The POS Library: a Highly-Customisable Coordinate System Library for C++

○ Francisco Jesús ARJONILLA GARCÍA (Shizuoka University)
Yuichi KOBAYASHI (Shizuoka University)

We present a coordinate system library based on the POS axioms defined in IEEE 1872-2015 Standard Ontologies for Robotics and Automation. This new library supports local kinematic trees and connections to other coordinate system frameworks such as ROS. The library is extendable with custom general coordinates by template meta-programming and custom implementations of coordinate systems by dynamic polymorphism. The POS library is released under the GNU Lesser General Public license.

1. Introduction

The physical nature of robotic systems implies some form of manipulation of coordinate systems during design stage and implementation of control algorithms. The relative locations of each link in the robot is described by a kinematic tree, which defines the position of a coordinate system fixed to each link in relation to the coordinate system fixed to the corresponding parent link. When considering an environment with multiple robots or robots with high redundancy, the complexity of the kinematic tree can quickly become an obstacle in designing robotic systems, specially in spatial (three-dimensional cartesian space) environments due to the dependency of translations with orientations. The problem may be tackled with basic trigonometry, matrices or quaternions, but there is a risk of introducing algorithmic errors due to the high number of very similar non-commutative operations. Moreover, coordinates and transformation functions are often expressed under the same mathematical representation. For example, points and translations of points are indistinguishable under vector representation. The same goes for orientations and rotations under matrix or quaternion representation. Frequently, the result is a frustrated engineer that cannot find the right combination of operations.

Coordinate system libraries facilitate the manipulation of coordinates in complex kinematic trees by offering simple and easy-to-use application programming interfaces (API) with abstractions that guide the composition of transformation functions even in complex environments. Many of these libraries are designed without considering the specific application to robotics. However, robotic environments have extra requirements compared to other applications of coordinate systems such as computer graphics. Robot controllers must be reliable, robust, run in real-time, as well as having support for distributed environments and be certifiable, to name a few.

There are many libraries capable of coordinate transformation that are ready for production use. For example, the libraries `QTransform` [1], `Cairo` graphics [2] and `OpenGL` [3] are widely-used graphic libraries that support coordinate system trees in planar and spatial space. `PyProj` library [4] for Python, which is a programming language often used in robotics research, is focused on cartographic projections and geodetic transformations. Specialized tools such as `Matlab` [5] have explicit support for robot kinematics in the toolboxes `Robotics Systems` and `Navigation`. In C++ there are many relevant libraries available, with `Robotics Library` [6] being one of the most advanced in addition to including support for many other robotics-related tools. Furthermore, the library `tf2` for the `ROS` ecosystem [7] stands out for its support for distributed environments and has language bindings for Python and C++. While all these libraries have been used successfully in many applications, there is much room for improvement. This paper presents a new library that aims at solving many of the issues that the authors have encountered while working with the above libraries.

In this paper we present a coordinate systems library, called the POS library, for the C++ programming language with usability in mind and extended capabilities compared to existing libraries. The `POS` library is based on the POS axioms defined in IEEE 1872-2015 (*IEEE Standard Ontologies for Robotics and Automation*) [8]. It comes with an internal units library that mandates the correct use of magnitudes at the language level, as well as having support for spatial kinematic trees and transparent connection to other coordinate system libraries, both local and distributed (*i.e.* MuJoCo [9] and ROS [7]). Other generalized coordinates such as joint spaces are supported by template-metaprogramming, whilst support for custom connections to other libraries, even in the same kinematic tree, are supported through a virtual interface.

2. Basics

We now explain the basic concepts of the library. In this paper, we refer to a developer that codes a program using the POS library as a *user* of the library.

Based on the POS ontology [8] convention, the POS library is based on two basic notions. First, coordinates are elements of some generalized coordinate space. When operating with two or more coordinates, they must belong to the same coordinate system. The template type `coord_qty` is an alias of the underlying raw coordinate type. Template types

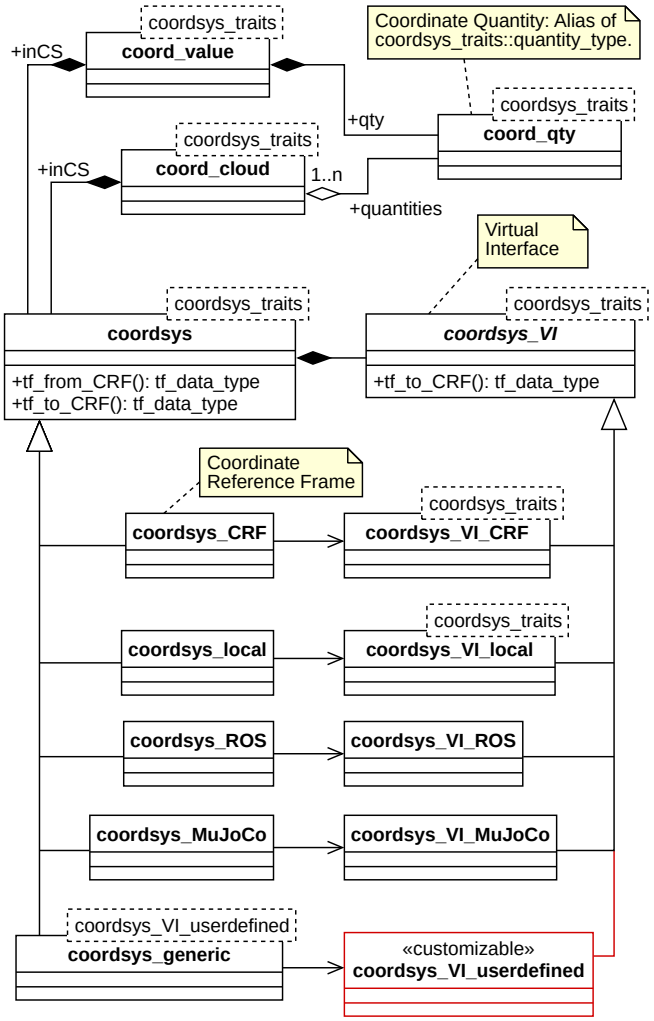

Fig. 1: Class diagram of coordinate systems and the virtual interface in POS library. See main text for more details.

`coord_value` and `coord_cloud` are coordinate quantities and lists of coordinate quantities associated to a specific coordinate system. When there is a need to operate with coordinates associated to different coordinate systems, they must first be expressed in the same coordinate system by transforming either to the coordinate system of the other. Coordinate transformation functions fulfill this role and are the second basic notion.

Coordinate systems As shown in Figure 1, the template type `coordsys` represents a coordinate system, with the template parameter specifying the class of generalized coordinates. The following classes of generalized coordinate traits are included: points (*a.k.a.* position), orientations and poses. Additionally, planar points and planar poses are included as experimental interfaces together with the corresponding coordinate bridges to spatial coordinates. The POS library includes a units library to support the values of these generalized coordinates. The use of units when specifying planar or spatial coordinates is

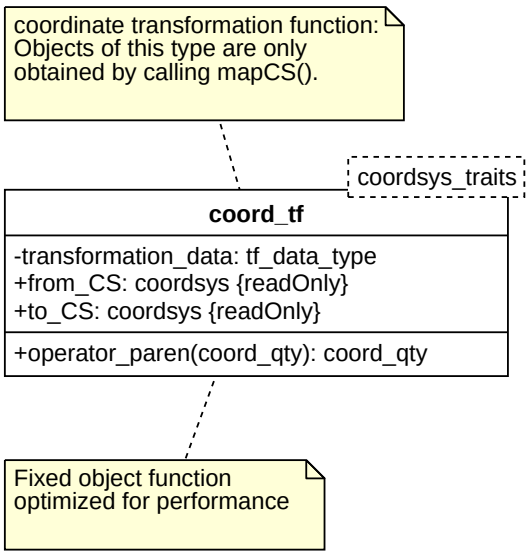

Fig. 2: Class diagram of coordinate transformation functions in POS library.

mandatory to reduce the probabilities of making mistakes when mixing magnitudes expressed in different units.

Transformation data Coordinate transformation functions (Figure 2) are only generated by the function `mapCS()`, which queries the origin and destination coordinate systems for the transformation data. `mapCS()` provides highly-efficient, self-contained, static transformation functions for repeated invocation. It combines the transformation data of the coordinate system of each coordinate, such that every transformation function involves a transformation to the reference frame followed by a transformation from the reference frame to the target coordinate system. Therefore, the role of coordinate systems is only to provide data for creating transformation functions and any changes in the kinematic tree requires regeneration of the transformation function. The functions `tf_to_CRF()` and `tf_from_CRF()` in class `coordsys` query the transformation data to and from the reference frame, respectively. The user does not have to call these functions directly, but rather call `mapCS()` instead. CRF stands for the Coordinate Reference Frame, which is a unique and common coordinate system from which all other coordinate systems are placed.

For example, the transformation data of a spatial point is a translation vector; the conversion functions from (to) the joint space of a specific manipulator to (from) spatial pose corresponds to forward (inverse) kinematic functions.

Virtual interface The ability to support connectivity to other frameworks is based on a virtual interface `coordsys_VI` internal to `coordsys`. Actually, `coordsys` is a wrapper of `coordsys_VI` with value semantics. This way, the kinematic tree is transpar-

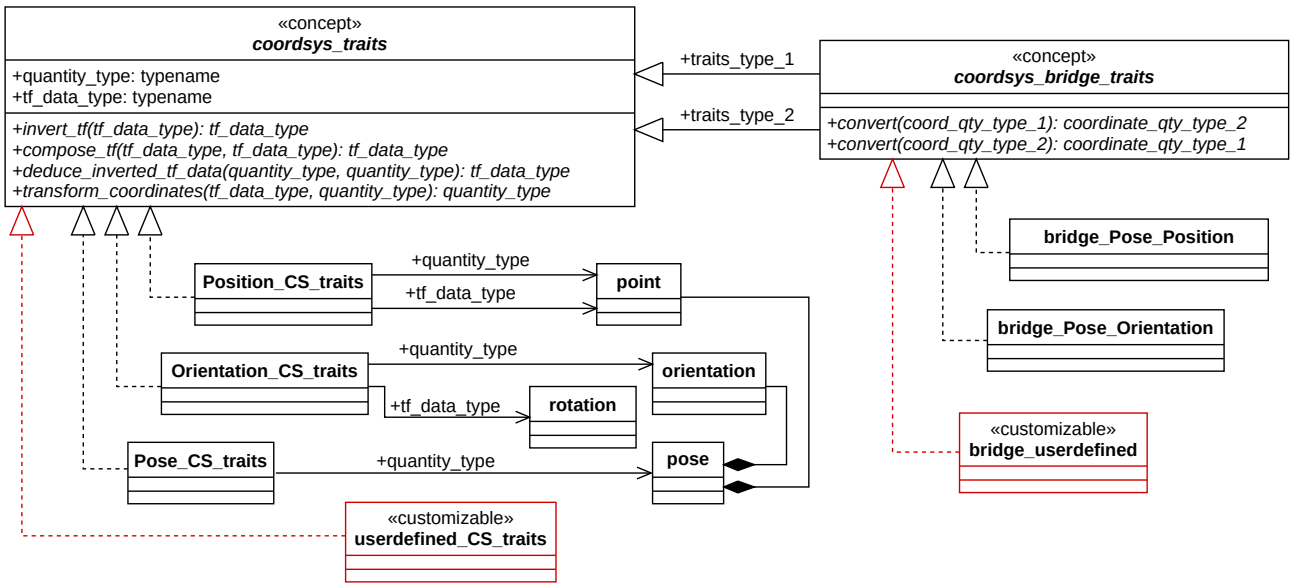

Fig. 3: Class diagram of coordinate system traits and IEEE 1872-2015 conforming coordinate types. In red, an example of customized generalized coordinates as a realization of trait concepts.

ent to how and where the transformation data is produced.

The POS library includes several implementations of `coordsys_VI`: `coordsys_VI_CRF` stands at the top level of the kinematic tree by always producing an identity transformation. The singleton object `CRF` instantiates the top level coordinate system in any kinematic tree and receives a special treatment in the library. It is noteworthy that `CRF` objects represent the same global coordinate system even in different processes, devices or networks.

Non-special implementations of `coordsys_VI` are `coordsys_VI_local`, which implements a local tree by querying the transformation data of the parent coordinate system and composing it with its relative transformation data, `coordsys_VI_ROS`, which implements a ROS node to query and provide transformation data in the ROS ecosystem as geometry messages, and `coordsys_VI_MuJoCo`, which queries the kinematic data of a mjData structure. Each implementation of `coordsys_VI` is specified in its own particular way. Local coordinate systems are defined by a reference to the parent coordinate system and the relative transformation data to that parent coordinate system. ROS coordinate systems are defined by a URI to the master node and the name of a topic. Mujoco coordinate systems are defined by pointers to mjModel and mjData structures, and the name of a body, site, *etc.*.

Thus, transparent use of multiple realizations of the virtual interface is possible, *e.g.* combining a local kinematic tree with ROS nodes in a MuJoCo simulation. The `coordsys_VI` virtual interface is also useful to create non-regular transformations, such as non-affine spatial transformations.

Bridges Another feature of the POS library is the ability to combine different classes of coordinate systems in a single kinematic tree, *e.g.* embed a planar coordinate system into a spatial one seamlessly. Bridges are instances of two coordinate systems with different coordinate traits and two overloaded functions `convert` with the actual conversion operations. Having instances of coordinate systems allows to make the conversion anywhere in both of the coordinate space by adjusting the transformation functions of the coordinate system instances. This interface was chosen for increased flexibility, specially when considering all the features of the POS library as a whole.

Example of use Here we present a short example of using the POS library to convert coordinates using a bridge.

```
// Position Coord.Sys. at (4, 0, 0.03)
pos_coordsys CSpt = pos_coordsys_local
      (CRF, {4_m, 0_m, 3_cm});
// Planar position CS at origin
planar_pos_coordsys CSPlpt =
    planar_pos_coordsys_local(CRF,{});
// Instantiate bridge at origin
bridge_Position_PlanarPosition bridge;

// Generate transform function
auto tf = mapCS(CSPlpt, bridge, CSpt);

// Transform a planar point to spatial
planar_pos_point plpt
                 {CSPlpt, {5_m, 0_m}};
position_point result = tf( plpt );
// result.qty = {1_m, 0_m, -3_cm});
```

3. Customization

Besides local trees and connection to MuJoCo simulations and ROS nodes, user-defined providers

of transformation data are available by realizing the abstract template class `coordsys_VI`. To enable modifications to the data of these virtual interfaces, the template class `coordsys_generic` is derived from the class `coordsys` and holds a reference to the user-defined virtual interface, that is implicitly convertible to a reference to the abstract class `virtual interface`.

Each `coordsys` object deals with a single class of generalized coordinates. The type `coordsys` has a unique template parameter that specifies the type of the generalized coordinates, the type of the transformation data, and four functions that describe the operations involving the previous two types (Figure 3). The user may add custom classes of generalized coordinates by realizing the C++ concept `coordsys_traits`. Likewise, the user may specify conversions between classes of coordinate systems by realizing the C++ concept `coordsys_bridge_traits`.

We have included the derived template class `coordsys_generic` to facilitate users the creation of `coordsys`-conforming wrappers. Different classes of generalized coordinate systems can be combined by using bridges, unifying them into a single kinematic tree.

4. Conclusions

We have introduced a novel coordinate system library with several improvements over similar existing libraries. Basically, in this library coordinates are brought together under the same coordinate system by coordinate transformation functions, which are generated by `mapCS()` by querying `coordsys` objects.

The POS library supports standards-based interfaces, connectivity to other coordinate system frameworks, support for custom generalized coordinate spaces, enforced use of units in planar and spatial coordinate systems, and transparent bridges between coordinate system types, amongst other features. Due to the requirements of efficiency and the use of C++ specific features such as C++ concepts, this library is only available in C++ and there are no plans to provide other language bindings.

We hope that this library will simplify operations with coordinate systems to a wide audience, and help bring together different technologies that improve the scalability, efficiency and flexibility of robotic systems.

The POS library is released under the Lesser General Public License [10] and is available at

`https://www.gitlab.com/ninbot/pos`

In the future, we want to integrate IEEE 1873-2015 (IEEE Standard for Robot Map Data Representation for Navigation) [11] into the POS library. [11] standardizes the notions and naming conventions of metric maps (continuous planar coordinates), occupancy grid maps (discrete planar coordinates) and topological maps (graph representation of planar maps), as well as their relationships and uncertainties.

References

[1] https://doc.qt.io/qt-5/qtransform.html.

[2] https://www.cairographics.org/.

[3] John Kessenich, Graham Sellers, and Dave Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.5 with SPIR-V*. Addison-Wesley Professional, Boston, MA, 9th edition edition, 2016.

[4] https://pyproj4.github.io/pyproj/stable/.

[5] https://www.mathworks.com.

[6] Markus Rickert and Andre Gaschler. Robotics library: An object-oriented approach to robot applications. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 733–740, September 2017.

[7] Tully Foote. tf: The transform library. In *2013 IEEE Conference on Technologies for Practical Robot Applications (TePRA)*, pages 1–6, Woburn, MA, USA, 2013. IEEE.

[8] IEEE Standard for Autonomous Robotics Ontology. In *IEEE Std 1872-2015*, pages 1–60, 2015.

[9] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *IEEE/RSJ international conference on intelligent robots and systems*, pages 5026–5033. IEEE, 2012.

[10] https://www.gnu.org/licenses/lgpl-3.0.html.

[11] IEEE Standard for Robot Map Data Representation for Navigation. In *IEEE Std 1873-2015*, 2015.