Autonomous Systems - Homework 2

Christoph Killing

October 28, 2021

1 Installing ROS

NOTE:

The following instructions are written for ROS melodic (compatible with Ubuntu 18). If you are using Ubuntu 20 replace every 'melodic' you see with 'noetic'. Do not copy-past commands in this case! You can find the official ROS setup instructions here: http://wiki.ros.org/ROS/Installation

1.1 Getting started with ROS

The Robot Operating System (ROS) is a crucial middleware (a.k.a. collection of software packages) that enables roboticists all over the world to implement their algorithms in a clean and modular fashion and share their work effectively with the community.

In addition, it is at the very core of our class, so we'd better start playing with it!

1.2 Installing ROS

By now, you should have a working (preferably fresh) install of Ubuntu 18.04 and have become accustomed with the basics of Linux, Git and C++. The most efficient way to install ros is through the Debian (binary) packages.

To install ROS Melodic on it, we will follow the official guide to install the Desktop-Full Install option.

1.2.1 Setup repositories

Let's add the packages.ros.org repository to our system

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" >
↪ /etc/apt/sources.list.d/ros-latest.list'
```

and setup the keys

```
sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key
↪ C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
```

1.2.2 Installation

Before installing ROS we need to update the apt index

```
sudo apt update
```

Now let's install ROS

```
sudo apt install ros-melodic-desktop-full
```

1.2.3 Installation

Before using ROS, we need to install and initialize rosdep. We will do that by using pip a package managment system for python

```
sudo apt-get install python-pip
sudo pip install -U rosdep
sudo rosdep init
rosdep update
```

1.3 Environment setup

It's convenient if the ROS environment variables are automatically loaded as soon a new shell is launched, let's edit /.bashrc to do so

```
echo "source /opt/ros/melodic/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

subsectionPython workspace tool We will need the python package to manage ROS workspaces, but this is distributed separately, so we need to install it

```
sudo apt install python-catkin-tools python-wstool
```

2 ROS Intro

2.1 ROS file system structure

2.1.1 General structure

Similar to an operating system, ROS files are also organized in a particular fashion. The following graph shows how ROS files and folder are organized on the disk:

The ROS packages are the most basic unit of the ROS software. They contain the ROS runtime process (nodes), libraries, configuration files, and so on, which are organized together as a single unit. Packages are the atomic build item and release item in the ROS software.

Inside a package we can find the package manifest file, which contains information about the package, author, license, dependencies, compilation flags, and so on. The package.xml file inside the ROS package is the manifest file of that package.

The ROS messages are a type of information that is sent from one ROS process to the other. They are regular text files with .msg extension that define the fields of the messages.

The ROS service is a kind of request/reply interaction between processes. The reply and request data types can be defined inside the srv folder inside the package.

For example, the package we will develop in this lab will be like

```
.
└── two_drones_pkg
    ├── CMakeLists.txt
    ├── README.md
    ├── config
    │   └── default.rviz
```

```
├── launch
│   └── two_drones.launch
├── mesh
│   └── quadrotor.dae
├── package.xml
└── src
    ├── frames_publisher_node.cpp
    └── plots_publisher_node.cpp
```

2.1.2 The workspace

In general terms, the workspace is a folder which contains packages, those packages contain our source files and the environment or workspace provides us with a way to compile those packages. It is useful when we want to compile various packages at the same time and it is a good way of centralizing all of our developments.

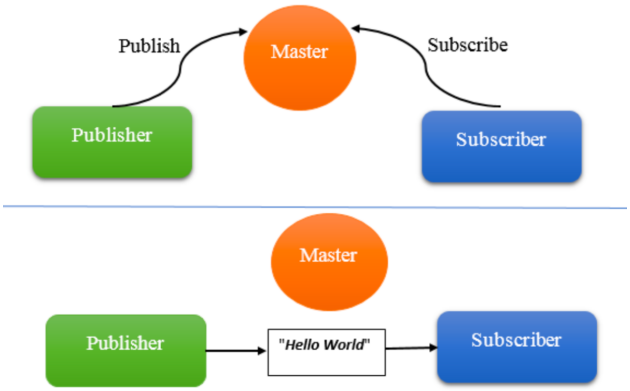
2.2 ROS Master, nodes and topics

One of the primary purposes of ROS is to facilitate communication between the ROS modules called nodes. Those nodes can be executed on a single machine or across several machines, obtaining a distributed system. The advantage of this structure is that each node can control one aspect of a system. For example you might have several nodes each be responsible of parsing row data from sensors and one node to process them.

2.2.1 ROS Master

Communication between nodes is established by the ROS Master. The ROS Master provides naming and registration services to the nodes in the ROS system. It is its job to tracks **publishers** and **subscribers** to the **topics**.

ROS master works much like a DNS server. Whenever any node starts in the ROS system, it will start looking for ROS master and register the name of the node with ROS master. Therefore, ROS master has information about all the nodes that are currently running on the ROS system. When information about any node changes, it will generate a call back and update with the latest information.



ROS Master distributes the information about the topics to the nodes. Before a node can publish to a topic, it sends the details of the topic, such as its name and data type, to ROS master. ROS master will check whether any other nodes are subscribed to the same topic. If any nodes are subscribed to the same topic, ROS master will share the node details of the publisher to the subscriber node.

After receiving the node details, these two nodes will interconnect using the TCPROS protocol, which is based on TCP/IP sockets, and ROS master will relinquish its role in controlling them.

To ROS master, open a terminal and run

```
roscore
```

Any ROS system must have only one master, even in a distributed system, and it should run on a computer that is reachable by all other computers to ensure that remote ROS nodes can access the master.

2.3 Anatomy of a ROS node

The simplest C++ ROS node has a structure similar to the following

```
#include "ros/ros.h"

int main(int argc, char **argv) {
  ros::init(argc, argv, "example_node");
  ros::NodeHandle n;

  ros::Rate loop_rate(50);

  while (ros::ok()) {
    // ... do some useful things ...
    ros::spinOnce();
    loop_rate.sleep();
  }
  return 0;
}
```

Let's analyze it line by line: the first line

```
#include "ros/ros.h"
```

adds the header containing all the basic ROS functionality. At the beginning of the main of the program

```
ros::init(argc, argv, "example_node");
```

ros::init initialize the node, it is responsible for collecting ROS specific information from arguments passed at the command line and set the node name (remember: names must be unique across the ROS system). But it does not contact the master. To contact the master and register the node we need to call

```
ros::NodeHandle n;
```

When the first ros::NodeHandle is created it will call ros::start(), and when the last ros::NodeHandle is destroyed (e.g. goes out of scope), it will call ros::shutdown(). This is the most common way ot handling the lifetime of a ROS node.

Usually we want to run our node at a given frequency, to set the node frequency we use

```
ros::Rate loop_rate(50);
```

which is setting the desired rate at 50 Hz. Than we have the main loop of the node. Since we want to run this node until the ROS we need to the check the various states of shutdown. The most common way to do it is to call ros::ok(). Once ros::ok() returns false, the node has finished shutting down. That's why we have

```
while (ros::ok()) {
  // ...
}
```

Inside the loop we can make interesting thing happen. In our example we simply run

```
ros::spinOnce();
loop_rate.sleep();
```

The function ros::spinOnce() will call all the callbacks waiting to be called at that point in time while. If you remember we set the node frequency to 50Hz, the code we are running will probably take less than 20ms. The function loop_rate.sleep() will pause the node the remaining time.

2.4 Launch files

Launch files are the preferred way to run ROS nodes. The launch files provide a convenient interface to execute multiple nodes and a master (if is not already running), as well as other initialization requirements such as parameters.

Usually the launch files are located in the launch folder of the package and have .launch extension. If the package provide one you can use roslaunch to use it.

```
roslaunch <package_name> <launch_file>
```

Note: Pushing CTRL+c in a terminal with a launch file running will close all nodes that were started with that launch files.

An example of launch file is

```
<launch>
  <node name="map_server" pkg="map_server" type="mapserver" />
  <node name="stageros" pkg="stage" type="stageros" args="$(find navigation_stage)/stage
  ↪  config/worlds/willow-pr2-5cm.world" >
    <param name="base_watchdog_timeout" value="0.2"/>
  </node>
  <include file="$(find navigation_stage)/move_base_config/amcl_node.xml"/>
</launch>
```

This example will run three nodes (plus the master if not already running), each ¡node¿...¡/node¿ is equivalent to a rosrun call, for example

```
<node name="stageros" pkg="stage" type="stageros" args="$(find navigation_stage)/stage
↪  config/worlds/willow-pr2-5cm.world" />
```

is equivalent to

```
rosrun stage stageros
↪  <path-to-navigation-stage-package>/stage_config/worlds/willow-pr2-5cm.world
```

This tool gives the possibility to add complex and dynamic runtime behavior such as $(find path), unless and if or include other launch files.

3 Exercises

To complete this exercise, you will have two weeks, until November 11th, 2021. We also extend the group registration deadline to November 10th, 23:59. After that, you will start working on problems in your groups.

3.1 Setup workspace

3.1.1 Create the catkin workspace

To get started, we must create a ROS workspace for the Autonomous Systems class. Choose your favorite working directory, we will assume you are creating one in your home directory (i.e ~/). In a terminal, run:

```
$ mkdir -p ~/as_ws/src
$ cd ~/as_ws/
$ catkin init
Initializing catkin workspace in `/home/antonap/as_ws`.
------------------------------------------------------------
Profile:                     default
Extending:             [env] /opt/ros/melodic
Workspace:                   /home/antonap/as_ws
------------------------------------------------------------
Build Space:       [missing] /home/antonap/as_ws/build
Devel Space:       [missing] /home/antonap/as_ws/devel
Install Space:      [unused] /home/antonap/as_ws/install
Log Space:         [missing] /home/antonap/as_ws/logs
Source Space:       [exists] /home/antonap/as_ws/src
DESTDIR:            [unused] None
------------------------------------------------------------
Devel Space Layout:          linked
Install Space Layout:        None
------------------------------------------------------------
Additional CMake Args:       None
Additional Make Args:        None
Additional catkin Make Args: None
Internal Make Job Server:    True
Cache Job Environments:      False
------------------------------------------------------------
Whitelisted Packages:        None
Blacklisted Packages:        None
------------------------------------------------------------
Workspace configuration appears valid.
------------------------------------------------------------
```

3.2 Getting the Lab code

Go the folder where you cloned the Labs codebase and run git pull. This command will update the folder with the latest code. Let's suppose we have the codebase in ~/Labs. In ~/Labs/Lab_2/code you now have the two_drones_pkg folder, which is a ROS package. Copy this folder in your **Autonomous Systems** (e.g. **as_ws**) workspace and build the workspace as follows:

```
cp -r ~/autonomous-systems-2020/Labs/Lab_2/code/two_drones_pkg ~/as_ws/src
```

3.3 Building the code

Building the code is as easy as running:

```
catkin build
```

Now that you built the code you see that catkin added a bunch of new folders. In order to use our workspace, we need to make ROS aware of all the components by sourcing the corresponding environment. This is done by running the following in **every single terminal** where you intend to use the workspace:

```
source devel/setup.bash
```

For the rest of the assignment, we assume that you are performing this operation whenever necessary.

3.4 A two-drone scenario

In this part, we are going to work with 3D Rigid Transformations and with tf, a basic tool provided by ROS to keep track of multiple coordinate frames over time. To get started, let's go back to the **Autonomous Systems** workspace and bring up the two-drones static scenario. In this environment, we have two aerial vehicles, AV1 [blue] and AV2 [red] that are not moving, but it serves as a good starting point! With the **Autonomous Systems** workspace sourced in a terminal, run:

```
roslaunch two_drones_pkg two_drones.launch static:=True
```

You should see the following window, which shows the initial positions of the two AVs.

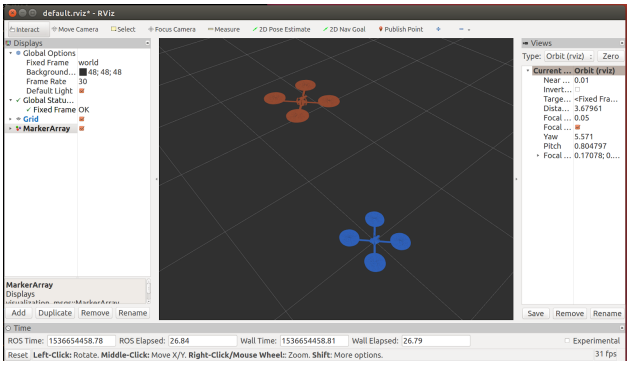


Figure 1: Static quads

This window is rViz, the mighty ROS visualizer! Just like most processes in the ROS ecosystem, rViz is a ROS node. Like all other nodes, rViz can subscribe to ROS topics. However, its specialty is converting them into graphics on your screen to display the state of your robotic system.

As a first experience with rViz, let us:

- Add the visualization of \tf. In the Displays panel, click Add, select the By display type Tab in the pop-up and finally select "TF" and confirm with Ok. You should see all the reference frames, with names and their axes represented in red (x), green (y) and blue (z).
- Save the configuration file. So that we don't have to repeat the step above every time we launch it! Hit CTRL + s or select File > Save Config.

Other published topics can be added to the visualizer in a similar way.

3.5 Problem formulation

We consider the scenario illustrated in the picture below, where two aerial vehicles, AV1 [blue] and AV2 [red] are following different trajectories: a circle and an arc of parabola, respectively. In the scene, we have highlighted the following elements:

- The world frame x_w, y_w z_w

- The AV1 body frame, centered in O_1 with axes $x_1,\ y_1\ z_1$
- The origin of AV2, denoted with O_2 with axes $x_2,\ y_2\ z_2$

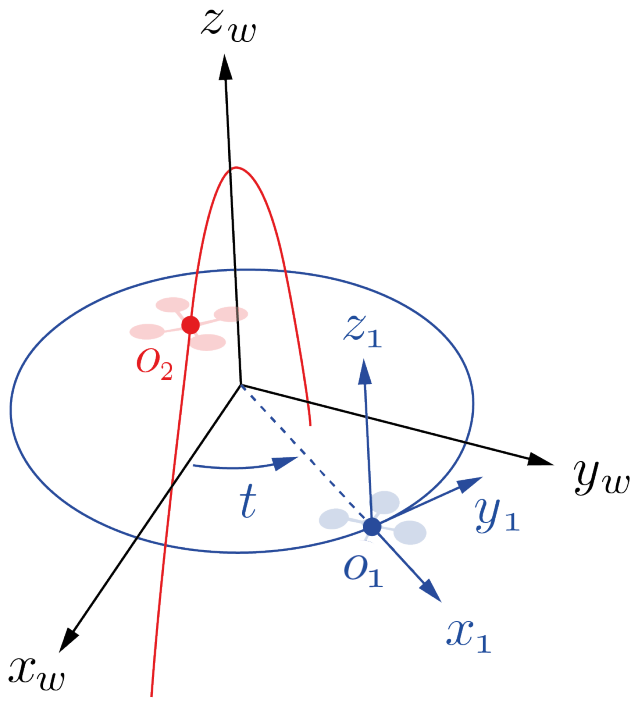


Figure 2: Static quads

Positions:
In the world frame, AV1 and AV2's origins are given by:

$$o_1^w = \begin{bmatrix} cos(t) \\ sin(t) \\ 0 \end{bmatrix}, \quad o_2^w = \begin{bmatrix} sin(t) \\ 0 \\ cos(2t) \end{bmatrix} \tag{1}$$

where t denotes time.

Orientations:
We will make the following simplifying assumptions:

- AV1's reference frame is such that y_1 stays tangent to AV1's trajectory for all t and z_1 is parallel to z_w for all t (i.e., equivalently, roll = pitch = 0, yaw = t)
- AV2's reference frame moves with pure translation and we can assume that its axes are parallel to the world axes for all times t

NOTE:

- Given the dynamics of a quadrotor, these motions are dynamically infeasible. However, for the purpose of this lab, we disregard this fact and focus on the study of rigid transformations
- To make the math of this problem more interesting, we chose the y_1 axis to point in the direction of motion of the drone. However, do not forget that the standard convention is that x_1 should point forward!

In the sequel, we reproduce the above scenario in ROS and study the trajectory of AV2 relative to AV1's coordinate frame.

3.6 Basic ROS commands

3.6.1 Task 1 - Nodes, topics, launch files

With the roslaunch command above we have spawned a number of ROS nodes at the same time. Using your knowledge of ROS, answer the following questions:

1. List the nodes running in the two-drone static scenario.
   Hint: you can directly inspect the launch file or get some help from rqt_graph. You will notice that rViz is initially not shown in it but you can uncheck the Debug option for a full picture, feel free to ignore all \rosout nodes and topics and \rqt_gui_py_* that may appear

2. How could you run the two-drone static scenario without using the roslaunch command? List the commands that you would have to execute, in separate terminals, to achieve the same result.
   Hint: rosrun [...], try things out before finalizing your answer!

3. List the topics that each node publishes \ subscribes to. What nodes are responsible for publishing the av1, av2, frames? Which topic causes rViz to plot the drone meshes?
   Hint: uncheck items on the left pane in rViz until the meshes disappear, then check what node is publishing the corresponding topic

4. What changes if we omit static:=True? Why? *Hint: check out and briefly explain the if and unless keywords in the launch file*

3.7 Let's make things move! Publishing the transforms using tf

After exploring the static scenario, it's time to implement the motions described in the problem formulation section and visualize them in rViz. With the editor of your choice, open frames_publisher_node.cpp in the src folder of two_drones_pkg. In this file, we provide a basic structure of a ROS node.

3.7.1 Some Context

Please take your time to familiarize with this code before modifying it. Although not mandatory, the pattern found in it is a very common way to implement ROS nodes:

- The node's source code is encapsulated into a class, FramesPublisherNode (line 6), which keeps a nodeHandle as a private member.
- In the constructor of the class (lines 14 to 19), one performs operations that need to be executed only once upon startup (e.g. typically, initializing subscribers and publishers),
- Using a Timer (lines 10 and 17), the onPublish() method is called periodically - at a 50Hz - and all the operations within it are performed ad libitum,
- In the body of main() (towards the end of the file):
  - the node is registered with ros::init(...)
  - an instance of the node class is created
  - a blocking call to ros::spin() is issued, which starts ROS's main loop.

3.7.2 Task 2 - Publishing transforms

In frames_publisher_node.cpp, follow the instructions in the comments and fill in the missing code. Your objective is to populate the provided AV1World and AV2World variables to match the motions described in the problem formulation. These objects are instances of the tf::transform class, which is ROS jargon for a homogeneous transformation matrix.

Keep in mind:

Ensure that the orientation of the AV1 frame is in accordance with the assumptions made in the problem formulation, as this is of crucial importance for the final result!

How to test:
Once you are ready to compile your code, run:

```
catkin build
```

from the workspace folder ∼/as_ws.

To try out your code, launch the two-drone scenario in non-static mode, i.e. run:

```
roslaunch two_drones_pkg two_drones.launch
```

What to expect You should finally see the drones moving! Check that the trajectories reflect those illustrated in the figure in the problem formulation.

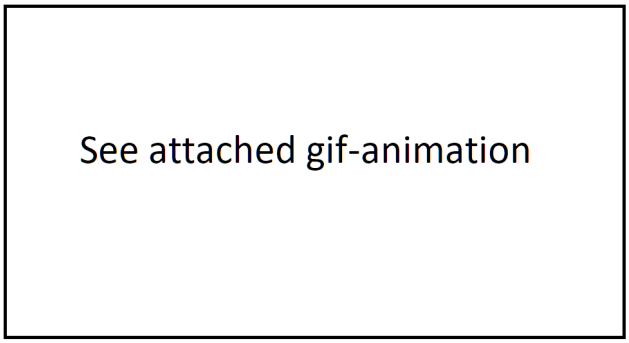


Figure 3: Flying quads

3.7.3 Changing the rViz fixed reference frame.

As mentioned, we are interested in the motion of AV2 relative to AV1's reference frame. In the Displays panel (left, by default), under the Global Options section, rViz offers the possibility to change the Fixed Frame to any frame published in tf. Try it out yourself and change "world" into "av1" by typing in the corresponding field. From this perspective, AV1 appears static, the world frame spins around its z axis and AV2 seems to be following a closed-curve trajectory.

3.7.4 Task 3 - Looking up a transform

In plots_publisher_node.cpp, follow the instructions in the comments and fill in the missing code. Your objective is to populate the provided object, transform, with the relative transform between two given frames with names ref_frame and dest_frame.

Compile your code and try it out as previously explained.

What to expect:
You should eventually see three trajectories, namely:

- AV1's trajectory [blue, solid] in the world frame (circle on the x-y plane)
- AV2's trajectory [red, solid] in the world frame (parabola on the z-x plane)
- The trajectory of AV2 in AV1's frame [red, dashed]. You should now have a strong hunch that this curve is a ellipse on a "slanted" plane!

Note: if the results you are observing do not seem to make sense, try swapping ref_frame and dest_frame when interrogating \tf.