# Robotics Library: An Object-Oriented Approach to Robot Applications

Markus Rickert and Andre Gaschler

*Abstract*— **We discuss the architecture and software engineering principles of the Robotics Library (RL). Driven by requirements of robot systems, research projects, industrial applications, and education, we identify relevant design requirements and present an approach to manage hardware and real-time, provide a user-friendly, object-oriented interface to powerful kinematics and dynamics calculations, and support various platforms. After over ten years of development that started in 2004 and evaluating many variants of the architecture, we discuss the design choices for the components of the library in its current version.**

## I. Introduction

Developing software to control intelligent robots is challenging in multiple ways: Robot setups typically involve wildly different pieces of hardware running on different platforms, some of them with hard real-time communication. Today's motion planning algorithms and rigid-body dynamics calculations are highly non-trivial, require efficient implementations, but should be easy to use for the average programmer. In short, designing a programming library for robotics is a difficult undertaking. Some complexity can be hidden by modern object-oriented programming concepts. Other more conflicting design goals can only be achieved by careful trade-offs. For this reason, even though there are many existing robotics frameworks following different choices [1], [2], [3], [4], we present a pure C++ library approach covering the whole range from hardware abstraction to motion planning, named *Robotics Library* (RL).

### A. Design Principles

To help position our work among the many existing frameworks, we first discuss the main design principles applied throughout the library, some of which show notable differences to related approaches.

*a) Pure library with a single API:* Interfaces are consistent throughout the components of the library, and all classes inherit from defined base classes. Contrary to many other robotics frameworks, RL is not a middleware and it is independent from any distributed communication approach that may be used.

*b) Consistent and complete implementation:* All important algorithms in the main robotics field, including kinematics, dynamics, trajectory generation, and path planning, are available and use identical notation and interfaces. Robot kinematics and dynamics support generic joint types in a tree structure and are implemented using spatial vector algebra [5]. Sensor and robot hardware drivers, as well as various types of motion planning algorithms, are unified in a class hierarchy (Section II).

*c) Library reuse:* Well-designed libraries from related fields, such as collision detection or visualization, are made available through wrappers. Especially in collision detection, several high-quality libraries already exist with different compromises in performance, precision, and advanced functionality (such as distance queries or raycasts). RL provides access to them under a common scene graph data structure and encapsulates their functions in an abstraction layer (Section II-E).

*d) Platform independence:* The pure ISO C++ implementation using cross-platform libraries compiles on a wide range of systems, ranging from an embedded QNX, a Debian-based Raspberry Pi, to a PC running Windows.

Apart from these software design principles, it is worth noting that RL is open source[1] and licensed under a permissive 2-clause BSD license. It is therefore free for noncommercial and commercial applications. Open source enables debugging the whole software stack from user interfaces down to real-time hardware control. A short summary of an early version of the Robotics Library is given in [6].

### B. Related Work

The Robot Operating System (ROS) is probably the most active middleware and software framework in robotics research, indicated by frequent commits, an active mailing list, and citations [1]. ROS is mostly motivated by the integration of service robot systems and has a strong emphasis on a peer-to-peer topology between many processes (called nodes) and a large common set of tools for software integration (such

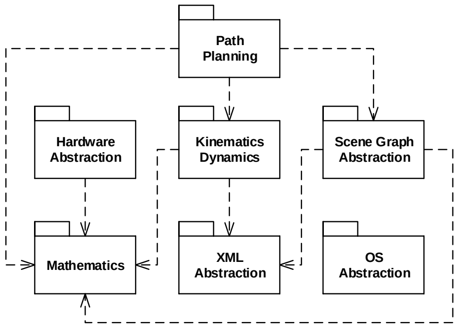


Fig. 1: Overview of the Robotics Library's packages and their interdependencies.

Markus Rickert is with fortiss, An-Institut Technische Universität München, Guerickestr. 25, 80805 Munich, Germany

Andre Gaschler is with Robotics and Embedded Systems, Department of Informatics, Technische Universität München, Boltzmannstr. 3, 85748 Garching b. München, Germany

[1]`https://www.roboticslibrary.org/` (paper describes v0.7)

as build system, data and topology visualization, plotting, or documentation). In the early days, the design of ROS was mainly guided by the PR2 mobile manipulator [7]. While this design gained great popularity and was successfully adapted to a much wider range of humanoids, mobile robots, and aerial vehicles, some specifications that were not covered in the original design—especially real-time requirements—are hard to add to the existing middleware concept. For this reason, developers currently discuss plans to support embedded platforms and real-time in a different API version [7]. Core functionality inside nodes relies on a number of existing frameworks such as Orocos' kinematics library [8]. In contrast to the ROS collection of packages, RL is a homogeneous library with full real-time support.

Among the earliest approaches to object-oriented path planning software is the Components for Path Planning library [9], which has been refactored to a component framework [10], but is no longer maintained. For real-time robot control, object-oriented, model-based designs [11] and component-based frameworks [4] were presented. Later, the OpenRAVE architecture was proposed [12], which is geared toward autonomous robot motion planning and control. In contrast to OpenRAVE, RL implements robot and sensor drivers. More recently, RobWork [13] was presented, which likewise provides simulation and control features, as well as an extensible graphical user interface. Compared to the RobWork library, RL offers real-time support. For collision-free path planning, the Open Motion Planning Library [3] provides a large collection of state-of-the-art algorithms, but no means for executing these paths on physical robots. Further related are robot simulation frameworks, such as Gazebo [14], V-REP [15], and Webots [16].

## II. Approach

In this section, we discuss the software design principles of the library and its components. The package architecture (Fig. 1) comprises several domain-independent mathematics and generic features together with robotics-specific packages, where high-level packages depend on more basic ones. Our discussion roughly follows the package architecture from bottom to top.

### A. Numerical Library

A well-designed numerical mathematics library has great influence on all other parts of a robotics library. Almost all robotics-specific algorithms in kinematics, dynamics, path planning, trajectory generation, and collision avoidance need efficient vector and matrix operations as well as numerically stable calculations of advanced matrix and eigenvalue decompositions. Choosing powerful, numerically stable, and efficient numerical routines therefore has a great impact on the overall performance of the whole library. Apart from efficiency, other requirements of the API are readability, user-friendliness, and consistency.

Through the version history of the Robotics Library, we had four mathematical branches in total, which finally converged into the current implementation using Eigen [17]. Eigen offers fast and readable code through the use of expression templates and explicit vectorization (e.g., SSE4, AVX512). RL makes a number of extensions to the native Eigen types and adds additional features, especially to the quaternion implementation. These features include power and exponential functions of quaternion types, as well as conversion to angular velocity and acceleration. For smooth interpolation of robot rotations, RL implements the slerp function and its derivative as defined by [18]. When tangency conditions are given for a rotation interpolation, the squad function can satisfy a smooth interpolation with valid quaternions. RL implements the squad function and the cubic quaternion interpolation as defined by [19].

### B. Mathematical Functions

Many types of geometric computation are used frequently in robotics applications, ranging from basic 3D transformations, various types of rotations, to trajectory interpolation and spatial vector calculations. In order to have a clear separation of concerns, we implement generic geometric functions that are independent from a robot's kinematic or dynamic model in the mathematics component, which is kept free from any software dependencies.

The Robotics Library makes two important choices that characterize the design of its mathematics components: First, trajectory generation takes a strongly algebraic approach where trajectories are represented as piecewise algebraic functions, allowing exact interpolation and differentiation. Second, all rigid-body dynamics functions are formulated in terms of spatial vectors, which allow particularly concise formulation and highly efficient computation.

*1) Algebraic function formulation:* It is a major challenge in trajectory generation to avoid discontinuities in position, velocity, and acceleration. Only smooth trajectories with continuous derivatives enable safe, fast, and precise robot motion. However, when defining a path in the operational space or switching between joint space or operational motion, robotics frameworks commonly resort to simple sampling, filtering, or approximation by low-level polynomials.

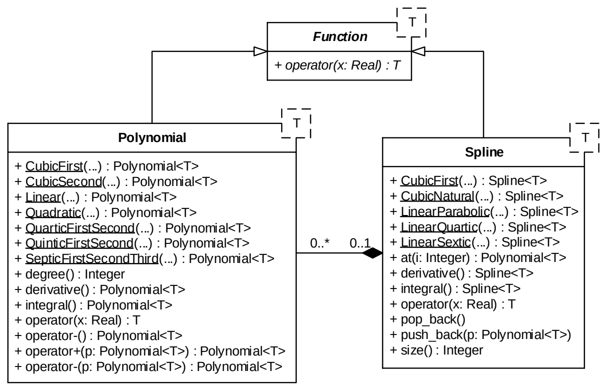

T
Function
+ operator(x: Real) : T
T
Polynomial
+ CubicFirst(...) : Polynomial<T>
+ CubicSecond(...) : Polynomial<T>
+ Linear(...) : Polynomial<T>
+ Quadratic(...) : Polynomial<T>
+ QuarticFirstSecond(...) : Polynomial<T>
+ QuinticFirstSecond(...) : Polynomial<T>
+ SepticFirstSecondThird(...) : Polynomial<T>
+ degree() : Integer
+ derivative() : Polynomial<T>
+ integral() : Polynomial<T>
+ operator(x: Real) : T
+ operator-() : Polynomial<T>
+ operator+(p: Polynomial<T>) : Polynomial<T>
+ operator-(p: Polynomial<T>) : Polynomial<T>
0..*
0..1
T
Spline
+ CubicFirst(...) : Spline<T>
+ CubicNatural(...) : Spline<T>
+ LinearParabolic(...) : Spline<T>
+ LinearQuartic(...) : Spline<T>
+ LinearSextic(...) : Spline<T>
+ at(i: Integer) : Polynomial<T>
+ derivative() : Spline<T>
+ integral() : Spline<T>
+ operator(x: Real) : T
+ pop_back()
+ push_back(p: Polynomial<T>)
+ size() : Integer

Fig. 2: Class hierarchy of mathematical functions. A *Function* is a vector-valued mapping that can be evaluated efficiently and stably, including its derivatives (only part of the definition and methods are shown).

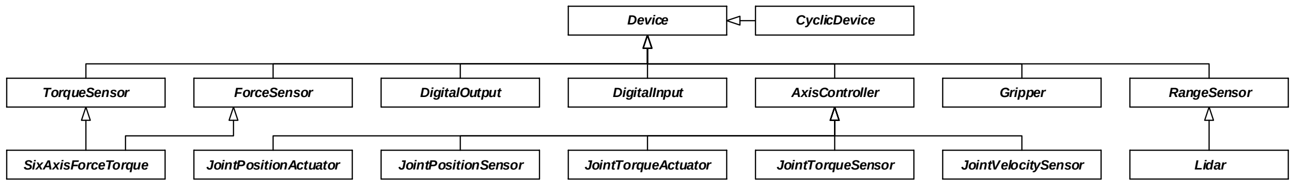


Fig. 3: Excerpt of classes for hardware abstraction. Similar devices offer common methods such as reading distance values for a *RangeSensor* or reading a manipulator's position values for each axis of a *JointPositionSensor*.

RL avoids this approximation and offers a framework for constructing piecewise, vector-valued algebraic functions, allowing to represent all types of trajectories on an algebraic level. As shown in Fig. 2, the abstract *Function* class defines a mapping from a real value (usually time) to a vector (usually, but not limited to a joint space or operational space configuration), whose values and derivative values can be evaluated efficiently. Concrete classes are implemented for all basic types of motion: Polynomials for joint and operational space positions, quaternion polynomials for rotation and operational space orientations, and motion along circular segments. Piecewise functions and splines (piecewise polynomials) are organized in the shape of a composite pattern, allowing the construction of complex trajectories both in joint space and in operational space. The benefit of this approach is twofold: First, continuity and smoothness can be guaranteed and verified exactly and for multiple orders of derivatives, including acceleration and jerk. Second, trajectories are formulated on a semantic, abstract level, on the same level as a typical computer aided design environment, allowing easy integration with advanced robot programming user interfaces.

In the mathematics component, the *Function* classes offer generic implementations that do not require any kinematic model. In particular, static functions are available for constructing algebraic functions for given boundary conditions, including higher-order (quaternion) spline interpolation and motion along circular segments. When such a function is to be run as a trajectory for a particular robot, the kinematics package in Section II-D will solve the inverse kinematics mapping for trajectories in operational space, check for kinematic limits, and allow execution by evaluating the piecewise trajectory function with respect to the robot's control cycle.

*2) Spatial vector formulation:* Rigid-body dynamics have been implemented by many robotics frameworks [3], [11], [4], [8]. However, RL chooses a different formulation from these and uses spatial vector algebra, as proposed by Featherstone [5], [20]. A spatial vector is defined as a six-dimensional vector that may either be a motion vector representing a velocity or acceleration, or a force vector representing momentum or impulse. Spatial vectors are not part of Euclidean space, with the dot product being defined only between a motion and a force vector. In this notation, a spatial transformation can be expressed as a 6-by-6 matrix. While dynamics algorithms appear rather convoluted when written using 4-by-4 transformation matrices, spatial vectors enable a very concise formulation of the recursive Newton-Euler algorithm in a single formula for all types of joints. The $n$ degrees of freedom of a joint are defined by a 6-by-$n$ matrix for all general types of joints [20, p. 78f], including revolute, prismatic, spherical, and helical joints.

Besides brevity in notation and high generality for all joint types, the spatial vector implementation has the important benefit of higher computational efficiency. Featherstone shows [20, pp. 201–204] that this implementation minimizes the number of floating point operations for forward and inverse dynamics computation for kinematic chains.

## *C. Hardware Abstraction Layer*

Support for controlling various types of hardware devices is a major feature of a robotics framework. Today, robot setups rely on several types of sensors, including cameras, range sensors, or force-torque sensors. On the actuator side, industrial robots and grippers use a wide range of different protocols, including various fieldbuses and custom Ethernet protocols. Open-source driver implementations are rather scarce. In many cases, precise protocol documentations are not part of the manual and not open to the public.

*1) Hierarchical hardware driver implementation:* Hardware devices lend themselves well for an object-oriented, hierarchical implementation. While devices of similar types, such as actuators, force-torque sensors, grippers, or cameras, share a common set of basic features, individual devices may provide additional features specific to a certain manufacturer [6, pp. 104f]. These common interfaces are organized as a class hierarchy in an object-oriented hardware abstraction layer as shown in Fig. 3.

For devices with an extremely diverse and fine-grained feature set, structuring them into multiple layers is always an option. For instance, a humanoid can be modeled as a device with flat access to all position-controlled joints, but may also be designed as a class with access to individual arms, legs, etc. A robot manipulator can provide an interface for position control, but also give access to single motors with interfaces including access to individual components and their temperature readings and control values.

On top of the hierarchy is the abstract *Device* class, defining methods common to all devices: Opening and closing the connection to the hardware device, starting and stopping operations, as well as the *step* method, which performs all communication to and from the device.

Contrary to other robotics frameworks, which choose to encapsulate the communication with a hardware device completely and implement asynchronous function calls and

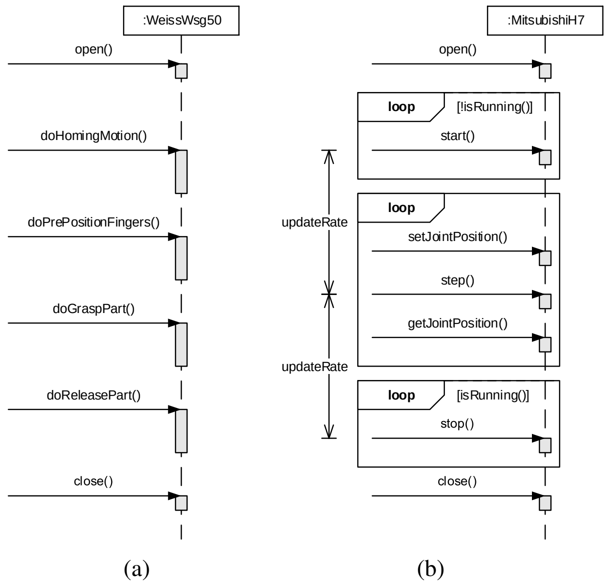


Fig. 4: States of (a) *Device* and (b) *CyclicDevice*. The latter exchanges all messages to and from the device in the *step* function, which needs to be called in a real-time loop.

callbacks on data reception, RL chooses to avoid any type of hidden concurrency. Concerning the hardware component, it is an important design choice of the library not to create threads or processes (apart from the actual real-time thread abstraction class meant for this purpose). Concurrent behavior is a critical aspect in interfacing hardware with real-time constraints and should therefore be in the hand of the programmer. On the contrary, RL requires the application to call the *step* method of the device periodically within its control cycle.

The states and state transitions of a device object are shown in Fig. 4. Communication to a newly constructed device is initiated by the *open* method call. For real-time devices, successive *step* method calls are necessary within real-time constraints. Its operations begin after a *start* method call and end with *stop*. Importantly, all communication with the device is performed by this method and is buffered in the object. The programmer accesses cached sensor data or actuator target positions through getter and setter methods.

This hardware interface design allows all types of thread implementations, including platform-specific real-time threads. However, the design expects the programmer to actively choose how real-time method calls are guaranteed and does not hide this important choice. A programmer can design his application as a single process with one thread or individual threads of different cycle times for each hardware device. He can also choose to have one process for each device together with a shared memory communication based on his favorite API. To balance computational load, the programmer can move devices and processes performing heavy calculation to a separate computer and use basic TCP/UDP communication, a fieldbus, or one of the many available middlewares.

*2) Operating system abstraction:* Before the introduction of the new C++-11 standard with interfaces for threads, mutual exclusion, condition variables, clocks, and asynchronous programming, RL provided abstract thread, mutex, semaphore, and timer classes for common pthread, Windows, RTAI, and Xenomai implementations. In the latest version, RL is built on top of the C++-11 standard, but provides extensions for setting process and thread priorities, as well as a standard-compatible interface for RTAI and Xenomai's native interface.

RTAI and Xenomai are open-source kernel extensions to Linux and handle real-time scheduling [21]. With both originating from the same project, RTAI behaves differently in that it intercepts interrupt notifications directly and avoids some interrupt handling overhead [21]. Some native real-time operating systems such as QNX offer regular POSIX threads through the pthread interface. All of these real-time frameworks are fast and reliable enough for all implemented devices, with TCP round-trip delays well below 100 µs, as measured by [21]. With this abstract thread class, programmers can develop applications that compile on multiple real-time operating systems or real-time kernel extensions.

## D. Kinematics and Dynamics

Modeling the kinematic structure is an integral part of controlling a robot system. Together with the robot's dynamic properties, this includes the calculation of link frames based on joint values, the Jacobian matrix and its derivative, as well as mass matrix, Coriolis vector, and gravity compensation.

Common algorithms include the recursive Newton-Euler algorithm for inverse dynamics and the Articulated-Body algorithm for forward dynamics. Classical Denavit-Hartenberg notation is used to describe systems with a number of revolute and prismatic joints that move the robot's links. Different formulas are required for each joint type in order

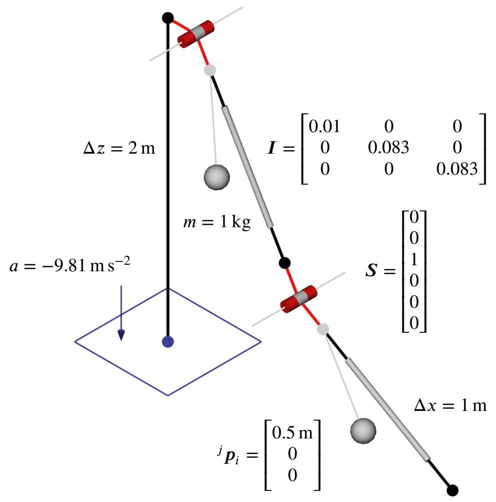

$\Delta z = 2\,\mathrm{m}$
$\boldsymbol{I} = \begin{bmatrix} 0.01 & 0 & 0 \\ 0 & 0.083 & 0 \\ 0 & 0 & 0.083 \end{bmatrix}$
$m = 1\,\mathrm{kg}$
$a = -9.81\,\mathrm{m\,s^{-2}}$
$\boldsymbol{S} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$
$\Delta x = 1\,\mathrm{m}$
${}^{j}\boldsymbol{p}_i = \begin{bmatrix} 0.5\,\mathrm{m} \\ 0 \\ 0 \end{bmatrix}$

Fig. 5: Kinematics and dynamics of a double pendulum. Model description in graph form with world (*blue*), rigid body (*gray*), and frame vertices (*black*) as well as fixed translation/rotation (*black*) and revolute joint edges (*red*).

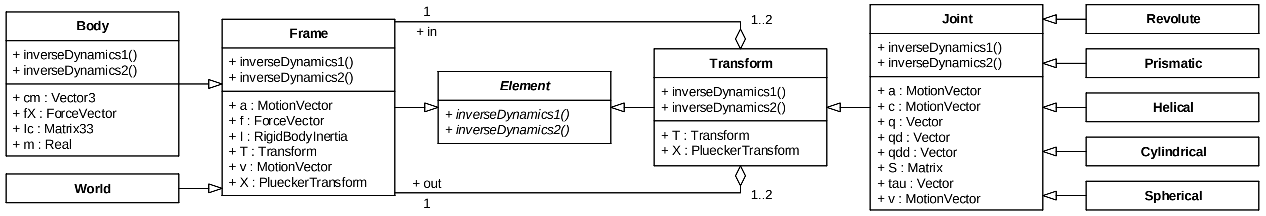


Fig. 6: Overview of classes for kinematics and dynamics representation (only class members relevant for Newton-Euler algorithm are shown). Bodies and joints in a tree-like structure represent a model and its current state.

to describe these algorithms. Object-oriented modeling of multibody systems [22], [11] can be used to model joints and other transformation components as objects that influence elements such as frames, velocities, accelerations, or forces.

By combining these two approaches, spatial vector algebra [20] can be used to model joint types such as helical, cylindrical, planar, spherical, or 6-DOF joints with a single notation. Algorithms are modeled in a way that these transformation objects modify corresponding input and output values. Joints and fixed transformations between bodies can be seen as edges in a graph, while bodies, the world reference, and intermediate frames represent vertices in this graph (Fig. 5).

As an example, the recursive Newton-Euler algorithm uses two iterations: (i) Forward propagation of velocities and accelerations, (ii) back propagation of forces. In the object-oriented implementation, this is represented by two functions *inverseDynamics1()* and *inverseDynamics2()* implemented differently for each object (Fig. 6). The first iteration calculates current velocities $\vec{\boldsymbol{v}}$ and accelerations $\vec{\boldsymbol{a}}$ in the body frames based on joint velocities $\dot{\boldsymbol{q}}$ and accelerations $\ddot{\boldsymbol{q}}$. It then determines the net forces $\vec{\boldsymbol{f}}_i^{\mathrm{B}}$ in body coordinates generated by these accelerations and a body's rigid body inertia $\vec{\boldsymbol{I}}$. Plücker transforms $\boldsymbol{X}$ define the spatial transformation from frame $i-1$ to $i$. The directions of free motion of a joint are modeled by a matrix $\boldsymbol{S}$ and its derivative $\dot{\boldsymbol{S}}$. The second iteration/function propagates the forces $\vec{\boldsymbol{f}}$ generated by the individual links from the frame at the end effector back to the base and maps them to the torques $\boldsymbol{\tau}_i$ in the individual joints of the robot system.

Other recursive algorithms such as the Articulated-Body algorithm for forward dynamics are modeled in a similar fashion. In order to calculate the Jacobian or mass matrix for a given state, multiple queries to these algorithms can be used to create a full representation of a state. Tree-like structures are easily supported.

In combination with the muscle Jacobian, [23] shows an application of this part of RL for tendon-driven robots.

## *E. Scene Graph Abstraction*

Geometry data is necessary for 3D visualization (Fig. 8), collision checking, distance computation, and raycasting. While geometry is often modeled as a boundary representation (B-rep) in CAD programs, visualization and the listed queries typically require a polygon representation. For exchange of polygon data, the Robotics Library uses VRML (Virtual Reality Modeling Language), a common format supported by many CAD programs and 3D editors. Together with triangle meshes it offers support for basic primitives such as boxes, spheres, cylinders, and cones that can be utilized in collision detection engines. Named nodes can be used to reference individual robot models and their respective moving bodies.

A scene representation consists of a number of moving bodies that can be grouped into models to map to robots and obstacles (Fig. 7). In the scene graph, the bodies are not arranged in a tree-like structure. Instead, bodies are located in the world frame to support all kinds of connections to kinematics and dynamics models, physics simulations, and sensor input. In physics engines, bodies are connected by constraints such as joints or springs that can be removed or may break during interaction. Each body consist of a number of geometric shapes with static transformations. In order to update a robot's geometry model given a joint configuration, the individual frames are calculated by the robot's corresponding kinematic representation.

File import and 3D visualization is implemented using the Open Inventor API, an object-oriented scene graph implementation. VRML is supported with excellent loading performance, as it is an extension to the native Inventor file format. In order to group bodies into models, an additional scene description file specifies the names of individual models and bodies in the VRML file. Separate VRML files can be used for visualization and collision detection in order to

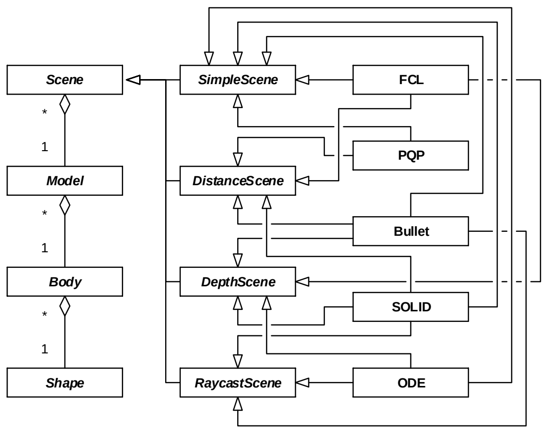


Fig. 7: Class overview of the scene graph abstraction with supported collision checking libraries.

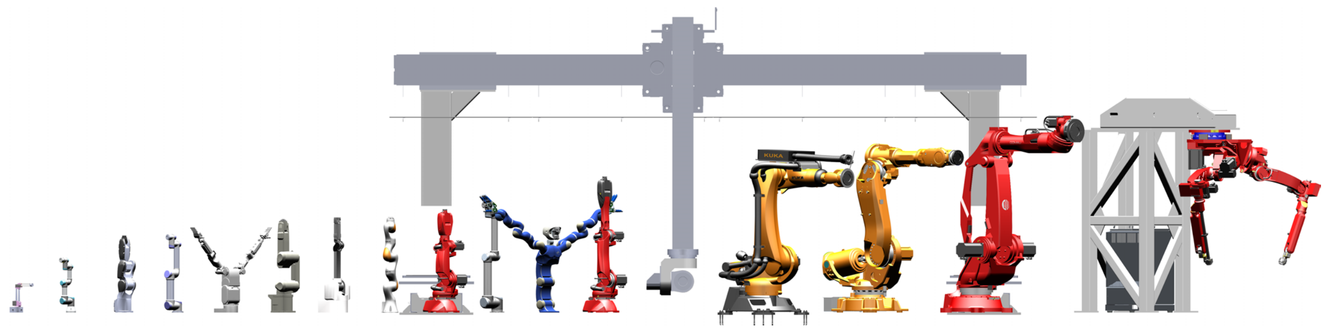


Fig. 8: RL supports visualization and kinematics of a wide range of robots. It can support a combination of various joint types, common tree-like kinematics such as dual-arm manipulators, as well as manipulators on top of holonomic platforms.

model primitive shapes such as boxes, spheres, or convex hulls for improved performance [24].

In order to compare different collision engines, the API defines interfaces for simple collision/distance/raycast queries and penetration depth computation. Features include queries between shapes, bodies, models, or the whole scene.

A branch of the Robotics Library explored replacing VRML with COLLADA, as this offers a description for a *visual* (detailed graphics) and a *physics* (collision shapes and dynamic properties) scene in a single file. COLLADA 1.5.0 even offers support for B-rep geometry descriptions. The support for exporting geometry files in this format is however still limited in current 3D software programs. While visual scenes in version format 1.4.0 can be exported in some programs, support for modeling and exporting physics scenes is still very poor. In addition, the official COLLADA DOM API only provides a basic C++ object representation of the COLLADA XML schema and leaves the majority of the work up to the user. This branch of RL was used in [25] to simulate a tendon-driven robot.

### F. Path Planning

Finding a collision-free path from one robot configuration to another is a common task in robotics. Various planning algorithms have been developed over the years, with sampling-based approaches such as Probabilistic Roadmaps (PRM) and Rapidly-Exploring Random Trees (RRT) and their variations. A comparison of various planning algorithms and their performance using RL can be found in [26].

Path planning requires a kinematic representation of the robot together with a geometric model and an engine for collision detection (Fig. 9). In order to determine if a state is colliding, the kinematics map an $n$-dimensional joint configuration to frames for the geometry in the three-dimensional workspace and the collision engine computes the result.

The kinematics also define a metric space as a combination of the manifolds of its individual joints. A prismatic or revolute joint with upper and lower bounds is represented by $\mathbb{R}^1$, a revolute joint with no limits however by $\mathbb{S}^1$. Spherical joints represent the real projective space $\mathbb{RP}^3$ and the correct manifold for free-flying objects is given by $\mathbb{R}^3 \times \mathbb{RP}^3$. These metrics have to be considered when calculating a global distance function and when interpolating between configurations.

Apart from the complexity of the geometry and the performance of the collision engine, nearest neighbor calculation is the most expensive operation in a lot of planning algorithms. Linear search becomes increasingly expensive with the numbers of vertices in a tree or graph. Structures such as $k$-dimensional tree only scale to kinematics with approximately 20 DOF and require special adjustments for metrics other than Euclidean [27]. With similar restrictions regarding the number of DOF, Geometric Near-Neighbor Access Trees (GNAT) only require a global distance function and are an alternative for other metrics [28]. Parallelization (OpenMP) can be used to some extent to improve the performance of these expensive queries.

Sampling-based planners rely on a proper pseudorandom number generator [29]. Apart from uniform sampling, various other sampling techniques have been introduced—especially for PRM-like planners—and can be chosen for such an algorithm. Similarly, different strategies are available

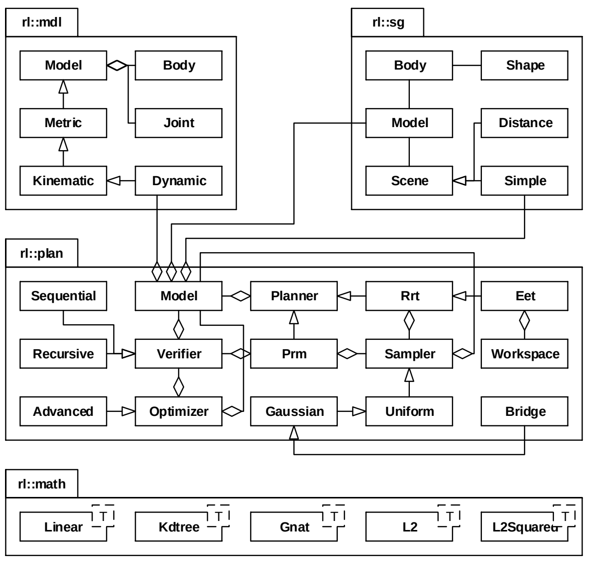
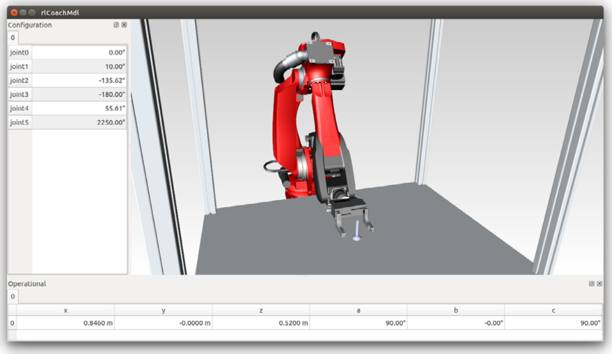


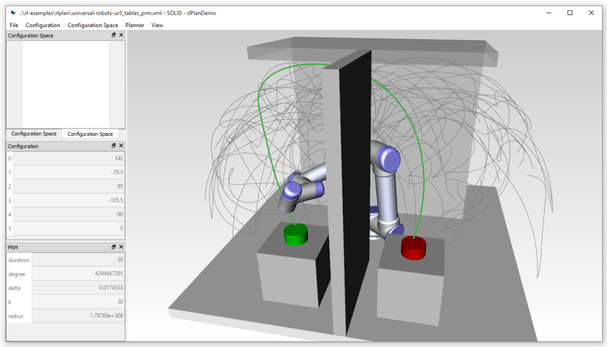
Fig. 9: Excerpt of classes relevant to path planning, with models for collision detection and kinematics calculation, as well as data structures for nearest neighbor calculation.



| 0 | |
|---|---|
| joint0 | 0.00° |
| joint1 | 10.00° |
| joint2 | -135.62° |
| joint3 | -180.00° |
| joint4 | 55.61° |
| joint5 | 2250.00° |

| | x | y | z | a | b | c |
|---|---|---|---|---|---|---|
| 0 | 0.8460 m | -0.0000 m | 0.5200 m | 90.00° | -0.00° | 90.00° |

Fig. 10: *rlCoachMdl* included in RL for visualization of forward and inverse kinematics. It provides a TCP port for remote update of joint configurations and body/shape frames.



| | |
|---|---|
| 0 | 142 |
| 1 | -79.5 |
| 2 | 95 |
| 3 | -105.5 |
| 4 | -90 |
| 5 | 0 |

| | |
|---|---|
| duration | 30 |
| degree | 4294967295 |
| delta | 0.0174533 |
| k | 30 |
| radius | 1.79769e+308 |

Fig. 11: *rlPlanDemo* is a demo for evaluation of different path planners and visualization of their results. It supports different collision engines and can load XML scenarios.

for quick verification of graph edges. After a solution path was generated, the user can select among a number of optimization approaches to minimize path length.

## III. Applications

The development of RL is mainly driven by the requirements of research and industrial projects. The following sections present a small selection of previous use cases. Videos of example applications can be found online[2].

### A. Basic Visualization of Kinematics and Geometry

One of the simplest demo programs included in the open-source release for combining kinematics and geometry in a basic robot visualization is shown in Fig. 10. It uses the Qt framework to provide a simple GUI where the user can experiment with forward and inverse kinematics.

The program can load a scene definition specified in the XML format of Section II-E together with multiple XML kinematics definitions for the robots included in the scene. It will then create a corresponding `rl::sg::Scene` for visualization and `rl::mdl::Kinematic` models for kinematics. After a change of joint positions it performs forward kinematics to update the corresponding link frames and will use these for updating the matching bodies in the geometric scene.

The demo application also provides an open TCP socket that can be used to update robot configurations and individual body/shape matrices. The hardware abstraction part of RL includes a corresponding joint position actuator and sensor for visualizing motions before execution on actual hardware.

### B. Basic Path Planning Application

For testing motion planning algorithms and comparing their performance, the demo program shown in Fig. 11 can be used. It can load an XML scenario definition defining a path planning algorithm and its parameters, together with a kinematic model and scenes for visualization and collision checking. The Qt application uses a design based on the one shown in Fig. 9 of Section II-F and adds additional features such as a visualization of the configuration space and an approximation of the swept volume of a solution path. The collision engine used in the running program can be switched and specified as an optional command-line parameter to the application.

[2]`https://www.youtube.com/roboticslibrary`

### C. Distributed Real-Time Task-Based Control

In the human-robot cooperation system shown in Fig. 13, collisions are avoided by distance sensing and online control in the nullspace of manipulation tasks. The tracking of the worker's motion was performed using a commercial optical tracking system that was delivered with an SDK running on Windows. The real-time control of the robot was implemented in a single process on a real-time Linux system (Fig. 12).

The task controller developed in [30] uses a `rl::sg::solid::Scene` instance to calculate the minimum distances between obstacles in the environment and the moving bodies of the worker. The nullspace calculations are based on a `rl::mdl::Kinematic` model of the robot and add a task-based control structure on top of this. The joint positions of the Mitsubishi RV-6SL robot are accessed and updated in 7.11 ms over an

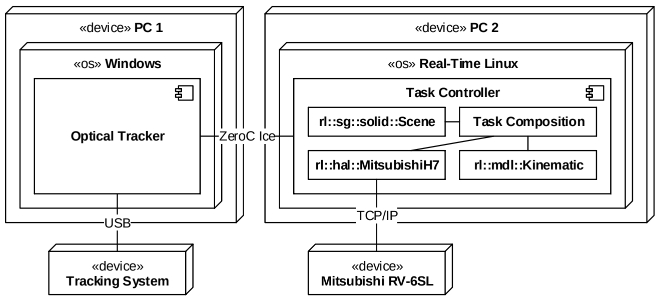


Fig. 12: Overview of the setup for the human-robot cooperation application with the optical tracking system running on a separate system than the real-time control of the robot. ZeroC Ice was chosen as the middleware for this use case.

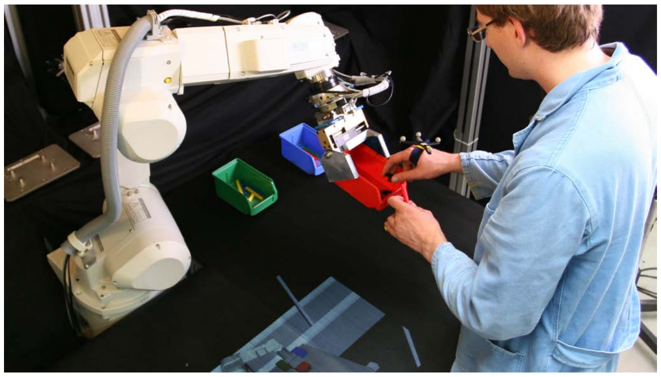


Fig. 13: Human-robot cooperation scenario with collision avoidance via an optical tracking system. Different task priorities such as keeping the orientation of the box and tracking the worker's hand are observed.

implementation of the robot's custom real-time interface via `rl::hal::MitsubishiH7`. For testing purposes, it can be easily replaced with `rl::hal::Coach` to use the visualization shown in Fig. 10.

For the communication between the two processes in this use case, remote procedure calls provided by the cross-platform middleware ZeroC Ice were used. The optical tracker on the first PC sends updates of the trackers on the worker's body to the scene graph on the second PC. The choice of the communication channel between the two PCs is up to the decision of the programmer based on the use case requirements.

## IV. Conclusion and Future Work

After over ten years of development, we have designed a software architecture for robotics suitable for a wide range of applications. In contrast to related work, we follow a platform-independent and pure library approach, which fulfills several design properties relevant to robotics, such as real-time control, feature-rich geometric algorithms, and powerful robot kinematics. In the future, we plan to integrate our work in the area of constraint-based task programming [31] and to develop a user-friendly API for it.

## References

[1] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: An open-source robot operating system," in *Proc. of the IEEE Intl. Conf. on Robotics and Automation, Workshop on Open Source Software in Robotics*, 2009.

[2] D. Brugali and P. Scandurra, "Component-based robotic engineering (Part I) [Tutorial]," *IEEE Robotics & Automation Magazine*, vol. 16, no. 4, pp. 84–96, 2009.

[3] I. A. Sucan, M. Moll, and E. Kavraki, "The open motion planning library," *IEEE Robotics & Automation Magazine*, vol. 19, no. 4, pp. 72–82, 2012.

[4] H. Bruyninckx, P. Soetens, and B. Koninckx, "The real-time motion control core of the Orocos project," in *Proc. of the IEEE Intl. Conf. on Robotics and Automation*, vol. 2, 2003, pp. 2766–2771.

[5] R. Featherstone, "A beginner's guide to 6-D vectors (Part 2) [Tutorial]," *IEEE Robotics & Automation Magazine*, vol. 17, no. 4, pp. 88–99, 2010.

[6] M. Rickert, "Efficient motion planning for intuitive task execution in modular manipulation systems," Dissertation, Technische Universität München, 2011.

[7] B. Gerkey *et al.*, "Why ROS 2.0?" http://design.ros2.org/articles/why_ros2.html, accessed Mar. 1st, 2017.

[8] R. Smits *et al.*, "KDL: Kinematics and Dynamics Library," http://www.orocos.org/kdl, accessed Mar. 1st, 2017.

[9] M. Strandberg, "Robot path planning: An object-oriented approach," Ph.D. dissertation, School of Electrical Engineering, Royal Institute of Technology, 2004.

[10] D. Brugali, W. Nowak, L. Gherardi, A. Zakharov, and E. Prassler, "Component-based refactoring of motion planning libraries," in *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, 2010, pp. 4042–4049.

[11] R. Hopler and M. Otter, "A versatile C++ toolbox for model based, real time control systems of robotic manipulators," in *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, vol. 4, 2001, pp. 2208–2214.

[12] R. Diankov and J. Kuffner, "OpenRAVE: A planning architecture for autonomous robotics," Robotics Institute, Tech. Rep. CMU-RI-TR-08-34, 2008.

[13] L.-P. Ellekilde and J. A. Jorgensen, "RobWork: A flexible toolbox for robotics research and education," in *Proc. of the Intl. Symposium on and German Conf. on Robotics*, 2010, pp. 1–7.

[14] N. Koenig and A. Howard, "Design and use paradigms for Gazebo, an open-source multi-robot simulator," in *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, vol. 3, 2004, pp. 2149–2154.

[15] E. Rohmer, S. P. Singh, and M. Freese, "V-REP: A versatile and scalable robot simulation framework," in *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, 2013, pp. 1321–1326.

[16] O. Michel, "Webots: Professional mobile robot simulation," *Intl. Journal of Advanced Robotic Systems*, vol. 1, pp. 39–42, 2004.

[17] G. Guennebaud, B. Jacob, *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010, accessed Mar. 1st, 2017.

[18] E. B. Dam, M. Koch, and M. Lillholm, "Quaternions, interpolation and animation," Department of Computer Science, University of Copenhagen, Tech. Rep. DIKU-TR-98/5, 1998.

[19] D. Eberly, "Quaternion algebra and calculus," http://www.geometrictools.com/Documentation/Quaternions.pdf, 2010.

[20] R. Featherstone, *Rigid Body Dynamics Algorithms*. Springer, 2008.

[21] A. Barbalace, A. Luchetta, G. Manduchi, M. Moro, A. Soppelsa, and C. Taliercio, "Performance comparison of VxWorks, Linux, RTAI and Xenomai in a hard real-time application," in *Proc. of the IEEE-NPSS Real-Time Conf.*, 2007, pp. 1–5.

[22] A. Kecskeméthy and M. Hiller, "An object-oriented approach for an effective formulation of multibody dynamics," *Computer Methods in Applied Mechanics and Engineering*, vol. 115, no. 3–4, pp. 287–314, 1994.

[23] M. Jäntsch, S. Wittmeier, K. Dalamagkidis, and A. Knoll, "Computed muscle control for an anthropomimetic elbow joint," in *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, 2012, pp. 2192–2197.

[24] A. Gaschler, Q. Fischer, and A. Knoll, "The bounding mesh algorithm," Technische Universität München, Germany, Tech. Rep. TUM-I1522, 2015.

[25] S. Wittmeier, M. Jäntsch, K. Dalamagkidis, M. Rickert, H. G. Marques, and A. Knoll, "CALIPER: A universal robot simulation framework for tendon-driven robots," in *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, 2011, pp. 1063–1068.

[26] M. Rickert, A. Sieverling, and O. Brock, "Balancing exploration and exploitation in sampling-based motion planning," *IEEE Transactions on Robotics*, vol. 30, no. 6, pp. 1305–1317, 2014.

[27] J. Ichnowski and R. Alterovitz, "Fast nearest neighbor search in SE(3) for sampling-based motion planning," in *Proc. of the Intl. Workshop on the Algorithmic Foundations of Robotics*, 2014, pp. 197–214.

[28] S. Brin, "Near neighbor search in large metric spaces," in *Proc. of the Intl. Conf. on Very Large Data Bases*, 1995, pp. 574–584.

[29] M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 1, pp. 3–30, 1998.

[30] C. Lenz, M. Rickert, G. Panin, and A. Knoll, "Constraint task-based control in industrial settings," in *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, 2009, pp. 3058–3063.

[31] N. Somani, M. Rickert, A. Gaschler, C. Cai, A. Perzylo, and A. Knoll, "Task level robot programming using prioritized non-linear inequality constraints," in *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, 2016, pp. 430–437.