Institute for Cognitive Systems
Technische Universität München
Dr.-Ing. Rogelio Guadarrama

Modeling and control of legged robots
Tutorial 1.1: ROS visualizations and spatial algebra

1 Introduction

This tutorial requires basic knowledge of ROS and Python. As a starting point, you can check the basic and intermediate ROS tutorials.

Specifically, make sure you can implement and understand the following tasks

- Create a new ROS workspace and package specifying its dependencies.
- Build a ROS workspace by setting up the fields in the CMakeLists.txt
- Write a Python ROS node with a publisher and a subscriber

For the spatial algebra required in this tutorial, we will use the equations from Lecture 1 and the `numpy` and `pinocchio` libraries. You can install numpy in Ubuntu 20 with the following command.

```
pip3 install numpy
```

To install the `pinocchio` libraries, you can follow the installation procedure from its main web page. Take a look at the python cheat sheet and the main documentation.

2 ROS workspace setup

a Creating and setting up a ROS workspace

a.1 How to create a new ROS workspace and initialize it using catkin

Start by creating a general folder for all the workspaces you will be developing:

```
$ cd              #Move to the home directory
$ mkdir -p ros/workspaces  #Create a new directory for your ROS workspaces
$ cd ros/workspaces       #Move to this subdirectory
```

Within this folder, create a new ROS workspace:

```
$ mkdir -p my_workspace/src  #Create your workspace directory
$ cd my_workspace       #Switch to the src subfolder.
```

a.2 How to build your ROS workspace

Once your workspace is created, you have to compile it:

```
$ source /opt/ros/noetic/setup.bash  #
$ cd ~/ros/workspaces/my_workspace
$ catkin_make #Compile your workspace
```

Even though the workspace is empty (there are no packages in the "`src`" folder), you can still build it. If you now look at your root directory, you should now have a "`build`" and a "`devel`" folder. The "`build`" folder mainly contains executables of the nodes that are placed inside the catkin workspace "`src`" folder. The "`devel`" folder contains bash script, header files, and executables in different folders generated during the build process. After building the empty workspace, we should set the environment of the current workspace to be visible by the ROS system. This process is called **overlaying a workspace**. Inside the "`devel`" folder you can see that there are now several "`setup.*sh`" files. Sourcing any of these files will overlay this workspace on top of your environment so that you can use it:

```
$ cd ~/ros/workspace/my_workspace
$ source devel/setup.bash #Source your workspace so that ROS is aware of it
```

To make sure your workspace is properly overlayed by the setup script, make sure `ROS_PACKAGE_PATH` environment variable includes the directory you're in:

```
$ echo $ROS_PACKAGE_PATH
#Should give something like:
#/home/youruser/ros/workspaces/my_workspace/src:/opt/ros/kinetic/share
```

b Creating a ROS package

Packages are the most basic unit of the ROS middleware. They contain the ROS runtime process (called nodes), libraries, configuration files, headers and so on, which are organized together as a single unit. The `catkin_create_pkg` command is used to create a ROS package. It has the following syntax:
`catkin_create_pkg [package_name] [dependency 1] [dependency 2] ... [dependency n]`
You can generate a new ROS package within the previously created workspace:

```
$ cd my_workspace/src #Switch to the src subfolder.
$ catkin_create_pkg my_package std_msgs rospy roscpp #Create a new package
```

In this example, the dependency are as follows:

- **roscpp**: is a ROS library which provides APIs to C++ developers to make ROS nodes with ROS topics, services, parameters, and so on. You must include this dependency if you want to write a ROS C++ node.
- **rospy**: is the equivalent of roscpp for Python.
- **std_msgs**: contains basic ROS primitive data types such as integer, float, string, array, and so on. We can directly use these data types in our nodes without defining a new ROS message.

After creating this package, build the package without adding any nodes using the `catkin_make` command. After a successful build, we can start adding nodes to the "`src`" folder of this package.

The standard for ROS packages is to place all the python script in a folder named `scripts` in the root of the package folder. Also, remember to declare all of the scripts you create in the `CMakelists.txt`.

```
catkin_install_python(PROGRAMS
              scripts/python_script_1.py
              scripts/python_script_2.py
DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION})
```

3 Homework

In this tutorial, you need to create ros package called `ros_visuals`, and implement three python nodes that test the spatial algebra formulations learned in Lecture 1. Please note that there are a few questions in the exercises that you must answer in a text file.

a Exercise 1: SE(3)

1. Create a ROS workspace for the tutorial.
2. Create a ROS package called `ros_visuals` with dependency to `roscpp`, `rospy`, `std_msgs`, `geometry_msgs`, `visualization_msgs`.
3. Take a look at figure 1. Are all the frames correct? If not, Which ones are not and why? Write your answers in a questions.txt file.

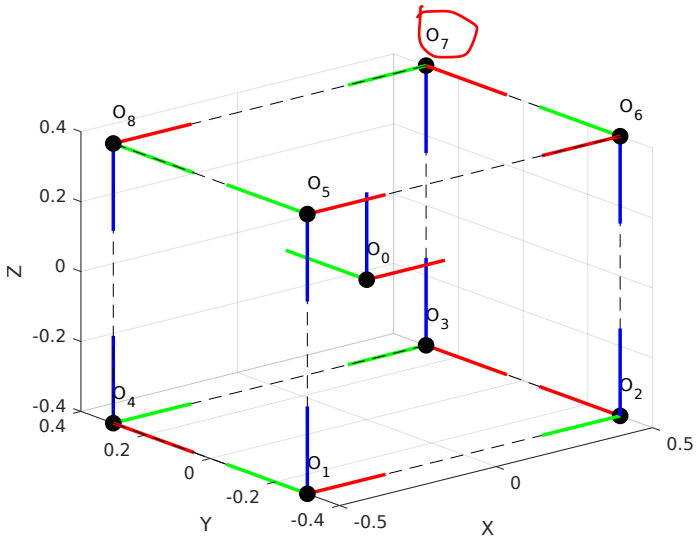

Figure 1: Example cube with coordinate frames at the corners.

4. Create a Python ros node that builds a cage similar to figure 1 but using an array of `pinocchio` **SE**(3) transformations. Name the script **t11.py**
5. Broadcast the Homogeneous transformations that build the cage as ROS tfs adding one more transformation from the `center` frame to the `world` frame as shown in Fig 2. Make sure that the world and the cage are not aligned.
6. Create a launch file for ROS rviz that loads the rviz configuration file from your package. Add the tf tree in the rviz view and all the markers that will be produced in the following steps. Remember to save the configuration file in the name and location you specified in the launch file.

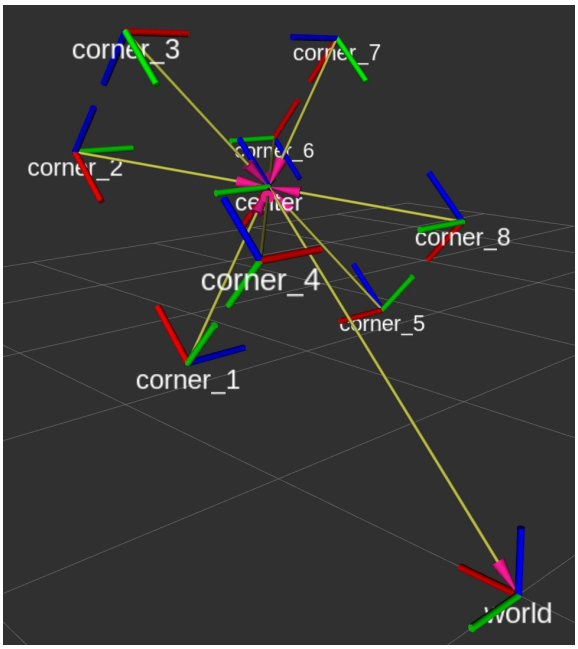

Figure 2: Example cube with coordinate frames at the corners visualized with ROS TFs.

7. Implement a linear and angular integration to make the cage rotate and translate in the with respect to the `world` coordinate frame. Use the formulations from Lecture 1 and the `exp6` function from `pinocchio` and compare the results.
8. Create point vector $\mathbf{p} \in \mathbb{R}^3$ as a `numpy` array. Publish ${}^{c}\mathbf{p}$ with respect to one of the corners of the cage as a ROS `visualization_marker`.
9. Transform ${}^{c}\mathbf{p}$ to the world coordinate frame. Then publish ${}^{w}\mathbf{p}$ with another ROS visualization marker.
10. Are they exactly the same in the rviz view? if you change the markers for cubes, what do you notice regarding the orientation and why is it?

b Exercise 2: Twist

1. Make a copy of **t11.py** and save it as **t12.py**
2. Implement the twist coordinate transformation covered in Lecture 1 as a python function.
3. Define a spatial twist ${}^{c}\mathbf{V} \in \mathbb{R}^6$ as a `pinocchio.Motion` vector you can use any constant vector. However make sure that neither the linear and angular part are zero (e.g. ${}^{c}\mathbf{V} = [\omega_x, \omega_y, \omega_z, v_x, v_y, v_z]^\top = [1.0, 0, 0, 0.5, 0, 0]^\top$). Publish ${}^{c}\mathbf{V}$ with respect to one of the corners of the cage as a ROS `TwistStamped`.
4. Transform ${}^{c}\mathbf{V}$ to the `world` coordinate frame using the function implemented above. Then publish ${}^{w}\mathbf{V}$ with another `TwistStamped`.
5. Repeat the process but defining first the twist ${}^{w}\mathbf{V}$ in the `world` coordinate frame and then transform it to another corner of the cage ${}^{c_2}\mathbf{V}$. Use different colors for the different twists in the rviz visualization.
6. Implement steps 4 and 5 using the `action` matrix of the `pinocchio.SE3` class. Compare the transformed twists from your implementation and `pinocchio` using different twist markers.
7. (Optional) compare your wrench transformation to the output of simply multiplying the `pinocchio.SE3` and `pinocchio.Motion` objects.
8. When you make the cage spin (as in exercise 1), which component of the transformed twists stays constant? why not the other?

c Exercise 3: Wrench

1. Make a copy of **t11.py** and save it as **t13.py**
2. Implement the wrench coordinate transformation covered in Lecture 1 as a python function.
3. Define a spatial wrench ${}^{c}\mathbf{W} \in \mathbb{R}^6$ as a `pinocchio.Force` vector you can use any constant vector but make sure that neither the linear and angular part are zero. Publish ${}^{c}\mathbf{W}$ with respect to one of the corners of the cage as a ROS `WrenchStamped`.
4. Transform ${}^{c}\mathbf{W}$ to the `world` coordinate frame using the function implemented above. Then publish ${}^{w}\mathbf{W}$
5. Repeat the process but defining first the wrench ${}^{w}\mathbf{W}$ in the `world` coordinate frame and then transform it to another corner of the cage ${}^{c_2}\mathbf{W}$. Use different colors for the different wrenches in the rviz visualization.
6. Implement steps 4 and 5 using the `action` matrix of the `pinocchio.SE3` class. Compare the transformed wrenches from your implementation and `pinocchio` using different twist markers.
7. (Optional) compare your wrench transformation to the output of simply multiplying the `pinocchio.SE3` and `pinocchio.Force` objects.
8. When you make the cage spin, which component of the transformed wrench stays constant? why not the other?

4 What to deliver?

This tutorial belongs to the first block deliverable. You must deliver the `ros_visuals` package with all the python scripts, config and launch files including your answers to the questions in a text file.

- Write clean readable code with comments highlighting the sections solving the steps of this tutorial.
- Do not deliver compiled binaries (the `build` and `devel` folders of the workspace). Only deliver the source code and needed files (launch, config, rviz configs, etc.).
- Do not copy the solution from other students.
- You can use any free online tool to help you solve this tutorial.
- It is recommended to use a python debug console (such as the one included in vscode) to solve this tutorial.
- Do not submit the solution of this tutorial yet. After the next two tutorials, a deadline for the first deliverable will be released.