Comparing FABRIK and neural networks to traditional methods in solving Inverse Kinematics

Renzo Poddighe

June 18, 2013

Abstract

In this paper, two alternative methods to the Inverse Kinematics problem are compared to traditional methods regarding computation time, accuracy, and convergence rate. The test domain is the arm of the NAO humanoid robot. It shows that FABRIK, a heuristic iterative approximation algorithm that treats joint coordinates as being points on a line, outperforms the two traditional methods, which are both based on the Jacobian inverse technique, on all aspects. A neural network, a machine learning architecture which simulates the interconnections between the brain, vastly outperforms all algorithms regarding computation time, but lacks accuracy.

Keywords: Robot kinematics, machine learning.

1 Introduction

The Inverse Kinematics (IK) problem is a non-linear optimization problem which tries to optimize the position of the end effector of a robot's kinematic chain with respect to a certain target position, by manipulating the intermediate joint configurations in the chain[9]. These joint configurations have to be calculated in a backwards manner from the Cartesian coordinates. Since not all points in Cartesian space map to a joint configuration, there is not always a solution. It is very exceptional for a kinematic chain to have a complete analytically derivable solution. Therefore, Inverse Kinematics solvers rely on numerical approaches.

1.1 Traditional methods

The most commonly used method is using the inverse of the Jacobian matrix, a matrix of first-order partial derivatives of the joint system, to make a linear approximation of the non-linear function that describes the Inverse Kinematics. Since the Jacobian is not always nonsingular, the inverse can be approximated by a number of methods[5], each of which has its drawbacks. The transpose of the Jacobian is proven to be a good replacement for the inverse[14]. However, the joint configurations are often unpredictable and many iterations are needed for convergence. The Moore-Penrose pseudoinverse of the Jacobian is a better estimate, but often performs poorly because of instability for configurations near singularities. The Damped Least Squares (DLS) method avoids the use of pseudoinverses and provides a more stable solution near singularities. A damping constant has to be chosen carefully: a larger constant makes the algorithm more numerically stable, but also lowers the convergence rate. Pseudoinverse DLS attempts to overcome this problem by applying Singular Value Decomposition. Pseudo-inverse DLS performs similar to regular DLS away from singularities, and smooths out the performance of regular DLS near singularities. Selectively Damped Least Squares (SDLS) is an extension of Pseudoinverse DLS that also takes into account the relative positions of the end effector and the target position when choosing the damping constraint, resulting in fewer iterations needed for convergence, but a slower performance time. A general problem with all these methods is applying constraints, which is not at all straightforward to do using any Jacobian method. The existing methods do not guarantee optimal solutions and slow down performance times[7][13]. When coping with computational limitations on a robot, it is therefore appealing to search for effective alternatives that have low on-line computational cost and are robust against singularities. In this paper, two alternative approaches are proposed and compared on several criteria, such as speed and precision. The standard algorithm the two techniques are measured against uses the Jacobian inverse, estimated by the Moore-Penrose pseudoinverse.

1.2 Alternatives

In this paper, two alternatives are proposed.

FABRIK[3] (short for Forward And Backward

Reaching Inverse Kinematics) is a heuristic iterative method that tries to solve the IK problem by treating the joint coordinates as being points on a line. It uses the previously calculated positions of the joints to find the updates in a forward and backward iterative manner. The algorithm has low computational cost and converges quickly. Furthermore, FABRIK does not suffer from singularity problems, since the use of matrix inverses is completely avoided.

A neural network is a supervised learning method that is inspired by the interconnections between the neurons in the brain[12]. It consists of small computational units, called artificial neurons, that receive signals from other neurons, produce a signal from that, and pass it on to the next neurons. The goal of the neural network is to process the input signal so that it fits the output signal by training it. The network is fed training examples and adjusts its weights on the connections between the neurons to minimize the error on a specified output signal. Neural networks allow for non-linear function approximation and are therefore natural candidates for tackling a problem such as Inverse Kinematics.

1.3 Problem Statement

The problem statement for this research is as follows:

What are good and efficient alternatives for solving the Inverse Kinematics problem, compared to the traditional approaches?

1.4 Research Questions

The problem statement is accompanied by the following research questions:

- What test criteria are relevant for determining the better algorithm?
- Are the differences gathered from the experiments statistically significant?
- Is it possible to draw a general conclusion on which algorithm is better, based on the found results?

2 Environment

The robot that is used in this paper is the NAO humanoid robot, developed by the French company Aldebaran Robotics, founded in 2005.[1] A full list of technical specifications can be found in Appendix A. In this section, an overview of the software that was used will be listed.

2.1 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.
- **Linear algebra:** Eigen[6]. Eigen is a highly optimized C++ template library used for linear algebra. Mapping to and from joint coordinates/configurations makes use of matrix-vector operations, for which Eigen is used. Version 3.1.3 was used in this research.
- **Neural networks:** Encog[10]. Encog is an open source machine learning framework that focuses on neural networks. It contains classes used to create neural networks, to process and analyze data for these networks, and to train them using various training algorithms. A GUI workbench, programmed in Java, is available to design and train neural networks. Furthermore, there are libraries available for Java, .Net and C/C++.
- **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).
- **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

The background knowledge about the matter that is presented in this paper is provided in this section. The topics that are explained are Forward Kinematics and

Inverse Kinematics. Since Forward Kinematics is analytically solvable, its solution will be explained in this section. Only a short description of the much harder to solve Inverse Kinematics problem is given, since the remainder of the paper is devoted to its solutions.

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*[9] 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 4, 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 5.1.

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[9].

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.

4 The Jacobian inverse technique

The Jacobian inverse technique is a solution to the Inverse Kinematics problem that linearly approximates the inverse function $f^{-1}(\boldsymbol{t})$ using the Jacobian matrix[9]. The Jacobian is a function of the $\boldsymbol{\theta}$ values:

$$J(\theta)_{ij} = (\frac{\delta f_i(\theta_j)}{\delta \theta_j})_{ij} \tag{6}$$

In the case of a single end effector, $i = 1$, so J will be a 1-by-n matrix for n values of θ, whose entries are vectors in $\mathbb{R}^3$. An alternative representation is a 3-by-n matrix, one row for each coordinate. The matrix entries can be approximated as follows:

$$(\frac{\delta f_i(\theta_j)}{\delta \theta_j})_{ij} \approx (\frac{f_i(\theta_j + \epsilon) - f_i(\theta_j)}{\epsilon})_{ij} \tag{7}$$

for some small number ϵ. Since the entries in the Jacobian are first partial derivatives of the joint system, the relative movement of the end effector $\Delta\boldsymbol{s}$ can be estimated as

$$\Delta\boldsymbol{s} \approx J\Delta\boldsymbol{\theta} \tag{8}$$

Recall that the Inverse Kinematics problem is about finding the right joint configurations corresponding to a certain target position. In terms of the Jacobian, the IK problem can therefore be rewritten as:

$$\Delta\theta = J^{-1}\Delta\boldsymbol{s} \tag{9}$$

Unfortunately, the Jacobian is not always invertible. There are various ways to approximate the Jacobian inverse. It is possible to take the transpose instead, which is proven to be a good approximation when scaled by some small scalar α[5]. Another technique is taking the Moore-Penrose pseudoinverse, which is defined for all m-by-n matrices[4]. This is a better approximation, and generally converges to a solution more quickly. The Jacobian pseudoinverse, denoted by $J^\dagger$, can be calculated using one of the two following equations, depending on the number of rows and columns:

$$\begin{aligned} J^\dagger &= J^T(JJ^T)^{-1} \quad \text{if } m < n \\ J^\dagger &= (J^TJ)^{-1}J^T \quad \text{if } m > n \end{aligned} \tag{10}$$

which can be used to approximate the joint configurations:

$$\Delta\theta = J^\dagger\Delta\boldsymbol{s} \tag{11}$$

When J is full row rank, (JJ^T) and (J^TJ) are guaranteed to be invertible. A general formula for the pseudoinverse for J not of full row rank can be found in [4]. Equation 11 can be applied iteratively until the error drops down to a certain threshold. The algorithm is easily implemented and is computationally fast. The big downside is its instability for configurations near singularities.

The Jacobian transpose as well as the Jacobian pseudo-inverse method are used in this paper for comparison purposes. These are natural choices for baseline algorithms, since the Jacobian inverse methods have been the standard in solving Inverse Kinematics for a very long time[9].

5 FABRIK

This section covers the way the FABRIK algorithm works. In addition, the algorithm used to calculate the joint coordinates is described, as well as the algorithm to calculate back from the joint coordinates found by FABRIK to the joint coordinates.

5.1 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:

$$T_1 = R_x(\theta_1)R_z(\theta_2)Tr(\boldsymbol{l}_1) \tag{12}$$

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.

$$\begin{aligned} T_2 &= R_y(\theta_3)R_z(\theta_4)Tr(\boldsymbol{l}_2) \\ T &= T_1T_2 \end{aligned} \tag{13}$$

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.

5.2 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 1, a visualization of the algorithm is shown. The various steps of the algorithm, indicated with the letters (a) through (f) in Figure 1, 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.

5.3 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:

$$\begin{aligned} R_x(\theta) &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ R_y(\theta) &= \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ R_z(\theta) &= \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ Tr(\boldsymbol{t}) &= \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned} \tag{14}$$

Assume that the FABRIK algorithm outputs the three joint coordinates

Algorithm 1 The FABRIK algorithm.

Input: The joint positions $\boldsymbol{p}_i$ for $i = 1, ..., n$, the target position $\boldsymbol{t}$ and the distances between each joint $d_i = |\boldsymbol{p}_{i+1} - \boldsymbol{p}_i|$ 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|
        λ_i = d_i / r_i
        % Find the new joint positions p_i
        p_{i+1} = (1 − λ_i) p_i + λ_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 effector 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|
            λ_i = d_i / r_i
            % Find the new joint positions p_i
            p_i = (1 − λ_i) p_{i+1} + λ_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}
            λ_i = d_i / r_i
            % Find the new joint positions p_i
            p_{i+1} = (1 − λ_i) p_i + λ_i p_{i+1}
        end for
        dif_A = |p_n − t|
    end while
end if
```

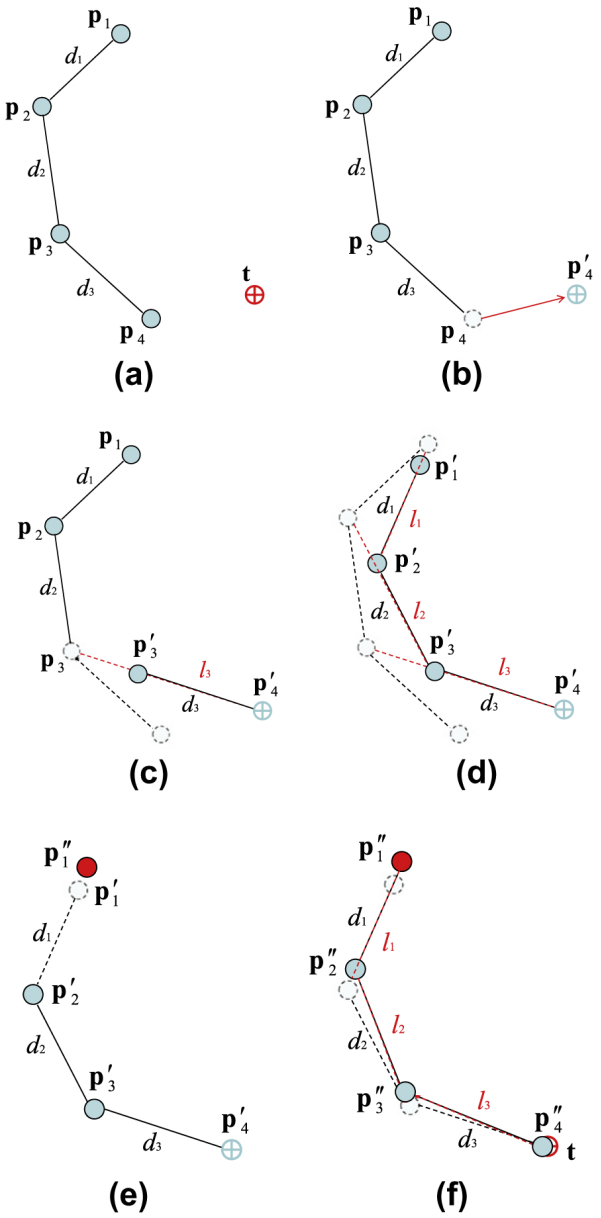

Figure 1: A visualization of one iteration of the FABRIK algorithm.

$(0, 0, 0, 1), (x_1, y_1, z_1, 1), (x_2, y_2, z_2, 1)$ to be the solution to the inverse kinematics problem for a target position $\boldsymbol{t} = (x_2, y_2, z_2, 1)$. Equation 15 describes the forward kinematics equation for mapping from the first two (currently unknown) rotations θ_1 and θ_2 to the second joint coordinate $\boldsymbol{p}_1$:

$$\boldsymbol{p}_1 = R_x(\theta_1) R_z(\theta_2) Tr(\boldsymbol{l}_1) \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{15}$$

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 15 as follows:

$$\begin{aligned} x_1 &= -l_1 \sin(\theta_2) \\ y_1 &= l_1 \cos(\theta_2) \sin(\theta_1) \\ z_1 &= -l_1 \cos(\theta_2) \sin(\theta_1) \end{aligned} \tag{16}$$

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:

$$\theta_2 = \frac{-\arcsin(x_1)}{l_1} \tag{17}$$

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

$$\theta_1 = \frac{-\arcsin(z_1)}{l_1 \cos(\theta_2)} \tag{18}$$

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:

$$T = R_x(\theta_1) R_z(\theta_2) Tr(\boldsymbol{l}_1) \tag{19}$$

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

$$\boldsymbol{p}_2' = T^{-1} \boldsymbol{p}_2 \tag{20}$$

Equation 21 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 $\boldsymbol{p}_2'$:

$$\boldsymbol{p}_2' = R_y(\theta_3) R_z(\theta_4) Tr(\boldsymbol{l}_2) \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{21}$$

Which can be rewritten in the same manner as in Equation 16, which yields the following expressions for the end effector coordinates:

$$\begin{aligned} x_2' &= -l_2 \cos(\theta_3) \sin(\theta_4) \\ y_2' &= l_2 \cos(\theta_4) \\ z_2' &= l_2 \sin(\theta_3) \sin(\theta_4) \end{aligned} \tag{22}$$

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:

$$\theta_4 = \frac{-\arccos(y_2')}{l_2} \tag{23}$$

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

$$\theta_3 = \frac{-\arcsin(z_2')}{l_2 \sin(\theta_4)} \tag{24}$$

6 Neural networks

A neural network[12] is a computational architecture used in the field of machine learning, inspired by the highly interconnected structure of the brain, aiming to benefit from its beneficial properties such as parallelism, generalization, fault tolerance, and adaptivity. It is a learning method that learns a mapping from input to output by being fed training examples and minimizing the error between the network's output and the desired output using a learning algorithm. Such a mapping is called a neural network *model*. A neural network is composed of *artificial neurons*: highly interconnected processing elements working in parallel to solve a specific problem. An artificial neuron's architecture is modeled after the structure of a natural neuron. It has a number of weighted signals coming from other neurons, the weights of which have to be determined by training. The sum of the weighted signals is then processed by an activation function, which specifies the output signal from given inputs according to a certain function. This function can be a discrete function such as the step function (1 if there is a signal, 0 if not), or a continuous function, which is a sigmoid ("S-shaped") function most of the time.

A neural network has a layered structure. The number of layers in a neural network is at least two: the input layer and the output layer. Linear functions can be learned using only these two layers. However, like Inverse Kinematics, most applications of neural networks are trying to solve nonlinear problems. One or more extra layers or *hidden layers* have to be added to the network for it to be able to solve these. They are called hidden layers because its inner workings are a black box process (i.e. it is not clearly interpretable what goes on). Choosing the right number of hidden layers and the number of neurons inside these layers is an important part of structuring a neural network and, among other things, depends on the number of input and output neurons, the complexity of the function, the amount of training data, the amount of noise in the training data, and so on.

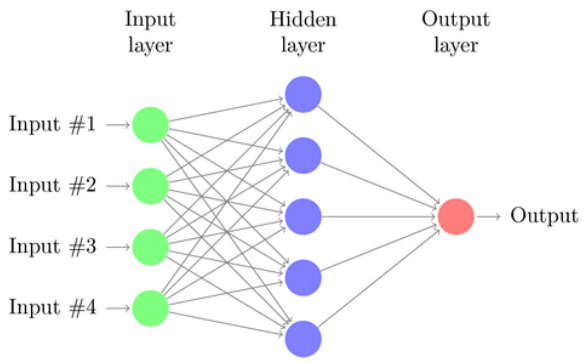

Figure 2: A simple feedforward neural network with one hidden layer.

Typically, neurons in the same layer are not connected to each other or themselves, and there are no cycles, so that data flows in one direction: from the input node to the output node. This type of network is called a *feedforward neural network*, and is pictured in Figure 2.

6.1 Training a neural network

As said earlier, neural networks learn from experience. A neural network is trained if the error between the network's output and the desired output is minimized. The weights of the neural network are the variables that have to be adjusted in order to change the output signal. This is done by a learning algorithm, the most widely used one being the *backpropagation algorithm*[12].

Backpropagation tries to find the minimum of the error function by following using *gradient descent*: it minimizes the function by taking steps towards the negative gradient of the error function. Therefore, in order to be able to use backpropagation, the error function has to be differentiable. This is achieved by choosing a continuous activation function, since the network function is a composition of these activation functions and is therefore by definition also continuous. This makes the error function with respect to the network weights continuous as well.

The advantage of backpropagation is that it is straightforward to understand and implement. However, convergence of the algorithm is not guaranteed and is typically very slow. It may also converge to a local minimum instead of a global minimum. To overcome these problems, an adaptive gradient search-based method has been developed. Resilient backpropagation[11] does only take into account the sign of the gradient and not the magnitude, and scales it by a weight-specific update value. This allows for faster convergence speed and provides more robustness against local minima (although they may still occur).

The design choices of the neural network used in this paper are the number of input and output nodes, the number of hidden layers and nodes, the activation function, and the learning algorithm used to train the neural networks. The network consists of three input nodes, one for each target coordinate, and four output nodes, one for each of the joint configurations that have to be adjusted. If the function you are trying to fit is linear, there is no need for a hidden layer at all. Since Inverse Kinematics is a non-linear optimization problem, this is not the case. In practice, one or two layers is sufficient to approximate any function, if there are enough hidden nodes in the layers. By trial and error, one layer has shown to be best suited for this problem, since adding an extra layer only increases training time and does not decrease the error (which is not odd, since the function that is being approximated is still fairly simple for a kinematic chain with only two joints). The number of hidden nodes in the layer is 40. This has been determined experimentally as described in Section 7. The training algorithm of choice is the RPROP algorithm as described in [11], since this proposed modification of the backpropagation algorithm has shown to be a great improvement in the training speed. The activation function has to be a continuous function, since RPROP relies on the fact that the error function has to be differentiable. Again, by trial and error, the function of choice is the sigmoid activation function.

7 Experiments

Several experiments have been performed on the traditional algorithms and their alternatives, the results of which will be listed in this section. All four algorithms are compared to each other regarding computation time. The Jacobian transpose, Jacobian pseudoinverse and FABRIK are compared to each other regarding number of iterations, and change in computation time when decreasing the error tolerance. The neural network experiments regarding the choice of the number of hidden nodes in the layer are listed as well.

Table 1 shows the experimental results of the four algorithms regarding computation time, the average number of iterations and the standard deviation of the number of iterations. The methods are evaluated using a dataset consisting of 10,000 uniformly distributed random samples from the set of reachable target positions. This set was generated by applying Forward Kinematics to all possible combinations of allowed joint angles. For the neural network, the dataset was enlarged to

10,000,000 samples, in order to minimize the overhead computation costs due to its low computation time. The results were found using an error tolerance of 0.1. There are no results for the number of iterations for the neural network, since the neural network is not an iterative method and computes the outputs instantly. The percentage of unsolved instances from the Jacobian pseudoinverse method is due to the unstable behaviour near singularities. If a joint configuration is close to a singularity, the pseudoinverse method will lead to large changes in joint angles, even for small movement of the target position[5].

Algorithm	**Time** (ms)	μ **iterations**	σ **iterations**
FABRIK	0.78	7.26	7.38
J. pseudoinverse	2.21	9.99	10.02
J. transpose	27.65	141.8	176.4
Neural network	0.002771	n/a	n/a

Table 1: Comparing the algorithms regarding computation time, the average number of iterations and its standard deviation.

Figure 3 shows the results of increasing the error tolerance for the three iterative algorithms. The Jacobian transpose method clearly performs worse for a lower error tolerance, whereas the Jacobian pseudoinverse shows only a slight increase in computation time and FABRIK remains constant. FABRIK was the only algorithm that yielded results when the error tolerance was set to zero, taking the same computation time as it needed for other error tolerances. Figure 3 does not include the neural network, since it is not an iterative method and the error is not a variable that can be set but a given number measured from the output.

Table 2 shows the error corresponding to the number of nodes used in the neural network. All networks were trained until the error rates did not decrease anymore. Again, the neural networks was were trained on a training set with 10,000 uniformly distributed random samples from the set of reachable target positions, and their corresponding joint configurations. The entries were normalized so that their values lie between 0 and 1. The trained networks were tested on a validation set of identical size to avoid overfitting. Since the error rate does not decrease any more when more than 40 nodes are added to the network, this is the number of nodes that was used. An error of 2.38% for a kinematic chain of length 21.87 centimeters corresponds to a maximum error of 0.52 centimeters at the end effector.

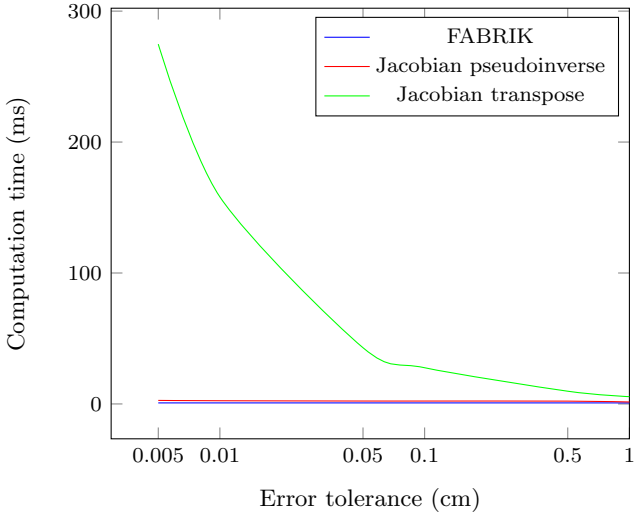

Figure 3: Comparing the change in computation time when decreasing the error tolerance.

# of hidden nodes	**Training error**	**Validation error**
5	2.70%	2.73%
10	2.70%	2.71%
15	2.37%	2.41%
20	2.37%	2.40%
30	2.37%	2.40%
40	2.35%	2.38%
50	2.35%	2.38%
75	2.35%	2.38%
100	2.35%	2.38%

Table 2: Experimentally determining the number of nodes in the hidden layer.

8 Conclusion

FABRIK outperforms the Jacobian transpose method as well as the Jacobian pseudoinverse method in terms of calculation time and number of iterations. Furthermore, it is the only algorithm that could always yield results with the error tolerance set to zero, and is well-behaved near singularities. The Jacobian pseudoinverse performs slightly slower than FABRIK, but is still decent. However, it does not always yield a solution near singularities, or when the error tolerance is set to zero. The Jacobian transpose method does not suffer from singularity problems, but its computation time explodes when the error tolerance is reduced.

If, however, computation time is of the highest priority, and precision is secondary, neural networks might be a more suitable approach. It greatly outperforms all the other methods regarding speed, but is the least accurate method of all, with a possible

error of 2.38%. This translates to 0.52 centimeters on the NAO, but when scaled to bigger applications, this might be a more significant number. Therefore, this neural network is suited for applications which require a fast response time and do not need to be that accurate.

It is safe to say that FABRIK is a better alternative to the Jacobian pseudo-inverse method and the Jacobian transpose method when applied to the NAO humanoid robot, since it performs better, is more accurate, and converges more quickly. Moreover, it does not suffer from singularity problems and is therefore a well-behaved approximation in each possible case. A general conclusion about FABRIK and neural networks, however, cannot be drawn from this paper. The two alternatives have not been compared to all existing techniques that are worth comparing. Also, the test domain only concerned the arm from the NAO humanoid robot, which is a kinematic chain consisting of only two joints. The performance of the techniques to bigger kinematic chains with more joints have to be evaluated, in order to fully compare these two alternatives to the traditional approaches.

8.1 Future research

There are some aspects that this paper did not address. For instance: a follow-up research comparing FABRIK to other traditional methods such as DLS/SDLS as described in Section 1.1, or techniques that were not discussed in this paper such as Cyclic Coordinate Descent (CCD)[13], would be necessary to determine if FABRIK really is the superior algorithm. Furthermore, FABRIK should be compared to kinematic chains with more than two joints to see if it scales as well as the traditional algorithms.

Regarding neural networks, architectures different from feedforward neural networks might be interesting to look into. For instance, Elman networks have shown to be good candidate architectures for solving Inverse Kinematics[8]. The same paper describes a different learning method, which uses a genetic algorithm instead of a gradient-based learning method, which seems to work well with Elman networks.

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] Buss, Samuel R. (2003). *3D Computer Graphics: A Mathematical Introduction with OpenGL*. Cambridge University Press.

[5] Buss, Samuel R. (2009). Introduction to inverse kinematics with jacobian transpose, pseudoinverse and damped least squares methods.

[6] (2013). Eigen. `http://eigen.tuxfamily.org/`.

[7] Fedor, Martin (2003). Application of inverse kinematics for skeleton manipulation in real-time. *Proceedings of the 19th spring conference on Computer graphics*.

[8] Kker, Rait (2011). A neuro-genetic approach to the inverse kinematics solution of robotic manipulators. *Scientific Research and Essays*.

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

[10] Research, Heaton (2013). Encog machine learning framework. `http://www.heatonresearch.com/encog`.

[11] Riedmiller, Martin and Braun, Heinrich (1993). A direct adaptive method for faster backpropagation learning: The rprop algorithm. *IEEE International Conference on Neural Networks*.

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

[13] Welman, Chris (1993). Inverse kinematics and geometric constraints for articulated figure manipulation. M.Sc. thesis, Simon Fraser University.

[14] Wolovich, W. and Elliott, H. (2009). A computational technique for inverse kinematics. *IEEE Conference on Decision and Control*.

Appendices

Hardware

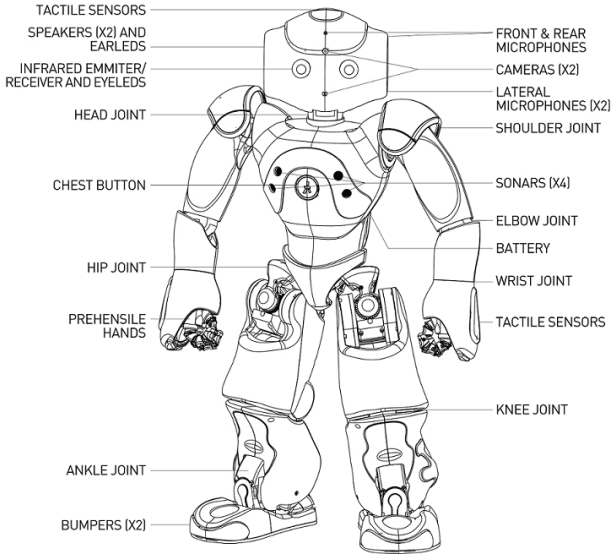

Figure 4: NAO joints and sensors.

All of the joints in Figure 4 can move independently of each other. The technical specifications of the NAO are listed in Table 3.

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 3: Technical Specifications.

The arm of the NAO is a kinematic chain consisting of three joints and two links. The length of the arm is 21.87 centimeters. Table 4 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 4: List of joints and their rotation axes in the arm of the NAO.