Generic Kinematics Library

Kinematica Marvin (v1.2.0)

Application Programming Interface

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. The Library is currently in its first official release and has been extensively tested.

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 definition of a segment.

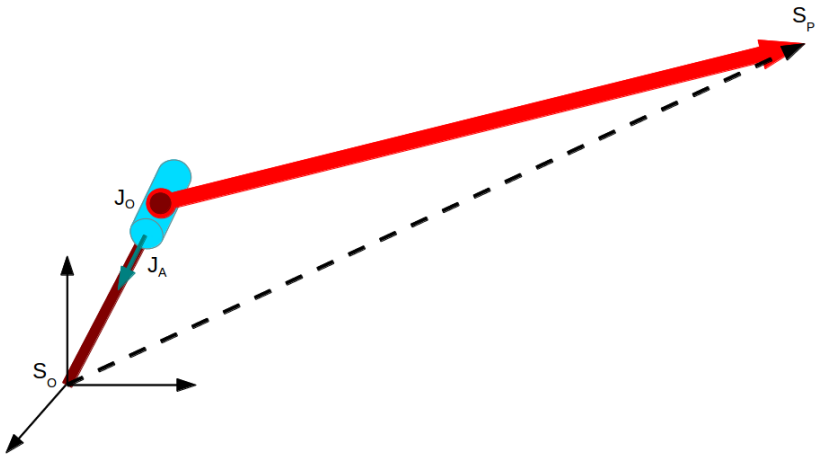

Figure 1: Definition of a Segment in Kinematica

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.

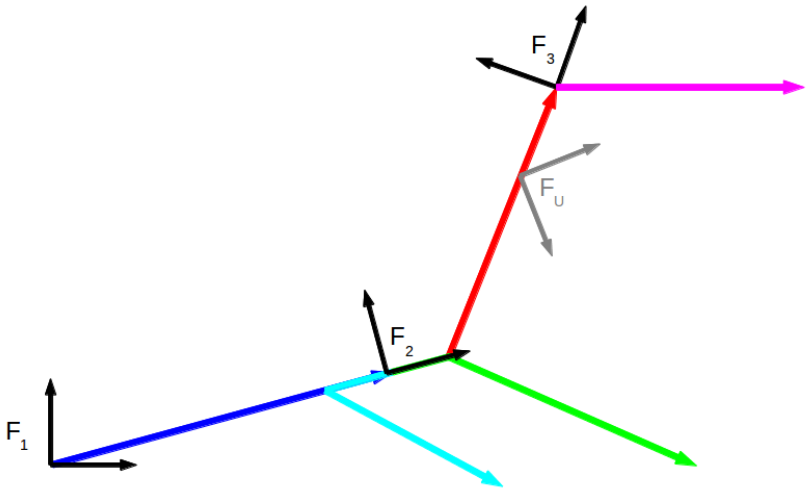

Figure 2: Description of Robot in Kinematica

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. 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 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 (including the segment which 'houses' the root), 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.
Consider the computation of the Jacobian entry for End Effector (e) with respect to joint q_j (column $J_e^{q_j}$), 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.

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

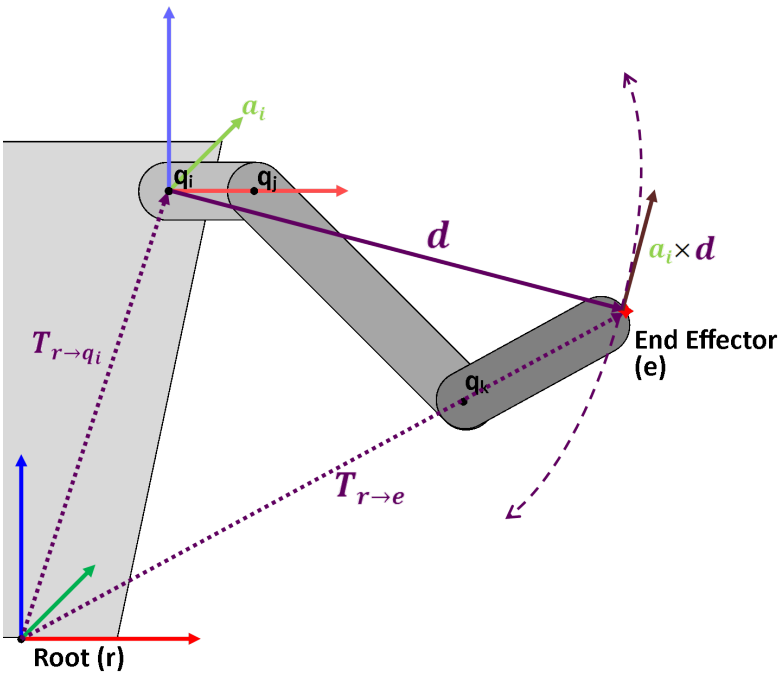

Figure 3: Derivation for the End-Effector Jacobian

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

with d being defined in the root frame.
The process is even simpler for prismatic joints. In this case, the joint axis (along which the joint slides) is directly the positional jacobian entry for that joint.

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.
- **kdl_parser**, a ROS package which is used to parse URDF files to construct the original KDL Tree.
- **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. The only limitation is that it uses the ***kdl_parser*** package which is a ROS package: however, future implementations will attempt to address this.

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 ***"kinematica/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.

Solution Parameters

The library consists of two main public data types. The first is a structure for setting solution parameters (in the interest of efficiency, we choose to pre-define these) and is described hereunder in Table 1. The other is the actual KinematicTree class which is detailed in the next subsection.

Parameter	Type	Description
root_segment	std::string	The name of the segment to use as root. The empty string defaults to 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 (*end_effector_segs*) 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 (if for example, multiple end-effectors are connected to the same segment).
end_effector_offs	std::vector of KDL::Frame	The pose of the end-effector w.r.t. the **tip** of 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. The reason for this is to provide more efficiency in cases where the same set of queries will be set (as in optimisation/control problems). Although these can be changed later on, constantly re-initialising with different parameters should be avoided: rather multiple objects should be instantiated. In this case 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[1].

```
solution_form.joints_update = {"joint_1", "joint_3", "joint_4"}; //!<
    Fill joints
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 **full** path to the urdf file, plus the initialisation structure we just defined.

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

[1]Note that the initialiser-list syntax requires C++11.

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 actual **positions** 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[2]. The result will consist of the 3D **position** of each pre-defined end-effector, in the order specified in *SolutionForm_t:: end_effector_segs*. For convenience a wrapper function for the above two is also implemented which simultaneously generates and returns the forward-map vector:

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

Similarly the user may request computation of the position Jacobian[3], however, this will **only** be valid if the Forward Map was **previously generated**. Each column in the Jacobian represents the contribution of **only** those joints and specified in *SimulationForm_t::joints_update* with the same ordering. The rows on the other hand represent the effect on each position co-ordinate (3D) of each segment in the order specified by *SolutionForm_t:: end_effector_segs*. The functions are:

[2]Note that although the name refers to Forward Maps and Φ, this is actually the result of the forward mapping and **not** the function itself: i.e. it gives the actual positions.

[3]Currently, only the position Jacobian is implemented since in theory this can be used to implemented any other kinematic Jacobian by appropriate 'trickery': in future implementations, this functionality may be extended.

```
bool generateJacobian() // Generate Jacobian
bool generateJacobian(Eigen::MatrixXd &) // ... and pass result

bool getJacobian(Eigen::MatrixXd &) // Get a pre-computed Jacobian
```

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.

We anticipate that in certain situations the state of the environment may change. To this end, a function is provided to update **only** the end-effector segments.

```
bool updateEndEffectors(const SolutionForm_t &)
```

This function takes in a *SolutionForm_t* structure, but the only parameters which need to be set are:

- **ignore_unused_segs** flag
- **end_effector_segs** vector (compulsary)
- **end_effector_offs** vector (may be optionally left empty)

The elements have the same meaning as during initialisation.

In order to provide additional 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 might be invalid if the required pose is of a segment which itself does not affect the end-effectors specified. Four methods are provided, two taking string arguments and two taking integer index arguments: the latter two are much more efficient but the results are identical.

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

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

In each case, the function requires the name/index of the query node (labelled *child*) and optionally that of 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)[4]. In each case the last argument is the place-holder for storing the result, since the return value is used to indicate success (true) or failure (false).

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);
```

[4]The major scenario where this is confusing is if the parent is the root segment itself. If the root segment name is specified the pose will be w.r.t. the tip of the root segment: if not, it will be w.r.t. the true root frame, which was specified in initialisation as an offset from the root segment tip.

4 The Kinematica Implementation

This section contains more advanced concepts and is intended for future developpers/maintainers. As such, it describes the design decisions, computations etc... within the API.

4.1 Major Design Decisions

The purpose of Kinematica is to provide a generic but efficient Forward Kinematics library for robotic applications. This has led to certain design choices:

1. The library is targeted at optimisation problems, whereby in most cases, the computations center on the same subset of end-effectors. In light of this, we choose to specify such parameters during initialisation, since this is a performance-costly process. We also do some considerable pre-processing during this stage.

2. We favour memory use over re-computations wherever applicable, on the assumption that matrix operations are resource intensive and that several end-effectors might require the same subset of data within one iteration.

3. Almost all functions indicate success or failure through a boolean return value (True/False respectively).

4. Efficiency comes second only to thread-safety and proper information hiding, so that the library can be used by non-expert software engineers. The rule is that:

   - Public Methods (accept for constructor) are THREAD-SAFE through a **common** mutex.
   - Private Methods are inherently NOT THREAD-SAFE.

   The motivation is that public functions may internally make use of multiple private functions (so locking would block). However, since the public methods do not spawn any threads, thread-safety is maintained.

The overall architecture consists of the **KinematiceTree** class which encapsulates all the functionality plus a number of structures for data organisation and some helper functions.

4.2 Data Types and Auxiliaries

Structures

There are three main data types apart from the Kinematics Class itself.

The **SolutionForm_t** structure was defined within the **Using the Library** section. Refer to subsection 3.2.

The **KinematicElement_t** structure is used to define our notion of a tree element. Due to the potential change of root, it was decided to build our own tree structure. Table 2 describes each element.

Finally, the **JointType_t** enum, defines the three types of joints supported by Kinematica:

- ***JNT_UNUSED*** : An unused/undefined joint.... typically this is a fixed joint.
- ***JNT_PRISMATIC*** : A prismatic joint (sliding)
- ***JNT_ROTARY*** : A rotary joint

Helper Functions

Eigen::Vector3d **vectorKdlToEigen**(const KDL::Vector & kdl_vec)
This Function converts between the two main position types in the library, namely from *KDL::Vector* to *Eigen::Vector3d*.

```
bool recursivePrint(kinematica::KinematicTree & robot, std::string
    node, std::string tab)
```

This is a debugging function and recursively prints the tree structure of a KinematicTree object, passes by reference as the first parameter, the current node (by name: start at the root), and the current preceding tab-width. The last parameter is used to provide a more intuitive display in characters, and in the first call to the function should be an empty string.

The logic behind the *tab* field is as follows:

- If it is the empty string, then this is the root and the node name is printed.
- Otherwise, print the 'elbowed' arrow, taking care to avoid printing the vertical line multiple times.

Now depending on the number of children:

- If no children, this is the base case and we return.
- If not, then increase indent and:
  - If only one child, we append just a space (so that the recursive call will do the full elbow)

Table 2: Definition of the **KinematicElement_t** structure

Parameter	Type	Description
segment	KDL::Segment	Our copy of the KDL::Segment, used to compute local (within segment) forward kinematics computation based on joint values.
needed	bool	Efficiency flag: indicates whether the contribution of this segment is necessary for the queries the library will be set.
parent	int	Index into the vector of Kinematic Elements pointing to the parent (of the new tree structure) of this node. For the Root node, will have the value -1.
from_tip	bool	Flag for indicating the traversal of segments (due to change of roots). True signifies that we are connected to *parent* from its tip (as is normal): otherwise we are connected to its base.
to_tip	bool	Flag for indicating the traversal of segments: True indicates that we are moving towards the tip of the segment and False v.v.
joint_type	kinematica:: JointType_t	Defines the type of joint we are dealing with (refer to the enum definition for more information).
joint_index	int	Index for this segment's joint into the configuration array specified by the user. Note that this (and all joint-related members) refer to the **original** joint associated with this *segment*.
tip_pose	KDL::Frame	The latest update of the pose of the **tip** of this segment w.r.t. the **root frame**.
joint_pose	KDL::Frame	This is the pose of the **base** of the segment as a whole (S_O in Fig. 1) with respect to the **root frame**.
joint_origin	Eigen::Vector3d	The latest update of the **position** of the joint's origin (J_O in Fig. 1) in the **root frame**.
joint_axis	Eigen::Vector3d	The Joint Axis of rotation (Fig. 1, J_A), oriented w.r.t. the root frame.
child	std::vector of int	Indices (within the vector of Kinematic Elements) of (any) children of this node: these are the children in the **new** tree structure.

– If more children, then we need to append a vertical line so that the remaining children will still be connected.

Finally, we recurse on all the children, removing the vertical line from the last one (since there will be no trailing siblings under it).

4.3 Class Attributes

All the class attributes are declared private to enforce information hiding and thread-safety: where necessary, access to the underlying variables is through getter/setter functions. These are displayed in table 3.

Table 3: The Attributes of **KinematicTree**: KE_t signifies KinematicElement_t

Parameter	**Type**	**Description**
robot_tree_	std::vector <KE_t>	Holds the segments making up the tree structure. Numeric indexing provides an efficient implementation (over maps).
segment_map_	std::map <std::string, int>	Maps from segment names to indices into *robot_tree_* for convenience.
zero_undef_jnts_	bool	Flag with same meaning as *SolutionForm_t::zero_other_joints*.
num_jnts_spec_	int	(For Error-checking) The size of the expected configuration (joint-values) vector.
eff_segments_	std::vector <int>	The list of end-effector segments (same as *SolutionForm_t::end_effector_segs* but transformed into numeric indices).
eff_seg_offs_	std::vector <KDL::Frame>	The set of end-effector offsets (may be empty: same as *SolutionForm_t::end_effector_offs*).
forward_map_	Eigen::VectorXd	Internal copy of the most recent Φ vector containing the poses of the end effectors.
jacobian_	Eigen::MatrixXd	Internal copy of the most recent Jacobian computation.
member_lock_	boost::mutex	Lockable mutex for thread-safety and synchronisation.

4.4 Initialisers

This subsection introduces the initialisation-related functions.

Constructors

A default and two initialisation constructors are provided.

```
KinematicTree(void);
KinematicTree(const std::string &, const SolutionForm_t & );
KinematicTree(const KDL::Tree &, const SolutionForm_t & );
```

The initialisation constructors take two parameters.

1. The path to the URDF-file containing the robot description in the first case and the pre-initialised KDL::Tree in the second.
2. The structure of solution parameters.

which it then passes to the ***initKinematics(...)*** function. All constructors are guaranteed to initialise the object in a stable state[5].

initKinematics

These functions provide a thread-safe wrapper for a private common initialise function (to allow flexibility in initialising from a urdf file or a KDL::Tree). Their signature is similar to the initialisation constructors (above), with success or failure being indicated through the return value (True/False respectively).

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

initialise

This is the private common wrapper for initialising the object from a KDL::Tree, and the Solution Form parameters.

```
bool initKinematics(const KDL::Tree &, const SolutionForm_t &);
```

In order, the function:

1. Cleans up the internal data members.
2. Calls ***buildTree(...)*** to build the internal tree structure, potentially changing the root.
3. Sets the joint ordering that will be used using ***setJointOrder(...)***.
4. Calls ***setEndEffectors(...)*** to settle end-effector constants/indices.

If any of the sub-methods fail, all data members are cleaned up and the function returns. Success or failure is indicated.

[5]By this we mean that the object has all its members initialised to default values such as empty vectors, reset flags etc.

buildTree

This is a private (non-thread-safe) function, which aims to build the tree structure using the user-defined segment/frame as root.

```
bool buildTree(const KDL::Tree &, std::string, std::map<std::string,
    int> &);
```

The three arguments are:

1. The KDL::Tree structure previously loaded
2. The name of the segment to use as root
3. Placeholder for building the joint map (from joint name to the index into the *robot_tree_* vector corresponding to the segment to which the joint pertains)

In order to ensure that the passed segment does indeed exist, we take a detour by first requesting the whole map of segments from the KDL::Tree and then attempting to *find()* the segment (rather then just dereferencing it). The tree is then built in a recursive manner through the ***addSegment(...)*** function (see below).

addSegment

This is a recursive private (non-thread-safe) function which traverses the **original** tree and converts it into the new-rooted structure.

Consider the simplified robot description, replicated hereunder as Fig. 4. Assume that initially the root segment is the blue one, with F_1 being the associated root frame (the base of the segment), and that we now wish the red segment to be our new root, with the actual frame at F_U. In starting from the (new) root node, we need to consider three cases:

- The **Root** segment: then all segments attached to it will become its children: this includes its original children, its original parent as well as its original siblings (children of the same parent). In this example, this amounts to all the segments!
- We are moving **along the original tree direction** (such as for the magenta segment): then the structure remains the same (parents remain parents, children remain children).
- We are moving **against the original tree direction**: in this case both the original parent and the siblings become children in the new tree. For example, moving from red to blue, the parent (originally blue) becomes a child, as do the cyan and green segments since these are siblings.

The function has the following signature:

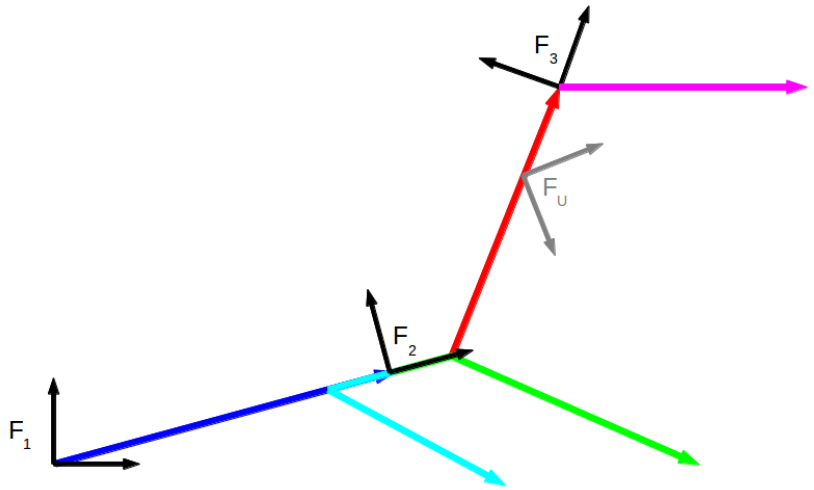

Figure 4: Description of Robot in Kinematica

```
bool addSegment(KDL::SegmentMap::const_iterator, int, int &, bool,
    bool, const std::string &, std::map<std::string, int> &)
```

with the following parameters:

1. Iterator to the next element in the **original** KDL::Tree
2. Index into the new *robot_tree_* pointing to the parent to which this segment will be child.
3. Placeholder for the position of this segment (once it is created) within the *robot_tree_* (since the parent will need references to its children).
4. Indicator whether this segment is connected to its **new** parent through the parent's tip (True) or base (False).
5. Indicator whether we are moving towards the tip of this segment (True) or towards its base (False).
6. Name of the original root (for checking)
7. Reference to the joint map (for updating)

The function first creates a new Kinematic Element with default values and pushes it into the *robot_tree_* vector. This ensures that:

- The new root segment is **always** at index 0 (for consistency).
- The segment ordering is such that a child **never** appears before its (new) parent. This is necessary for the efficient and correct updating of Forward Kinematics.

It also records the index in the *segment_map_* and updates the *joint_map* to associate the joint name with its original segment. The three scenarios are identified through the use of the direction flags (with the root using a special invalid combination 'true/false'). The function then iterates through the **original** children of this segment (and optionally the parent) and calls itself recursively on each. Finally, it assigns the returned child index to the respective segment (due to conditions 1 and 3). Checks are carried out to ensure that we do not dereference the parent of the original root (which does not exist).

setJointOrder

This function iterates through all the segments in *robot_tree_*, specifying whether this joint will be controlled or not (through the *joint_type* paramater) and if it will be controlled, what type it is and the index into the array the user will use to specify the joint values. This is built using the previously generated *joint_map* and the vector of joint names in *SolutionForm_t::joints_update*. The signature is:

```
bool setJointOrder(const std::vector<std::string> &, bool, const std
    ::map<std::string, int> &);
```

with the parameters:

- Vector of joint names in the order they will be specified
- Flag to indicate whether unspecified joints should be zeroed out: if not set and not all the joints are specified the function fails.
- The joint map previously generated.

setEndEffectors

This is the last function to be called during initialisation: it is a private non-thread-safe function taking the *SolutionForm_t* specification as parameter, and returning an indication of success/failure.

```
bool setEndEffectors(const SolutionForm_t &);
```

It initialises the root segment's tip pose (since this will be constant) and then populates the vector of segment indices which will be used for computation of end-effector forward kinematics and jacobians. It makes use of the ***recurseNeedFlag()*** function to specify that this segment and all its parents up to the root are required for computation. It also pre-allocates memory for the forward map and jacobian matrices.

recurseNeedFlag

As the name implies this is a recursive function, with the aim of traversing up the tree from a segment all the way to the root, setting the *needed* flag for the corresponding segment. It takes in one parameter (the index of the segment in the *robot_tree_* vector) and calls itself recursively until the root is reached.

```
bool recurseNeedFlag(int);
```

isInitialised

Coming Soon

4.5 Updaters

The following set of functions are expected to be called repeatedly during iterations to update the state of the kinematic tree.

updateConfiguration

Coming Soon

generateForwardMap

Coming Soon

computePhi

Coming Soon

generateJacobian

Coming Soon

computeJacobian

Coming Soon

4.6 Accessors

These functions provide access to the underlying data structures. Usually, they must be called after appropriate initialisation/updating for the results to be correct.

getJacobian

Coming Soon

getPhi

Coming Soon

getPose

Coming Soon

getSegmentMap

Coming Soon

getParent

Coming Soon

getChildren

Coming Soon

A Known Limitations

1. There is currently no collision-detection support

B Version History

Version	Library Changes	Documentation Changes
0.0.1	• First Iteration of Kinematica	• First Documentation
1.0.0 (Marvin)	• First Official Release • Deprecated the ***KinematicsType_t*** enum • Added Support for initialisation directly from KDL::Tree object. • Extensively Tested and Debugged	• Clarified some concepts in the Public API • Documenting Detailed API for developers (in progress)
1.1.0	• Added support for prismatic joints: fixed joints should be simply ignored by the user	• Clarified the changes in the documentation
1.2.0	• Added ability to change the configuration of end-effectors as needed.	