Playing Tic-tac-toe with the NAO humanoid robot

Renzo Poddighe

July 1, 2013

Abstract

This paper describes the challenges that have to be tackled when playing a game of tic-tac-toe with the NAO humanoid robot. The most important aspects of tic-tac-toe for a robot are solving the Inverse Kinematics problem, and analyzing the game board using image processing. It succeeded in drawing om the board using the FABRIK algorithm for moving the arm, and Hough line and circle transform for recognizing the board and the symbols.

1 Introduction

This paper covers the aspects involved in enabling a NAO humanoid robot to physically play a game of Tic-tac-toe against a human player. The main focus is on coordinating the NAO's arm movements, which concerns solving the Inverse Kinematics problem, but also recognizing the board using image processing.

Moving the arm may seem like a straightforward task, but is in fact quite difficult to accomplish for a robot. A human's capability of interpreting its environment more heuristically outperforms the robot's generally more mathematical approach in this case. The mathematical description of the Inverse Kinematics problem is a non-linear system of equations with possibly multiple solutions. Traditional methods are computationally expensive, because they rely on constructing and operating on large and complex matrices. Furthermore, the NAO does not possess that much processing power. An computationally cheap approach that comes up with solutions more efficiently is therefore the preferred approach in this case.

Recognizing the board is another example of task which is obvious for a human because of the brain's ability to interpret and match patterns, but a more difficult task for a robot. Fortunately, image processing software containing algorithms that are sufficient for this relatively simple task is freely available. Playing Tic-tac-toe is a well-known problem, and is easily solvable by a computer. A basic minimax search algorithm is sufficient in this research.

1.1 Research questions

The research questions this paper focuses on are:

1. How is the drawing of a circle and a cross in the correct square of the game board coordinated?
2. Is the NAO capable of recognizing the figures on the board and interpreting them in the context of tic-tac-toe?

1.2 Structure

This section contains an overview of all the tasks that are tackled in this paper, as well as the structure of this paper and an overview of the terms used in this paper. In Section 2, information about the hardware and software is listed. Section 3 describes the preliminaries and identifies relevant research challenges. Section 4 describes the approaches used to enable the NAO to play tic-tac-toe. The implementation of these approaches can be found in Section 5. Finally, the conclusion about the research and some suggestions for future research can be found in Section 6.

1.3 Terminology

In this paper, the following terminology will be used.

- **Kinematic chain:** a series of links connected by joints.
- **Joint:** a point in a kinematic chain rotatable along one or multiple axes.
- **Link:** a rigid body (i.e. solid, not deformable) within a kinematic chain.
- **End effector:** the final link in a kinematic chain.
- **Root:** a fixed point, the beginning of a kinematic chain.
- **Degree(s) of freedom (DOF):** the number of independent parameters that define the configuration of a kinematic chain.

2 Environment

This section describes the hardware components and the technical specifications of the NAO. An overview of the software that was used will be given as well.

2.1 Hardware

The NAO is a humanoid robot developed by the French company Aldebaran Robotics, founded in 2005.[1]

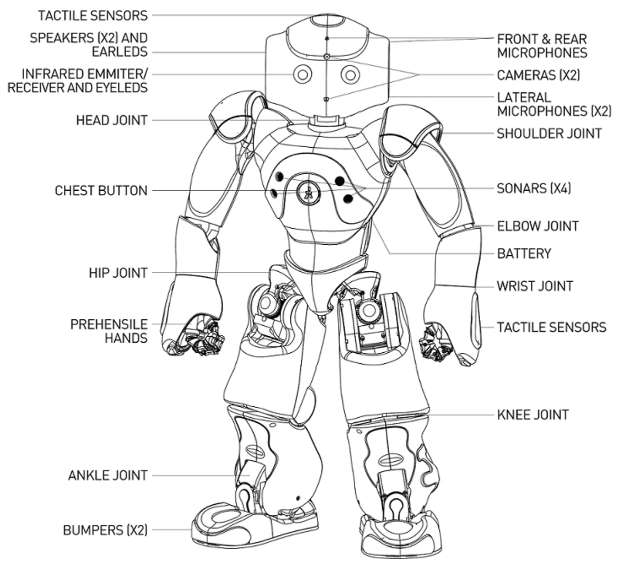

Figure 1: NAO joints and sensors.

All of the joints in Figure 1 can move independently of each other. The technical specifications of the NAO are listed in Table 1.

Technical Specifications	
Version	3.2
Body type	H25
Degrees of freedom	25
Height	573,2 mm
Weight	4,8 kg
Autonomy	60-90 min.
CPU	x86 AMD GEODE 500MHz CPU
Memory	256 MB SDRAM / 2 GB flash memory
Cameras	2 x VGA@30fps
Connectivity	Ethernet, Wi-Fi

Table 1: Technical Specifications.

The arm

The arm of the NAO is a kinematic chain consisting of three joints and two links. Table 2 lists the joints in the arm, and the axes around which they are able to rotate. The x-axis is parallel to the shoulders of the NAO, the y-axis points in front of the NAO and the z-axis is the vertical axis. Note that the shoulder joint and the elbow joint both have two degrees of freedom, whereas the hand joint only has one.

Joint name	**Axes**
Shoulder	x-axis, z-axis
Elbow	y-axis, z-axis
Hand	y-axis

Table 2: List of joints and their rotation axes in the arm of the NAO.

2.2 Software

The NAOqi SDK is a cross-platform, cross-language programming framework in which all programs for the NAOs are written. The framework allows creating new modules intercommunicating with standard and/or custom modules, and loading these as programs onto the NAO. It is cross-platform because it is possible to run it on Windows, Mac or Linux. It is cross-language because it supports a wide range of programming languages. It is only possible to write local modules using C/C++ and Python. For remotely accessing and controlling the NAOs, however, NAOqi also supports .NET, Java, Matlab and Urbi. The NAOqi SDK version 1.14 was used in this research.

- **Programming language:** C/C++. This was the obvious choice to make, since C++ is described on the Aldebaran website as the 'most complete framework', and it is the only language that allows the writing of real-time code, making the software run much faster on the NAO. Furthermore, when using C++, there is no need for a wrapper for another language in order to make OpenCV (the image processing toolbox used for this research) work, since this is also written in C++.
- **Linear algebra:** Eigen[5]. Eigen is a highly optimized C++ template library used for linear algebra. All operations involving matrices or vectors are done using Eigen. Version 3.1.3 was used in this research.
- **Building/Cross-compilation:** qiBuild[2]. This cross-platform compiling tool, based on CMake, makes creating and building NAOqi projects easy by managing dependencies between projects and supporting cross-compilation (ability to build binary files executable on a platform different from the building platform).
- **Image processing:** Open Source Computer Vision (OpenCV).[6] This is the image processing library that is supported by the NAOqi SDK. Version 2.3.1 was used in this research, since this is the latest version supported by the NAOqi SDK v1.14.
- **Higher-level robot control:** Choregraphe. Choregraphe is a graphical tool developed by Aldebaran Robotics that allows easy access to and control of the robot. From within Choregraphe, it is

possible to control individual joints or create a sequence of existing modules to be executed by the robot.

3 Preliminaries

3.1 Forward Kinematics

Forward Kinematics can be described as the problem of calculating the position of the end effector (or any other joint) of a kinematic chain from the current joint angles of that chain. In other words, Forward Kinematics is the problem of mapping the joint space of a kinematic chain to the Cartesian space. Unlike Inverse Kinematics, Forward Kinematics is straightforward in deriving the equations, always has a solution, and can be solved analytically.

The *kinematics equations*[8] are the equations in which the position and orientation of the target joint are described. These equations are a sequence of *affine transformations*, a transformation in which the ratios of distances between every pair of points are preserved. To represent affine transformations, so-called *homogeneous* coordinates must be used. This means describing an n-vector as an $(n+1)$-vector, by adding a 1. For example, when applying it to the case of the NAO, a joint coordinate in three dimensions (x, y, z) is represented by the vector $(x, y, z, 1)$. This is necessary because it is now possible to describe translations using matrix multiplication, as shown in Equation 1:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{1}$$

The translation is described by the 4-by-4 matrix, which is a transformation matrix containing the translation's homogeneous coordinates:

$$Tr(\boldsymbol{t}) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2}$$

where $\boldsymbol{t}$ is the vector (t_x, t_y, t_z). This matrix can be included in the kinematics equations to describe the translations along the links of the kinematic chain. The rotations around the given joint angles are represented by *rotation matrices*: matrices describing a rotation around an axis with a certain angle θ. In three dimensions, there are three rotation matrices, one for each axis:

$$\begin{aligned} R_x(\theta) &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \\ R_y(\theta) &= \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \\ R_z(\theta) &= \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned} \tag{3}$$

To include these in the kinematics equations, they have to be rewritten as homogeneous matrices. This is done by adding a row of zeros and a column of zeros to the matrix, and replacing the bottom-right entry with a 1:

$$\begin{aligned} R_x(\theta) &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ R_y(\theta) &= \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ R_z(\theta) &= \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned} \tag{4}$$

Using the information about the joint axes in Table 2, the final transformation matrix T can be constructed using the corresponding sequence of homogeneous rotation and translation matrices, as shown in Equation 5:

$$T = R_x(\theta_1)R_z(\theta_2)Tr(\boldsymbol{l}_1)R_y(\theta_3)R_z(\theta_4)Tr(\boldsymbol{l}_2) \tag{5}$$

$\boldsymbol{l}_1$ is the translation along the first link of the chain l_1 (represented by the homogeneous vector $(0, l_1, 0, 1)$), and $\boldsymbol{l}_2$ is the translation along the second link. Equation 5 can be broken down into smaller pieces and solved sequentially, in order to calculate the coordinates of the intermediate joints, as explained in Section 4.1.

3.2 Inverse Kinematics

Inverse Kinematics (IK) is the exact opposite of Forward Kinematics: the problem of calculating the joint configurations of a kinematic chain corresponding to the desired position of the end effector, or in other words, mapping the desired joint coordinate in Cartesian space back to the corresponding configurations in the joint space[8].

Let $\boldsymbol{\theta} = \theta_1, \theta_2, ..., \theta_n$ be the n joint configurations in the kinematic chain. Then let $\boldsymbol{s}$ be the end effector position, which can be described as a function of the joint configurations $\boldsymbol{s} = f(\boldsymbol{\theta})$, and $\boldsymbol{t}$ the target position. The Inverse Kinematics problem is to find values for $\boldsymbol{\theta}$ such that $\boldsymbol{s} = \boldsymbol{t}$. Since not all points in Cartesian space map to a joint configuration, there is no straightforward inverse function $f^{-1}(\boldsymbol{t}) = \boldsymbol{\theta}$ for Inverse Kinematics, as opposed to Forward Kinematics, for which a completely analytically derivable solution exists. Therefore, Inverse Kinematics solvers rely on numerical approaches.

3.3 Tic-tac-toe

Tic-tac-toe is a game traditionally played with pencil and paper, where two players, player X and player O, take turns drawing either an X or an O on a 3-by-3 grid. A player who draws three of the same marks in a horizontal, vertical or diagonal row wins the game. The game is a zero-sum game. When both players play with an optimal strategy, the game will always result in a tie. From a game theory perspective, this game is not very interesting. However, since it can be drawn on a whiteboard, is well interpretable by image processing techniques, and the strategy is straightforward to implement, it is a well-suited game for this research. The algorithm of choice is the minimax algorithm[9], a search algorithm which tries to minimize the maximum possible loss.

4 Algorithms

The algorithms that are relevant to this research are explained in this section. The algorithms involved in the coordination of the arm (i.e. forward and inverse kinematics) are considered relevant.

4.1 Calculating the joint coordinates

As explained in Section 3.1, the position of the end effector in a kinematic chain can be calculated by multiplying a sequence of affine transformations. When doing these transformations sequentially, the intermediate results contain the other joints in the chain.

Obviously, the root joint has coordinates $(0, 0, 0)$. Calculating the coordinates of the next joint involves a rotation along the z-axis with a given angle θ_1, a rotation along the x-axis with a given angle θ_2, and a translation along the y-axis of length l_1, which is the length of the first link. Thus, the kinematics equation of the second coordinate is as follows:

$$T_1 = R_x(\theta_1)R_z(\theta_2)Tr(\boldsymbol{l}_1) \tag{6}$$

The last column of the resulting matrix contains the homogeneous coordinates of the second joint in the chain. The first three columns contain its orientation.

To get from the intermediate joint to the end effector, two calculations have to be made. The relative rotation and translation of the second link has to be calculated. Then, this has to be translated along the first link in order to convert the relative transformation into an absolute transformation.

$$\begin{aligned} T_2 &= R_y(\theta_3)R_z(\theta_4)Tr(\boldsymbol{l}_2) \\ T &= T_1T_2 \end{aligned} \tag{7}$$

The matrix T contains the homogeneous coordinates of the end effector in the last column, and the orientation of the joint in the other columns.

4.2 The FABRIK algorithm

FABRIK (short for Forward And Backward Reaching Inverse Kinematics) is a novel heuristic method, developed by Aristidou and Lasenby[3], that tackles the Inverse Kinematics problem described in Section 3.2. Unlike traditional methods, FABRIK does not make use of calculations involving matrices or rotational angles. Instead, the IK problem is solved by finding the joint coordinates as being points on a line. These points are iteratively adjusted one at a time, until the end effector has reached the target position, or the error is sufficiently small. FABRIK starts at the end effector of the chain and works forwards, adjusting each joint along the way. Thereafter, it works backwards in the same way, in order to complete a full iteration. Since the use of rotational angles and matrices is avoided, the algorithm has low computational cost, converges quickly, and does not suffer from singularity problems. Furthermore, the algorithm produces realistic human-like poses and is easily implemented.

Algorithm 1 describes the FABRIK algorithm in pseudo-code. In Figure 2, a visualization of the algorithm is shown. The various steps of the algorithm, indicated with the letters (a) through (f) in Figure 2, are described in words below.

Since homogeneous coordinates are only used in Forward Kinematics, the n joint positions of the kinematic chain can be represented by the triplets $\boldsymbol{p}_i = (x_i, y_i, z_i)$ for $i = 1, 2, ..., n$, where $\boldsymbol{p}_1$ is the root joint and $\boldsymbol{p}_n$ the end effector (a). The target position is named $\boldsymbol{t}$ and the initial root position is named $\boldsymbol{b}$. The target position is reachable if the distance between the root joint and the target position, denoted as *dist*, is smaller than or equal to the sum of the distances between the joints $d_i = |\boldsymbol{p}_{i+1} - \boldsymbol{p}_i|$ for $i = 1, 2, ..., n-1$. If the target is reachable, the first stage of the algorithm starts. In this stage, named 'forward reaching', the joint positions are estimated by positioning the end effector on the target position $\boldsymbol{t}$ (b). The new position of the $n-1^{\text{th}}$ joint, $\boldsymbol{p}'_{n-1}$, lies on the line l_{n-1}, which passes through the point $\boldsymbol{p}_{n-1}$ and the new end effector position $\boldsymbol{p}'_n$, and has distance d_{n-1} from $\boldsymbol{p}'_n$ (c). Subsequently, the new joint position $\boldsymbol{p}'_{n-2}$ can be calculated by taking the point on the line l_{n-1} with distance d_{n-2} from $\boldsymbol{p}'_{n-1}$. The first stage of the algorithm is completed when all new joint positions have been calculated (d). The current estimate is not a feasible one, though, since the position of the root has changed. Therefore, a second stage of the algorithm is necessary to achieve a solution. This stage, named 'backward reaching', is similar to the first stage of the algorithm, only the operations are carried out the other way around: from the root to the end effector. The new root position $\boldsymbol{p}_1''$ is the initial root position $\boldsymbol{b}$ (e). The next joint position $\boldsymbol{p}_2''$ is then determined by taking the point on the line l_1, that passes through the points $\boldsymbol{p}_1''$ and $\boldsymbol{p}_2'$, with distance d_1 from $\boldsymbol{p}_1''$. This procedure is repeated for all other joints, and a full iteration is complete (f). The end effector is now closer to its target position. The algorithm is repeated until the end effector has reached its target, or the distance to the target is smaller than a user-defined threshold.

4.3 Calculating back to joint coordinates

The FABRIK algorithm provides a solution to the inverse kinematics of the arm, by giving the Cartesian coordinates of each joint relative to the root joint. However, the NAO needs to know the joint configurations corresponding to these coordinates. A mapping from the Cartesian coordinates to joint configurations is therefore necessary in order to make the NAO move its arm.

Recall the three rotation matrices and the translation matrix from Section 3.1:

$$\begin{aligned}
R_x(\theta) &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
R_y(\theta) &= \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
R_z(\theta) &= \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
Tr(\boldsymbol{t}) &= \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{aligned} \tag{8}$$

Assume that the FABRIK algorithm outputs the three joint coordinates $(0,0,0,1), (x_1,y_1,z_1,1), (x_2,y_2,z_2,1)$ to be the solution to the inverse kinematics problem for a target position $\boldsymbol{t} = (x_2,y_2,z_2,1)$. Equation 9 describes the forward kinematics equation for mapping from the first two (currently unknown) rotations θ_1 and θ_2 to the second joint coordinate $\boldsymbol{p}_1$:

Algorithm 1 The FABRIK algorithm.

```
Input: The joint positions $p_i$ for $i = 1,...,n$, the target
  position $t$ and the distances between each joint $d_i =$
  $|p_{i+1} - p_i|$ for $i = 1,...,n-1$.
Output: The new joint positions $p_i$ for $i = 1,...,n$.
  % The distance between root and target
  $dist = |p_1 - t|$
  % Check whether the target is within reach
  if $dist >= d_1 + d_2 + ... + d_{n-1}$ then
    % The target is unreachable
    for $i = 1,...,n-1$ do
      % Find the distance $r_i$ between the target $t$ and
      the joint position $p_i$
      $r_i = |t - p_i|$
      $\lambda_i = d_i/r_i$
      % Find the new joint positions $p_i$
      $p_{i+1} = (1-\lambda_i)p_i + \lambda_i t$
    end for
  else
    % The target is reachable; thus, set $b$ as the initial
    position of the joint $p_1$
    $b = p_1$
    % Check whether the distance between the end effec-
    tor $p_n$ and the target $t$ is greater than a tolerance
    $dif_A = |p_n - t|$
    while $dif_A > tol$ do
      % STAGE 1: FORWARD REACHING
      % Set the end effector $p_n$ as target $t$
      $p_n = t$
      for i = n-1, ..., 1 do
        % Find the distance $r_i$ between the new joint
        position $p_{i+1}$ and the joint $p_i$
        $r_i = |p_{i+1} - p_i|$
        $\lambda_i = d_i/r_i$
        % Find the new joint positions $p_i$
        $p_i = (1-\lambda_i)p_{i+1} + \lambda_i p_i$
      end for
      % STAGE 2: BACKWARD REACHING
      % Set the root $p_1$ at its initial position
      $p_1 = b$
      for i = 1, ..., n-1 do
        % Find the distance $r_i$ between the new joint
        position $p_i$ and the joint $p_{i+1}$
        $\lambda_i = d_i/r_i$
        % Find the new joint positions $p_i$
        $p_{i+1} = (1-\lambda_i)p_i + \lambda_i p_{i+1}$
      end for
      $dif_A = |p_n - t|$
    end while
  end if
```

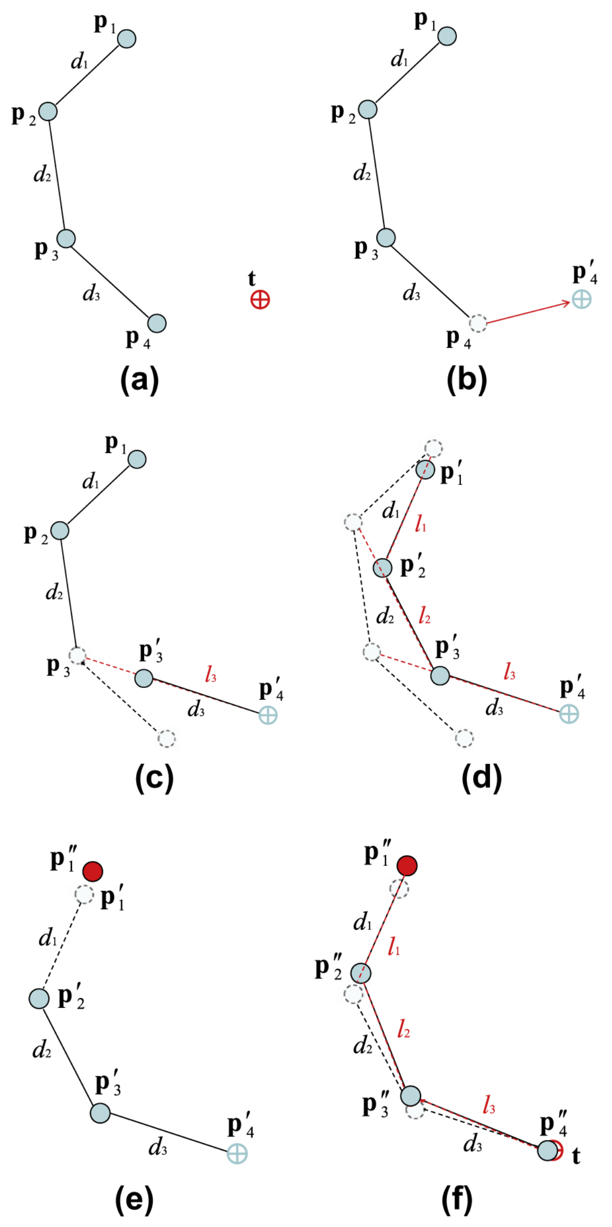

Figure 2: A visualization of one iteration of the FABRIK algorithm.

$$\boldsymbol{p}_1 = R_x(\theta_1)R_z(\theta_2)Tr(\boldsymbol{l}_1)\begin{bmatrix}0\\0\\0\\1\end{bmatrix} \tag{9}$$

which simply means taking the last column of the resulting transformation matrix. Because $\boldsymbol{p}_1$ is known, the joint angles can be derived from this equation. The expressions for the coordinates of the second joint can be found by rewriting Equation 9 as follows:

$$\begin{aligned} x_1 &= -l_1 \sin(\theta_2) \\ y_1 &= l_1 \cos(\theta_2)\sin(\theta_1) \\ z_1 &= -l_1 \cos(\theta_2)\sin(\theta_1) \end{aligned} \tag{10}$$

Since the second rotation (the one around the x-axis) does not affect the x-coordinate itself, the following expression for the first rotation θ_2 can be derived:

$$\theta_2 = \frac{-\arcsin(x_1)}{l_1} \tag{11}$$

Now that θ_2 is known, the other rotation θ_1 can also be derived:

$$\theta_1 = \frac{-\arcsin(z_1)}{l_1\cos(\theta_2)} \tag{12}$$

When expressing the end effector coordinates in the same manner (i.e. expressing the coordinates relative to the root joint), the expressions are not that simple anymore and the joint angles are not easily derivable anymore. So, the end effector coordinates have to be expressed relative to the second joint in the chain. Because the first joint angles are calculated, the orientation and position of the second joint can be captured in the following transformation matrix:

$$T = R_x(\theta_1)R_z(\theta_2)Tr(\boldsymbol{l}_1) \tag{13}$$

The end effector can be expressed relative to the second joint by multiplying its coordinates by the inverse of T:

$$\boldsymbol{p}'_2 = T^{-1}\boldsymbol{p}_2 \tag{14}$$

Equation 15 describes the forward kinematics equation for mapping from the last two rotations (to be calculated) θ_3 and θ_4 to the relative end effector coordinate $\boldsymbol{p}'_2$:

$$\boldsymbol{p}'_2 = R_y(\theta_3)R_z(\theta_4)Tr(\boldsymbol{l}_2)\begin{bmatrix}0\\0\\0\\1\end{bmatrix} \tag{15}$$

Which can be rewritten in the same manner as in Equation 10, which yields the following expressions for the end effector coordinates:

$$\begin{aligned} x'_2 &= -l_2 \cos(\theta_3)\sin(\theta_4) \\ y'_2 &= l_2 \cos(\theta_4) \\ z'_2 &= l_2 \sin(\theta_3)\sin(\theta_4) \end{aligned} \tag{16}$$

Again, since the second rotation (the one around the y-axis) does not affect the y-coordinate itself, the following expression for the first rotation θ_4 can be derived:

$$\theta_4 = \frac{-\arccos(y_2')}{l_2} \tag{17}$$

after which the last rotation θ_3 can be derived as well:

$$\theta_3 = \frac{-\arcsin(z_2')}{l_2 \sin(\theta_4)} \tag{18}$$

4.4 Hough line transform

The Hough line transform[6] is an algorithm that can be used to detect straight lines in an image. It is recommended to pre-process the image (e.g. converting it to a binary image) before applying the transform. Hough line transform first creates a sinusoid for each point in the image. This sinusoid represents the family of lines that go through this point using the polar coordinate system. A line equation can be written in polar coordinates using Equation 19:

$$r = x\cos\theta + y\sin\theta \tag{19}$$

The family of lines going through a point (x_0, y_0) can be found by substituting x_0 and y_0 in Equation 20:

$$r_\theta = x_0 \cdot \cos\theta + y_0 \cdot \sin\theta \tag{20}$$

meaning that each pair r_θ, θ represents each line that passes by (x_0, y_0).

A simple example is provided in Figure 3. The four white dots on the left are several points in the Cartesian coordinate frame. They each have a corresponding sinusoid in the polar coordinate frame, shown on the right side of the figure. The white dot on the right is the intersection between these four sinusoids. As you can see, that exact intersection corresponds to the line going right through all four points in the polar coordinate frame. The formula of this line is then determined by a line that comes from the origin (the center) of the image, and is perpendicular to the line.

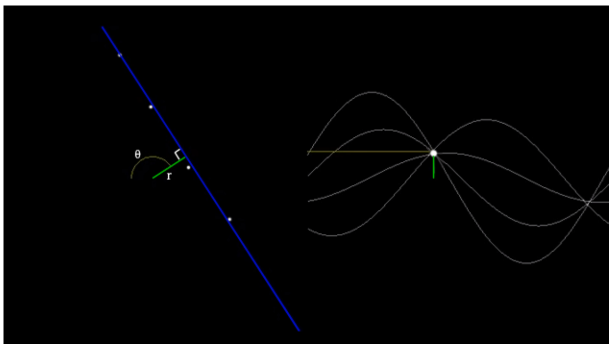

Figure 3: A visualization of the various sinusoids in the Cartesian coordinate frame corresponding to the white dots in the polar coordinate frame.

By applying Equation 20 to all points, (such that $r > 0$ and $0 < \theta < 2\pi$), we get a number of sinusoids equal to the number of points in the picture. If the curves of two different points intersect in the plane (θ, r), that means that both points belong to the same line. The result of this is that we can find straight lines by looking at the intersections of all the drawn curves. The more curves that intersect in one point, the more points there are on the line represented by that intersection. A threshold can be set to define the minimal number of intersections needed to detect a line. Hough line transform keeps track of the intersection between curves of every point in the image. If the number of intersections is above some threshold, it declares it as a line with parameters (θ, r) of the intersection point.

The Hough line transform has two implementations. The probabilistic version of the Hough line transform is a more efficient implementation for the algorithm described above. The output from the probabilistic Hough line transform differs; The Standard Hough line transform returns the lines in polar coordinates, whereas the Probabilistic Hough line transform returns the start and end point of each detected line in Cartesian coordinates. In this research, the probabilistic Hough line transform is used, because it is essential that the start and end points of the lines are defined.

4.5 Hough circle transform

Hough circle transform[7] works roughly the same as the Hough line transform mentioned in Section 4.4. The difference is that an extra dimension has to be added. Recall from Section 4.4 that a line can be defined in polar coordinates using the variables (r, θ). To define a circle, however, three parameters have to be used: (x, y, r), where (x, y) is the center of the circle and r the radius. This extra dimension in the definition of a circle means far greater memory requirements and much slower speed when using the same operations described in Section 4.4. A method to avoid the majority of unnecessary votes, called the Hough gradient method, is therefore used. This method uses the local gradient as a guideline to select candidate centers of the circle[4].

5 Implementation

In the previous sections, the solutions to all subproblems that have to be tackled have been proposed. In this section, the way in which the above algorithms are implemented to enable the NAO to actually play tic-tac-toe is explained. It is assumed that the whiteboard on which the game is drawn is clean, apart from the game board itself. Furthermore, the 3-by-3 grid is drawn symmetrically and the size is known.

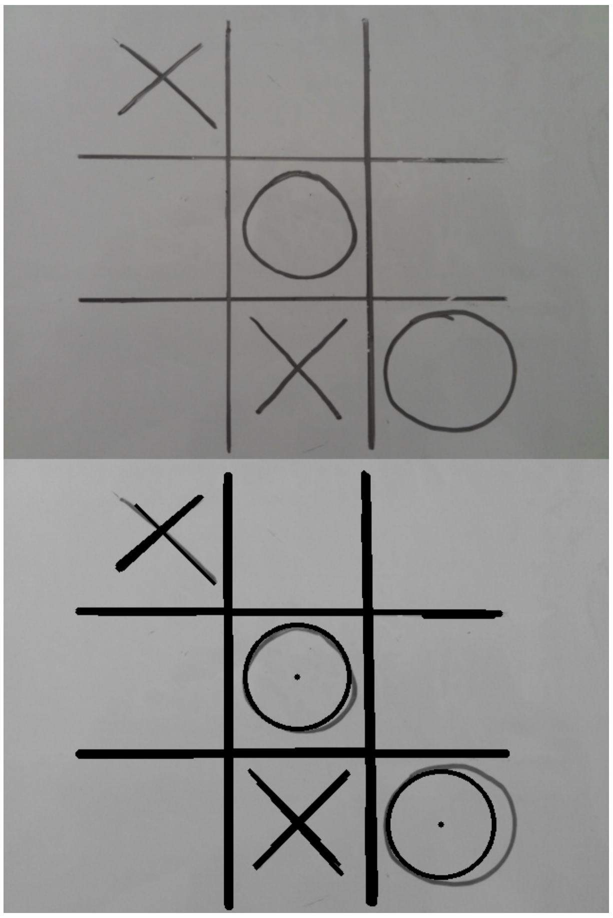

Figure 4: A picture of a tic-tac-toe board before and after line and circle detection. The detected lines and circles are drawn on top of the original image.

5.1 Processing the board

There are several types of information that have to be gathered from the board. Image processing is important for analyzing the state of the game so that the next move can be determined, and subsequently for determining the position to which the arm has to move next.

Analyzing the state of the game

In order to determine the next move, the state of the game has to be retrieved from the board. The game state can be represented by an array of nine entries, each of which is filled with a 0 if it is empty, a 1 if it contains a cross and a 2 if it contains a circle. Before analyzing the image, it is first preprocessed by a built-in OpenCV function **inRange()**. This function converts the image into a binary image, replacing a pixel by a 1 if it lies in the specified color range, and 0 otherwise. The algorithms mentioned in Section 4, the Hough line and Hough circle transform, are then used to find all lines and circles in the image. The image can be split up in 9 segments using the minimum and maximum x- and y-coordinates of the found lines. Then, for each found line, it can be checked whether the line falls into one of the segments. This is done by checking if the start and end coordinates of the line do so. If they do, the corresponding array entry is filled with a 1. In the same manner, for each found circle, it can be checked whether the circle falls into one of the segments by checking if the center of the circle does. If it does, the array entry is filled with a 2. When the array matches the game state, it can be processed by a minimax algorithm to determine the next move.

Calculating the coordinates

As explained in Section 4.2, FABRIK can be used to make the end effector of the NAO's arm move to a desired position in Cartesian space. In order for the NAO to be able to do this, however, the pixel coordinates first have to be converted to Cartesian coordinates. Since the size of the board is known, calculating the x- and y-coordinates is simple. The magnification M of the image can be found by:

$$M = \frac{X}{x} \tag{21}$$

where X is the actual width of the board and x the width of the projected board in pixels. The x- and y coordinates can then be calculated as follows:

$$\begin{aligned} x &= Mx' \\ y &= My' \end{aligned} \tag{22}$$

where x' and y' are the desired pixel coordinates the arm needs to move to.

The z-coordinate equals the distance d to the board. To know the relation between the distance of the board and the pixel size of the image, for instance, the focal length of the camera has to be calculated. This can be done by placing the NAO at a known distance d from the board, taking an image, and measuring the pixel width x of the board. The focal length f can then be calculated as follows:

$$f = x \times \frac{d}{X} \tag{23}$$

where X is the actual width of the board. The distance to the board d can now be calculated as follows:

$$d = X \times \frac{f}{x} \tag{24}$$

5.2 Drawing a cross

Once the coordinates of the target square are known, moving the NAO's arm is not difficult. In order for the NAO to be able to draw, it needs to slightly press the pen against the board. To do this, simply lower the stiffness of the arm to make it more flexible, and set the target slightly behind the board. The NAO starts its drawing at the top left of the square. The first diagonal line can now be drawn by making the arm move to a target at the bottom right of the square. The pen then has to be lifted from the board, by slightly adjusting the z-coordinate. Finally, the NAO moves its arm to the top right, presses the pen onto the board again, and then draws a similar line to the bottom left. To make the lines more straight, several intermediate targets can be set. So, a simple trajectory can be created by simply specifying targets in the direction the arm needs to move.

6 Conclusion

The proposed methods in this paper have dealt with the challenging subproblems - solving Inverse Kinematics and processing the game board - that arise when playing tic-tac-toe with the NAO. Using image processing, the NAO could analyze the game state, and make a decision based on that. Executing the decision implies drawing a cross on the board, which requires solving an inverse kinematic problem. The new algorithm FABRIK has a lot of advantages over traditional methods for solving an inverse kinematic problem: it is very accurate, converges quickly, and is well-behaved for every reachable point in space. Using this algorithm, the NAO was able to let its hand follow a trajectory, which results in drawing a cross on the board.

Future research

There is still room for improvement. The NAO's arm has a very limited range. The program can be made more flexible by increasing the space in which the NAO can draw. This can be done by involving the rest of the body in the drawing. For instance, walking along the board to increase the horizontal range, or using its knees to increase the vertical range. The NAO would then be able to play on a bigger game board.

Another improvement could be to provide some kind of feedback about the amount of pressure the NAO applies to the marker. If this can be kept constant, it can result in smoother drawings and more straight lines.

References

[1] (2013). Aldebaran Robotics. `http://www.aldebaran-robotics.com`.

[2] Aldebaran Robotics (2013). qibuild 1.14 documentation. `http://www.aldebaran-robotics.com/documentation`.

[3] Aristidou, Andreas and Lasenby, Joan (2011). Fabrik: A fast, iterative solver for the inverse kinematics problem. *Graphical Models*.

[4] Bradski, Gary and Kaehler, Adrian (2008). *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media.

[5] (2013). Eigen. `http://eigen.tuxfamily.org/`.

[6] OpenCV Development Team (2011). Opencv 2.3.1 documentation. `http://docs.opencv.org/2.3/modules/refman.html`.

[7] OpenCV Development Team (2012). Hough circle transform. `http://docs.opencv.org`.

[8] Paul, Richard P. (1981). *Robot Manipulators: Mathematics, Programming, and Control*. MIT Press.

[9] Russell, Stuart and Norvig, Peter (2009). *Artificial Intelligence: A Modern Approach*. Prentice Hall, third edition.