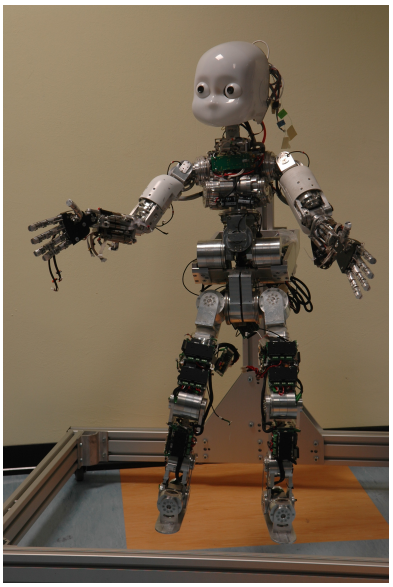
A kinematic model for the iCub

Julia Jesse

Semester project



Biologically inspired robotics group (BIRG)
Department of computer science - EPFL

January 10, 2009

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

BIOLOGICALLY INSPIRED
ROBOTICS GROUP (BIRG)

Acknowledgments

First I would like to thanks Sarah Degallier for her help and support through all this project. I would like also to thanks Ruben Smits from the Katholieke Universiteit Leuven for his help on KDL and for having given me the great opportunity to work with him on KDL.

Contents

Introduction

One of the research fields roboticists are working on nowadays is humanoid robotic. They try to develop humanoid robots in order to help old people alone at home or to understand some human behaviors like autism for example. Among all these different projects, there is the european RobotCub project which aims to study cognition through a robot looking like a two years old child and called iCub.

The goal of this semester project is to develop a kinematic model of the iCub that could later be used for studying gait stability for instance. In order to achieve this, we need several things. First we need to have a general model of the robot. In chapter 1, we will explore some existing modelings and focus on the Denavit-Hartenberg parameters. As the modeling of a robot and the study of its motion can become complicated, we will help us with KDL, an open source Kinematic and Dynamic Library. In chapter 2 we will have an overview of what KDL is and what it can do. In order to make a robot move, we need to describe its motion with kinematics theory. In chapter 3 and 4 we will present what is called *forward position kinematics* and *inverse position kinematics* and how KDL handles them. In chapter 5 we will see how to model the iCub with KDL, Matlab and Webots. Finally in chapter 6, we will analyze some results found on controling the Webots model using KDL.

Chapter 1

Modeling of a rigid body

A rigid body system, like a robot, is composed of rigid bodies, joints and various force elements. In biology a joint represents a connection between two bodies. For example the elbow is a joint that connects the arm with the forearm and the shoulder a joint that connects the trunk to the arm. However, when we use a model, a joint represents one degree of freedom (DoF), that is one direction in which a body can move. In our previous example, the elbow has one degree of freedom, whereas the shoulder has three degrees of freedom (moving the arm laterally, along the body or rotating the shoulder). To describe a rigid body system, we need to define what are the different bodies and how they are connected together. These informationsjust are represented with what we call the connectivity model.

1.1 The connectivity model

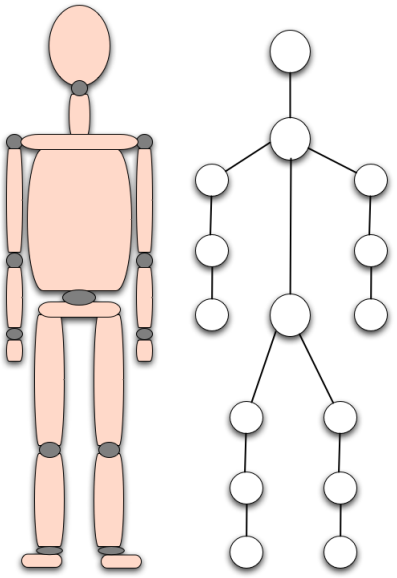
In this model we use a *connectivity graph*, where the nodes represent bodies and the arcs joints. Besides, we need to define one node as being the *base* node, which describes a fixed body. The graph has to be undirected and connected (i.e. there is a path between every two nodes).

Figure 1.1 shows an example of the connectivity model. On the left we see a humanoid body, where the joints are represented by grey dots. On the right, this body is represented with a connectivity graph. In this example, the graph has the structure of a tree. Thus, we call it a *kinematic tree*. If for example we take just the description of the arm, we call it a *kinematic chain*, i.e. each node has only one child in the graph.

As the iCub may move in space, we have to know its position with respect to a given referential. In order to know this position, we introduce a 6 DoF joint : three degrees for the position with respect to the x, y, z axis and three degrees which will describe the orientation of the robot. As a connectivity graph needs to be connected and to have a fixed base, we will design the ground as the fixed base and attach the 6 DoF joint to the robot. If we attach the 6 DoF to the middle of the torso, the orientation information tells us if the robot is bending in one direction or another. A body that is connected to a fixed base through a 6 DoF joint is called *floating base*. An example is shown on figure 1.2, where the red dot represents the fixed base, the gray joint the floating base and the blue line, the 6 DoF joint which links the floating base to the fixed referential.

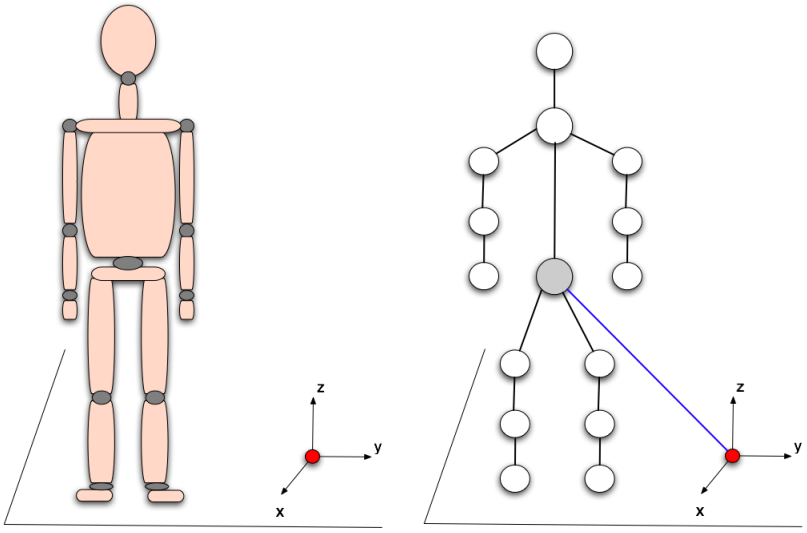
1.1.1 The geometric model

The part of the connectivity model which is called geometric model consists of the following. When we link to bodies together by a joint, their movement is constrained. Indeed, if we



(a) Simplified humanoid body (b) Connectivity graph

Figure 1.1: Connectivity model



(a) Simplified humanoid body with referential (b) Connectivity graph with 6 DoF joint

Figure 1.2: Floating base

take again the example of the elbow which links the arm to the forearm, we can see that the forearm can not move in certain directions. To study this constraint, we need to know the motion allowed by the joint, i.e. for example the angle values that a revolute joint can take. We also need to know the relative position of each joint. In order to do this, we attach to each body a local coordinate system which we call *frame*. This frame defines what we call *body coordinates* or *link coordinates*. To go from frame F_i to frame F_j, we use a transformation matrix $A_{i,j}$. This matrix represents the translation and rotation of frame F_j with respect to frame F_i. Figure 1.3 illustrates the geometric model. We start at the base frame F_0. When we want to go to frame F_1, we will use the transformation matrix $A_{0,1}$, which describes how we have to translate and rotate the base frame to get to the new frame.

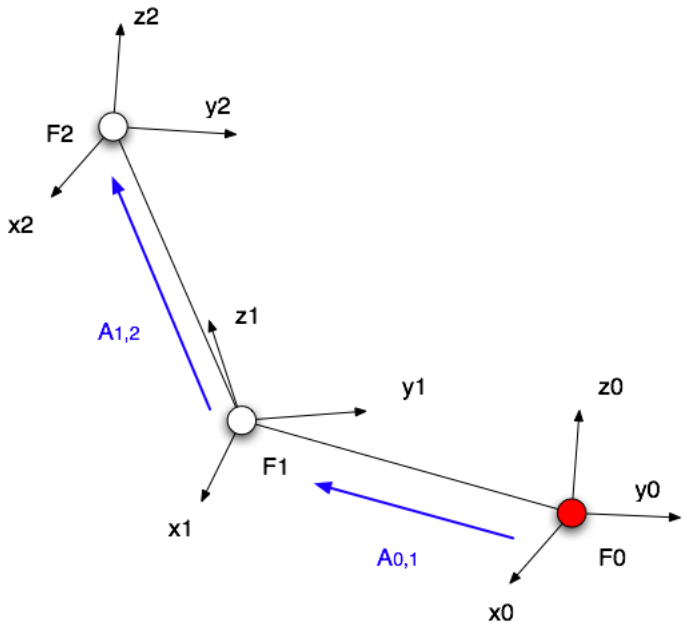


Figure 1.3: Geometric model of a rigid-body system

1.2 The Denavit-Hartenberg parameters

As seen previously, to go from one coordinate system to another, we need six parameters to describe the roto-translation. Three parameters represents the x,y, and z parameters for the translation, and three others represent the rotation around each axis. In 1955, Denavit and Hartenberg developed a method which needs only four parameters. The idea is based on the fact that most common joints can be located by a line in space, and that we only need four parameters to locate a line. Let us first see what these parameters are and how they are used to describe each frame position.

The four Denavit-Hartenberg (DH) parameters associated to each joint i are :

- *the joint angle* θ_i
- *the link/body length* a_i
- *the link/body offset* d_i
- *the link/body twist* α_i

Usually for revolute joints a_i, α_i and d_i are constants and θ_i is the only variable. Before describing in more detail what these parameters represent, let us just have a look at their main features.

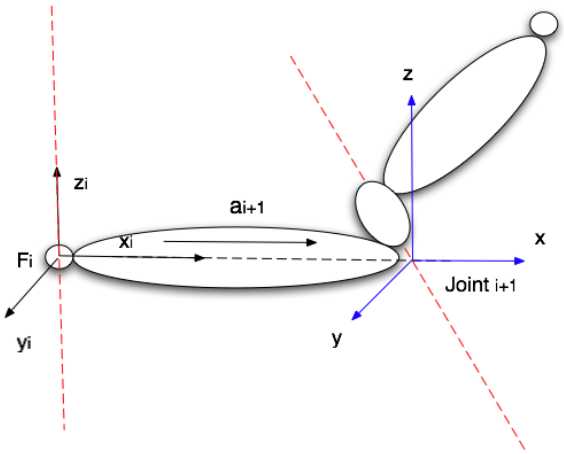
First we need a base coordinate frame, n joint axes, where n represents the number of DoF, and an end-effector frame embedded in the last body. After this, we can place the DH coordinate frames according to the following rules.

- Axis z_0 is aligned with the z axis of the base frame and axis z_{n+1} is aligned with the z axis of the end-effector frame.
- Each axis z_i is aligned with the joint axis i. In the case of a revolute joint, the axis z_i is aligned with the rotation axis.

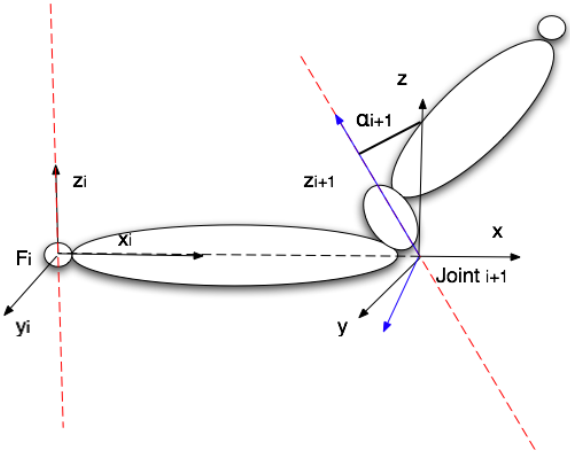
- Axis x_i intersects z_i and z_{i+1} and is perpendicular to them.
- Axis y_i is derived from x_i and z_i such that the coordinate system is right-handded.

Let us now have a look on how we place these parameters. Suppose we want to go from frame F_i of joint i to frame F_{i+1} of joint *i+1*. As shown on Figure 1.4(a), the axis x_i is parallel to the body *i+1*. Notice that in this example, we only have revolute joints and that the rotation axis is shown with a red dashed line. The first parameter we use to get the new frame is a_i. As its name suggests it, this parameter represents the length of the body. So we will translate the frame F_i along the x_i axis, with a distance of a_{i+1}. After this, we will rotate (twist) the z axis around the x axis with an angle α_{i+1} to get the new z axis z_{i+1} (Figure 1.4(b)).

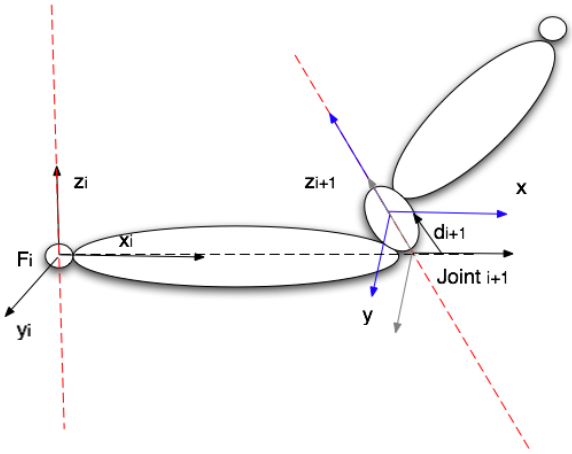
The next parameter we will use is d, the body offset, i.e. how far along the z_{i+1} axis is the new referential placed (Figure 1.4(c)). Finally, we rotate the x axis around z_{i+1} with an angle θ_{i+1} to get the new x_{i+1} axis, which will be aligned with body *i+2*. As z_{i+1} is aligned with the joint axis, θ_{i+1} represents the joint angle (Figure 1.4(d)). Finally, we place the new y_{i+1} axis such that the new coordinate system is right handed.



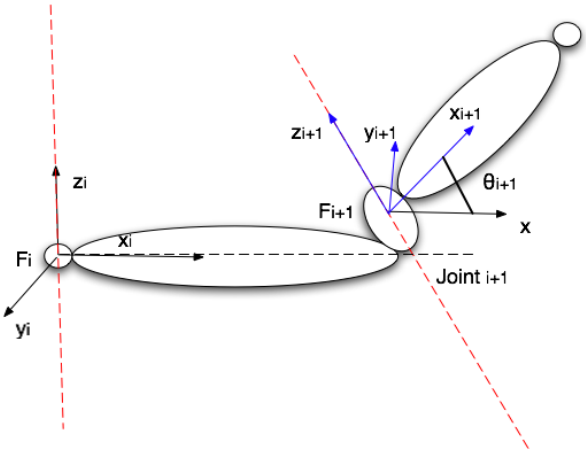
(a) Body length a_{i+1}



(b) Body twist α_{i+1}



(c) Body offset d_{i+1}



(d) Joint angle θ_{i+1}

Figure 1.4: Denavit-Hartenberg's parameters

Chapter 2

KDL : a Kinematic and Dynamic Library

The acronym KDL stands for Kinematic and Dynamic Library. It is an open source library written in C++ which is part of the Open Robot Control Project (Orocos). According to [2] the Orocos project aims to develop a free-software and modular framework for robot and machine control. One of its projects is the Kinematic and Dynamic Library (KDL) which allows to model and calculate kinematic chains such as robots. It provides different objects : *geometric primitives* like rotations or vectors, *kinematic chains* which allow us to describe a serial chain of bodies connected by joints, *kinematic trees* which describes bodies which are placed in a tree structure, and *kinematic solvers* which provides several algorithms to calculate forward or inverse kinematics.

The forward kinematics is used to find the position of the end-effector, knowing the joint values. The solution is always unique. On the contrary, the inverse position kinematics calculates the joint angles for a given end-effector position. As for one position, the joint positions can take different values, the inverse kinematics has not a unique solution (Figure 2.1). A system which has several solutions is said to be *redundant*. This is the case for systems which have more than six DoFs, i.e. systems which have more DoF than constraints. Because of this multiplicity of solutions, the inverse kinematics is difficult to solve, especially for a robot as complex as the iCub, which has a lot of degree of freedom. It is why we will use KDL to help us with that. In the two next chapter we will see more in detail what is the forward and inverse kinematics, and how KDL implements it.

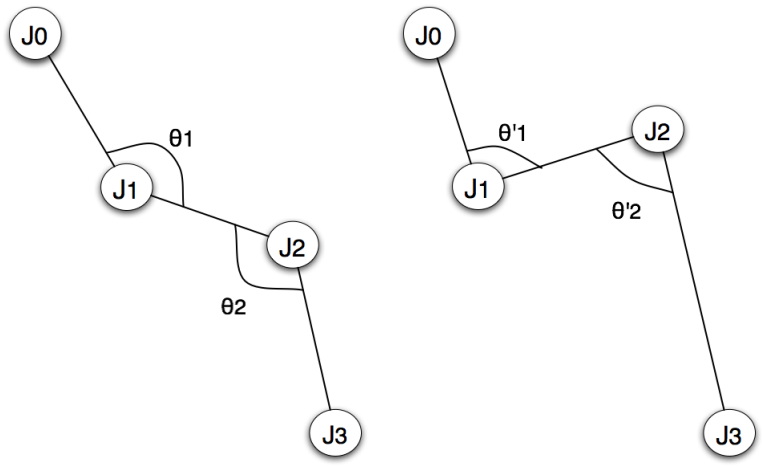


Figure 2.1: Inverse kinematic : two different solutions with the same end-effector positions

Chapter 3

Forward position kinematics

The aim of the forward position kinematics is to determine the position of the end-effector, knowing the joint angles. For each given joint angles, there is only a unique solution. In this chapter we will see how to calculate the forward kinematics for a chain and for a tree. Furthermore, we will also have a look at how KDL implements the forward position kinematics.

3.1 Forward position kinematics for a chain

Remember that in chapter 1 we introduced the notion of local coordinate system called *frame*, and that to go from one frame to another, we use a transformation matrix A_i (Figure 1.3). This matrix can be represented with the *homogeneous transformation matrix* :

$$H_i = \begin{bmatrix} R_i & O_i \\ 0 & 1 \end{bmatrix} \tag{3.1}$$

where R_i is a 3x3 rotation matrix representing the orientation of the frame, and O_i is a vector of size three representing its position.

When we use the Denavit-Hartenberg representation also defined in chapter 1, the homogeneous transformation H_i is the product of four basic transformations : a rotation of θ about z, a translation of d along z, a translation of a along x and a rotation of α about x (where θ, d, a and α are the D-H parameters).

$$H_i = R_{z,\theta_i} Trans_{z,d_i} Trans_{x,a_i} R_{x,\alpha_i} = \begin{bmatrix} \cos\theta_i & -\sin\theta_i \cos\alpha_i & \sin\theta_i \sin\alpha_i & a_i \cos\theta_i \\ \sin\theta_i & \cos\theta_i \cos\alpha_i & -\cos\theta_i \sin\alpha_i & a_i \sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.2}$$

where

$$R_{z,\theta_i} = \begin{bmatrix} \cos\theta_i & -\sin\theta_i & 0 & 0 \\ \sin\theta_i & \cos\theta_i & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad Trans_{z,d_i} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.3}$$

$$Trans_{x,\alpha_i} = \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad R_{x,\alpha_i} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha_i & -\sin\alpha_i & 0 \\ 0 & \sin\alpha_i & \cos\alpha_i & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.4}$$

Now, when we want to go from frame O_i to frame O_j, we can define O_j with respect to O_i by using the transformation matrix H_i^j.

$$H_i^j = H_{i+1}H_{i+2}...H_{j-1}Hj \ \ if \ i < j \tag{3.5}$$

When we want to know the position of the end-effector with respect to the base frame we just need to calculate H_0^n. As we can see, this method is really simple and can be implemented in Matlab juste by defining the matrices H_i and by multiplying them together. However, let us notice that the relation between joint positions θ_i and the end-effector is not linear as the transformation matrices contain sines and cosines functions. Now that we have defined what the forward position kinematics is, let us have a look at how KDL implements it.

3.1.1 KDL's forward position kinematics for a chain

First of all, we need to define a kinematic chain in KDL. This is done with the class *KDL::Chain*. A chain is made of segments and each segment is defined by a joint, describing its degree of freedom, and a coordinate frame, which can also be expressed with the Denavit-Hartenberg parameters. In the following example, we build a chain *my_chain* and add a segment to it. The segment contains a joint which can rotate around the z axis, and a frame which is defined with the DH parameters.

```
KDL::Chain my_chain;
my_chain.addSegment(KDL::Segment(KDL::Joint(KDL::Joint::RotZ),
                    KDL::Frame().DH(0.0320,  M_PI/2.0,  0.0,  0.0);
```

Now that we have defined our chain, we can compute the forward kinematics. KDL provides a recursive algorithm represented by the class *KDL::ChainFkSolverPos_recursive*. Given an array of the values for each joint of the chain and a frame in which the result will be expressed, the algorithm works as follows. We start at the root of the chain. For each segment, we get its frame, which is located at the tip of the segment, and multiply it by the frame we got previously. Note that this frame is sometimes called *pose* in the API of KDL. The algorithm expresses the same idea as explained in section 3.1, and is illustrated in fig 3.1

3.2 Forward position kinematics for a tree

As a humanoid robot like the iCub has a tree structure, we will need to define a tree if we want to study kinematics for the whole body. One possibility to compute the forward kinematics is to select a chain in the tree and then to use the same technique as explained in 3.1.

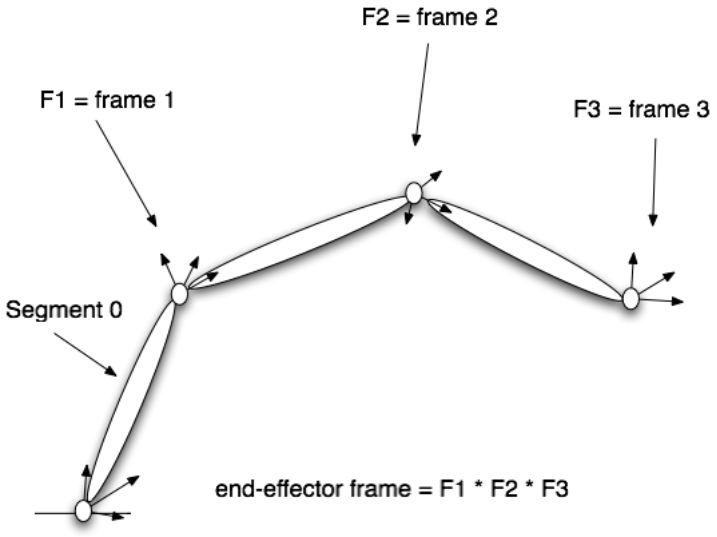


Figure 3.1: Finding the end-effector frame with the forward kinematics.

3.2.1 KDL's forward position kinematics for a tree

Again we need first to define what is a tree in KDL. The library provides a class *KDL::Tree* to describe a kinematic tree. It is made out of segments. When we first create a *KDL::Tree* it is empty, and by default contains only one segment with the name "root". We then build our tree by adding segments or chains to a given segment. In the following example, we first create an empty tree *tree* and a segment *base_segment* containing a joint which does not move (*KDL::Joint::None*), and a frame defined with the DH parameters. We then add our segment to the default "root" segment of the tree. Finally we add the chain "torso" to the "base_segment".

```
KDL::Tree tree = Tree();
KDL::Segment base_segment = KDL::Segment(KDL::Joint(KDL::Joint::None),
                                         KDL::Frame().DH(0.0, 0.0, 0.0, 0.0));

tree.addSegment(base_segment, "base_segment", "root");
tree.addChain(torso.get_chain(), "torso", "base_segment");
```

Now that we have defined our tree, we can compute the forward kinematics. At the time I started the project, the code for trees was quite experimental and there were no kinematic solvers for them. I had the opportunity to go to Leuven in Belgium and to develop some parts of the solvers with the help of Ruben Smits, one of the main developer for this part of KDL. We extended the code with a recursive forward position kinematics solver for trees. The algorithm, which is implemented in the class *KDL::TreeFkSolverPos_recursive*, takes as input an array which contains all the joint positions of the tree, a frame in which the result will be expressed and the name of the end-effector for which we want to compute the solution. To have a better understanding on how the algorithm works, let us look at the example of Figure 3.2. The algorithm starts by getting the frame of the end-effector, i.e $F_6(q_6)$. Note that all the frames depend on the joint positions which are given as input. As this frame is not the root frame $F_0(q_0)$, we get the parent of the current frame, i.e. $F_4(q_4)$ and return the recursive call of the algorithm with the parent as argument multiplied by the current frame. At the first step we will have something like

$$\textit{End-effector} = \textit{RecursiveForwardKinematicAlgorithm}(F_4(q_4)) \ * \ F_6(q_6).$$

At the second step we restart the same procedure and get

$$\textit{End-effector} = \textit{RecursiveForwardKinematicAlgorithm}(F_3(q_3)) \ * \ F_4(q_4).$$

We repeat this recursive procedure until we reach the root frame. At the end of the recursion, we will have

$$\textit{End-effector} = F_0(q_0) * F_1(q_1) * F_2(q_2) * F_3(q_3) * F_4(q_4) * F_6(q_6).$$

As we can see on Figure 3.2, this algorithm works similarly to the forward kinematics algorithm for a *KDL::Chain*. Here we start getting the information from the end-effector, because of the way KDL represents a tree.

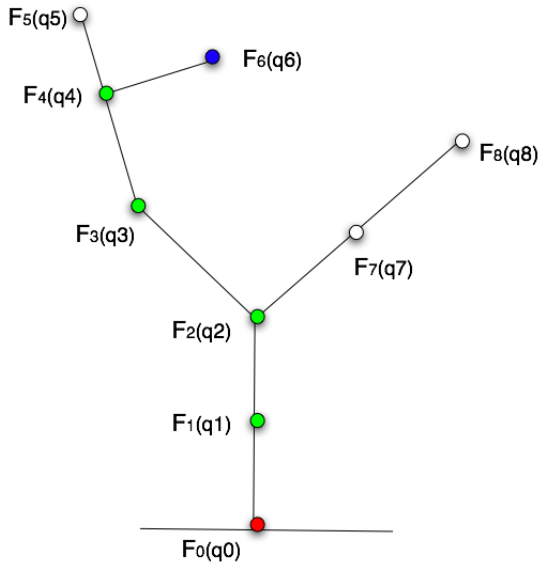


Figure 3.2: Finding the end-effector frame with KDL's tree forward kinematics.

Chapter 4

Inverse position kinematics

The aim of the inverse position kinematics is to determine the joint values knowing the end-effector position. As seen in chapter 2, the solution for the joint values is not unique, and it depends on the choice of the algorithm. In this chapter we will see how KDL calculates the inverse kinematics for a chain and for a tree [1].

4.1 KDL's inverse position kinematics for a chain

To calculate the inverse kinematics, KDL uses an algorithm based on what is called the Newton-Raphson iterations. This algorithm is provided by the classes *KDL::ChainIkSolverPos_NR* and *KDL::ChainIkSolverPos_NR_JL*, where NR stands for Newton-Raphson and JL for Joint Limits. As we are interested to take the joint limits into account, we will use the second class. As the algorithm is based on inverse velocity kinematics, we will first present what it is and then discuss the inverse position kinematics.

4.1.1 Inverse velocity kinematics for a chain

KDL provides three algorithms. We will focus on the *ChainIkSolverVel_wdls* algorithms, where "wdls" stands for "weighted damped least square". Before we discuss how the algorithm works, let us introduce the notions of Jacobian, pseudo-inverse and Singular Value Decomposition (SVD).

Jacobian

When the end-effector moves, it has a certain velocity. This velocity, which is also called *twist* $\overrightarrow{T}$, has a translational and rotational component. We can express $\overrightarrow{T}$ with the following equation :

$$\overrightarrow{T} = \frac{d\overrightarrow{x}}{dt} = \frac{\partial A(\overrightarrow{q})}{\partial \overrightarrow{q}} \frac{d\overrightarrow{q}}{dt} \tag{4.1}$$

where $A(\overrightarrow{q}) = \overrightarrow{x}$ is the forward kinematics equation. The term $\frac{\partial A(\overrightarrow{q})}{\partial \overrightarrow{q}}$ is called the *Jacobian* $J(\overrightarrow{q})$.

[1] For more references on inverse kinematics :
"A theory of generalized inverses applied to robotics", by Doty et al
"Manipulator Inverse Kinematic Solutions Based on Vector Formulations and Damped Least-Squares Methods", by Wampler

So we can rewrite the equation as :

$$\overrightarrow{T} = \overrightarrow{\dot{x}} = J(\overrightarrow{q})\,\overrightarrow{\dot{q}} \tag{4.2}$$

From this equation we can see that the Jacobian gives us a relation between the joint velocities and the Cartesian velocity of the end-effector. The relation 4.2 is linear in $\overrightarrow{\dot{q}}$. It means that, in this case, if given some joint velocities, we double the speed of the joints, the end-effector's velocity will double too. This linearity property make the inversion much more simpler, and it is why we use the inverse velocity kinematics to solve the inverse position.

The Singular Value Decomposition (SVD)

In the problem of inverse velocity kinematics, we are interested in finding the joint velocities. If the Jacobian J is an invertible $n \times n$ matrix, we can rewrite equation 4.2 as :

$$\overrightarrow{\dot{q}} = J^{-1}(\overrightarrow{q})\,\overrightarrow{T} \tag{4.3}$$

But if we have more degrees of freedom than constraints (i.e. 3 constraints for position and 3 for rotation), then the Jacobian is not a square matrix and we can not invert it. As we will see it in chapter 5, this is the case for the iCub. For example, it has 7 degrees of freedom for each arm, and even 10 if we consider the chain from the torso to the arm. So in this case we will use the technique of the Singular Value Decomposition (SVD) to find the pseudo-inverse in order to calculate the joint velocities with respect to the end-effector's velocity.

The idea of the SVD is the following. Let us take a matrix $\mathbf{M} \in \Re^{n\times m}$. Recall from the linear algebra that σ is a singular value[2] of M if and only if there exists a vector $\overrightarrow{u} \in \Re^m$ and a vector $\overrightarrow{v} \in \Re^n$ such that $\mathbf{M}\overrightarrow{u} = \sigma\overrightarrow{v}$ and $\mathbf{M}^t\overrightarrow{v} = \sigma\overrightarrow{u}$. If $\mathbf{M}$ has singular values $\sigma_1 \cdots \sigma_n$, then we can decompose it in

$$M = U\Sigma V^T \tag{4.4}$$

where $\mathbf{U} \in \Re^{n\times n}$, $\mathbf{V} \in \Re^{m\times m}$ and $\Sigma \in \Re^{n\times m}$, with $\mathbf{U}$ and $\mathbf{V}$ such that Σ has the form

$$\Sigma = \begin{bmatrix} \sigma_1 & 0 & 0 & \cdots & 0 & \cdots & 0 \\ 0 & \sigma_2 & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & \sigma_n & 0 & \cdots & 0 \end{bmatrix} \tag{4.5}$$

The pseudo-inverse M^* of M is given by the following equation :

$$M^* = V\Sigma^* U^T \tag{4.6}$$

[2] If $\mathbf{M} \in \Re^{n\times n}$, σ is called eigenvalue

where Σ^* is the pseudo-inverse of Σ and has the form

$$\Sigma^* = \begin{bmatrix} \frac{1}{\sigma_1} & 0 & \cdots & 0 \\ 0 & \frac{1}{\sigma_2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{1}{\sigma_n} \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \cdots & 0 \\ 0 & 0 & \cdots & 0 \end{bmatrix} \tag{4.7}$$

As we can see with equation 4.5 and 4.7 it is quite easy to find the pseudo-inverse for Σ : we just need to inverse each σ_i. Resolving the inverse velocity consists now in solving

$$\dot{\overrightarrow{q}} = J^*(\overrightarrow{q})\overrightarrow{T} \tag{4.8}$$

where $J^*(\overrightarrow{q})$ is the pseudo-inverse of the Jacobian.

However, there is still an issue with this resolution. If σ_i is equal to 0, then $\frac{1}{\sigma_i}$ goes to infinity and thus we can not define the pseudo inverse. In robotics, having σ_i equal to 0 means that we can not move in a given direction anymore. Indeed, if $\sigma_i = 0$, we can say from equation 4.2 that $\dot{\overrightarrow{x}} = 0$ for each $\dot{\overrightarrow{q}}$.

To avoid this problem, we introduce a parameter called λ and replace the $\frac{1}{\sigma_i}$ terms seen before by $\frac{\sigma_i}{\sigma_i^2+\lambda}$. This solution is used in the "damped least square" method, where λ is also called *damping parameter*. It works for $\lambda << \sigma$.

In order to properly choose the damping parameter, we need to take into account two things. First, if λ increases, then the error for the approximation of the pseudo-inverse will also increase. Secondly, if λ decreases, then the damping will also decrease, and thus we may not avoid singular configurations.

The KDL::ChainIkSolverVel_wdls algorithm

The class *KDL::ChainIkSoverVel_wdls*, where "wdls" means "weighted damped least square" computes inverse velocity kinematics for a given chain. Let us have a look how it works and where the name "wdls" comes from.

First, if we want to specify that a given joint should not move, we use a method which works as if we put an infinite "weight" to these joints. This can be useful, for example, if we do not want to use the joints of the torso in the computation of the inverse kinematics for an arm. We can see from equation 4.3 that forbidding a joint to move means to set the corresponding value of the Jacobian to 0. To achieve this, KDL provides two matrices called the *joint space weighting matrix* and the *task space weighting matrix*, where task space is the same as the cartesian space. These matrices have to be symmetric and their default value is the identity matrix. One possibility is to take a diagonal matrix. Let us take the joint space matrix for example. If we do not want joint i to move, we set the diagonal value of row i to 0. On the other hand, we can also enable a joint to move more that others by increasing its corresponding diagonal value. Similarly setting a 0 on the task space weighting matrix means that we will not take into account the corresponding coordinate for the movement.

The algorithm begins by calculating the Jacobian **J** for a given chain according to some joint values. It then calculates the weighted Jacobian **WJ** by multiplying the joint space

weighting matrix **JS** by the Jacobian, and by multiplying this result by the task space weighting matrix **TS**, i.e.

$$\mathbf{WJ} = \mathbf{TS} * \mathbf{J} * \mathbf{JS} \tag{4.9}$$

It then calculates the SVD for this weighted Jacobian, i.e. it computes the matrices **U**, Σ and **V** we have defined before. Thanks to these matrices, the algorithm finally computes the joint velocities. The term "least square" comes from the fact that the algorithm tries at each step to minimize the joint velocities such that it gets the closest solution. Indeed, if we tend to go into a singular configuration, the algorithm avoids to get there by setting the velocity which brings us there as small as possible.

The inverse position algorithm

Now that we have introduced what the inverse velocity is, let us see how the inverse position kinematics algorithm works. The idea is the following: we start with an estimate $\overrightarrow{q_{est}} = (q_{1,est} ... q_{2,est})^T$ of the joint angles and compute the forward kinematics to get the frame of the end-effector, using the forward kinematic solver *KDL::ChainFkSolverPos_recursive* introduced in 3.1.1. Let us call this frame $\mathbf{F}(\overrightarrow{q_{est}})$ (Figure 4.1). We then want to know the velocity to go from the estimated end-effector frame to the real end-effector frame $\mathbf{F}(\overrightarrow{q})$, i.e we want to know $\Delta Twist$. If this velocity is zero, we have reached the solution. If not, we calculate the inverse joint velocity with the algorithm provided by *ChainIkSolverVel_wdls*, which was explained previously. We then add this joint velocity to the estimate $\overrightarrow{q_{est}}$ to get a new estimate $\overrightarrow{q'_{est}}$. We finally verify that the new estimate respects the joint limits and we restart again the procedure, until we find a solution or until we have reached the maximum number of specified iterations.

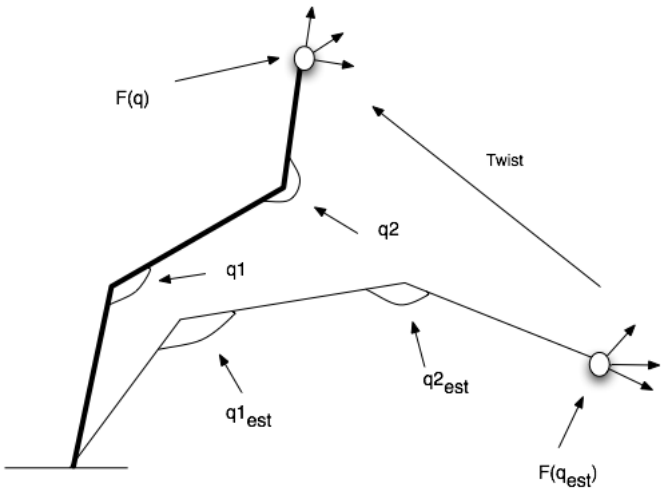


Figure 4.1: Illustration of the inverse position algorithm

4.2 KDL's inverse position kinematics for a tree

At the time I began this project, there were no inverse position kinematics solvers for a tree. Ruben Smits developed them when I was in Leuven. The class *KDL::TreeIkSolverPos_NR_JL*,

where again NR stands for Newton-Raphson and JL for Joint Limits, implements an inverse position kinematics algorithm. It works as follows. We are given a list of end-effectors we are interested in. For each end-frame of this list, we get the corresponding twist and calculate the joint position values in the same manner as with the *Chain::IkSolverPos_NR_JL* algorithm which was explained in 4.1. The only thing which changes is that the algorithm for the forward position kinematic is now implemented by the *KDL::TreeFkSolverPos_recursive* and the inverse velocity by the class *KDL::TreeIkSolverVel_wdls*. The latter works the same way as the *KDL::ChainIkSolverVel_wdls* algorithm, except that in this case we need to calculate at the same time the inverse position for all the end-effectors we are interested in. Therefore, we gather all the Jacobians, each corresponding to one end-effector, into one "global" Jacobian.

Chapter 5

Modeling the iCub

The european RobotCub project aims to study cognition through a humanoid robot. This robot is called iCub and looks like a 2 years old child, as it is 94 cm tall. It has 53 degree of freedom and its head and eyes are completly articulated. Besides this it has several sensory capabilities: it can see, hear, and feel with its fingers. It even has a vestibular capability, which alows it to have a sense of balance and distinguish what is up and what is down. Thanks to all these features, the iCub will be able to crawl, sit up, and manipulate many things with dexterity [8].

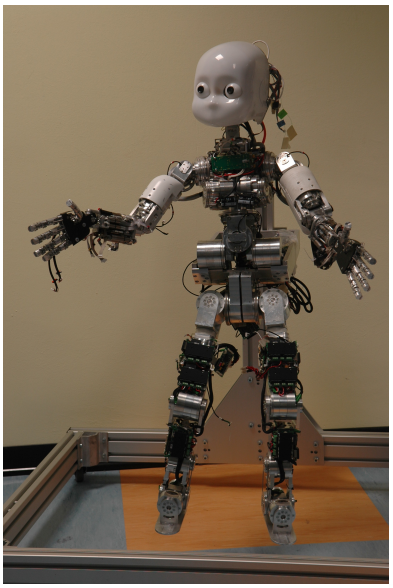
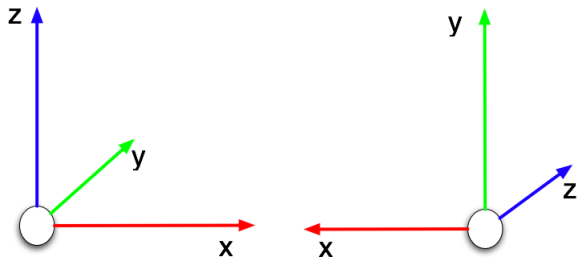


Figure 5.1: The iCub robot [http://www.robotcub.org]

In this chapter we will see how to model the iCub with Matlab, KDL and how to use this model under Webots, a commercial mobile robot simulation software developed by Cyberbotics Ltd [6].

5.1 Modeling the iCub with KDL

To model the iCub with KDL, I created a *KDL::Chain* for each limb (the right and left arm, the right and left leg and the torso) using the Denavit-Hartenberg coordinates defined on the wiki for the iCub [9] (see Appendix A). The values are from the 8th of October 2008, and at that time the coordinate system was as shown on Figure 5.2(a):



(a) Wiki (b) Webots

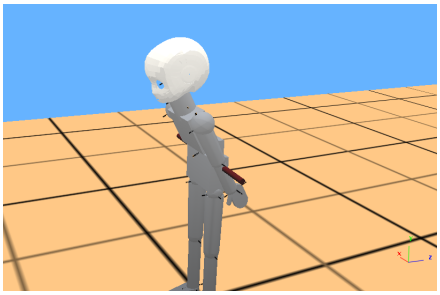
Figure 5.2: Coordinate systems

One has to be careful with theses DH coordinates for two reasons. First we have to take into account the roto-translation to align the root frame with the frame of the first joint of the chain. As we will use KDL together with Webots, we also need to make the coordinate systems of KDL and Webots consistent. Therefore I added a segment containing a joint *KDL::Joint::None* which cannot move and a frame which aligns the root frame with the frame of the first joint of the chain and which turns the whole coordinate system into the coordinate system for Webots. Note that all values are either in meters or in radians, again for compatibility purpose between KDL and webots. Secondly, we need to pay attention on how the arms are build. From the values in Appendix A we can see that joint 2 (i.e. the torso yaw) is not defined with the same values for the right arm and for the left arm. This is due to the fact that, starting from the torso yaw, we do not get the same position for the first joint of the right arm and for the first joint of the left arm. Therefore, the yaw joint in the torso will be defined with a joint which can rotate around z, but which contains no frame. This frame is added at the beginning of the arm chains with the corresponding frame and with a fixed joint as it is shown in the following example.

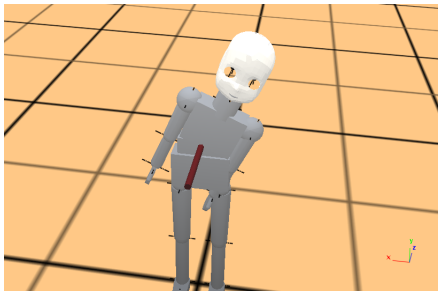
```
KDL::Chain torso_chain;

torso_chain.addSegment(KDL::Segment(KDL::Joint(KDL::Joint::None),
   KDL::Frame(Rotation::Rotation(0,0,-1, 1,0,0, 0,-1,0))));
torso_chain.addSegment(KDL::Segment(KDL::Joint(KDL::Joint::RotZ),
   KDL::Frame().DH( 0.0320   ,  M_PI/2.0,  0.0   ,  0.0    )));
torso_chain.addSegment(KDL::Segment(KDL::Joint(KDL::Joint::RotZ),
   KDL::Frame().DH( 0.0    ,  M_PI/2.0,  0.0   , -M_PI/2.0)));
torso_chain.addSegment(KDL::Segment(KDL::Joint(KDL::Joint::RotZ)));


KDL::Chain right_arm_chain;

right_arm_chain.addSegment(KDL::Segment(KDL::Joint(KDL::Joint::None),
   KDL::Frame().DH(-0.0233647,  M_PI/2.0, -0.1433 , -(105.0/180.0)*M_PI)));
right_arm_chain.addSegment(KDL::Segment(KDL::Joint(KDL::Joint::RotZ),
   KDL::Frame().DH( 0.0, M_PI/2.0, -0.10774, -M_PI/2.0)));
right_arm_chain.addSegment(KDL::Segment(KDL::Joint(KDL::Joint::RotZ),
   KDL::Frame().DH( 0.0, -M_PI/2.0,  0.0   , -M_PI/2.0)));
...
```

Once we have defined all the chains we need, we can build a *KDL::Tree* by adding them on the right place. We have now five different chains : one fore the torso, one for each

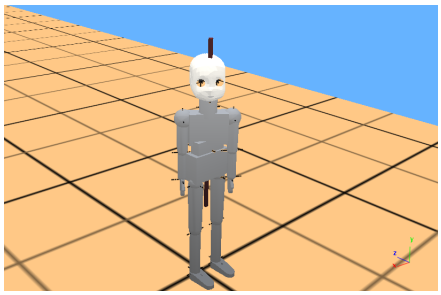
arm and one for each leg. We first create a *KDL::Tree* and a fixed base segment. Then we attach this base segment to the "root" of the tree as explained in 3.2.1. We then attach the torso pitch and the legs to the base segment and the arms to the torso yaw. To make the intuition of what is called pitch, yaw and roll more clear, think of the following. If somebody is bending forward or backwards he makes a rotation around the pitch angle (Figure 5.3(a)). If this same person is bending laterally it makes a rotation around the roll angle (Figure 5.3(b)). Finally if it is like "twisting" his body, it makes a rotation about the yaw angle (Figure 5.3(c)).



(a) Pitch angle



(b) Roll angle



(c) Yaw angle

Figure 5.3: Pitch, roll and yaw angles

5.2 Modeling the iCub with Matlab

On the wiki for the iCub [9], there are several Matlab programs, one for each limb, calculating the forward kinematics with matrix multiplication as explained in 3.1. They also provide a way to draw the iCub. To draw the entire robot, I gather some of these Matlab programs together to get the shape shown on Figure 5.4.

Besides this, the graphical tool for Matlab enables us to get the coordinates for the points we are clicking on with the mouse. I used this feature to get the coordinates for the joints, and thus derive the translation from one joint to the next joint in a given chain. This translation information is used to draw the model under Webots as I will explain in the next section. Notice that in the Matlab model units are in millimeters and radians, while in Webots there are in meters and radians. Besides, the Matlab model has the same coordinate system as in Figure 5.2(a) and Webots has the same coordinate system as in Figure 5.2(b).

5.3 Modeling the iCub with Webots

As seen before, Webots is a commercial mobile robot simulation software developed by Cyberbotics Ltd [6]. The software distribution contains several models of robots called *world*,

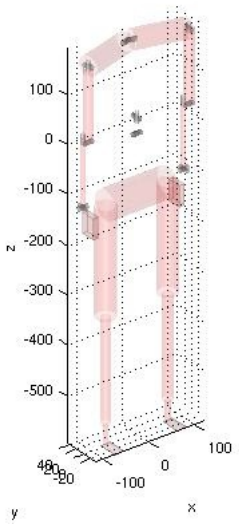


Figure 5.4: iCub drawn under Matlab

including the iCub. The worlds are described in a tree structure very similarly to the VMRL (Virtual Reality Markup Language) language. In this tree structure we can define what are the geometrical and physical properties of the robot, and which sensor or actuator we need. We can then write programs in C, C++, Java or Python to control the robot. In this project the controllers are written in C++ because they integrate KDL which is also written in C++.

The first tests I made to control the robot showed that the model of the iCub was not the same as the official model defined on [9]. The updates concerned the following points :

- The length of the limbs
- The order of the torso joints
- The eyes
- The ankle roll

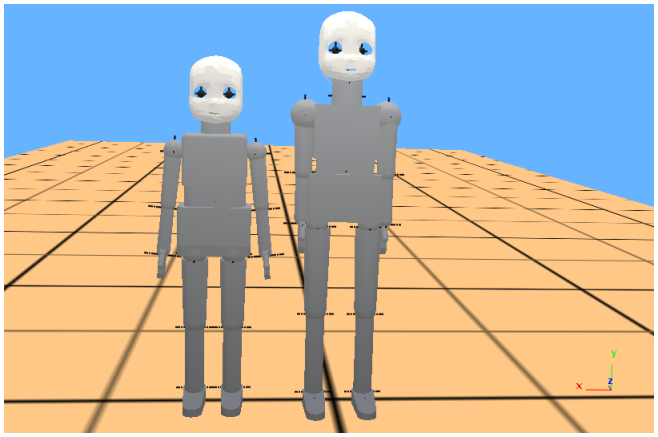
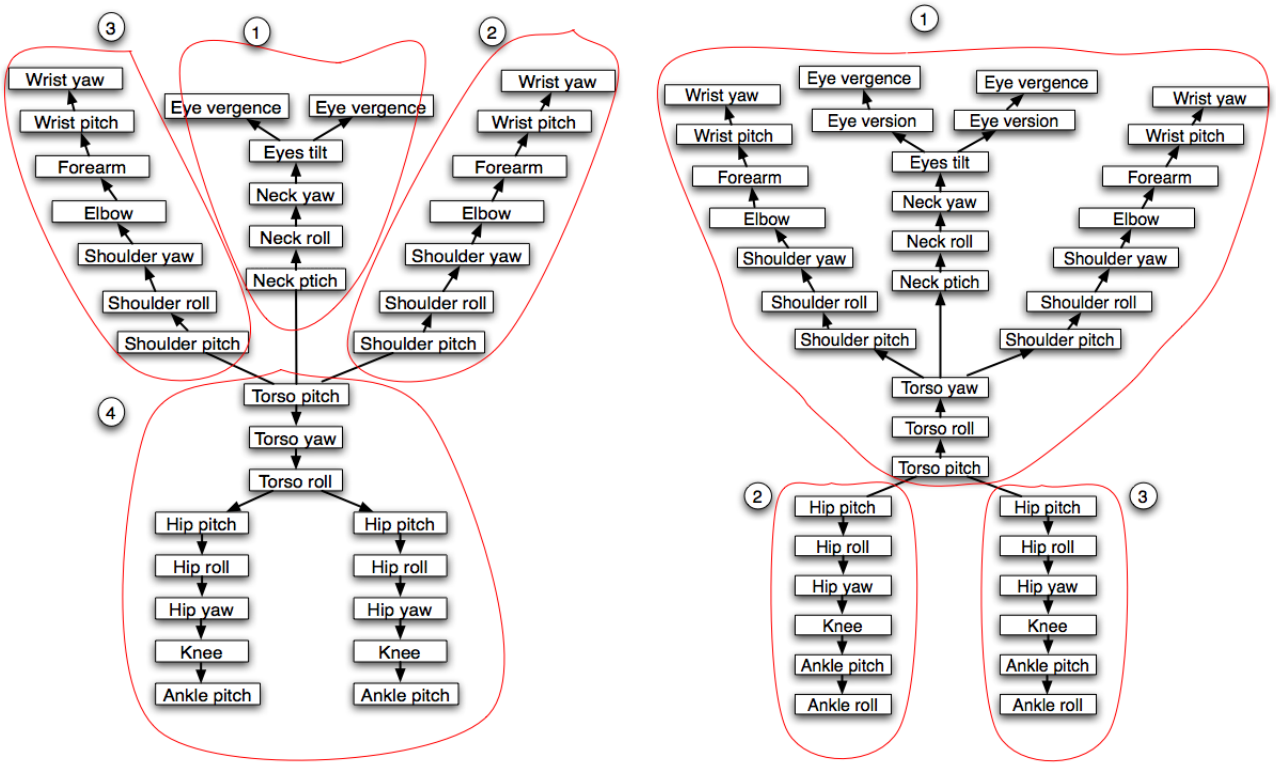


Figure 5.5: iCub model under Webots

Indeed, as we can see on figure 5.5 the limbs of the iCub differ in size, especially for the arms and legs. Besides, the order of the torso joints is not the same in the new model and

in the old model (Figure 5.6). In the old one, we first define the torso pitch, then the torso yaw and finally the torso roll. In the official model, we first define the torso pitch, followed by the torso roll and the torso yaw. We also need to add four new joints which do not exist in the old model : left and right ankle roll and left and right eye version.

In order to facilitate the modelisation, we set the new origin (0,0,0) to be the middle of the torso. This comes from the fact that when we use the DH coordinates, we usually take the middle of the torso as fixed base. Moreover, we also need to define a new order of definition for the different parts of the body (Figure 5.6). In the new model, we first describe the torso to which we attach the right and left arm and the head. Secondly we define the right and left leg.



(a) Old tree

(b) New tree

Figure 5.6: Tree describing the iCub under Webots

Under Webots, a joint is represented by a servo motor node. Each node has a rotational and translational part. The translation is the translation to go from the parent node to the child node. The rotation defines rotation axis of the servo motor. The values for the rotation and translation of each servo node is defined on Appendix B according to the Matlab model. So for example, when we want to build the tree starting at the torso pitch, we first create a new node. According to Appendix B, we set the translation to (0, 0, 0) and the rotation to (-1, 0, 0). We then add a new node for the torso roll to the children of the torso pitch. The other nodes are created the same way, such that we obtain the tree structure shown in Figure 5.6(b).

Chapter 6

Results

This chapter presents the results found by testing KDL's capabilities to model the iCub and to use it for controlling Webots' new model presented on chapter 5.

6.1 Forward position kinematics for a chain

The results for the chain including the torso and the right arm chain show that the forward kinematics for a chain works correctly. Indeed, given some initial joint values, we obtain the same end-effector for the calculations made under KDL and under Matlab, using the "official" code as explained in 5.2. When we give the joint values to the Webots model, we can see that it reaches the calculated position. Figure 6.1 shows one of the results made for the forward kinematics for a chain.

	Torso pitch	**Torso roll**	**Torso yaw**	**Shoulder pitch**	**Shoulder roll**	**Shoulder yaw**	**Elbow**	**Forearm**	**Wrist pitch**	**Wrist yaw**
θ	0	0	0	0	0	0	1.85	0	0	0

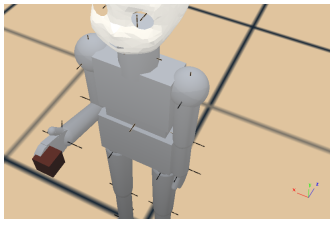
(a) Initial joint values for the right arm chain

$$\begin{bmatrix} 6.71^{-17} & 2.96^{-16} & -1 & 0.0941161 \\ 0.27559 & -0.961275 & -2.4^{-16} & 0.0636638 \\ -0.961275 & -0.27559 & -1.39^{-16} & -0.201513 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(b) KDL's end-effector frame

$$\begin{bmatrix} 0.0 & 0.0 & -1.0 & 0.09411608 \\ 0.27559 & -0.96128 & -0.0 & 0.06366380 \\ -0.96128 & -0.27559 & -0.0 & -0.20151324 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(c) Matlab's end-effector frame



(d) Webots cube position: (0.0941161, 0.0636638, -0.201513)

Figure 6.1: Chain forward kinematic

6.2 Forward position kinematics for a tree

We compute the forward kinematics for a tree with a given end-effector and for a chain containing the same end-effector. We see that in both cases we have the same result, and that the Webots models goes to the desired position. This shows that the forward position for a tree works. Here are some results obtained for the left arm.

	Torso pitch	**Torso roll**	**Torso yaw**	**Shoulder pitch**	**Shoulder roll**	**Shoulder yaw**	**Elbow**	**Forearm**	**Wrist pitch**	**Wrist yaw**
θ	0	0	0	0	0	0	1.85	0	0	0

(a) Initial joint values for the left arm chain

Torso	**Torso pitch**	**Torso roll**	**Torso yaw**
θ	0	0	0

Right arm	**Shoulder pitch**	**Shoulder roll**	**Shoulder yaw**	**Elbow**	**Forearm**	**Wrist pitch**	**Wrist yaw**
θ	0	0	0	0.1	0	0	0

Left arm	**Shoulder pitch**	**Shoulder roll**	**Shoulder yaw**	**Elbow**	**Forearm**	**Wrist pitch**	**Wrist yaw**
θ	0	0	0	1.85	0	0	0

Right leg	**Hip pitch**	**Hip roll**	**Hip yaw**	**Knee**	**Ankle pitch**	**Ankle roll**
θ	0	0	0	0	0	0

Left leg	**Hip pitch**	**Hip roll**	**Hip yaw**	**Knee**	**Ankle pitch**	**Ankle roll**
θ	0	0	0	0	0	0

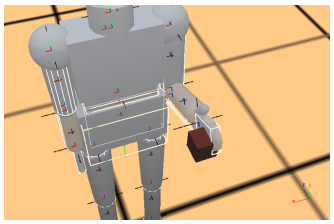
(b) Initial joint values for the tree

$$\begin{bmatrix} 1.25^{-16} & 6.87^{-17} & -1 & -0.0941161 \\ 0.27559 & -0.961275 & -3.17^{-17} & 0.0636638 \\ -0.961275 & -0.27559 & -1.39^{-16} & -0.201513 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(c) KDL's chain end-effector frame

$$\begin{bmatrix} 1.25^{-16} & 6.87^{-17} & -1 & -0.0941161 \\ 0.27559 & -0.961275 & -3.17^{-17} & 0.0636638 \\ -0.961275 & -0.27559 & -1.39^{-16} & -0.201513 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(d) KDL's tree end-effector frame



(e) Webots cube position: $(-0.0941161, 0.0636638, -0.201513)$

Figure 6.2: Tree forward kinematic

6.3 Inverse position kinematics for a chain

The first idea was to make the iCub draw a circle in space using its right arm. To do that, we need to give an end-effector frame to the program which will compute the inverse position. We have to be careful here, because if we build a new end-effector frame with only the x, y, z position coordinates, the algorithm will not work properly as it needs also the rotational part of the frame. So besides the desired cartesian position, we need also to choose an orientation.

The first result was done by calculating some points of a circle and pass these points as argument to the *KDL::ChainIkSolverPos_NR_JL* using the *KDL::ChainIkSolverVel_wdls* velocity solver. As the right arm is defined with respect to the torso and as we do not want the torso to move, we set the diagonal values in the joint space matrix to 0 for the torso and to 1 for the other (see chapter 4.1). The damping parameter lambda is set to 0.2 and the joints have 0 as initial value. When the program calculates the joint values, it emits an error message telling us that it can not compute the inverse kinematics. Sending the found joint values to the iCub model under Webots, he draws approximately a circle, but does not folds the elbow to get his hand at the right position (Figure 6.3).

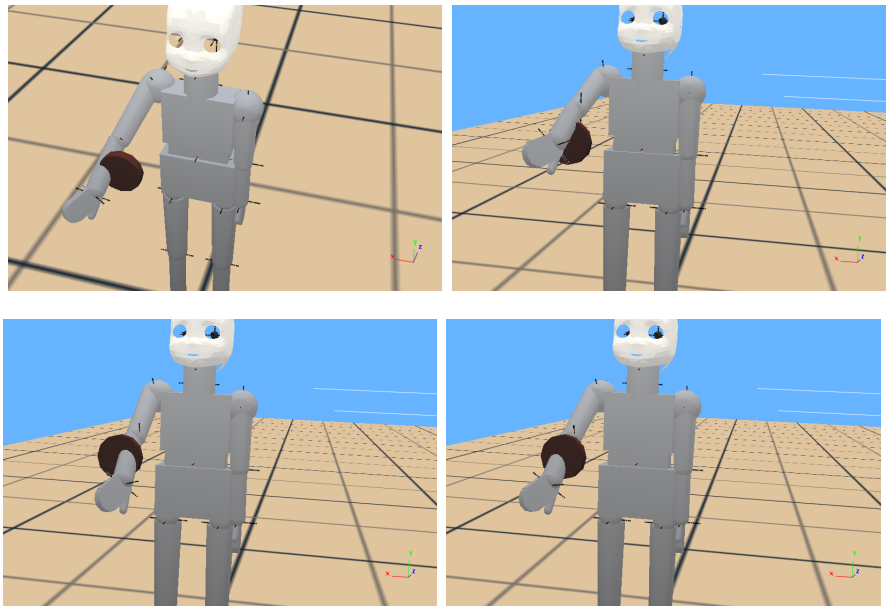


Figure 6.3: First attempt for drawing a circle with the inverse kinematic

Increasing the number of iterations of the algorithm shows that the error does not come from there, as we still have the same behavior. As the inverse position kinematics uses inverse velocity, the next idea was to try to use this velocity directly by creating a velocity controller to control the end-effector's velocity. The idea was also to try first to reach just a point in space, rather than making a circle.

Velocity controller

The velocity controller works similarly to the inverse position solver explained in 4.1. We create a forward position solver using the class *KDL::ChainFkSolverPos_recursive* and an inverse velocity solver using the class *KDL::ChainIkSolverVel_wdls*. We set the damping λ and the initial joint positions to the desired value. We then enter a loop where we try to find the joint values which will bring us to the desired position, with an error less than e. What is different here, is that before computing the forward kinematics, we can control the

joint velocities, by setting the *timestep* to go from the estimated end-effector position to the desired end-effector position. This is done by solving the following equation

$$\overrightarrow{q}' = \overrightarrow{q} + time\_step * \overrightarrow{\dot{q}} \tag{6.1}$$

Once we have calculated the estimates end-effector position, we calculate the twist the following way:

$$\overrightarrow{twist} = \frac{\overrightarrow{end\_effector} - \overrightarrow{current\_end\_effector}}{\Delta t} \tag{6.2}$$

where Δt is set to 1 by default. A gain term is then used to modulate the speed.

$$\overrightarrow{twist}' = gain * \overrightarrow{twist} \tag{6.3}$$

Velocity controller for a point

In this case as we only need to reach a given point, we send the joint positions at each step of the main loop of the controller where he tries to find the right end-effector position. Here are some results

The first test was made with the following parameters. Here again the torso does not move, and we set the initial joint position to 0 in KDL and in Webots. The damping parameter λ is set to 0 as well as the initial joint velocities. Now the control parameter are set to 0.1 for the time step and 1.0 for the gain. The convergence is attained when the error e is below 0.1. When we launch the simulation, the robot falls and behaves as if it "convulses" (Figure 6.4). This is due to the fact that we are in a singularity. Indeed, if we look at the values the pseudo inverse gives us, we have "nan" values, i.e there are divisions by 0 in this matrix. Thus we can not calculate the SVD and not solve the problem.

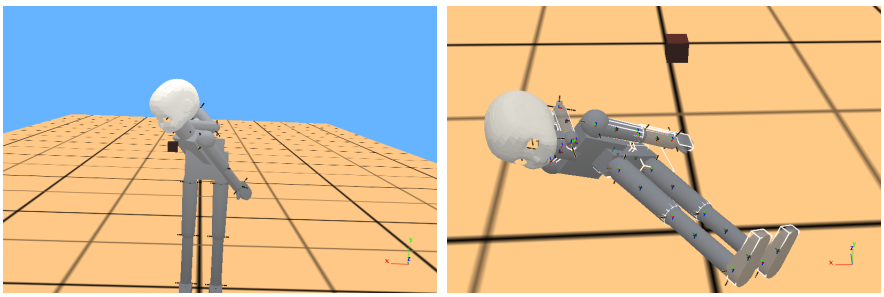


Figure 6.4: $\lambda = 0, time\_step = 0.1, gain = 1.0, e = 0.1$

The two next steps are done with the same parameters as previously except for lambda. In Figure 6.3 the damping is set to 0.1. We can see that the iCub first goes up vertically with its arm, and then goes down until its position stops evolving. Its arm is aligned with the cube, but the elbow is still not bent.

In Figure 6.6, where $\lambda = 0.2$, the iCub reaches the same final position as previously, but here it starts moving its arm laterally instead of moving it vertically. If we try to set lambda to 0.3, the robot behaves like in Figure 6.6 except that it gets stuck with its arm in

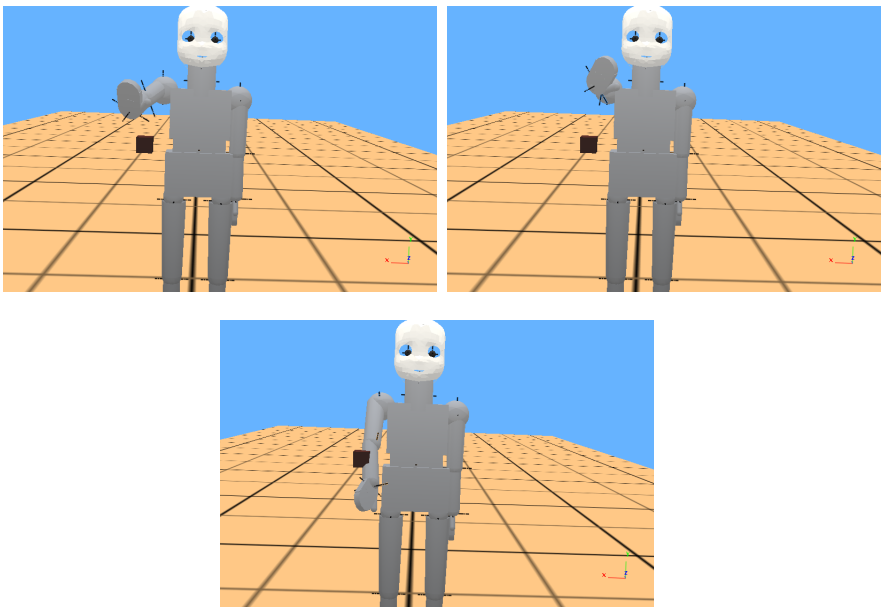


Figure 6.5: $\lambda = 0.1, time\_step = 0.1, gain = 1.0, e = 0.1$

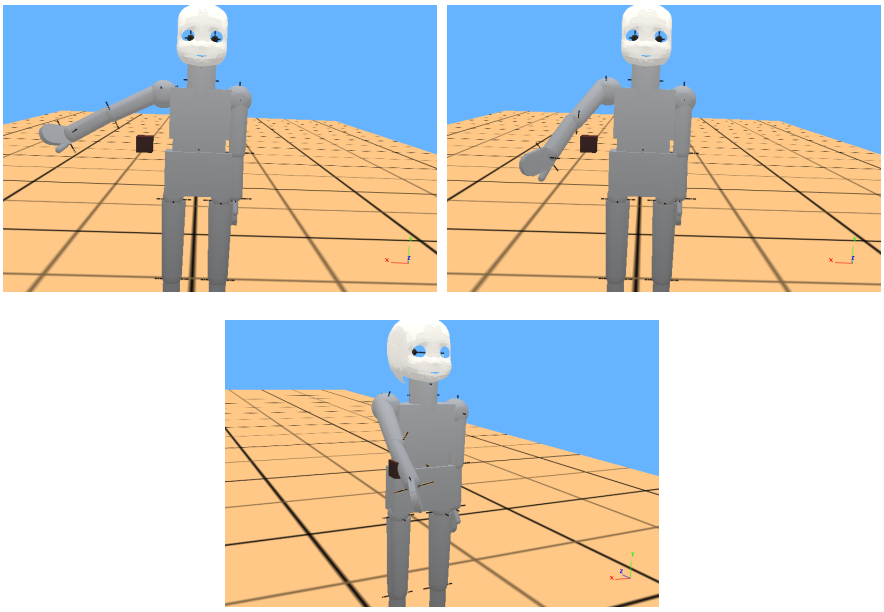


Figure 6.6: $\lambda = 0.2, time\_step = 0.1, gain = 1.0, e = 0.1$

a horizontal position and thus does not converge at all to the solution. As seen in 4.1.1 the damping parameter is used to avoid singular configurations, but if lambda increases, we have a greater error for the pseudo inverse. It is why we do not reach any solution with $\lambda = 0.3$.

Let us now try the controller with $\lambda = 0.2$ and initial joint values all set to 0 except for the elbow. Why just for the elbow? As seen in section 4.1.1, the damping parameter λ can avoid to enter in a singular configuration, but if we are already in such a configuration, we also can not get out of it. Now, for the arm, the possible singular configurations are when the elbow is totally streched or totally bent.

If we try for initial elbow joint values 0, 0.1 and 0.2, we observe the same behavior than in Figure 6.6. For joint values greater or equal to 0.3, we have the following behavior. Now, the iCub folds its elbow, but does not really reaches the solution (Figure 6.7). If we try with $\lambda = 0.1$ and $\theta_{elbow} = 0.3$ we can see that the icub reaches the point with its hand (Figure 6.8).

It works better with $\lambda = 0.1$ than with $\lambda = 0.2$ or 0.0 because, as said before, if lambda increases, the error increases too and the robot will not reach the desired solution. Besides if lambda is too small, we might get into a singularity and not be able to calculate the pseudo-inverse.

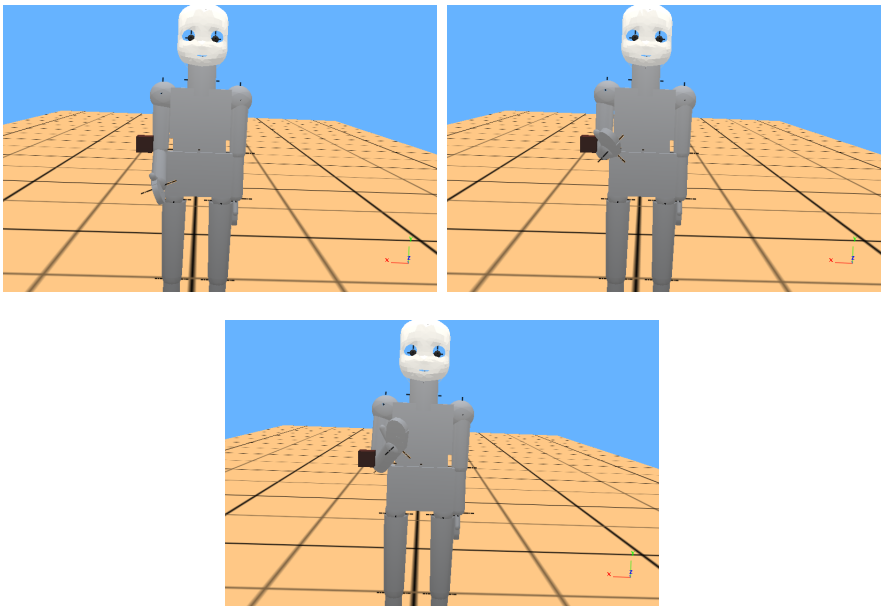


Figure 6.7: $\lambda = 0.2, time\_step = 0.1, gain = 1.0, e = 0.1, \theta_{elbow} = 0.3$

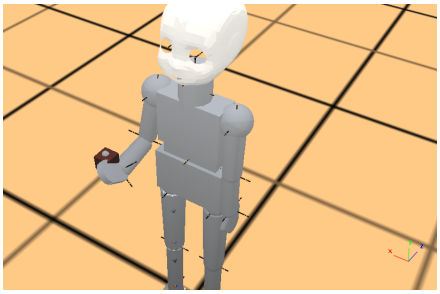


Figure 6.8: $\lambda = 0.1, time\_step = 0.1, gain = 1.0, e = 0.1, \theta_{elbow} = 0.3$

Now we can deduce that the best parameters are :

- $\theta_{0,i} = 0.0$ except $\theta_{0,elbow} = 0.3$
- $\lambda = 0.1$
- time_step = 0.1
- gain = 1.0
- e = 0.1

If we try other values for the *time_step* or the *gain* we can observe that it only changes the speed of the convergence to the solution. Changing the values for the error does not change anything.

Now if we try to reach a point which is too far away from the iCub, we can see that he nevertheless approaches the solution (Figure 6.9).

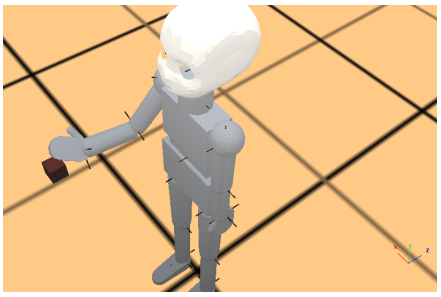


Figure 6.9: $\lambda = 0.1, time\_step = 0.1, gain = 1.0, e = 0.1, \theta_{elbow} = 0.3$

6.3.1 Velocity controller for a circle

Now that reaching a point with the inverse kinematics works, let us retry with a circle. In this case we have more than one point for which we want to find the joint positions. As the controller goes into an infinite loop, we decide to let him calculate during 16, 100 or 1000 loops, and then send the current joint positions to the controller of the Webots model. We can observe that the more loops we have, the better the shape of the circle is. As the shape does not improve for an iteration number greater than 1000, we will keep this number of iterations to find the best values for the other parameters.

Let us first set all joints to an initial value of 0, except for the elbow which will have an initial value of 0.3. The damping parameter λ is set to 0.2, the time step to 0.1, the gain to 1.0 and the error to 0.1. We still do not want the torso to move, so we set the corresponding values to 0 in the weighting matrix. We can see on Figure 6.10 that the iCub draws a circle, but not at the right place. Again this might be due to the fact that the value for λ is too big.

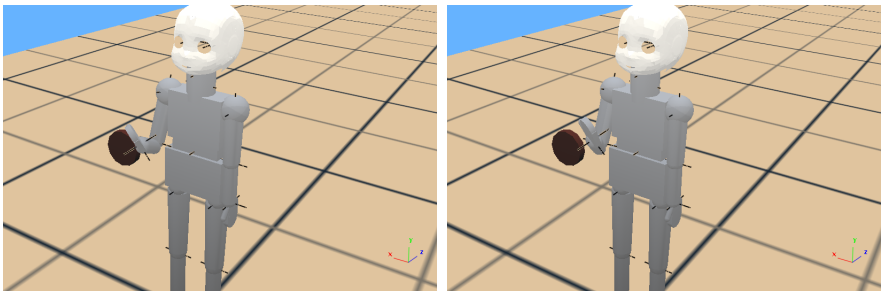


Figure 6.10: $\lambda = 0.2, time\_step = 0.1, gain = 1.0, e = 0.1, \theta_{elbow} = 0.3$

If we try to set lambda to 0.1, the iCub draws the circle almost at the right place (Figure 6.11). If we have a closer look to the calculations done by the program, we can observe that the minimum is attained for the forearm and for the wrist yaw, and that the maximum is attained for the wrist pitch.

So the next step is to make the same calculations, but without joint limits in the velocity controller as well as in Webots. We can notice that the controller does not go into an infinite loop. This means that the controller has reached a solution. In Webots we see that the iCub draws a circle with the elbow bent and that the end-effecor position is almost, but not exactly, at the right place (Figure 6.12).

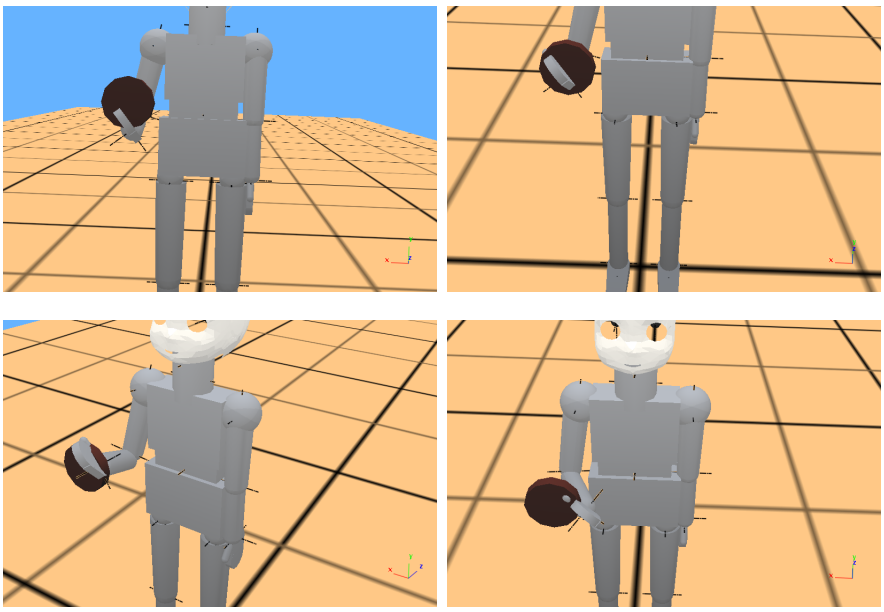


Figure 6.11: $\lambda = 0.1, time\_step = 0.1, gain = 1.0, e = 0.1, \theta_{elbow} = 0.3$

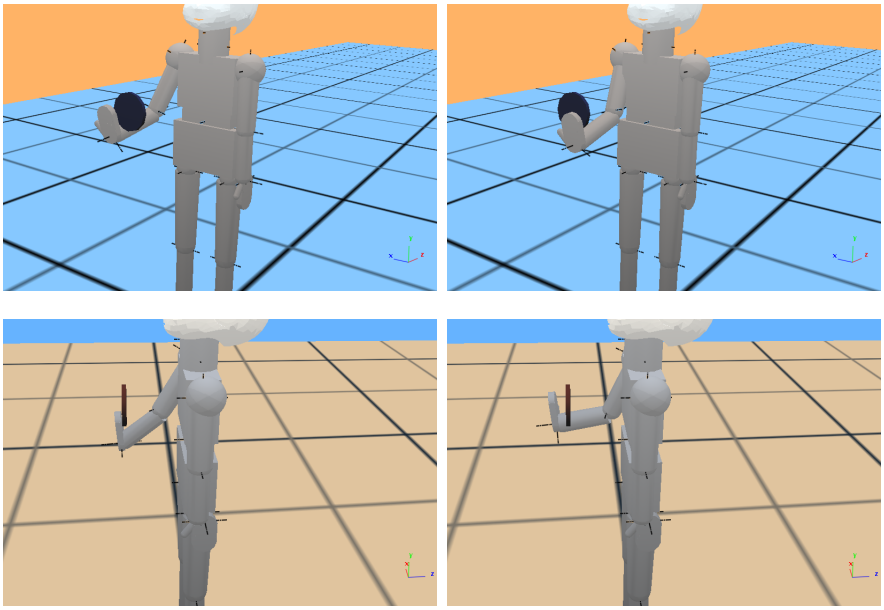


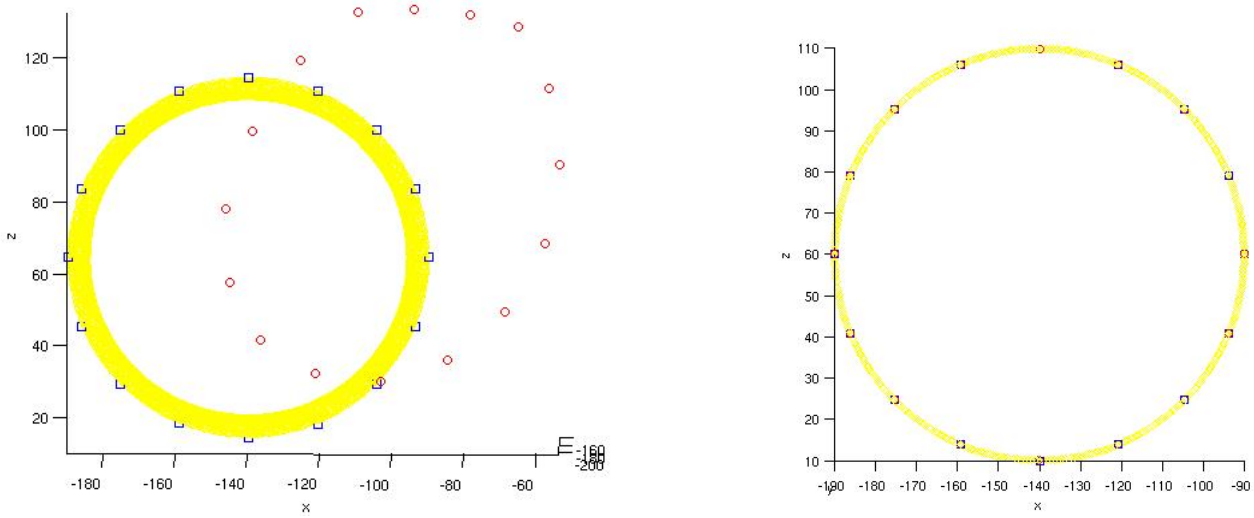
Figure 6.12: No joint limits

6.3.2 Inverse kinematic for a circle

Before we analyze the difference between the input points and the points reached with the inverse joint positions, let us compute the inverse position kinematics as done at the begining of section 6.3, but by setting the following parameters : $\lambda = 0.1$, $\theta_i = 0$ and $\theta_{elbow} = 0.3$. We can observe the same shape as with the velocity controller (Figure 6.11). If we do not take the joint limits into account as well as in KDL than in Webots, we also obtain the same shape as with the velocity controller (Figure 6.12).

If we plot the points of the circle that were calculated and those found with the inverse kinematics with and without the joint limits, we have the following figure.

In yellow, we have the orginal circle. The squares represent the points on the circle which are given as end-effector positions. The dots represent the end-effector positions found



(a) With joint limits

(b) Without joint limits

Figure 6.13: Circle : yellow circle = original circle; □ = calculated input circle points; ∘ = circle points calculated by KDL

by applying the forward position kinematics on the joint positions found with the inverse kinematics. As we can see on Figure 6.13(b) the input circle points and the points calculated by KDL are at the same place. Thus, when we do not take the joint limits into account, the inverse position algorithm reaches the desired solution. On the other hand, if we take the joint limits into account, we can observe that the shape we obtain with KDL is still a circle, but that it is not at the desired position. This is due to the following reasons. First, as we have chosen not to move the torso, it is more difficult to reach some positions as we only can use 7 DoFs instead of 10. Moreover, if we have some constraints on the joint positions, it is even harder to reach this position (In 6.3.1 we have seen that we attained joint limits for the forearm and the wrist). As the inverse position kinematics works well and as the iCub in Webots does not make a circle with its arm at the same position as the red circle, it might be that there is a bug with the way the model under Webots is controlled.

Conclusion

In this project we have updated the Webots model of the iCub such that it is conform to the official version found in [9]. It has been shown that forward position kinematics is quite easy to implement and works well under KDL, whereas the inverse position problem is more difficult to solve. Indeed, we have to handle with the multiplicity of the solutions and the singular configurations. In spite these drawbacks, the iCub can nevertheless reach a simple point in space, and even draw a more complex shape like a circle.

Future work

To extend this work, we can do the following. First, we could try to resolve the bug under with Webots, in order to have a better simulation. As mentioned in 6.3.2, when we take the joint limits into account, we do not reach the desired position. We could try to allow the torso to move and see if the iCub can reach the desired position even if the joint angles are constrained. As the part of the code which tests if the joint limits are reached is quite simple, we could try to find another way to respect these limits and at the same time to reach the desired solution. Besides, as the algorithm needs an orientation of the end-effector, we could try to see if we can improve the solution by taking another orientation. Finally, because of lack of time, we could not test the inverse position kinematics for a tree, but it might be interesting to test it and compare it with the inverse kinematics for a chain. On a long term view, the model we have developed can be used to control gait stability or to impose kinematic constants for instance.

Appendix A

Denavit-Hartenberg's parameter for the iCub

Here are the Denavit-Hartenberg parameters for the iCub. They come from the wiki for the iCub [9] and are from the 8th of October 2008. The matrix T_Ro0 describes the rigid roto-translation to aligne the root reference frame with the z axis of the first joint. When we model the iCub with these parameters, the only variables are θ_i and they take their values in the interval specified in "[ ]".

A.1 Torso and right arm

Joint name	**Link i**	A_i **[mm]**	d_{i+1} **[mm]**	α_i **[rad]**	θ_{i+1} **[deg]**
Torso pitch	i = 0	32	0	$\pi/2$	[-22 ; 84]
Torso roll	i = 1	0	0	$\pi/2$	-90 + [-39 ; 39]
Torso yaw	i = 2	-23.3647	-143.3	$\pi/2$	-105 + [-59 ; 59]
Shoulder pitch	i = 3	0	-107.74	$\pi/2$	-90 + [95 ; -95]
Shoulder roll	i = 4	0	0	$-\pi/2$	[0 ; 160.8] - 90
Shoulder yaw	i = 5	0	-152.28	$-\pi/2$	[-37 ; 100] -105
Elbow	i = 6	15	0	$\pi/2$	[5.5 ; 106]
Forearm	i = 7	0	-137.3	$\pi/2$	[-50 ; 50] - 90
Wrist pitch	i = 8	0	0	$\pi/2$	[10 ; -65] + 90
Wrist yaw	i = 9	62.5	16	0	[-25 ; 25] + 180

$$T\_Ro0 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A.2 Torso and left arm

Joint name	**Link i**	A_i **[mm]**	d_{i+1} **[mm]**	α_i **[rad]**	θ_{i+1} **[deg]**
Torso pitch	i = 0	32	0	$\pi/2$	[-22 ; 84]
Torso roll	i = 1	0	0	$\pi/2$	-90 + [-39 ; 39]
Torso yaw	i = 2	23.3647	-143.3	$-\pi/2$	105 + [-59 ; 59]
Shoulder pitch	i = 3	0	107.74	$-\pi/2$	90 + [95 ; -95]
Shoulder roll	i = 4	0	0	$\pi/2$	[0 ; 160.8] - 90
Shoulder yaw	i = 5	0	152.28	$-\pi/2$	75 + [-37 ; 100]
Elbow	i = 6	-15	0	$\pi/2$	[5.5 ; 106]
Forearm	i = 7	0	137.3	$\pi/2$	[-50 ; 50] - 90
Wrist pitch	i = 8	0	0	$\pi/2$	[10 ; -65] + 90
Wrist yaw	i = 9	62.5	-16	0	[-25 ; 25]

$$T\_Ro0 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A.3 Right leg

Joint name	**Link i**	A_i **[mm]**	d_{i+1} **[mm]**	α_i **[rad]**	θ_{i+1} **[deg]**
Hip pitch	i = 0	0	0	$\pi/2$	[-44 ; 132]
Hip roll	i = 1	0	0	$\pi/2$	90 + [-119 ; 17]
Hip yaw	i = 2	0	-223.6	$-\pi/2$	-90 + [-79 ; 79]
Knee	i = 3	213	0	0	90 + [-125 ; 23]
Ankle pitch	i = 4	0	0	-$\pi/2$	[-42 ; 21]
Ankle roll	i = 5	-41	0	π	180 + [-24 ; 24]

$$T\_Ro0 = \begin{bmatrix} 0 & 0 & -1 & -68.1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & -119.9 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A.4 Left leg

Joint name	**Link i**	A_i **[mm]**	d_{i+1} **[mm]**	α_i **[rad]**	θ_{i+1} **[deg]**
Hip pitch	i = 0	0	0	-$\pi/2$	[-44 ; 132]
Hip roll	i = 1	0	0	$-\pi/2$	90 + [-119 ; 17]
Hip yaw	i = 2	0	223.6	$\pi/2$	-90 + [-79 ; 79]
Knee	i = 3	213	0	0	90 + [-125 ; 23]
Ankle pitch	i = 4	0	0	$\pi/2$	[-42 ; 21]
Ankle roll	i = 5	-41	0	π	180 + [-24 ; 24]

$$T\_Ro0 = \begin{bmatrix} 0 & 0 & -1 & 68.1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & -119.9 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Appendix B

Coordinates for the Webots model

Here are the coordinates of the joints for the iCub. There are read on the Matlab model presented in section 5.2. The Matlab model is based on the .m files downloaded on the 8th of October 2008. The translation column represents the translation with respect to the previous joint. It is this value which will be added to the translation field of each servo in the Webots model. The rotation axis is defined in Webots such that if we pass a given angle value to joint i, it behaves the same way in Matlab than in Webots.

B.1 Torso

Joint name	**Matlab coord [m]**	**Webots coord [m]**	**Webots Translation[m]**	**Webots rotation**
Torso pitch	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(-1, 0, 0)
Torso roll	(0, 0, 0.032)	(0, 0.032, 0)	(0, 0.032, 0)	(0, 0, 1)
Torso yaw	(0, 0, 0.032)	(0, 0.032, 0)	(0, 0, 0)	(0, -1, 0)

B.2 Right arm

The shoulder pitch is defined with respect to the torso yaw.

Joint name	**Matlab coord [m]**	**Webots coord [m]**	**Webots Translation[m]**	**Webots rotation**
Shoulder pitch	(-0.00605, 0.0226, 0.175)	(0.00605, 0.175, 0.0226)	(0.0065, 0.143, 0.0226)	(-1, 0, 0)
Shoulder roll	(-0.110, -0.00532, 0.175)	(0.110, 0.175, -0.00532)	(0.10395, 0, -0.02792)	(0, 0, 1)
Shoulder yaw	(-0.110, -0.00532, 0.175)	(0.110, 0.175, -0.00532)	(0, 0, 0)	(0, 1, 0)
Elbow	(-0.110, -0.00532, 0.023)	(0.110, 0.023, -0.00532)	(0, -0.152, 0)	(1, 0, 0)
Forearm	(-0.110, -0.00968, 0.023)	(0.110, 0.023, -0.00968)	(0, 0, 0.015)	(0, 1, 0)
Wrist pitch	(-0.110, -0.00968, -0.114)	(0.110, -0.114, -0.00968)	(0, -0.137, 0)	(0, 0, -1)
Wrist yaw	(-0.110, -0.00968, -0.114)	(0.110, -0.114, -0.00968)	(0, 0, 0)	(-1, 0, 0)

B.3 Left arm

The shoulder pitch is defined with respect to the torso yaw.

Joint name	Matlab coord [m]	Webots coord [m]	Webots Translation[m]	Webots rotation
Shoulder pitch	(0.00605, 0.0226, 0.175)	(-0.00605, 0.175, 0.0226)	(-0.0065, 0.143, 0.0226)	(-1, 0, 0)
Shoulder roll	(0.110, -0.00532, 0.175)	(-0.110, 0.175, -0.00532)	(-0.10395, 0, -0.02792)	(0, 0, -1)
Shoulder yaw	(0.110, -0.00532, 0.175)	(-0.110, 0.175, -0.00532)	(0, 0, 0)	(0, 0, 1)
Elbow	(0.110, -0.00532, 0.023)	(-0.110, 0.023, -0.00532)	(0, -0.152, 0)	(1, 0, 0)
Forearm	(0.110, -0.00968, 0.023)	(-0.110, 0.023, -0.00968)	(0, 0, 0.015)	(0, 1, 0)
Wrist pitch	(0.110, -0.00968, -0.114)	(-0.110, -0.114, -0.00968)	(0, -0.137, 0)	(0, 0, 1)
Wrist yaw	(0.110, -0.00968, -0.114)	(-0.110, -0.114, -0.00968)	(0, 0, 0)	(-1, 0, 0)

B.4 Right leg

Joint name	Matlab coord [m]	Webots coord [m]	Webots Translation[m]	Webots rotation
Hip pitch	(-0.0681, 0, -0.120)	(0.0681,-0.120, 0)	(0.0681, -0.120, 0)	(1, 0, 0)
Hip roll	(-0.0681, 0, -0.120)	(0.0681,-0.120, 0)	(0, 0, 0)	(0, 0, -1)
Hip yaw	(-0.0681, 0, -0.120)	(0.0681,-0.120, 0)	(0, 0, 0)	(0, 1, 0)
Knee	(-0.0681, 0, -0.360)	(0.0681,-0.360, 0)	(0, -0.240, 0)	(1, 0, 0)
Ankle pitch	(-0.0681, 0, -0.580)	(0.0681,-0.580, 0)	(0, -0.220, 0)	(1, 0, 0)
Ankle roll	(-0.0681, 0, -0.580)	(0.0681,-0.580, 0)	(0, 0, 0)	(0, 0, -1)

B.5 Left leg

Joint name	Matlab coord [m]	Webots coord [m]	Webots Translation[m]	Webots translation
Hip pitch	(0.0681, 0, -0.120)	(-0.0681,-0.120, 0)	(-0.0681, -0.120, 0)	(1, 0, 0)
Hip roll	(0.0681, 0, -0.120)	(-0.0681,-0.120, 0)	(0, 0, 0)	(0, 0, 1)
Hip yaw	(0.0681, 0, -0.120)	(-0.0681,-0.120, 0)	(0, 0, 0)	(0, -1, 0)
Knee	(0.0681, 0, -0.360)	(-0.0681,-0.360, 0)	(0, -0.240, 0)	(1, 0, 0)
Ankle pitch	(0.0681, 0, -0.580)	(-0.0681,-0.580, 0)	(0, -0.220, 0)	(1, 0, 0)
Ankle roll	(0.0681, 0, -0.580)	(-0.0681,-0.580, 0)	(0, 0, 0)	(0, 0, 1)

B.6 Head

The neck pitch is defined with respect to the torso yaw.The right and left eye version is defined with respect to the eyes tilt.

Joint name	**Matlab coord [m]**	**Webots coord [m]**	**Webots Translation[m]**	**Webots rotation**
Neck pitch	(0, -0.00231, 0.225)	(0, 0.225, -0.00231)	(0, 0.193, -0-231)	(1, 0, 0)
Neck roll	(0, -0.00231, 0.258)	(0, 0.258, -0.00231)	(0, 0.033, 0)	(0, 0, 1)
Neck yaw	(0, -0.00231, 0.258)	(0, 0.258, -0.00231)	(0, 0, 0)	(0, 1, 0)
Eyes tilt	(0, -0.0563, 0.341)	(0, 0.341, -0.0563)	(0, 0.083, 0)	(1, 0, 0)
Right eye version	(-0.034, -0.0563, 0.341)	(0.034, 0.341, -0.0563)	(0.034, 0, 0)	(0, 1, 0)
Right eye vergence	(-0.034, -0.0563, 0.341)	(0.034, 0.341, -0.0563)	(0, 0, 0)	(0, 0, 1)
Left eye version	(0.034, -0.0563, 0.341)	(-0.034, 0.341, -0.0563)	(0.034, 0, 0)	(0, 1, 0)
Left eye vergence	(0.034, -0.0563, 0.341)	(-0.034, 0.341, -0.0563)	(0, 0, 0)	(0, 1, 0)

Bibliography

[1] Herman Bruyninckx. Open robot control software: the OROCOS project. In *IEEE Int. Conf. Robotics and Automation*, pages 2523–2528, 2001.

[2] Herman Bruyninckx. Open RObot COntrol Software : http://www.orocos.org/, 2008.

[3] Samuel R. Buss. Introduction to inverse kinematics with jacobian transpose, pseudoinverse and damped least squares methods. Technical report, Department of Mathematics, University of California, San Diego, April 17 2004.

[4] Forward Kinematics : The Denavit-Hartenberg Convention. http://www.cs.dartmouth.edu/ donaldclass/bio/current/papers/chap3-forward-kinematics.pdf.

[5] R. Featherstone. *Rigid Body Dynamcic*. Springer, 2008.

[6] O. Michel. Webots: Professional mobile robot simulation. *Journal of Advanced Robotics Systems*, 1(1):39–42, 2004.

[7] Inverse position kinematics. http://www.roble.info/robotics/serial/html/serialrobots-1se8.html.

[8] RobotCub. http://www.robotcub.org. RobotCub website.

[9] RobotCub wiki : iCub Forward Kinematics. http://eris.liralab.it/wiki/icubforwardkinematics.