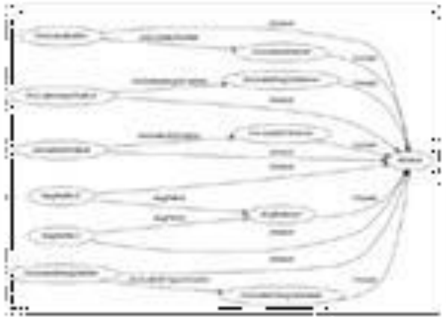
Module B – Programming robot motions in ROS

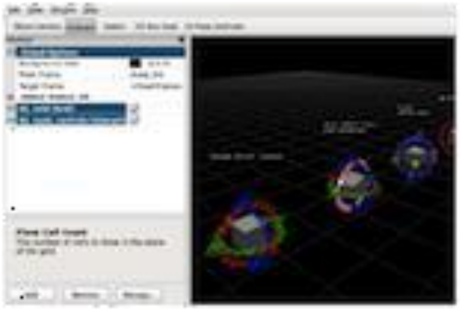
Robot Programming and Control
Accademic Year 2021-2022

Diego Dall'Alba diego.dallalba@univr.it
Department of Computer Science – University of Verona
Altair Robotics Lab

ROS Characteristics



+



+

+

Plumbing

- Process management
- Inter-process communication
- Device drivers

Tools

- Simulation
- Visualization
- Graphical user interface
- Data logging

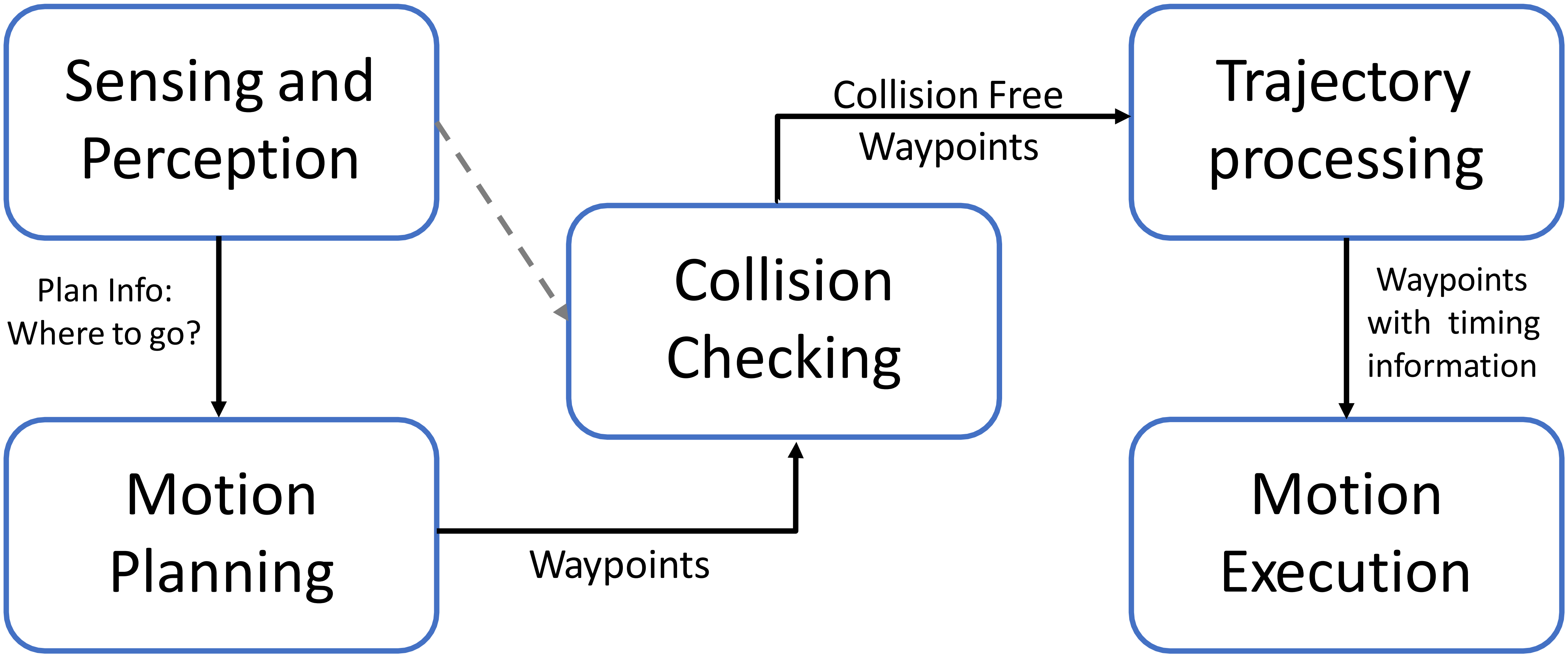
Capabilities

- Control
- Planning
- Perception
- Mapping
- Manipulation

Ecosystem

- Package organization
- Software distribution
- Documentation
- Tutorials

Motion planning basics components recap



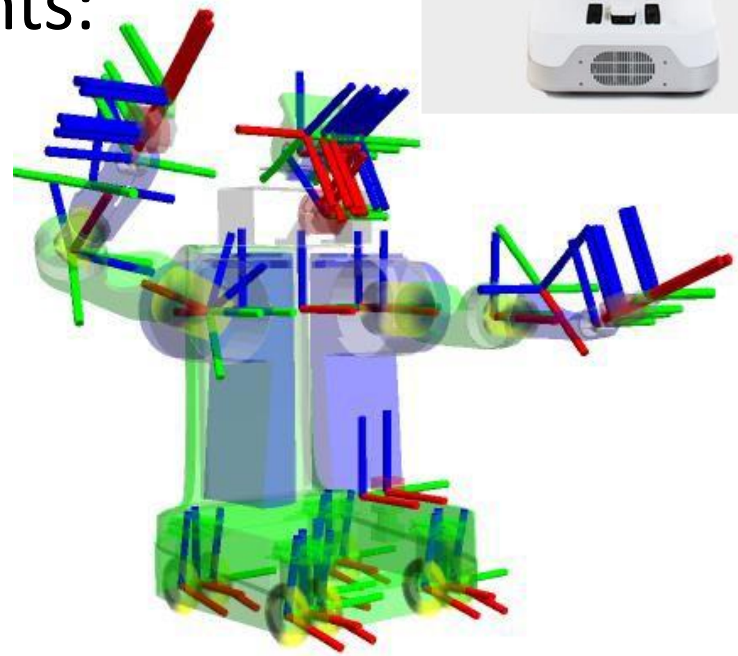
MoveIt Capabilities

https://www.youtube.com/watch?v=7KvF7Dj7bz0

MoveIt

MoveIt Essential Components

- MoveIt has been developed for quickly start using advanced robotic platform in complex applications.
- Similarly to ROS, it has been originally introduced by Willow Garage as development environment for the PR2 robot
- MoveIt provides the following essential components:
  - **Motion planning**
  - **Advanced Kinematics**
  - **Collision checking**
- It is a flexible and extensible platform, suitable for supporting state of the art motion planning algorithms/methods.

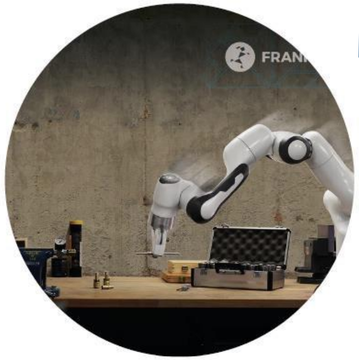


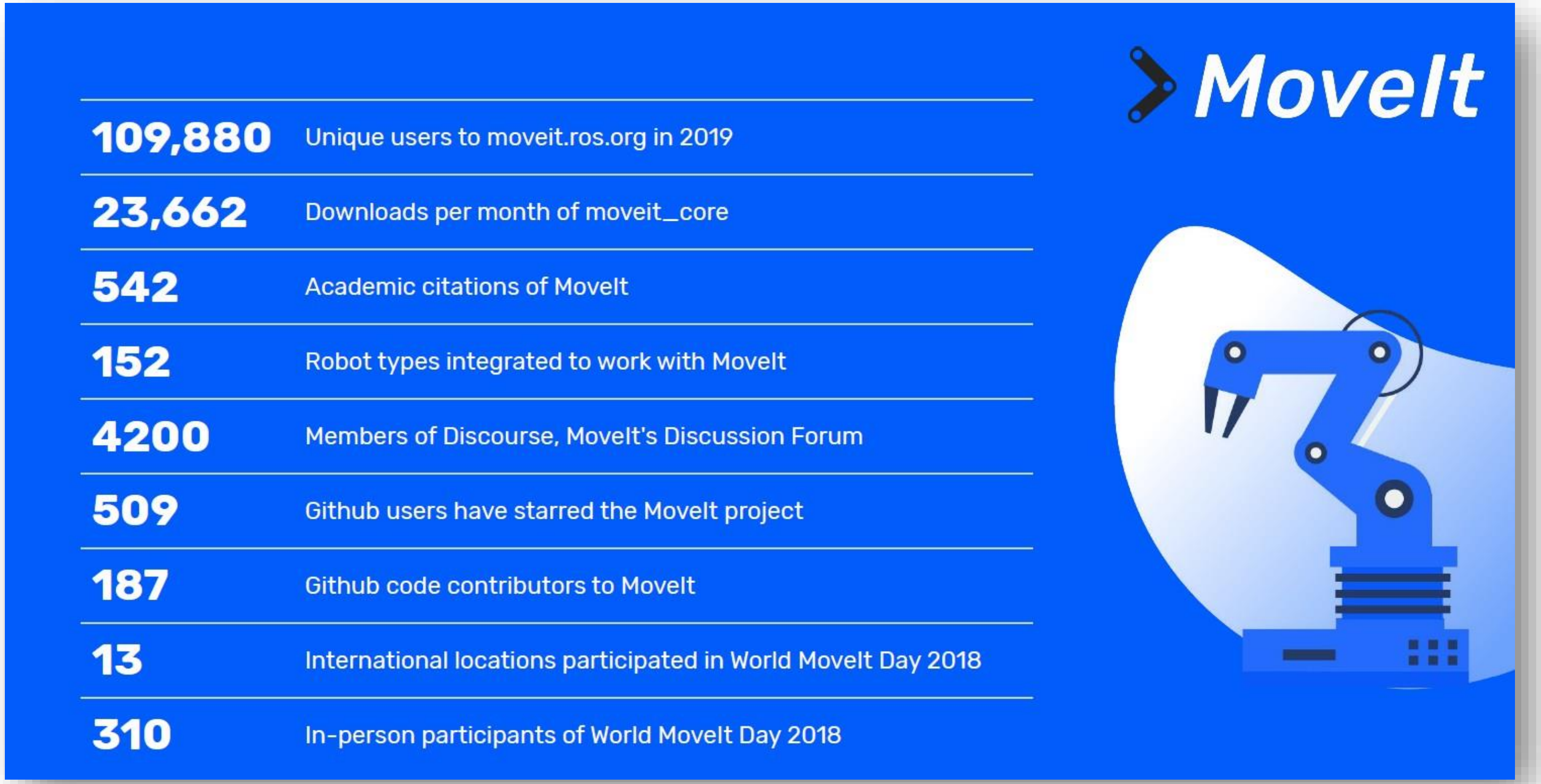
MoveIt: Some numbers

arm_navigation

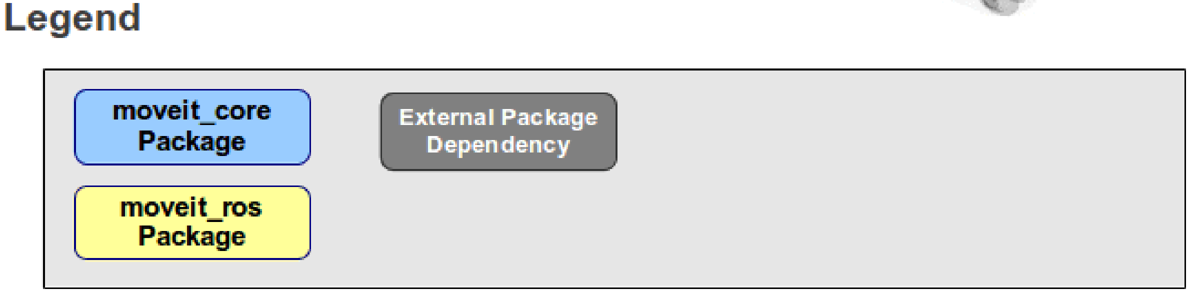
MoveIt!

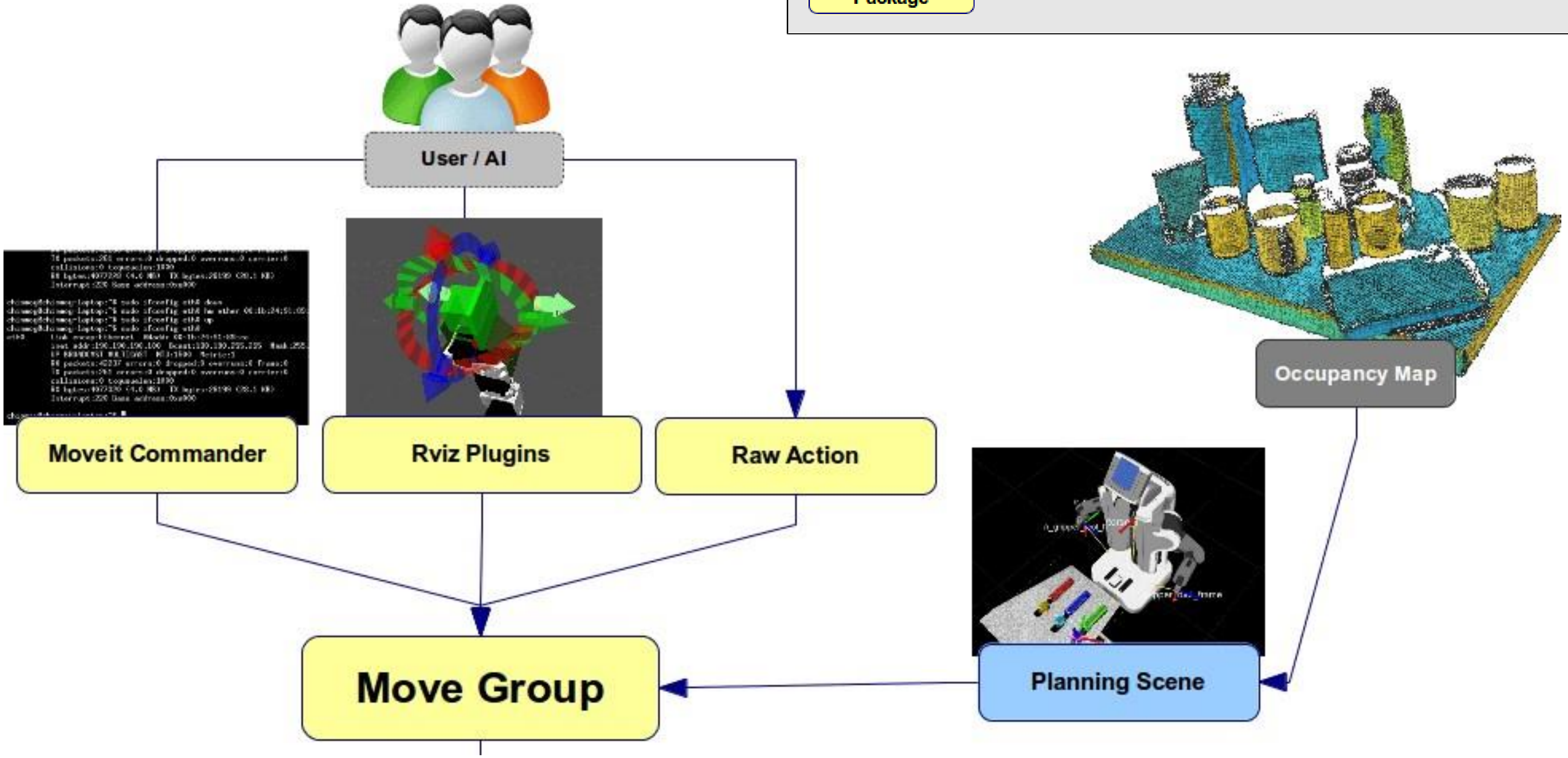
MoveIt!



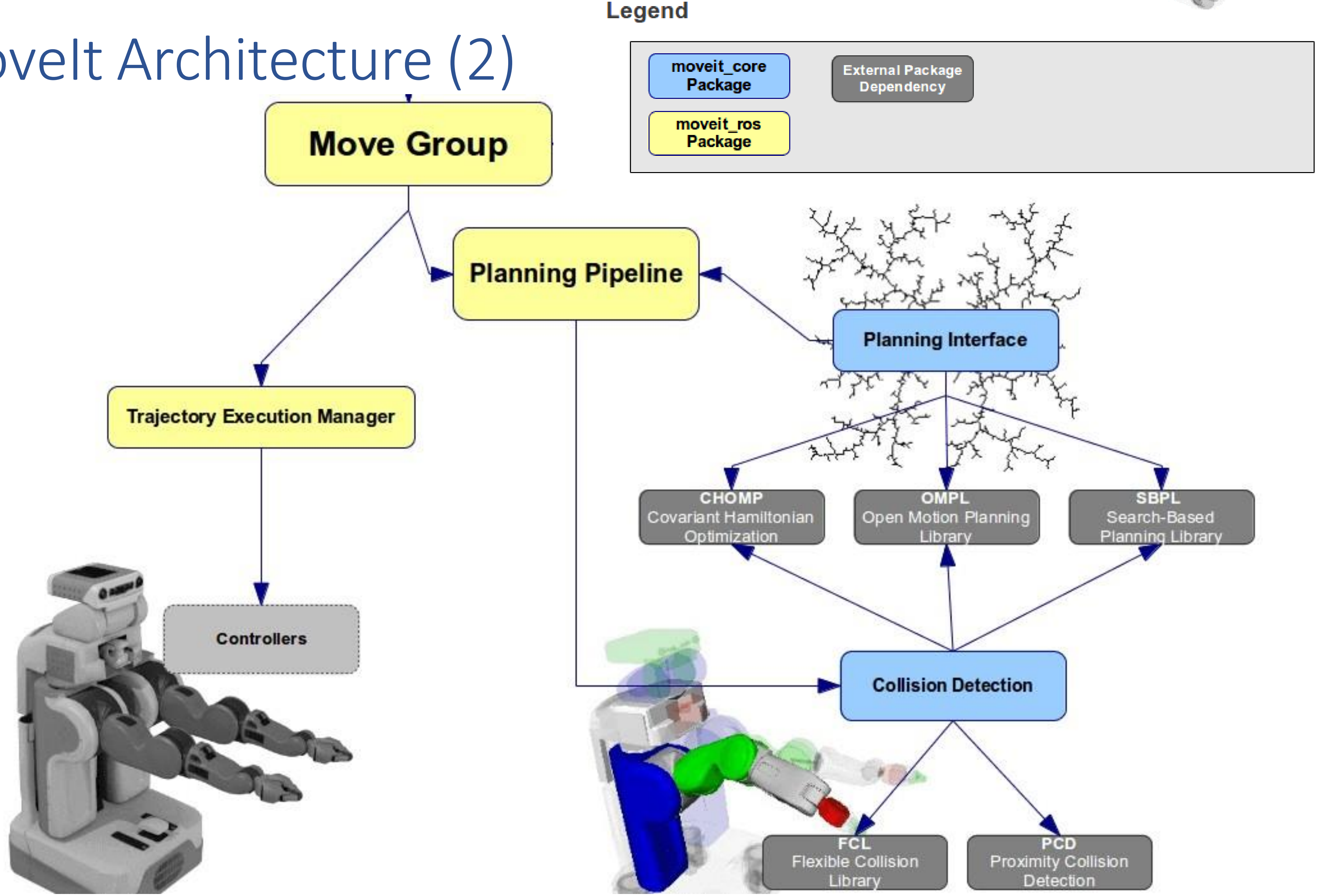


MoveIt Architecture (1)



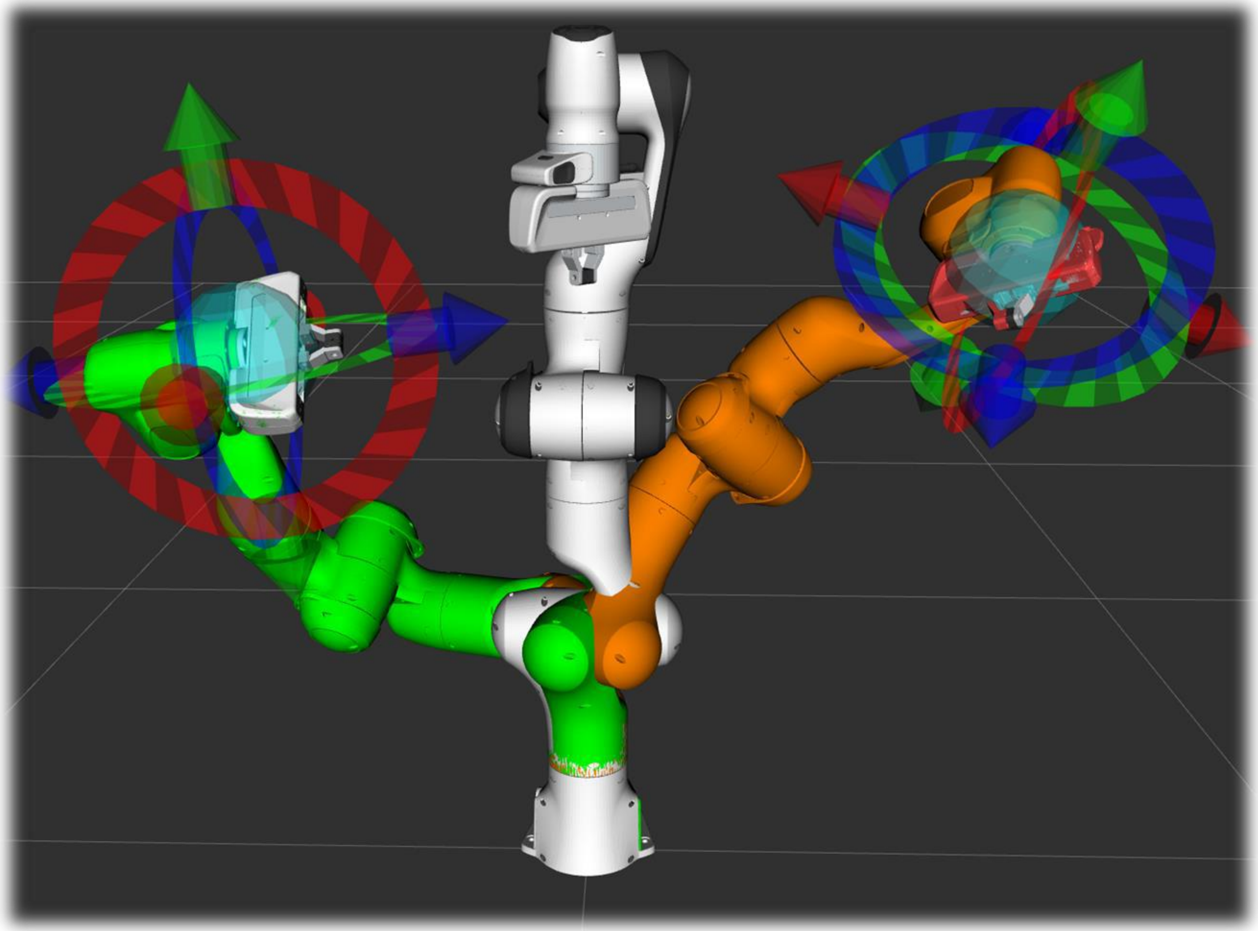


MoveIt Architecture (2)



MoveIt main/hidden functions

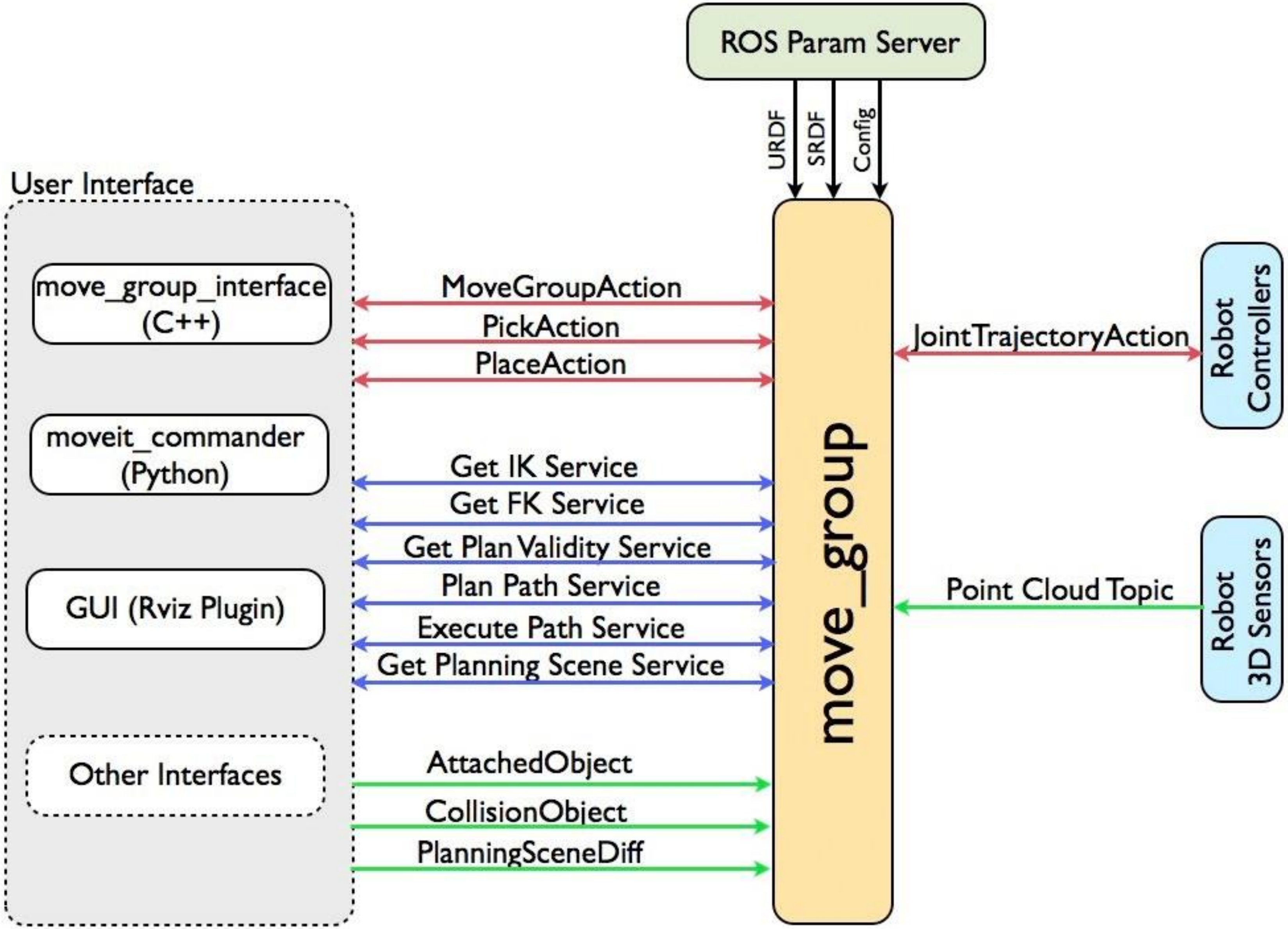
- Maintain information consistency.
- Integrate robot kinematic information with planning.
- Report and request alternative motion plans in case of collisions.
- Account for any hardware limitations such as joint limits.
- Keep track of the current state of the robot and its environment while performing a task.
- Talk to the robot hardware/simulation and notify the ROS application once a desired manipulation task is complete.



MoveIt

Move_group node

This node serves as an integrator: pulling all the individual components together to provide a set of ROS actions and services for users to use.

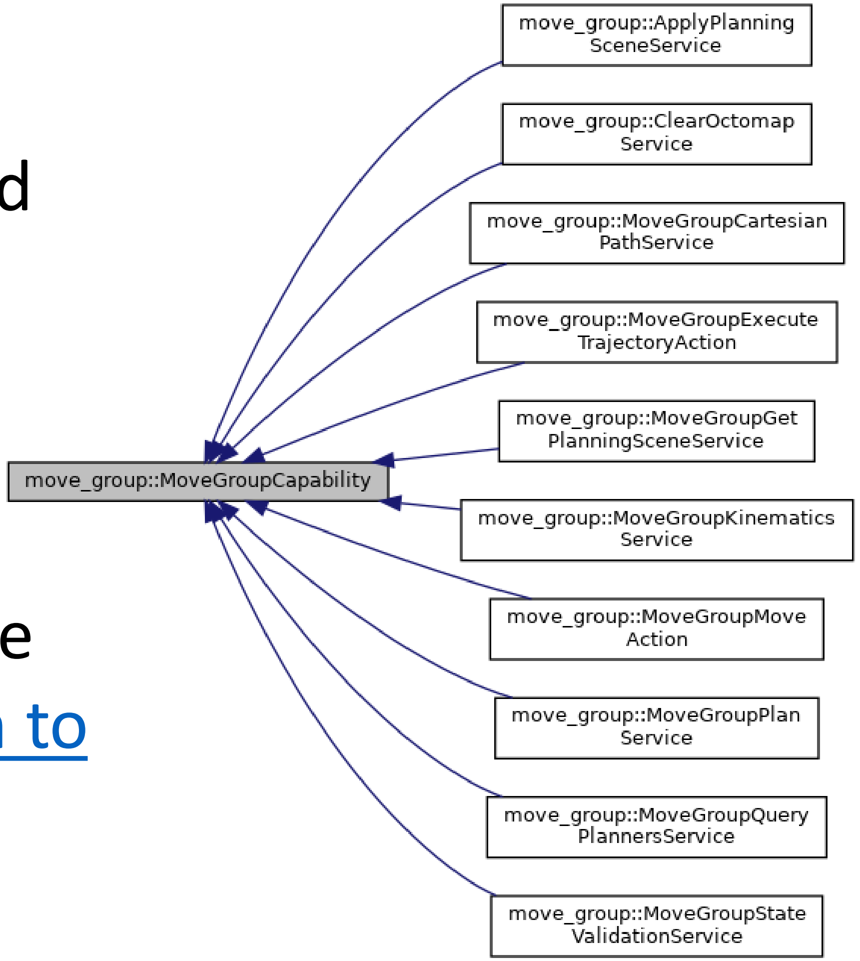


Move_group node: User Interface

The users can access the actions and services provided by *move_group* in one of three ways:

- **In C++** - using the [move_group_interface] package that provides an easy to setup C++ interface to move_group
- **In Python** - using the [moveit_commander] package
- **Through a GUI** - using the [Motion Planning plugin to Rviz] (the ROS visualizer)

move_group can be configured using the ROS param server from where it will also get the URDF and SRDF for the robot (see next slide)



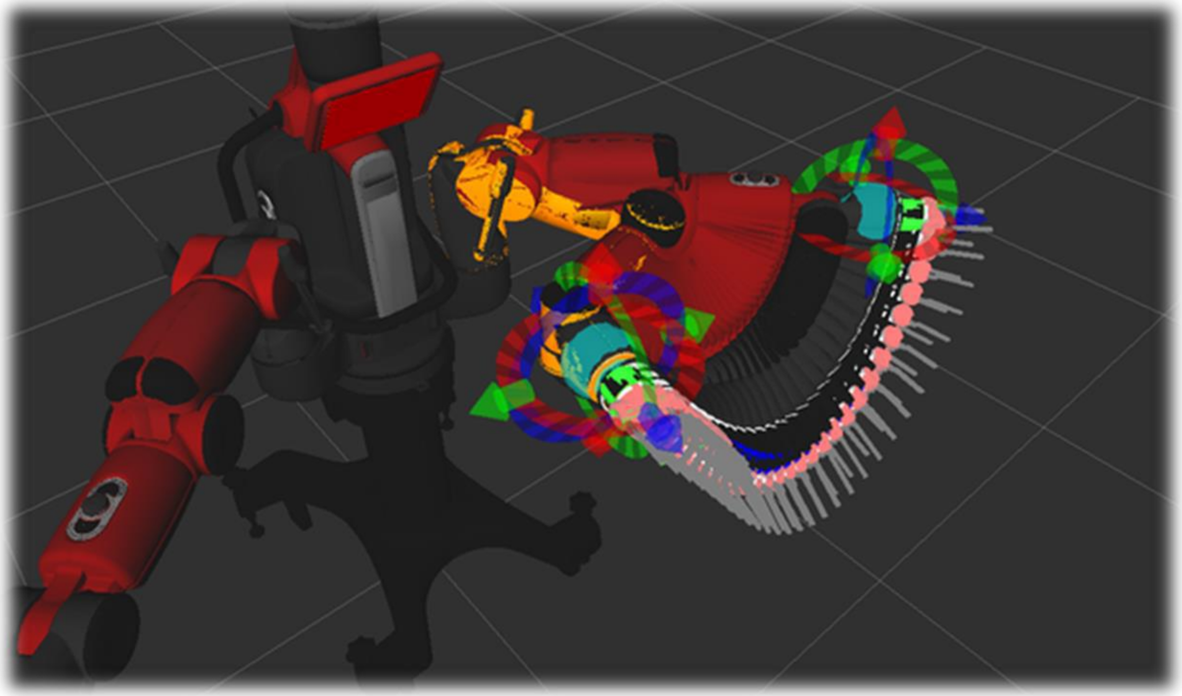
Move_group node: Configuration

- move_group uses the ROS param server to get three kinds of information:
  1.URDF - *move_group* looks for the *robot_description* parameter on the ROS param server to get the URDF for the robot.
  2. SRDF - *move_group* looks for the *robot_description_semantic* parameter on the ROS param server to get the SRDF for the robot. <u>The SRDF is typically created (once) by a user using the MoveIt Setup Assistant.</u>
  3.MoveIt configuration - *move_group* will look on the ROS param server for other configuration specific to MoveIt including joint limits, kinematics, motion planning, perception and other information.
- Config files for these components are automatically generated by the MoveIt setup assistant and stored in the *config* directory of the corresponding MoveIt config package for the robot.

Move_group node: Robot Interface

move_group talks to the robot through ROS topics and actions. It communicates with the robot to:

- get current state information (positions of the joints, etc.),
- get point clouds and other sensor data from the robot sensors,
- talk to the controllers on the robot.

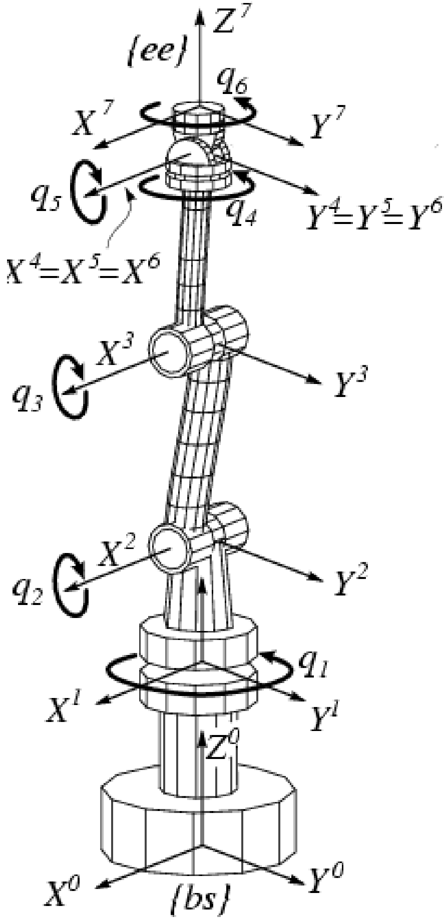


Move_group node: Robot Interface Information

- **Joint State Information:** *move_group* listens on the */joint_states* topic for determining the current state information.
- **Transform Information:** *move_group* monitors transform information using the ROS TF library.
- **Controller Interface:** *move_group* talks to the controllers on the robot using the FollowJointTrajectoryAction interface. This is a ROS action interface.
- **Planning Scene:** *move_group* uses the Planning Scene Monitor to maintain a *planning scene*, which is a representation of the world and the current state of the robot. The robot state can include any objects carried by the robot which are rigidly attached to the robot.
- **Extensible Capabilities:** *move_group* is structured to be easily extensible - individual capabilities like pick and place, kinematics, motion planning are implemented as separate plugins with a common base class.

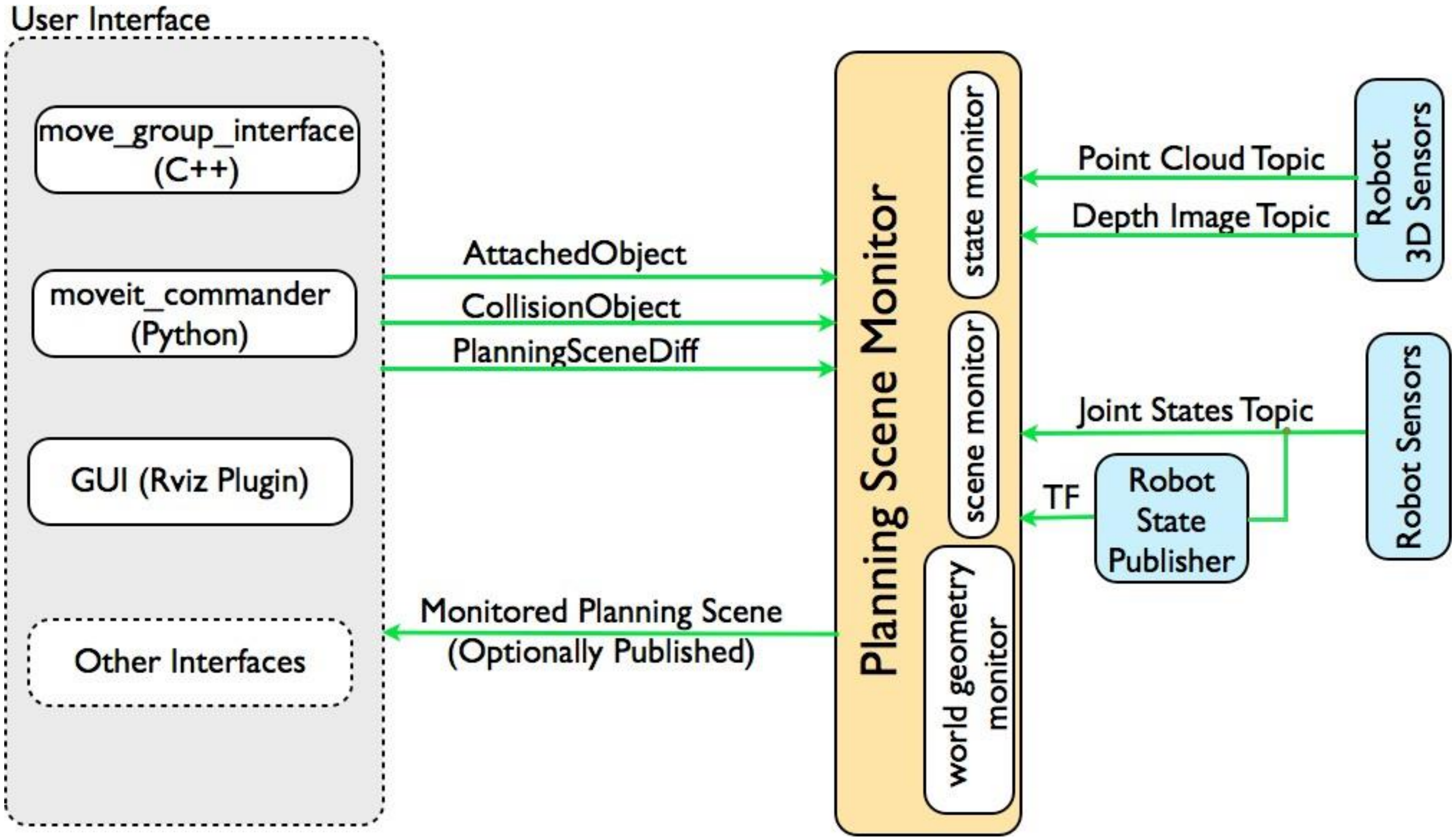
The Kinematics Plugin

- MoveIt uses a plugin infrastructure, especially targeted towards allowing users to write their own inverse kinematics algorithms.
- Forward kinematics and finding Jacobians is integrated within the RobotState class itself.
- The **default inverse kinematics plugin for MoveIt is configured using the KDL numerical jacobian-based solver**. This plugin is automatically configured by the MoveIt Setup Assistant.
- Often, users may choose to implement their own kinematics solvers, e.g. the PR2 has its own kinematics solvers.
- A popular approach to implementing such a solver is using the **IKFast package** to generate the C++ code needed to work with your particular robot. In alternative **TRAC-IK** should be considered.



Planning Scene

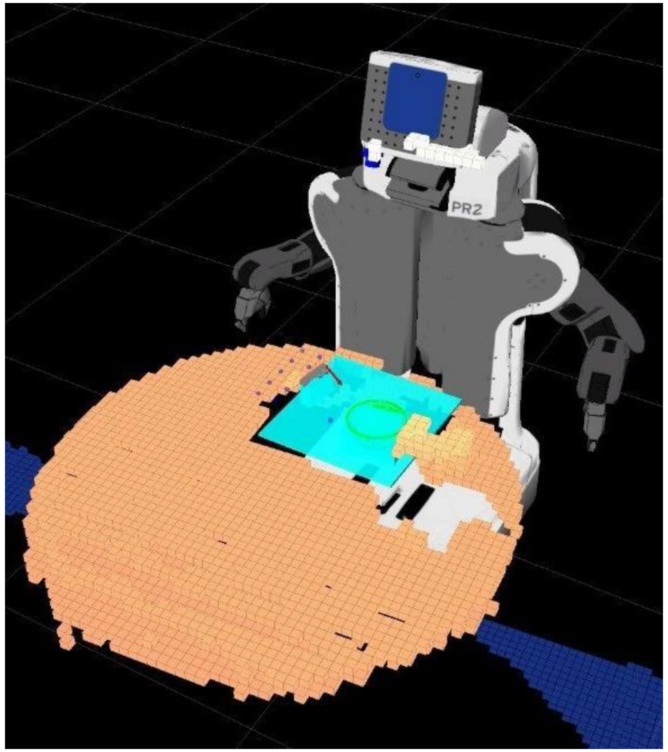
- The planning scene is used to represent the world around the robot and also stores the state of the robot itself.
- It is maintained by the *planning scene monitor* inside the move_group node.
- The *planning scene monitor* listens to:
    - State Information
    - Sensor Information
    - World geometry information



Collision Checking

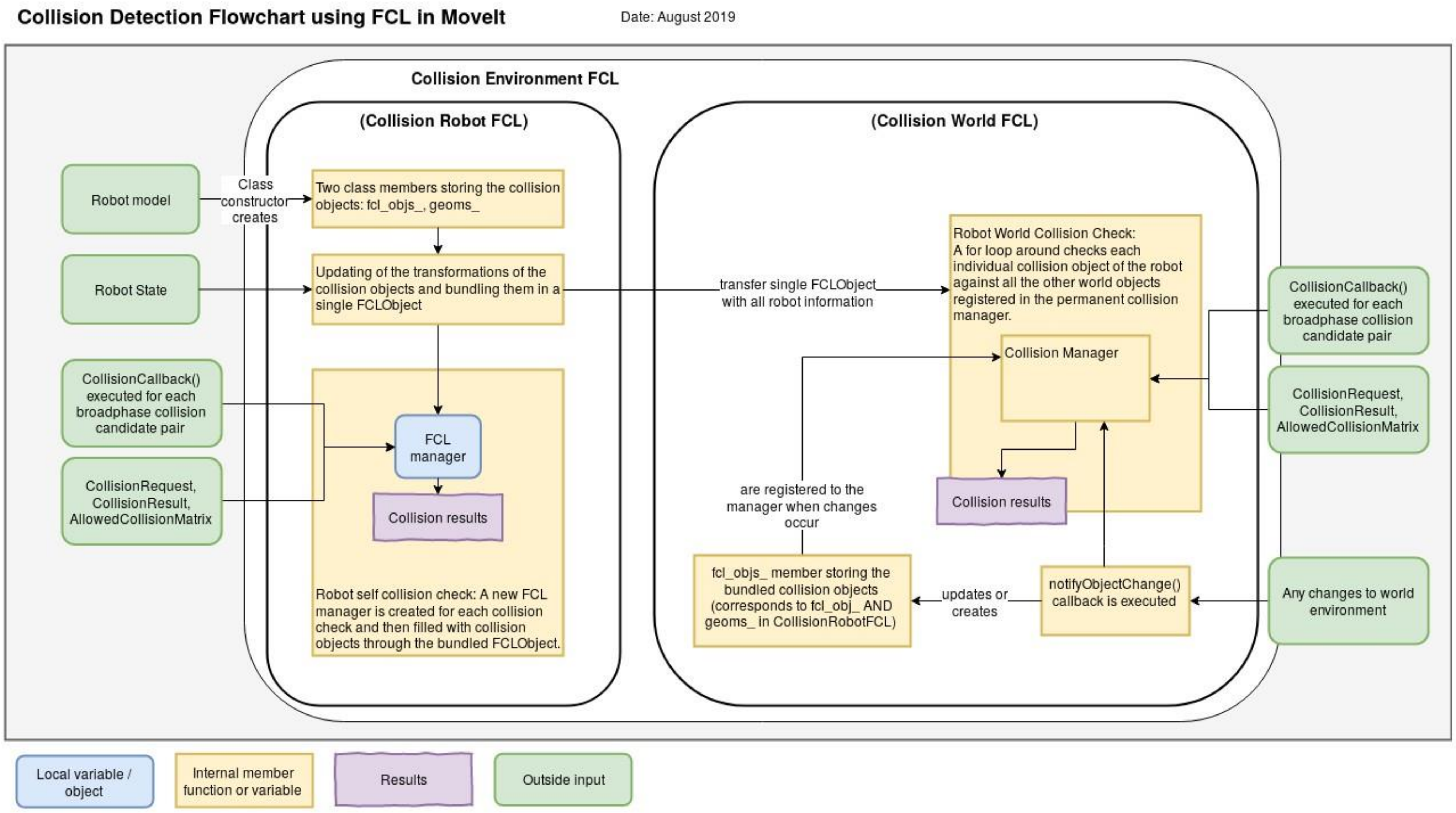
- Collision checking in MoveIt is configured inside a *Planning Scene* using the *Collision Environment* object.
- Fortunately, MoveIt is setup so that users never really have to worry about how collision checking is happening.
- MoveIt supports collision checking for different types of objects including:
    - <u>Meshes</u>
    - <u>Primitive Shapes</u> - e.g., boxes, cylinders, cones, spheres and planes
    - <u>Octomap</u> - the Octomap object can be directly used for collision checking

Flexible Collision Library (FCL)



BULLET PHYSICS LIBRARY

Collision Checking Scheme: FCL Example

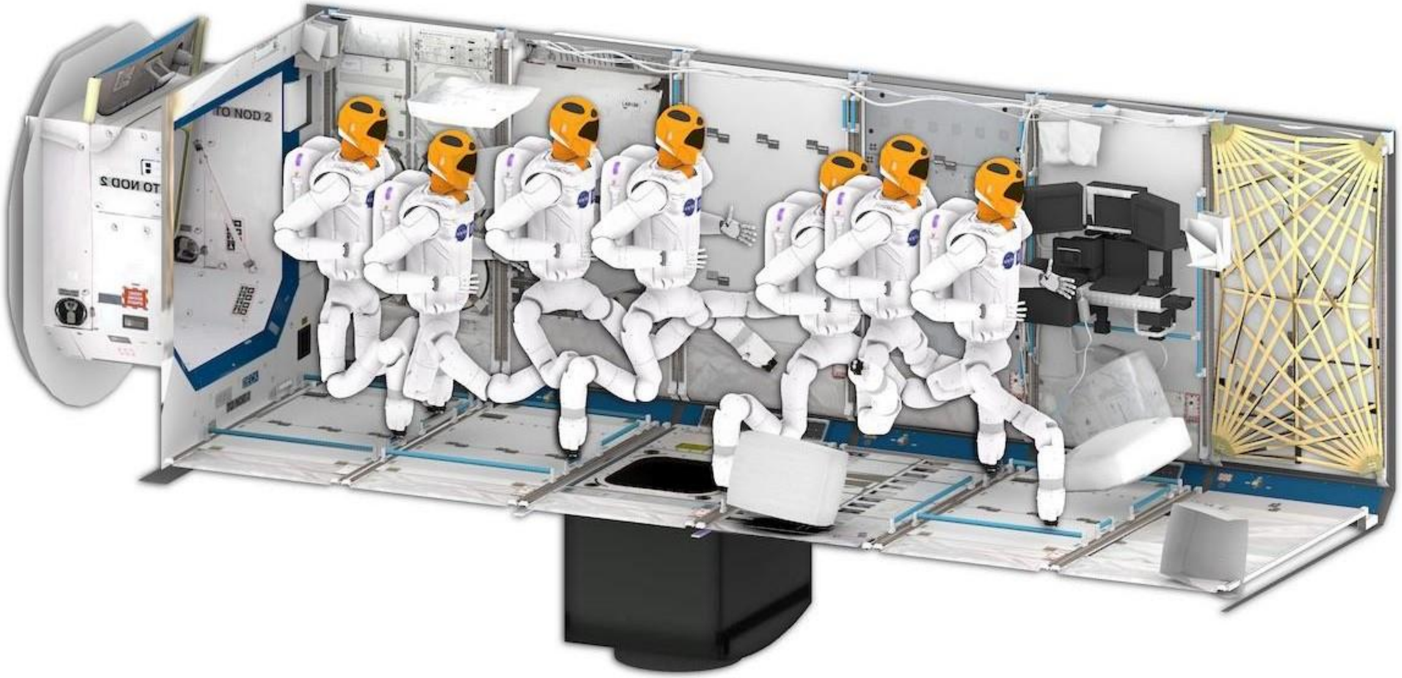


Collision Checking - Allowed Collision Matrix (ACM)

- **Collision checking is a very expensive operation** often accounting for close to 90% of the computational expense during motion planning.
- The **Allowed Collision Matrix or ACM** encodes a binary value corresponding to the need to check for collision between pairs of bodies (which could be on the robot or in the world).
- If the value corresponding to two bodies is set to 1 in the ACM, this specifies that a collision check between the two bodies is not needed.
- This would happen if, e.g., the two bodies are always so far way that they would never collide with each other.

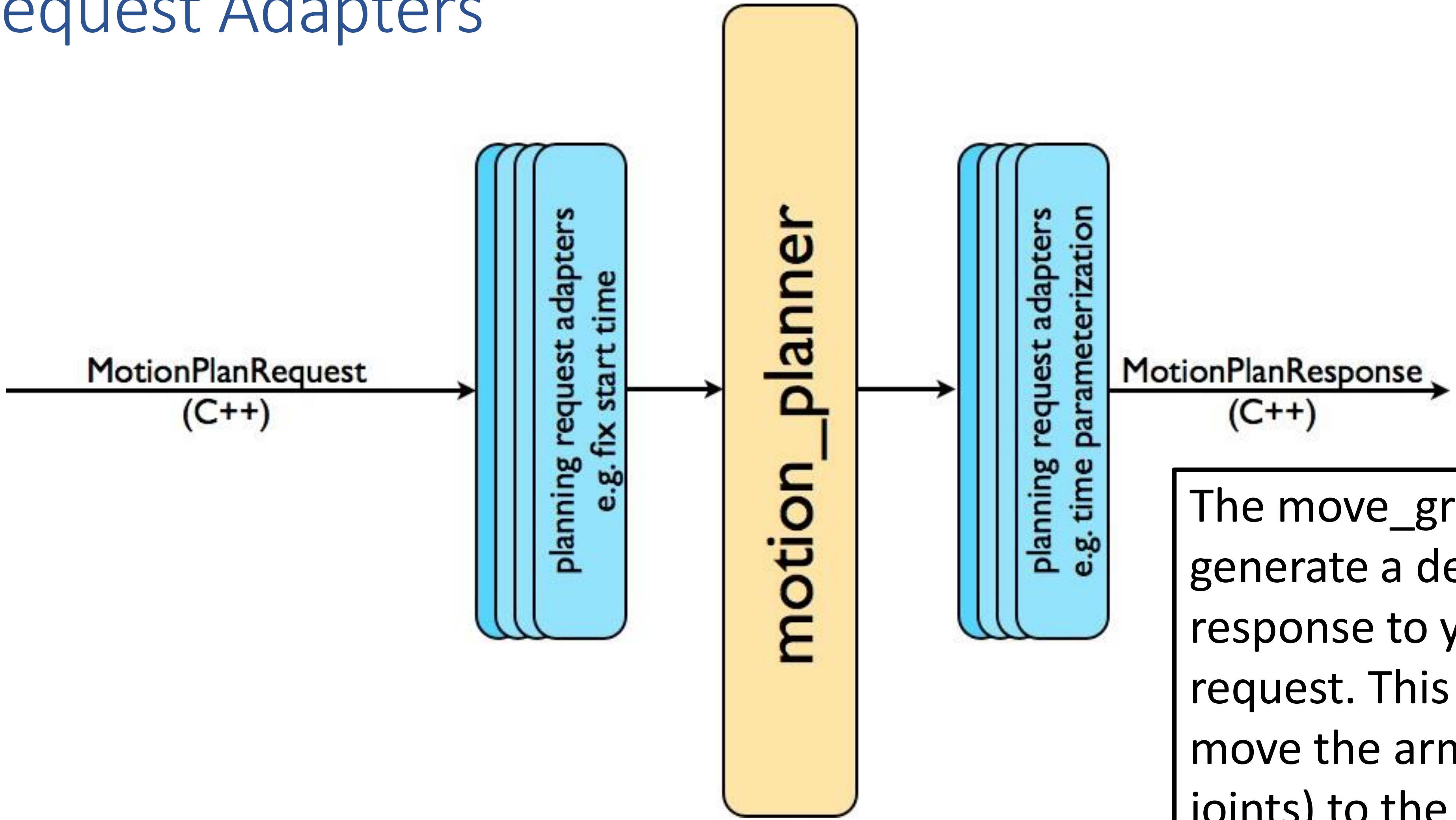
	spacecraft_link0	spacecraft_link1	spacecraft_link2	spacecraft_link3	spacecraft_link4	spacecraft_link5	spacecraft_link6	spacecraft_link7	spacecraft_hand	spacecraft_leftfinger	spacecraft_rightfinger
spacecraft_link0		✓	✓	✓	✓	✓	✓	✓	☐	☐	☐
spacecraft_link1	✓		✓	✓	✓	✓	✓	☐	☐	✓	✓
spacecraft_link2	✓	✓		✓	✓	✓	✓	☐	☐	☐	☐
spacecraft_link3	✓	✓	✓		✓	✓	✓	☐	✓	✓	✓
spacecraft_link4	✓	✓	✓	✓		✓	☐	✓	✓	✓	✓
spacecraft_link5	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓
spacecraft_link6	✓	✓	✓	✓	☐	✓		✓	✓	✓	✓
spacecraft_link7	✓	☐	☐	☐	✓	✓	✓		✓	✓	✓
spacecraft_hand	☐	☐	☐	✓	✓	✓	✓	✓		✓	✓
spacecraft_leftfinger	☐	✓	☐	✓	✓	✓	✓	✓	✓		✓
spacecraft_rightfinger	☐	✓	☐	✓	✓	✓	✓	✓	✓	✓	

Motion Planning



- **MoveIt works with motion planners through a plugin interface**.
- This allows MoveIt to communicate with and use different motion planners from multiple libraries, making MoveIt easily extensible.
- <u>The interface to the motion planners is through a ROS Action or service</u>
- The default motion planners for move_group are configured using OMPL and the MoveIt interface to **OMPL** by the MoveIt Setup Assistant.

The Motion Planning Pipeline: Motion planners and Plan Request Adapters



The move_group node will generate a desired trajectory in response to your motion plan request. This trajectory will move the arm (or any group of joints) to the desired location.

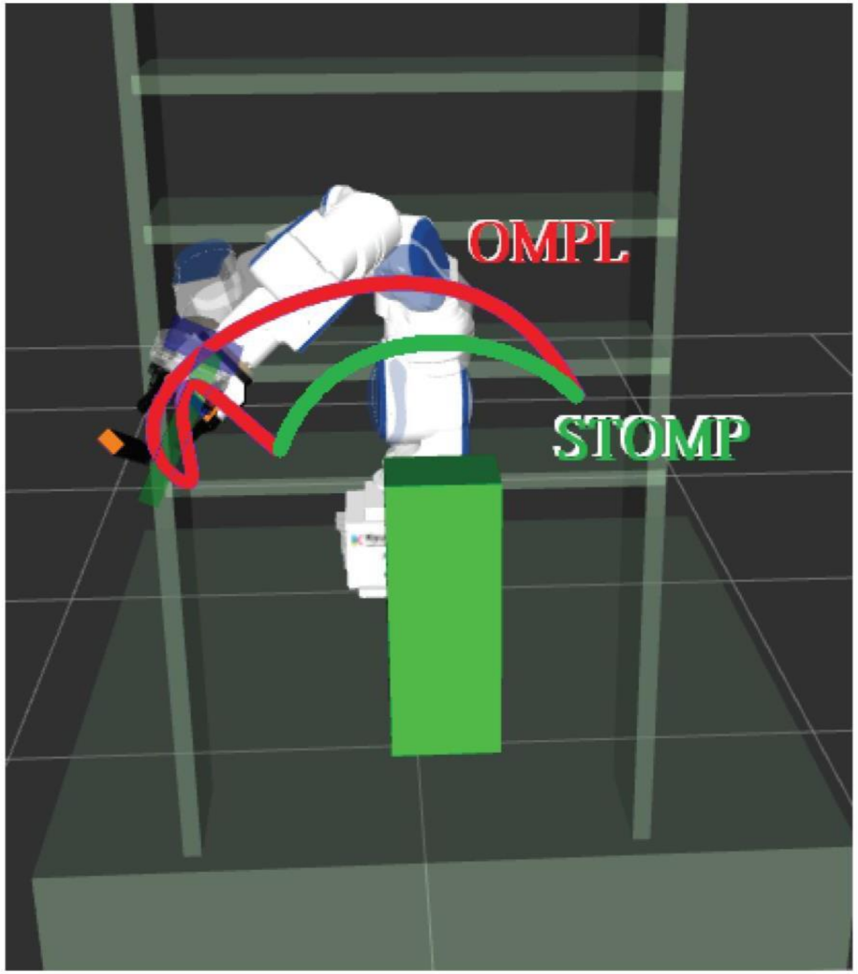
Motion Planning: The Motion Plan Request

- **The motion plan request** clearly specifies what you would like the motion planner to do. Typically, you will be asking the motion planner **to move an arm to a different location** (in joint space) or the end-effector to a new pose.
- **Collisions are checked for by default** (including self-collisions).
- **You can attach an object to the end-effector** (or any part of the robot), e.g., if the robot picks up an object. This allows the motion planner to account for the motion of the object while planning paths.
- You can also specify the following (kinematic)constraints for the motion planner to check:
  - **Position constraints** - restrict the position of a link to lie within a region of space
  - **Orientation constraints** - restrict the orientation of a link to lie within specified roll, pitch or yaw limits
  - **Joint constraints** - restrict a joint to lie between two values
  - **User-specified constraints** - you can also specify your own constraints with a user-defined callback.

OMPL (Open Motion Planning Library)

- **OMPL** (Open Motion Planning Library) is an open-source motion planning library that primarily implements **randomized motion planners.**
- MoveIt integrates directly with OMPL and uses the motion planners from that library as its primary/default set of planners.
- The planners in OMPL are abstract; i.e. OMPL has no concept of a robot.
- Instead, MoveIt configures OMPL and provides the back-end for OMPL to work with problems in Robotics.

Complete list of OMPL planners available at:
http://ompl.kavrakilab.org/planners.html



Other planners (partially) supported in MoveIt

- **Pilz Industrial Motion Planner**
  Pilz industrial motion planner is a deterministic generator for circular and linear motions. Additionally, it supports blending multiple motion segments together using a MoveIt capability.
- **Stochastic Trajectory Optimization for Motion Planning (STOMP)**
  It can plan smooth trajectories for a robot arm, avoiding obstacles, and optimizing constraints. The algorithm does not require gradients and can thus optimize arbitrary terms in the cost function like motor efforts.
- **Search-Based Planning Library (SBPL)**
  A generic set of motion planners using search based planning that discretize the space.
- **Covariant Hamiltonian Optimization for Motion Planning (CHOMP)**
  It is a novel gradient-based trajectory optimization procedure. Given an infeasible naive trajectory, CHOMP reacts to the surrounding environment to quickly converges to a smooth collision-free trajectory that can be executed efficiently on the robot.

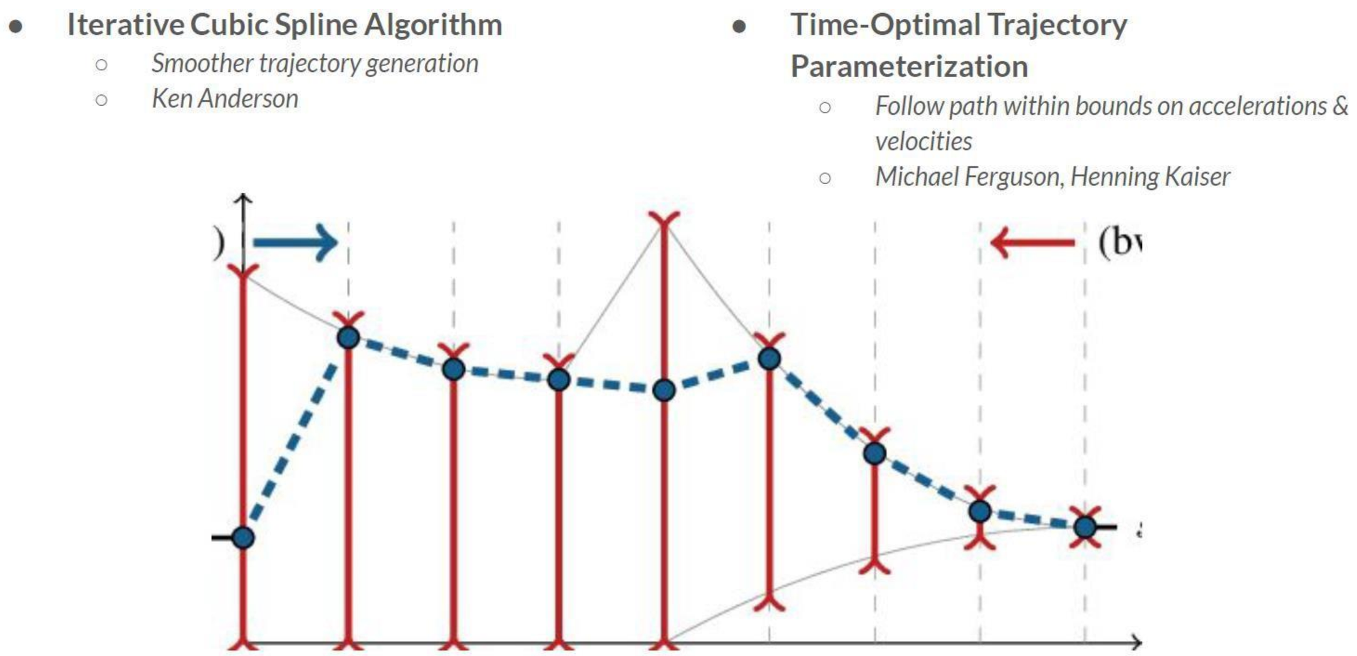
Trajectory Processing - Time parameterization

- Motion planners will typically only generate “paths”, i.e. there is no timing information associated with the paths.
- MoveIt includes a trajectory processing routine that can work on these paths and generate trajectories that are properly time-parameterized accounting for the maximum velocity and acceleration limits imposed on individual joints.
- These limits are read from a special joint_limits.yaml file that is specified for each robot.

Time parameterization algorithms

MoveIt can support different algorithms for post-processing a kinematic trajectory to add timestamps and velocity/acceleration values:

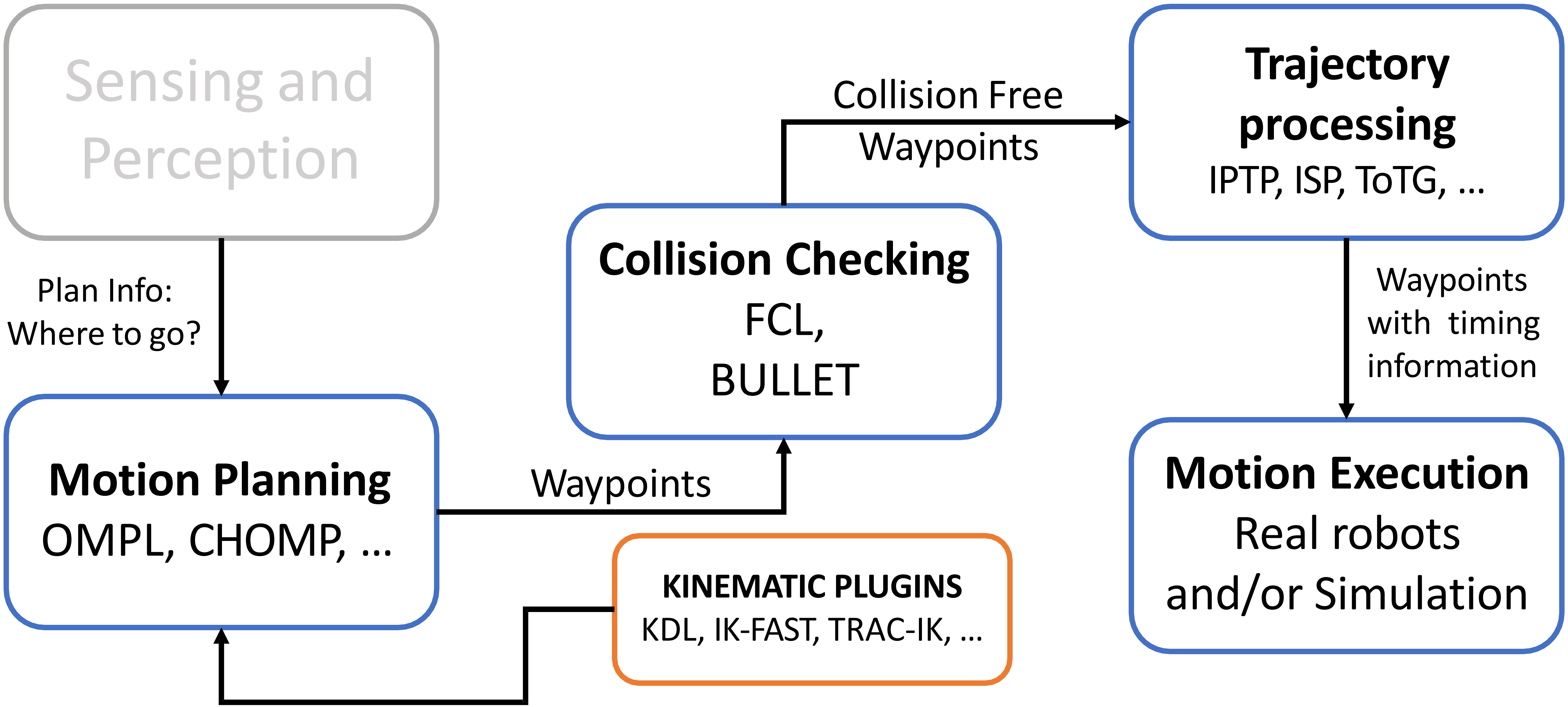
- **Iterative Parabolic Time Parameterization**
- **Iterative Spline Parameterization**
- **Time-optimal Trajectory Generation**



Time parameterization algorithms

- The **Iterative Parabolic Time Parameterization (IPTP)** algorithm is used by default in the Motion Planning Pipeline as a Planning Request Adapter. Although the Iterative Parabolic Time Parameterization algorithm MoveIt uses has been used by hundreds of robots over the years, **there are known issues with it**.
- The **Iterative Spline Parameterization (ISP)** algorithm was recently introduced to deal with these issues. While preliminary experiments are very promising, we are waiting for more feedback from the community before replacing the new default choice.
- **Time-optimal Trajectory Generation (ToTG)** produces trajectories with very smooth and continuous velocity profiles.
  The method is based on fitting path segments to the original trajectory and then sampling new waypoints from the optimized path.
  This is different from strict time parameterization methods as resulting waypoints may divert from the original trajectory within a certain tolerance. Thus, **additional collision checks might be required when using this method**.

Motion planning basics components recap



Exercises Intro – Simulated e.Do

sudo apt install ros-noetic-moveit

Add the following packages to your catkin workspace:

https://github.com/Pro/eDO_moveit
https://github.com/Pro/edo_gazebo
https://github.com/Pro/edo_gripper_moveit
https://github.com/Pro/edo_gripper
https://github.com/Pro/eDO_description

- **INTERACTIVE SESSION**

```
roslaunch edo_gazebo edo_gripper.launch
roslaunch edo_gripper_moveit edo_moveit_planning_execution.launch __ns:=edo sim:=true
roslaunch edo_gripper_moveit moveit_rviz.launch __ns:=edo config:=true
```

Exercise 1: MoveIt RViz User Interface

Starting from the Exercise Intro, complete the following points:

A. Learn how to interact with MoveIt through the Rviz plugin, change the visualization settings (left panels)

B. Get familiar with the different components required to run MoveIt

C. Play with the different parameters, try to understand how to control the velocity/acceleration of the motions

D. Jog the robot in Joint space, try to find a Singular configuration, see how joints limit and singularities are handled in RViz

E. Optional: Look to different planners provided by OMPL, take a look to the documentation to understand the difference among them

Exercise 2: MoveIt Commander

Starting from the Exercise Intro, complete the following points:

A. In an additional terminal launch the following command

```
rosrun moveit_commander moveit_commander_cmdline.py
__ns:=edo
```

B. Try to understand the commands available (type “help”)

C. Play with the commands available, pay attention that SI units are used

D. Write an external script to command a sequential linear motion of 0.25 m along X, Y, Z directions (see “load”)

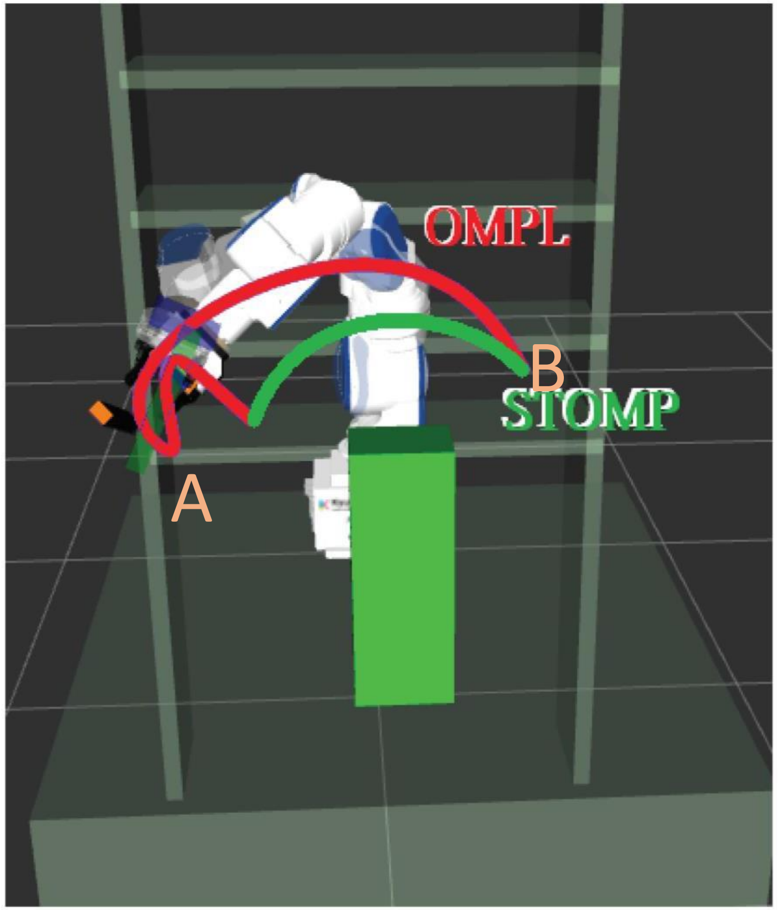
E. Write an external script for moving the robot on a square trajectory (side length 0.2 m) paralled to X-Y world plane and with Z coordinate 0.3

F. **Optional:** Test different positions for the square

List all usable commands
$ help
Select the “group” to use
$ use <group_name>
Plan and execute motion from srdf
$ go <named_target>
Plan and execute linear motions
$ go <up | down | left | right | forward | backward> <distance_in_m>
Get current joint state and pose
$ current
Execute multiple commands
$ load <path_to_script_file/scrip_file_name>

Other Exercises

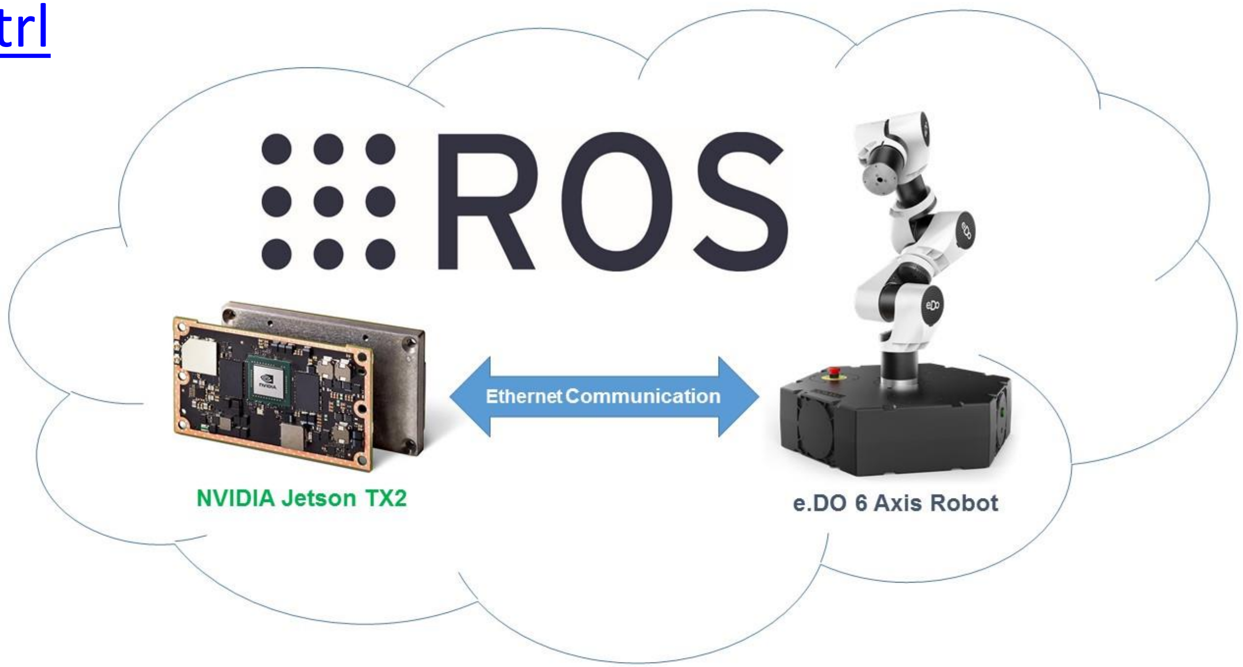
- Using the Rviz plan a motion between point A and B having same Z and Y coordinates
- Introduce an external object in Rviz (Box or Cylinder Object) between point A and B and see how the trajectory is changed to avoid the collision
- **Optional**, try to import a URDF model a different robot using the MoveIt Setup Assistant.
- **Optional**, take a look on how to control MoveIt from external node, using C++ or Python interface



Exercises – Basic control of real e.Do: C++ Example

Create a new catkin workspace with the following package:

https://github.com/jshelata/eDO_manual_ctrl
https://github.com/Comau/eDO_core_msgs



INTERACTIVE SESSION

You might need to install also some supporting libraries

```
sudo apt-get install libncurses5-dev libncursesw5-dev
```

Exercise 3: Jog the robot

Start the node contained in the package just installed:

```
rosrun edo_manual_ctrl edo_manual_ctrl
```

A. Get familiar with the jogging interface provided by the node (both Joint and Cartesian space, pay attention to the signs and value, in particular for the orientation value)
B. Understand the implementation details (Message types, main loop, etc…)

Exercises – Basic control of real e.Do: Python Example


```
sudo apt-get install ipython3
sudo apt-get install python3-pip
pip install pyedo
pip install roslibpy
pip install pynput
```

https://github.com/Comau/pyedo

INTERACTIVE SESSION

HOMEWORK

Implement a pick and place task using this python interface and having a flexible definition of pick and place positions according to letters defined on the e.Do dashboard

Questions?

The contents of these slides are partially based on:

www

Other online sources

MoveIt