User Manual
Kinematics & Dynamics Library

Ruben Smits, Herman Bruyninckx

June 16, 2012

Chapter 1

Introduction to KDL

1.1 What is KDL?

Kinematics & Dynamics Library is a c++ library that offers

- classes for geometric primitives and their operations
- classes for kinematic descriptions of chains (robots)
- (realtime) kinematic solvers for these kinematic chains

1.2 Getting support - the Orocos community

- This document!
- The website : `http://www.orocos.org/kdl`.
- The mailinglist. **To do** (??)

1.3 Getting Orocos KDL

First off all you need to succesfully install the KDL-library. This is explained in the **Installation Manual**, it can be found on `http://www.orocos.org/kdl/Installation_Manual`

Chapter 2

Tutorial

Important remarks

- All geometric primitives and there operations can be used in realtime. None of the operations lead to dynamic memory allocations and all of the operations are deterministic in time.
- All values are in [m] for translational components and [rad] for rotational components.

2.1 Geometric Primitives

Headers

- frames.hpp: Definition of all geometric primitives and there transformations/operators.
- frames_io.hpp: Definition of the input/output operators.

The following primitives are available:

- Vector 2.1.1
- Rotation 2.1.2
- Frame 2.1.3
- Twist 2.1.4
- Wrench 2.1.5

2.1.1 Vector

Represents the 3D position of a point/object. It contains three values, representing the X,Y,Z position of the point/object with respect to the reference frame:

$$\text{Vector} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Creating Vectors There are different ways to create a vector:

```
Vector v1; //The default constructor, X-Y-Z are initialized to zero
Vector v2(x,y,z); //X-Y-Z are initialized with the given values
Vector v3(v2); //The copy constructor
Vector v4=Vector::Zero(); //All values are set to zero
```

Get/Set individual elements The operators [] and () use indeces from 0..2, index checking is enabled/disabled by the DEBUG/NDEBUG definitions:

```
v1[0]=v2[1];//copy y value of v2 to x value of v1
v2(1)=v3(3);//copy z value of v3 to y value of v2
v3.x( v4.y() );//copy y value of v4 to x value of v3
```

Multiply/Divide with a scalar You can multiply or divide a Vector with a double using the operator ∗ and /:

```
v2=2*v1;
v3=v1/2;
```

Add and subtract vectors You can add or substract a vector with another vector:

```
v2+=v1;
v3-=v1;
v4=v1+v2;
v5=v2-v3;
```

Cross and scalar product You can calculate the cross product of two vectors, which results in new vector or calculate the scalar(dot) product of two vectors:

```
v3=v1*v2; //Cross product
double a=dot(v1,v2)//Scalar product
```

Resetting You can reset the values of a vector to zero:

```
SetToZero(v1);
```

Comparing vectors With or without giving an accuracy:

```
v1==v2;
v2!=v3;
Equal(v3,v4,eps);//with accuracy eps
```

2.1.2 Rotation

Represents the 3D rotation of an object wrt the reference frame. Internally it is represented by a 3x3 matrix which is a non-minimal representation:

$$\text{Rotation} = \begin{bmatrix} Xx & Yx & Zx \\ Xy & Yy & Zy \\ Xz & Yz & Zz \end{bmatrix}$$

Creating Rotations

Safe ways to create a Rotation The following result always in consistent Rotations. This means the rows/columns are always normalized and orthogonal:

```
Rotation r1; //The default constructor, initializes to an 3x3 identity matrix
Rotation r1 = Rotation::Identity();//Identity Rotation = zero rotation
Rotation r2 = Rotation::RPY(roll,pitch,yaw); //Rotation build out off Roll-P
Rotation r3 = Rotation::EulerZYZ(alpha,beta,gamma); //Rotation build out off
Rotation r4 = Rotation::EulerZYX(alpha,beta,gamma); //Rotation build out off
Rotation r5 = Rotation::Rot(vector,angle); //Rotation build out off an equiva
```

Other ways The following should be used with care, they can result in inconsistent rotation matrices, since there is no checking if columns/rows are normalized or orthogonal

```
Rotation r6( Xx,Yx,Zx,Xy,Yy,Zy,Xz,Yz,Zz);//Give each individual element (Colu
Rotation r7(vectorX,vectorY,vectorZ);//Give each individual column
```

Getting values Individual values, the indices go from 0..2:

2.1.3 Frame

2.1.4 Twist

2.1.5 Wrench

2.2 Kinematic Structures

2.2.1 Joint

2.2.2 Segment

2.2.3 Chain

2.3 Kinematic Solvers

2.3.1 Forward position kinematics

2.3.2 Forward velocity kinematics

2.3.3 Jacobian calculation

2.3.4 Inverse velocity kinematics

2.3.5 Inverse position kinematics

2.4 Motion Specification Primitives

2.4.1 Path

2.4.2 Velocity profile

2.4.3 Trajectory