Exercises

The tutorial is structured in exercises of increasing involvement. The initial tasks (`0.x`), that deal with the setting up of the working environment, are **strongly recommended** to be carried out before the day of the tutorial.

In particular, setting up the *Webots* simulator requires a large download of ~1.5GB (see task `0.3`).

If you find any issues with preparations for the tutorial, please contact me at `amosteo@unizar.es` or open an issue at https://github.com/ada-ros/tutorial-aeic21/issues.

Exercises

- 0. Setup of the working environment
  - 0.1 ROS2 setup
    - A note on terminals
  - 0.2 RCLAda setup
  - 0.3 Extra tools
  - Alternatives: gitpod
  - Alternatives: docker
  - Alternatives: VirtualBox image
- 1. Ada development environments for ROS2
  - 1.1 Creating a new ROS2 package
  - 1.2 Plain build & edit with colcon
    - In desperate times
  - 1.3 Create a GNAT project for your package
    - Creating the Ada project
  - 1.4 Edit with VSCode
  - 1.5 Edit and compile with GNATstudio/GPS
  - Conclusion
- 2. Blackboard communication (topics/publishers/subscriptions)
  - Using RCLAda dynamic vs static messages
  - 2.1 Writing a Publisher, dynamic version
  - 2.2 Writing a Publisher, static version
  - 2.3 Writing a Subscriber, dynamic version
  - 2.4 Writing a Subscriber, static version
  - TurtleSim: first steps
  - 2.5 Launching a *TurtleSim* and identifying relevant information
  - 2.6 Move the turtle in a fixed sequence
  - 2.7 Create a remote controller for the turtle
  - 2.8 Drive the *ePuck* robot
  - Conclusion
- 3. RPC communication (services/servers/clients)
  - 3.1 Servers
  - 3.2 Asynchronous clients
  - 3.3 Synchronous clients
  - 3.4 Control the TurtleSim with service calls
  - Conclusion
- 4. Realistic exercises
  - Wandering robot with obstacle avoidance (Roomba-like)
  - 4.1 Wandering ePuck
  - 4.2 Wandering TurtleBot
  - 4.3 Navigate to a local goal (VFH local planner)
  - 4.4 Working with reference frame transformations
  - Conclusion

0. Setup of the working environment

Firstly, we will set up a plain ROS2 environment. With this, we would be ready to develop ROS2 packages in C, C++ or Python. Subsequently, we will set up the Ada-specific environment.

This tutorial has been designed to be carried out on Ubuntu 20.04 LTS.

0.1 ROS2 setup

The simplest way of setting up ROS2 in Ubuntu is from binary packages. Follow the instructions at https://docs.ros.org/en/foxy/Installation/Ubuntu-Install-Debians.html, which are summarized here:

1. Ensure your system locale supports UTF-8 (`locale` command).
2. Install pre-requisite packages:
    1. `sudo apt update && sudo apt install curl gnupg2 lsb-release`
3. Set up the ROS2 keys and apt origin:
    1. `sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -o /usr/share/keyrings/ros-archive-keyring.gpg`
    2. `echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/ros-archive-keyring.gpg] http://packages.ros.org/ros2/ubuntu $(lsb_release -cs) main" | sudo tee /etc/apt/sources.list.d/ros2.list > /dev/null`
4. Install the ROS2 packages:
    1. `sudo apt update`
    2. `sudo apt install ros-foxy-desktop python3-argcomplete`

Our shell is unaware of ROS2 tools until we activate some ROS version. This is done by sourcing the `/opt/ros/<version>/setup.bash` file (a `zsh` alternative exists too):

> `source /opt/ros/foxy/setup.bash`

Afterwards, we can verify our system is ready by running, for example, the command `ros2` which should now be in our `PATH`.

The `ros-foxy-desktop` package will install a complete suite of packages that also include demos and tutorials. We are going to use some of these, but otherwise a more limited set of packages could be installed with `ros-foxy-ros-base`.

ROS/ROS2 distributions use packages with the prefix `ros-<version>-*`. You may see other ROS2 versions with their base Ubuntu system and support life at https://docs.ros.org/en/rolling/Releases.html.

ROS Foxy, which is used in this tutorial, is the current Long Term Service (LTS) ROS2 distribution. Very recently the next non-LTS version was released. Also, since June 2020, a rolling distribution exists on which development is carried out. RCLAda is developed on the latest LTS release.

A note on terminals

Development for ROS/ROS2 tends to require multiple commands run from several terminals. For this reason, it is advisable to use a terminal that allows splitting it into sub-terminals. A graphical popular option is `terminator`. An alternative that will work in most consoles, even non-graphical ones, is `tmux`, although this one is a bit more complex to learn as it requires learning shortcuts. (See, e.g., https://tmuxcheatsheet.com/, in particular the section about *"panes"*.)

Likewise, a terminal that can be called/dismissed quickly may result convenient to some people. A personal favorite (which also supports splitting) is `guake`, bound to some unused function key (e.g. `F12`).

0.2 RCLAda setup

With the previous steps, and after sourcing the `/opt/ros/foxy/setup.bash` script, we are ready to set up the Ada environment. Full instructions can be found at https://github.com/ada-ros/ada4ros2 and are summarized next:

1. Install the native Ada build tools:
    1. `apt install gnat gnat-gps gprbuild`
    - GNAT CE 2020/2021 will not work as it will complain about a few issues in the Ada codebase.
    - Even after fixing those, you may experience linking problems when mixing code built with the native `g++` and the one packaged with GNAT CE 2020.
    - You **may** use the GNATstudio editor from the Community Edition, if so you prefer, over the older GPS packaged in ubuntu as `gnat-gps`.
2. Clone the RCLAda sources, including submodules, and using the `aeic21` branch:
    1. `git clone --recurse-submodules -b aeic21 https://github.com/ada-ros/ada4ros2`
    - The `ada4ros2` is a mostly empty repository, that is used to bring in several ROS2 packages as submodules. It also contains a few convenience scripts, but nothing necessary *per se* to use RCLAda.
    - The Ada ROS2 packages are detailed in the first part of the tutorial presentation. You can find them under the `src` folder of the repository.
3. Enter the cloned repository:
    1. `cd ada4ros2`
4. Ensure the ROS2 environment is loaded: `source /opt/ros/foxy/setup.bash`
    - There is no ill effect by sourcing the script several times.
5. Build the Ada for ROS2 packages:
    1. `colcon build`
    - `colcon` is the build tool used by ROS2. It is an 'orchestrator' of the build, but it does not mandate a particular method for building a package. It determines dependencies and dispatches to the build method of each package.
    - The build should finish without errors; otherwise we cannot continue.
        - To retry a build from scratch, delete the `build` and `install` folders at the repository root.
    - As part of the build process, `colcon` will create a new `install/setup.bash` script under the current folder.
6. Load the new environment that includes the just-compiled Ada packages:

    1. `source install/setup.bash`
    - ROS2 environments are properly layered, so when the `install/setup.bash` script is generated by the build process, it also will load the pre-existing environment; in this case, the base ROS2 environment. This means that, in a new terminal, it is enough to source the `ada4ros2/install/setup.bash` script to be ready to go.
7. Verify the build succeeded by running, for example:
    1. `ros2 run rclada rclada_selftest_dynamic`
    - Although there may be an error message about a timer, as long as no exceptions are raised, everything is fine.
    - The run should end with a report on allocated memory similar to this one:

```
Total allocated bytes :  425
Total logically deallocated bytes :  425
Total physically deallocated bytes :  0
Current Water Mark:  0
High Water Mark:  425
```

If you experience difficulties setting up the environment, or do not have an Ubuntu base system, there are a couple of alternatives you may try. These alternatives are described in the "Alternatives" section.

0.3 Extra tools

A few useful packages are left out by the ros2 install desktop installation.

1. Install extra packages by running:
    1. `sudo apt install ros-foxy-rqt* ros-foxy-webots-ros2
2. Source the `setup.bash` script again to include these packages:
    1. `source /opt/ros/foxy/setup.bash`
3. Install the *Webots* simulator by running (ensure you have a few GBs of free space beforehand):
    1. ros2 launch webots_ros2_epuck robot_launch.py
        - You'll receive a prompt asking to install the simulator at a default location. After accepting, a large download will occur. After the installation, a simulator window should open if everything was installed correctly. You can close this window before continuing.

Alternatives: gitpod

GitPod is a remote development environment that allows to run VSCode in a browser, with a particular environment generated with Docker underneath. This approach serves for the first exercises, that do not require graphical windows. For this reason, it is recommended as a last resort for this tutorial, or as a temporary measure while the previous instructions are completed.

The gitpod service is free for open source projects, but it requires an account on Github/Gitlab/Bitbucket.

Launch the prepared gitpod environment by following this link:

> https://gitpod.io/#https://github.com/ada-ros/ada4ros2/tree/feat/tut-21

An error in the terminal of VSCode like this:

```
ls: cannot access '/home/gitpod/.bashrc.d/*': No such file or directory
```

comes from the own Gitpod service and is inoffensive.

The Gitpod session already has ROS2 preinstalled, and the underlying Docker is an Ubuntu 20.04. Thus, it is enough to do the following in the VSCode terminal to catch up to the end of the RCLAda setup section:

1. `source /opt/ros/foxy/setup.bash`
2. `colcon build`

Alternatives: docker

NOTE: this approach is only recommended for users already familiar with docker, and that already have docker installed.

A slightly more powerful alternative to GitPod is to use a Docker image that already contains the ROS2 environment. This may allow to successfully follow the tutorial in another Linux other than Ubuntu 20.04. The image tag is `mosteo/ada4ros2:foxy`

This image contains some graphical packages (like `turtlesim`) which enable the realization of a few more exercises. Still, the exercises involving the full blown simulator cannot be carried out this way.

To simplify the task of running graphical applications inside a docker, the recommended approach is to use the `rocker` tool. This tool is distributed via the same repositories as ROS2. Hence, after setting up the ROS2 sources (task `0.1`, step 3), it can be installed by running:

1. `sudo apt install python3-rocker`

Afterwards, the recommendation is to launch a terminator window from the docker, and keep this terminal always open:

1. `rocker --x11 --user --home mosteo/ada4ros2:foxy terminator`
    - The `--user` option may not work and can be left out if you get an error. In that case, the terminal will be open with root as the user, and the first step should be to create a regular user with `adduser <username>`, and then change to it with `su <username>`.
    - Note that the above command will mount your home directory inside the docker, at `/home/<your user>`. Thus, it is dangerous to run commands as root in that environment.

This solution already has the ROS2 environment loaded. (No need to source the `/opt/ros/foxy/setup.bash` script.)

Alternatives: VirtualBox image

An image containing all the necessary software is (or will be shortly) available at https://github.com/ada-ros/tutorial-aeic21/releases. The performance of the simulator in the VM is highly dependent on the capabilities of the host machine. It is recommended to test the suitability of this approach in advance.

1. Ada development environments for ROS2

In this section you will create your first ROS2 package containing a GNAT project. Afterwards, a couple of options for the edition/compiling/running cycle will be examined.

In the following, it is assumed that you are working inside a clone of the `ada4ros2` repository, as described in task 0.2. This may be located anywhere, but for the rest of the exercises the documentation will assume this repository is checked out at `$HOME/ada4ros2`, or `~/ada4ros2` for short.

1.1 Creating a new ROS2 package

ROS2 packages require the following elements to be ready to work:

1. A `package.xml` file that describes the package and allows `colcon` to invoke the proper build method on it, after all its dependencies have been built.
2. Build-method-specific files that carry out the actual compilation. In the case of C++/Ada packages, this is achieved by a CMake `CMakeLists.txt` file.
3. For Ada packages, in addition, a `gprbuild` project has to be provided.

Steps 1 and 2 can be jump-started by using the `ros2 pkg` subcommands. Follow these steps to create your first Ada ROS2 package:

1. Run `ros2 pkg create` without more arguments to get an idea of the parameters that can be used to create a package.
2. Create a package by running
    1. `ros2 pkg create my_ada_package --dependencies rclada_common --description "My first Ada ROS2 package"`
        - This command will create a `my_ada_package` folder under the folder where the command was run. By convention, packages are either at the root of the repository or under the `src` folder. You can leave the created `my_ada_package` folder at its current location or move it under `src`. This does not make a difference for `colcon`.
        - By convention, all ROS2 package names have to be in lowercase. If we were to use `my_Ada_package` we would get nagging warnings about it.
3. Verify the package was correctly created and is detected by `colcon` by running
    1. `colcon graph`
        - The previous command should show output similar to:

```
rclada_common             +***...***
my_ada_package_            +
rosidl_generator_ada        +*....**
rclada                       +******
rclada_client_skeleton        +
rclada_examples                +
rclada_fosdem20                 +
rclada_tf2                       +**
tutorial_exercises                +
tutorial_solutions                 +
```

        - Note how our package depends on `rclada_common`, as we requested during creation.
    2. Examine the `my_ada_package/package.xml` file to see how the description and package version could be updated; and how more dependencies could be added.

In the following exercises we will see a few alternatives for editing our package. For now, we have a plain C++ package; soon we will turn it into an Ada package in task 1.3.

1.2 Plain build & edit with colcon

The simplest way of working is by using any editor and compiling/running from the command line. Try the following alternatives:

1. Simply rebuild all packages in the workspace:
    1. `colcon build`
       This is an expensive option because it will rebuild all packages in the workspace. Although some recompilations are avoided by CMake and `gprbuild`, the process still can be time consuming. Also, some files are regenerated and are compiled every time.
2. Build our package and all its transitive dependencies:
    1. `colcon build --packages-up-to my_ada_package`
       This is a slightly better choice since any packages not needed by our target package are not recompiled.
3. Build just our package:
    1. `colcon build --packages-select my_ada_package`
       This option is the fastest one, but can only be used once all dependencies have been previously compiled and sourced with `source install/setup.bash`.

After a successful build, the environment must be updated so executables and libraries are properly located:

1. `source install/setup.bash`
   Test that ROS2 now detects our package, even if it does not yet build any executables or libraries. Try to autocomplete the package name by typing:
2. `ros2 pkg prefix my[⇥TAB]`
   And the response should be something similar to `/home/user/ada4ros2/install/my_ada_package`

In desperate times

At some point you may find yourself in a situation where compilation throws strange errors, the GNAT projects don't load, and the environment seems broken. At these times the safest fallback is to rebuild from scratch. To that end, you need to delete all build products and recompile with the initial ROS2 environment:

1. `source /opt/ros/foxy/setup.bash`
2. `rm -rf build install`
   Beware that you do not delete anything unintended with the previous command! This is to be run from the repository root.
3. `colcon build`
4. `source install/setup.bash`

1.3 Create a GNAT project for your package

If you examine the `my_ada_package/CMakeLists.txt`, you will see that there are no useful things built. Indeed, this file can be stripped down to the bare minimum:

```
# CMakeLists.txt

cmake_minimum_required(VERSION 3.5)
project(my_ada_package)

find_package(ament_cmake REQUIRED)
find_package(rclada_common REQUIRED)
```

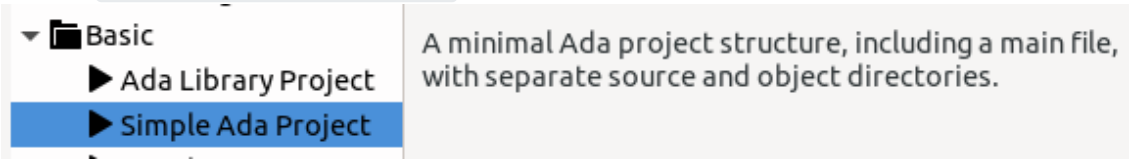
1. Do so and remove all lines but the ones show above.
   NOTE: in particular, the line `ament_package()` *must* be removed.

The `package.xml` and `CMakeLists.txt` are independent; the former express dependencies for colcon, while the latter does the same for CMake. Hence, any dependency additions have to be manually made in both files.

Creating the Ada project

The RCLAda project defines CMake functions to simplify integration of GNAT projects. Firstly, we will create an empty Ada project. Do it the way you would normally do. For example:

- If you have `alr` installed, enter the `my_ada_package` folder and run `alr init --bin ada_code`. This will create an `ada_code` nested folder with a ready-to-use GNAT project.
- If you do not have `alr`, you can run `gnat-gps` or `gnatstudio` and, in the welcome window, choose `"Create a new project..."`. Then, select the basic simple Ada project:



  - It is recommended to place the new project in its own subfolder. This way, several projects can cohabit inside one ROS2 package.

Verify that you can build the project, either using the GUI options or by running `gprbuild` inside the project folder.

We need now to include this project in the build process of ROS2. To do so,

1. Add the following lines to the end of the `my_ada_package/CMakeLists.txt` file:

```
ada_begin_package() # Retrieves Ada context from other Ada packages that are dependencies

ada_add_executables(
    ada_code    # This is a CMake target name, so anything goes.
    ${PROJECT_SOURCE_DIR}/ada_code
                # ABSOLUTE path to the GNAT project root inside the package.
                # By using ${PROJECT_SOURCE_DIR} provided by CMake, we ensure the proper
                #   location is picked even if the repository is moved.
    obj         # Relative path to the binary output, as defined in the GPR project file.
    main        # Executables built by your Ada project, without path.
    )

ada_end_package()   # Exports the updated Ada context for possible dependent packages.
```

NOTE that the second argument to `ada_add_executables` must be adjusted for the place were you created your GNAT project inside the package. `${PROJECT_SOURCE_DIR}` points to the `CMakeLists.txt` file containing folder, so it is handy for this purpose.

NOTE that the third argument is the relative path defined in your GPR file with the `for Object_Dir use "obj";` (default location for projects created with `gnat-gps` or `or Executable_Dir use "bin";` (default location for projects created with `alr`).

> The `ada_add_executables` ensures that the Ada environment is imported/exported properly in regard to other Ada ROS2 packages. Also it places the resulting executables where ROS2 expects to find them.

2. Run `colcon build --packages-up-to my_ada_package` until you get no errors. If you *do* get errors, remove the `build` and `install` folders before invoking colcon, as otherwise old versions of your `CMakeFiles.txt` can be used unexpectedly.
3. Source the environment:
   `source install/setup.bash`
4. Run your executable (adjust for the name you actually used for it -- also try using TAB completion):
   `ros2 run my_ada_package main`
5. Edit your main file so it prints something, and verify the changes by rebuilding and running your very first ROS2 executable.

CONGRATULATIONS! It may seem a trivial step, but having your Ada code built in the context of other ROS2 packages is the very first step needed to build actual robotics code.

1.4 Edit with VSCode

Now that we have a foundation to start doing actual code, usually we will do so using a GUI (if you are happy with editors like `vim`, you obviously can simply go ahead with it and build from another terminal.

Assuming you want a graphical alternative, the first stop may be the now popular VSCode. It has the advantage of many plugins (including one for Ada), and flexible task definitions. Also, as VSCode does not depend on successfully loading a GNAT project file, it is an excellent fallback when problems arise with GNATstudio/GPS in that regard.

> If you wonder why a project file would not load, the trouble in the ROS2 context is that there is file generation involved. Any error in the `colcon build` process may leave us with an incomplete set of files/projects, and *opening* a project in that situation with GNAT is impossible.

If you do not have VSCode installed, you can get it with:

- `snap install code`

The recommended route is to use `File -> Open folder...` to open the `ada4ros2` repository root, and have easy navigation around all repository files in all the contained ROS2 packages.

There are already some tasks defined in the `ada4ros2/.vscode` folder to build, clean & build, update the build and build only the current file by using `Terminal -> Run task...`

Another straightforward possibility is to run `colcon` from the terminal embedded in VSCode.

> VSCode by itself, with the Ada plugin, has some awareness of Ada syntax. To obtain full advantages from the Ada Language Server, a project file should be defined, in `.vscode/settings.json`, entry `ada.projectFile`. This has already been set up for this tutorial. This project file should provide the complete environment, or the environment should exist when launching `code` from the command line. For this reason, it is strongly recommended to open VSCode after a successful `colcon build`.

1.5 Edit and compile with GNATstudio/GPS

You may start to realize that running `colcon build` every time and then navigating to errors by hand is going to be a suboptimal experience. The optimal situation, if you are already an user of GNATstudio/GPS, would be to be able to use them as with regular Ada standalone projects.

For an isolated project like the one we just created in the preceding exercises, there is indeed nothing preventing doing exactly that. The trouble starts when we are using Ada projects in other packages (as we will shortly doing to use the `RCLAda` binding, for example). `colcon build`, for starters, does an out-of-tree build in the `./build` folder; the final executables, libraries and generated GPR project files are furthermore located in the `./install` folder. This results in a fragile situation in which only after a complete successful build we can use GNATstudio/GPS (as they require a valid GPR project, with all the dependencies being available).

Just as an example, we will open the project containing the solutions to the remaining exercises now, with `gnat-gps` (or `gnatstudio`).

1. Build the complete workspace:
   `colcon build`
2. Run the `printenv_ada` script to inspect the environment that is needed to build/load projects with gnatstudio/gprbuild:
   `./printenv_ada`
   The output should be something similar to:

> export GPR_PROJECT_PATH="/home/user/prog/ada4ros2/src/my_ada_package_completed/ada_code:/home/user/prog/ada4ros2/src/rclada_client_skeleton:/home/user/prog/ada4ros2/src/rclada_common/gpr_aaa:/home/user/prog/ada4ros2/src/rclada_common/gpr_ament:/home/user/prog/ada4ros2/src/rclada_common/gpr_c_builtins:/home/user/prog/ada4ros2/src/rclada_common/gpr_cstrings:/home/user/prog/ada4ros2/src/rclada_examples:/home/user/prog/ada4ros2/src/rclada_fosdem20:/home/user/prog/ada4ros2/src/rclada/gpr_rcl:/home/user/prog/ada4ros2/src/rclada/gpr_selftest:/home/user/prog/ada4ros2/src/rclada_tf2/gpr_examples:/home/user/prog/ada4ros2/src/rclada_tf2/gpr_tf2_ros:/home/user/prog/ada4ros2/src/rosidl_generator_ada/gpr_c_typesupport:/home/user/prog/ada4ros2/src/rosidl_generator_ada/gpr_generator:/home/user/prog/ada4ros2/src/rosidl_generator_ada/gpr_rosidl:/home/user/prog/ada4ros2/src/rosidl_generator_ada/gpr_rosidl/dl-ada:/home/user/prog/ada4ros2/src/rosidl_generator_ada/gpr_rosidl/dl-ada/cstrings:/home/user/prog/ada4ros2/src/tutorial:/home/user/prog/ada4ros2/src/tutorial/common:/home/user/prog/ada4ros2/src/tutorial/exercises:/home/user/prog/ada4ros2/src/tutorial/solutions:/home/user/prog/ada4ros2/install/rclada_common/share/gpr:/home/user/prog/ada4ros2/install/rclada_examples/share/gpr:/home/user/prog/ada4ros2/install/rclada_fosdem20/share/gpr:/home/user/prog/ada4ros2/install/rclada/share/gpr:/home/user/prog/ada4ros2/install/rclada_tf2/share/gpr:/home/user/prog/ada4ros2/install/rosidl_generator_ada/share/gpr:/home/user/prog/ada4ros2/install/tutorial_common/share/gpr:/home/user/prog/ada4ros2/install/tutorial_exercises/share/gpr:/home/user/prog/ada4ros2/install/tutorial_solutions/share/gpr"

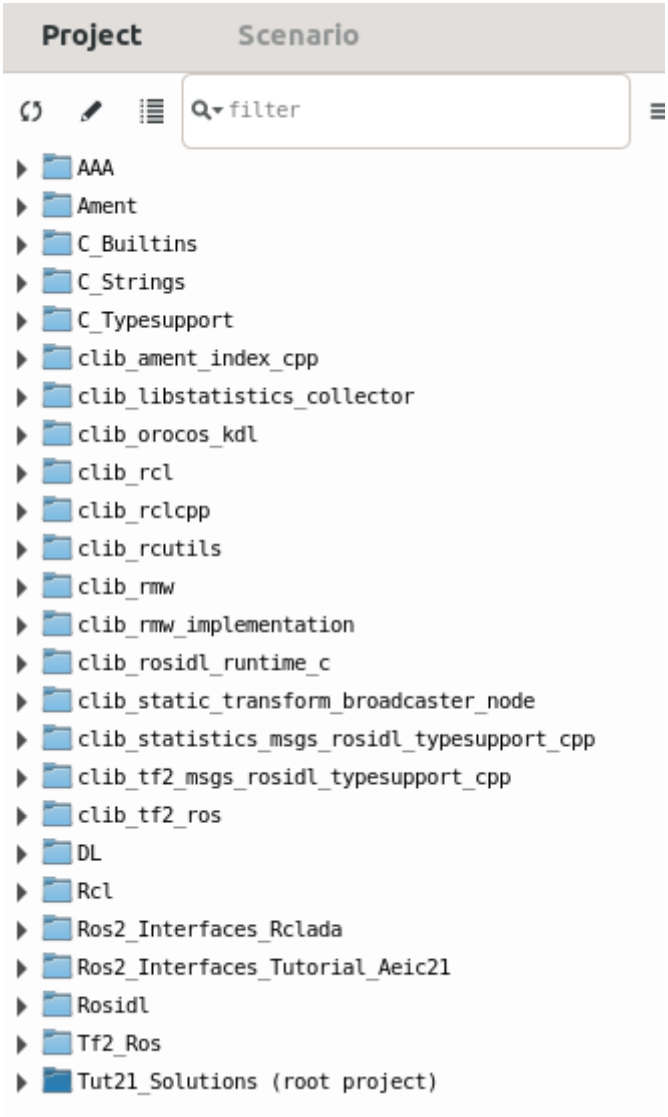
NOTE: the `printenv_ada` assumes that all packages are under `./src`.

3. Actually do load this Ada environment:
   `source <(./printenv_ada)`
4. Open the project with the solutions in GNATstudio/gnat-gps
   - With gnat-gps: `gnat-gps -P src/tutorial/solutions/tutorial_solutions.gpr`
   - With gnatstudio: `path/to/gnatstudio -P src/tutorial/solutions/tutorial_solutions.gpr`
     - NOTE: the gnat compiler in the path should be the one in `/usr/bin` and not the one in the Community Edition; otherwise compilation will fail. You can verify the version in use from within gnatstudio by going to `Help -> About...`:

       and the version should be, as shown, GNAT 9.3.0 from Ubuntu's `gnat` package.

- After successfully opening the root project, all *withed* projects and sources are accessible through the Project view:



5. Build everything with `Build -> Project -> Build all`
    - **Note**: since the compilation is now happening *in-tree*, everything will be recompiled and the final executables will be in the source tree, and not in the `build` or `install` folders. Keep this in mind to run the appropriate final binaries in case of changes.
    - The advantage of editing this way is that now `colcon` is not needed to make changes and rebuild, and `gprbuild` will indeed recompile only the required sources for any changes. This is in general much faster. Also, navigation through Ada sources with Ctrl+Click is much more robust in GNATstudio vs VSCode.
6. Verify that only minimal recompilation is happening by opening, for example, `sol_publisher_static.adb` in the root project (`Tutorial_Solutions`) and modifying some of the strings in there, and rebuilding like in the previous step.
    - Undo the changes after the test.

Conclusion

In this block we have seen, firstly, how to create a new ROS2 package and how to integrate in it a GNAT project; and finally, how edition and compilation of Ada sources in the context of ROS2 packages can be achieved, from raw `colcon` in the terminal to the practicality of a full-fledged Ada GUI such as GNATstudio.

2. Blackboard communication (topics/publishers/subscriptions)

From this point on, all exercises are ROS2 nodes written in Ada. Each exercise source code is provided twice:

1. In project `ada4ros2/src/tutorial/exercises/tutorial_exercises.gpr`
    - These files are intended to guide you via comments on the objectives of the exercise, and to have a file ready to start without requiring you to create a new package, Ada project, etc. The intention is that you edit these files directly to complete the exercise.
2. In project `ada4ros2/src/tutorial/solutions/tutorial_solutions.gpr`
    - The files in this project are prefixed with `sol_` and are a reference implementation of the exercise. You can use these to get ideas, check the intended behavior of the result, quickly try modifications on a complete solution, etc.

For convenient edition/comparison of any of these files, the project `ada4ros2/src/tutorial/tutorial.gpr` is recommended. It is an aggregate project that merely includes both `tutorial_*.gpr` projects.

> This `tutorial.gpr` project file requires a previous successful build to find all required dependencies.

In summary, from the `ada4ros2` repository root:

1. `colcon build`
2. `source <(./printenv_ada)`
3. `gnat-gps -P src/tutorial/tutorial.gpr`

Using RCLAda dynamic vs static messages

For technical reasons, RCLAda offers two ways of accessing information in a ROS2 message:

1. Through runtime introspection, with a small runtime overhead, but with full guarantees that the worst that can happen is an exception when erroneous access is attempted.
2. Through record types generated during the build, following ROS tradition. Improper use of these types may result in unguarded accesses to invalid memory, as they closely match C memory layouts. This is, however, the expected usage. Future releases of RCLAda will improve the safety aspect of this approach.

For the exercises in this block 2, and to get acquainted with both methods, it is recommended to complete either exercises 2.1+2.4 or 2.2+2.3.

- 2.1/2.2 do the same, but using the dynamic/static version.
- 2.3/2.4 do the same, but using the dynamic/static version.
- In tandem, you will be creating a publisher and a subscriber able to relay and receive information.

For each exercise you will receive this information here (and the comments in the exercise main file):

- **Objective**: *A brief description of the purpose of the exercise and learning goal.*
- **Main file**: The name of the Ada file to be completed; e.g., `cool_exercise.adb`. This means that a `sol_cool_exercise.adb` also exists with the complete implementation.
- **APIs**: A list of `project.gpr/specification.ads` files that are relevant to the completion of the exercise, and that have not appeared before.

- **Tools**: A list of command-line commands from ROS2 that can be helpful for the exercise. Running them without arguments or with `--help` to get details is recommended.
- **Notes**: any additional information.

2.1 Writing a Publisher, dynamic version

- **Objective**: *Create a publisher of type `std_msgs/String` and publish an string using it. Use only dynamic message facilities.*
- **Main file**: `publisher_dynamic.adb`
- **APIs**:
  - `rcl.gpr/rcl-logging.ads`
  - `rcl.gpr/rcl-nodes.ads`
  - `rcl.gpr/rcl-publishers.ads`
  - `rosidl.gpr/rosidl-dynamic.ads`
  - `rosidl.gpr/rosidl-typesupport.ads`
- **Tools**: `ros2 topic`
  - Verify that your node works with `ros2 topic echo <topic>`
  - Also `ros2 topic {info|type} [-v]` is useful to check a topic information.

2.2 Writing a Publisher, static version

- **Objective**: *Create a publisher of type `std_msgs/String` and publish an string using it. Use static messages already generated in the `rclada` package (`std_msgs/String`).*
- **Main file**: `publisher_static.adb`
- **APIs**:
  - `rcl.gpr/rcl-logging.ads`
  - `rcl.gpr/rcl-nodes.ads`
  - `rosidl.gpr/rosidl-types.ads`
  - `ros2_interfaces_rclada.gpr/ROSIDL.Static.Rclada.Std_Msgs.Messages.String`
- **Tools**: `ros2 topic`
  - Verify that your node works with `ros2 topic echo <topic>`
  - Also `ros2 topic {info|type} [-v]` is useful to check a topic information.

2.3 Writing a Subscriber, dynamic version

- **Objective**: *Create a subscriber of type `std_msgs/String` and read and echo incoming strings. Use only dynamic message facilities.*
- **Main file**: `subscriber_dynamic.adb`
- **APIs**:
  - `rcl.gpr/rcl-logging.ads`
  - `rcl.gpr/rcl-nodes.ads`
  - `rosidl.gpr/rosidl-dynamic.ads`
  - `rosidl.gpr/rosidl-typesupport.ads`
- **Tools**: `ros2 topic`
  - Verify that your node works with `ros2 topic pub /chatter std_msgs/msg/String 'data: "hello"'`
    - See how TAB autocompletion works for every part of `ros2 topic`.

- Verify that your subscriber can also hear your publisher from tasks 2.1/2.2.

2.4 Writing a Subscriber, static version

- **Objective**: *Create a subscriber of type* `std_msgs/String` *and read and echo incoming strings. Use static messages already generated in the* `rclada` *package (*`std_msgs/String`*).*
- **Main file**: `subscriber_static.adb`
- **APIs**:
  - `rcl.gpr/rcl-logging.ads`
  - `rcl.gpr/rcl-nodes.ads`
  - `rosidl.gpr/rosidl-types.ads`
  - `ros2_interfaces_rclada.gpr/ROSIDL.Static.Rclada.Std_Msgs.Messages.String`
- **Tools**: `ros2 topic`
  - Verify that your node works with `ros2 topic pub /chatter std_msgs/msg/String 'data: "hello"'`
    - See how TAB autocompletion works for every part of `ros2 topic`.
  - Verify that your subscriber can also hear your publisher from tasks 2.1/2.2.

TurtleSim: first steps

Using the *TurtleSim* simulator it is possible to practice control notions of actual robots. From this point on, all solutions will use static messages.

2.5 Launching a *TurtleSim* and identifying relevant information

In this exercise you will get to know this basic simulator in preparation for the following exercises.

1. Launch an instance of the simulator with `ros2 run turtlesim turtlesim_node`
2. Inspect the topics that it creates for communication `ros2 topic list`
3. Identify basic information of the topics `/turtle1/cmd_vel` and `/turtle1/pose` using `ros2 topic info`
   - You will find that these topics use messages of type `geometry_msgs/msg/Twist` and `turtlesim/msg/Pose`, respectively.
4. Display the composition of the `turtlesim/msg/Pose` message with `ros2 interface show`.
   - You should get this output:

     ```
     float32 x
     float32 y
     float32 theta

     float32 linear_velocity
     float32 angular_velocity
     ```

   - Compare this information with the contents of the files:
     - `/opt/ros/foxy/share/turtlesim/msg/Pose.msg`
     - `/opt/ros/foxy/include/turtlesim/msg/detail/pose__struct.h`

        - `./build/tutorial_solutions/rosidl_generator_ada/rosidl-static-tutorial_solutions-turtlesim-messages-pose.ads`
5. Move the turtle manually from the command line with `ros2 topic pub` (remember the TAB completion!):

```
$ ros2 topic pub --times 1 /turtle1/cmd_vel geometry_msgs/msg/Twist
"linear:
  x: 1.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0"
```

2.6 Move the turtle in a fixed sequence

- **Objective**: *Draw a figure or text using a predefined sequence of instructions.*
- **Main file**: `turtlesim_mosaic.adb`, `turtlesim_hello.adb`
- **APIs**:
    - `ros2_interfaces_tutorial_exercises.gpr/rosidl-static-tutorial_exercises-geometry_msgs-messages-twist.ads`
    - **Note**: this project file and specification will not exist until you modify the `tutorial_exercises/CMakeLists.txt` file to import the interfaces from the `geometry_msgs` package.`
        - To import the interfaces you need to use the `ada_import_interfaces` CMake function.
- **Tools**: to conveniently rebuild only the `tutorial_exercises` ROS2 package from scratch you may use the `dev/make-package.sh <package>` script.
- **Notes**: Since you need to import new interfaces for the first time, the steps to follow are:
    1. Modify the `src/tutorial_exercises/CMakeLists.txt` file to import the required interfaces.
    2. Build the package so the interfaces are generated: `colcon build --packages-select tutorial_exercises`
        - **Verify** that the project file for the interfaces exist at the expected location: `./install/tutorial_exercises/share/gpr/ros2_interfaces_tutorial_exercises.gpr`
    3. Edit the Ada node normally.

2.7 Create a remote controller for the turtle

- **Objective**: *Create a node that allows to drive the turtle with the keyboard*.
- **Main file**: `turtlesim_commander.adb`
- **APIs**:
    - `procedure Ada.Text_IO.Get_Immediate (Item : out Character; Available : out Boolean)`
    - `ros2_interfaces_tutorial_exercises.gpr/rosidl-static-tutorial_exercises-geometry_msgs-messages-twist.ads`

        - **Note**: this project file and specification will not exist until you modify the `src/tutorial/exercises/CMakeLists.txt` file to import the interfaces from the `geometry_msgs` package.
        - To import the interfaces you need to use the `ada_import_interfaces` CMake function.
- **Tools**: to conveniently rebuild only the `tutorial_exercises` ROS2 package from scratch you may use the `dev/make-package.sh <package>` script.

2.8 Drive the *ePuck* robot

The ePuck robot is a real educational robot that can be simulated with good fidelity with the Webots simulator. You can launch an instance of the simulator with an ePuck ready to be commanded with:

```
ros2 launch webots_ros2_epuck robot_launch.py
```

> The `launch` subcommand orchestrates the launch of possibly several nodes and sets up tunable parameters. This is out of the scope of the tutorial, and it suffices to know at this time that is a kind of "uber"-run.

Use the tools learned in exercise 2.5 to identify the available topics and their type.

> You will find that the ePuck, despite being real, unlike the turtle, is driven by the same `/cmd_vel` topic, using the same message type. Observe that the only difference is that TurtleSim prefixed the topic as `/turtle1/cmd_vel`, as several turtles may be created. In multi-robot, this kind of topic renaming is usual and ROS2 has ways to rename topics for this purpose.

- **Objective**: *Create a node that allows to drive the ePuck with the keyboard.*
- **Main file**: `epuck_commander.adb`
- **APIs**:
    - `procedure Ada.Text_IO.Get_Immediate (Item : out Character; Available : out Boolean)`
    - `ros2_interfaces_tutorial_exercises.gpr/rosidl-static-tutorial_exercises-geometry_msgs-messages-twist.ads`
        - **Note**: this project file and specification will not exist until you modify the `src/tutorial/exercises/CMakeLists.txt` file to import the interfaces from the `geometry_msgs` package.
        - To import the interfaces you need to use the `ada_import_interfaces` CMake function.
- **Tools**: to conveniently rebuild only the `tutorial_exercises` ROS2 package from scratch you may use the `dev/make-package.sh <package>` script.

Conclusion

In this section we have seen how most communications take place in the ROS2 framework, and how the information can be published and retrieved from Ada nodes. With this information, it is easy to program our first robots, which share the ROS2 convention of using the `/cmd_vel` topic to listen to commands.

3. RPC communication (services/servers/clients)

Although topics are used for the bulk of sensor data, services are also used for low-frequency communications that require acknowledgement. In this section we will see the basic usage of ROS2 services.

3.1 Servers

- **Objective**: *Create a server that waits to be triggered (`std_srvs/srv/Trigger`).*
- **Main file**: `server_example.adb`
- **APIs**:
  - `rcl.gpr/RCL.Nodes.Typed_Serve`
  - `ros2_interfaces_rclada.gpr/ROSIDL.Static.Rclada.Std_Srvs.Services.Trigger`
    - **Note**: this project file and specification already exists as it is imported by the `rclada` package.
- **Tools**: `ros2 service`
  - Inspect available services with `ros2 service list`.
  - Identify the type of a service with `ros2 service type <service>`
  - Identify the types of requests/responses of a service with `ros2 interface show <service>`
  - Test your service with `$ ros2 service call /ada_service std_srvs/srv/Trigger {}`
    - The empty JSON object, `{}`, can be omitted because the request is an empty message.

3.2 Asynchronous clients

ROS2 provides only an asynchronous facility for RPC calls which client libraries can leverage to create higher-level abstractions. With RCLAda, you can use both asynchronous and a synchronous clients.

- **Objective**: *Create an asynchronous client for the server defined in 3.1 (`std_srvs/srv/Trigger`).*
- **Main file**: `client_async.adb`
- **APIs**:
  - `rcl.gpr/RCL.Nodes.Typed_Client_Call_Proc`
    - This is a generic that must be instantiated with the appropriate type and callback.
  - `ros2_interfaces_rclada.gpr/ROSIDL.Static.Rclada.Std_Srvs.Services.Trigger`
    - **Note**: this project file and specification already exists as it is imported by the `rclada` package.
- **Notes**: although the reply will be received asynchronously, the call allows two timeouts, for the service to become available and for the call to get through.
  - Experiment with these timeouts.
    - Using no timeouts may result in calls being lost because topic discovery takes a few seconds.
  - Experiment with launching the server before/after the client.

3.3 Synchronous clients

- **Objective**: *Create a synchronous client for the server defined in 3.1 (`std_srvs/srv/Trigger`).*
- **Main file**: `client_sync.adb`
- **APIs**:
  - `rcl.gpr/RCL.Nodes.Typed_Client_Call_Func`
    - This is a generic that must be instantiated with the appropriate service type.
  - `ros2_interfaces_rclada.gpr/ROSIDL.Static.Rclada.Std_Srvs.Services.Trigger`
    - **Note**: this project file and specification already exists as it is imported by the `rclada` package.
- **Notes**: as the response is received synchronously, there is no need to spin the node in this case.

3.4 Control the TurtleSim with service calls

The *TurtleSim* allows spawning more turtles and changing the pen color via service calls. Modify your `turtlesim_commander` to set different pen colors on demand.

Conclusion

In this section we have seen how RPC-like communication takes place in ROS2 and with RCLAda. These calls are usually used for important communications that should not be lost: changing configuration parameters, starting robot behaviors, etc.

4. Realistic exercises

In the last block of exercises we will see a couple of simple robotic behaviors: a Roomba-like wander (without wall following), and local navigation with obstacle avoidance.

Wandering robot with obstacle avoidance (Roomba-like)

One of the first behaviors that are tested in a robot is *wander*, which consists in moving randomly but avoiding frontal collisions. Depending on the sensor suite of the robot this can be achieved with more or less accuracy.

4.1 Wandering ePuck

The ePuck has two kinds of sensors that can help in this task: an array of sonar emitters/receivers, and a time-of-flight (ToF) range finder. The schematics can be consulted in this page: https://github.com/cyberbotics/webots_ros2/wiki/Tutorial-E-puck-for-ROS2-Beginners.

Although the sonar may seem more complete, given the small size of the ePuck the ToF sensor is enough and simpler to get started, so it is recommended for this exercise.

- **Objective**: *Create a node that sends the ePuck in a wandering errand. Suggestions:*
  - *Move in a straight line until an obstacle is detected. Then, stop and rotate a random amount. Repeat.*
    - *Instead of always moving until obstacle detection, move up to a maximum time and then rotate.*
  - *Further suggestions:*
    - *Light the frontal leds when obstacles are detected*
    - *Light the leds on the side to which the ePuck is rotating.*

- **Main file**: `epuck_wander.adb`
- **APIs**:
  - `Ada.Numerics.Float_Random`
  - `ros2_interfaces_tutorial_exercises.gpr/ROSIDL.Static.Tutorial_Exercises.Geometry_Msgs.Messages.Twist`
  - `ros2_interfaces_tutorial_exercises.gpr/ROSIDL.Static.Tutorial_Exercises.Sensor_Msgs.Messages.Range_Data`

4.2 Wandering TurtleBot

The TurtleBot (not to be confused with the TurtleSim) is an educational robot equipped with a 360º laser range finder. Also, it is a bit bigger than the ePuck, so relying only on a single straight reading will result in many collisions.

NOTE: the laser readings are published with "Best Effort" reliability. Your listener must be configured with the same Quality of Service or no readings will be received. (You can check these details with `ros2 topic info -v /scan`.) See related APIs below.

- **Objective**: *Create a node that sends the TurtleBot in a wandering errand. Use a range or readings in front of the robot to make navigation safer.*
- **Main file**: `turtlebot_wander.adb`
- **APIs**:
  - `rcl.gpr/RCL.Subscriptions`
    - See the `Defaults` object to configure the Quality of Service.
  - `rcl.gpr/RCL.QoS`
    - See the `Profiles.Sensor_Data` profile that you will need to configure the scan subscriber.
  - `Ada.Numerics.Float_Random`
  - `ros2_interfaces_tutorial_exercises.gpr/ROSIDL.Static.Tutorial_Exercises.Geometry_Msgs.Messages.Twist`
  - `ros2_interfaces_tutorial_exercises.gpr/ROSIDL.Static.Tutorial_Exercises.Sensor_Msgs.Messages.Range_Data`
- **Notes**:
  - Launch the TurtleBot simulation with `ros2 launch webots_ros2_turtlebot robot_launch.py`

4.3 Navigate to a local goal (VFH local planner)

Local obstacle avoidance is a must in any robotic navigation stack. Depending on your previous experience, you might try to come up with your own local avoidance algorithm, or reuse the supplied Vector Field Histogram (VFH) algorithm, already tuned for the TurtleBot.

- **Objective**: *Create a node that waits for a goal (in local coordinates). When received, steer the TurtleBot to it while avoiding collisions.*
- **Main file**: `turtlebot_VFH.adb`
- **APIs**:
  - `tutorial_common.gpr/VFH.Steer`
    - Use the version that directly receives a raw LaserScan reading.

    - `rcl.gpr/RCL.QoS`
        - See the `Profiles.Sensor_Data` profile that you will need to configure the scan subscriber.
    - `ros2_interfaces_tutorial_common.gpr/ROSIDL.Static.Tutorial_Common.Sensor_Msgs.Messages.Laserscan`
        - This message is already generated in the `tutorial_common` package, because it has to be used to call the VFH algorithm.
    - More messages for the goal, odometry, velocity commands, that you must identify with `ros2 topic info`.
- **Notes**:
    - Launch the TurtleBot simulation with `ros2 launch webots_ros2_turtlebot robot_launch.py`
    - The topics of interest in this exercise are `/cmd_vel`, `/goal_pose`, `/odom`, `/scan`.
    - There is a `rviz2` configuration tailored for this exercise in `ada4ros2/src/tutorial/turtlebot.rviz`
        - Use `rviz2` to easily send goals to the robot in its local coordinate frame.

4.4 Working with reference frame transformations

Using laser readings directly relies on hidden information like the placement of the laser on the robot. Robust robotics do not rely on such assumptions, but utilize reference frame transformations to work with information in the appropriate reference frame. In this exercise we can take advantage of the `tf2` package to transform laser readings into their actual positions in robot coordinates. RCLAda offers at the time a bare-bones binding to the `tf2` and `tf2_ros` packages that enables using these transformations.

- **Objective**: *Create a node that waits for a goal (in local coordinates). When received, steer the TurtleBot to it while avoiding collisions using the VFH algorithm in `tutorial_common`. To improve navigation, transform the laser scan into a cloud of points using TF2 before calling the VFH.Steer function.*
- **Main file**: `turtlebot_VFH_TF2.adb`
- **APIs**:
    - `tf2_ros.gpr/RCL.TF2`
    - `tutorial_common.gpr/VFH.Steer`
        - Use the version that receives a cloud of points in the local frame of the robot.
    - `rcl.gpr/RCL.QoS`
        - See the `Profiles.Sensor_Data` profile that you will need to configure the scan subscriber.
    - `ros2_interfaces_tutorial_common.gpr/ROSIDL.Static.Tutorial_Common.Sensor_Msgs.Messages.Laserscan`
        - This message is already generated in the `tutorial_common` package, because it has to be used to call the VFH algorithm.
    - More messages for the goal, odometry, velocity commands, that you must identify with `ros2 topic info`.
- **Tools**: The package `tf2_ros` contains utilities to monitor existing transforms:
    - `ros2 run tf2_ros tf2_monitor`

- **Notes**:
    - A good primer on coordinate frames can be found at https://blog.hadabot.com/ros2-navigation-tf2-tutorial-using-turtlesim.html

Conclusion

In this part we have put together our RCLAda API knowledge and a few basic robotics notions to program simple behaviors in models of real education robots.