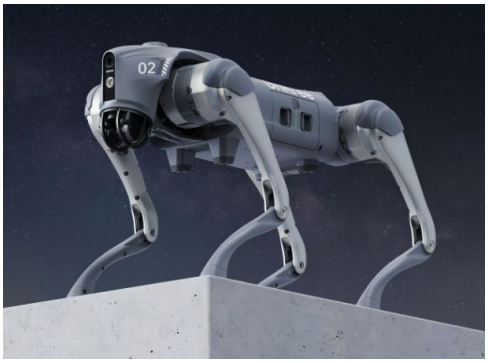
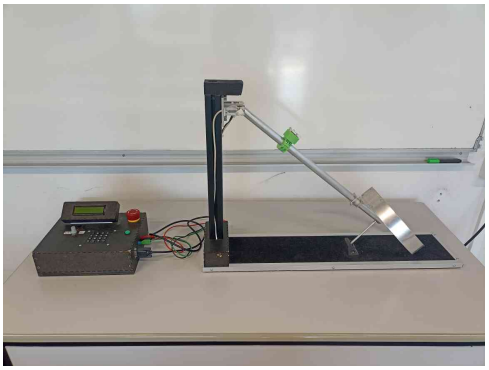
Polytech Nice Sophia
Robotics and Autonomous Systems — 5th year

UNITREE GO2 & FLYING ARM PROJECTS

Ronan Le Corronc & Maximilien Kulbicki

Supervisor:
Frédéric Juan





1 Introduction

During our third year, we focused on the design of a quadruped robot. The aim was to be able, from scratch, to build and use a quadruped robot to evolve in a stable environment such as a building. The aim of this project was to give us an initial insight into all areas closely and remotely related to robotics. Unfortunately, due to lack of time, we were only able to concentrate on the physical part of the robot. We were able to develop all the robot's mechanical systems and study its resistance to shock and weight. We were also able to carry out the electrical installation and power supply of each primary component, such as the sensors and actuators (brushless motors). Then, to round off the year, we were able to carry out various joint servo-control tests, but without much success. This project was therefore a great success, but only allowed us to focus on the fundamental basis of the robot.

When it came to choosing a new project, we wanted to explore a new area, and the idea of continuing with our quadruped robot was compromised by its challenging mechanical design: Its legs couldn't support its weight. If we'd continued with it, we'd have had to start from scratch with its mechanical design, rather than focusing on control algorithms. So the idea was to order a commercially available quadruped robot and focus solely on its programming and control. It wasn't until the fifth year of the project that we had confirmation that a quadruped robot would be entrusted to us. The Unitree Go2 project was thus born.

At the same time, during our fourth year, we worked on the Flying Arm project, a simple working model. The aim of this project was to give us a first approach to low-level programming on STM32 microcontrollers, and also to put into practice servo-controls using controllers such as PID, phase advance and so on. We were able to continue this project in seventh grade during the Robotics and Sensor Fusion 2 courses, to study the implementation of the Kalman filter. As a result of all our work on this project, we decided to continue it in our fifth-year project, so as to finalize the code and have a fully usable, finished model.

We therefore worked in parallel on these two projects. Projects that we will detail in this report in two main parts: the Flying Arm and the Unitree Go2 project.

Contents

2 Flying Arm

2.1 Introduction

The Flying Arm project is a model designed to be used by students to put into practice the various controllers and data filters seen in class. It could be used as a robotics project or as a lab model. The model is simple. It's an oscillating arm mounted on a frame. The arm has an actuator, a propeller mounted on a brushless motor at its end. The model has three sensors. The first sensor is a potentiometer, mounted on the arm's axis of rotation. It returns the arm's angle of rotation via an analog signal. The second is a gyroscope mounted on the arm. It provides the arm's angular velocity via UART communication. The last is a sensor measuring the rotation speed of the brushless motor. However, due to signal disturbances, it is not used, as it would require analog filtering.

The model also has a control box. This interface box includes an LCD screen, a matrix keyboard, a potentiometer, LEDs, a push-button and an emergency stop button. The latter is connected to the arm via a DB9 connector, and the ESC's three-phase power is transmitted via three banana cables. An STM32F407VGT6 microcontroller soldered onto a discovery-kit processes and coordinates all project components.

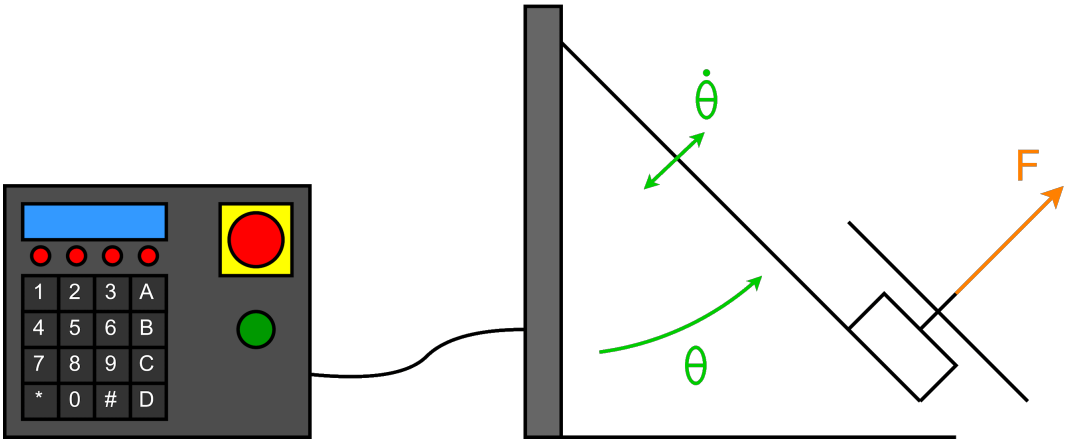


Figure 1: Flying Arm schematic

During our fourth year, we did several things on the model.

The first thing was to refactor the code. In fact, the model already had several functional programs, but unfortunately these were codes using outdated and no longer used libraries. This prevented the use of new libraries and current functionalities. So we started from scratch and reprogrammed the entire model using standard STMCubeIDE libraries.

The second thing we did was to design different controllers on the model and set up different data filters. We were able to implement, on independent codes, a PID controller, a phase advance controller and a cascade controller. Then, mainly two data filters. A classical averaging process and an extended Kalman filter. These processes work well today, but are found on draft projects and on different projects. They can't all work at the same time and require re-uploading of new code.

The objective here was to take all the work done and bring it together in a single code. A single code that allows each controller and data filter to be used as required, without the need for computer intervention. All this while allowing them to be configured by adjusting their various parameters. Our first task was to restructure the entire code to make it much simpler and easier to understand. And secondly, we wanted to make the interface between human and machine much more usable, clearer and easier to understand.

2.2 Structure

2.2.1 Menu

The aim here is to use an adjustable system with different parameters. parameters, it's essential to have a user-friendly interface to be able to configure everything quickly. This interface will be provided through the box. A key element of this interface is an LCD display with 2x20 characters. This screen is key because it displays information, such as the controller to be used or sensor measurements in real time.

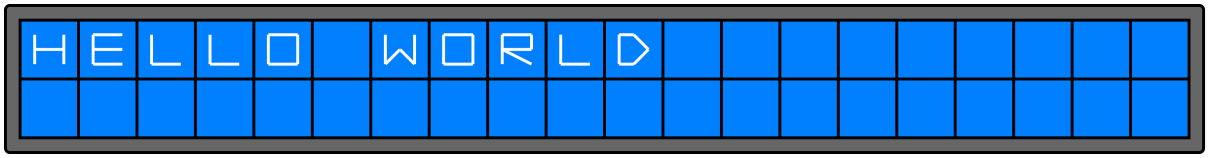


Figure 2: LCD screen schematic

The idea was to use this screen to display not just information information, but an entire menu in which we can select in which we can select options and move between different sub-menu. This menu would be usable via the numeric keypad. Certain buttons, defined in the code, allow you to move around, back or select different items. We'll come back to this later.

A first step was to design the menu. To make it as as simple as possible, while still providing all the necessary settings. To achieve this, we decided to set up the menu as follows.

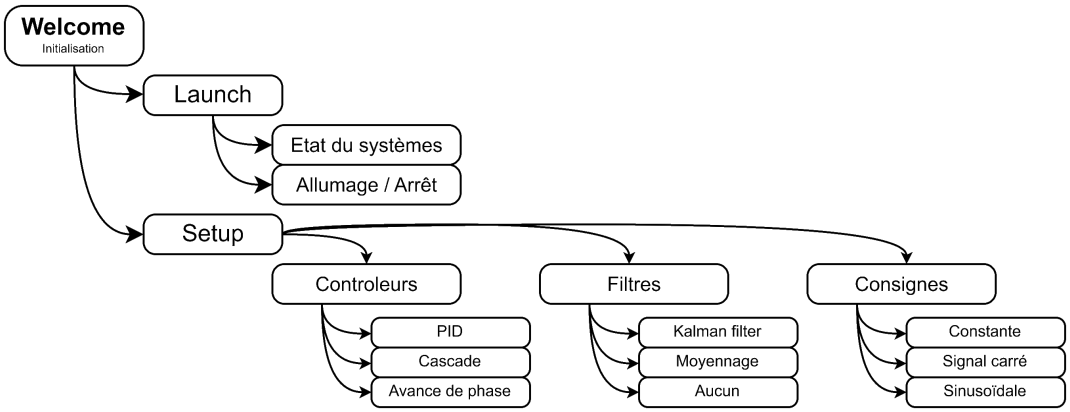


Figure 3: Menu structure

When the system is started up, it goes through an initialization phase. It initializes all the elements required for proper operation, from the central clock to the keyboard interrupt and gyroscope calibration. This first phase takes just a few milliseconds, but is essential. For this purpose, the `Welcome` menu has been set up to enable the user to visualize the various stages of initialization and thus detect any faults.

Then, if everything has initialized correctly, the system is ready for operation. At this point, the menu displays two sub-menus. The `Launch` menu and the `Setup` menu.

The "Launch" menu is the one and only menu for which the model can enter into operation. This menu requires prior configuration of the system, where default items in the code will be selected. In it, several items of information are indicated.

The first is the system status, indicated in the top right-hand corner, with `STOP` or `RUN` indicating whether the motor is armed or not (i.e. `STOP`: the motor is disarmed, `RUN`: the motor is armed). For safety reasons, this information is essential.

The second piece of information, in the top left-hand corner, is the desired setpoint. This is represented by an angle in degrees, and can be fixed or variable according to system settings. This information informs you of the system's objective when in operation.

To match this setting, the system's current angle is shown in the bottom left-hand corner. This angle is displayed in real time and is updated regularly. Note that the value displayed is not the angle measured by the potentiometer, but the angle used by the system to locate it. In other words, the angle displayed is the result obtained after using the selected filter, be it averaging or the Extended Kalman filter.

As the system tries to stick to the setpoint, it's important to know what it's doing. To this end, the bottom right-hand corner shows the percentage of power transmitted to the brushless motor. This information is necessary when the system is running, but not when it's stopped. In this case, we indicate the controller used.

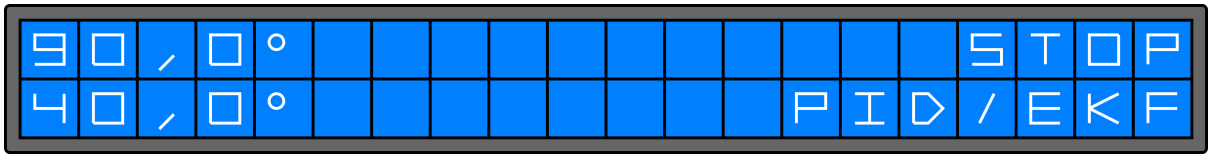


Figure 4: Menu launch Stop example

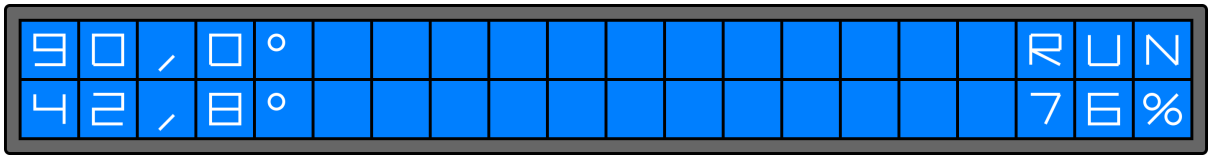


Figure 5: Menu launch Run example

The second menu available after initialization is the "Setup" menu. This menu is very comprehensive, but in its entirety, it enables you to set up and configure the entire system for operation. This "setup" menu is made up of 3 sub-menus: "Controllers", "Filters" and "Setpoints". For each, you can select the desired item and configure its parameters. Respectively, we can see the sub-menu as follows:

- `Controllers`: PID, Cascade, Phase advance
- `Filters`: Averaging, EKF, none
- `Setpoints`: Constant, sinusoidal signal, square-wave signal

Note that other controllers and filters can be set up, but require system reprogramming.

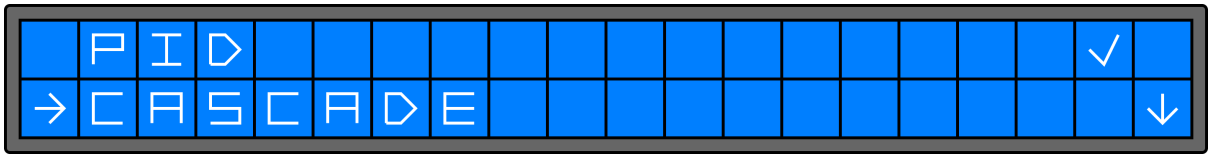


Figure 6: Menu setup controllers example

The submenus (see figure 6) are composed as follows. First, an arrow on the left indicates the position of our cursor for selection or adjustment. Next to this is the name of the item we wish to use, in this case for the example of the controllers menu, the PID controller and the Cascade. Then, on the right, a check indicating which element is currently being used by the system, in this case the PID controller. And finally, on the far left, a down or up arrow indicates whether there are other items available further down the menu.

2.2.2 Code

Before starting to write the code, as with the menu, it's very important to define our algorithm and the structure of our code. This part is very important and complicated to set up due to the complexity of the code. The system, using an STM32 microcontroller, is programmed in C using the STM32CubeIDE software. Once completed, it is compiled and transmitted to the microcontroller via UART communication. As this is low-level programming, it's important to configure every component of the system, from the sensors to the clock. This part has already been programmed, but due to the re-structuring, it was important to take this into account so as not to lose the work already done.

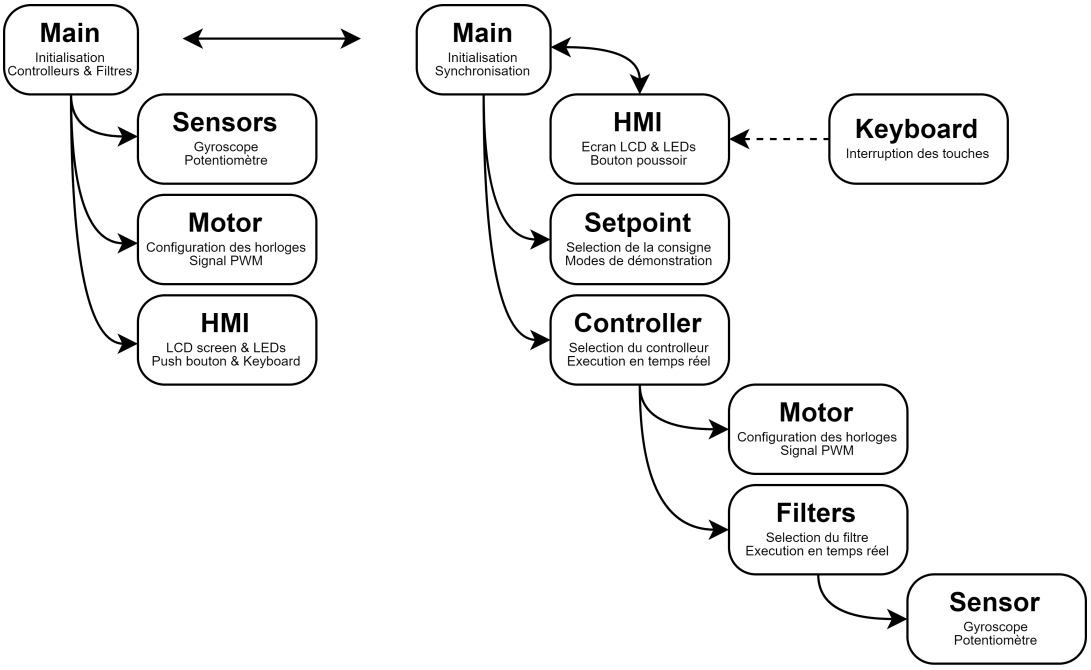


Figure 7: Enter Caption

The old code was very sober. It consisted mainly of 4 files, 3 of which were sub-files. The idea was to create a first low-level layer to be able to implement different algorithms and controllers simply. To achieve this, we structured the project as follows:

- `Sensors.c`

  The purpose of this file was to take care of everything to do with sensors. It took care of initializing the sensors, calibrating them and calculating potential offsets. Following this essential first part, we could use several functions to access sensor values. To simplify programming, we implemented data filters, notably an averaging filter, directly in this code. So all we had to do was call a function to obtain the large physical value, such as the angle, already filtered and reliable.

- `Motors.c`

  In the same way as sensor, this file took care of everything to do with the motor. As the motor uses signals operating on very precise frequencies and delays, we decided to add the function for setting the microcontroller's clocks to this file. We could then initialize the PWM signal, useful for controlling motor speed. This file therefore had various functions for changing the value of the PWM signal. We could directly give a register value, a pulse time value, or a percentage representing the motor's minimum and maximum speed.

- `HMI.c`

  This file was used for communication with the user. It contained the vast majority of the control unit's components, such as functions for the LCD screen and LEDs. But also the functions that

indicate which keys have been pressed on the keyboard, or whether the launch button has been pressed.

- `Main.c`

  This file was basically blank, allowing us to model our systems as we wished. Thanks to the 3 previous files, we could call up simple functions to directly obtain an angle or to set a precise speed for the motor. Real-time controllers were configured in this file.

This code model was therefore very efficient, but very limited in its use. As explained in the introduction, we needed a much more flexible and configurable model. Keeping the same idea of operation, we therefore re-structured the code, keeping this low-level layer but adding a higher-level layer with main.

- `HMI.c`

  This file has not changed its position, the point being to be able to communicate directly with the user from main. However, the big difference is that the entire menu is coded in this file. It therefore has the different menu types explained above and their display. The code still includes low-level functions such as LCD display and LED lighting. However, a new file has been created, the `Keyboard` file. This file contains all the functions required to operate the keyboard. It has been separated from the HMI file as it now has an interrupt function. When a key is pressed, the main switches off the program for a very short time, allowing the HMI file to retrieve the key pressed and use it later when updating.

- `Setpoint.c`

  This file didn't exist before. The setpoint was previously configured in main and was a fixed value. This file allows you to do several things. In particular, it can be used to create a setpoint as a function of time. The idea was to keep things as simple as possible, with a `GetSetpoint()` function returning the desired setpoint value. This function fetches the setpoint from the `ConstantSetpoint`, `SinusSetpoint` or `SquareSetpoint` function. To find out whether the function should return this or that setpoint, a function called up via the HMI allows you to select the desired setpoint. So, from the main's point of view, all you have to do is ask for the setpoint, or indicate which type of setpoint you wish to use.

- `Controller.c`

  The `controller` file is not very different from the `setpoint` file. In the same way, it has a function for choosing which controller to use. There's also a function to enable or disable it. The difference is that here, we don't return controller values. All actions are performed directly in this file. The file has direct access to the engine file, similar to that explained above. This means that the controller can take the action directly, without having to go back to main. In the same way, the controller needs sensor data to function. Here, it accesses the `Filters` file directly, without going back to the main unit. It is therefore quite autonomous. When it receives a command to activate a given controller, it calculates the new action to be performed, based on the setpoint and sensor data, and then executes it.

- `Filters.c`

  This file, very low down in the tree structure, directly manages sensor data. As with the `setpoint` or `controller` file, it has a function for obtaining data from specific sensors. However, this data is filtered by the selected filter. Isolating it at such a low level therefore makes it possible to obtain an autonomous code that generates reliable data that can be used by the controller on demand.

2.3 Conclusion

Restructuring the code in this way was a very good idea, as the way it works is clear and sober. And it makes it very easy to add or remove new filters or controllers.

However, it does create a few problems. The communication between the different codes is complicated to set up, and this regularly creates dependency problems between the different files. The project is therefore more complicated to configure on STM32CubeIDE and therefore less usable.

To date, the final code has progressed very well, but is unfortunately not yet finished. We've managed to re-divide all our code into sub-programs. This was the biggest step, as we had to be very careful to put each function and declaration in the right place so as not to pollute the program and make it unusable. We also succeeded in designing a menu, with the various pages displaying correctly and reacting in real time.

However, we can't navigate through the menu, which is only functional if you specify which page to display in the code. This problem is due to a keyboard interrupt that we haven't had time to fix. During the last tests, the keyboard returned inconsistent values for the key pressed. The problem is that some are good, some are not, and we haven't been able to find any consistency.

One of our biggest problems was also synchronizing the display and arm control. As both operations had to be carried out at the same time and in different files, it was very difficult to structure the code. As the microcontroller used was single-core, it would have been necessary to set up processes or even an RTOS system for more reliable operation. We were therefore unable to test the operation of the model with the menu. I suppose there are still a number of bugs to be corrected in the calls to the various functions.

So the project remains unfinished. We're not far from the goal, but it would still take several hours of debugging to analyze each problem and fix them. The great structure of the code is there, however, and we hope that future students will be brave enough to finalize this model.

3 Unitree Go2 Edu

3.1 Introduction

The Unitree Go2 Edu is a versatile and agile quadruped robot manufactured by Unitree. It is deployable using the native Unitree application or can be controlled through the onboard Nvidia Jetson Orin. It embeds a wide range of sensors, communication modules, actuation modules can be added, and a 33.6V battery sets its usage from 2 to 4 hours.
Some of the components include :

- Sensors : 360°x90° Lidar, wide-angle camera, RGB-D camera, IMU, wireless positioning module, pressure sensors, microphone.
- Communication features : WiFi6 dual-band, Bluetooth 5.2, 4G module.
- Miscellaneous : Charging station, remote controller, speakers, front lamp, communication sockets (see figure 8)...

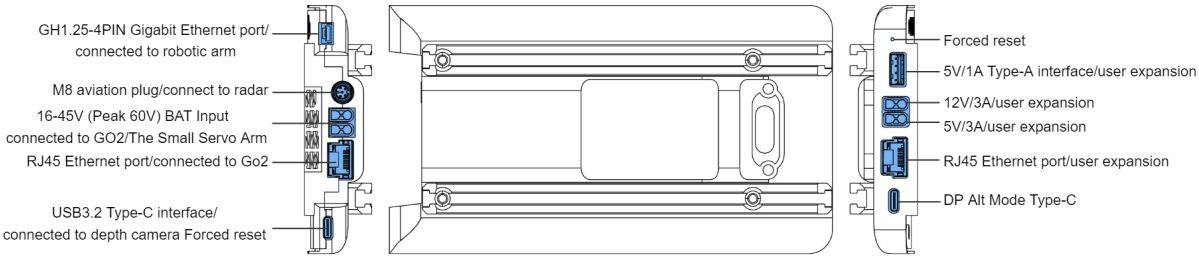


Figure 8: Communication sockets.

It is a highly programmable robot, running under Ros2, with application fields such as embedded artificial intelligence deployment, image tracking, control theory, sensor fusion or 3D path planning. Therefore, it appears as a brilliant vessel for the application of the courses followed during our robotics years.
Knowing that we had worked on a similar project in the past with an emphasis on mechanical design, it seemed interesting to develop the software aspect which presents a more extensive field of application.

3.2 Objectives

Given all the previously mentioned features, the Unitree Go2 Edu can be used for a wide range of robotic applications. Therefore, our initial goal was to develop and apply software structures to reinvest the knowledge acquired in control theory, S.L.A.M and navigation. Hence, these notions cover high-level robotic usages.
Before being able to export any sort of algorithm to the physical robot, we want to debug and test our programs in a simulated environment to prevent hardware damage or unexpected/dangerous behaviour. This environment must have a physical engine to properly simulate the robot's dynamics. Moreover, the simulator we need to use must have an interface with the Ros2 domain on which our applications will be executing.

Unitree proposes a simulation package for developers that runs on Mujoco. This package includes a representation of the robot's model, an interface to communicate between the simulation engine and the Ros2 domain, and a graphical window that renders the robot in its environment.
However, very few features (actuators and IMU) of the robot are included, which limits the development range for robotic purposes. Therefore, we will focus our work on the implementation of the most common sensor used in computer vision : the RGB-D camera. This camera plugin will be used for S.L.A.M purposes, such as the creation of depth-image, point clouds or mapping algorithm when coupled to the IMU.
The aim here is to use this camera to rebuild a 3D map of the robot's environment. To do so, we will have to edit the robot's physical model, acquire the simulation data and publish it to the Ros2 domain. But before doing so, we must configure our working environment to be able to exploit Unitree's development packages.

3.3 Workspace

The development of applications destined for Unitree robots requires specific packages depending on the target use of the robot. In our case, we seek to control the robot in a virtual environment and retrieve data from this same space. To achieve this, we use the simulator `Unitree_mujoco` built by Unitree and based on Google's Mujoco. However, before being able to exploit this simulator, some other tools are required. These packages can be found on Unitree's official Github site (see here) and each one comes with a setting up example.

- `unitree_ros2`

  This first package is necessary in the case of the development of applications that run on Ros2's domain. It establishes a communication mechanism based on CycloneDDs to ensure the exchange of control messages destined to actuate the target robot.
  CycloneDDs is one of the middlewares on which Ros2 can run. It acts as a protocol for data serialization and transportation. It is the default middleware used in the Unitree robots, and therefore needs to be installed by the user in order to develop user applications.
  Basically, this package defines the various topics used by Unitree's robots to communicate in a Ros2 domain. Once the package installed and CycloneDDs compiled, the message definitions can be found in `unitree_ros2/cyclone_ws/src/unitree/unitree_go/msg`. This folder lists at least 25 messages whose format can be observed for use.
  This package is usable in both virtual and real environment for the Unitree robots.

- `unitree_sdk2`

  This next package is necessary in the case of the development of user applications for real and virtual Unitree robots. It consists in the main API proposed by Unitree with high and low-level robot commands. It also contains libraries, compiling and debugging tools. A set of high-level command programs is also provided, showing the main architecture of a user application. An explication of the key concepts of the API can be found here.

- `unitree_mujoco`

  This last package is necessary in order to develop user application in a virtual environment. It integrates a template for robot simulation in the Mujoco simulator and many control programs are also provided. The core of the simulation is written in the `unitree_mujoco/simulate/main.cc` and includes Mujoco's physics engine, OpenGL's Glfw library (for graphical rendering) and Unitree's bridge for the Ros2 domain.
  This package also provides the robot's physical model (`unitree_mujoco/unitree_robots/go2/go2.xml`) and a scene description for the virtual environment (`unitree_mujoco/unitree_robots/go2/scene.xml`). The scene can easily be edited using the `unitree_mujoco/terrain_tool/terrain_generator.py` file.
  Unfortunately, this package only supports low-level development with a limited number of messages (`rt/LowCmd`, `rt/LowState`, `rt/SportModeState`).

For our project, we use the C++ simulator from `unitree_mujoco`. Hence, the `unitree_sdk2` package is installed at the`/opt` of the RTFS and compiled. In addition, the Mujoco application is necessary for our use case. The `unitree_ros2` package can be installed but is not used in our case since our application is limited to the virtual environment and the edition of Unitree's bridge (between simulation and Ros2 domain).
Finally, once all the packages are properly installed and compiled, the execution of the simulation should be possible and the topic exchange can be reviewed by sourcing a local setup file. This is aciheved using the command : `source ../unitree_ros2/setup_local.sh`.

3.4 Mujoco

3.4.1 API reference

Mujoco is a general purpose physics engine that aims to facilitate application development in robotics, biomechanics, graphics, animation and other areas requiring light and accurate simulation tools. It is made freely by Google DeepMind and completely open source. This engine uses the built-in C/C++ library and C API, with a runtime tuned to maximize performance and operate on low-level data structures.
The models and scenes are defined by the user with MJCF (a XML format with an emphasis on human readability). These models are then compiled before simulation and transformed into data structures exploitable by the simulation engine.

Some of the keys features covered by Mujoco include :

- Interactive graphical menu
  The 3D visualizer provides rendering of meshes, geometric primitives, textures, reflections, shadows etc. Other data such as inertia boxes (for the body elements of the robots), contact points, contact normals or perturbation forces can be viewed.

- General actuation model
  Complex actuation models can be defined using a single abstract actuation element. This element can be precised and used in many types of actuation models. Futhermore, Mujoco diagnoses for faulty definition of actuation chains.

- Reconfigurable computation pipeline
  A single function, `mj_step()`, is used to run the entire forward dynamics and change the state of the simulation. This offers a flexible simulation engine that can be used to compute or render a scene from different perspectives.

Mujoco's API is user-friendly and offers many details on the available types and functions used by the simulation engine. The types we exploit the most are `mjModel`, `mjData` and `mjrContext`, with all the underlying types.

All runtime computations are performed with `mjModel` which is too complex to create manually. This is why Mujoco defined their own format MJCF. Its purpose it to translate the user representation of a dynamic model into a compilable file. This file is used to generate a low-level model and remains constant throughout the whole simulation. It contains data about the physical structures (e.g bodies, cameras, actuators, joints, etc.), environmental data (e.g lighting, materials), visualization options and many more informations (see here). It is the central data structure of the simulation application.

Furthermore, another essential structure to understand is the `mjData` data structure. It contains all dynamic variables and intermediate results. It is used as a scratch pad where all functions read their inputs and write their outputs. Its size is preallocated and internally managed so the user or runtime application should not allocate the unused buffers. We use this data container in many functions to update the simulation state but never access its components directly.

Finally, the last noticeable data structure majorly exploited in the simulation is the `mjrContext` container. It informs the renderer about the simulation's graphical context (e.g fog, color and depth, buffer, shadow map, etc.). It is the simulation's component we are most likely to use in the creation of a RGB-D camera plugin. Its color and depth buffers are essential in the generation of depth images.

3.4.2 Main program

The main program at the heart of the simulation is `mujoco_main.c`. It consists in four threads running in parallel at execution time (see figure 4). These threads share the same resources (`mjModel, mjData,`

`mjrContext`), hence indirectly exchanging data.

The `main()` function starts by loading the user's configuration file (`conf.yaml`) and creating a model and data structure from the pointed robot model and scene. This file contains details about the robot's name, the Ros2 domain identification number, etc. We added a "camera_view" field to specify which camera from the robot's model we are using to render the color and depth buffers.

Once the global parameters such as the model an data containers are set, we launch the `Unitree_bridge_sdk2()` thread whilst passing to it the rendering context of the simulation's model. At the same time, we create and execute our custom RGB-D thread with the simulation model and data. The functionalities of these two threads will be developed later.

The physics loop is next. It loads the model structure and data to start the simulation engine. It allocates and frees all the memory buffer used in the simulation. Each 1ms, a new step is computed and the result is added to a history buffer to keep track of each time frame. Many functions are also included in the thread to synchronize the steps with the graphical rendering. This is to prevent time shifts when the dynamics computation are too fast.

Finally, the main simulation object is called using a `RenderLoop()` function to initiate the 3D visualization window. All the user interface parameters, callbacks and events are set. The rendering is done each time the simulation data is modified.

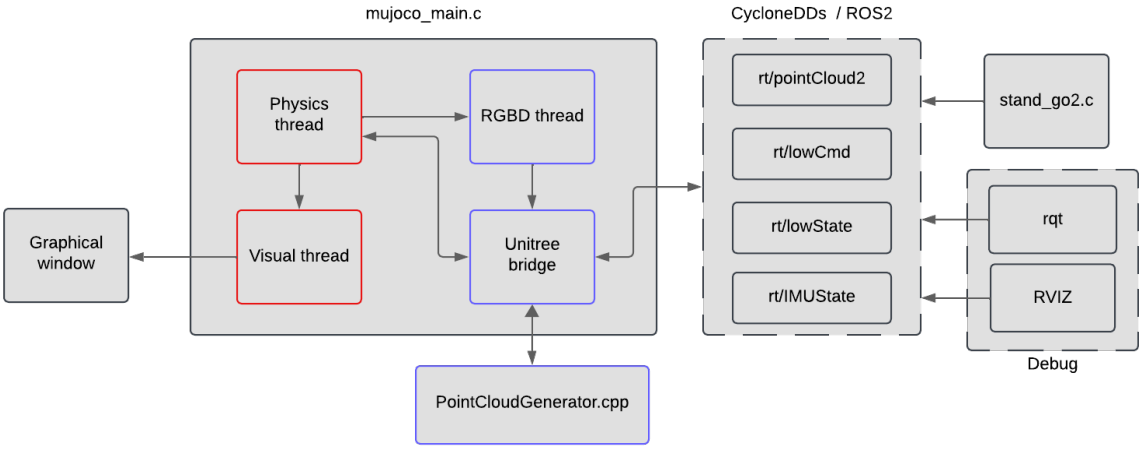


Figure 9: mujoco_main.c

3.5 Contributions

3.5.1 Unitree_bridge_sdk

This C++ class is a template provided by Unitree to interface the Mujoco simulation and the Ros2 domain. It uses many classes and methods defined in `unitree_sdk2`, so a good knowledge of their work is essential to understand the architecture of their class.

Many parameters can be found in the header file. These mainly consist in the type of objects being exchanged with the Ros2 domain (`rt/LowCmd`, `rt/LowState` or `rt/HighState`), or the Mujoco simulation structures. Pointers to sub-threads each used to publish or subscribe to different topics are declared. For our project, we added parameters relative to the publication of point cloud message (`rt/PointCloud2`).

In terms of methods, the class provides examples of thread definition and creation in the class' constructor. Each thread is bound to a callback function and can be associated to specific CPU with a preallocated stack size if needed. We chose to keep the default CPU and double the stack size allocated

for the other threads since the point cloud message structure is heavy.

After adding our thread definition for the point cloud publisher, we define the callback of the thread as a simple call to the `pointCloudGenerator.cpp` class to create the point cloud from the passed `mjrContext`.

This publisher communicates with the Ros2 domain at a frequency of 248Hz (see figure 10). However, the observed bandwidth being high (50Mb/s), we can suppose that exchanging point clouds is far from being the optimal solution to recreate 3D maps.

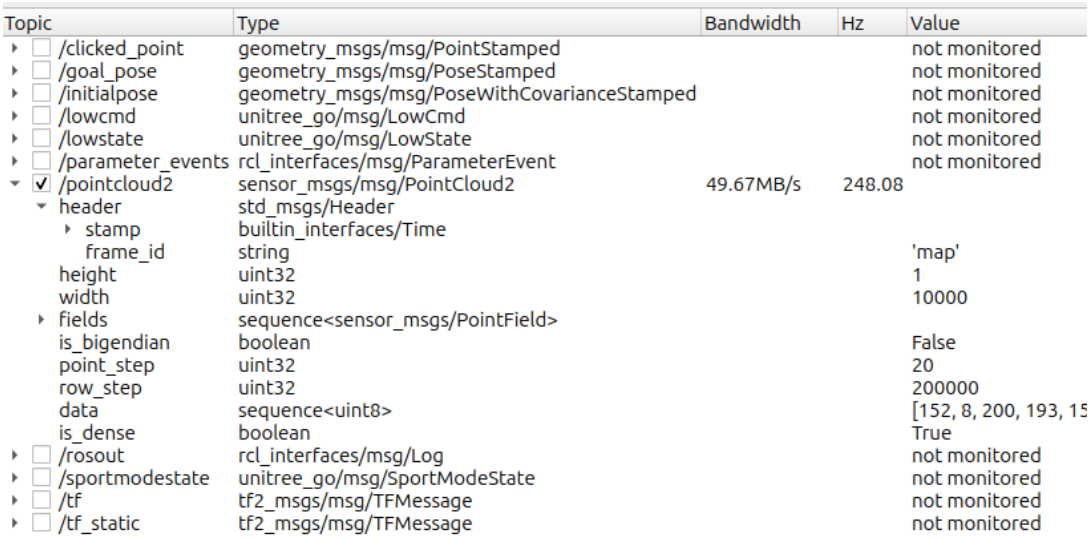


Figure 10: Rqt, topic exchange monitor.

3.5.2 RGBD thread

The `RGBD_thread` is a process that runs in parallel to the main physics loop. It aims to render the scene through the camera onboard the robot and export the related context to the `unitree_bridge_sdk` thread.
To achieve this goal, we need to get the simulation engine to render a second window from the point of view of the onboard camera. Hence, this thread instantiates a new graphical window with OpenGL's function `GLFWindow * window = glfwCreateWindow(...)`. We also copy the simulation parameters from the physics loop simulation by passing them as arguments to the current thread. From there, we are able to create and initialize new `mjvScene` and `mjrContext` objects.
During the runtime of the main program, the thread updates the scene and the context through the specified camera. The responsible function are `mjv_updateScene(...)` and `mjr_render(...)`. The scene update and rendering is done at a fixed time step of 1 second. This can be decreased, but must be considered with caution to prevent OpenGL's GPU management overload.

3.5.3 pointCloudGenerator

This C++ class acts as a tool for the `Unitree_bridge_sdk.cc` class. Its purpose is to create point clouds from a `mjrContext` structure. These point clouds can then be published through the `rt/PointCloud2` topic on the Ros2 domain.
The `pointCloudGenerator.cpp` class has private parameters that help to correctly rebuild a depth image from a picture. A few methods are also proposed to help produce point clouds.

On the one hand, the class' parameters are described as :

- `u_char * color_buffer` : This buffer contains the RGB components of each pixel from the `mjrContext`. This last data structure is also used to allocate the size of this buffer. The buffer is usable once filled. Its size is numPixels * 3 (three u_char per pixel).

- `float * depth_buffer` : This buffer contains the depth component of the pixel from the RGB-D camera specified by the `mjrContext`. Its values are normalized between 0 and 1, where 0 represents the closest distance at which points are detected (exported in `z_near`) and 1 represents the furthermost distance at which points are detected (exported in `z_fear`). Before being usable, this buffer must be linearized to obtain the real distances of the points (in meters). This is achieved with the function `void linearize_depth(float * depth)`. Its size is numPixels * 1 (one float per pixel).
- camera's intrinsic parameters : The focal lengths and center point offsets are used to correct the point position in the point cloud. The `extent` parameter acts as a depth scaler.
- `mjrRect viewport` : The viewport is similar to the resolution of the camera. It specifies the number of points in the image through its fields `viewport.width` and `viewport.height`.

On the other hand, the available methods are :

- `void linearize_depth(float * depth)` : This function admits a normalized buffer and linearizes it using the depth class' depth parameters.
- `void set_camera_intrinsics(const mjsCamera * camera)` : This function sets the intrinsic parameters of the camera used for the point cloud rendering.
- `void get_RGBD_buffer(const mjModel * model, const mjrContext * context)` : Retrieves the color and depth buffers from the given context.
- `PointCloud2_ generate_color_pointcloud(void)` : This function creates a point cloud and sets its size and fields. The point cloud object can be viewed in Rviz2 thanks to its `pcl.header.frame_id` field which specifies the coordinate frame used to position the point cloud. A more precise definition of the fields is given here.

3.5.4 Miscellaneous

The project's Cmake file is crucial in the addition of the required dependencies. For instance, to compile `mujoco_main.cc` the package `mujoco` and `unitree_sdk2` must be specified. Other includes such as the `pointCloudGenerator.cpp` must be added. In the perspective of new contributions, this file must be updated with any new folder needed for the main simulation.

Our work on the Unitree simulation can be found here.

3.6 Conclusion

These contributions achieve to create, fill and publish point clouds at a regular pace. They can be observed with Rviz2 but the results are not satisfying. Indeed, we do not obtain a smooth point cloud representing the objects seen in the simulation's graphical window.
In addition, when putting the robot in movement (by alternating between standing pose and sitting pose) using

`unitree_mujoco/example/ros2/build/stand_go2` program, the point cloud does not evolve even though it should. This is linked to an incorrect use of the simulation's context.

Furthermore, the `RGBD_thread` has not been linked to the interface nor has it been linked to the simulation correctly. Possible solutions have been commented in the code at the proper section of the main program.

Even though the end result is not convincing, the experience acquired using Mujoco's simulator is precious and can easily be reinvested in another project requiring physics simulation. This tool is powerful and the documentation that comes with it details the use perfectly.