INVERSE KINEMATICS MANIPULATION REPORT

Rviz is the primary visualizer in ROS and an incredibly useful tool for debugging robotics. The MoveIt Rviz plugin allows you to setup virtual environments, create start and goal states for the robot interactively, test various motion planners, and visualize the output

To launch:

Sometimes the bash file may not be sourced so type
source .bashrc (type this before typing the command below)
roslaunch panda_moveit_config demo.launch rviz_tutorial:=true

For the first time only an empty world in Rviz will be seen and we will have to add the Motion Planning Plugin

After adding the Motion Planning Plugin set the Fixed Frame field in the Global Options tab to /panda_link0

Robot Description is robot_description
Planning Scene Topic is move_group/monitored_planning_scene
Trajectory topic under planned path is :
/move_group/display_planned_path
In Planning Request change the Planning Group to panda_arm

There are 4 different overlapping visualizations:

- Robots configuration in the move_group/monitored_planning_scene planning environment
- The planned path for the robot
- Green: The start state for the motion planning
- Orange: The goal state for the motion planning

The display states for each of these visualizations can be toggled on and off using checkboxes:

1. The planning scene robot using the Show Robot Visual checkbox in the Scene Robot tab.
2. The planned path using the Show Robot Visual checkbox in the Planned Path tab.
3. The start state using the Query Start State checkbox in the Planning Request tab.
4. The goal state using the Query Goal State checkbox in the Planning Request tab.

If we only want the scene robot, start state and goal state then we will see two interactive markers, one the goal state and one the start state. If interactive markers are not seen then press the Interact button at the top of the display tab

If we move one of the arms now into collision with the other then the links that are in collision will turn red. If the Use Collision Aware IK checkbox is ticked then solver attempts to find collision free solution for desired end effector pose. Otherwise it allows collisions to happen. This checkbox is in MotionPlanning plugin under Planning tab

The planning tab lets you choose start and goal states for a planning request for robot using options:

- The current state
- The previous state
- A randomly sampled state
- A named state of the selected planning group as defined in the .srdf file of the robot

In the Start state option of the planning tab the previous option refers to the start state of the previous planning attempt.
We can also select a motion planning slider from the panels menu in Rviz to introspect the trajectory
The robot will attempt to move end effector in cartesian space if the Use Cartesian Path checkbox is ticked

Python and Moveit

To use the Python MoveIt interfaces we will import the moveit_commander namespace. This namespace provides us with a MoveGroupCommander class, PlanningSceneInterface class and RobotCommander class

The zip() function returns a zip object, which is an iterator of tuples where the first item in each passed iterator is paired together, and then the second item in each passed iterator are paired together etc

After importing all the namespaces such as rospy, moveit_commander, sys, moveit_msgs.msg and geometry_msgs.msg initialize a moveit_commander using:
moveit_commander.roscpp_initialize(sys.argv)
rospy.init_node("move_group_python_interface_tutorial", anonymous=True)

Instantiate RobotCommander object which provides info such as the robot's kinematic model and the robot's current joint states:
robot=moveit_commander.RobotCommander()

Instantiate a PlanningSceneInterface object. Provides a remote interface for getting, setting and updating the robot's internal understanding of the surrounding world.scene = moveit_commander.PlanningSceneInterface()

Instantiate MoveGroupCommander object. This object is an interface to a planning group(group of joints). This interface can be used to plan and execute motions.
group_name = "panda_arm"
move_group = moveit_commander.MoveGroupCommander(group_name)
Create a DisplayTrajectory ROS publisher which is used to display trajectories in Rviz:
display_trajectory_publisher = rospy.Publisher(
  "/move_group/display_planned_path",
  moveit_msgs.msg.DisplayTrajectory,
  queue_size=20,
)
DisplayTrajectory msg has model id(string model_id), RobotTrajectory and RobotState.

The joint_goal are the angles of the joints and the planning groups are the links and joints we want to move. This joint_goal is used to set the desired angles of each of the joints.
After setting the goal we can implement the move_group.go(joint_goal, wait=True) function
at the end put move_group.stop()

pose_goal is used to set the pose of the end-effector
pose_goal=geometry_msgs.msg.Pose()
pose_goal.orientation.w=1.0 (w+ix+jy+kz=cos(theta/2)+sin(theta/2)*(ix0+jy0+kz0))
pose_goal.position.x=0.4
..

move_group.set_pose_target(pose_goal)

Some other things are also there but I am too dumbheaded to do them

The moveit_commander package also includes a command line interface, moveit_commander_cmdline.py.This means we can operate the arm from the command line

Use this command to use the cmd line to operate the panda arm
rosrun moveit_commander moveit_commander_cmdline.py
Type the command use panda_arm (panda_arm is the group name you wish to command)

rec c
This will remember the current joint values of the robot group under the name c

RobotModel and RobotState classes are the core classes that give access to the robot's kinematics..

RobotModel class contains relationships between all the links and joints including their joint limit properties as loaded from the URDF. Also separates the robot's links and joints into planning groups defined in the SRDF.

RobotState contains info about the robot at a certain point in time. Stores vectors of joint positions and optionally velocities and accelerations which can be used to obtain kinematic info about the robot that depends on its current state, such as the Jacobian of the end effector.

The device attached to the manipulator which interacts with its environment to perform tasks is called the end-effector.Jacobian is Matrix in robotics which provides the relation between joint velocities ( ) & end-effector velocities ( ) of a robot manipulator.

RobotState also contains helper functions for setting the arm location based on the end effector location and for computing Cartesian trajectories

An application programming interface(API) is a way for two or more computer programs to communicate with each other. It is a type of software interface, offering a service to other pieces of software.

We have to first instantiate a RobotModelLoader object which looks up the robot description and constructs a RobotModel for us to use.
I am assuming this is written in c++.

Using RobotModel we can construct RobotState that maintains the configuration of the robot. We can set the joints to default and then get JointModelGroup which represents robot model for particular group eg “panda_arm” of robot.

When a variable is declared as a reference, it becomes an alternative name for an existing variable. A variable can be declared as a reference by putting ‘&’ in the declaration

```
moveit::core::RobotStatePtr kinematic_state(new
moveit::core::RobotState(kinematic_model));
kinematic_state->setToDefaultValues();
const moveit::core::JointModelGroup* joint_model_group =
kinematic_model->getJointModelGroup("panda_arm");

const std::vector<std::string>& joint_names = joint_model_group->getVariableNames();
```

No need to use std if we have put using namespace std;

This is used to get the joint values

This is used to set the joint limits. Has setJointGroupPositions instead of copyJointGroupPositions.
The meaning of DISTAL is situated away from the point of attachment or origin or a central point especially of the body.

We want to compute forward kinematics to find the pose of panda_link8 which is most distal link in the panda_arm group of robot. Based on the set of joint angles that are set it will give the translation of the end effector and the rotation of the end effector.

To solve IK, we need:
desired pose of the end effector.

the timeout:0.1s

we are finding ik position for the endeffector state we found from FK.

Somehow this gives the Jacobian.

PlanningScene class provides the main interface that we use for collision checking and constraint checking.

The ways to setup and configure PlanningScene class is using RobotModel or a URDF and SRDF. However the PlanningSceneMonitor is the recommended method to create and maintain the current planning scene using data from the robot's joints and the sensor data
Here we are instantiating a PlanningScene directly

To check the collision in the initial position we construct a CollisionRequest and a Collision Result object and pass into the collision checking function. The result of whether the robot is in self-collision or not is contained within the result.

```
collision_detection::CollisionRequest collision_request;
collision_detection::CollisionResult collision_result;
planning_scene.checkSelfCollision(collision_request, collision_result);
ROS_INFO_STREAM("Test 1: Current state is " << (collision_result.collision ? "in" : "not in") << " self collision");
```

Now to change the state of robot and check again for collision. The planning scene maintains the current state internally.

```
moveit::core::RobotState& current_state = planning_scene.getCurrentStateNonConst();
current_state.setToRandomPositions();
collision_result.clear();
planning_scene.checkSelfCollision(collision_request, collision_result);
ROS_INFO_STREAM("Test 2: Current state is " << (collision_result.collision ? "in" : "not in") << " self collision");
```

Now to do collision checking for a specific group like panda_hand which will check if there is any collision between panda hand and any other part of the body

To get contact info first set manually the Panda arm to a position where we know internal collisions happend and remember that this state is now actually outside the joint limits of the Panda

```
std::vector<double> joint_values = { 0.0, 0.0, 0.0, -2.9, 0.0, 1.4, 0.0 };
const moveit::core::JointModelGroup* joint_model_group =
current_state.getJointModelGroup("panda_arm");
current_state.setJointGroupPositions(joint_model_group, joint_values);
ROS_INFO_STREAM("Test 4: Current state is "
                << (current_state.satisfiesBounds(joint_model_group) ? "valid" : "not valid"));
```

We can ask for contact info by filling in the appropriate field in the collision request and specifiying the max number of contacts to be returned as a large number.

```
collision_request.contacts = true;
collision_request.max_contacts = 1000;
collision_result.clear();
planning_scene.checkSelfCollision(collision_request, collision_result);
ROS_INFO_STREAM("Test 5: Current state is " << (collision_result.collision ? "in" : "not
in") << " self collision");
collision_detection::CollisionResult::ContactMap::const_iterator it;
for (it = collision_result.contacts.begin(); it != collision_result.contacts.end(); ++it)
{
  ROS_INFO("Contact between: %s and %s", it->first.first.c_str(), it->first.second.c_str());
}
```

The AllowedCollisionMatrix(ACM) provides a mechanism to tell the collision world to ignore collisions between certain objects between parts of robot and objects of the world. We can tell the collsion checker to ignore collisions between links reported ie even if they are in collision the collision checker will ignore them and return not in collision

```
ROS_INFO_STREAM macro accepts C++ style std::iostream
ROS_INFO_STREAM("Sending random velocity command: " << " linear=" << msg. linear . x
<< "angular=" << msg. angular . z)
```

Instead of checking for self collisions we can use the checkCollision functions to check for both self collisions and for collisions with the environment

There are 2 types of constraints a) constraint chosen from the KinematicConstraint ie JointConstraint, PositionConstraint, OrientationConstraint and VisibilityConstraint. b) user defined constraint specified through callback

First define simple position and orientation constraint on the end-effector of panda_ard group of Panda Robot.

Now check a state against this constraint using isStateConstrained functions in planningscene class

More efficient way to check constraints is to construct a KinematicConstraintSet which pre processes ROS constraints messages.

There's a direct way to do this using the KinematicConstraintSet class.

```
kinematic_constraints::ConstraintEvaluationResult constraint_eval_result =
   kinematic_constraint_set.decide(copied_state);
ROS_INFO_STREAM("Test 10: Random state is " << (constraint_eval_result.satisfied ?
"constrained" : "not constrained"));
```

Then there are also user defined constraints but I am not doing them right now
roslaunch moveit_tutorials planning_scene_tutorial.launch

RobotState: is a snapshot of robot and contains RobotModel and set of joint values.
CurrentStateMonitor: subscribes to a topic for JointState messages that provide up to date sensor values for singel dof actuators.and updates internal robot state with those joint values. To maintain up to date transform info for links and other fromes attached with multiple dof joints the CSM stores a TF2 buffer that used TF2 TransformListener to set their transforms in its internal data.

JointState message type
std_msgs/Header header (time at which the joint states were recorded)
string[] name
float64[] position (rad or m)
float64[] velocity (rad/s or m/s)
float64[] effort (Nm or N)

PlanningSceneMonitor wraps planningscene with ROS interfaces for keeping the PlanningScene up to date.Use the provided LockedPlanningSceneRW and LockedPlanningSceneRO classes to access PlanningSceneMonitor underlying the PlanningScene

Look at the topics it publishes to , subscribes to and services and all in tutorials
PlanningSceneInterface is a useful class for publishing updates to a MoveGroup's PlanningSceneMonitor through C++ API.
primary node provided by MoveIt called move_group. 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.

Planning Scene ROS API
we will perform two operations:
Adding and removing objects into the world
Attaching and detaching objects to the robot
To run this code open two shells. In first shell start Rviz and wait for everything to load using "roslaunch panda_moveit_config demo.launch"
In second shell run the launch file "roslaunch moveit_tutorials planning_scene_ros_api_tutorial.launch"

MoveItVisualTools provides many capabilities for visualizing objects, robots and trajectories in Rviz as well as debugging tools such as step by step introspection of a script.

A planning scene diff is the difference between the current planning scene(maintained by move_group node) and the new planning scene desired by the user. ROS API to planning scene publisher is through a topic interface using diffs

advertise the required topic
The first line of the picture is shown below
ros::Publisher planning_scene_diff_publisher =
node_handle.advertise<moveit_msgs::PlanningScene>("planning_scene", 1);

define attached object message and use this message to add or subtract object from the world and to attach object to robot.
Note that attaching an object to the robot requires the corresponding operation to be specified as an ADD operation.
attached_object.object.operation = attached_object.object.ADD;
we want the collision checker to ignore collisions between the object and the robot hand.
attached_object.touch_links = std::vector<std::string>{ "panda_hand", "panda_leftfinger", "panda_rightfinger" };

Add the object into the environment by adding it to the set of collision objects in the "world" part of the planning scene.
A planning scene represents all the information needed to compute motion plans: The robot's current state, its representation (geometric, kinematic, dynamic) and the world representation.

There are 2 separate mechanisms available to interact with move_group node using diffs:

- Send diff via a rosservice call and block until the diff is applied(synchornous update)
- Send a diff via a topic and continue even though the diff might not be applied yet (asynchronous update) and send diffs to planning scene via service call

When a robot picks up an object from the environment we need to attach the object to the robot . Attaching an object requires to
1) removing the original object form the environment
2) Attaching the object to the robot

Note how we make sure that the diff message contains no other attached objects or collision objects by clearing those fields first.

Detaching an object from the robot requires two operations

- Detaching the object from the robot
- Re-introducing the object into the environment

Removing object from the collision world just requires using the remove object message

In MoveIt the motion planners are loaded using a plugin infrastructure which allows moveit to load motion planners at runtime

ROS MOVEIT SETUP ASSISTANT

The MoveIt Setup Assistant is **a graphical user interface for configuring any robot for use with MoveIt**. Its primary function is generating a Semantic Robot Description Format (SRDF) file for your robot. Additionally, it generates other necessary configuration files for use with the MoveIt pipeline.
The URDF file describes the robot, while the SRDF file complements the URDF and specifies additional information for motion planning

First step is to install the Franka description package for Noetic. We can do that by entering the following prompt -

sudo apt install ros-noetic-franka-description

The panda_moveit_config package has to be removed in order to set up the moveit assistant from scratch. Enter the following prompt in the terminal to do so:

cd ~/ws_moveit/src
rm -rf panda_moveit_config
catkin clean panda_moveit_config

Start the Moveit_Setup Assistant :

roslaunch moveit_setup_assistant setup_assistant.launch

This will bring up the start screen, click on the **Create New MoveIt Configuration Package** option and source the urdf/xacro file that you want to plan the motion for.