**KDL使用方法小结**

[这里](https://github.com/AngelLM/Thor/issues/30)提到了KDL是Moveit默认的IK解决方案，参考[链接](http://www.orocos.org/kdl/installation-manual)进行源码下载，即：  
git clone https://github.com/orocos/orocos\_kinematics\_dynamics.git

采用cmake的方式编译如下：

mkdir <kdl-dir>/build ; cd <kdl-dir>/build //在orocos\_kinematics\_dynamics/orocos\_kinematics\_dynamics目录创建build文件夹

ccmake .. //启动ccmake，按c进行配置，如选择安装路径，按g生成makefile

make

make install //可能需要sudo权限

编写[测试程序](http://www.orocos.org/kdl/examples)：

mkdir test-ik

vi test.cpp

复制粘贴如下内容：

// Copyright (C) 2007 Francois Cauwe <francois at cauwe dot org>

// This library is free software; you can redistribute it and/or

// modify it under the terms of the GNU Lesser General Public

// License as published by the Free Software Foundation; either

// version 2.1 of the License, or (at your option) any later version.

#include <kdl/chain.hpp>

#include <kdl/chainfksolver.hpp>

#include <kdl/chainfksolverpos\_recursive.hpp>

#include <kdl/frames\_io.hpp>

#include <stdio.h>

#include <iostream>

using namespace KDL;

int main( int argc, char\*\* argv )

{

//Definition of a kinematic chain & add segments to the chain

KDL::Chain chain;

chain.addSegment(Segment(Joint(Joint::RotZ),Frame(Vector(0.0,0.0,1.020))));

chain.addSegment(Segment(Joint(Joint::RotX),Frame(Vector(0.0,0.0,0.480))));

chain.addSegment(Segment(Joint(Joint::RotX),Frame(Vector(0.0,0.0,0.645))));

chain.addSegment(Segment(Joint(Joint::RotZ)));

chain.addSegment(Segment(Joint(Joint::RotX),Frame(Vector(0.0,0.0,0.120))));

chain.addSegment(Segment(Joint(Joint::RotZ)));

// Create solver based on kinematic chain

ChainFkSolverPos\_recursive fksolver = ChainFkSolverPos\_recursive(chain);

// Create joint array

unsigned int nj = chain.getNrOfJoints();

KDL::JntArray jointpositions = JntArray(nj);

// Assign some values to the joint positions

for(unsigned int i=0;i<nj;i++){

float myinput;

printf ("Enter the position of joint %i: ",i);

scanf ("%e",&myinput);

jointpositions(i)=(double)myinput;

}

// Create the frame that will contain the results

KDL::Frame cartpos;

// Calculate forward position kinematics

bool kinematics\_status;

kinematics\_status = fksolver.JntToCart(jointpositions,cartpos);

if(kinematics\_status>=0){

std::cout << cartpos <<std::endl;

printf("%s **\n**","Succes, thanks KDL!");

}else{

printf("%s **\n**","Error: could not calculate forward kinematics :(");

}

}

采用g++ test.cpp -lkdl\_parser -lorocos-kdl -o main进行编译生成main可执行文件。

To build one of the examples do:

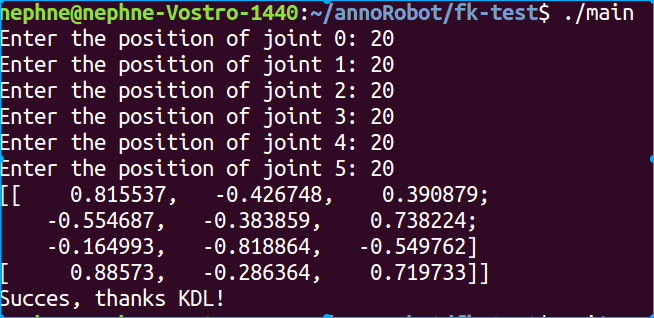
g++ -I<KDL\_INCLUDE\_DIR> -L<KDL\_LIB\_DIR> -lorocos-kdl <example.cpp> -o <example>

可能需要注意更改c默认搜索路径：

export CPLUS\_INCLUDE\_PATH=/opt/ros/kinetic/include:/usr/include/eigen3 //该路径可以通过locate kdl/chain.hpp进行确认

export LIBRARY\_PATH=/opt/ros/kinetic/lib //该路径可以通过grep -nr addSegment进行确认

最后执行可执行文件如下：



关于该程序的解析过程见下面的[文章分析](https://medium.com/@sarvagya.vaish/forward-kinematics-using-orocos-kdl-da7035f9c8e)。

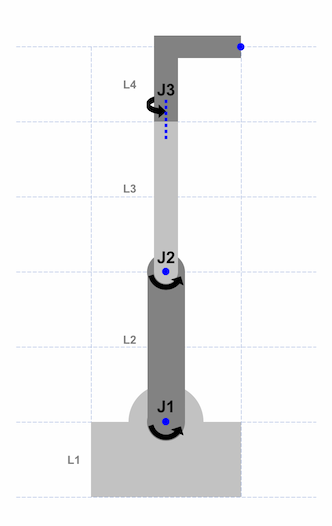
**Forward Kinematics using Orocos KDL**

This post briefly describes how to define coordinate systems for robots and then explains how to use [Orocos KDL](http://www.orocos.org/kdl) to perform [forward kinematics](https://en.wikipedia.org/wiki/Forward_kinematics).

**A Simple Robot**

To make things a little easier, we’ll be using the simple robot model shown below. It has four links (L1 — L4) and three degrees of freedom. Link 1 is fixed to the ground while Link 2 and Link 3 can rotate in the screen plane. Link 4 can rotate around like a periscope, about the axis shown in the schematic.

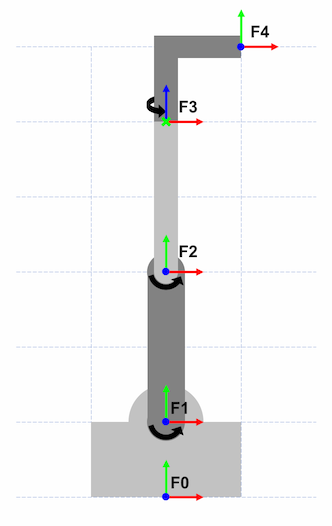
Our goal is to find the pose of the last link, the end effector, given the measurement of three joint angles (J1, J2, J3).

Links and Joints

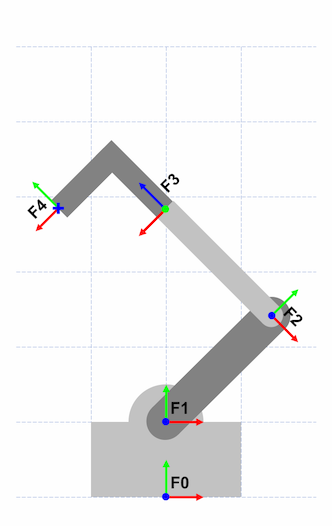
**Defining Coordinate Frames**

Let’s start with defining the coordinate frames for each link. **F0** is our reference coordinate frame, the frame relative to which we would like to find our end effector’s pose, **F4**.

I am using the widely used Denavit-Hartenberg convention to define the intermediate links. Specifically, it says that the Z axis should be in the direction of the joint axis.

Coordinate frames attached to links

* **F1** is attached to Link 1 with the Z axis aligned to Joint 1, coming out of the plane.
* **F2** is attached to Link 2 with the Z axis aligned to Joint 2, coming out of the plane.
* **F3** is attached to Link 3 with the Z axis aligned to Joint 3, along the axis of rotation for Link 4.

Robot pose for joint angles -45°, 90°, 180°

Here’s another image to illustrate how the coordinate frames are attached to the links and joints.

* Joint 1 is rotated -45°
* Joint 2 is rotated 90°
* Joint 3 is rotated 180°

Assuming that the origin is at F0, the end effector’s position, F4, is roughly at (-1.5, 3.8, 0). We’ll calculate this as well the the full pose using forward kinematics in the next section.

**Forward Kinematics**

**What is Orocos KDL?**

[Orocos](http://www.orocos.org/) (Open Robot Control Software) is a suite of libraries for robot arm control. Orocos KDL (Kinematics and Dynamics Library) provides the ability to create kinematic chains to perform forward and inverse kinematics.

**Step 1: Creating Segments**

In KDL, kinematic chains consist of segments. Segments are essentially the links of the robot. A segment is a combination of a Joint and a Frame. The joint tells the segment how a frame moves as the joint angle changes. For example, it specifies if the frame rotates about the Z axis.

Frames are specified relative to the preceding frame:

* **F1** is defined relative to **F0**
* **F2** is defined relative to **F1**
* **F3** is defined relative to **F2**
* **F4** is defined relative to **F3**

Let’s create an empty chain to which we’ll add the segments.

**Chain** kdlChain = **Chain**();

For the first segment, note how **F1** is oriented the same way as **F0**, just located at (0, 1, 0). Hence, we can initialize our frame using a simple vector. Also note that **F1** doesn’t move relative to **F0**. In this case we’ll use a Joint::None while creating the segment.

**Joint** joint1(**Joint::None**);

**Frame** frame1 = **Frame**(**Vector**(0.0, 1.0, 0.0));

kdlChain.**addSegment**(**Segment**(joint1, frame1));

Similarly, for **F2** there is again no change in orientation of the frame, just a translation to (0, 2, 0). Link 2 however rotates about the Z axis of the previous frame, so we’ll use Joint::RotZ.

**Joint** joint2(**Joint::RotZ**);

**Frame** frame2 = **Frame**(**Vector**(0.0, 2.0, 0.0));

kdlChain.**addSegment**(**Segment**(joint2, frame2));

**F3** is rotated about the X axis by -90° and then translated 2 units about the new Z axis. We can multiply two intermediate frames to construct the final one.

**Joint** joint3(**Joint::RotZ**);

**Frame** frame3 = **Frame**(**Rotation::EulerZYX**(0.0, 0.0, -M\_PI / 2)) \*

**Frame**(**Vector**(0.0, 0.0, 2.0));

kdlChain.**addSegment**(**Segment**(joint3, frame3));

**F4** first reverses the previous rotation and is then translated 1 unit in the X and Y directions.

**Joint** joint4(**Joint::RotZ**);

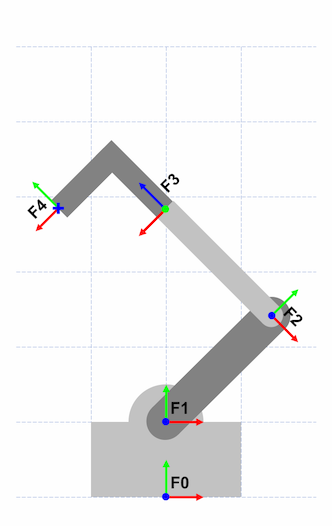
**Frame** frame4 = Frame(**Rotation::EulerZYX**(0.0, 0.0, M\_PI / 2)) \*

Frame(**Vector**(1.0, 1.0, 0.0));

kdlChain.**addSegment**(**Segment**(joint4, frame4));

**Step 2: Joint Angles**

Next we’ll construct a joint angles variable that will contain the three angles for which we wish to perform forward kinematics.

Robot pose for joint angles -45°, 90°, 180°

**JntArray** jointAngles = **JntArray**(3);

jointAngles(0) = -M\_PI / 4.; // Joint 1

jointAngles(1) = M\_PI / 2.; // Joint 2

jointAngles(2) = M\_PI; // Joint 3

**Step 3: Forward Kinematics**

Finally, we can run forward kinematics for the joint angles in step 2, using the chain we constructed in step 1. This will give is the end effector pose.

**ChainFkSolverPos\_recursive** FKSolver =

**ChainFkSolverPos\_recursive**(kdlChain);

**Frame** eeFrame;

**FKSolver**.**JntToCart**(jointAngles, eeFrame);

Forward kinematics solution (contents of eeFrame):

-0.7071 -0.7071 0. -1.414

-0.7071 0.7071 0. 3.828

0. 0. -1. 0.

0. 0. 0. 1.

Let’s verify a couple of things against the diagram.

* the end effector’s position is roughly at (-1.5, 3.8, 0); the calculated position is (-1.414, 3.828, 0.)
* the end effector frame’s Z axis unit vector is pointing along **F0**’s -Z Axis
* the X axis unit vector is (-0.7071, -0.7071, 0.) which checks out too since in the image it’s in the screen plane and pointing down and to the left — negative x, negative y relative to **F0**
* similarly, the Y axis unit vector is (-0.7071, 0.7071, 0.) which means it should be in the screen plane and pointing up and to the left — negative x, positive y relative to **F0**

**The code**

Code is available on [Github](https://github.com/SarvagyaVaish/Orocos-KDL-Forward-Kinematics).

Here’s the relevant code all at once:

|  |  |
| --- | --- |
| // |  |
|  | // Create a KDL kinematic Chain. |
|  | // |
|  | // A Chain is made up of Segments. Each Segment consists of a Joint and a Frame. |
|  | // The Joint indicates how the Frame moves - rotation or translation about / along an axis. |
|  | // |
|  |  |
|  | Chain kdlChain = Chain(); |
|  |  |
|  | Joint joint1(Joint::None); |
|  | Frame frame1 = Frame(Vector(0.0, 1.0, 0.0)); |
|  | kdlChain.addSegment(Segment(joint1, frame1)); |
|  |  |
|  | Joint joint2(Joint::RotZ); |
|  | Frame frame2 = Frame(Vector(0.0, 2.0, 0.0)); |
|  | kdlChain.addSegment(Segment(joint2, frame2)); |
|  |  |
|  | Joint joint3(Joint::RotZ); |
|  | Frame frame3 = Frame(Rotation::EulerZYX(0.0, 0.0, -M\_PI / 2)) \* Frame(Vector(0.0, 0.0, 2.0)); |
|  | kdlChain.addSegment(Segment(joint3, frame3)); |
|  |  |
|  | Joint joint4(Joint::RotZ); |
|  | Frame frame4 = Frame(Rotation::EulerZYX(0.0, 0.0, M\_PI / 2)) \* Frame(Vector(1.0, 1.0, 0.0)); |
|  | kdlChain.addSegment(Segment(joint4, frame4)); |
|  |  |
|  | // |
|  | // Joint Angles |
|  | // |
|  |  |
|  | JntArray jointAngles = JntArray(3); |
|  | jointAngles(0) = -M\_PI / 4.; // Joint 1 |
|  | jointAngles(1) = M\_PI / 2.; // Joint 2 |
|  | jointAngles(2) = M\_PI; // Joint 3 |
|  |  |
|  | // |
|  | // Perform Forward Kinematics |
|  | // |
|  |  |
|  | ChainFkSolverPos\_recursive FKSolver = ChainFkSolverPos\_recursive(kdlChain); |
|  | Frame eeFrame; |
|  | FKSolver.JntToCart(jointAngles, eeFrame); |