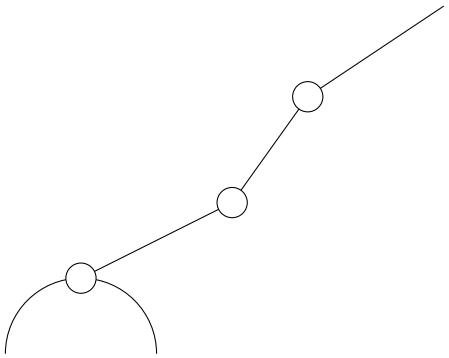


Figure 6: Joints and their associated frames. The direction of rotation of each joint is indicated by an arc.

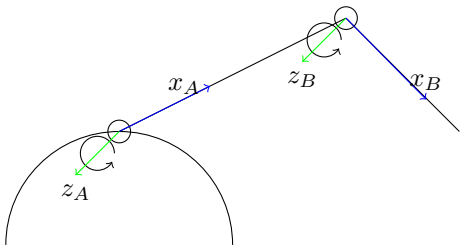
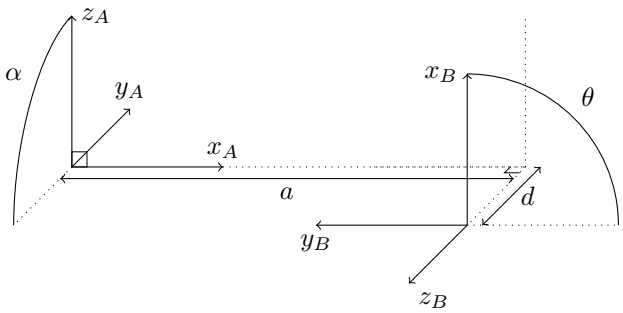


Figure 7: The frames A and B and the DH-parameters of the transformation between them



remaining parameter, θ, is a variable giving the position of a joint. Thus, the transformation matrix as defined by Equation 52 can be viewed as a function from the joint's angle, that is, its parameter θ, to its transformation matrix.

$$T_B^A(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & a \\ \sin\theta\cos\alpha & \cos\theta\cos\alpha & -\sin\alpha & -d\sin\alpha \\ \sin\theta\sin\alpha & \cos\theta\sin\alpha & \cos\alpha & d\cos\alpha \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{52}$$

Usually, we are not interested in the positions of indidivual joints in the chain, but rather in the final joint of the chain. 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 51, we can define the forward kinematics for an n-joint chain as the matrix F defined in equation 53. Notice that F is given as a function of q, the vector of joint angles of the chain.

$$F(q) = T_n^0 = \prod_{i=1}^{n} T_i^{i-1}(q_i) \tag{53}$$

Though the forward kinematics completely specify the position and orientation of a chain, they are somewhat redundant. In particular, the rotation matrix uses 9 parameters to encode orientation, while smaller representations using 3 or 4 parameters exist. An additional problem with the rotation matrix is that its time derivative is not very convenient to work with. To solve both these problems, we will represent orientations by quaternions, which require 4 parameters. For a rotation about an axis v over an angle ϕ, the corresponding quaternion $\begin{bmatrix}\mathbf{r} & r_0\end{bmatrix}^T$ [5] is given below in Equation 54.

$$\begin{bmatrix} \mathbf{r} \\ r0 \end{bmatrix} = \begin{bmatrix} v\sin(\frac{\phi}{2}) \\ \cos(\frac{\phi}{2}) \end{bmatrix} \tag{54}$$

Though Equation 54 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. This calculation is significantly more complex and is deferred to an appendix. 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 50. 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, as described in Appendix B. We can then define $p(q)$ as the vector consisting of $o(q)$ and $r(q)$, as given by Equation 55.

$$p(q) = \begin{bmatrix} o(q) \\ r(q) \end{bmatrix} \tag{55}$$

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

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.[6] 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 56.[7] The matrix $J(q)$ that appears in this equation is called the Jacobian of the system.

$$\begin{bmatrix} v \\ \omega \end{bmatrix} = J(q)\dot{q} \tag{56}$$

To compute the Jacobian, we shall use its explicit form, as discussed in the Stanford introductory course in robotics [15, 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 57[8] , 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.

$$J_i = \begin{bmatrix} z_i^0 \times p_i^0 \\ z_i^0 \end{bmatrix} \tag{57}$$

We can now compute the Jacobian of a robotic system and therefore its linear and angular velocity. This is not yet sufficient for inverse kinematics, as 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 58, where $\dot{r}$ is the time derivative of the quaternion corresponding to the end-effector's orientation.

$$\begin{bmatrix} v \\ \dot{r} \end{bmatrix} = J_r(q)\dot{q} \tag{58}$$

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. The formula for this matrix will be given later. First, we define the cross product matrix (cpm) operator, which is used in this formula. 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 59[9] .

$$\text{cpm}(v) = \begin{bmatrix} 0 & -v_z & v_y \\ v_z & 0 & -v_x \\ -v_y & v_x & 0 \end{bmatrix} \tag{59}$$

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 60.[10] 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 $J(q)$ by means of a linear transformation, as given in Equation 61.

[6] 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.

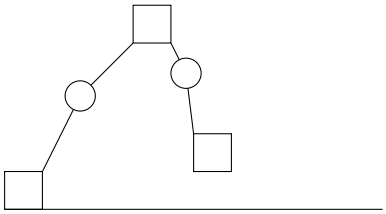
[7] v denotes the linear velocity of the end-effector. The angular velocity is denoted by ω.

[8] 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.

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

[10] I_3 denotes the 3×3 identity matrix.

Figure 8: The nao's lower body



$$H(r) = \frac{1}{2}\begin{bmatrix} r_0 I_3 - \text{cpm}(\mathbf{r}) \\ \mathbf{r}^T \end{bmatrix} \tag{60}$$

$$J_r(q) = \begin{bmatrix} I_3 & 0 \\ 0 & H(r(q)) \end{bmatrix} J(q) \tag{61}$$

3.2.2 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 8. 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 8 to Figure 5, 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, as shown below in Equation 62.[11]

$$p(q) = \begin{bmatrix} p_T(q) \\ p_F(q) \end{bmatrix} \tag{62}$$

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 62, $J_r(q)$ has a similar structure, shown in Equation 63. 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

[11] The subscript T refers to the torso, the subscript F to the moving foot.

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 57.

$$J_r(q) = \begin{bmatrix} J_{T,r}(q) \\ J_{F,r}(q) \end{bmatrix} \tag{63}$$

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 above 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 and the calculation procedure we described 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 the base- and end-transforms of each chain.

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 legs of the Nao: 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 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.

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

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

$$F_{be}(q) = T_0^b \cdot F(q) \cdot T_e^n \tag{64}$$

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):

$$J_b = \begin{bmatrix} R_0^b & 0 \\ 0 & R_0^b \end{bmatrix} \cdot J \tag{65}$$

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 59:[12]

$$\begin{bmatrix} v_e \\ \omega_e \end{bmatrix} = \begin{bmatrix} I_3 & -\operatorname{cpm}(p_e^n) \\ 0 & I_3 \end{bmatrix} \begin{bmatrix} v_n \\ \omega_n \end{bmatrix} \tag{66}$$

Equation 66 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.

$$\begin{bmatrix} v_e \\ \omega_e \end{bmatrix} = \begin{bmatrix} I_3 & -R_n^b \operatorname{cpm}(p_e^n) \left(R_n^b\right)^T \\ 0 & I_3 \end{bmatrix} \begin{bmatrix} v_n \\ \omega_n \end{bmatrix} \tag{67}$$

Premultiplying the Jacobian by the transformations shown in Equations 65 and 67 yields a new Jacobian matrix which takes into account both base- and end transforms. Also taking into account the quaternion transform shown in Equation 60 leads to Equation 68 below.

$$J_{be}(q) = \begin{bmatrix} I_3 & 0 \\ 0 & H(r(q)) \end{bmatrix} \begin{bmatrix} I_3 & -R_n^b \operatorname{cpm}(p_e^n) \left(R_n^b\right)^T \\ 0 & I_3 \end{bmatrix} \begin{bmatrix} R_0^b & 0 \\ 0 & R_0^b \end{bmatrix} J(q) \tag{68}$$

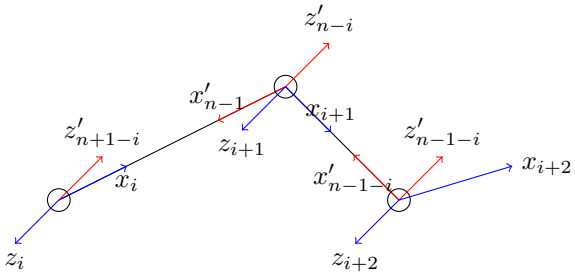
The second way to approach this problem is to add static joints at the beginning and end of the chain. Unlike regular joints, which are defined by their DH-parameters, static joints are defined by given transformation matrices. Thus, the transformation matrix of such a joint is a constant. With these 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 53, 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 57 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

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

Figure 9: Original and reversed frames for one chain of joints.



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 9:

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

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 69 and 70 let $2 \leq i \leq n$ and for Equations 71 and 72 let $1 \leq i \leq n$.

$$a'_i = a_{(n+2-i)} \tag{69}$$
$$\alpha'_i = \alpha_{(n+2-i)} \tag{70}$$
$$\theta'_i = \theta_{(n+1-i)} \tag{71}$$
$$d'_i = d_{(n+1-i)} \tag{72}$$

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. The resulting reverse base- and end-transforms are given by Equations 74 and 73 respectively. In these equations, $R_z(\phi)$ and $R_x(\phi)$ denote a rotation of ϕ radians around the z and x axes respectively.

$$\left(T_e^n\right)' = R_z\left(\pi\right) R_x\left(\pi\right) \left(T_0^b\right)^{-1} \tag{73}$$
$$\left(T_0^b\right)' = \left(T_e^n\right)^{-1} R_z\left(\pi\right) R_x\left(\pi\right) \tag{74}$$

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. Let furthermore T_0^b be the old base-transform. Then

an equivalent formulation can be given by setting $a_1^{new} = \alpha_1^{new} = 0$ and using the matrix $(T_0^b)^{new}$ derived below as a new base-transform:

$$\left(T_0^b\right)^{new} = T_0^b T_x\left(\alpha_1\right) R_x\left(a_1\right) \tag{75}$$

Combined Chain for Nao-Legs The LIPM requires the control of the moving foot from the support foot frame. 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 3.2.2 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. Here, HipOffsetY is a constant defined in the documentation [1].

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

The chain from the right foot to the left foot can be calculated using the technique derived in section 3.2.2.

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 an 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.

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

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:

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

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 52, 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 53 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.

3.2.3 Using the LIPM

To complete our preliminaries, we will now consider how to specify target positions for inverse kinematics. In general, inverse kinematics attempts to solve the problem of finding joint angles q such that $p(q)$ takes on some desired value p_d. Applying this to the Nao, we will need to find a position p_d of the form given by Equation 62, using the LIPM. In this equation, we see that the position consists of two sub-positions, one for the torso and one for the foot. Each of these consists of a linear portion and a portion specifying orientation. We will treat these linear and angular portions separately.

First, we will find the linear positions we require. In principle, these positions are the functions specified by the LIPM. The only problem is that the positions obtained from the LIPM are given relative to a coordinate system where the torso's original position is at the origin. Since we wish to work relative to the stationary support foot, we will need to add a translation to these positions. This translation depends on the type of step we are executing. For initial steps, we find that the torso and support foot start at the same x-coordinate, but the torso starts $-s_y$ to the left of the support leg. In normal and final steps, the torso additionally starts s_x behind the support leg. From these considerations, we find that the translations δ_x and δ_y are given by Equations 76 and 77. Our linear positions for inverse kinematics are then found by applying the transformation given by Equations 78 and 79 to both the foot and torso positions. In this transform, x' is the position we require, while x is the position given by the LIPM, and similarly for y.

$$\delta_x = \begin{cases} 0 & \text{for initial steps} \\ -s_x & \text{otherwise} \end{cases} \tag{76}$$

$$\delta_y = -s_y \tag{77}$$

$$x'(t) = x(t) + \delta_x \tag{78}$$

$$y'(t) = y(t) + \delta_y \tag{79}$$

Second, we now consider the orientation the torso and foot should have. Here, the LIPM does not give us any information, and so we must use other considerations.

For simplicity, we will specify a constant orientation that should help us avoid certain causes of instability. For example, we would like to keep the torso from tipping forward, since that could cause the Nao to fall. Similarly, we wish to keep the foot from hitting the ground at an angle, since this could also lead to falls. Both of these problems can be avoided by keeping the torso vertical while holding the foot horizontal. To implement this, we must specify the quaternions corresponding to these orientations. Fortunately, the coordinate frames used on the Nao are such that the orientations we require are the default orientations of both the foot and the torso. Therefore, the quaternion we need corresponds to a rotation of 0 degrees, and so is given by Equation 80.

$$r_d = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{80}$$

To summarize, if $(x_T(t), y_T(t), z_T(t))$ and $(x_F(t), y_F(t), z_F(t))$ are the positions given by the LIPM for the torso and foot respectively at time t, the corresponding reference position for inverse kinematics is given by Equation 81. Here, $x'_T(t)$ is $x_T(t)$ transformed using Equation 78, and $y'_T(t)$, $x'_F(t)$ and $y'_F(t)$ are defined similarly.

$$p_d(t) = \begin{bmatrix} x'_T(t) \\ y'_T(t) \\ z_T(t) \\ r_d \\ x'_F(t) \\ y'_F(t) \\ z_F(t) \\ r_d \end{bmatrix} \tag{81}$$

3.3 Inverse Kinematics

We will now consider the problem of inverse kinematics and how it can be solved. In this problem, we are given a reference position p_d such as we have determined in the previous section. We are asked to find joint angles q such that $p(q) = p_d$, where $p(q)$ is the position of the Nao's lower body for the angles q. We will consider three methods that solve this problem, each of which is based on the Jacobian.

3.3.1 Newton-Raphson

Newton-Raphson is an iterative process. It starts from some initial guess q. If this q is a solution of $p(q) = p_d$, the process is complete. If it is not a solution, Newton-Raphson calculates a step Δq to a new guess q'. The process then continues with q' as our new initial guess. Since this process is not guaranteed to converge to a solution, we require it to terminate after a set number of steps. If it terminates in this way, we do not have a solution to our problem. A complete listing summarizing the algorithm in pseudocode is given in algorithm 1.

It is instructive to look more closely at the update equation used by Newton-Raphson, $\Delta q_k = J^+(p_d - p(q_k))$. In this equation, p_d is the position we are attempting to reach, J is the Jacobian of the system and J^+ is its Moore-Penrose pseudoinverse. We will now derive this equation by considering a linear approximation to $p(q)$. Such an approximation around q_k is given by $p(q_k+\Delta q) = p(q_k)+J\Delta q$. Setting this approximation equal to p_d, we find $p_d = p(q_k) + J\Delta q$. From this equation, we can find Δq by solving $J\Delta q = p_d - p(q_k)$. Since J is in general not square

Algorithm 1 Newton-Raphson

Let q0 be an initial guess, k = 0
loop
  if $|p(q_k) - p_d| < \epsilon$ **then**
    Terminate with solution q_k
  else if $k \geq n_{max}$ **then**
    Terminate without a solution
  end if
  Calculate the Jacobian $J(q_k)$
  Calculate $\Delta q_k = J^+(p_d - p(q_k))$
  Set $q_{k+1} = q_k + \Delta q_k$
  Set $k = k + 1$
end loop

or non-invertible, we use the Moore-Penrose pseudoinverse, arriving at the update equation mentioned earlier.

The primary drawback of Newton-Raphson follows directly from the update equation we have just derived. This drawback is that, if J has singular values close to zero, the pseudo-inverse will have very large entries. The Δq calculated from the update equation will then be very large. In practice, this tends to lead to divergence. Our next method, Levenberg-Marquardt, can deal with these singularities. Our discussion of it will also provide another perspective on Newton-Raphson.

3.3.2 Levenberg-Marquardt

Like Newton-Raphson, it is an iterative process that begins with an initial guess q. Levenberg-Marquardt also starts by checking if q is already a solution. If it is not, the algorithm first calculates the gradient of the error function $E(q) = |p(q) - p_d|^2$. If the gradient of the error is very small, the algorithm has converged and terminates without a solution. In this case, q is a local minimum of the error function. If the gradient is large enough, Levenberg-Marquardt calculates a step value Δq. Unlike Newton-Raphson, Levenberg-Marquardt verifies that $q' = q + \Delta q$ results in a smaller error value before continuing with this new initial guess. If the guess indeed results in a smaller error, we decrease our parameter μ by dividing it by the parameter $\theta > 1$. If the guess does not result in a smaller error, we continue with $q' = q$ and multiply μ by θ. Since this process is not guaranteed to converge, we require it to terminate after a fixed number of iterations, as we did for Newton-Raphson. The process is summarized in Algorithm 2. [13]

As with Newton-Raphson, we will derive Levenberg-Marquardt's update equation. To do so, we will apply Newton's method to minimize the error function $E(q) = |p(q) - p_d|^2$. In other words, we minimize its second-order Taylor polynomial around q, which is given by $E(q + \Delta q) = E(q) + \nabla_q E(q)^T \Delta q + \frac{1}{2} \Delta q^T H \Delta q$, where H is the Hessian of E evaluated at q. The gradient of this expression with respect to Δq is $H\Delta q + \nabla_q E(q)$. Setting this gradient to zero in order to find an optimum, we find $\Delta q = -H^{-1} \nabla_q E(q)$. Next, we require expressions for the gradient and Hessian of our error function $E(q)$. One can verify that the gradient is given by $\nabla_q E(q) = 2J^T(p(q) - p_d)$, where J is the Jacobian of $p(q)$. The Hessian is given by $H = 2(J^T J + S)$, where S is a linear combination of Hessians of the elements of $p(q) - p_d$. Assuming $S \approx 0$, we find $H = 2J^T J$. Inserting this in the equation we found earlier for Δ_q gives $\Delta_q = -(J^T J)^{-1} J^T (p(q) - p_d)$. One problem with this equation is that if the matrix J does not have full column rank, or equivalently, it

[13] In this algorithm and the remainder of this discussion, J^T denotes the matrix transpose of J.

Algorithm 2 Levenberg-Marquardt

Let q0 be an initial guess, k = 0
loop
  if $|p(q_k) - p_d| < \epsilon$ **then**
    Terminate with solution q_k
  else if $k \geq n_{max}$ **then**
    Terminate without a solution
  end if
  Calculate the Jacobian $J(q_k)$
  Calculate the gradient $g = 2J^T v$
  if $|g| < \epsilon_g$ **then**
    Terminate with the local minimum q_k
  end if
  Calculate $\Delta q_k = -(J^T J + \mu I)J^T v$
  if $|p(q_k + \Delta q_k) - p_d| \geq |p(q_k) - p_d|$ **then**
    Set $q_{k+1} = q_k$
    Set $\mu = \mu \times \theta$
  else
    Set $q_{k+1} = q_k + \Delta q_k$
    Set $\mu = \frac{\mu}{\theta}$
  end if
  Set k = k + 1
end loop

has some singular values equal to zero, $J^T J$ is singular. To avoid this singularity, we introduce a parameter μ, and replace $J^T J$ by $J^T J + \mu I$.

Next, let us consider the parameter μ. If $\mu = 0$, we can justify Levenberg-Marquardt as an approximation of Newton's method. Since $J^+ = (J^T J)^{-1} J^T$ if and only if $J^T J$ is invertible, one can also notice that Levenberg Marquardt is equivalent to Newton-Raphson in this case. For very large values of μ, we find that $\Delta q \approx -\frac{1}{\mu} J^T (p(q) - p_d)$. Since the gradient of $E(q)$ is given by $2J^T(p(q) - p_d)$, this is a form of gradient descent. Since μ was assumed to be very large, the step size for gradient descent would be very small. In between these two extremes, Levenberg-Marquardt can be understood as a trade-off between minimizing error and minimizing the length of Δq, as described by Buss [3]. Note that Buss refers to this algorithm as "Damped least squares".

We can now understand the adaptive calculation of μ used by this version of Levenberg-Marquardt. If an iteration does not reduce the error value, Levenberg-Marquardt increases μ, putting more emphasis on making small steps and moving towards gradient descent. Since gradient descent should be able to reduce the error for small enough step sizes and reasonably well-behaved functions, the algorithm will then eventually find a better guess. If an iteration reduces error directly, μ is decreased, allowing the algorithm to make larger steps. Thus, if a series of iterations continues to reduce error with every step, Levenberg-Marquardt can move quickly towards a solution.

We have now considered Levenberg-Marquardt and Newton-Raphson, two methods strongly related to Newton's method. Our final method will be gradient descent, using the error function we used for Levenberg-Marquardt.

3.3.3 Gradient descent

We conclude our survey of methods with a variant of gradient descent using a constant step size and smoothing using a momentum parameter. Like our previous

methods, this method iteratively improves an initial guess q. As in the other algorithms, if q is already a solution, we are done. If q is not a solution, we calculate a new step and continue from there. If the step we make is too small, we terminate the algorithm with a local minimum. Since this method is also not guaranteed to converge, we once again terminate it after a given number of iterations. The process is summarized in algorithm 3.

Algorithm 3 Gradient descent

Let q0 be an initial guess, k = 0
loop
  if $|p(q_k) - p_d| < \epsilon$ **then**
    Terminate with solution q_k
  else if $k \geq n_{max}$ **then**
    Terminate without a solution
  end if
  if $k = 0$ **then**
    Set $\Delta q_0 = -2\alpha J^T(p(q) - p_d)$
  else
    Set $\Delta q_k = -2\alpha(1-\gamma)J^T(p(q) - p_d) + \gamma\Delta q_{k-1}$
  end if
  if $|\Delta q_k| < \epsilon$ **then**
    Terminate with q_k as a local minimum
  end if
  Set $q_{k+1} = q_k + \Delta q_k$
  Set $k = k + 1$
end loop

To calculate a new step, gradient descent moves along the negative of the gradient of the error function $E(q) = |p(q) - p_d|^2$. As discussed in our derivation of Levenberg-Marquardt, this gradient is given by $g = 2J^T(p(q) - p_d)$. We can then take a step of size α along the negative of this gradient, giving $\Delta q_n = -2\alpha J^T(p(q) - p_d)$. To damp out oscillations, we apply a smoothing filter to this sequence, using the recurrence $\Delta q_{k+1} = (1-\gamma)\Delta q_n + \gamma\Delta q_k$. The recurrence is initialized using $\Delta q_0 = \Delta q_n$. Combining these equations leads to the update function given in algorithm 3.

The primary drawback of gradient descent is that it takes a large number of iterations to converge. In our tests, this variant of the algorithm took too long to converge for practical use, requiring far more iterations and far more computation time than Levenberg-Marquardt or Newton-Raphson. Other variants of this basic scheme may work better, such as the one described by Buss [3]. The scheme proposed by Buss is similar to the approximate minimization performed by Levenberg-Marquardt. Hence, we predict that Levenberg-Marquardt will still converge faster than gradient descent using this minimization scheme.

4 Stability Measurement and Controller Performance

4.1 Introduction

In this section, we consider the two kinds of experiments we performed. The first kind attempts to quantify the stability of the Nao while walking for various settings of the LIPM's parameters. The second kind measures how well our controller is satisfying its real-time requirements and tracking the trajectory the LIPM generates.

As mentioned, our first kind of experiment is concerned with quantifying the stability of the Nao. A number of measures of stability have been proposed. Generally, stability is measured by considering the position of a certain point, such as the zero-moment point or center of pressure, relative to the convex hull of the robot's feet. Information on many of these points can be found in references [13, 5, 4, 16, 14].

For our work, we require a point that is easy to measure on the Nao. Since the Nao does not provide accelerations, we cannot calculate any points whose formulae require the acceleration or angular acceleration of the robot. This requirement excludes most of the calculation procedures given by Poskriakov [13]. The Nao does, however, have force sensors on its feet. Using a formula given by Goswami [5], we can use these sensors to calculate the center of pressure. The center of pressure (CoP) is the point where the resultant force of the ground's pressure field normal to the ground acts[5]. We measure stability by considering the position of the CoP relative to the convex hull of the robot's feet. If the CoP is within this convex hull, the area of support (AoS), the robot is stable. If the CoP is on the edge of the AoS, the robot may be falling over. In both cases, the distance from the center of pressure to the nearest edge of the AoS indicates how stable the robot is.

We begin the remainder of this section by considering how we can measure the CoP and AoS on the Nao. Thereafter, we consider how a number of decisions and approximations we make in the course of implementing this measurement affect the results. Then, we apply this measurement to the walk we implemented, allowing us to look at how the parameters of the LIPM influence the stability of the resulting motion. We then conclude this section by taking a look at the controller's performance.

4.2 The measurement procedure

We now consider how we can measure the CoP, AoS and the distance from the CoP to the nearest edge of the AoS in practice. To do so, we will first consider how we can find the CoP and AoS separately. The resulting values can then be combined to find the distance we require, which will tell us how stable the robot is.

To measure the CoP, we can use a formula given by Goswami [5]. As he indicates, the CoP can be found as a weighted sum over the pressure forces exerted by the ground on the robot's feet. For the purposes of our measurement, we assume that these pressure forces are exerted at the locations of the force sensors in the Nao's feet. Given the location x_i of sensor i and the force F_i exerted there, we can find the CoP using equation 82. The positions x_i must all be given in the same reference frame. By taking each position relative to the support foot, we ensure that the positions of the force sensors in this foot are constant. The positions of the sensors in the moving foot are known relative to the moving foot. Hence, we use the forward kinematics of this foot relative to the support foot to transform these positions. The procedure we have just described is summarized in algorithm 4.

$$x_{CoP} = \frac{\sum_i x_i F_i}{\sum_i F_i} \tag{82}$$

Algorithm 4 Measuring the CoP

Measure the force F_i at each force sensor
Find the forward kinematics of the moving foot, T
Find x_i for each sensor, transforming using T if the sensor is in the moving foot
Use equation 82 to compute the position x_{CoP} of the CoP

Next, we will find the AoS. The AoS is formally defined as the convex hull of the

robot's feet. The Nao's feet have a non-polygonal shape that is somewhat difficult to use directly. Instead, we approximate the convex hull of the actual feet by the convex hull of the force sensor locations. Since the force sensors are located close to the edge of the feet, this should be a reasonable approximation. This hull should be taken over only those force sensors that are in contact with the ground. We consider a force sensor in the moving foot to be on the ground if its vertical coordinate after transformation is below a treshold z_ϵ. The force sensors in the support foot are assumed to always be on the ground. Once the sensors that are on the ground have been found, we can apply a standard 2D convex hull algorithm, such as the Graham Scan (see, for instance [12]). This procedure is summarized in algorithm 5.

Algorithm 5 Measuring the AoS

Let X_g contain the positions of the force sensors in the support foot
For each force sensor i in the moving foot, if its transformed position x_i' has a z-coordinate below z_ϵ, add x_i' to X_g
Use the Graham Scan to compute the AoS as the convex hull of the points in X_g.

Once we have applied algorithms 4 and 5 to find both the CoP and the AoS, we can find the distance from the CoP to the nearest edge of the AoS. This distance, which we call the *stability margin*, indicates how stable the Nao is. The margin can be found using straightforward geometry, and so the calculation will not be discussed in more detail here. In the following subsection, we consider a number of potential issues with the measurement we discussed above.

4.3 Analyzing the measurement procedure

In this section, we consider the accuracy of the measurement of the previous section. We consider the following three topics:

1. The measurement procedure we use assumes that a force sensor that is not touching the ground will register zero force. In particular, if the moving foot is moving through the air, all of its force sensors should give a value of zero.

2. To determine the AoS, we decided to use the position of a sensor to tell whether or not it is on the ground. In principle, we could have also used the sensor's value.

3. As mentioned above, we use the position of a sensor to determine if it is on the ground. This position can be calculated either from reference angles, i.e. "Where the robot should be", or sensor angles, i.e. "Where the robot is".

Our first topic is the assumption that a sensor that is not in contact with the ground will measure zero force. In practice, this assumption does not appear to hold, as all of the Nao's sensors give a small non-zero force when moving through the air. In theory, the procedure of algorithm 4 would take these sensors into account with non-zero weight. This would lead to an error in the CoP we calculate, moving it towards the moving foot. This error should affect the stability margin as well. This effect could either reduce or increase the margin, depending on where the real center of pressure is located. We could avoid this error by explicitly testing if a force sensor is on the ground or not. If it is not on the ground, we assign it a weight of zero. Figure 10 shows the stability margin in a standardised walking situation, with and without the change to the measurement we have just described. Overall, the changed results indicate higher stability. These results are likely to be closer to the truth than the original results, provided we can accurately determine whether or not the force sensors are in contact with the ground.

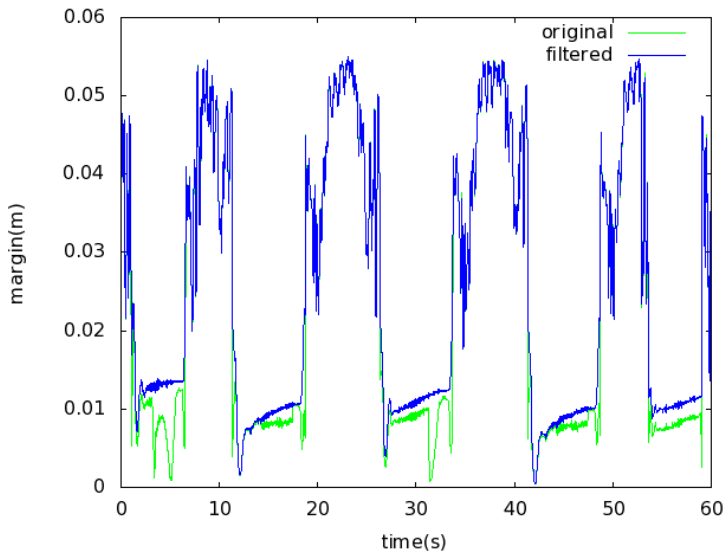


Figure 10: Stability margin versus time, filtered and unfiltered

Our second topic is our decision to test whether a sensor is on the ground or not by checking its position. An alternative would be to check the force measured by a sensor. However, as we have just remarked, the Nao's sensors give non-zero force even when not on the ground, making such a procedure impractical. Given that we must use the position of a sensor, particularly its vertical position, to determine if it is on the ground, one issue remains. This issue is the selection of the treshold that separates being "on the ground" from being "in the air". In our work, we selected a treshold value of 1 centimeter, which appears intuitively to be rather high. As an alternative, we considered the value of 1 millimeter and compared the results. This comparison is shown in figure 11. One can see a large spike at a time of 8 seconds in the results for 1 millimeter. This spike indicates a value of the stability margin that is unusually large, making it rather implausible. One explanation for this phenomenon is that the measured CoP is in fact far out of the measured AoS, making this result erroneous. Given this potential error, we consider the treshold of 1 centimeter to be the more reasonable choice.

In order to use the position of a force sensor to determine whether or not it is on the ground, this position needs to be calculated. To do so, we calculate the forward kinematics. The forward kinematics could be calculated using either the actual joint angles of the Nao, as measured by the angle sensors, or the reference joint angles computed by our controller. In theory, the reference angles are unaffected by noise, while the sensor values could have significant measurement error. On the other hand, the reference angles could be relatively far removed from the actual joint angles if the overall tracking system has significant error. In figure 12 we have graphed the stability margin while using the reference angles and while using the sensors. Both measurements were performed during the same walking sequence. We do not notice any "obvious" measurement artifacts in these results. As such, we cannot tell from these results alone whether or not we should use the angle sensors. Somewhat arbitrarily, we will use the reference angles for all the experiments we describe in the remainder of this paper. We will also use a treshold for vertical positions of 1 centimeter and filter out any force sensors known to not be on the ground.

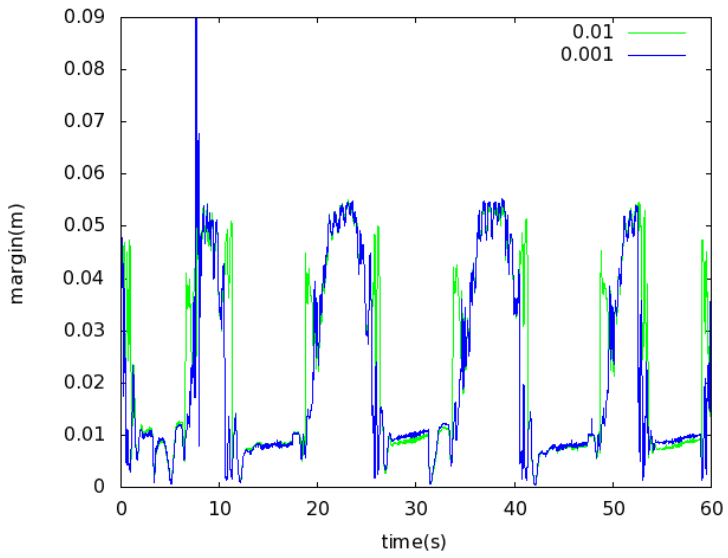


Figure 11: Stability margin versus time, tresholds 1 and 0.1 cm

4.4 Analyzing parameter settings

We will now consider how the various parameters of the LIPM influence the stability of the resulting motion. We will begin these considerations by examining the ratio between t_s and t_d, that is, the time spent in the single- and double-support phases of the walk. To do so, we keep the total time taken for each step constant, but vary how it is divided over the two phases. The resulting stability margin is shown in Figure 13.

With only a single-support phase (the line for 0 in the figure), we see that the stability margin is low at the end of the first step and at the start of the final step. In our tests, the Nao usually fell over at the start of the final step. With a relatively short double-support phase, we see that the walk is generally stable. A longer double-support phase also leads to similar stable results. We conclude that the LIPM requires a double-support phase in order to be stable, but does not seem to depend on the length of this phase.

The next aspect of our parameters we will examine is the total time taken per step. In these experiments, we will keep $\frac{t_d}{t_s}$ equal to 0.5, which appeared to be stable in our previous tests. The results for various total times are shown in Figure 14. From this figure, it appears that the differences between the results are not very large, especially between the settings of 11.25 seconds and 15 seconds. The exceptions to this are the first and last step, where the setting of 9 seconds differs significantly from the others. At the start of the last step, this setting seems to lead to instability. Thus, it appears that 9 seconds may be slightly too low, but the other two settings seem to lead to generally stable walks.

We continue by checking how the parameter s_x influences stability. For various settings of this parameter, we have plotted the stability margin versus time in Figure 15. We see here that the setting of 4.5 centimeters appears to lead to the lowest stability margin. Furthermore, we see that during a number of time intervals, the setting of 3.5 centimeters leads to greater stability than the setting of 2.5 centimeters. However, the opposite happens in another time interval. Still, it appears that overall, 3.5 centimeters is the most stable setting we have examined here.

Next, we consider the parameter s_y. We have chosen a number of settings for

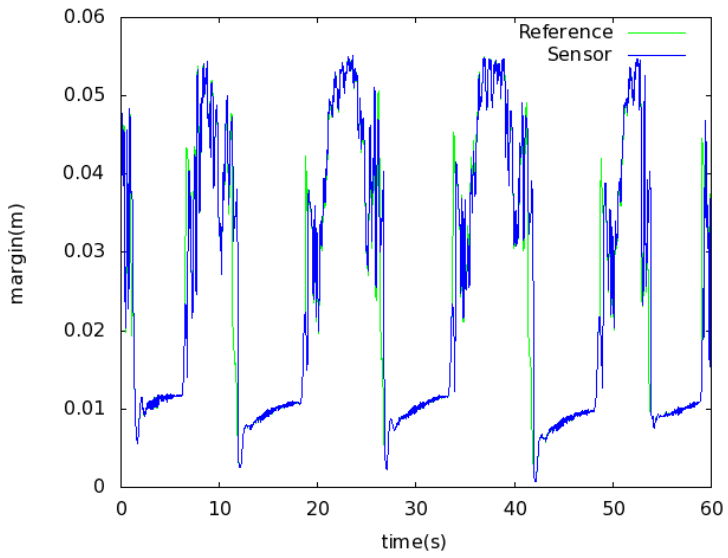


Figure 12: Stability margin versus time, reference and sensor angles

this parameter and have plotted the resulting stability margin versus time in Figure 16. For this parameter, we see a clear difference between the three settings. The least stable setting appears to be 6 centimeters, followed by 5.5 centimeters and then 5 centimeters.

Our next parameter is y_{off}. For this parameter, we have attempted experiments with values lower than 14 millimeters. Generally, the Nao would consistently fall over during almost every step in these experiments. For this reason, we consider offsets of 14, 15 and 16 millimeters, which do not lead to falls. The results of these experiments are shown in Figure 17. Like with s_y, we see clear differences between settings. The least stable setting in this case is 14 millimeters, followed by 15 millimeters and then 16 millimeters.

Our second-to-last parameter is z_c, the height of the torso. In this case, settings larger than 31 centimeters lead to trajectories for which our inverse kinematics could not find the corresponding joint angles. Whether this is due to the Nao's inability to reach these positions or our inverse kinematics is unknown. Values lower than 29 centimeters did, on the other hand, lead to trajectories our inverse kinematics could find joint angles for. However, the resulting trajectories were rather difficult for the Nao to execute, commonly leading to falls. Hence, we consider only values between 29 and 31 centimeters. The results for these values are shown in Figure 18. In this case, it appears that the setting of 31 centimeters leads to a number of unstable moments, with stability margins that are nearly zero. The other two settings lead to very similar and generally stable results.

Our final parameter is z_m, the maximum height of the moving foot. For this parameter, we show the results for settings between 5 and 7.5 centimeters in Figure 19. In this figure, we see only small differences between the different settings. We also see a few places where the stability margin temporarily becomes very low. In these places, the setting of 7.5 centimeters leads to instability. Whether the other settings also have this problem is difficult to see from the figure.

Figure 13: Stability margin versus time, for a number of ratios between t_d and t_s

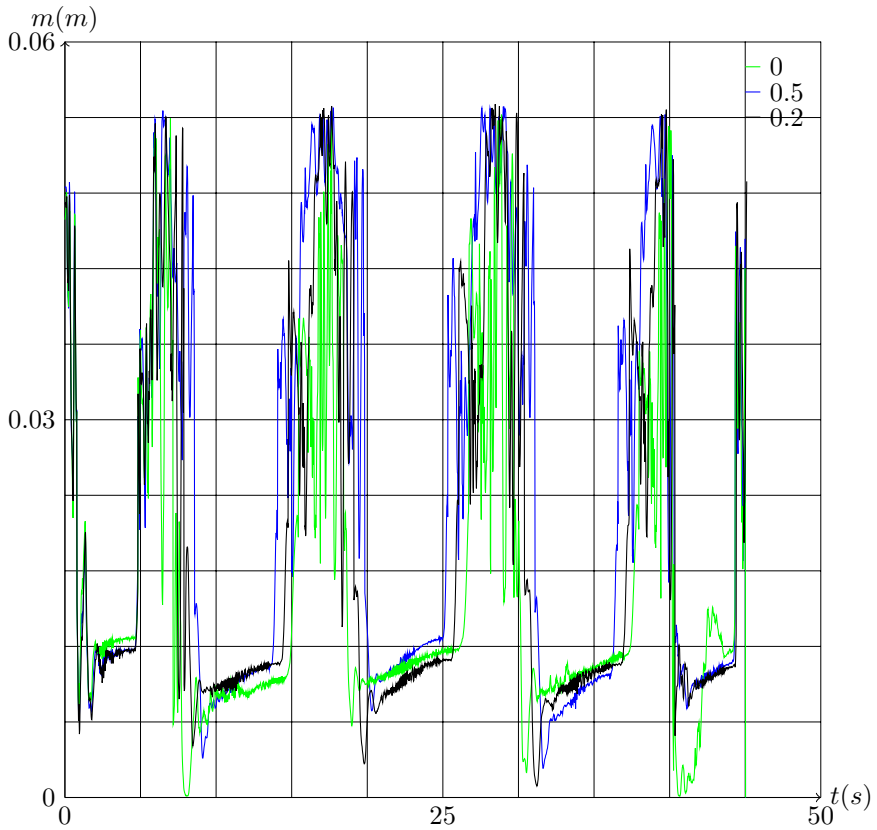


Figure 14: Stability margin versus normalized time, for a number of total times per step.

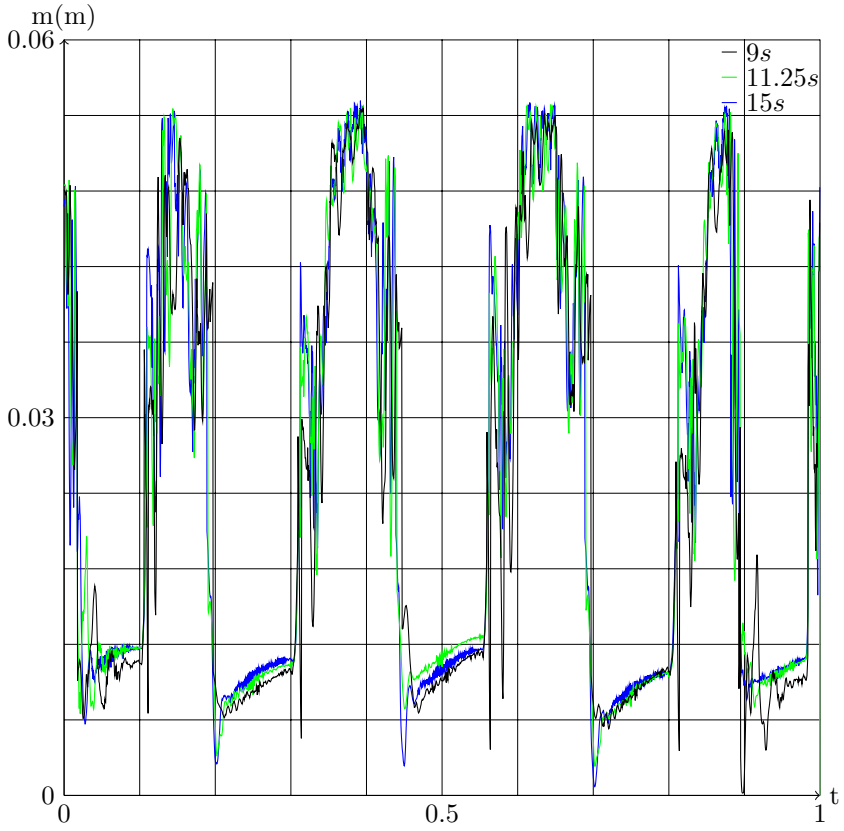


Figure 15: Stability margin versus time for various settings of s_x

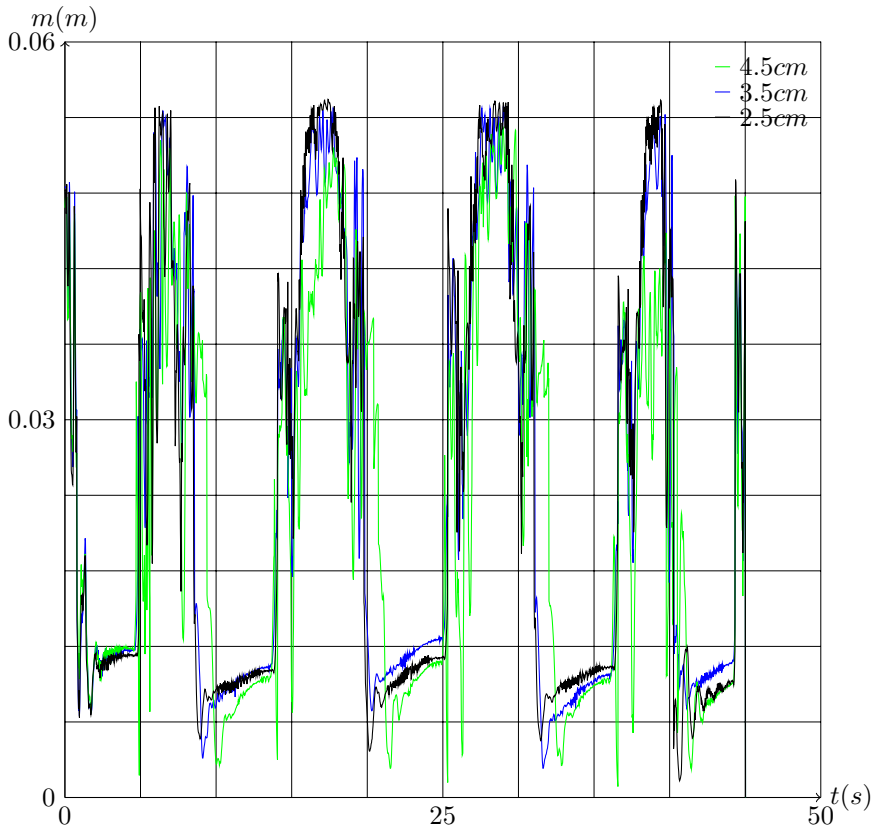


Figure 16: Stability margin versus time for various settings of s_y

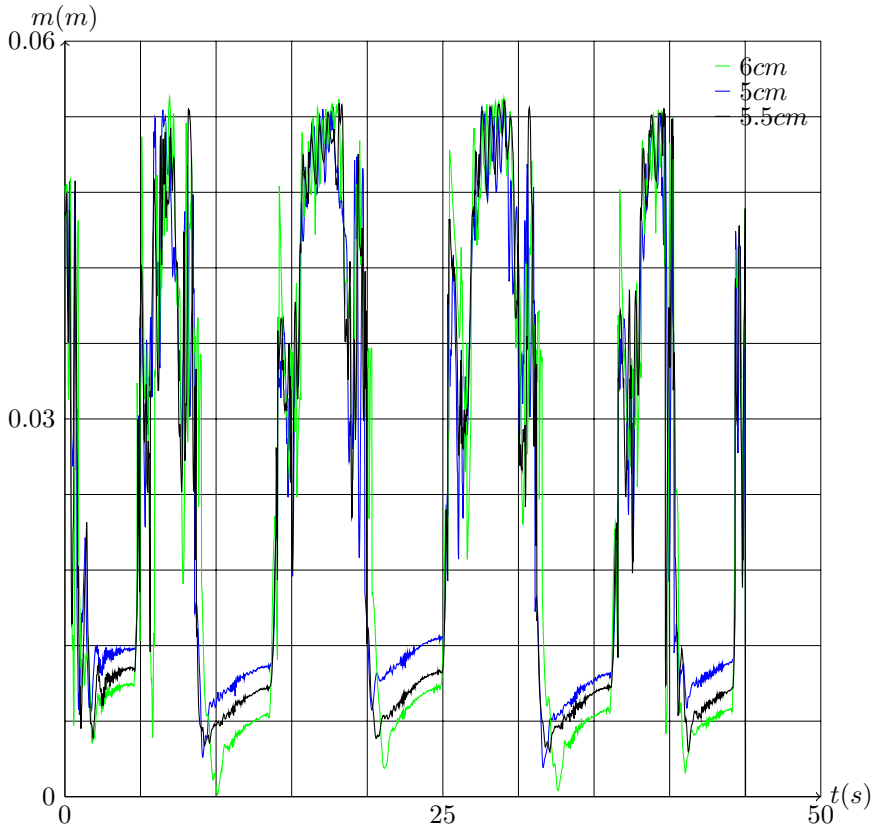


Figure 17: Stability margin versus time for various settings of y_{off}

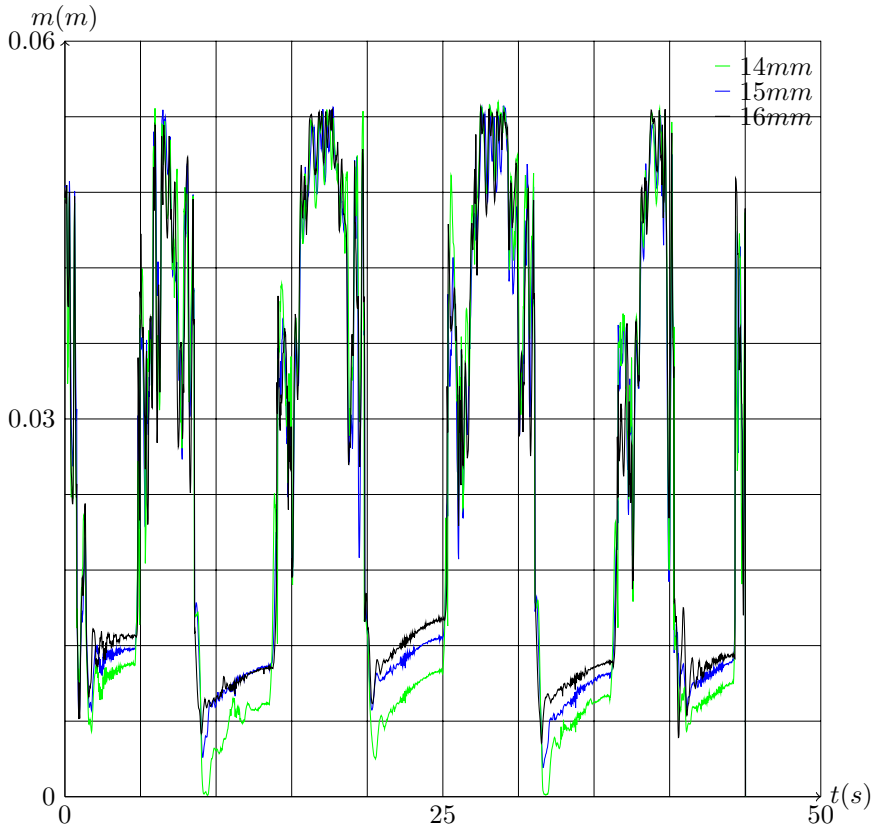


Figure 18: Stability margin versus time for various settings of z_c

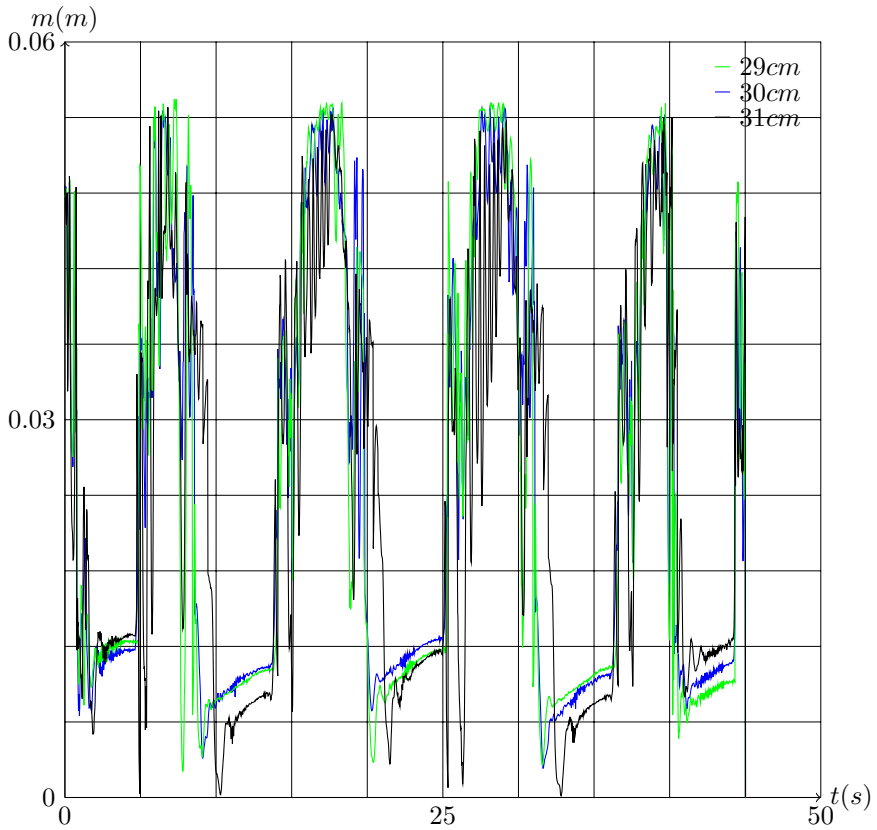


Figure 19: Stability margin versus time for various settings of z_m

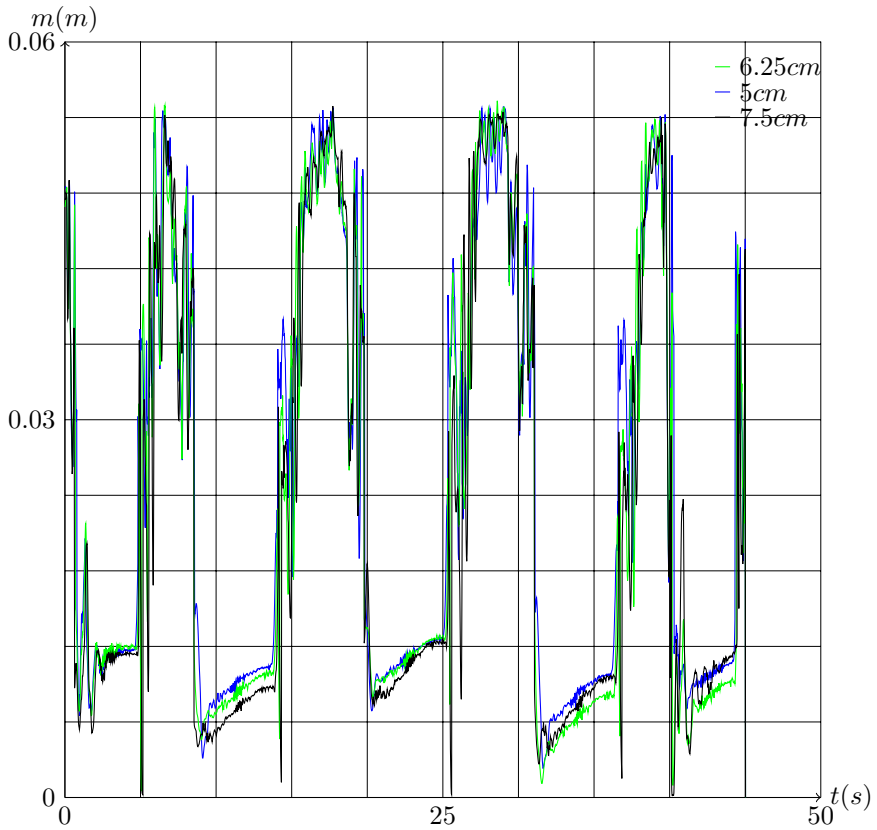
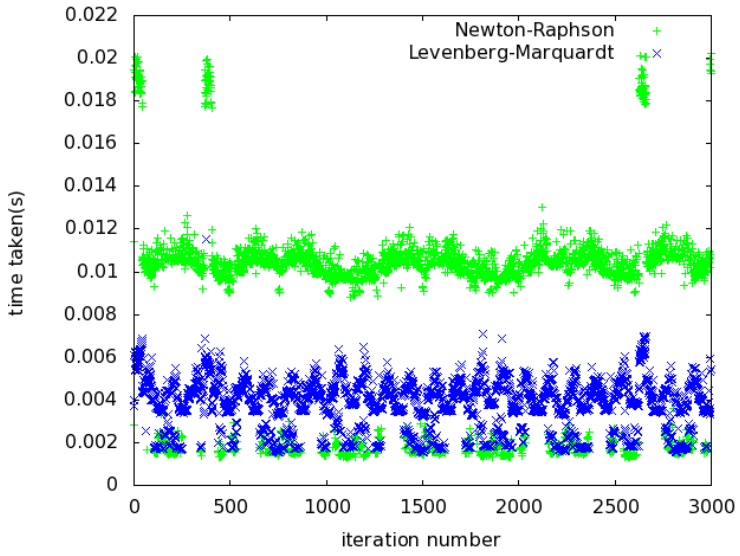


Figure 20: Time taken per iteration, Newton-Raphson and Levenberg-Marquardt



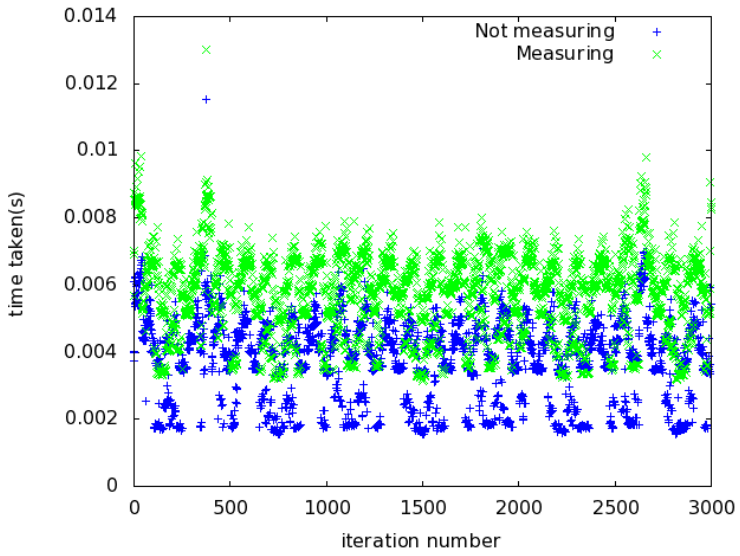
4.5 Controller performance

In this subsection, we consider two aspects of the controller's performance. First, we consider to what degree the controller works in real time. To make this precise, we note that the Nao runs a number of processes, including our controller, in a 20-millisecond cycle. To allow processes other than our controller to take place, the controller should take strictly less than 20-milliseconds. Second, we test how well the controller executes the trajectory it computes using inverse kinematics. In other words, we consider the difference between the angles measured by the Nao's sensors and the reference angles we provide.

As mentioned, we begin by considering the controller's real-time performance. To do so, we have measured the time taken for each iteration of the controller in a standardized walk. Since the time our controller takes depends on which algorithm is used for inverse kinematics, we have tested both Newton-Raphson and Levenberg-Marquardt. In these tests, the controller is not performing any measurements of the center of pressure. Thus, we quantify only how long the controller needs to calculate the target position using the LIPM and perform inverse kinematics. The resulting time taken per iteration is plotted versus iteration number in Figure 20. In this figure, we see that Levenberg-Marquardt is significantly faster than Newton-Raphson in all iterations. With a number of exceptions, both algorithms are fast enough to run in real-time in most iterations. Even in the exceptional cases where Newton-Raphson is too slow, Levenberg-Marquardt is still fast enough. Thus, provided we use Levenberg-Marquardt, our controller can satisfy its real-time requirements.

In addition to performing inverse kinematics and calculating the LIPM trajectory, our controller performs a number of measurements. Ideally, it should be able to perform these measurements within the same 20-millisecond cycle. To test this, we consider the difference in time taken with and without measuring the stability margin. For this test, we have used Levenberg-Marquardt and the same standardized walking situation as above. The resulting time taken is shown in Figure 21. In this figure, we see that measurement appears to take a relatively small constant amount of time. Even with measurement, the controller can satisfy its real-time

Figure 21: Time taken per iteration, with and without measuring the stability margin, using Levenberg-Marquardt



requirements even in exceptional cases.

We will now continue by considering how well the controller executes the trajectory it generates. To do so, we have measured the difference between the reference angles provided by the controller and the angles measured by the Nao's sensors for a number of joints in a standardized walking situation. Ideally, these differences, also known as tracking errors, should be very small. Unfortunately, this does not appear to be the case for our controller, as shown in Figure 22. In this figure, we see that the tracking error of the moving foot's ankle joint tends to be rather high. As we see in Figure 23, the hip's YawPitch joint has smaller, but still significant tracking error. As our final example, we consider another hip joint in Figure 24. In this figure, we see a tracking error much smaller than those of the other joints we've considered. In our experience, the other joints tend to give similar results to these examples. As such, we conclude that our controller suffers from significant tracking error. One potential method to eliminate this error might be to change the speed parameter passed to the Nao's controllers. This speed value, ranging from 0 to 1, allows one to indicate with which fraction of its total speed the joint should move. In our tests, no value of this parameter resulted in significantly lower mean tracking error across all joints. Hence, we suspect that a different solution is needed.

5 Conclusions and future work

To conclude our paper, we will consider each of the aspects of walking we considered in the introduction and evaluate how well our system performs each of them. The first of our aspects was trajectory generation, that is, how the Nao should move while walking. Our approach here was to generate a trajectory where the linear inverted pendulum mode of Kajita et al. [9, 8] defines the torso motion and the foot motion is specified using polynomials. As our experiments in the previous section show, the motion specified by this trajectory is stable for a reasonably wide range of its parameters. Thus, we have succeeded in creating a trajectory for a

Figure 22: Tracking error versus time for the moving foot's ankle

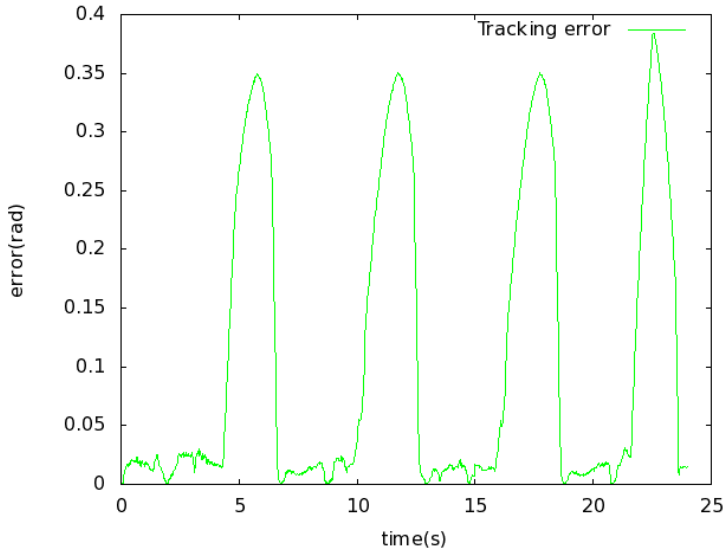


Figure 23: Tracking error versus time for the hip's YawPitch joint

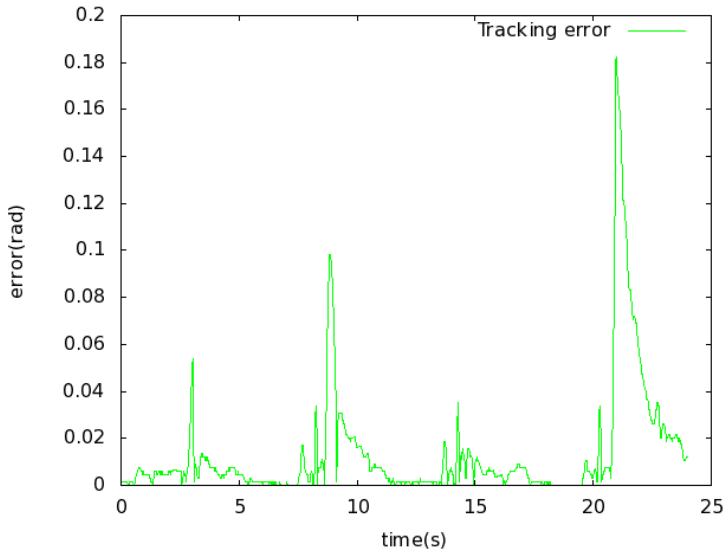
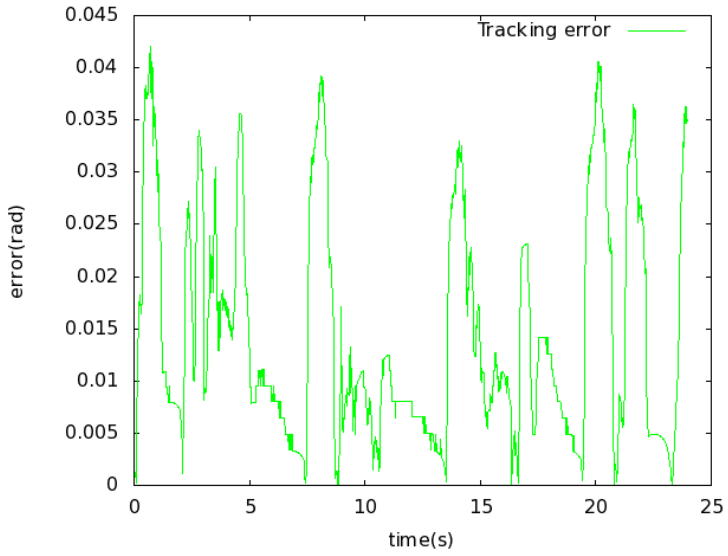


Figure 24: Tracking error versus time for a hip joint



stable walk. The primary limitation of this trajectory is that it is entirely specified in advance, that is, that it does not depend on anything that occurs while the Nao is walking. An interesting area for future research would be to see how we can modify this trajectory while walking to deal with unstable motion, uneven floor surfaces and other unexpected situations.

Our second aspect was trajectory execution. To do this, we used a number of numerical techniques of inverse kinematics to transform the trajectory to a series of joint angle vectors, which could then be passed to the Nao's controllers. Our best algorithm for inverse kinematics, Levenberg-Marquardt, is able to handle kinematic singularities. Its primary limitation is that it does not take into account the lower and upper bounds on the angles of the Nao's joints, causing it to sometimes give answers which the Nao cannot reach. On the other hand, as we have seen in the previous section, it allows us to run our controller in less than 20 milliseconds, thus allowing it to run in real time alongside other processes. As we have also seen previously, this controller still suffers from significant tracking error. In future research, we may be able to eliminate this error by changing the controller.

Our final aspect was measuring the stability of a robot while walking. To measure stability, we measured the center of pressure using the force sensors on the Nao's feet. We then computed the distance between the center of pressure and the area of support. We called this distance the stability margin, as it allowed us to quantify the stability of the Nao. As we have seen, this procedure allowed us to measure the stability of the Nao in real time. A downside of this approach is that we did not have any test data for which the correct value of the center of pressure was known. As a result, we had to rely on isolating sources of error by reasoning and experimentation. In future research, a method could be found to calculate the center of pressure and related quantities on the Nao using formulae given in the literature, allowing us to verify and improve our measurements.

A Numerical evaluation of the LIPM equations

We remarked in the main text that the solution to the LIPM equations given in equation 16 was ill-suited to numerical evaluation. The most pressing problem of this kind is that in a number of conditions, equation 17 for A_p will involve a subtraction of nearly equal quantities, leading to catastrophic cancellation. The noisy result for A_p will then be multiplied by a potentially rather large exponential, which will lead to significant deviation from our analytical results.

To illustrate the cause of this problem and how we can avoid it, consider the x-direction. Theoretically, numerical problems can occur if either t_s is low and t_d is high or vice-versa. We will consider the second case, where t_s is high and t_d is low. Then, equation 24 implies that $\alpha_x \approx s_x\sqrt{\frac{g}{z_c}}$. At the same time, we find that $\beta_x \approx 0$, using equation 23. Substituting this in equation 17, we find $A_x \approx \frac{0-s_x+s_x\sqrt{\frac{g}{z_c}}\sqrt{\frac{z_c}{g}}}{2} = \frac{-s_x+s_x}{2}$. One can clearly see the catastrophic cancellation of the nearly equal quantities $-s_x$ and $\alpha_x\sqrt{\frac{z_c}{g}}$. To avoid this, one can substitute the solutions for α_x and β_x given by equations 24 and 23 into equation 17. The resulting expression after simplification does not suffer from this cancellation. The same procedure should be applied in the y-direction, where these problems occur under similar conditions.

With these modifications, the equations we gave earlier can be evaluated reasonably. One can still find numerical problems due to cancellation in B_p if t_s is low but t_d is high, which primarily impact the early parts of steps. Another problem one may still encounter is that the exponentials in equation 16 can overflow, which can lead to nonsensical results.

B Computing the quaternion corresponding to a rotation matrix

As mentioned in the main text, we have yet to consider how to find the quaternion r corresponding to an arbitrary rotation matrix M. The method we give here works in all cases, which makes it suitable for numerical use. The approach we take is to first calculate the rotation axis v and the associated rotation angle ϕ. The quaternion we're looking for is then given by Equation 54.

First, we consider how to find v. As Angeles [2, Section 2.3] mentions, the axis v is a unit eigenvector of M corresponding to eigenvalue 1.[14] Thus, v can be found by calculating the unit eigenvectors of M and selecting one corresponding to the eigenvalue 1.

Calculating the rotation angle ϕ is more complex. To do this, we will use a number of concepts and formulas from Angeles' textbook [2, Section 2.3.4]. First, we will recall the concepts of the trace $\mathrm{tr}(M)$ and axial vector $\mathrm{vect}(M)$ of M. The trace is given by the sum of the diagonal elements of M. Thus, $\mathrm{tr}(M) = \sum_{i=1}^{3} M_{ii}$, where M_{ij} is the element of M in the i-th row and j-th column. The axial vector $\mathrm{vect}(M)$ is given by $\mathrm{vect}(M) = \frac{1}{2}\begin{bmatrix} M_{32}-M_{23} \\ M_{13}-M_{31} \\ M_{21}-M_{12} \end{bmatrix}$. For rotation matrices, Angeles [2, Section 2.3.4] shows that $\mathrm{tr}(M) = 1 + 2\cos\phi$, and $\mathrm{vect}(M) = v\sin\phi$. Therefore, $\cos\phi = \frac{\mathrm{tr}(M)-1}{2}$, and $\sin\phi = \mathrm{vect}(M)\cdot v$.[15] Then, we can find ϕ using

[14] Equivalently, $Mv = v$, and v has magnitude 1.

[15] $a \cdot b$ is the dot product of the vectors a and b, defined as $\sum_{i=1}^{n} a_i b_i$

$\phi = \text{atan2}(\sin\phi, \cos\phi) = \text{atan2}\left(\text{vect}(M)\cdot v, \frac{\text{tr}(M)-1}{2}\right)$.[16] As mentioned, we can now compute the quaternion using Equation 54.

References

[1] Aldebaran Robotics (2009). Nao documentation.

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

[3] Buss, Samuel R. (2009). Introduction to inverse kinematics with jacobian transpose, pseudoinverse and damped least squares methods. `http://math.ucsd.edu/~sbuss/ResearchWeb/ikmethods/index.html` Retrieved April 26th 2011.

[4] Goswami, Ambarish and Kallem, Vinutha (2004). Rate of change of angular momentum and balance maintenance of biped robots. *Proceedings of the 2004 IEEE International Conference on Robotics and Automation*.

[5] Goswami, Ambarish (1999). Foot rotation indicator (FRI) point: a new gait planning tool to evaluate postural stability of biped robots. *Proceedings of the 1999 IEEE International Conference on Robotics and Automation*.

[6] Gouaillier, David, Hugel, Vincent, Blazevic, Pierre, Kilner, Chris, Monceaux, Jérôme, Lafourcade, Paascal, Marnier, Brice, Serre, Julien, and Maisonnier, Bruno (2008). The NAO humanoid: a combination of performance and affordability. *arXiv.org*.

[7] Hirai, Kazuo, Hirose, Masato, Haikawa, Yuji, and Takenaka, Toru (1998). The development of Honda humanoid robot. *Proceedings of the 1998 IEEE International Conference on Robotics & Automation*.

[8] Kajita, Shuuji and Tani, Kazuo (1991). Study of dynamic biped locomotion on rugged terrain - derivation and application of the linear inverted pendulum mode -. *Proceedings of the 1991 IEEE International Conference on Robotics and Automation*.

[9] Kajita, Shuuji, Kanehiro, Fumio, Kaneko, Kenji, Yokoi, Kazuhito, and Hirukawa, Hirohisa (2001). The 3d linear inverted pendulum mode: A simple modeling for a biped walking pattern generation. *Proceedings of the 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems*.

[10] Kajita, Shuuji, Kanehiro, Fumio, Kaneko, Kenji, Fujiwari, Kiyoshi, Harada, Kensuke, Yokoi, Kazuhito, and Hirukawa, Hirohisa (2003). Biped walking pattern generation by using preview control of zero-moment point. *Proceedings of the 2003 IEEE International Conference on Robotics and Automation*.

[11] Mitobe, K., Capi, G., and Nasu, Y. (2000). Control of walking robots based on manipulation of the zero moment point. *Robotica*, Vol. 18, pp. 651–657.

[12] O'Rourke, Joseph (1998). *Computational Geometry in C*. Cambridge University Press.

[16] The function atan2 is an analogue of the standard arctangent that takes into account the quadrant in which its argument lies. It is available in the ISO C library.

[13] Poskriakov, Sergei (2006). Humanoid balance control: A comprehensive review. M.Sc. thesis, University of Geneva, Faculty of Science, Computer Science Dept.

[14] Sardain, Philippe and Bessonnet, Guy (2004). Forces acting on a biped robot. center of pressure - zero moment point. *IEEE Transactions on Systems, Man and Cybernetics - Part A: Systems and Humans*, Vol. 34.

[15] Stanford University (Winter 2007/2008). CS223A: Introduction to robotics. `http://see.stanford.edu/see/courseinfo.aspx?coll=86cc8662-f6e4-43c3-a1be-b30d1d179743`, retrieved April 25st 2011.

[16] Vukobratović, Miomir and Borovac, Branislav (2004). Zero-moment point - thirty five years of its life. *International Journal of Humanoid Robotics*, Vol. 1, pp. 157–173.