Kinematica v0.1 (pre-release)

Michael Camilleri

February 11, 2014

Abstract

This document describes the Implementation of the Kinematica Library. This library builds upon the KDL framework to provide a more generalistic set of tools: in particular, it allows arbitrary definitions of root nodes for arbitrary tree-structures and computation of Forward Kinematics and Jacobians.

Contents

1 Introduction

This document describes the Kinematica library. It is intended to serve as a reference for all users as well as future maintainers of the tool-set.

1.1 The Kinematica Library

Kinematica is a kinematics computation library built on top of KDL. It utilizes the KDL framework to represent robot segments and allow parsing of URDF files. However, it extends this framework through:

- providing Jacobian and Forward Kinematics solver for arbitrary tree structures rather than just chains.
- providing a more efficient Jacobian/Forward Kinematics solver for repetitive forms by batch processing
- allowing the arbitrary specification of the root link, which is not necessarily the same as the urdf root.

1.2 Structure of the Document

This document is divided in the following manner:

- The first section describes the notation/conventions used by the library. This is necessary for almost all users of the library.
- The second section details the public API, dependencies and general usage of the library.
- The final section, aimed at maintainers and future developers, describes the actual implementation.

2 Kinematics Theory

This chapter documents the Conventions in use by the library. Since this project builds on top of KDL it inherits most of its notations and definitions.

2.1 Robot Structure

A robot, loaded from a URDF file, is represented in Kinematica by a tree of interconnected segments. Fig. 1 illustrates the segment components.

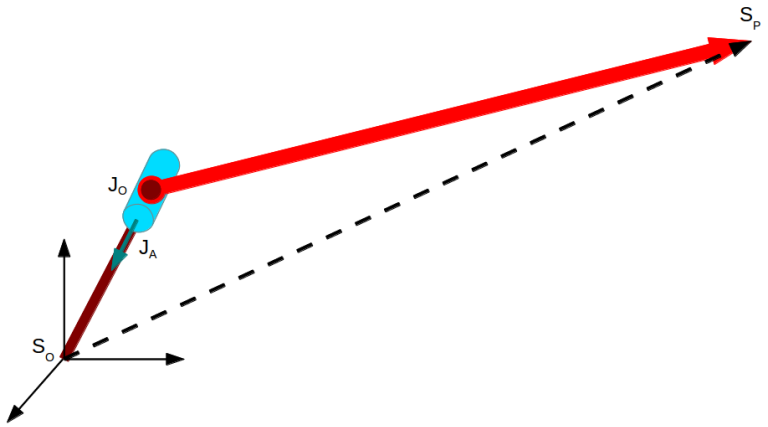

Figure 1: Definition of a Segment in Kinematica

Each segment consists of a joint (brown) and a link (red). The joint itself may be offset (fixed) from the the Segment origin (S_O), with position J_O (Joint Origin) and moves/rotates along/around a Joint Axis (J_A). The link is of a fixed length. The Segment Pose is defined as the pose of the tip of the segment (S_P) relative to the segment origin.
A robot consists of a tree of such segments. In the original URDF, by definition, a tree is built by connecting segments to the **ends/tips** of parent segments as in Fig. 2. In this case, the blue segment is taken as the root segment, with F_1 being the root frame of reference for the whole tree. Notice for example how all its children (cyan, green and red) are connected to its tip (F_2) although offset by some amount.

2.2 Redefining the Root Frame

When redefining the root frame, care must be taken regarding the direction of the segment links. Let us consider taking the red segment to be the new root link, with some frame on this segment F_U as the root frame of reference.

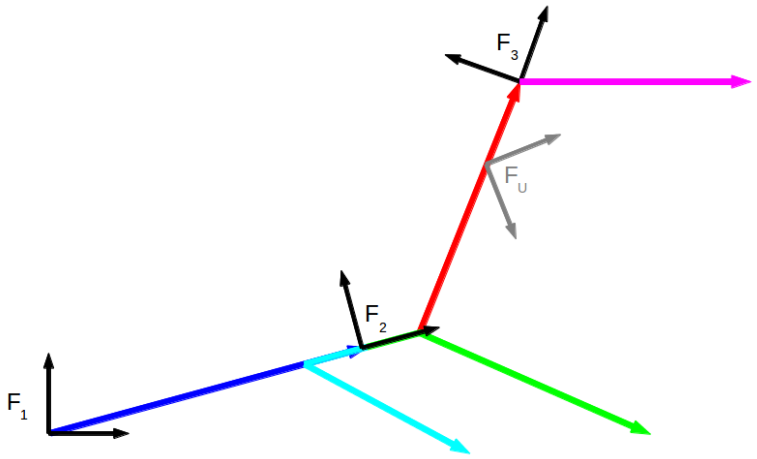

Figure 2: Description of Robot in Kinematica

In this case, the blue segment will be connected to the base of the root, with its tip actually facing towards it rather than away from it. This is intentional and is retained as such to avoid confusion with redefining the tree structure/order. The rule to keep in mind is that whatever the root frame, the segments will:

1. Always have their origin frame at their base (this will be oriented with the tip of their **original** parent).

2. Always point in the same direction as defined by the original URDF file.

3. Always be defined by the pose of their tip, whether this is in a global or local frame of reference.

2.3 Computing Forward Kinematics

There are two conventions when computing Forward Kinematics:

1. When requesting the 'global' pose of a segment, this refers to the pose of the tip in the user-defined root frame of reference (F_U in Fig 2).

2. When requesting the pose of a segment with respect to some other segment, this refers to the pose of the tip of the first with respect to the frame at the tip of the second.

2.4 Computing Jacobians

The next major component of the library is the computation of Jacobians. The library computes the First Order linear approximation to the Analytical Jacobian around a particular configuration. Consider Fig. 3.

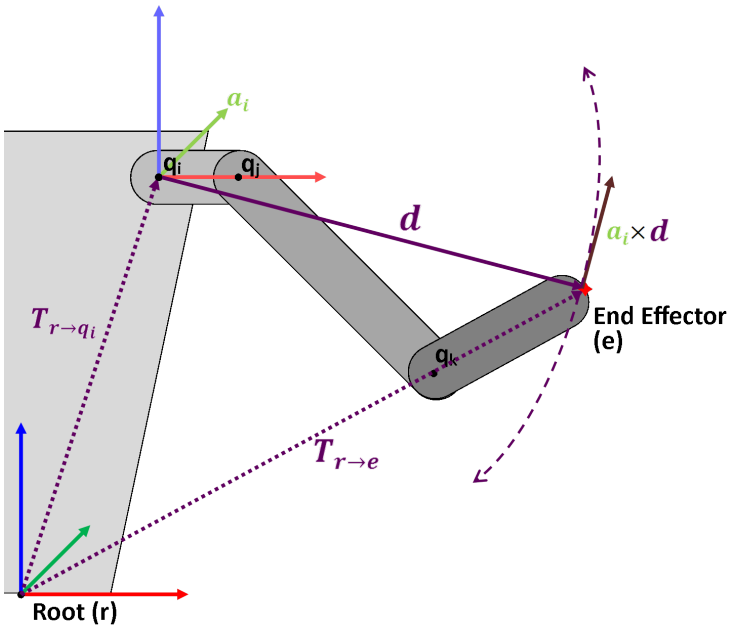

Figure 3: Derivation for the End-Effector Jacobian

Consider the computation of the Jacobian entry for joint q_j (column $J_i^{q_j}$), which is a revolving joint rotating about a_j (specified in terms of the root frame). Fixing all other joints, different angles for q_j will position e on a circular path of radius $|d_j|$. In the neighbourhood of the current value for q_j, the change in position and hence the Jacobian entry for q_j can be linearised to the cross product of a_j and d_j i.e.

$$J_i^{q_j} = a_j \times d_j \tag{1}$$

$$d = T_{r \to e} - T_{r \to q_i} \tag{2}$$

with d being defined in the root frame.

3 Using the Library

This section describes general usage of the library, including installation, building, dependencies etc...

3.1 Setup

Pre-Requisites

The Kinematica Library is designed to be as standalone a package as possible. Nevertheless, it does make use of a number of system dependencies:

- **Boost** Library for thread synchronisation
- **Orocos KDL** which provides the base infrastracture.
- **EIGEN** which is used for the matrix manipulations within the project

Moreover, although Kinematica is independent of ROS, it utilizes the ROS Catkin build system but this can be circumvented if needed by building using cmake.

Installation and Building

The library is built from the source provided. Currently the Catkin build system is used, making for an easier integration with ROS packages. Alternatively if you wish to build it as a standalone system, you must modify the CMakeLists.txt file to accomodate the normal cmake conventions.

Notes

The library tools exist within the ***kinematica*** namespace. Usage of the library requires including the ***"KinematicTree.h"*** header file but that is all.

Finally, the library is fully thread-safe in its current implementation.

3.2 Public API

This section will guide the reader through general usage of the library.

Data Types

The library consists of two main public data types. The first is a structure for setting solution parameters and is described hereunder in Table 1. The other is the actual KinematicTree class which is detailed in the next subsection.

In the interest of efficiency, the computations make use of several pre-defined parameters which are encapsulated in a structure for convenience.

Parameter	Type	Description
root_segment	std::string	The name of the segment to use as root. The empty string retains the URDF root.
root_seg_off	KDL::Frame	The position of the root frame relative to the tip of *root_segment*.
joints_update	std::vector of std::string	Specifies which joints will be modified and in which order they will be specified. If the *zero_other_joints* flag is not set, this vector must contain **all** the joints in the robot.
zero_other_joints	bool	Indicates whether unspecified joints in the *joints_update* vector will be zeroed out in any kinematics computations. Any zeroed out joints will also not factor in the Jacobian computation.
ignore_unused_segs	bool	Again an efficiency-related flag. If set, segments whose poses do not factor into any of the end-effectors specified for the Forward Kinematics/Jacobian computation are skipped in such calculations. Otherwise, all segments will be updated.
end_effector_segs	std::vector of std::string	For batch computation of Forward Kinematics/Jacobians. Lists the set of segments to which the end-effectors are attached. Each segment may be used as many times as necessary.
end_effector_offs	std::vector of KDL::Frame	The tranformation of the end-effector to the corresponding segment. The vector must be the same size as *end_effector_segs* or left empty.

Table 1: Definition of the **SolutionForm_t** structure

The KinematicTree class

The KinematicTree is designed to handle all low-level tasks involved in computing forward kinematics. Its usage will be introduced through the use of an example.

The first thing to do is to create the SolutionForm_t structure to specify the parameters we will be interested in. We keep the same root as the URDF by passing in the empty string for *root_segment*. We also desire that the root is situated at the tip of the root segment (and hence we pass in the identity transformation).

```
kinematica::SolutionForm_t solution_form;

solution_form.root_segment = ""; //!< Retain root joint
solution_form.root_seg_off = KDL::Frame::Identity();
```

We also must specify the joints we will be using: since we did not specify all joints, we set the flag to zero out unspecified joints.

```
solution_form.joints_update.push_back("joint_1"); //!< Fill
    joints
solution_form.joints_update.push_back("joint_3");
solution_form.joints_update.push_back("joint_4");
solution_form.zero_other_joints = true; //!< We have not
    specified all joints
```

Finally, we indicate the position of the end-effector we are interested in.

```
solution_form.ignore_unused_segs = false; //!< Compute for all
    segments
solution_form.end_effector_segs.push_back("segment_4"); //!< End
    -effector

KDL::Frame eff_offs;            //!< Prepare frame for the end-
    effector offset
eff_offs.p = KDL::Vector(0.5, 0.1, 0.0);  //!< Translation
    component
eff_offs.M = KDL::Rotation::EulerZYZ(0.1, 0.2, 0.05); //!<
    Rotation component
solution_form.end_effector_offs.push_back(eff_offs);
```

Having prepared the initialisation parameters we may go ahead and create a KinematicTree object. We must pass two variables to the constructor: the path to the urdf file, plus the initialisation structure we just defined.

```
kinematica::KinematicTree robot_tree("test/robot.urdf",
    solution_form);
```

Alternatively, we may call the default constructor and initialise the class at a later stage by calling the ***initKinematics(...)*** function, with the following signature:

```
bool initKinematics(const std::string &, const SolutionForm_t &)
```

Again, the first parameter is the path of the urdf describing the robot, and the second the solution parameters.
That is all as far as setup is concerned. Once the object has been initialised, Forward Kinematic computations may be invoked by calling:

```
bool updateConfiguration(const Eigen::VectorXd &)
```

The function takes one argument, the Eigen::Vector of joint values. This vector must be the same size as the joints_update vector defined previously and the joint values must be specified in the same order. This function will generate the internal computations to update the state of the tree structure based on the joint angles. Having done this, the user may request the Forward Map of the pre-defined end-effectors by using:

```
bool generateForwardMap()
```

If the function returns with success (true), the user may call

```
bool getPhi(Eigen::VectorXd &)
```

to copy the result into the passed vector. For convenience a wrapper function for the above two is also implemened which simultaneously generates and returns the forward-map vector:

```
bool generateForwardMap(Eigen::VectorXd &)
```

Similarly the user may request computation of the Jacobian, however, this can only be done if the Forward Map was previously generated. A similar repertoire of functions is available:

```
bool generateJacobian()

bool generateJacobian(Eigen::MatrixXd &)

bool getJacobian(Eigen::MatrixXd &)
```

where again, the first two generate the Jacobian from the underlying kinematics and the last function simply gives a copy of the most recent Jacobian computation stored internally.

In order to provide flexibility for the general user, arbitrary pose calculators are also implemented. Again, the updateConfiguration(...) function must be called first. In this case, the ignore_unused_segs flag must be *false*, otherwise results will be invalid (since if the requested pose is not of a segment which effects the end-effectors, it will not be updated when updating the configuration). Four variants are provided, two taking string arguments and two taking integer index arguments.

The string variants require the names of the query node (labelled *child*) and optionally the node to use as reference frame (labelled *parent*). If this is specified, the resulting frame is the pose of the tip of the child w.r.t. the frame at the tip of the parent. If the parent is omitted, the returned frame is the pose of the tip of the child w.r.t. the root frame (and **not** necessarily the tip of the root segment). In each case the last argument is the placeholder for storing the result, since the return value is used to indicate success (true) or failure (false).

```
bool getPose(std::string child, std::string parent, KDL::Frame &
    pose);
bool getPose(std::string child, KDL::Frame & pose);
```

In the interest of efficiency, the above two functions are replicated taking integer indices into the list of segments: other than that their function is identical.

```
bool getPose(int child, int parent, KDL::Frame & pose);
bool getPose(int child, KDL::Frame & pose);
```

The mapping from segment names to segment indices may be obtained by calling (after appropriate initialisation) the function:

```
bool getSegmentMap(std::map<std::string, int> &)
```

Finally, to support external traversal of the tree, the object offers functions to obtain the parent/children of a particular node. In each case, variants using string and integer input are provided, giving back names and indices respectively.

```
std::string getParent(std::string child);
int         getParent(int         child);

std::vector<std::string> getChildren(std::string parent);
std::vector<int>         getChildren(int         parent);
```