Home

Edit New Page

Martin Molina edited this page 7 days ago · 34 revisions

Example of search and rescue mission using Aerostack:

Example of object delivery mission using Aerostack:

Pages 37

Introduction

- What is Aerostack?
- Getting Started
- License
- List of distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Catalog of Behaviors
- Memory of Beliefs

Mission Plan Specification

- TML Language
- Behavior tree
- Python Language

Tutorials [Beginners]

- Prepare the Mission:
  - Configure Map
  - Specify Mission Tasks
  - Configure Launcher
- Execute the Mission:
  - Launch Real Flight
  - Practical Tips for Real Flights
- Mission examples:
  - Simulated Quadrotor
  - AR Drone and Visual Markers

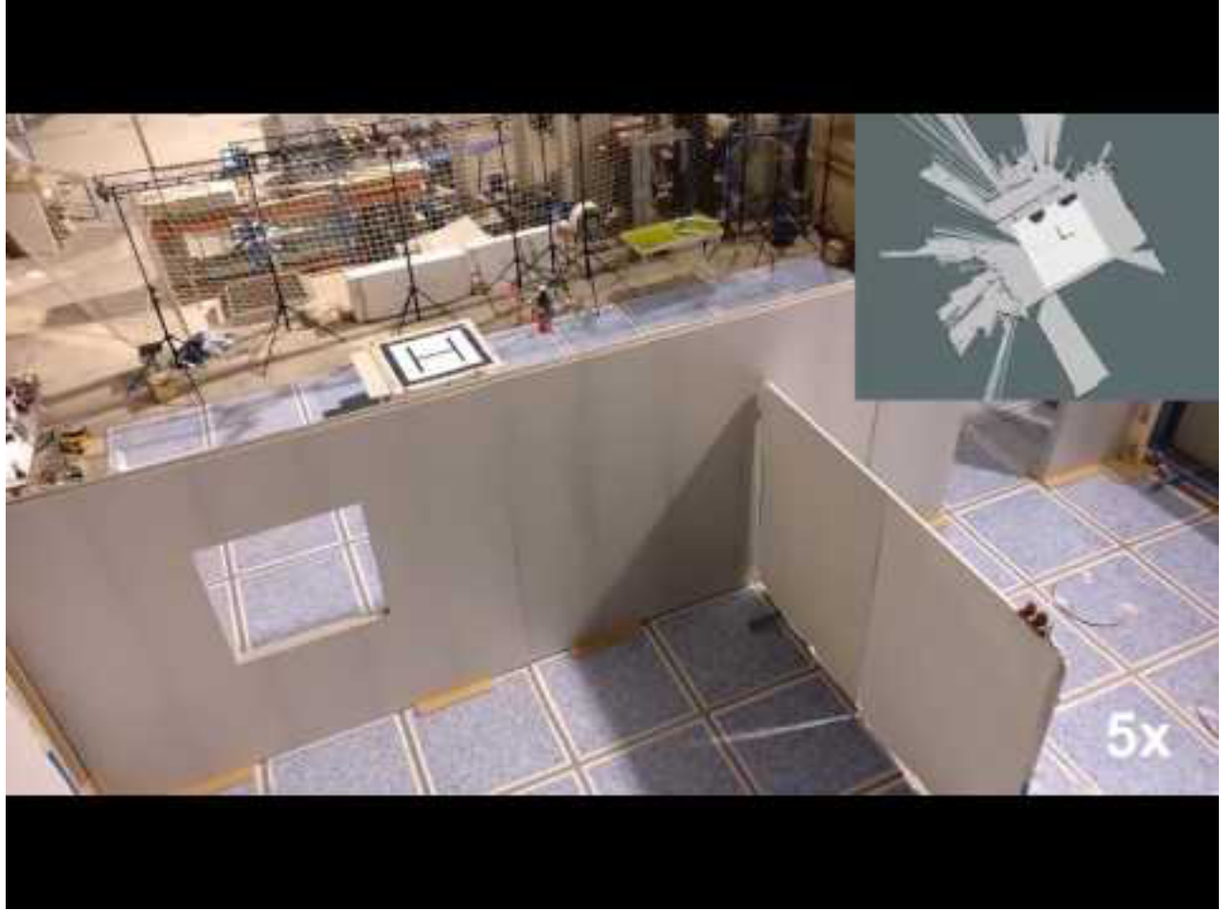
Tutorials [Advanced]

- How to program a new behavior
- Setup the Communication Network
- Wi Fi Communication with Multiple Drones
- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

More information

- Publications
- Vision & Aerial Robotics:
  - First prize at IMAV 2013
  - Stack pre-release 2013

Clone this wiki locally



https://github.com/Vision4U

Clone in Desktop

Contact

We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page.

You can also contact the team support at the following address: aerostack.vision4uav@gmail.com

What is Aerostack?

Edit New Page

Martin Molina edited this page a day ago · 29 revisions

Aerostack is a software framework that helps developers design and build the control architecture of aerial robotic systems, integrating multiple heterogeneous computational solutions (e.g., computer vision algorithms, motion controllers, self localization and mapping methods, planning algorithms, etc.).

Aerostack is useful for building autonomous aerial systems in complex and dynamic environments and it is also a useful research tool for aerial robotics to test new algorithms and architectures.

Aerostack was created to be available for communities of researchers and developers and it is currently an active open-source project (with license GPL) with periodic software releases.

Multipurpose software framework

Aerostack is versatile for building different system configurations with various degrees of autonomy:

- **From teleoperation to autonomous flight**. Aerostack can be used for teleoperation flights (with manual control) but it can also be used for building autonomous robot systems to perform aerial missions without operator assistance.
- **Single robots or multi-robot systems**. Aerostack can be used to fly a swarm of heterogeneous drones to perform multi-aerial-robot missions. It has been validated to operate with up to five drones simultaneously.
- **Flexible for different applications**. Aerostack can be used by system designers to develop their own systems in a wide range of applications. Aerostack provides languages and graphical tools to configure specific aerial missions.
- **Hardware independent**. Aerostack runs on conventional laptops and it has also run on onboard computers like mastermind or odroid XU3. Aerostack has been used in different aerial platforms, including, but not limited to:
  - Parrot AR Drone 1.0 & 2.0
  - Asctec Pelican
  - Mikrokopter Oktokopter
  - Pixhawk
  - Bebop

Operational features

The main operational features of Aerostack are the following:

- **Library of software components for aerial robotics**. Aerostack provides a library of software components for aerial robotics programmed in C++ and Python. The components are flight-proven and ready to use and the library is open to add new components. For example, these components use general algorithms of aerial robotics for motion control, feature extraction (e.g., with computer vision algorithms), self localization and mapping, motion planning, etc.
- **Compositional modeling**. Aerostack provides an architectural pattern with a combination scheme, based on the composition of configurable building blocks, to develop different specific architectures.

Pages 37

Introduction

- What is Aerostack?
- Getting Started
- License
- List of distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Catalog of Behaviors
- Memory of Beliefs

Mission Plan Specification

- TML Language
- Behavior tree
- Python Language

Tutorials [Beginners]

- Prepare the Mission:
  - Configure Map
  - Specify Mission Tasks
  - Configure Launcher
- Execute the Mission:
  - Launch Real Flight
  - Practical Tips for Real Flights
- Mission examples:
  - Simulated Quadrotor
  - AR Drone and Visual Markers

Tutorials [Advanced]

- How to program a new behavior
- Setup the Communication Network
- Wi Fi Communication with Multiple Drones
- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

More information

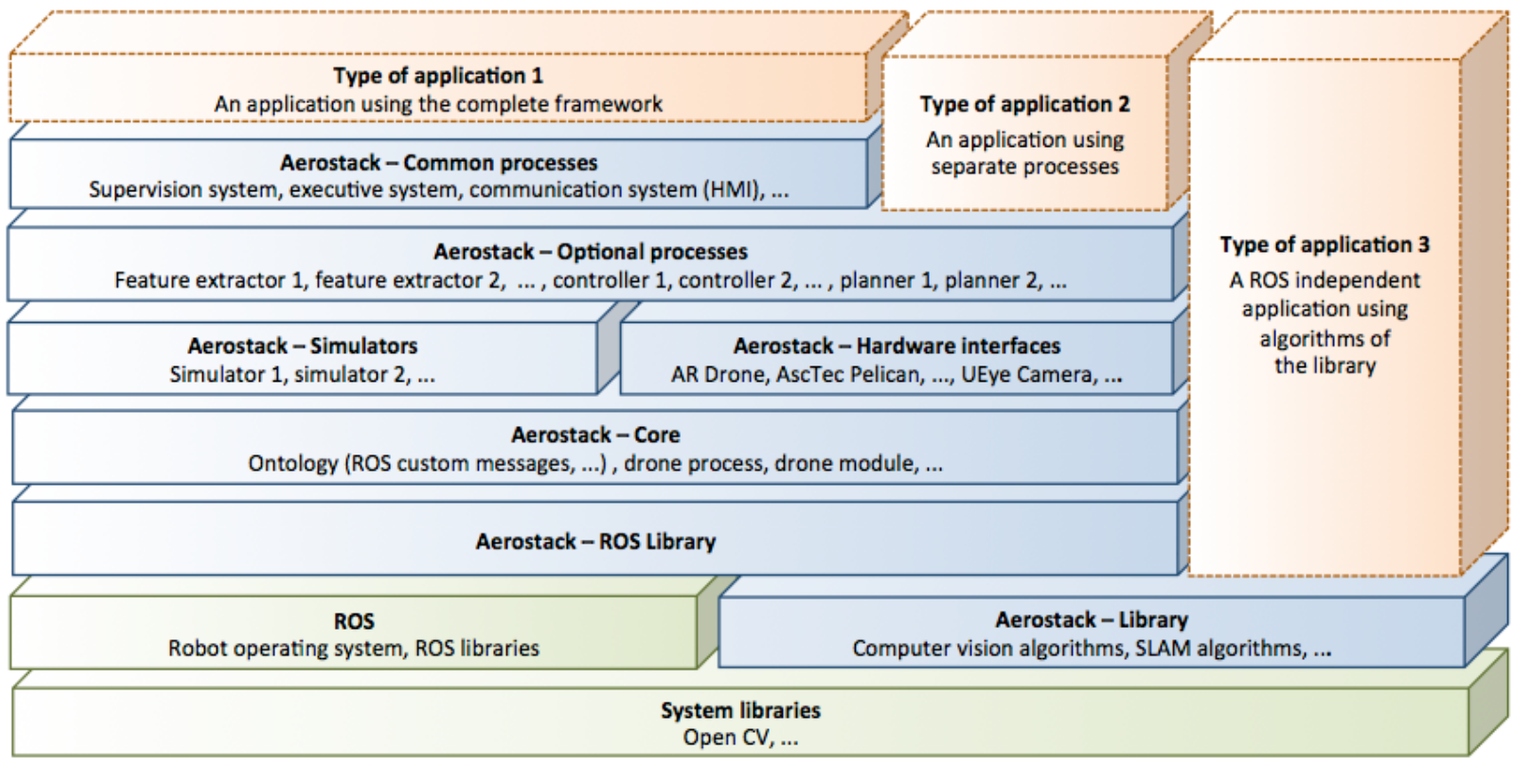
- Publications
- Vision & Aerial Robotics:
  - First prize at IMAV 2013
  - Stack pre-release 2013

Clone this wiki locally

- **Distributed processing**. Distributed processing is supported by the middleware ROS (Robot Operating System). Each software component can be executed in an on-board computer or in ground computers connected in a local area network (LAN) or in a wireless local area network (WLAN), communicated using ROS.

- **Open source**. License GPL v3.

- **Reusable components**. Aerostack uses ROS as the middleware, but we divided all the functionalities into non-ROS libraries and ROS-nodes. That means, all the functionalities and algorithm cores are coded without ROS, in a pure C++ or Python style.



Experience of Use

Aerostack has been tested by real flight tests and simulations for different types of hardware platforms. Aerostack has been used as a tool in the following situations:

- Develop new applications in aerial robotics in problems such as infrastructure/surface inspection and air quality analysis.
- Create and test new methods in aerial robotics such as the following: using computer vision to track any object with an autonomous UAS, using a coordinator to accomplish high-level missions requested by the user with an autonomous swarm of UAS, experimenting with Natural User Interfaces for Human-Drone Interaction using hand gestures, speech, body movements and visual cues, etc.
- Participate in UAS competitions such as IMAV 2013, IARC 2014, IMAV 2016, and IMAV 2017 with successful results.

Types of Applications

Developers can use Aerostack to fly UAV platforms in the following way:

- Perform tele-operated flights controlled by user commands using a graphical user interface. Examples of user commands are: take off, move forward, move backward, land, etc. The user operates with the keyboard and the mouse of the ground station computer to control the flight of the UAV.

- Tracking a selected object by visual contact. The user can select the object to track by selecting camera images using the graphical user interface.

- Following a specific mission plan, defined by the user as a set of waypoints or a set of tasks to be performed in a specific spatial area, with certain limits defined by the user.

- Use a specific physical configuration that modifies significantly the size or weight of the UAV. Our flight controllers can be adapted by the user to the new configuration (in a XML file).

Developers who are familiar with software programming (e.g., C++ language) can use Aerostack to fly new UAV platforms with the following characteristics:

https://github.com/Vision4U
Clone in Desktop

- A UAV with new specific types of sensors. The developer can reuse Aerostack but she/he needs to program and integrate new software modules to process the information from the new sensors.

- A UAV with different physical platform. The developer can reuse Aerostack for different UAV platforms, but she/he needs to program the interfaces between our architecture and the actuators of the new physical platform.

- A UAV with one or several software components that substitutes an existing software component (for example, a new localization and mapping). The developer can reuse Aerostack but she/he might need to substitute the core algorithm used by the previous component within the package of optional processes.

- A UAV with new functionalities that needs additional software components. The developer can reuse Aerostack but she/he needs to program and integrate the new software modules for the new functionalities.

- Developers can also reuse specific algorithms (e.g., perception algorithms or certain controllers) that are encapsulated in their corresponding C++ classes in Aerostack. Developers can reuse these C++ classes with their algorithms to be part of their own development.

Contributors

The main kernel of Aerostack was initially created and pre-released in 2013 by Vision4UAV (Technical University of Madrid - Spain) based on the experience of this group in aerial robotics and computer vision during more than 20 years. It was firstly introduced and demonstrated during the participation of Vision4UAV in the 2013 International Micro Air Vehicle Indoor Flight Competition IMAV2013 (Toulouse, France).

In 2015, with the contribution of other members of the Technical University of Madrid from the Department of Artificial Intelligence, the software was redesigned and extended to create the present version with the name of Aerostack (AErial RObotics STACK). This new version was the evolution from a previous control-oriented architecture to a more structured multi-layered architecture following hybrid paradigms (e.g., with deliberative and reactive layers) and social coordination. The first release of Aerostack (Version 1.0, Gregale distribution) was in June 2016.

Currently, a team from the Technical University of Madrid provides support for Aerostack. You can contact the team support at the following address:

- aerostack.vision4uav@gmail.com

The following list includes Aerostack contributors:

Aerostack administrators:

- Hriday Bavle
- Alberto Camporredondo

Research professors:

- Pascual Campoy
- Martin Molina
- Paloma de la Puente

Research assistants:

- Adrian Carrio
- Jean-François Collumeau
- Jesus Pestana Puerta
- Alejandro Rodriguez Ramos
- Carlos Sampedro
- Jose Luis Sanchez-Lopez

- Ramon Suarez-Fernandez

Software developers:

- Jorge Martin [2013-2014]
- Marcos Bernal [2014-2015]
- Yolanda de la Hoz [2014-2015]
- Enrique Ortiz [2015]
- Adrian Navarro [2015]
- Angel Luis Gonzalez [2015]
- Laura Garcia [2015]
- Oscar Fernandez [2015]
- Adrian Diaz-Moreno [2015-2016]
- David Palacios [2015-2016]
- German Quintero [2016]
- Guillermo de Fermin [2016-2017]
- Alberto Camporredondo [2016-2018]
- Carlos Valencia [2016-2017]
- Pedro Frau Amar [2017]
- Jorge Sanchez [2017]
- Jorge Pascual [2017]
- Rafael Artiñano [2017-2018]
- Daniel Rabasco [2017]
- Daniel del Olmo [2018]
- Abraham Carrera [2018]

Funding Support

The development of Aerostack has been partially supported by the Spanish Ministry of Economy and Competitiveness through the project VA4UAV (Visual autonomy for UAV in Dynamic Environments) reference DPI2014-60139-R.

Contact

We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page.

You can also contact the team support at the following address: aerostack.vision4uav@gmail.com

Getting Started

Martin Molina edited this page on 26 Oct 2017 · 8 revisions

This section shows with a simple example how to install Aerostack and control manually the flight of a drone.

1. Hardware Configuration

In this example we assume that the following hardware is used:

- An air vehicle Parrot AR Drone (other aerial platforms can be used with Aerostack).
- A computer with wifi connection.

2. Software Installation

Aerostack has been tested in a computer with the following software:

- Linux operating system Ubuntu 16.04 (64 bit).
- Robot Operating System (ROS), Kinetic distribution.

However Aerostack should also work on any other mainstream Linux distributions. The code requires a compiler that is compatible with the C++11 standard.

Aerostack requires additional libraries like: boost and ncurses (among others). Aerostack has ROS packages dependencies like: ardrone_autonomy, opencv2, roscpp, image_transport, sensor_msgs and std_srvs (among others). All the dependencies are automatically installed during the installation process.

The installation follows the same steps needed usually to compile a self-contained ROS stack:

1. Before the installation, you must read and accept the conditions defined by the software License.
2. Download the installation script file:

```
$ git clone https://github.com/Vision4UAV/Aerostack_installer ~/temp
```

3. Run it in a non-ARM architecture computer:

```
$ ~/temp/installation_script.sh
```

4. Re-open the terminal and Aerostack is ready to be used.

If you need to update the software:

1. Go to the Aerostack folder:

Pages 37

Introduction
- What is Aerostack?
- Getting Started
- License
- List of distributions

Components
- Aerostack Architecture
- Graphical User Interface
- Catalog of Behaviors
- Memory of Beliefs

Mission Plan Specification
- TML Language
- Behavior tree
- Python Language

Tutorials [Beginners]
- Prepare the Mission:
  - Configure Map
  - Specify Mission Tasks
  - Configure Launcher
- Execute the Mission:
  - Launch Real Flight
  - Practical Tips for Real Flights
- Mission examples:
  - Simulated Quadrotor
  - AR Drone and Visual Markers

Tutorials [Advanced]
- How to program a new behavior
- Setup the Communication Network
- Wi Fi Communication with Multiple Drones
- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

More information
- Publications
- Vision & Aerial Robotics:
  - First prize at IMAV 2013
  - Stack pre-release 2013

Clone this wiki locally

https://github.com/Vision4U
Clone in Desktop

```
$ cd $AEROSTACK_STACK
```

2. Update the stack:

```
$ git pull origin master
```

3. Update submodules:

```
$ git submodule update --init --recursive
```

Sometimes, you will need to install new dependencies or update the old dependencies what is done by executing the following command:

```
$ $AEROSTACK_STACK/installation/installation_dep_script.sh
```

> NOTE: Be sure that you are connected to Internet when you compile Aerostack. This is because certain packages (e.g., opentld) need to be updated from remote repositories.

3. A First Flight

To launch and start running the system, the user has to open a new terminal and, then, perform the following tasks:

1. Make sure the drone is on, and reset it if needed.
2. Connect the computer to the drone wifi (e.g., ardrone2_002410).
3. Execute the ROS master node:

```
$ roscore
```

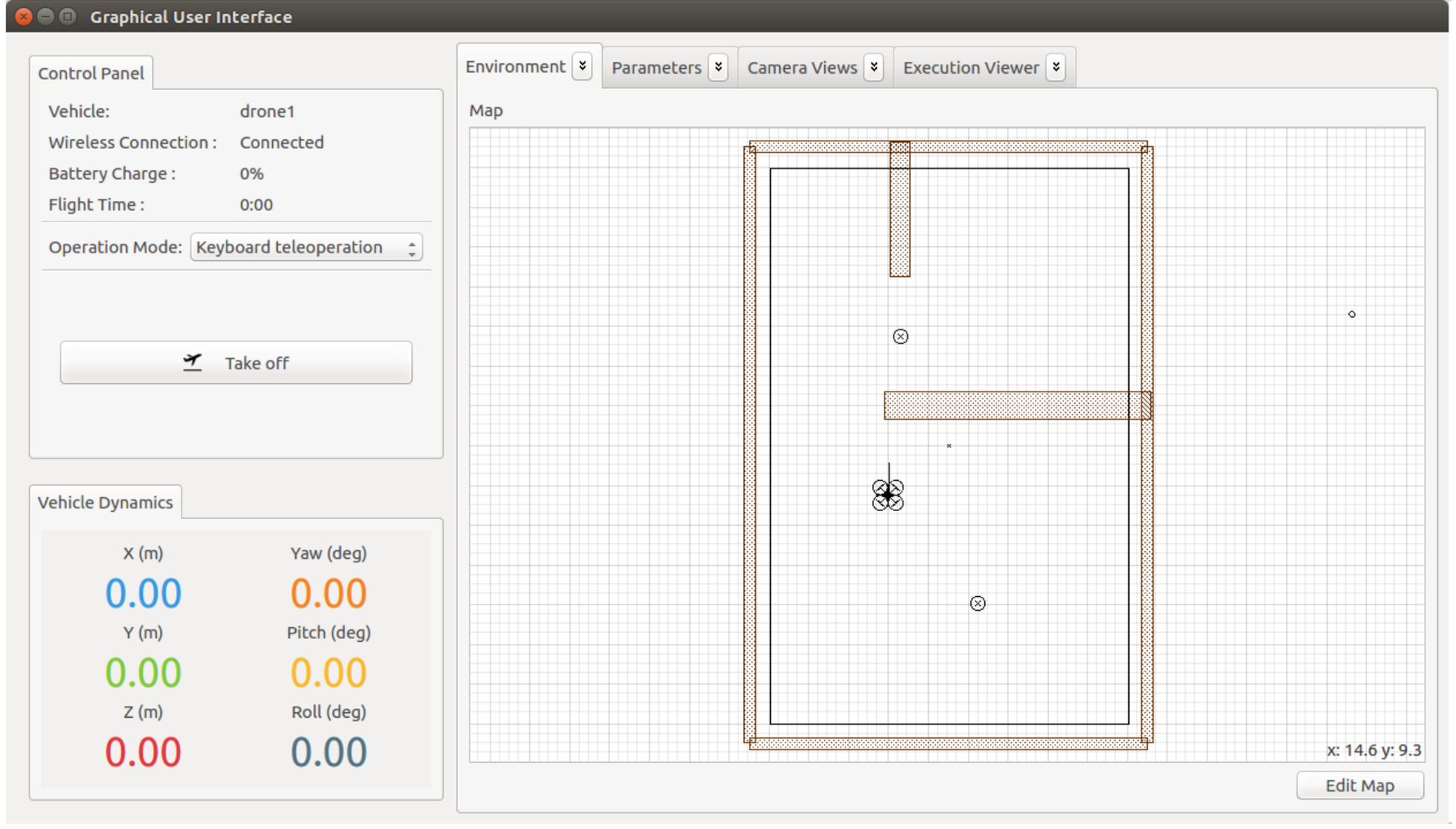
4. In a new terminal window, go to the directory which contains the launchers:

```
$ cd ${AEROSTACK_STACK}/launchers
```

5. Execute the script to launch all processes and the Human Machine Interface:

```
$ ./ardrone_basic.sh
```

Now, the installation is complete and it is possible to operate with the drone through the user interface controls (see details about the operation control in the chapter about the human machine interface).



To stop and terminate all program executions, it is necessary to perform the following steps in the terminal console:

```
$ cd ${AEROSTACK_STACK}/launchers
$ ./stop.sh
```

Contact

We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page.

You can also contact the team support at the following address: aerostack.vision4uav@gmail.com

License

Edit New Page

Martin Molina edited this page 19 days ago · 8 revisions

Aerostack - A Software Framework for Aerial Robotic Systems

Copyright (C) 2017 Technical University of Madrid

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

Safety Warning and Disclaimer

You are using this software at your own risk. The authors decline any responsibility for personal injuries and/or property damage.

Some drones supported by this framework ARE NOT TOYS. Even operation by expert users might cause SERIOUS INJURIES to people around. Therefore, be sure that you fly in a properly screened or isolated flight area.

Some components of this software were developed as part of research work in an academic context in the form of prototypes with LIMITED VALIDATION. If you plan to use this software in a final application, be sure that you perform an additional EXHAUSTIVE VALIDATION procedure.

Third-Party Libraries

Aerostack uses other third-party open-source libraries:

- Robot Operating System (ROS), license: `BSD` license, not included in the source of the stack.
- ardrone_autonomy ROS package (ardrone_autonomy), license: `BSD` license, not included in the source of the stack.
- Open Source Computer Vision library (OpenCV), license: `BSD` license, not included in the source of the stack.
- ArUco: a minimal library for Augmented Reality applications based on OpenCv (ArUco), original license: `BSD` license, included as source in the stack in the `lib_aruco` package.
- pugixml library (pugixml), original license: `MIT` license, included as source in the stack in the `lib_pugixml` package.
- newmat11 library, a matrix library in C++ (newmat11), original license: (newmat11 license), included as source in the stack in the `lib_newmat` package.

Pages 37

Introduction
- What is Aerostack?
- Getting Started
- License
- List of distributions

Components
- Aerostack Architecture
- Graphical User Interface
- Catalog of Behaviors
- Memory of Beliefs

Mission Plan Specification
- TML Language
- Behavior tree
- Python Language

Tutorials [Beginners]
- Prepare the Mission:
  - Configure Map
  - Specify Mission Tasks
  - Configure Launcher
- Execute the Mission:
  - Launch Real Flight
  - Practical Tips for Real Flights
- Mission examples:
  - Simulated Quadrotor
  - AR Drone and Visual Markers

Tutorials [Advanced]
- How to program a new behavior
- Setup the Communication Network
- Wi Fi Communication with Multiple Drones
- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

More information
- Publications
- Vision & Aerial Robotics:
  - First prize at IMAV 2013
  - Stack pre-release 2013

Clone this wiki locally

- other software libraries used by this stack also uses licenses similar to `BSD` and `MIT` licenses: ncurses and boost.

https://github.com/Vision4U

Clone in Desktop

Contact

We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page.

You can also contact the team support at the following address: aerostack.vision4uav@gmail.com

List of distributions

Edit New Page

Martin Molina edited this page on 3 Nov 2017 · 8 revisions

Distribution	Release date	Main features	Tested platforms	Requirements
Version 1.0 Gregale Distribution	June 2016 PDF wiki	Aerial robot library of software components (perception, control, planning, etc.), TML language for mission plan specification, human machine interface (HMI) for user-robot interaction.	Parrot AR Drone 1.0 and 2.0, Asctec Pelican, Mikrokopter Oktokopter, Pixhawk.	Ubuntu 14.04, ROS Jade
Version 2.0 Levant Distribution	October 2017	Behavior-based execution system, graphical user interface (GUI) for environment maps and mission plans, Python API for mission plans, Memory of beliefs	Parrot AR Drone 2.0, Pixhawk, Parrot Bebop 2 (with SLAM-dunk).	Ubuntu 16.04, ROS Kinetic

You can download a Virtual Machine (Virtual Box VM) with Aerostack 1.0 (Gregale) in the following link:

- Aerostack Virtual Machine

Pages 37

Introduction

- What is Aerostack?
- Getting Started
- License
- List of distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Catalog of Behaviors
- Memory of Beliefs

Mission Plan Specification

- TML Language
- Behavior tree
- Python Language

Tutorials [Beginners]

- Prepare the Mission:
  - Configure Map
  - Specify Mission Tasks
  - Configure Launcher
- Execute the Mission:
  - Launch Real Flight
  - Practical Tips for Real Flights
- Mission examples:
  - Simulated Quadrotor
  - AR Drone and Visual Markers

Tutorials [Advanced]

- How to program a new behavior
- Setup the Communication Network
- Wi Fi Communication with Multiple Drones
- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

More information

- Publications
- Vision & Aerial Robotics:
  - First prize at IMAV 2013
  - Stack pre-release 2013

Clone this wiki locally

https://github.com/Vision4U

Clone in Desktop

Contact

We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page.

You can also contact the team support at the following address: aerostack.vision4uav@gmail.com

Aerostack Architecture

Edit New Page

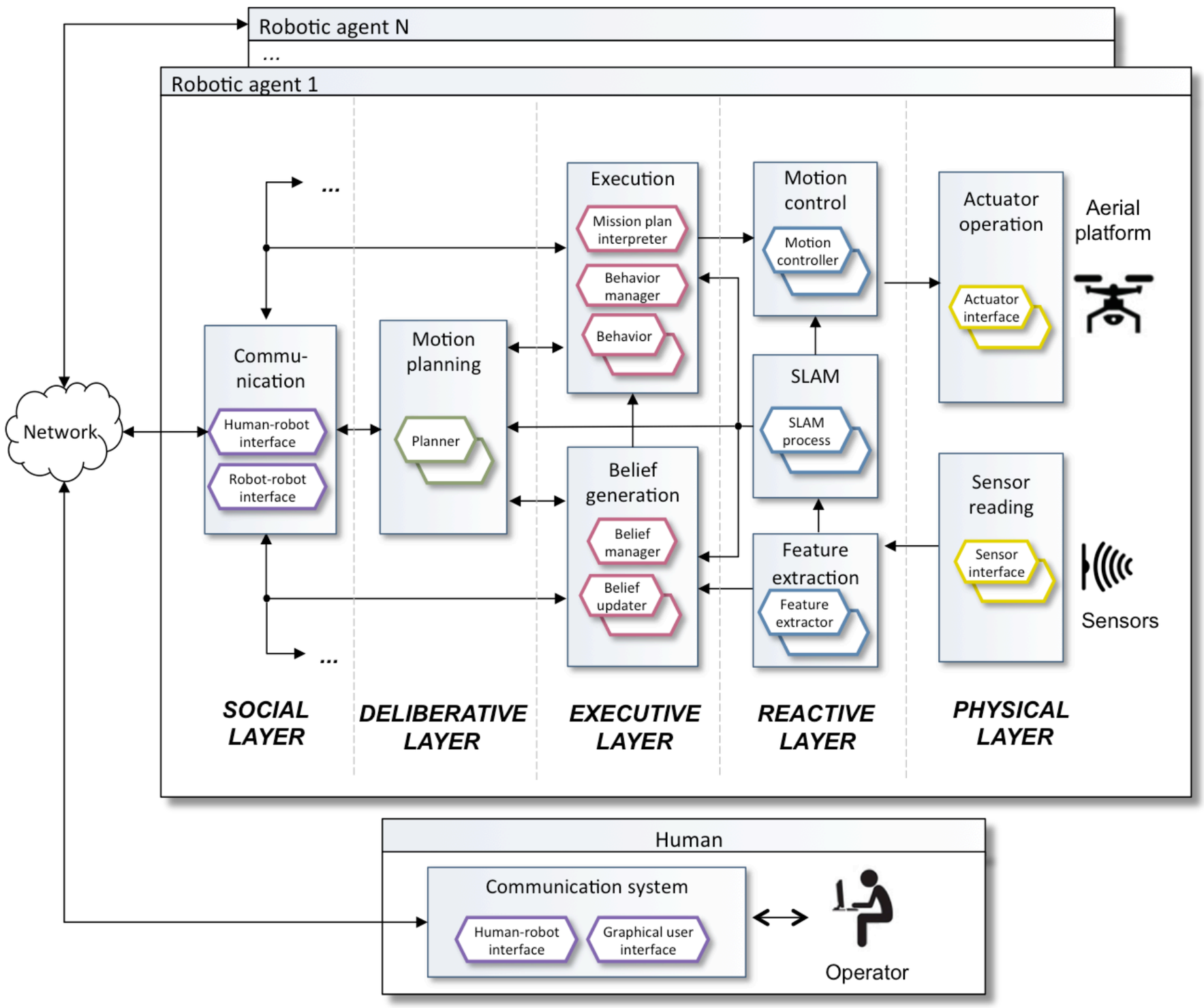
Martin Molina edited this page on 24 Nov 2017 · 10 revisions

This section summarizes the main components of the Aerostack architecture and its structural organization.

Main Components

The next figure shows the main functional components of the architecture. This design follows the hybrid reactive/deliberative paradigm, i.e., an architecture that integrates both a deliberative and reactive approaches [Arkin et al., 1987]. Our design includes several layers: physical, reactive, executive, deliberative and social.

The three layers reactive-executive-deliberative are based on the popular hybrid design known as the three-layer architecture [Gat ,1998] that organizes algorithms according to whether they contain no internal state (reactive layer), contain state reflecting memories about the past (executive layer) or contain state reflecting predictions about the future (deliberative layer).



The reactive layer creates a sensor-action loop that includes feature extractors and motion controllers. Feature extractors may read simple states of sensors or may implement complex vision and pattern recognition algorithms. Examples of feature extractors are: extract color, compute centroid of an image, recognize ArUco visual markers, detect power line tower, etc. A perceptual schema can integrate a set of perception procedures using different combination methods. Our reactive layer may also include methods for SLAM (self-localization and mapping).

Motion controllers typically implement combinations of proportional-integral-derivative (PID) controllers (e.g., cascade controllers). For example, this type of controllers can accept orders about a desired value for a variable (position, speed, altitude, and yaw) in form of single commands or simultaneous commands that are translated into low level commands to be sent to the actuator interface.

Pages 37

Introduction

- What is Aerostack?
- Getting Started
- License
- List of distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Catalog of Behaviors
- Memory of Beliefs

Mission Plan Specification

- TML Language
- Behavior tree
- Python Language

Tutorials [Beginners]

- Prepare the Mission:
  - Configure Map
  - Specify Mission Tasks
  - Configure Launcher
- Execute the Mission:
  - Launch Real Flight
  - Practical Tips for Real Flights
- Mission examples:
  - Simulated Quadrotor
  - AR Drone and Visual Markers

Tutorials [Advanced]

- How to program a new behavior
- Setup the Communication Network
- Wi Fi Communication with Multiple Drones
- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

More information

- Publications
- Vision & Aerial Robotics:
  - First prize at IMAV 2013
  - Stack pre-release 2013

Clone this wiki locally

https://github.com/Vision4U
Clone in Desktop

The executive layer accepts directives from the deliberative layer and sequences them for the reactive layer. The executive layer follows a behavior-based approach including a behavior manager. The behavior manager receives a high level command about a desired action (e.g., go to a desired position) and translates it into references for controllers. The executive layer also integrates the sensor information into an internal state representation using a memory of symbolic beliefs.

The architecture includes also a social layer with communication abilities, as it is proposed in multiagent systems and other architectures with social coordination (e.g., [Duffy, 2000]).

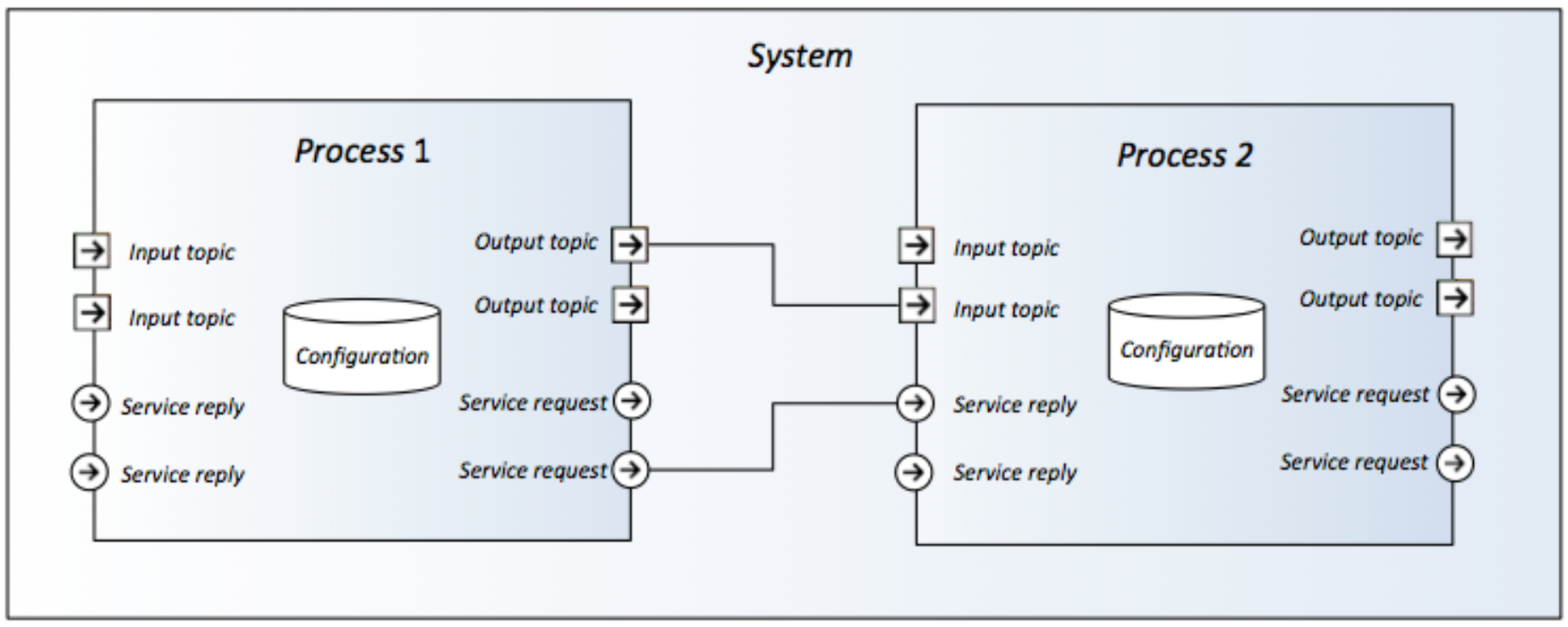
The organization in separate layers is useful, for example, to organize the independent execution of processes that operate at high frequencies (e.g., reactive processes, with frequencies between 10 Hz and 1000 Hz) and processes that operate at low frequencies (e.g., deliberative processes, with frequencies between 0.1 Hz and 10 Hz). For example, planning algorithms can be computationally expensive, so they should be decoupled of the processes in the reactive layer to avoid slow down the reaction time.

The architecture is also consistent with the usual components related to guidance, navigation and control of unmanned rotorcraft systems [Kendoul, 2012]. In particular, the Navigation System (NS) corresponds to our perception system and situation awareness system, the Guidance System (GS) corresponds to our executive system and planning system and, finally, the Flight Control System (FCS) corresponds to our motion system.

Architecture Organization: Processes and Systems

Aerostack uses the name of **process** to call each elemental component of the architecture. Each process has a function, i.e., a purpose or practical use for which the process is designed, and it is named as an agent according to its main function, for example: mission planner (main function: planning a mission) or obstacle recognizer (main function: recognize obstacles). Processes are grouped in **systems**. A system is a complex module that includes a set of interconnected processes that provides a common functionality. The ideas of process and system provide modularity and are useful to divide the whole problem of automated support for aerial robotic systems into partial functional roles.

The notions of process and system are similar to other concepts used in certain system design methodologies. For example, a process is similar to an atomic functional block used in SysML with input/output ports. The idea of system is also similar to the concept of functional block (composite block) used in SysML, with input/output ports. Next figure shows a graphical notation to describe the inter-process communication of data processors of the Aerostack architecture. This notation is based partially on functional block diagrams and SysML (internal block diagrams).



The computational support of a process is an atomic executable unit (a data processor) that receives input data and, as a result of a certain information processing, generates output data. The processes of Aerostack are implemented using ROS (Robot Operating System). Each process is a ROS node. When Aerostack is running for a particular robotic system, each process is normally an executable instance of a program running in a computer (in certain situations, a process can also have several executable instances). Aerostack executes independently processes in a multi-tasking operating system (Linux). Aerostack runs concurrently processes using the inter-process communication methods provided by ROS: (1) a publish-subscribe mechanism using messages and topics, and (2) a request-reply scheme (services).

Aerostack provides a library of reusable processes for aerial robotics. The library of processes is open, so new processes can be included easily in the future, without changing the core of the system and the rest of processes. To create a new process it is necessary to program the corresponding algorithms and data structures in a class (e.g., in C++ language) and apply certain Aerostack conventions to be part of the library. For example, it must be created as a ROS node and it must be subclass of a specific class (called RobotProcess) that provides a common functionality (supervision, execution control and standard error messages)

To consult the available processes provided by Aerostack, consult the following catalog:

- Catalog of Processes

References

- Arkin, R. C., Riseman, E. M., and Hansen, A. (1987): "AuRA: An Architecture for Vision-Based Robot Navigation," proceedings of the DARPA Image Understanding Work-shop, Los Angeles, CA, February, pp. 417–413.
- Duffy, B.R. (2001). "The Social Robot". Doctoral dissertation, Dept. of Computer Science, University College Dublin, Ireland.
- Gat , E. (1998): "On Three-Layer Architectures". In "Artificial Intelligence and Mobile Robots", (David Kortenkamp, R. Peter Bonnasso, and Robin Murphy, eds.) AAAI Press.
- Kendoul, F. (2012). Survey of advances in guidance, navigation, and control of unmanned rotorcraft systems. Journal of Field Robotics, 29(2), 315-378.

Contact

We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page.

You can also contact the team support at the following address: aerostack.vision4uav@gmail.com

Catalog of Processes

Martin Molina edited this page on 26 Oct 2017 · 1 revision

This catalog shows the available processes provided by Aerostack organized according to robot capabilities. The catalog also shows the launcher and the system corresponding to each process. In order to have a desired capability for a particular robot, the developer must select the launchers of processes described in this page.

> IMPORTANT NOTE: The list of available processes of this catalog is still incomplete and needs to be updated.

Capability: Operation with an aerial platform

Option: Operation with Parrot AR Drone 2.0

Process	System	Launcher
Drivers platform	Hardware interface	driverParrotARDroneROSModule.launch

This option also requires the launcher:

`${AEROSTACK_STACK}/launchers/ardrone_launch/ardrone_indoors.launch`

> NOTE: Other aerial platforms not described here can also be used with Aerostack (e.g., Asctec Pelican, Bebop, etc.).

Option: Operation with a RViz-based simulator

Process	System	Launcher
Simulator	Simulation system	SimulatorRvizROSModule.launch

Capability: Aruco visual markers recognition

Process	System	Launcher
Aruco eye	Feature extraction	droneArucoEyeROSModule.launch

Capability: Self-localization

Option: Self-localization by Aruco visual markers

Required capability: **Aruco visual markers recognition**

Process	System	Launcher
Visual markers localizer	Situation awareness	droneVisualMarkersLocalizerROSModule.launch
Obstacle detector visual marks	Situation awareness	droneObstacleProcessor2dVisualMarksROSModule.launch
Obstacle distance calculator	Situation awareness	droneObstacleDistanceCalculationROSModule.launch
	Situation	

Pages 37

Introduction

- What is Aerostack?
- Getting Started
- License
- List of distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Catalog of Behaviors
- Memory of Beliefs

Mission Plan Specification

- TML Language
- Behavior tree
- Python Language

Tutorials [Beginners]

- Prepare the Mission:
  - Configure Map
  - Specify Mission Tasks
  - Configure Launcher
- Execute the Mission:
  - Launch Real Flight
  - Practical Tips for Real Flights
- Mission examples:
  - Simulated Quadrotor
  - AR Drone and Visual Markers

Tutorials [Advanced]

- How to program a new behavior
- Setup the Communication Network
- Wi Fi Communication with Multiple Drones
- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

More information

- Publications
- Vision & Aerial Robotics:
  - First prize at IMAV 2013
  - Stack pre-release 2013

Clone this wiki locally

Self localizer	awareness	droneEKFStateEstimatorROSModule.launch

https://github.com/Vision4U

Clone in Desktop

Capability: Setpoint-based flight control

Required capability: **Self-localization**

Process	System	Launcher
Basic actions controller	Motor system	droneMidLevelAutopilotROSModule.launch
Trajectory controller	Motor system	droneTrajectoryControllerROSModule.launch

Capability: Vision-based flight control

Process	System	Launcher
Tracker eye	Feature extraction	droneTrackerEyeROSModule.launch
Visual servoing control	Motor system	DroneIBVSControllerROSModule.launch
OpenTLD translator	Motor system	droneOpenTLDTranslatorROSModule.launch

Required launchers:

`${AEROSTACK_STACK}/launchers/ardrone_launch/launch_files/opentld_for_IBVSController.launch`

`${AEROSTACK_STACK}/launchers/ardrone_launch/launch_files/opentld_gui_for_IBVSController.launch`

NOTE: To activate this Feature for the AR Drone it is necessary to copy the files from the config folder:

`ardrone_camera_bottom.yaml`

`ardrone_camera_front.yaml`

to the folder in the home directory:

`.ros/camera_info/`

Capability: Path planning

Required capability: **Setpoint-based flight control**

Process	System	Launcher
Trajectory planner	Planning system	droneTrajectoryPlanner2dROSModule.launch
Yaw commander	Planning system	droneYawCommanderROSModule.launch

Capability: Action execution

Required capabilities: **Path planning**, **Setpoint-based flight control**, **Vision-based flight control**

Option: Execution of simple actions

Process	System	Launcher
Manager of actions	Executive system	droneManagerOfActionsROSModule.launch

Option: Execution of advanced actions

Process	System	Launcher

Behavior manager	Executive system	behavior_manager_process.launch
Manager of actions	Executive system	droneManagerOfActionsROSModule.launch
Action monitor	Supervision system	action_monitor_process.launch

Capability: Mission plan interpretation

Option: Mission schedule processor

Required capability: **Action execution (Option: Execution of simple actions)**

Process	System	Launcher
Mission planner	Planning system	droneMissionSheduleProcessorROSModule.launch

Option: Task-based mission plan interpreter

Required features: **Action execution (Option: Execution of advanced actions)**

Process	System	Launcher
Mission planner	Planning system	task_based_mission_planner_process.launch
Action specialist	Planning system	action_specialist_process.launch

Capability: User interface of ground control station

Option: Alphanumeric user interface

Process	System	Launcher
Graphical User Interface	Ground control system	droneInterface_jp_ROSModule.launch

Option: Graphical User Interface

Process	System	Launcher
Graphical User Interface	Ground control system	human_machine_interface_process.launch

Option: Speech-based communication

Process	System	Launcher
Speech module	Ground control station	droneSpeechROSModule.launch
Sound module	Ground control station	droneSoundROSModule.launch

Required launcher:

`${AEROSTACK_STACK}/launchers/sound_play.launch`

Option: Communication with leap motion

Process	System	Launcher
Leap motion	Ground control station	droneLeapMotionDroneControlUserInterfaceROSModule.launch

Capability: Swarm Coordination

Process	System	Launcher

Communication manager	Communication system	droneCommunicationManagerROSModule.launch

Capability: Process monitoring

This capability must be always active.

Process	System	Launcher
Process monitor	Supervision system	process_monitor_process.launch

Contact

We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page.

You can also contact the team support at the following address: aerostack.vision4uav@gmail.com

Graphical User Interface

Edit New Page

Martin Molina edited this page on 26 Oct 2017 · 4 revisions

This page describes the graphical user interface (GUI) of Aerostack 2.0 (Levant distribution). The GUI allows an operator to operate with an aerial robot from a ground station, observing states and dynamics. The GUI presents graphical views and images to help the operator to understand both the external and internal behavior of the robot.

The GUI offers to the robot operators a bi-directional communication with each aerial robot system. Both communication directions are equally important because:

- The communication from the robotic agent to the operator is needed to monitor its state and the performance of the requested mission. This will allow the operator to actuate in case of a need to redefine or abort the mission. A good user graphical interface simplifies this task.
- The communication from the operator to the robotic agent is needed if a non-rigid behavior is required to accomplish the goal, making it possible to redefine the mission and also to start, stop, abort the mission or repair certain unexpected malfunctions that may occur during the mission execution.

In general, the operator can use a graphical user interface to perform the following types of tasks:

- Specify the robot's behavior in advance (robot set up). Before a mission starts, the operator can specify what kind of behavior the robot must follow by selecting an operation mode (teleoperated, guided by python mission plan or guided by behavior trees), selecting this way a mission to be done.
- Monitor the robot's behavior during a complex mission. The operator can monitor the state and the performance of a requested mission. Here, a non-rigid behavior is required, making it possible to redefine the mission and also to start, stop, abort the mission or repair certain unexpected malfunctions that may occur during the mission execution. A good graphical user interface simplifies this task.
- Operate manually with simple movements. The operator can operate manually the robot using the keyboard to send basic motion commands (move forwards, backwards, etc.) and emergency actions (land, stop, reset, etc.).
- Collect data for later use. The operator can also use the GUI to collect relevant data to be used later. This includes, for example, images (photographs, videos) or data from other types of sensors (lidar, temperature, etc.).

The GUI includes some procedures used in most of the standard user interfaces to operate with aerial vehicles. The GUI also includes other specific features. For example, when an operator monitors the robot's behavior during a mission, it is possible to distinguish between different levels of descriptions:

- *Mission level*. This level corresponds to the language used to describe the mission goals. Here the operator and the vehicle use concepts such as the following: mission, task, behavior, task or belief.
- *Operation level*. This level corresponds to observed facts or derived facts by the vehicle during its operation. Here the operator uses the following concepts: internal states, (e.g., battery state, armed or disarmed, flying or landed, state of wifi connection, etc.), parameters measured by sensors or computed by specific algorithms (pose, yaw, pitch, roll, etc.), and images obtained from cameras.

Pages 37

Introduction

- What is Aerostack?
- Getting Started
- License
- List of distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Catalog of Behaviors
- Memory of Beliefs

Mission Plan Specification

- TML Language
- Behavior tree
- Python Language

Tutorials [Beginners]

- Prepare the Mission:
  - Configure Map
  - Specify Mission Tasks
  - Configure Launcher
- Execute the Mission:
  - Launch Real Flight
  - Practical Tips for Real Flights
- Mission examples:
  - Simulated Quadrotor
  - AR Drone and Visual Markers

Tutorials [Advanced]

- How to program a new behavior
- Setup the Communication Network
- Wi Fi Communication with Multiple Drones
- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

More information

- Publications
- Vision & Aerial Robotics:
  - First prize at IMAV 2013
  - Stack pre-release 2013

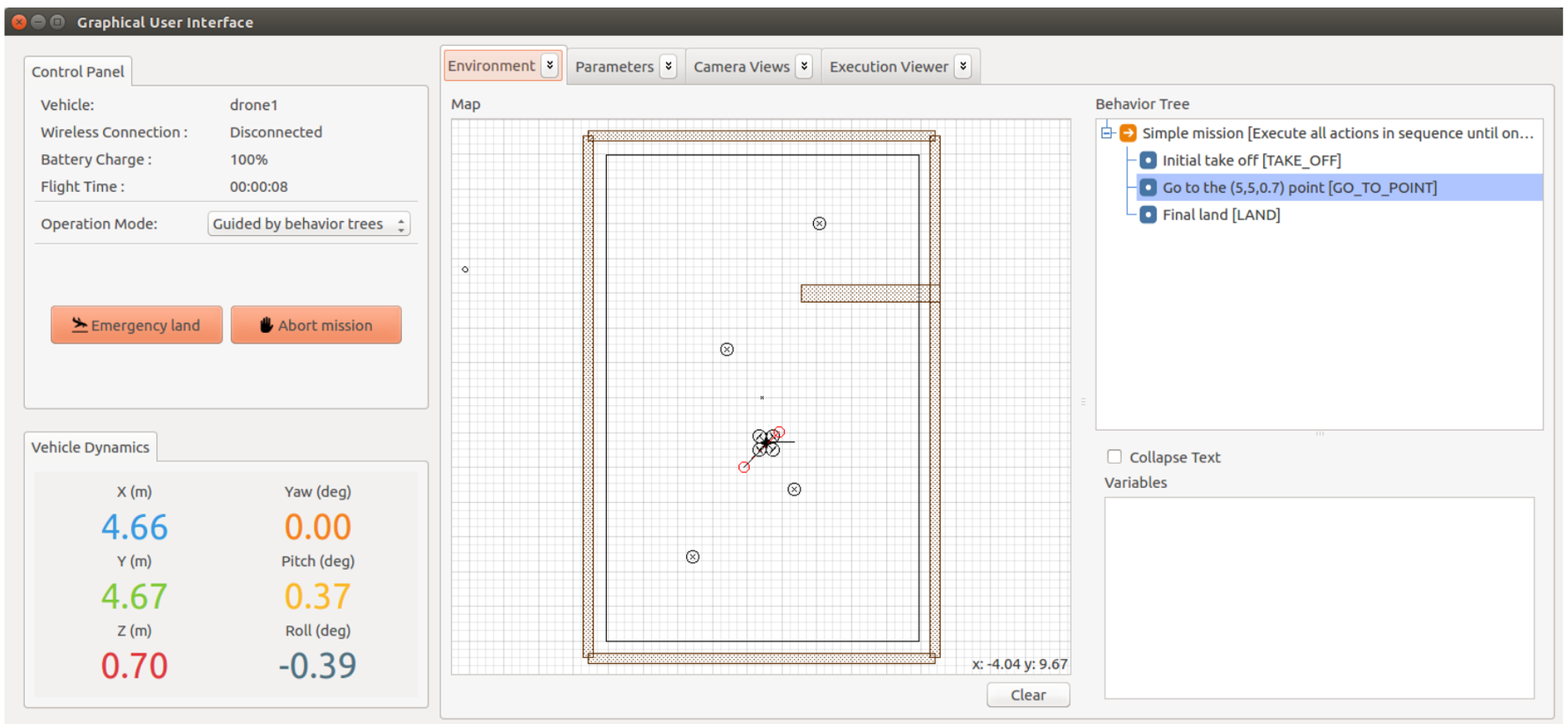
Clone this wiki locally

- *Software level*. This corresponds to the internal behavior of the vehicle, related to the execution of running processes and internal error messages.

The GUI presents information at the different levels. This requires the corresponding translation procedures and specific data structures. For example, the GUI keeps a relation between each capability (at mission level) and the corresponding set of running processes (at software level). Operators who are not familiar with software architectures can consult information at mission level, which is more abstract, avoiding specific details of software execution. On the other hand, operators who are interested in software debugging and maintenance can use the software level to get more details about the mission execution.

Windows layout

The figure below shows a sample screen of the user interface with the windows layout with main parts: the control panel on the upper left side of the image, the dynamics viewer on the lower left side, the windows for detailed content on the right hand side of the screen with different tabs (environment, parameters, the camera viewer or the execution viewer). At the top, there are drop-down menus to perform additional tasks related to standard procedures such as load and save files, help about user commands, etc.



The control panel

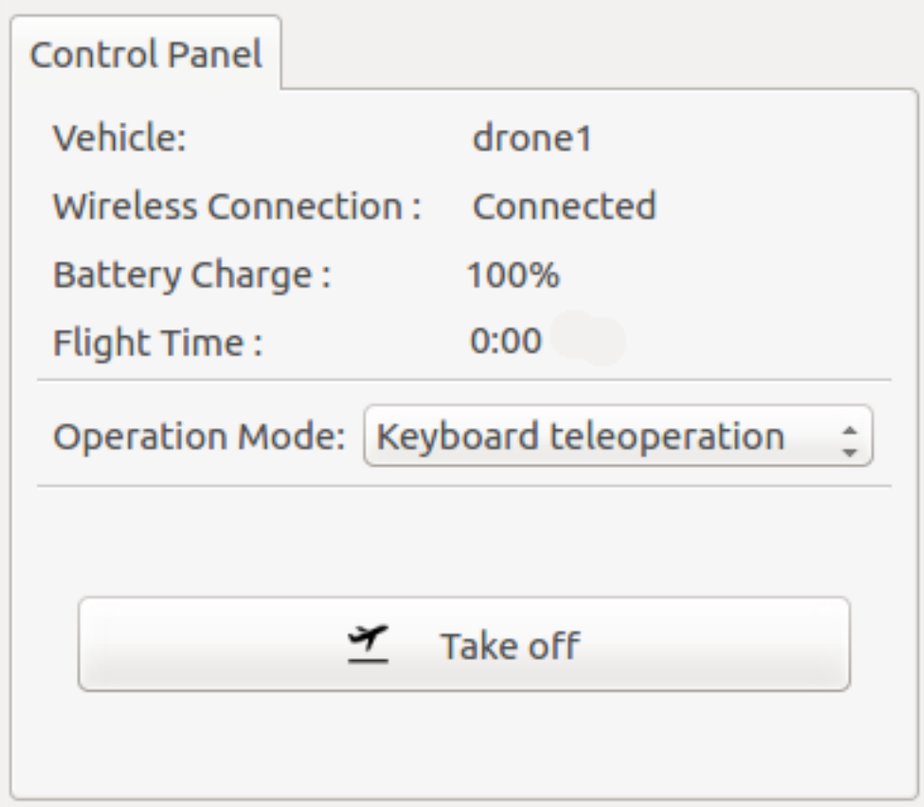
The control panel shows the general state of the UAV and the main control commands. This panel provides a summary of information considered critical that needs to be permanently present on the screen to be easily available to the operator.

The control panel is divided into three main areas: general state of the vehicle, operation mode and operation commands. The general state of the vehicle shows information such as wireless connection (connected, disconnected), battery charge (percentage of remaining battery charge) and the flight time (total time the vehicle has been flying).

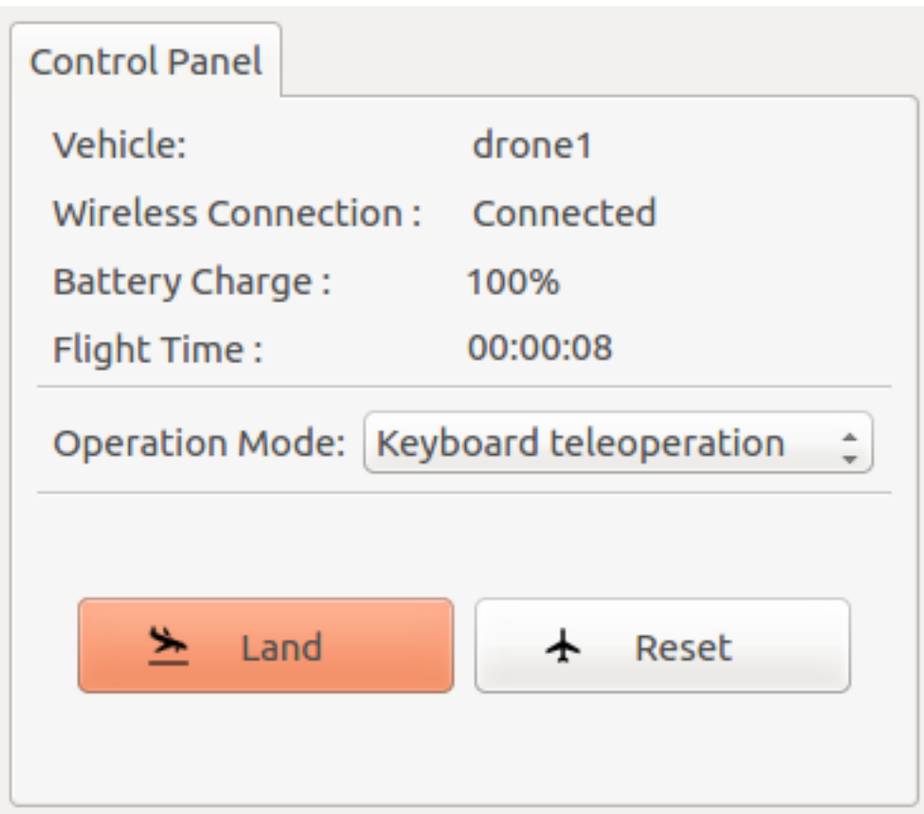
In the central area, the operator can change the operation mode: keyboard teleoperation (the user controls the movements using the keyboard), guided by Python mission (a mission file written in Python specifies the mission to be followed), guided by TML mission (a mission file written in TML language specifies the mission to be followed), and guided by behavior trees (a behavior tree created by the user specifies the mission the drone will follow).

Keyboard teleoperation

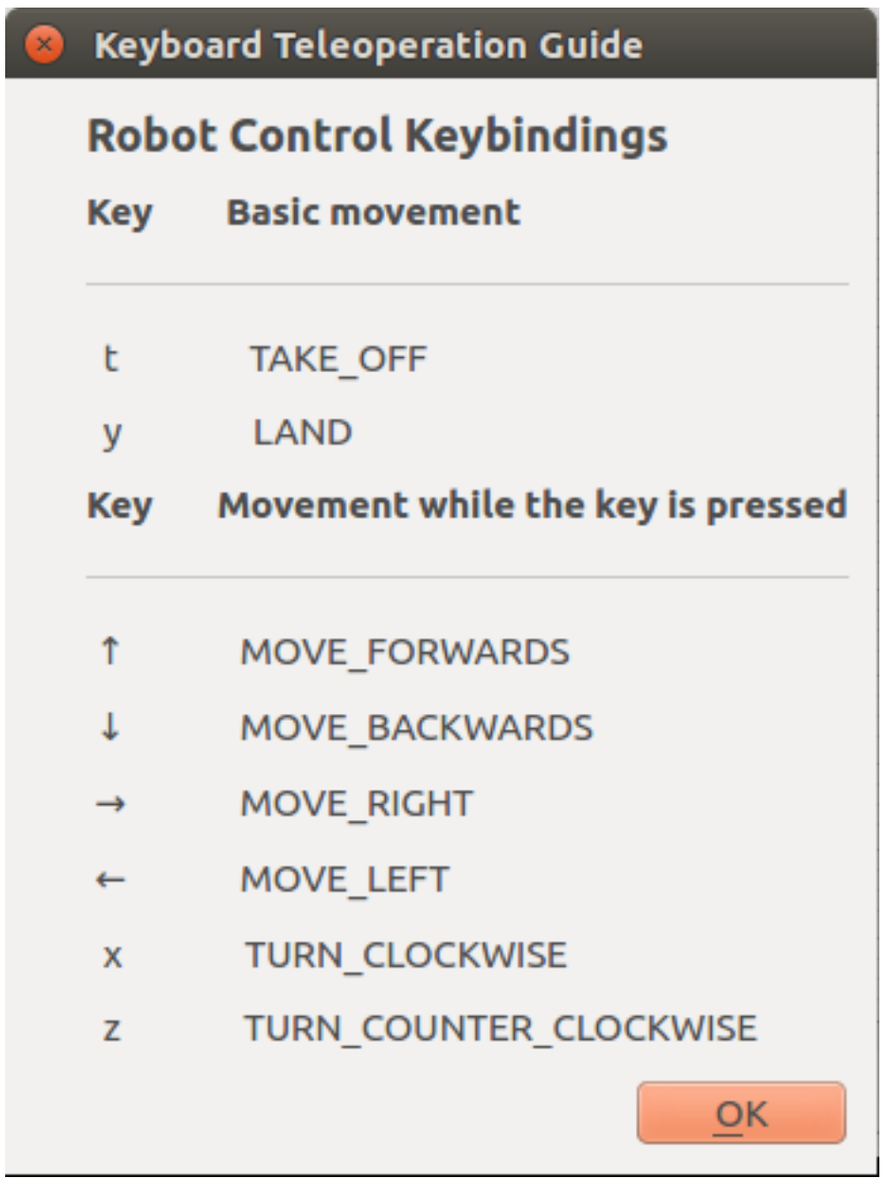
When the operation mode is set with the value keyboard teleoperation, the operator can control the movements of the robot with the keyboard. In the control panel, the available operation command is take off.



After pressing the take off button, the robot takes off and the layout changes showing two commands: land and reset (reset the vehicle position with zero values for rotation angles).

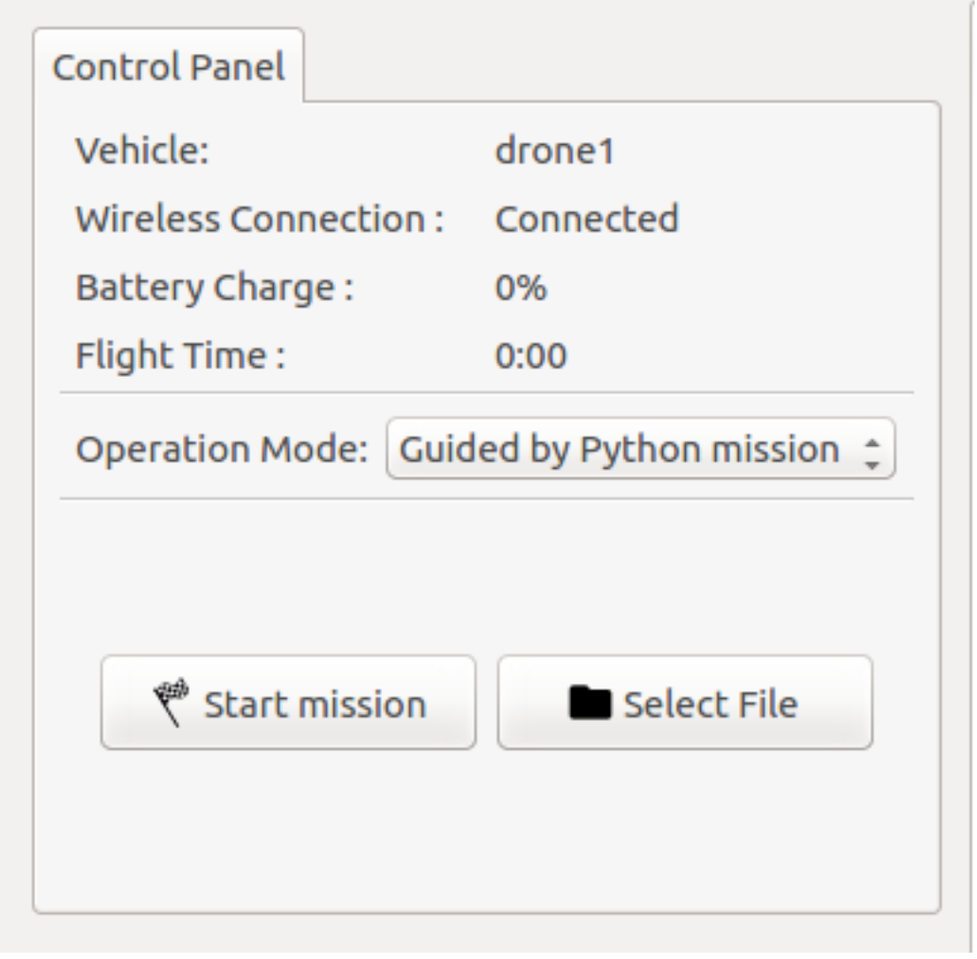


Besides these actions, the operator can control the movements using the following keys (this guide can be shown by clicking at the top of the screen, in the help menu):



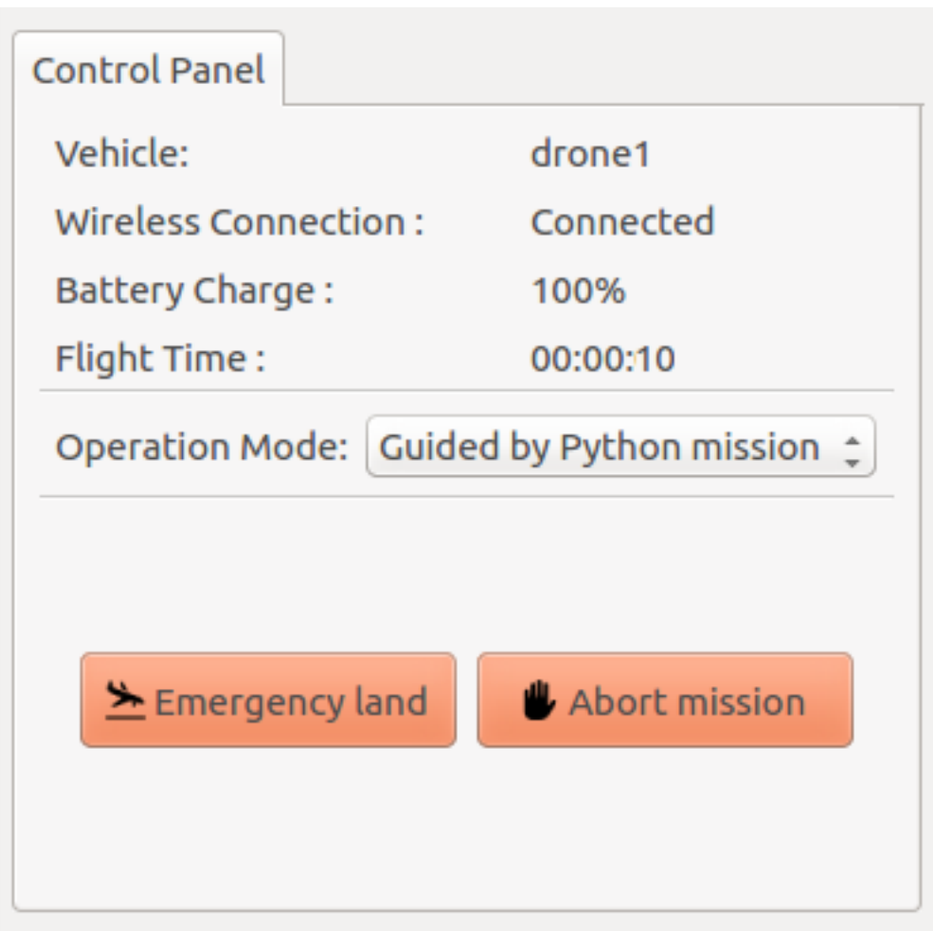
Operation guided by Python mission

If the operation mode is guided by a Python mission, the area for operation commands shows a button to start the mission and a button to select the mission file.



The operator can select a text file with the mission plan specified in Python Language (consult the previous link to know how to specify a mission plan in this language).

Once the operator starts the mission, the control panel shows the available commands. The operator can choose to perform an emergency land by pressing the "Emergency land" button or to stop the execution of the mission by pressing the "Abort mission" button but keep in mind this action will not land the robot. Pressing either one of these buttons will allow the user to select another Python mission file or to re-start the current mission.



Operation guided by TML mission

The operator can use the TML language to guide the mission execution. TML is a mission specification language designed specifically for Aerostack using a task-based approach. See the following link to know how to specify a mission in this language:

- TML Language

Follow these steps to specify the tasks of a mission:

- The task-based mission of the robot is specified in a XML file called `mission_specification_file.xml` (this is the default name, as it is established in the .launch file located within the `launch` directory in the repository `task_based_mission_planner_process`). This file must be located in `$AEROSTACK_STACK/configs/` within a folder selected by the operator called `drone{Id}` (`{Id}` = 00, 01, 02, ...).

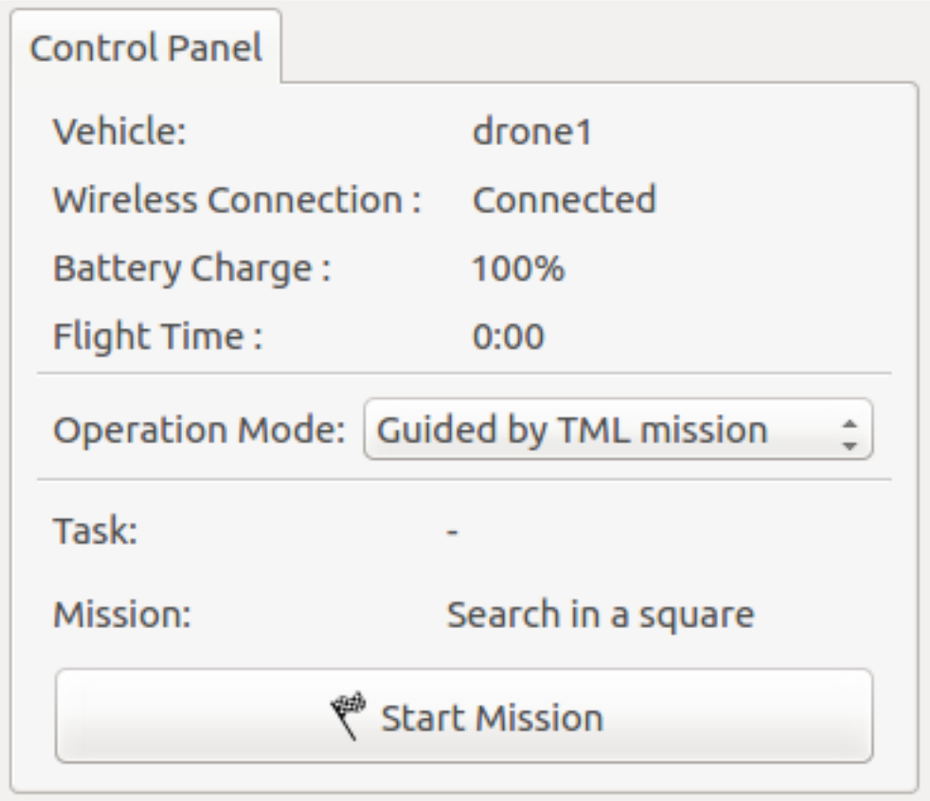
- To check if the file has syntax errors, launch the mission planner executing the following command:

$ roslaunch task_based_mission_planner_process task_based_mission_planner_process.launch

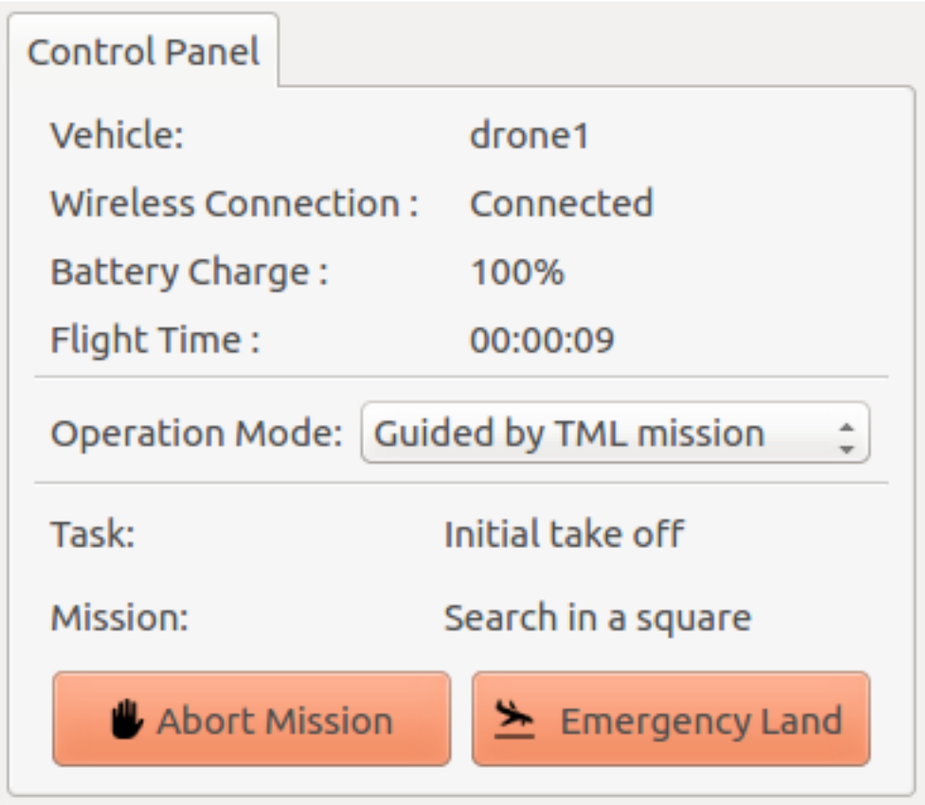
- If the xml file does not have any errors, the following message is presented:

  File loaded without errors! Waiting to start ...

To use the TML language for a mission, the operator sets the operation mode in the control panel to the value guided by TML mission. The TML mission can only be edited and loaded via the mission_specification_file.xml located in the correct '$AEROSTACK/configs' folder.

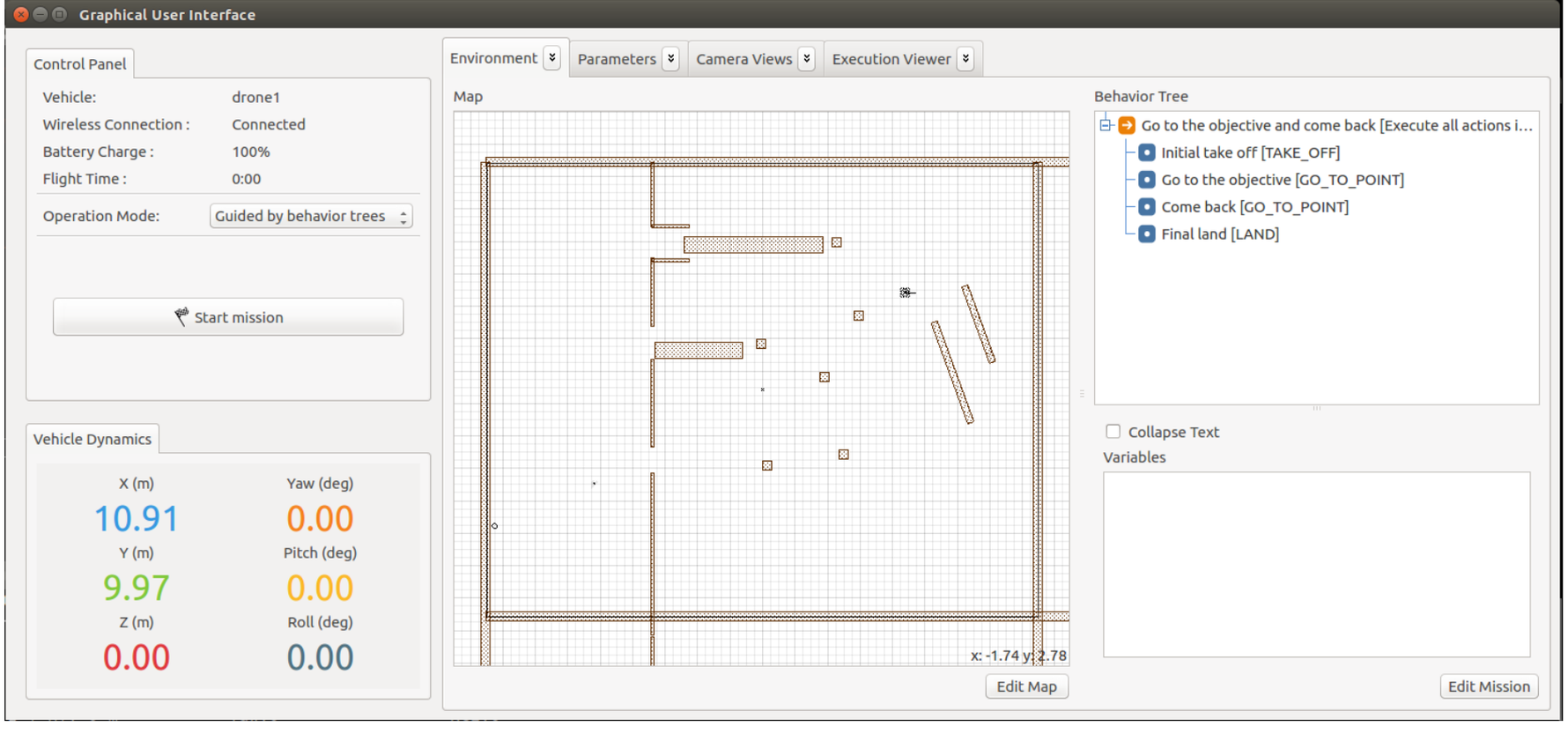


When the mission is started, control panel shows the current task being executed and the name of the mission. The control panel also presents buttons to abort the mission or execute an emergency land:

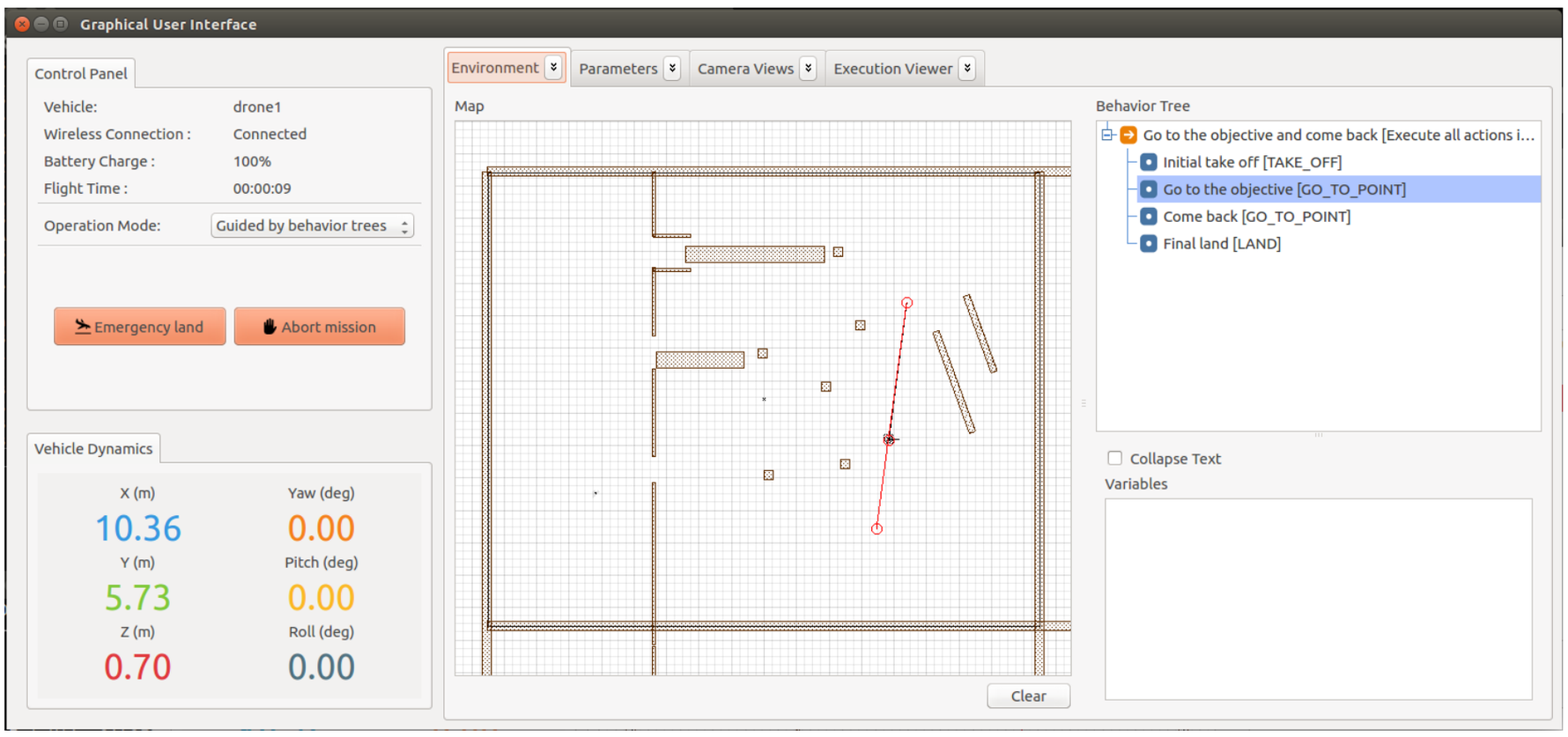


Operation guided by behavior trees

If the operation mode is guided by behavior trees, the area for operation commands shows a button to start the mission.



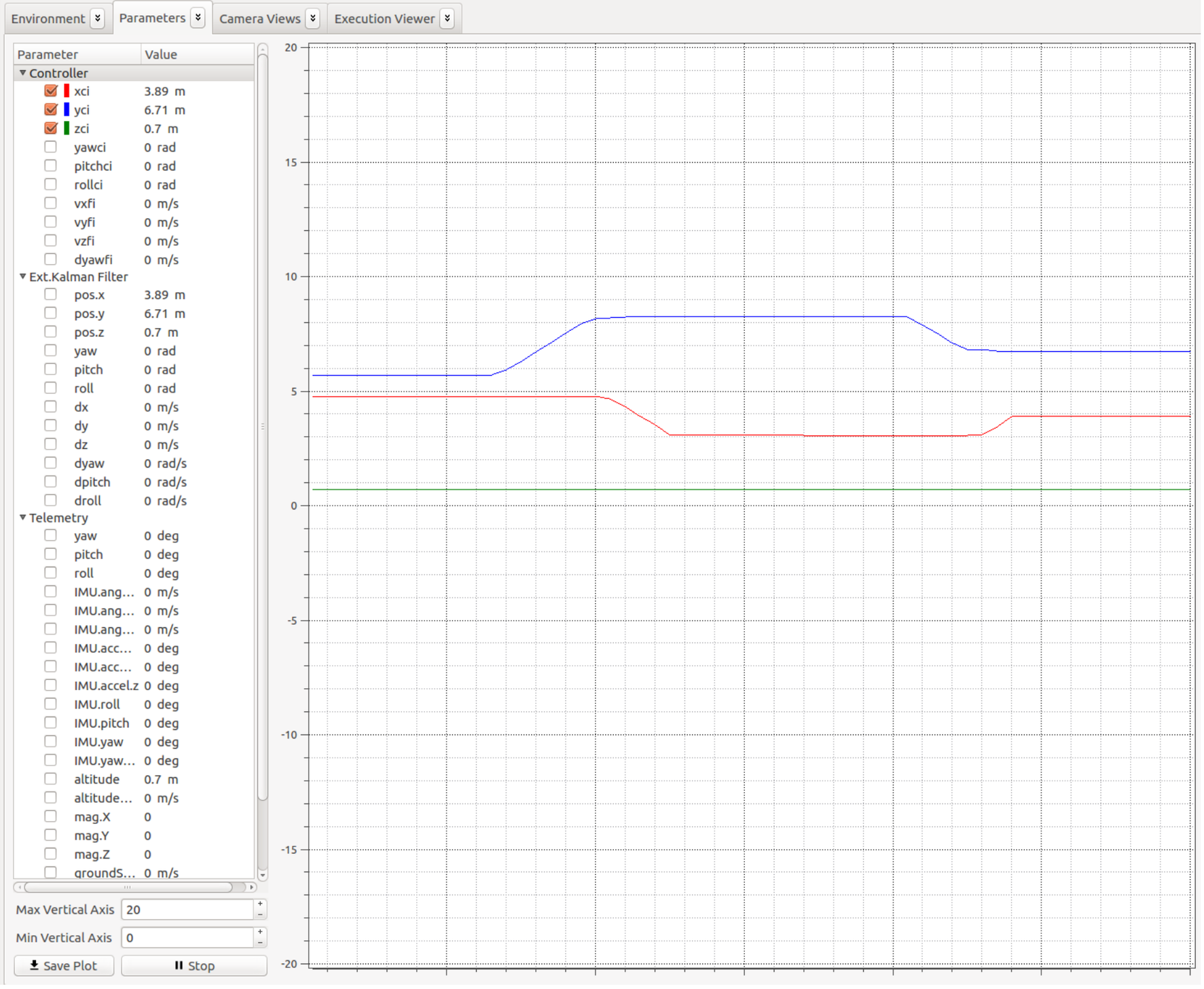
When this button is pressed, it will change itself to a "Emergency land" button which will land the robot and stop the execution of the Behavior Tree and a "Abort mission" button which will end the execution and show the "Start mission" button again.



The operator can specify a mission as a Behavior Tree (consult the previous link to know how to specify a mission plan with a behavior tree).

The parameter viewer

In order to monitor the vehicle's behavior in detail during a mission, the operator can observe the values of numerical parameters. The operator can display multiple plots of the parameters selected in real time to analyze and compare their values.



This type of view is useful to verify the right operation of certain quantitative sensors. The view is also useful during the development of a particular vehicle configuration to select appropriate components and algorithms. For example, this window can be used to analyze the quality of different perception algorithms that estimate the vehicle's position.

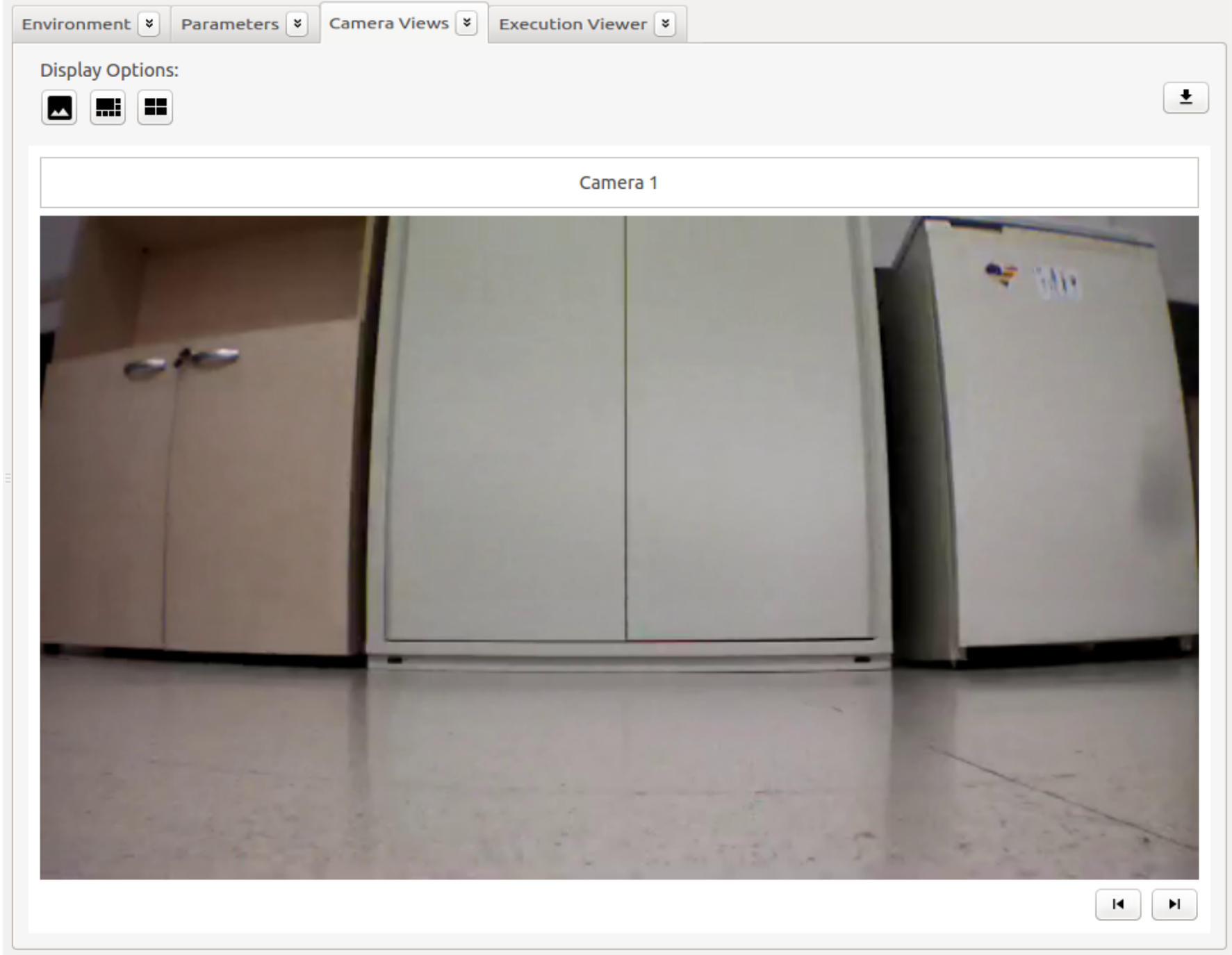
The window displays a plot for each parameter, identified by distinguishable colors. The graphic shows the evolution in a determined time interval, moving as time goes by from right to left. The vertical axis has been designed in a configurable way due to diversity of the parameters and the range of values. The maximum value can be set by a selector or directly zooming on the canvas.

As an additional feature, the window gives the possibility of saving the graphic plots for later use in various file formats (e.g., pdf, postscript).

The camera viewer

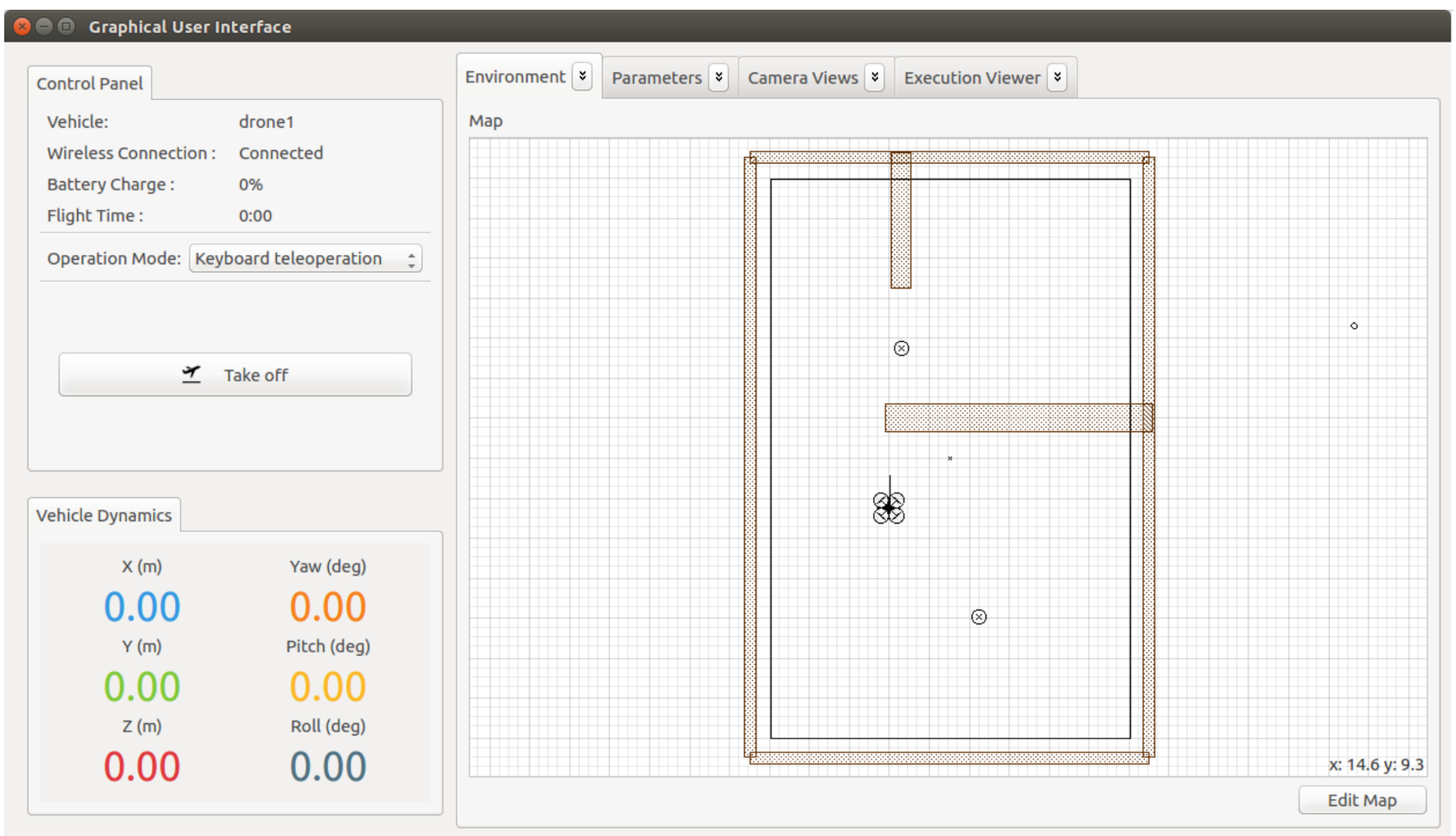
The camera viewer shows pictures and/or video images captured by the aerial vehicle during flight. The camera viewer displays a video streaming of the camera images.

Since the vehicle can use several cameras, the operator can select different display options to visualize the images. For example, the operator can select an option that displays only one camera (e.g., the frontal camera), an option that displays all cameras at the same time, or an option that displays a selected group of cameras.



The environment viewer

The environment viewer provides a 2D graphical representation of the environment. It gives dynamic information about the position of the robot and the obstacles as well.



The operator can edit the environment map by pressing the 'Edit' button. To consult more details about how to edit an environment map, please read the following page:

- How to create an environment map

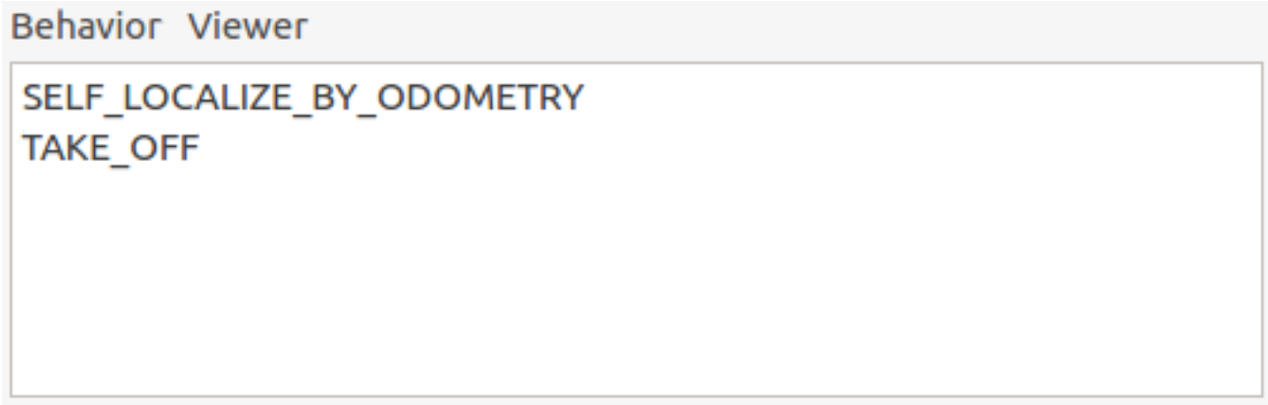
This window may also show information about behavior trees. This is presented when the operation mode (in the control panel) is set with the value 'guided by behavior trees'.

The execution viewer

The execution viewer shows the current state of the execution in two windows:

- The behavior viewer shows the list of active behaviors. The list is updated each time a behavior is started or stopped.

- The belief viewer shows the content of the memory of beliefs. This list is updated each time a belief is added, modified or removed.



Contact

We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page.

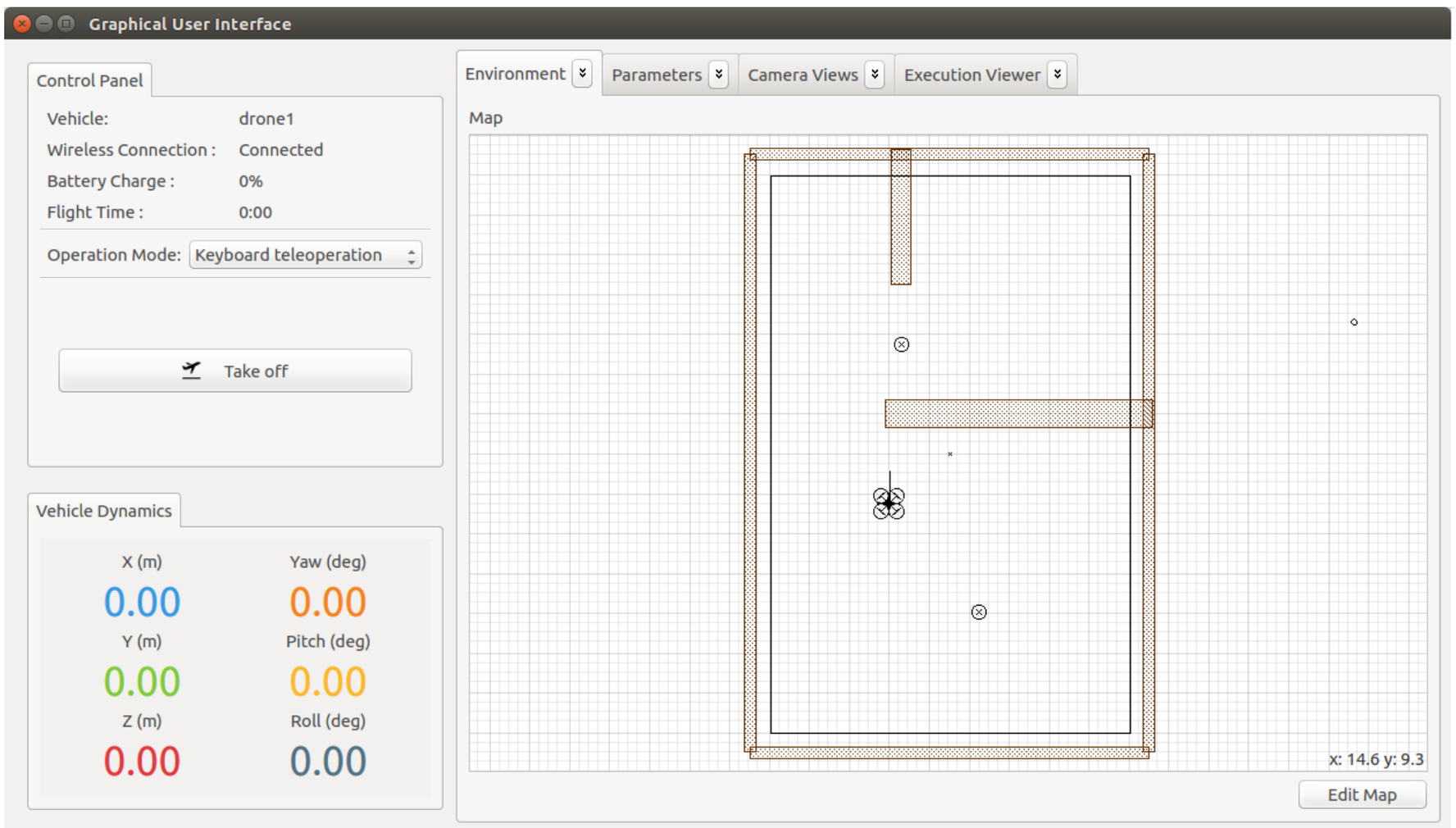
You can also contact the team support at the following address: aerostack.vision4uav@gmail.com

How to Create an Environment Map

Edit New Page

Martin Molina edited this page on 27 Oct 2017 · 5 revisions

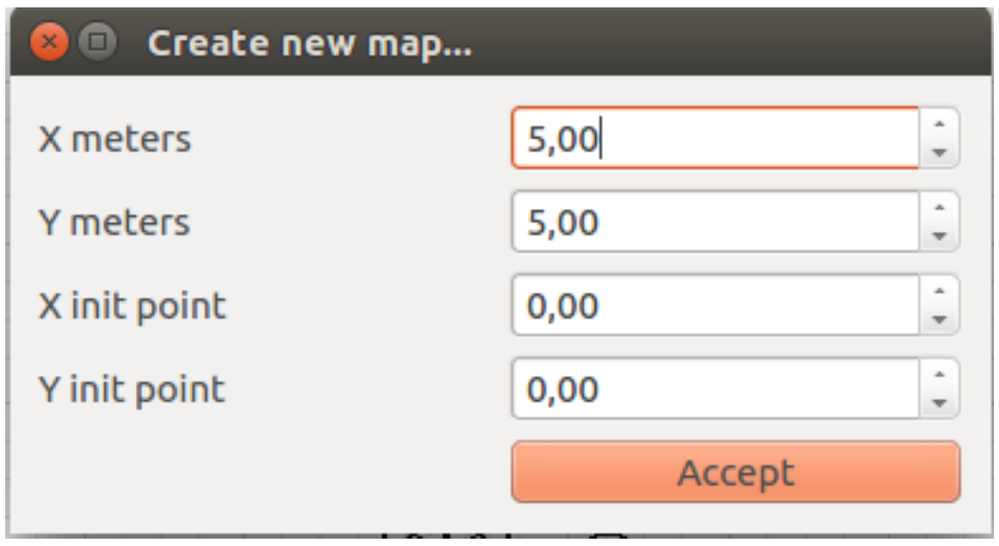
This page describes how the operator can create the environment map for a mission by using a graphical tool. This tool allows the operator to set the size of the environment map and to place the static physical objects and the visual markers used to the self-localization of the vehicle. In addition this tool allows the operator to import or export a map using external files and to load or save the current map to internal config files used by Aerostack.



To edit the current map or to create a new one, press the 'Edit Map' button in the 'Environment' tab at the lower right corner.

Creating a new map

To start the creation of a new map in the environment viewer widget, you have to press the 'New' button. A window dialog will be displayed, allowing the user to set the X size and the Y size of the map and its starting point. Press accept and the new map will be created. Four walls will be created to delimit the map zone.



Creating new objects

A set of tools were created to add a new object to the mission environment. To create a new object, just select the tool with the required object and right-click in the desired position. This will create an object of the specified type, that can be modified with the edition tool.

Pages 37

Introduction

- What is Aerostack?
- Getting Started
- License
- List of distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Catalog of Behaviors
- Memory of Beliefs

Mission Plan Specification

- TML Language
- Behavior tree
- Python Language

Tutorials [Beginners]

- Prepare the Mission:
  - Configure Map
  - Specify Mission Tasks
  - Configure Launcher
- Execute the Mission:
  - Launch Real Flight
  - Practical Tips for Real Flights
- Mission examples:
  - Simulated Quadrotor
  - AR Drone and Visual Markers

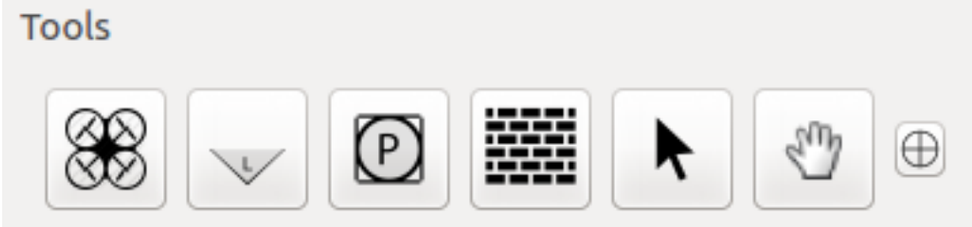
Tutorials [Advanced]

- How to program a new behavior
- Setup the Communication Network
- Wi Fi Communication with Multiple Drones
- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

More information

- Publications
- Vision & Aerial Robotics:
  - First prize at IMAV 2013
  - Stack pre-release 2013

Clone this wiki locally



https://github.com/Vision4U
Clone in Desktop

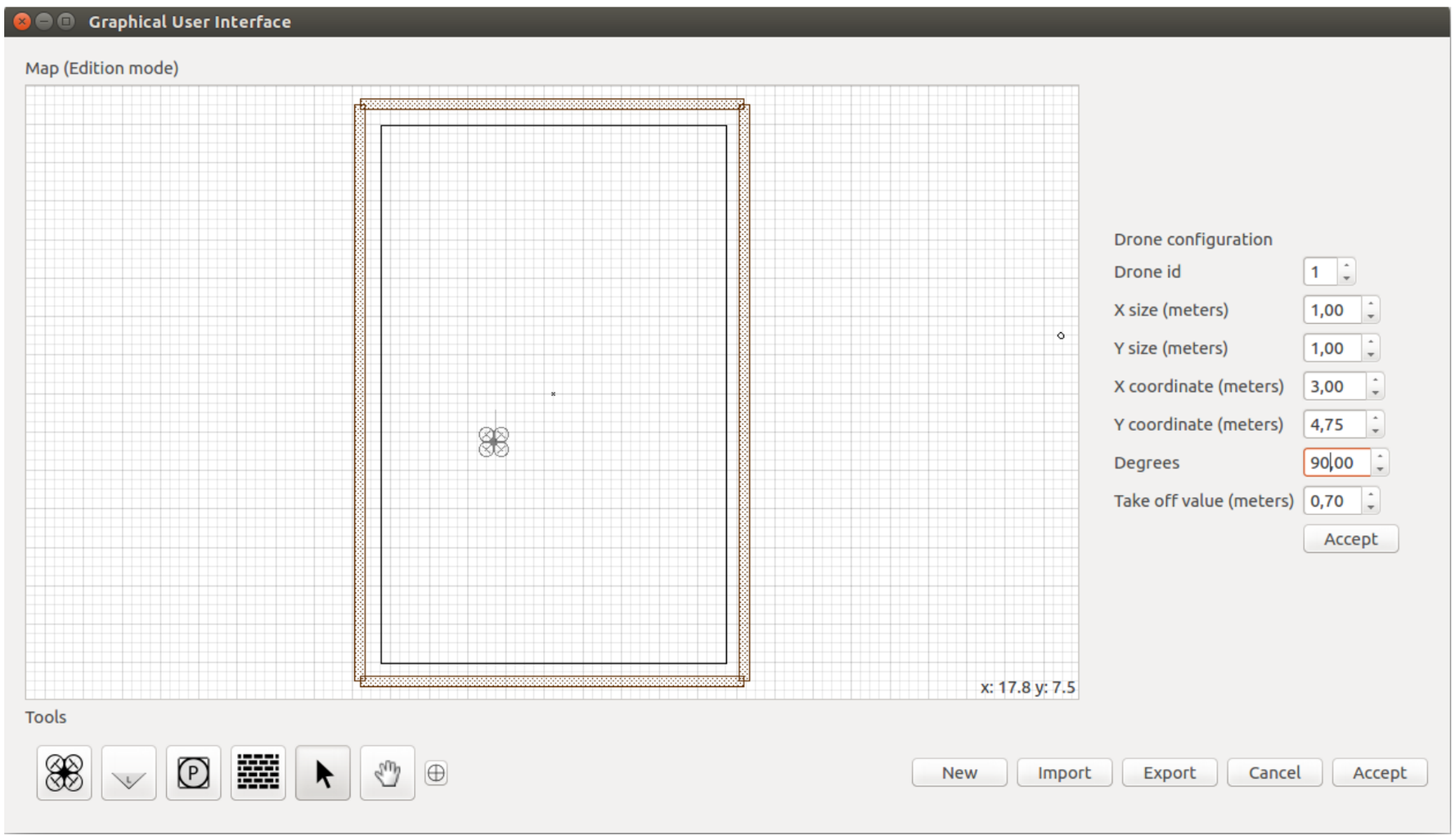
Editing an existing object

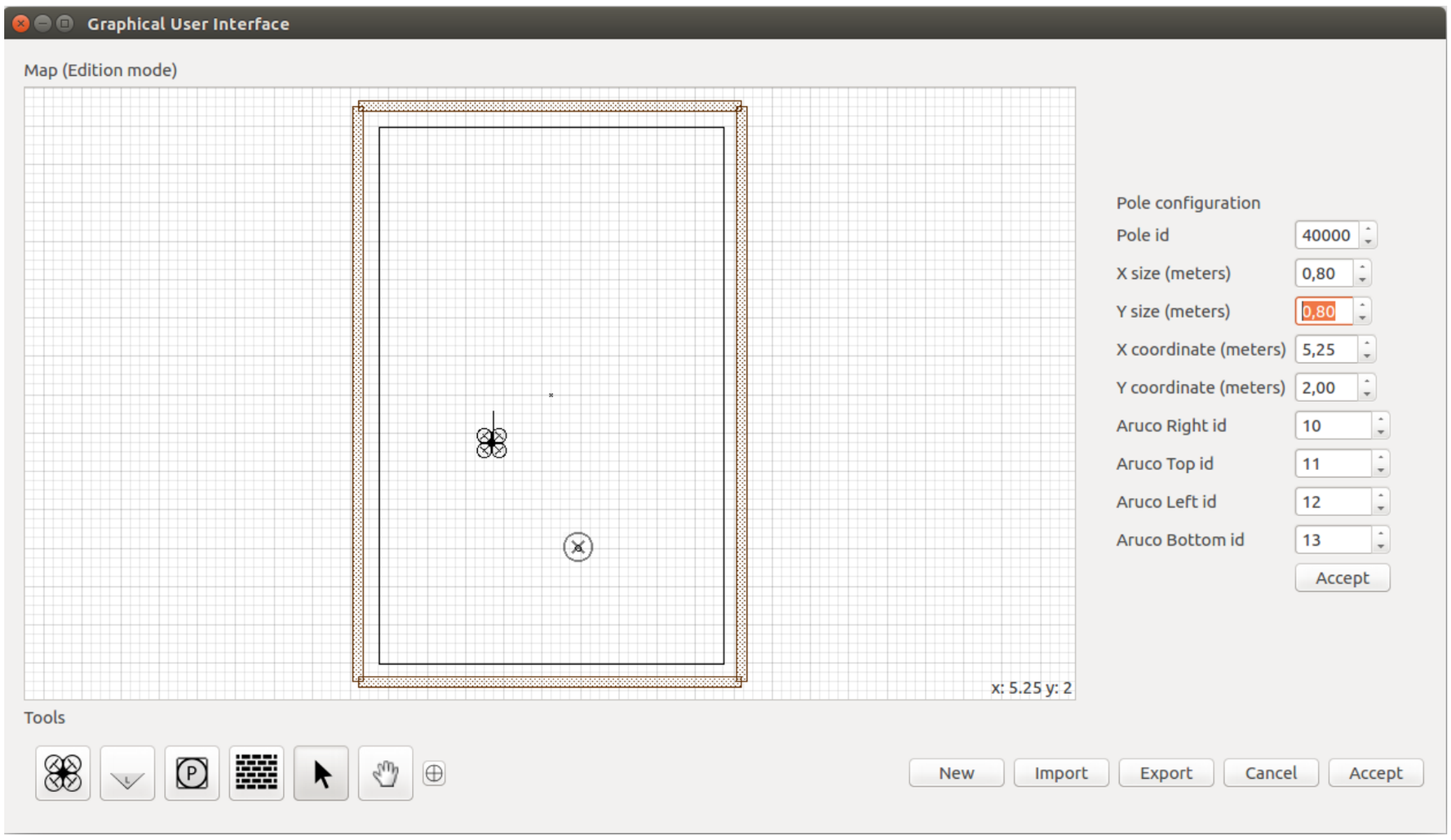
To edit an existing object, just use the selection tool and press the desired object in the map. That will display a dialog in the right side of the window allowing the user to modify the selected object. To delete an object, just select it and press the 'supr' key.

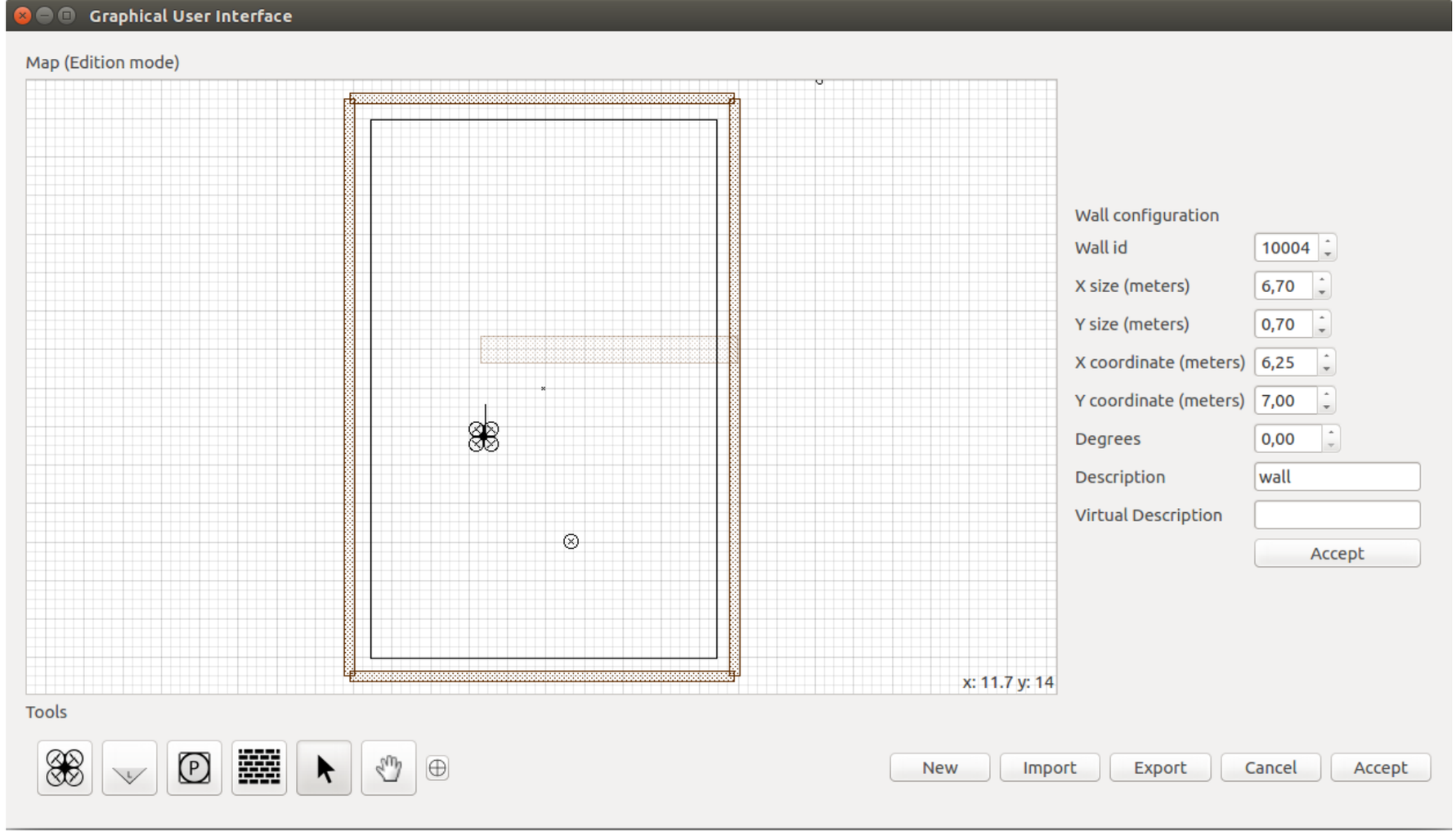
Additionally, the user can drag the objects and drop them wherever he desires to move them.

In order to improve the precision when selecting, creating or moving an object, you can use the precision tool or press 'shift' in order to adapt the selection pointer to the grid.

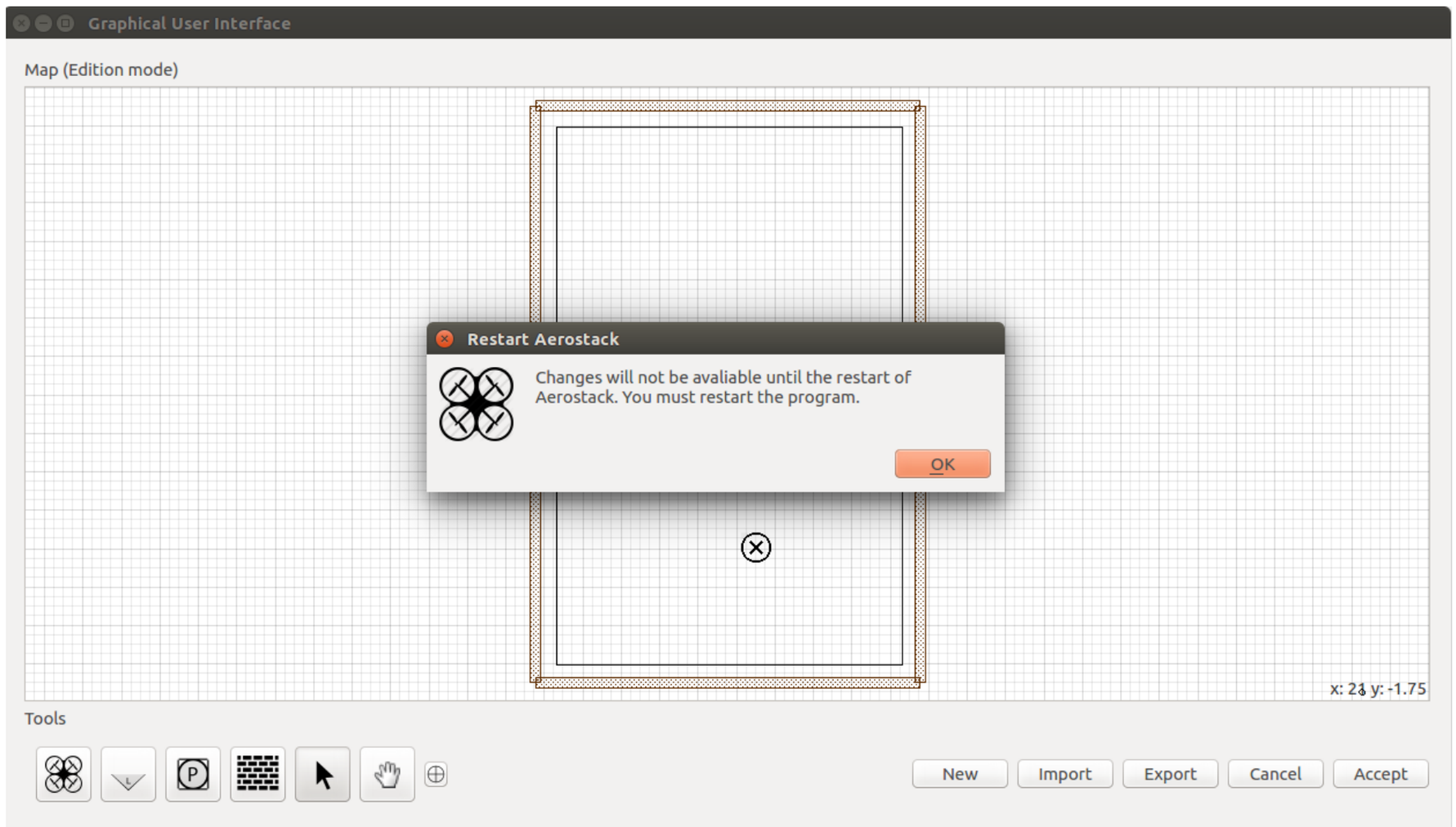
Here's a couple of screenshots that show how the addition and customization of a drone, a pole and a wall is done. Keep in mind that each object's ID number must all be different. If a pair of ID's are the same number the operator will be asked to check them and set them up properly.



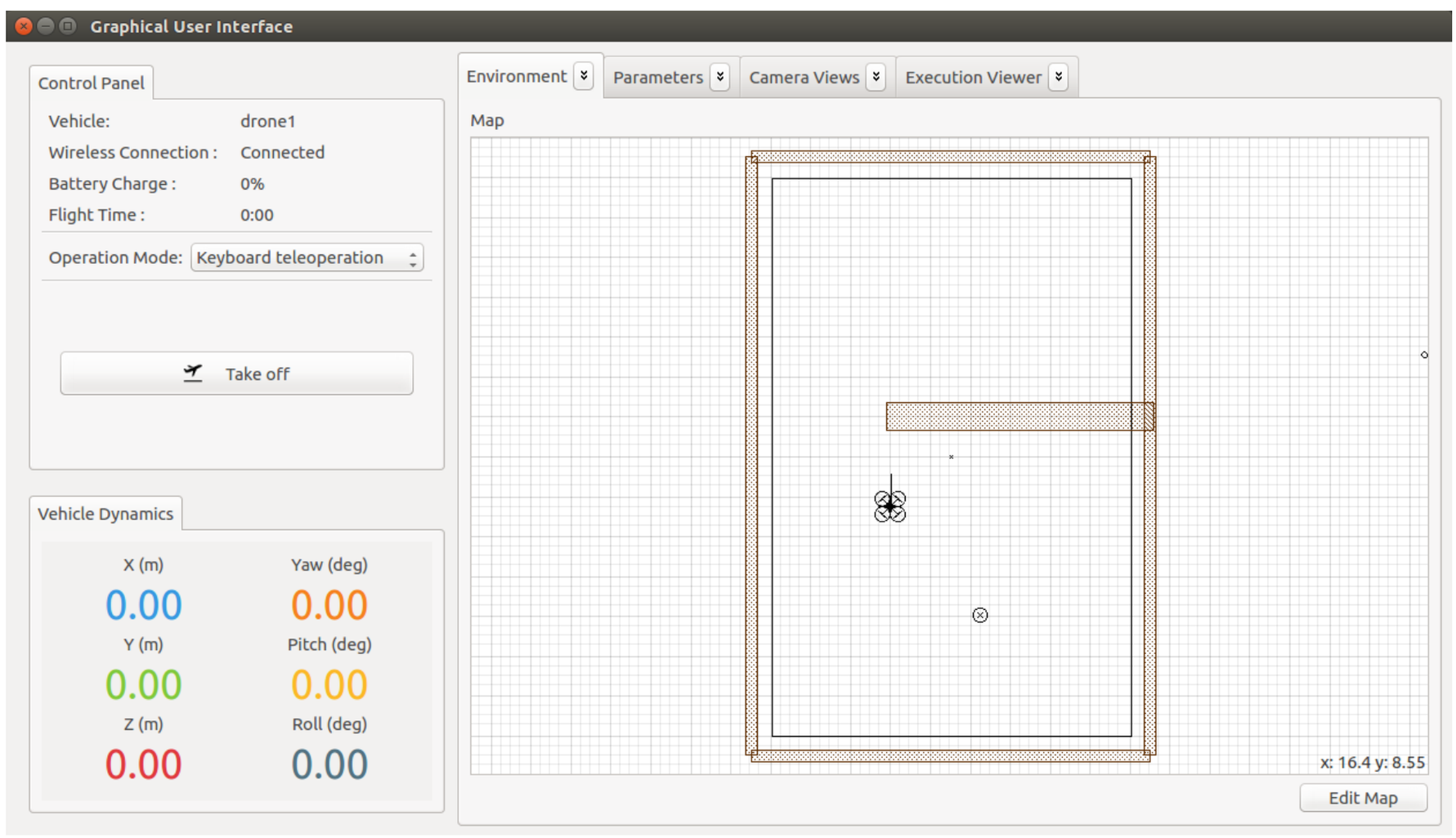




When you're finished editing your new Environment Map you can click the 'Accept' button. This will prompt a window as shown in the next picture that explains to you the new map wont be available until Aerostack is restarted.



Whenever you choose to restart Aerostack the latest map exported to the Aerostack configuration folder will be displayed.



File management

The map creation tool allows the user to save a file with all the current information of the map by pressing the 'Export' button. Likewise, the 'Import' button allows the user to select a file created by the tool in order to load a map created earlier.

To use the map in a mission, just press the button 'Accept', and the configuration files will be overwritten. In addition, the program will ask the user to restart Aerostack to make the changes available to use.

Example

Here you can download a video showing an example of map environment configuration:

- Example video

Contact

We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page.

You can also contact the team support at the following address: aerostack.vision4uav@gmail.com

Catalog of Behaviors

Martin Molina edited this page on 8 Nov 2017 · 6 revisions

Edit New Page

This catalog shows the available behaviors provided by Aerostack. They are presented according to the following categories: motion behaviors, and behaviors for environment understanding.

It is important to note that there are two types of behaviors:

- **Goal-based behaviors**. These behaviors are defined to reach a final state (attain a goal). For example, the behavior TAKE_OFF ends when the drone has reached a certain height.
- **Recurrent behaviors**. These behaviors perform an activity recurrently or maintain a desired state (for example, the behavior PAY_ATTENTION_TO_VISUAL_MARKERS).

You can consult the following link to know how to add your own behaviors:

- How to program a new behavior

Motion behaviors

Behavior `TAKE_OFF`

The robot takes off vertically from the surface to the normal altitude. If the altitude argument is not given, the robot reaches a default altitude. **Type:** goal-based behavior.

Behavior `LAND`

The robot lands vertically in the current position. This behavior assumes that the floor is static. **Type:** goal-based behavior.

Behavior `KEEP_HOVERING`

The robot keeps hovering. Hovering is a maneuver in which the robot is maintained in nearly motionless flight over a reference point at a constant altitude and on a constant heading. This behavior does not avoid moving obstacles. **Type:** recurrent behavior.

Behavior `KEEP_MOVING`

The robot keeps moving at a constant speed in some direction (forward, backward, left,right). If the speed value is not given a default value is considered. This behavior does not avoid obstacles. **Type:** recurrent behavior.

Arguments	Format	Example
direction	{FORWARD, BACKWARD, LEFT, RIGHT}	direction: FORWARD
speed	number (m/sec)	speed: 12

Behavior `ROTATE`

The robot rotates some degrees in a certain axis (yaw). **Type:** goal-based behavior.

Arguments	Format	Example
angle	number (degrees)	angle: 90
relative_angle	number (degrees)	relative_angle: 90

Behavior `WAIT`

Pages 37

Introduction

- What is Aerostack?
- Getting Started
- License
- List of distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Catalog of Behaviors
- Memory of Beliefs

Mission Plan Specification

- TML Language
- Behavior tree
- Python Language

Tutorials [Beginners]

- Prepare the Mission:
  - Configure Map
  - Specify Mission Tasks
  - Configure Launcher
- Execute the Mission:
  - Launch Real Flight
  - Practical Tips for Real Flights
- Mission examples:
  - Simulated Quadrotor
  - AR Drone and Visual Markers

Tutorials [Advanced]

- How to program a new behavior
- Setup the Communication Network
- Wi Fi Communication with Multiple Drones
- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

More information

- Publications
- Vision & Aerial Robotics:
  - First prize at IMAV 2013
  - Stack pre-release 2013

Clone this wiki locally

The robot keeps doing the action while waiting for timeout or seeing a certain visual marker or both. **Type:** goal-based behavior.

Arguments	Format	Example
timeout	number(seconds)	timeout: 30
until_observed_visual_marker	number (visual marker id)	until_observed_visual_marker: 1

Behavior `FOLLOW_OBJECT_IMAGE`

The robot follows the image of a moving object, keeping a certain constant distance between the robot and the object. The operator selects the image to follow through the user interface. **Type:** recurrent behavior.

Action `GO_TO_POINT`

The vehicle goes to a given poin avoiding obstacles. The trajectory follows horizontal movements in two dimensions (2D). The point can be expressed using absolute coordinates (argument “coordinates”) or coordinates relative to the vehicle (argument “relative coordinates”). **Type:** goal-based behavior.

Arguments	Format	Example
coordinates	Tuple of 3 numbers x, y, z (meters)	coordinates: [1.23, 2, 0.5]
relative_coordinates	Tuple of 3 numbers x, y, z (meters)	relative_coordinates: [1.23, 2, 0.5]

Behaviors for environment understanding

Behavior `PAY_ATTENTION_TO_VISUAL_MARKERS`

The robot pays attention to visual markers. **Type:** recurrent behavior.

Behavior `SELF_LOCALIZE_BY_ODOMETRY`

The robot self localizes using odometry. **Type:** recurrent behavior.

Behavior `SELF_LOCALIZE_BY_VISUAL_MARKERS`

The robot self localizes using visual markers (ArUco markers). **Type:** recurrent behavior.

Behavior `VERIFY`

The robot checks if a condition is satisfied. The result is true if the condition is satisfied. Otherwise, the result is false. **Type:** deliberative behavior.

Arguments	Format	Example	Comments
marker_is_observed	integer	marker_is_observed: 3	
hovering	Boolean (TRUE/FALSE)	hovering: TRUE	Not implemented yet
obstacle_distance	number	obstacle_distance: 0.5	Not implemented yet
voice_command	text	voice_command: "land"	Not implemented yet

https://github.com/Vision4U
Clone in Desktop

Contact

We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page.

You can also contact the team support at the following address: aerostack.vision4uav@gmail.com

Memory of Beliefs

Edit New Page

Martin Molina edited this page on 3 Nov 2017 · 1 revision

Aerostack uses a memory of symbolic beliefs to represent information about what the robot believes about the world. A belief is a proposition about the world that the robot thinks is true (the world here refers to both the external world and the internal state of the robot).

Beliefs are useful to be used in the specification of mission plans. During the execution of a mission plan, the robot uses beliefs as true facts while it reasons trying to adapt the mission plan to the current circumstances. Beliefs can also be used by the operator to help monitor the correct execution of a mission.

Aerostack stores the set of beliefs and tries to keep their consistency with the world and with the other beliefs. Aerostack represents beliefs using a logic-based approach with predicates. Next table shows examples of such predicates with the general format of predicate(object, value) or simpler forms such as property(object). The examples presented in this table are for illustration purposes (i.e., only some of these predicates have been implemented in the current version of Aerostack).

Predicate	Description
object(x, y)	The object x is an instance of the class y
position(x, y)	The object x is at the position y
name(x, y)	The name of object x is y
flight_state(x, y)	The aerial robot x is flight state y (e.g., landed or flying)
code(x, y)	The numerical code of x is y
color(x, y)	The color of x is y
frequency(communication,x)	The frequency of the communications is x
charge(battery, x)	The charge of battery is x
carry(self, x)	The own aerial robot carries the object x
image(x, y)	The image of object x is y
temperature(air, x)	The temperature of the air is x
visible(x)	The object x is visible
stability(ground, x)	The stability of the ground is x

To keep the consistency, the representation follows an object-oriented approach. Objects are instances of a class and may have attributes with values. Aerostack assumes that the values of attributes defined for an object using triplets are mutually exclusive. For example, the belief charge(battery, empty) is incompatible with the belief charge(battery, full) because the values empty and full are mutually exclusive. Aerostack keeps consistency among beliefs. When a belief is added, e.g., charge(battery, empty), the incompatible beliefs are automatically retracted, e.g., charge(battery, full).

The operator can consult the state of the memory using belief expressions (e.g., within languages for mission plan specification). For example the user can use the following belief expression to know the coordinates of the current position:

```
position(self,(?X,?Y,?Z))
```


Pages 37

Introduction

- What is Aerostack?
- Getting Started
- License
- List of distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Catalog of Behaviors
- Memory of Beliefs

Mission Plan Specification

- TML Language
- Behavior tree
- Python Language

Tutorials [Beginners]

- Prepare the Mission:
  - Configure Map
  - Specify Mission Tasks
  - Configure Launcher
- Execute the Mission:
  - Launch Real Flight
  - Practical Tips for Real Flights
- Mission examples:
  - Simulated Quadrotor
  - AR Drone and Visual Markers

Tutorials [Advanced]

- How to program a new behavior
- Setup the Communication Network
- Wi Fi Communication with Multiple Drones
- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

More information

- Publications
- Vision & Aerial Robotics:
  - First prize at IMAV 2013
  - Stack pre-release 2013

Clone this wiki locally

In this belief expression, X, Y, and Z are variables that store the coordinates corresponding to the current position. The question mark (?) indicates that such variables are substituted by the values found in the belief memory.

https://github.com/Vision4U

Clone in Desktop

Contact

We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page.

You can also contact the team support at the following address: aerostack.vision4uav@gmail.com

TML Language

Edit New Page

Martin Molina edited this page a day ago · 8 revisions

The operator can specify the tasks of a mission using the TML language (task-based mission specification language) specifically designed for Aerostack. TML uses XML syntax.

The following sections describe how to write in TML language a mission specification. As a complete example, a mission specification file is provided at the end of this document.

1. Mission

The mission is defined with the tag `<mission>` and the attribute `root_task`. The `root_task` attribute indicates which task tree should be executed when the mission starts. For example:

```
<mission name="example task">
</mission>
```

The body of the mission is separated in two main parts: (1) task tree definitions and (2) event handlers.

2. Task Trees

The first part of the mission specification includes one or more task trees with the tag `<task_tree>`. Each task tree contains a series of tasks organized in a tree, and can be identified by the name of its root task. Each task is specified using the tag `<task>` and the attribute `name`. For example:

```
<task_tree>
  <task name= "example root task">
    <task name="example child task">
      <task name="example leaf task 1">
      </task>
    </task>
    <task name="example leaf task 2">
    </task>
  </task>
</task_tree>
```

The body of a task can include the tag `<behavior>` with the attribute `name` to specify an interpretable behavior (e.g., TAKE_OFF, LAND, etc.). The body of a task can also include one or several tags `<recurrent_behavior>` with the attribute `name` to specify the recurrent_behaviors to be activated at the start of the task and deactivated at the end. For example:

```
<task name="track image">
  <recurrent_behavior name="SELF_LOCALIZE_BY_VISUAL_MARKERS"/>
  <behavior name="FOLLOW_IMAGE">
      <argument name="duration" value="60"/>
  </behavior>
</task>
```

A behavior can optionally include arguments. Arguments use the tag `<argument>` and the optional attribute `name`. For example:

```
<task name="move to the first point">
  <behavior name="GO_TO_POINT">
      <argument name="coordinates" value="[3.0, 4.2, 1.5]"/>
```


Pages 37

Introduction

- What is Aerostack?
- Getting Started
- License
- List of distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Catalog of Behaviors
- Memory of Beliefs

Mission Plan Specification

- TML Language
- Behavior tree
- Python Language

Tutorials [Beginners]

- Prepare the Mission:
  - Configure Map
  - Specify Mission Tasks
  - Configure Launcher
- Execute the Mission:
  - Launch Real Flight
  - Practical Tips for Real Flights
- Mission examples:
  - Simulated Quadrotor
  - AR Drone and Visual Markers

Tutorials [Advanced]

- How to program a new behavior
- Setup the Communication Network
- Wi Fi Communication with Multiple Drones
- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

More information

- Publications
- Vision & Aerial Robotics:
  - First prize at IMAV 2013
  - Stack pre-release 2013

Clone this wiki locally

```
  </behavior>
</task>
```

Instead of a behavior, a task may contain an operation with the belief memory. These operations are: `<add>`, `<remove>` and `<consult>`. Each of these contain a `belief` argument that specifies the belief to add or remove, or the belief expression to consult. Consulting an expression is used to obtain values from the belief memory that can be later used with the operand `+`. For example:

```
<task name="go to pickup site">
  <task name="obtain target coordinates">
    <consult belief="position(object_1, (?X, ?Y, ?Z))" />
  </task>
  <task name="hover over object to pick up">
    <behavior name="GO_TO_POINT">
      <argument name="coordinates" value="[+X, +Y, 1.5]" />
    </behavior>
  </task>
</task>
```

Tasks can be organized in a hierarchy. The body of a complex task (i.e., a task that use other tasks) can include skills that are active during the task execution. The body of a complex task does not include behaviors or belief operations. Instead, the body includes one or several simpler tasks that are executed in sequence. For example:

```
<task name="move then land">
  <task name="move to the first point">
    <behavior name="GO_TO_POINT">
        <argument name="coordinates"  value="[3.0, 4.2, 1.5]"/>
    </behavior>
  </task>
  <task name="land">
    <behavior name="LAND" />
  </task>
</task>
```

3. Event handlers

The second part of a mission is the a list of event handlers. This part of the mission specification is defined with the tag `<event_handling>`:

```
<event_handling>
</event_handling>
```

Each event handler describes how to react to a particular event, while the mission planner is waiting the completion of a behavior. Events are defined with the tag `<event>` and the attribute `name`. Each event includes a `<condition>` tag, a `<call>` tag and a `<termination>` tag. The `<condition>` tag contains a `belief` attribute which specifies the belief expression that has to be true in order to activate the event. The `<call>` tag contains a `root_task` attribute with the name of the root task of the task tree to activate when the condition is met. The `<termination>` tag contains a `mode` attribute that specifies what to do when the task tree executed finishes. The termination mode can be any of: `REPEAT_TASK`, `NEXT_TASK`, `END_MISSION`, `ABORT_MISSION`.

```
<event_handling>

  <event name="search for image when drone_1 reaches its objective">
    <condition belief="objective(drone_1, ?X), position(drone_1, ?X)"/>
    <call root_task="search for image" />
    <termination mode="REPEAT_TASK"/>
  </event>
```

https://github.com/Vision4U
Clone in Desktop

```
  <event name="low battery">
    <condition belief="battery_level(self, LOW)"/>
    <call root_task="emergency land"/>
    <termination mode="ABORT_MISSION"/>
  </event>

</event_handling>
```

To know the available behaviors the robot can perform, consult the following list:

- Catalog of Behaviors

Publications

Please, note that these publications corresponds to a previous version of TML that uses "actions" and "skills" instead of "behaviors":

- M. Molina, R. A. Suarez-Fernandez, C. Sampedro, J. L. Sanchez-Lopez, P. Campoy, (2017): TML: a language to specify aerial robotic missions for the framework Aerostack. International Journal of Intelligent Computing and Cybernetics, Vol. 10 Issue: 4, pp.491-512.
- M. Molina, A. Diaz-Moreno, D. Palacios, R. A. Suarez-Fernandez, J. L. Sanchez-Lopez, C.s Sampedro, H. Bavle and P. Campoy (2016): "Specifying Complex Missions for Aerial Robotics in Dynamic Environments". International Micro Air Vehicle Conference and Competition, IMAV 2016. Beijing, China.

Complete Example

This example is a complete specification of a mission in TML language.

```
<!--=====================================================
| FILE: mission_specification_file.xml
| BRIEF: A mission specification example.
| DETAILS:
|   Mission in which an object is found, picked up and
|   retrieved.
| AUTHORS:
|    Guillermo De Fermin
=====================================================-->
<mission root_task="Search for object">

  <task_tree>
    <task name="Search for object">
      <task name="Initial phase">
        <recurrent_behavior name="SELF_LOCALIZE_BY_ODOMETRY" />
        <task name="Take off">
          <behavior name="TAKE_OFF"/>
        </task>
        <task name="Memorize current point">
          <consult belief="position(self, (?HOME_X, ?HOME_Y, ?HOME_Z))" />
        </task>
      </task>

      <task name="Start search">
        <recurrent_behavior name="SELF_LOCALIZE_BY_VISUAL_MARKERS" />
        <task name="Moving to point A">
          <behavior name="GO_TO_POINT">
            <argument name="relative_coordinates" value="[1, 0, 0]"/>
          </behavior>
        </task>
        <task name="Moving to point B">
          <behavior name="GO_TO_POINT">
            <argument name="relative_coordinates" value="[0, 1, 0]"/>
          </behavior>
        </task>
        <task name="Moving to point C">
```

```
        <behavior name="GO_TO_POINT">
          <argument name="relative_coordinates" value="[-1, 0, 0]"/>
        </behavior>
      </task>
      <task name="Moving to point D">
        <behavior name="GO_TO_POINT">
          <argument name="relative_coordinates" value="[0, -1, 0]"/>
        </behavior>
      </task>
    </task>

  </task>
</task_tree>

<task_tree>
  <task name="Retrieve object">
    <recurrent_behavior name="SELF_LOCALIZE_BY_VISUAL_MARKERS" />
    <task name="Memorize object location">
      <consult belief="position(object_1, (?OBJ_X, ?OBJ_Y, ?))" />
    </task>
    <task name="Hover above object">
      <behavior name="GO_TO_POINT">
        <argument name="coordinates" value="[+OBJ_X, +OBJ_Y, 1.2]" />
      </behavior>
    </task>
    <task name="Pick object up">
      <add belief="picked_up(object_1)" />
    </task>
    <task name="Go to starting point">
      <behavior name="GO_TO_POINT">
        <argument name="coordinates" value="[+HOME_X, +HOME_Y, +HOME_Z]" />
      </behavior>
    </task>
    <task name="Drop oject">
      <remove belief="picked_up(object_1)" />
    </task>
    <task name="Final land">
      <behavior name="LAND" />
    </task>
  </task>
</task_tree>

<task_tree>
  <task name="Emergency land">
    <behavior name="LAND" />
  </task>
</task_tree>

<event_handling>
  <event name="Object found">
    <condition belief="visible(object_1)"/>
    <call root_task="Retrieve object" />
    <termination mode="END_MISSION" />
  </event>

  <event name="Low battery">
    <condition belief="battery_level(self, LOW)"/>
    <call root_task="Emergency land"/>
    <termination mode="ABORT_MISSION"/>
  </event>
</event_handling>
</mission>
```

Contact

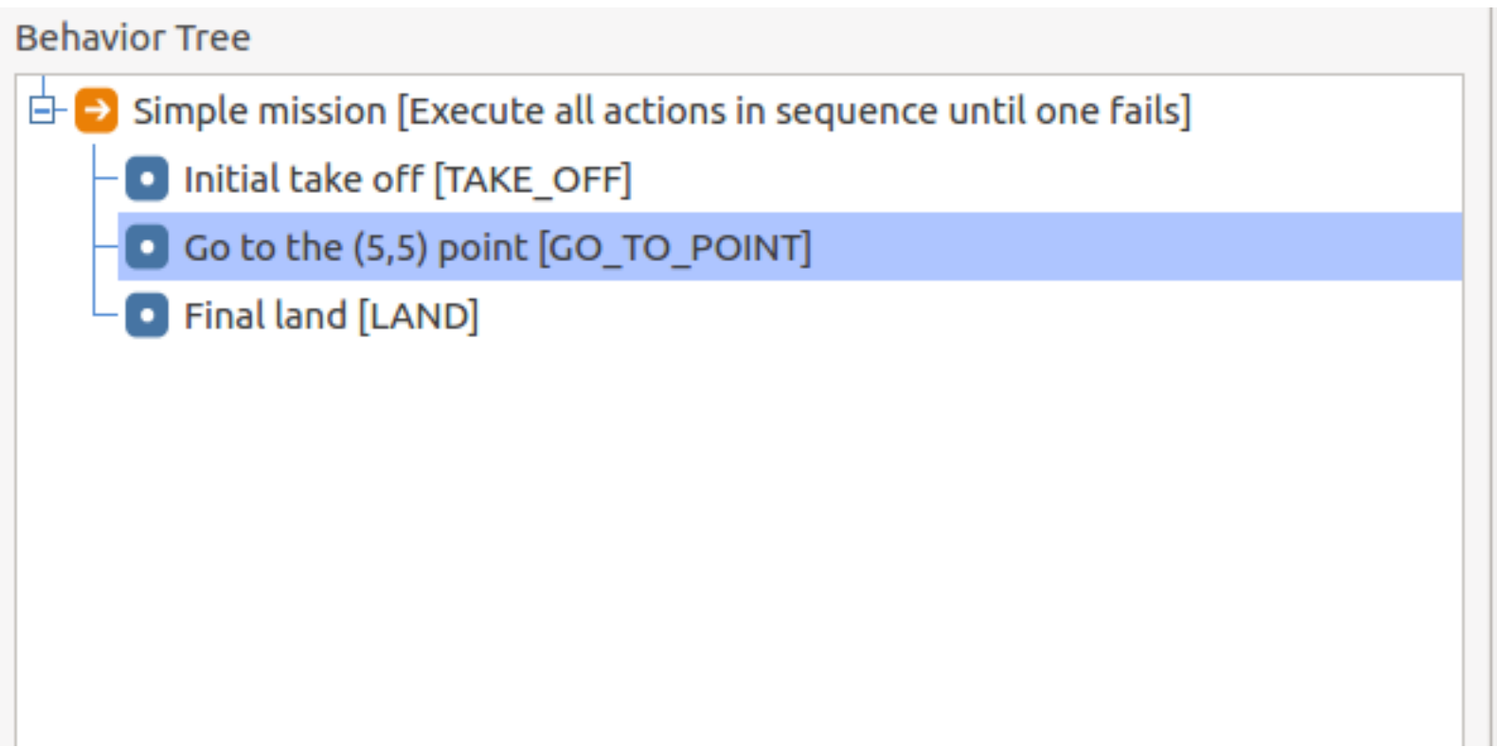
We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page.

You can also contact the team support at the following address: aerostack.vision4uav@gmail.com

Behavior Tree

Martin Molina edited this page on 3 Nov 2017 · 8 revisions

A behavior tree is a visual modelling language that employs a defined graphical notation to represent the behavior of a system. It is commonly used to create the artificial intelligence of games, thanks to their easy implementation and understanding.



Node types

A behavior tree is defined by a hierarchy of nodes. During the execution of a behavior tree, each node can succeed or can fail. When the operator creates a behavior tree, the operator gives a name to each node and associates a particular type.

Leaf nodes are used to execute simple actions. The types of leaf nodes are the following:

- **Behavior**: A behavior node specifies a particular behavior that Aerostack can execute, such as TAKE_OFF, LAND or GO_TO_POINT. To know the available list of behaviors, consult the following link: catalog of behaviors. If the behavior type is goal-based, this node succeeds when the behavior goal is accomplished. If the behavior type is recurrent, this node succeeds if the behavior is correctly initiated (or if it is correctly deactivated).
- **Add belief**: This node adds a belief expression to the memory of beliefs. It succeeds if the belief is correctly added, otherwise fails.
- **Remove belief**: This node removes a belief expression from the memory of beliefs. It succeeds if the belief is correctly removed, otherwise fails.
- **Query**: A query node corresponds to a question that is verified in the memory of beliefs. The question is formulated with a belief expression with a set of predicates that, optionally, can include variables. This node succeeds if the belief expression is true, otherwise it fails. Queries are useful to help control the mission workflow.

Intermediate nodes are used to control the execution of child nodes. The types of intermediate nodes are the following:

- **Sequence**: This node executes all its child nodes in sequence and succeeds when all the children succeed. Otherwise it fails.
- **Selector**: This node executes all its child nodes in sequence and succeeds when one of the children succeeds. If none of them succeed, it fails.

Pages 37

Introduction
- What is Aerostack?
- Getting Started
- License
- List of distributions

Components
- Aerostack Architecture
- Graphical User Interface
- Catalog of Behaviors
- Memory of Beliefs

Mission Plan Specification
- TML Language
- Behavior tree
- Python Language

Tutorials [Beginners]
- Prepare the Mission:
  - Configure Map
  - Specify Mission Tasks
  - Configure Launcher
- Execute the Mission:
  - Launch Real Flight
  - Practical Tips for Real Flights
- Mission examples:
  - Simulated Quadrotor
  - AR Drone and Visual Markers

Tutorials [Advanced]
- How to program a new behavior
- Setup the Communication Network
- Wi Fi Communication with Multiple Drones
- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

More information
- Publications
- Vision & Aerial Robotics:
  - First prize at IMAV 2013
  - Stack pre-release 2013

Clone this wiki locally

https://github.com/Vision4U
Clone in Desktop

- **Parallel**: This node executes all its child nodes in parallel. Let N be the number of child nodes. It returns success if the number of succeeding children is larger or equal than a local constant S, specified by the user. Returns failure if the number of failing children is larger than F = N - S.
- **Repeater**: This node repeats the execution of a child node a number of times. Returns success. It can only have one child node.
- **Repeat until fail**: This node repeats the execution of child nodes until the child node returns failure. Returns success. The child nodes are executed in sequence.
- **Inverter**: This node returns failure if the child node succeeds. Otherwise it succeeds.
- **Succeeder**: This node executes its child node and, no matter what is the result of the execution, it always succeeds.

Editing a behavior tree

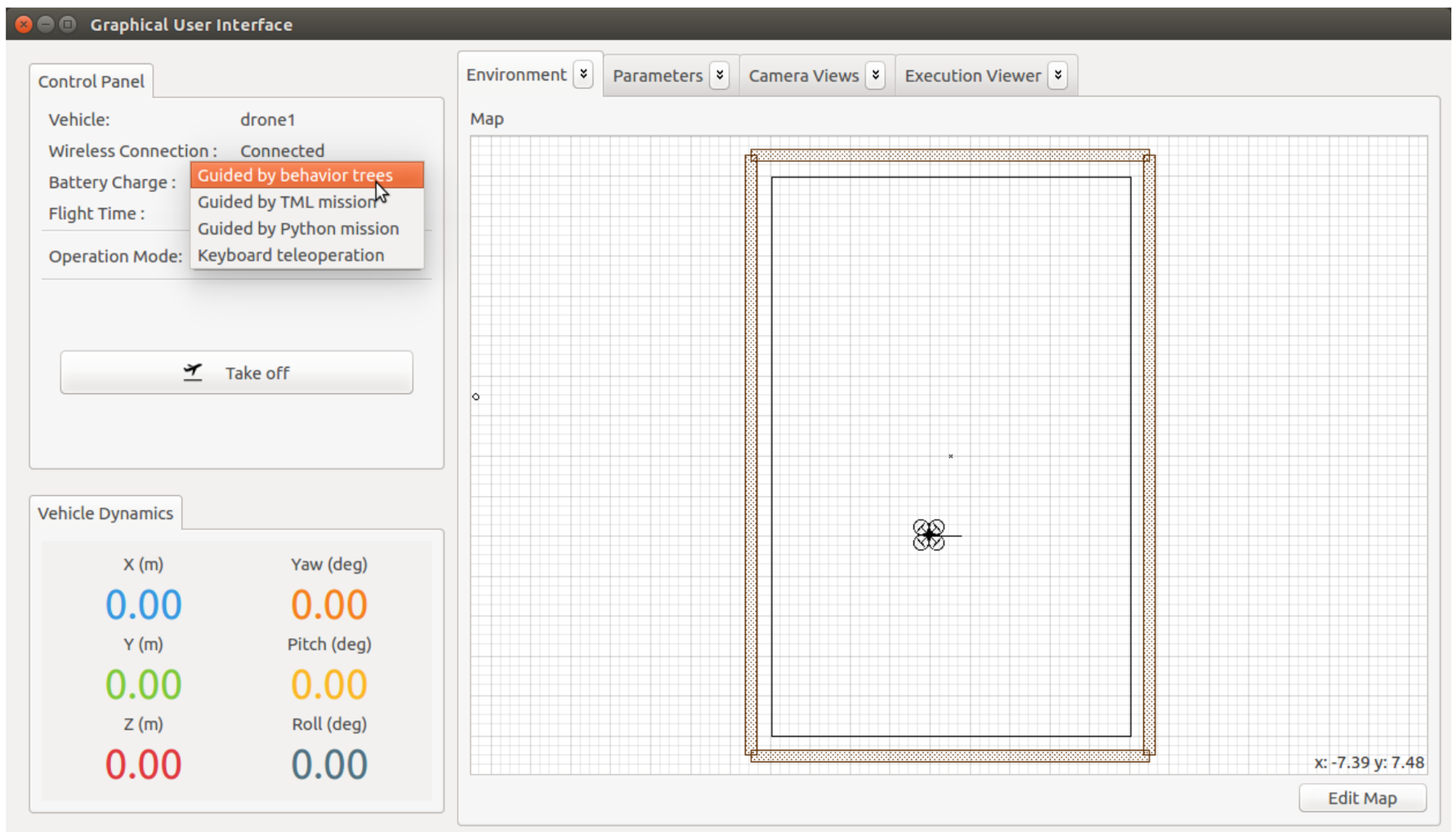
The operator can specify a mission plan using the behavior tree editor of the graphical user interface of Aerostack. Details about the graphical user interface can be consulted here:

- Graphical user interface

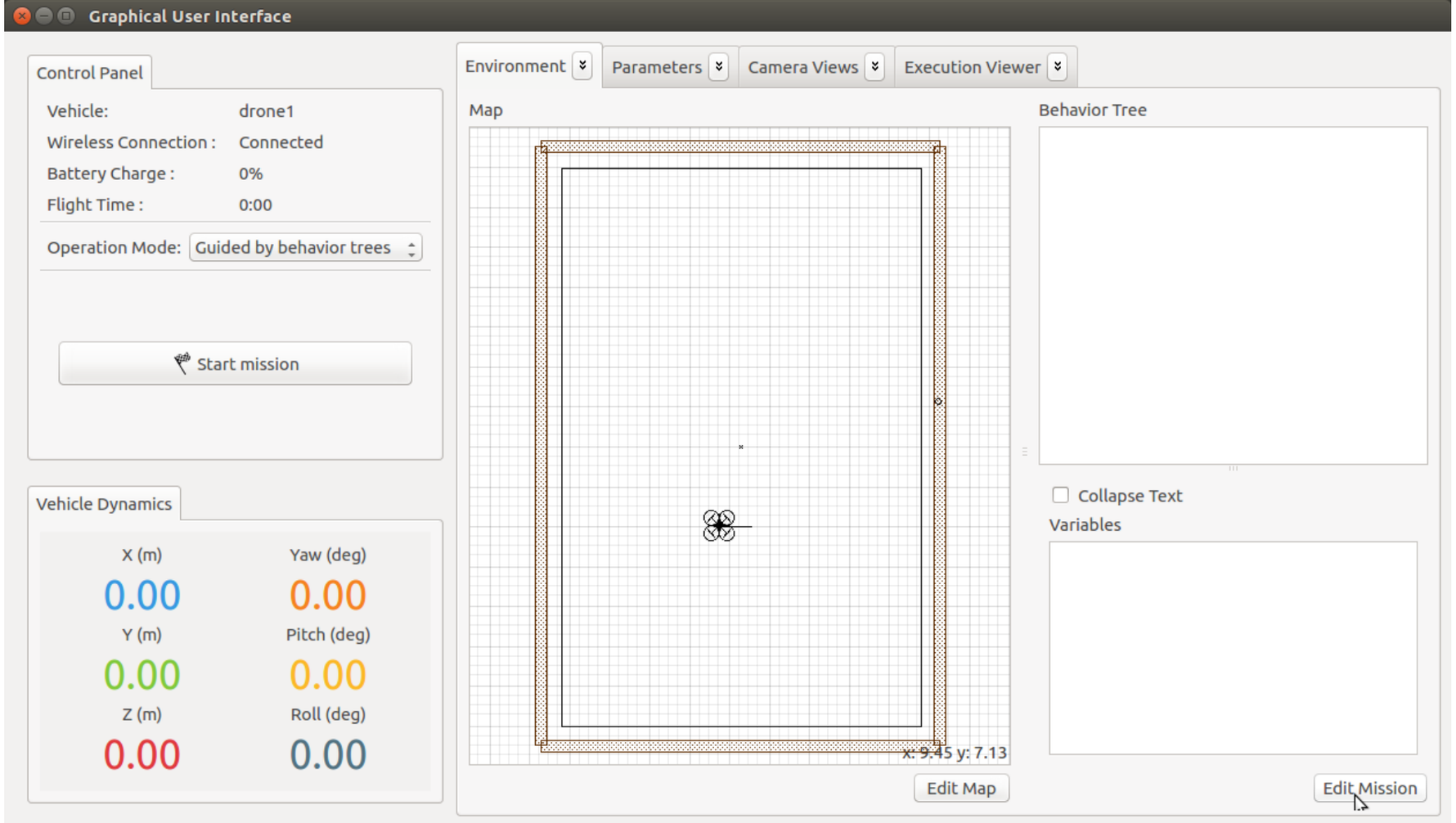
Keep in mind the behavior trees are not saved within the Aerostack configuration files. This means that a behavior tree is not persistent between different executions of Aerostack. If the operator wants to save his progress when editing a tree or doesn't want to lose the tree he just created, he can save the tree onto a *.yaml* file using the 'Export' button. Likewise you can import any tree using the 'Import' button.

The following steps describe how to create a behavior tree.

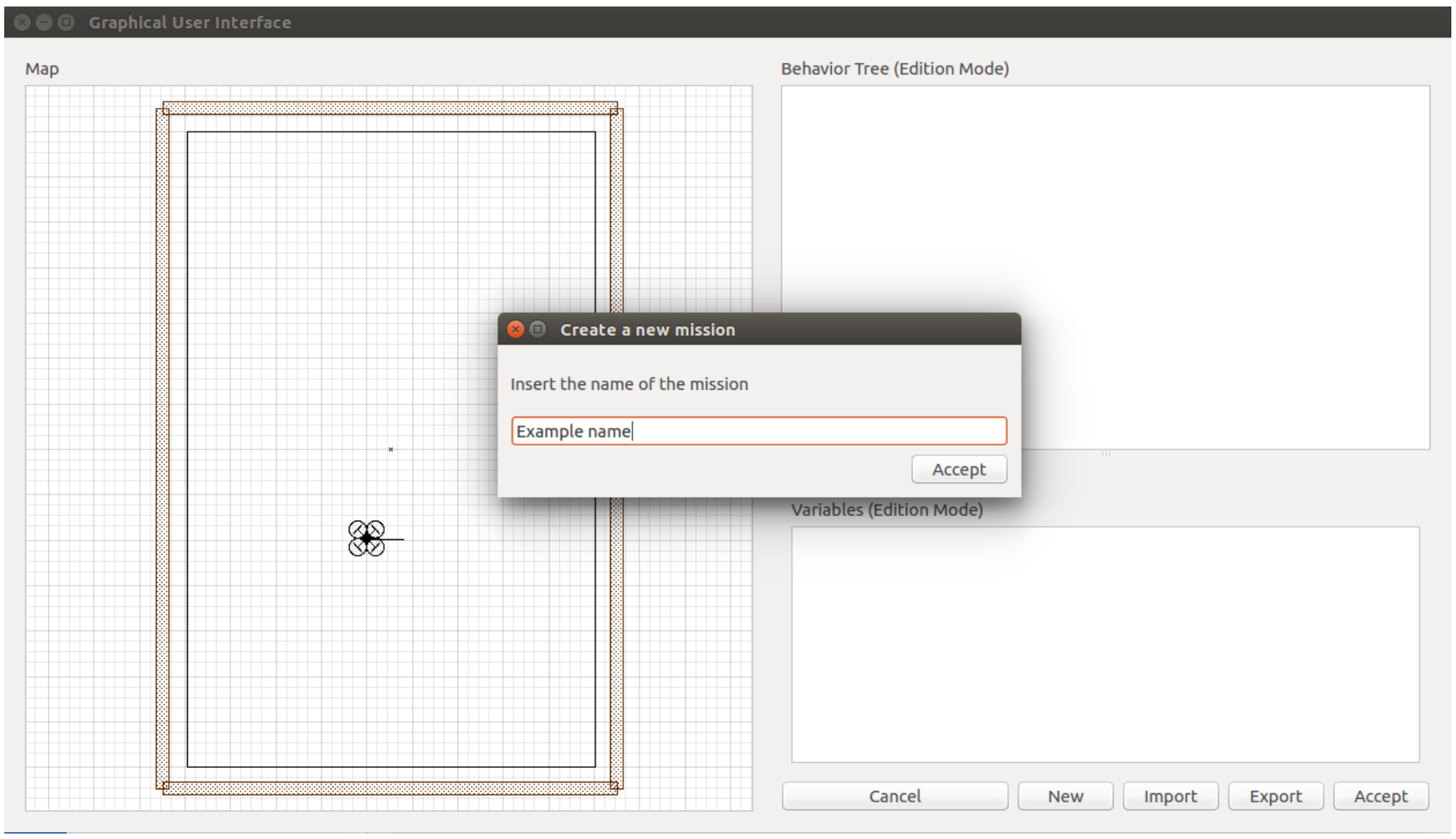
- Change the operation mode to 'Guided by behavior trees' in the control panel.



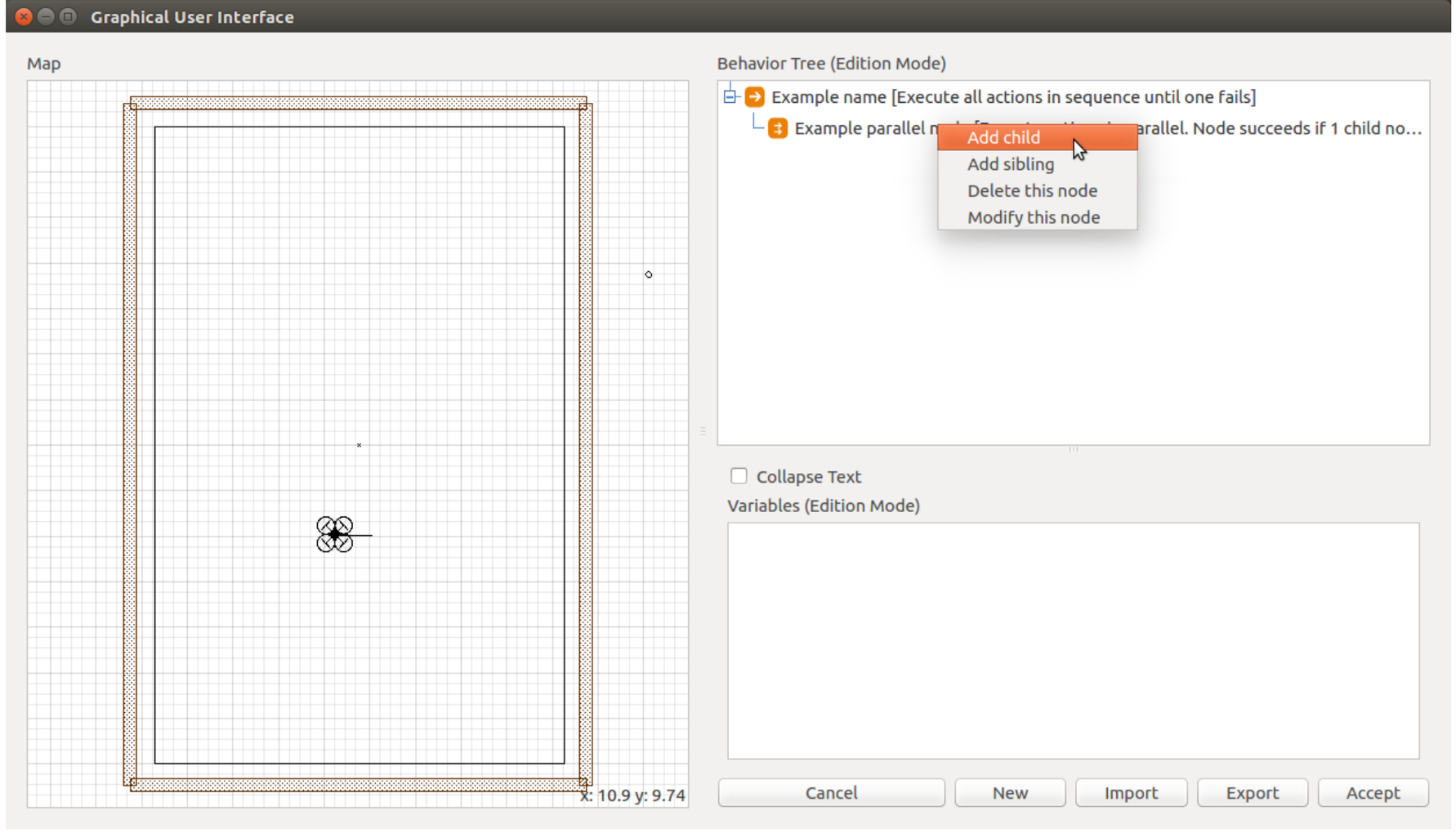
- Press the 'Edit mission' button at the right bottom corner of the window to edit the current behavior tree or to create a new one.

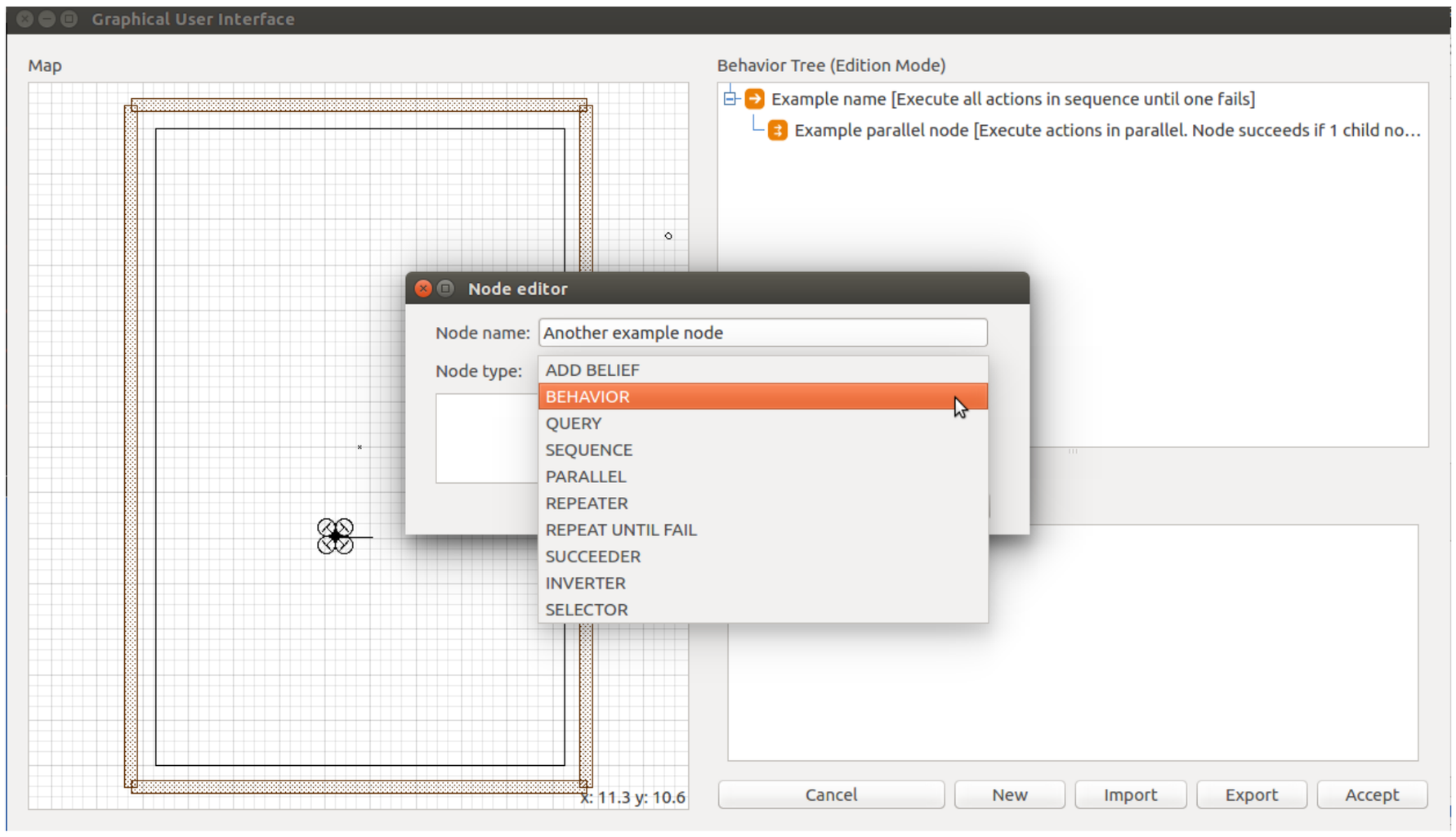


- Press 'New' button to create a new behavior tree or to replace the previous one. A dialog window is displayed asking for the name of the mission. It creates a new *sequence* root node.



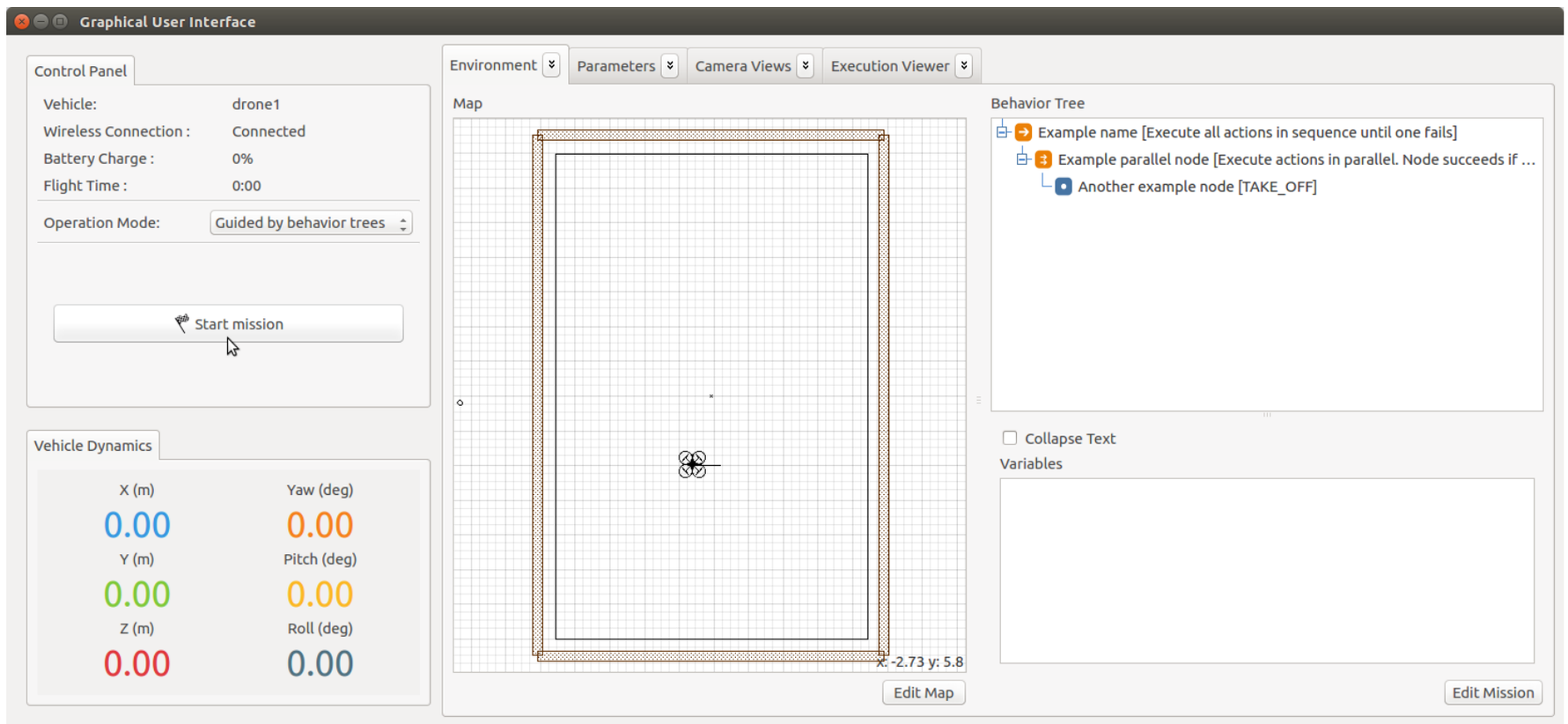
- By right-clicking in the desired node, a local menu is displayed to add a new child or sibling to the node, modify the node or remove it. The edition/creation dialog is displayed if those actions are chosen. It allows the user to select the type of node and its options. To make the changes available press 'Accept'.





- The operator can save the behavior tree in an external file (a *.yaml* file) by pressing the 'Export' button. Likewise, an existing external file corresponding to a mission that was previously saved, can be imported by pressing the 'Import' button.

- Finally, the user clicks 'Accept' button to exit from the edition mode and the behavior tree is ready to be executed.



Arguments, belief expressions and variables

When the user creates a node corresponding to a behavior, the user can write specific arguments. For example, if the user creates a node corresponding to the behavior GO_TO_POINT, the user can write the arguments in the corresponding box of the window as follows:

```
coordinates: [5, 4.2, 0.7]
```

When the user creates a node corresponding to a query for the memory of beliefs, the user can write the specific belief expression. For example the user can create a query node called "Save current position" with the belief expression:

```
position(self,(?X,?Y,?Z))
```

In this belief expression, X, Y, and Z are variables that store the coordinates corresponding to the current position. The question mark (?) indicates that such variables are substituted by the values found in the belief memory.

These variables can be used later in other behaviors or queries. For example the user can write a behavior node GO_TO_POINT with the following argument:

```
coordinates: [+X, +Y, +Z]
```

The sign plus (+) as a prefix of the variable name indicates that the variable is substituted by the value that the variable has in this moment.

The GUI shows a specific window box (below the behavior tree) with the list of variables used by the behavior tree and the values.

Complete example

Here's an example of a complex mission with different types of intermediate nodes.

Contact

We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page.

You can also contact the team support at the following address: aerostack.vision4uav@gmail.com

Python Language

Edit New Page

Martin Molina edited this page 2 days ago · 3 revisions

The operator can specify the tasks of a mission using Python language (based on Python 2) using certain predefined functions for mission plan execution. To be used by Aerostack, the specification must follow certain conventions that are explained below.

1. Simple example

The simplest mission one can do with the robot (taking off and landing) is the following:

```
import executive_engine_api as api

def runMission():
  api.executeBehavior('TAKE_OFF')
  api.executeBehavior('LAND')
```

The function `runMission()` is a special function that will get executed when the mission is started. As you can see, for this example it is necessary to import the `executive_engine_api` package in order to use its `executeBehavior()` function. This function executes any behavior and waits for it to complete. It returns a string with the termination status. The available behaviors can be found here:

- Catalog of behaviors

2. Recurrent behaviors

Recurrent behaviors don't terminate until they are inhibited, so they mustn't be executed with the function `executeBehavior()`, or they will block the program with and endless loop. In order to activate a behavior without waiting for its termination, the function `activateBehavior()` is provided. To inhibit a recurrent behavior, use the function `inhibitBehavior()`.

```
import executive_engine_api as api

def runMission():
  api.activateBehavior('SELF_LOCALIZE_BY_ODOMETRY')

  api.executeBehavior('TAKE_OFF')
  api.executeBehavior('LAND')

  api.inhibitBehavior('SELF_LOCALIZE_BY_ODOMETRY')
```

3. Accessing the belief memory

In order to add and remove beliefs from the belief memory the functions `addBelief()` and `removeBelief()` are provided. To consult the belief memory, two functions are provided: `consultBelief()` and `trueBelief()`. The latter simply returns true or false, while the first returns a boolean and a dictionary with the variables unified if it was successful.

```
import executive_engine_api as api

def runMission():
  api.addBelief('my_belief(something, something_else)')
  api.addBelief('my_other_belief(something)')

  result = api.trueBelief('my_belief(?X, ?Y)') # This will return true
```


Pages 37

Introduction
- What is Aerostack?
- Getting Started
- License
- List of distributions

Components
- Aerostack Architecture
- Graphical User Interface
- Catalog of Behaviors
- Memory of Beliefs

Mission Plan Specification
- TML Language
- Behavior tree
- Python Language

Tutorials [Beginners]
- Prepare the Mission:
  - Configure Map
  - Specify Mission Tasks
  - Configure Launcher
- Execute the Mission:
  - Launch Real Flight
  - Practical Tips for Real Flights
- Mission examples:
  - Simulated Quadrotor
  - AR Drone and Visual Markers

Tutorials [Advanced]
- How to program a new behavior
- Setup the Communication Network
- Wi Fi Communication with Multiple Drones
- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

More information
- Publications
- Vision & Aerial Robotics:
  - First prize at IMAV 2013
  - Stack pre-release 2013

Clone this wiki locally

```
  success, unification = api.consultBelief('my_belief(?X, ?), my_other_belief(?X)')
  if success: # Always check if query was successful before trying to access the unificat
    my_string = unification['X']

    api.removeBelief('my_belief(%s, something_else)' % my_string)
    api.removeBelief('my_other_belief(%s)' % my_string)
```

4. Complete example

This example is a complete specification of a mission in Python language.

```
import executive_engine_api as api
import random

def runMission():
  api.activateBehavior('SELF_LOCALIZE_BY_ODOMETRY')
  api.executeBehavior('TAKE_OFF')

  # Memorize starting point
  success, unification = api.consultBelief('position(self, (?X, ?Y, ?Z))')
  if success:
    startingPoint = [unification['X'], unification['Y'], unification['Z']]
  else:
    startingPoint = [0, 0, 0.7]

  api.inhibitBehavior('SELF_LOCALIZE_BY_ODOMETRY')

  api.activateBehavior('SELF_LOCALIZE_BY_VISUAL_MARKERS')
  objectFound = False
  # Start searching for object
  while not objectFound:
    nextPoint = [10*random.random(), 10*random.random(), 0.7]
    api.executeBehavior('GO_TO_POINT', coordinates=nextPoint)
    objectFound = api.trueBelief('position(object_1, ?)')

  success, unification = api.consultBelief('position(object_1, (?X, ?Y, ?Z))')
  if success:
    objectPosition = [unification['X'], unification['Y'], unification['Z']]

    # Set target position to 1m above object
    targetPosition = objectPosition
    targetPosition[2] += 1

    api.executeBehavior('GO_TO_POINT', coordinates=targetPosition)

    # Set object status to picked up
    api.addBelief('picked_up(object_1)')

    # Retrieve object
    api.executeBehavior('GO_TO_POINT', coordinates=startingPoint)

    # Drop object
    api.removeBelief('picked_up(object_1)')

  api.inhibitBehavior('SELF_LOCALIZE_BY_VISUAL_MARKERS')

  api.executeBehavior('LAND')
```

Contact

We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page.

You can also contact the team support at the following address: aerostack.vision4uav@gmail.com

https://github.com/Vision4U

Clone in Desktop

Configure Map

Martin Molina edited this page on 26 Oct 2017 · 3 revisions

The operator can configure the environment map using the graphical user interface (GUI). Please, consult the following link to know how to use the GUI to configure the environment map:

- How to create an environment map

Alternatively, the operator can also configure the environment map using a set of XML files. Please, consult the following link to know how to write these files:

- Configure the Environment Map Using XML Files

Pages 37

Introduction

- What is Aerostack?
- Getting Started
- License
- List of distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Catalog of Behaviors
- Memory of Beliefs

Mission Plan Specification

- TML Language
- Behavior tree
- Python Language

Tutorials [Beginners]

- Prepare the Mission:
  - Configure Map
  - Specify Mission Tasks
  - Configure Launcher
- Execute the Mission:
  - Launch Real Flight
  - Practical Tips for Real Flights
- Mission examples:
  - Simulated Quadrotor
  - AR Drone and Visual Markers

Tutorials [Advanced]

- How to program a new behavior
- Setup the Communication Network
- Wi Fi Communication with Multiple Drones
- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

More information

- Publications
- Vision & Aerial Robotics:
  - First prize at IMAV 2013
  - Stack pre-release 2013

Clone this wiki locally

https://github.com/Vision4U

Clone in Desktop

Contact

We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page.

You can also contact the team support at the following address: aerostack.vision4uav@gmail.com

Configure the Environment Map Using XML Files

Martin Molina edited this page on 26 Oct 2017 · 2 revisions

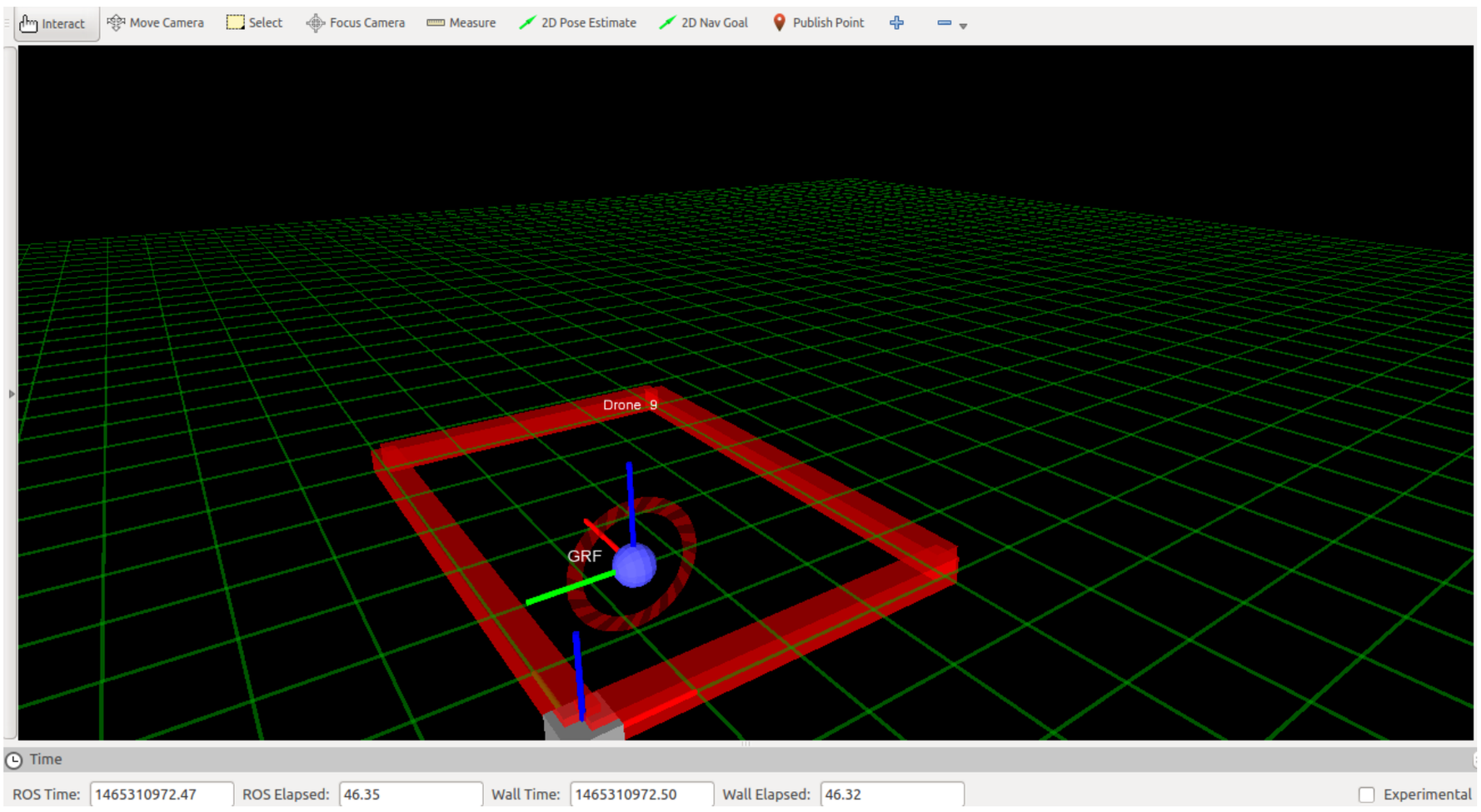
The environment map of the robot can be specified in a series of configuration files which follow XML syntax. These files are located at `$AEROSTACK_STACK/configs/` within a folder selected by the operator called `drone{Id}` (`{Id}` = 00, 01, 02, ...). The `Id` parameter is the identification of the drone used for the mission, which is specified in the launch file used. To change the configuration of the environment map follow these steps.

1.Defining the Size of the Map

The size of the map is defined in `configFile.xml`. For example:

```
<map>
  <config>
    <dimensions>4.0 6.0</dimensions>
    <initPoint>0.0 0.0</initPoint>
  </config>
</map>
```

The `<dimensions>` parameter specifies the size (x, y) in meters of the perimeter of the flight zone. The `<initPoint>` specifies where the origin to the perimeter of the map will be. This parameter can be set in any value and has to be taken into account when placing the obstacles in the map. In this example the map size is 4m x 6m and the init point is set to (0,0).



2.Defining the Take-off Point

The take-off point should be specified in two files:

- the file `ekf_state_estimator_config.xml`
- the localization initial point in the file `param_localization_obs.xml`

For this example the start point in the x and y coordinates is (1m, 1m). Therefore, both files have this point set to (1, 1) but some changes must be made for each of these files.

2.1 EKF Take-off Point

Pages 37

Introduction

- What is Aerostack?
- Getting Started
- License
- List of distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Catalog of Behaviors
- Memory of Beliefs

Mission Plan Specification

- TML Language
- Behavior tree
- Python Language

Tutorials [Beginners]

- Prepare the Mission:
  - Configure Map
  - Specify Mission Tasks
  - Configure Launcher
- Execute the Mission:
  - Launch Real Flight
  - Practical Tips for Real Flights
- Mission examples:
  - Simulated Quadrotor
  - AR Drone and Visual Markers

Tutorials [Advanced]

- How to program a new behavior
- Setup the Communication Network
- Wi Fi Communication with Multiple Drones
- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

More information

- Publications
- Vision & Aerial Robotics:
  - First prize at IMAV 2013
  - Stack pre-release 2013

Clone this wiki locally

The take-off point defined in `ekf_state_estimator_config.xml specifies the take-off point used by the (Extended Kalman Filter) EKF to start the estimation. This estimation is started with the drone in landed position so the height (z) is set to 0.

```
<take_off_site>
  <position>
    <x>1.0</x>
    <y>1.0</y>
    <z>0.0</z>
  </position>
  <attitude>
    <yaw>  90.00</yaw>
    <pitch> 0.00</pitch>
    <roll>  0.00</roll>
  </attitude>
</take_off_site>
```

Units:

- Units for tag `<position>`: meters
- Units for tag `<attitude>`: degrees (wYcPuR)

2.2 Aruco Visual Marker Take-off Point

The take-off point specified by the `param_localization_obs.xml` is used to determine where the Aruco visual marker localization start point is set in the map. This start point for the localization is done after takeoff and therefore is set to z = 0.7m.

```
<!-- Localization  -->
  <!-- Init pose  -->
  <x_0> 1.0</x_0>
  <y_0> 1.0</y_0>
  <z_0> 0.7</z_0>
  <yaw_0>  90.0</yaw_0>
  <pitch_0> 0.0</pitch_0>
  <roll_0>  0.0</roll_0>
```

Position is specified in meters and angles are given in degrees.

As you can see, both of these configuration files have initial values for Roll, Pitch and Yaw. Roll and Pitch will have a value of zero, since localization is started with these values. Nevertheless, the Yaw value is given according to the mission requirements of the user. If the user wishes to have the drone take-off facing the +x axis then this yaw value will be 0. In this example the dronewill take-off facing the +y axis so the rotation in yaw is +90deg.

3. Creating Obstacles and Virtual Walls

Walls and obstacles are defined in the `obstacle_config.xml` file. This file allows the creation of obstacles that can be real (to be detected by Aruco visual markers) or can be virtual.

Virtual walls cover the area of the map in order to restrain the movement of the drone to the desired map space. They can be created based on the size of the map. For example for a map of 4x6, the virtual borders can be defined as shown in the figure below.

```
<!-- borders (Virtual Obstacles): 10000-19999 -->
<rectangle description="bottom side">
  <id>10000</id>
  <centerPoint>2.0 0.0</centerPoint>
  <size>4.0 0.3</size>
  <yawAngle>0.0</yawAngle>
  <virtual description="define a virtual object">1</virtual>
  </rectangle>
```

https://github.com/Vision4U
Clone in Desktop

```
<rectangle description="right side">
  <id>10001</id>
  <centerPoint>4.0 3.0</centerPoint>
  <size>0.3 6.0</size>
  <yawAngle>0.0</yawAngle>
  <virtual description="define a virtual object">1</virtual>
  </rectangle>

<rectangle description="up side">
  <id>10002</id>
  <centerPoint>2.0 6.0</centerPoint>
  <size>4.0 0.3</size>
  <yawAngle>0.0</yawAngle>
  <virtual description="define a virtual object">1</virtual>
  </rectangle>

<rectangle description="left side">
  <id>10003</id>
  <centerPoint>0.0 3.0</centerPoint>
  <size>0.3 6.0</size>
  <yawAngle>0.0</yawAngle>
  <virtual description="define a virtual object">1</virtual>
  </rectangle>
```

Besides specifying the boundaries of the map to be flown, the `obstacle_config.xml` file relates Aruco visual markers to obstacles in the map.

```
<pole description="A">
  <id>50002</id>
    <radius>0.15 0.15</radius>
    <aruco>
      <id aruco_id = "17"/>
      <id aruco_id = "18"/>
      <id aruco_id = "19"/>
      <id aruco_id = "20/>
    </aruco>
</pole>
```

As can be seen from the figure above, four Aruco markers (id: 17, 18, 19, 20) are used to define an obstacle, in this case a pole of 0.15 radius. With this file the user can specify which Aruco markers make up an obstacle and the geometry it can have.

4. Placing Aruco Visual Markers in the Map

The location of the visual markers used in the world is defined in the 'param_localization_obs.xml' file. In this file the user can specify the Aruco Slam details. Here the visual marker Ids that can be mapped by visual slam can be specified with `<VisualMarkersRange>`.

Also in this file the user can specify the fixed visual markers in the map.

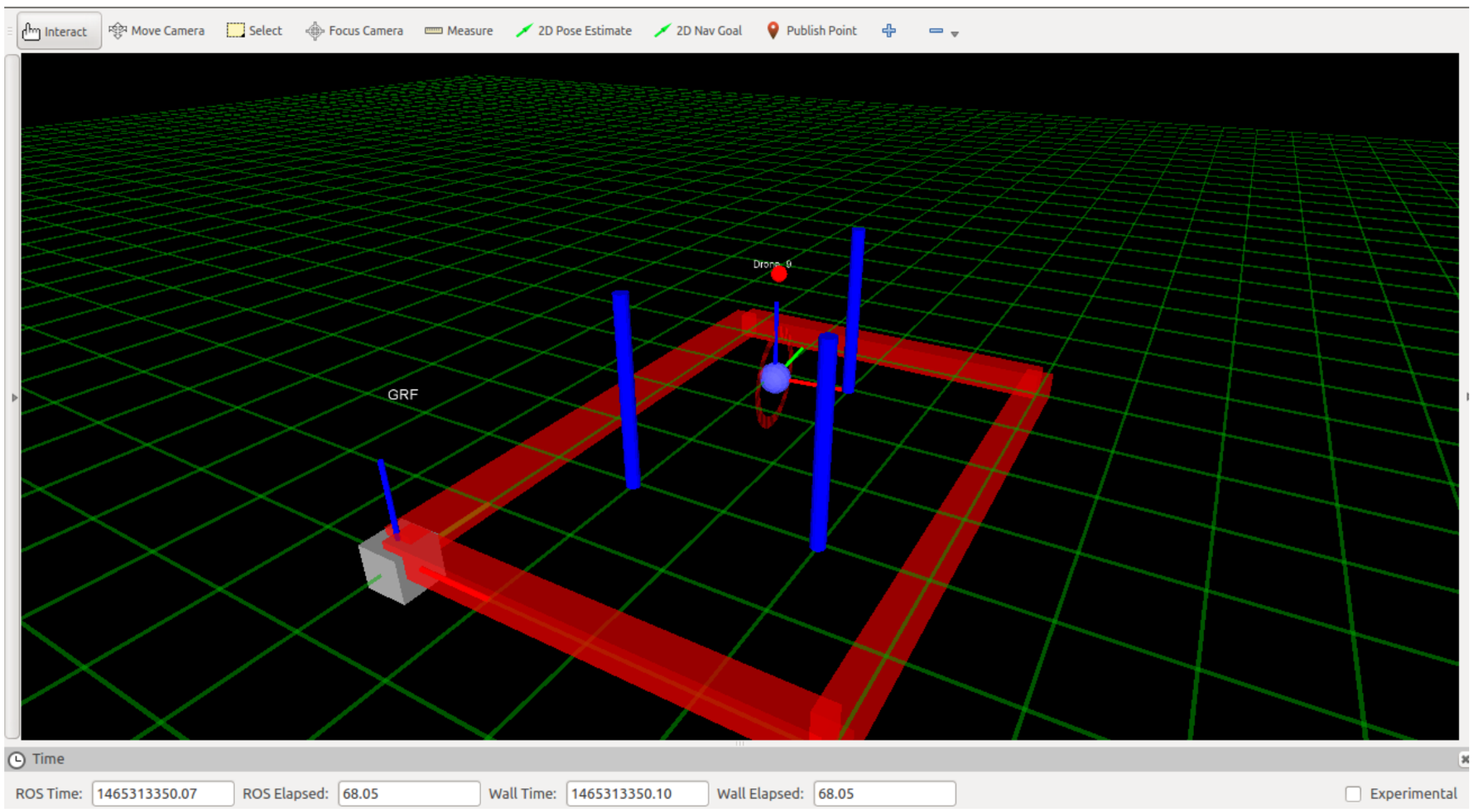
4.1 Visual Markers for Obstacles

This example uses three poles as known obstacles to navigate through and avoid. The location of each pole is given by the (x, y, z) coordinates of the center of the Aruco placed on the pole. Since the size of the poles used in this mission are 40cm in diameter, the location for the center of each Aruco is shown in the following example:

```
<!-- Pole 1 -->
<Aruco id="9"  x="3.0"  y="1.80" z="1.3" roll="0" pitch="0" yaw="-90"/>
<Aruco id="10" x="2.80" y="2.0"  z="1.3" roll="0" pitch="0" yaw="180"/>
<Aruco id="11" x="3.0"  y="2.20" z="1.3" roll="0" pitch="0" yaw="90"/>
<Aruco id="12" x="3.20" y="2.0"  z="1.3" roll="0" pitch="0" yaw="0"/>

<!-- Pole 2 -->
<Aruco id="13" x="1.0"  y="1.80" z="1.3" roll="0" pitch="0" yaw="-90"/>
```

```
<Aruco id="14" x="0.80" y="2.0"  z="1.3" roll="0" pitch="0" yaw="180"/>
<Aruco id="15" x="1.0"  y="2.20" z="1.3" roll="0" pitch="0" yaw="90"/>
<Aruco id="16" x="1.20" y="2.0"  z="1.3" roll="0" pitch="0" yaw="0"/>

<!-- Pole 3 -->
<Aruco id="25" x="2.0"  y="4.80" z="1.3" roll="0" pitch="0" yaw="-90"/>
<Aruco id="26" x="1.80" y="5.0"  z="1.3" roll="0" pitch="0" yaw="180"/>
<Aruco id="27" x="2.0"  y="5.20" z="1.3" roll="0" pitch="0" yaw="90"/>
<Aruco id="28" x="2.20" y="5.0"  z="1.3" roll="0" pitch="0" yaw="0"/>
```

Where the desired location of the obstacles is (3.0,2.0), (1.0,2.0) and (2.0, 5.0).



4.1 Visual Markers for Landmarks (for Self-Localization)

Besides the known pole location, landmarks are used for self-localization and are generally placed at the end of the map (e.g., where the mission of visual servoing will be performed). Each landmark carries two Aruco visual markers. If the landmark is placed on the y axis facing to the right, 0.11 needs to be subtracted from the y axis of the first aruco and 0.11 needs to be added to the y axis of the second aruco as shown in the figure below.

```
<!-- Landmark 1 -->
<Aruco id="61" x="0.0" y="3.89" z="1.3" roll="0" pitch="0" yaw="-45"/>
<Aruco id="62" x="0.0" y="4.11" z="1.3" roll="0" pitch="0" yaw="45"/>
```

If the landmark is placed on the x axis, facing the take-off point 0.11 needs to be subtracted from the x axis of the first aruco and 0.11 needs to be added to the x axis of the second aruco as shown in the figure below

```
<!-- Landmark 4 -->
<Aruco id="69" x="2.89" y="6.0" z="1.3" roll="0" pitch="0" yaw="225"/>
<Aruco id="70" x="3.11" y="6.0" z="1.3" roll="0" pitch="0" yaw="-45"/>
```

In this example, six landmarks are used:

```
<!-- Landmark 1 -->
<Aruco id="61" x="0.0" y="3.89" z="1.3" roll="0" pitch="0" yaw="-45"/>
<Aruco id="62" x="0.0" y="4.11" z="1.3" roll="0" pitch="0" yaw="45"/>

<!-- Landmark 2 -->
<Aruco id="63" x="0.0" y="4.89" z="1.3" roll="0" pitch="0" yaw="-45"/>
<Aruco id="64" x="0.0" y="5.11" z="1.3" roll="0" pitch="0" yaw="45"/>

<!- Landmark 3 -->
<Aruco id="65" x="0.89" y="6.00" z="1.3" roll="0" pitch="0" yaw="225"/>
<Aruco id="66" x="1.11" y="6.00" z="1.3" roll="0" pitch="0" yaw="-45"/>
```

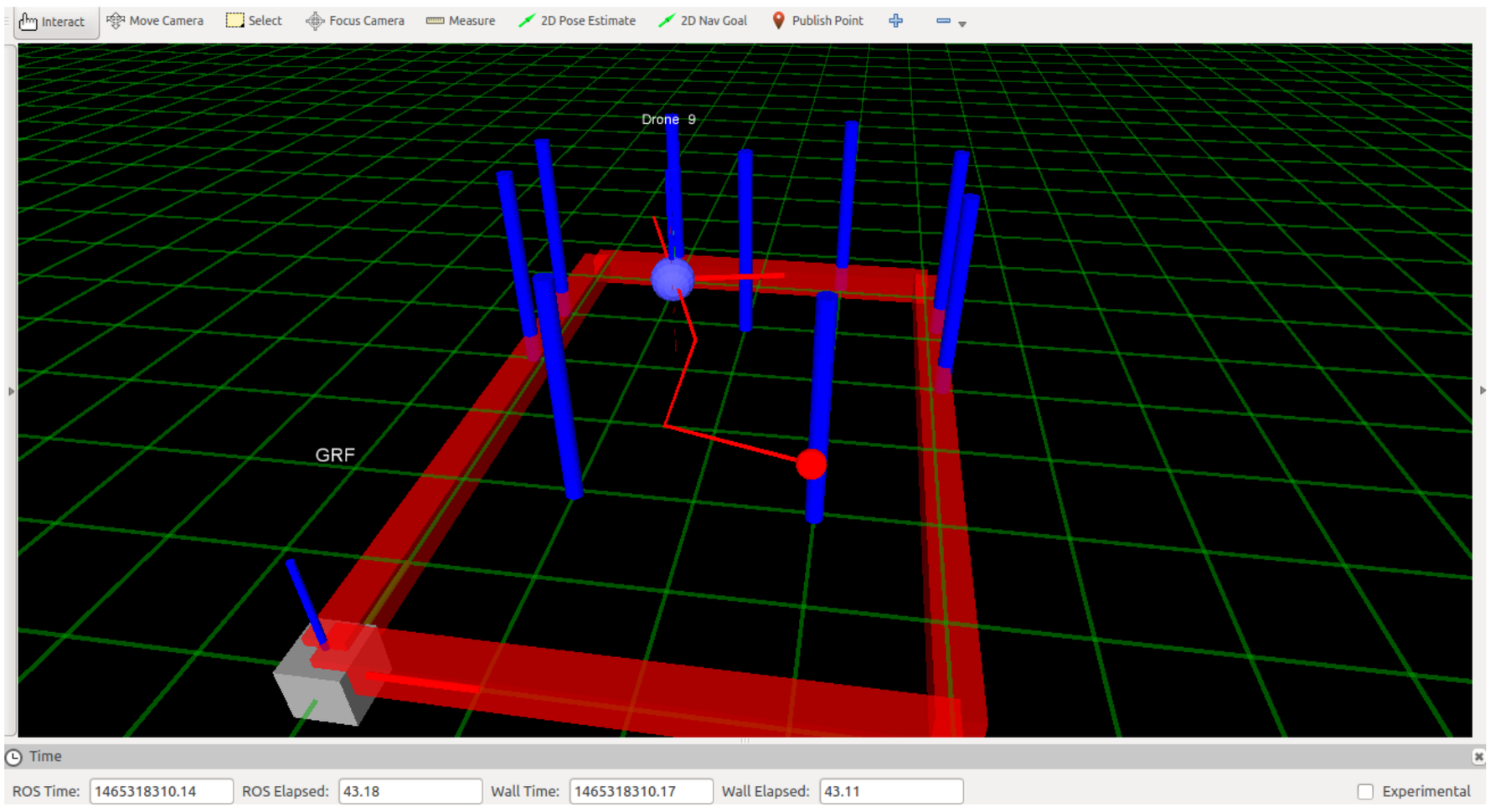
```
<!- Landmark 4 -->
<Aruco id="69" x="2.89" y="6.00" z="1.3" roll="0" pitch="0" yaw="225"/>
<Aruco id="70" x="3.11" y="6.00" z="1.3" roll="0" pitch="0" yaw="-45"/>

<!- Landmark 5 -->
<Aruco id="71" x="4.0" y="5.11" z="1.3" roll="0" pitch="0" yaw="225"/>
<Aruco id="72" x="4.0" y="4.89" z="1.3" roll="0" pitch="0" yaw="-45"/>

<!- Landmark 6 -->
<Aruco id="73" x="4.0" y="4.11" z="1.3" roll="0" pitch="0" yaw="135"/>
<Aruco id="74" x="4.0" y="3.89" z="1.3" roll="0" pitch="0" yaw="225"/>
```

5. Verify the Changes

After the environment is configured correctly, you can verify the changes using the RViz interface. For this purpose, run the following command:

```
$ ./rvizInterface_Test.sh drone{Id} localhost
```

It should presents a 3D image similar to this:



Contact

We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page.

You can also contact the team support at the following address: aerostack.vision4uav@gmail.com

Specify Mission Tasks

Martin Molina edited this page on 26 Oct 2017 · 3 revisions

The operator can specify a mission plan using three alternative options:

- TML language (a special-purpose language of Aerostack for mission plans).
- Behavior tree (with a graphical editor).
- Phyton language (with a library of functions).

Specify mission tasks in TML language

Follow these steps to specify the tasks of a mission using TML language:

- Edit the mission specification file. The TML mission of the robot is specified in a XML file called `mission_specification_file.xml` (this is the default name, as it is established in the .launch file located within the `launch` directory in the repository `task_based_mission_planner_process`). This file must be located in `$AEROSTACK_STACK/configs/` within a folder selected by the operator called `drone{Id}` (`{Id}` = 00, 01, 02, ...). Write your own mission specification file according to TML language.

- Check syntax errors. To check if the file has syntax errors, launch the mission planner executing the following command:

```
> roslaunch task_based_mission_planner_process
task_based_mission_planner_process.launch
```

- If the xml file does not have any errors, the following message is presented:

```
File loaded without errors! Waiting to start ...
```

Specify mission tasks using a behavior tree

Aerostack provides a graphical editor to specify missions using a visual modeling language. To know how to specify a mission with behavior trees, consult the following link:

- Behavior tree

Specify mission tasks in Python language

The operator of Aerostack may use Python language to specify a mission plan. To know how to specify a mission in Python language, consult the following link:

- Python language

Pages 37

Introduction

- What is Aerostack?
- Getting Started
- License
- List of distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Catalog of Behaviors
- Memory of Beliefs

Mission Plan Specification

- TML Language
- Behavior tree
- Python Language

Tutorials [Beginners]

- Prepare the Mission:
  - Configure Map
  - Specify Mission Tasks
  - Configure Launcher
- Execute the Mission:
  - Launch Real Flight
  - Practical Tips for Real Flights
- Mission examples:
  - Simulated Quadrotor
  - AR Drone and Visual Markers

Tutorials [Advanced]

- How to program a new behavior
- Setup the Communication Network
- Wi Fi Communication with Multiple Drones
- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

More information

- Publications
- Vision & Aerial Robotics:
  - First prize at IMAV 2013
  - Stack pre-release 2013

Clone this wiki locally

https://github.com/Vision4U

Clone in Desktop

Contact

We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page.

You can also contact the team support at the following address: aerostack.vision4uav@gmail.com

Configure Launcher

Martin Molina edited this page on 26 Oct 2017 · 5 revisions

The launcher is a file that specifies the set of processes that will be executed for a particular aerial robot system and and a specific mission to be done.

To configure a launcher, it is necessary to select a particular set of Aerostack processes. Each basic functionality (e.g., recognize ArUco visual markers, self-localization, mission planning, etc.) is provided by a set of processes. To configure the launcher, perform the following steps.

Go to the folder that contains examples of launchers. This folder includes .sh files that launch sets of processes for different architecture configurations:

```
$ cd ${AEROSTACK_STACK}/launchers
```

Select the .sh file that corresponds to an architecture similar to the architecture that you want to to create. Copy the selected file into a new .sh file. Edit the content of the new .sh file to add or remove processes. For example, to add the process task_based_mission_planner, add a new call to roslaunch such as the following:

```
gnome-terminal  \
...
--tab --title "Mission Planner" --command "bash -c \"
roslaunch task_based_mission_planner_process task_based_mission_planner_process.launch
--wait drone_id_namespace:=drone$NUMID_DRONE drone_id_int:=$NUMID_DRONE
my_stack_directory:=${DRONE_STACK};
                                                exec bash\"" \
...
```

To know the available functionalities and processes provided by Aerostack, see the following catalogue:

- Catalog of Processes

Pages 37

Introduction

- What is Aerostack?
- Getting Started
- License
- List of distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Catalog of Behaviors
- Memory of Beliefs

Mission Plan Specification

- TML Language
- Behavior tree
- Python Language

Tutorials [Beginners]

- Prepare the Mission:
  - Configure Map
  - Specify Mission Tasks
  - Configure Launcher
- Execute the Mission:
  - Launch Real Flight
  - Practical Tips for Real Flights
- Mission examples:
  - Simulated Quadrotor
  - AR Drone and Visual Markers

Tutorials [Advanced]

- How to program a new behavior
- Setup the Communication Network
- Wi Fi Communication with Multiple Drones
- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

More information

- Publications
- Vision & Aerial Robotics:
  - First prize at IMAV 2013
  - Stack pre-release 2013

Clone this wiki locally

https://github.com/Vision4U

Clone in Desktop

Contact

We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page.

You can also contact the team support at the following address: aerostack.vision4uav@gmail.com

Launch Real Flight

Martin Molina edited this page on 26 Oct 2017 · 3 revisions

To launch a real flight using an AR Drone perform the following steps:

- First, connect the AR Drone to the computer using wifi.
- Launch the roscore

```
$ roscore
```

- Go to the folder that contains launchers. It includes different pre-configured .sh files that launch the set of processes for different mission types:

```
$ cd ${AEROSTACK_STACK}/launchers
```

- In this example we select the file *ardrone_basic.sh*. Write the following to launch the mission:

```
$ ./ardrone_basic.sh
```

Then you can use the human machine interface to start the mission (see Graphical User Interface). The AR Drone will execute the mission defined in the config files.

Pages 37

Introduction

- What is Aerostack?
- Getting Started
- License
- List of distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Catalog of Behaviors
- Memory of Beliefs

Mission Plan Specification

- TML Language
- Behavior tree
- Python Language

Tutorials [Beginners]

- Prepare the Mission:
  - Configure Map
  - Specify Mission Tasks
  - Configure Launcher
- Execute the Mission:
  - Launch Real Flight
  - Practical Tips for Real Flights
- Mission examples:
  - Simulated Quadrotor
  - AR Drone and Visual Markers

Tutorials [Advanced]

- How to program a new behavior
- Setup the Communication Network
- Wi Fi Communication with Multiple Drones
- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

More information

- Publications
- Vision & Aerial Robotics:
  - First prize at IMAV 2013
  - Stack pre-release 2013

Clone this wiki locally

https://github.com/Vision4U

Clone in Desktop

Contact

We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page.

You can also contact the team support at the following address: aerostack.vision4uav@gmail.com

Practical Tips for Real Flights

Martin Molina edited this page on 30 Jan 2017 · 4 revisions

This section includes a set of practical tips to take into account in real flights.

Multi-drone flight

In order to avoid drone collissions during the flight, you have to publish manually a message in the missionState topic after the complete initialization of Aerostack.

```
rostopic pub -1 /<DRONE_ID>/missionState std_msgs/Bool "data: true"
```

You must publish the previous message for each drone you are using.

Correct topics for Aruco-based localization

If the flight requires aruco-based positioning, be sure that you use the topics `ArucoSlam_EstimatedPose` and `ArucoSlam_EstimatedSpeeds` in your launcher, instead of `EstimatedPose_droneGMR_wrt_GFF` and `EstimatedSpeed_droneGMR_wrt_GFF`.

NOTE: If you are experiencing problems with the drone's localization, consider increasing the density of Aruco visual markers.

Points for the Trajectory Planner

The trajectory planner uses a set of randomly generated points that can be modified in the file `configFile.xml`. If you find that trajectory calculation is too slow you can try decreasing this value to reduce the calculation cost. However, keep in mind that decreasing it too much might cause the trajectory planner not to find trajectories in certain situations.

In the file *configFile.xml* located in the *configs/droneN* directory, you can specify the number of points used by the Trajectory Planner's algorithm to calculate trajectories. A high number will result in better trajectories, but will impact performance. To increase the probability of a good trajectory without too low performance, you can add a subzone with additional nodes for a difficult area. This can also work if you want to increase the probability of generating trajectories that pass through a certain area, since the Trajectory Planner favours trajectories with a higher number of nodes.

```
<config>
    <numNodes>3500</numNodes>
    <neigbourhood>5</neigbourhood>

    <subzone>
        <numNodes>500</numNodes>
        <initPoint>4.0 0.0</initPoint>
        <size>2.0 3.0</size>
    </subzone>
</config>
```

In this example, we add an extra 500 nodes to a 2m by 3m zone in (4, 0) where there is a narrow window that can make trajectory calculation more difficult.

Collect data about flight

To record the data of the flight you just have to run the following script before starting the mission.

Pages 37

Introduction
- What is Aerostack?
- Getting Started
- License
- List of distributions

Components
- Aerostack Architecture
- Graphical User Interface
- Catalog of Behaviors
- Memory of Beliefs

Mission Plan Specification
- TML Language
- Behavior tree
- Python Language

Tutorials [Beginners]
- Prepare the Mission:
  - Configure Map
  - Specify Mission Tasks
  - Configure Launcher
- Execute the Mission:
  - Launch Real Flight
  - Practical Tips for Real Flights
- Mission examples:
  - Simulated Quadrotor
  - AR Drone and Visual Markers

Tutorials [Advanced]
- How to program a new behavior
- Setup the Communication Network
- Wi Fi Communication with Multiple Drones
- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

More information
- Publications
- Vision & Aerial Robotics:
  - First prize at IMAV 2013
  - Stack pre-release 2013

Clone this wiki locally

```
#!/bin/bash

OUTFILE=$1
TOPICS="/public_event /rosout /rosout_agg /societyBroadcast /tf"

for i in $*; do
  if [[ "$i" != "$1" ]]; then
    DRONE_ID="drone$i"
    TOPICS="$TOPICS /$DRONE_ID/ArucoSlam_EstimatedPose   /$DRONE_ID/ArucoSlam_EstimatedPo
  fi
done

rosbag record -O $OUTFILE $TOPICS
```

Run the script like this:

```
./rosbag-multidrone.sh outfile.bag <DRONE_ID_1> <DRONE_ID_2> ... <DRONE_ID_N>
```

Where DRONE_ID is the number id of the drone.

Battery levels

Keep in mind that the lower the battery level is, the lower the power the drones' rotor will generate. In order to avoid that kind of problem, it is very recommended to change the battery when it has 30 percent or less.

Aerostack reboot

Whenever the drone is disconnected (because of a battery change or a similar reason), it is highly reccommended to restart Aerostack and roscore to prevent unexpected behaviour.

If any of the processes crash, it is necessary to restart Aerostack to ensure everything is loaded correctly.

OpenTLD GUI for Visual Servoing

When you are using the OpenTLD GUI you have to press the F5 button to refresh the image o the screen, select the area you want the drone to follow and then press and hold (just for few seconds) the ENTER key. Then the drone will start following the selected area.

Emergency mode

This state is used by the AR Drone to stop each of the rotors. This can be useful if the drone stops responding or it runs into an obstacle. To enter emergency mode you have two ways:

- Grab the drone with your hands and flip it so it faces downwards. Useful when the drone stops responding and you have no other way to land.
- Use the Command Line Interface and press the SPACE key.

To exit Emergency mode, you can press the SPACE key again. You will not be able to start a mission until you exit Emergency mode.

Avoid duplicate identifiers

It is important to check that identifiers for poles, walls or landmarks in the map are unique. If you use the same identifier for different objects in the map, the trajectory planner cannot work properly and could crash.

Efficient communication

https://github.com/Vision4U
Clone in Desktop

To check if the connection to the drone is stable and fast enough, you can run the command `rostopic hz /drone0/camera/front/image_raw`. It should display frequencies of around 30Hz. If this is not the case, you should consider changing the drone's channel to a less occupied one (this can be done with a parameter in the launcher).

Visual servoing processes

If visual servoing produces unxpected behaviour, double check that the following processes have been started (with `rosnode list`):

- /droneN/droneIBVSController
- /droneN/open_tld_translator
- /droneN/ros_tld_tracker_node
- /droneN/ros_tld_gui_node
- /droneN/trackerEye

Distance between markers

The range for which the ARDrone's camera can detect Aruco markers is about 3 meters, so poles and landmarks should be placed in a way that the drone always detects at least 1 Aruco marker and can correctly estimate its position.

Contact

We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page.

You can also contact the team support at the following address: aerostack.vision4uav@gmail.com

Simulated Quadrotor

Edit New Page

Martin Molina edited this page on 14 Jun 2016 · 1 revision

Here you can find an example that describes how to perform a simulated mission. This example assumes that the vehicle is simulated.

This example uses the following configuration files:

```
./config/drone01/*
```

Execute the following launcher:

```
./launchers/simulated_quadrotor.sh
```


Pages 37

Introduction

- What is Aerostack?
- Getting Started
- License
- List of distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Catalog of Behaviors
- Memory of Beliefs

Mission Plan Specification

- TML Language
- Behavior tree
- Python Language

Tutorials [Beginners]

- Prepare the Mission:
  - Configure Map
  - Specify Mission Tasks
  - Configure Launcher
- Execute the Mission:
  - Launch Real Flight
  - Practical Tips for Real Flights
- Mission examples:
  - Simulated Quadrotor
  - AR Drone and Visual Markers

Tutorials [Advanced]

- How to program a new behavior
- Setup the Communication Network
- Wi Fi Communication with Multiple Drones
- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

More information

- Publications
- Vision & Aerial Robotics:
  - First prize at IMAV 2013
  - Stack pre-release 2013

Clone this wiki locally

https://github.com/Vision4U

Clone in Desktop

Contact

We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page.

You can also contact the team support at the following address: aerostack.vision4uav@gmail.com

AR Drone and Visual Markers

Martin Molina edited this page on 14 Jun 2016 · 2 revisions

Here you can find an example that describes how to perform a complete mission with a real flight. This example assumes that the self-localization uses Aruco visual markers.

This example uses the following configuration files:

```
./config/drone02/*
```

Execute the following launcher:

```
./launchers/ardrone_basic.sh
```


Pages 37

Introduction

- What is Aerostack?
- Getting Started
- License
- List of distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Catalog of Behaviors
- Memory of Beliefs

Mission Plan Specification

- TML Language
- Behavior tree
- Python Language

Tutorials [Beginners]

- Prepare the Mission:
  - Configure Map
  - Specify Mission Tasks
  - Configure Launcher
- Execute the Mission:
  - Launch Real Flight
  - Practical Tips for Real Flights
- Mission examples:
  - Simulated Quadrotor
  - AR Drone and Visual Markers

Tutorials [Advanced]

- How to program a new behavior
- Setup the Communication Network
- Wi Fi Communication with Multiple Drones
- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

More information

- Publications
- Vision & Aerial Robotics:
  - First prize at IMAV 2013
  - Stack pre-release 2013

Clone this wiki locally

https://github.com/Vision4U

Clone in Desktop

Contact

We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page.

You can also contact the team support at the following address: aerostack.vision4uav@gmail.com

How to Program a New Behavior

Alberto Camporredondo edited this page on 15 Nov 2017 · 2 revisions

Aerostack includes a behavior-based execution system that helps simplify the configuration of robots' mission plans. Each behavior is implemented as a separate process (see how to program a new process) that uses the class BehaviorProcess that, in its turn, is subclass of the class RobotProcess.

Steps to create a new behavior

Step 1. Create a new package and change the directory structure

The name of the package should be `behavior_<name>`. As an example, the package for the behavior `TAKE_OFF` is called `behavior_take_off`.

Then, the structure of the package created has to be like the following lines:

```
behavior_take_off/                    -- Name of the package
      CMakeLists.txt                  -- CMakeLists.txt file
      package.xml                     -- Package manifest
      launch/
        my_process_process.launch     -- Launcher
      src/
        include/
          behavior_take_off.h         -- Header file
        source/
          behavior_take_off.cpp       -- C++ file
          behavior_take_off_main.cpp  -- C++ main file
```

As you can see, there is a directory called `launch` which contains a file with topics names or other parameters. You can find an example of this file at the end of this tutorial.

Step 2. Edit the files CMakeLists and package.xml

In order to develop a new behavior, the class `BehaviorProcess` must be inherited. To inherit the class in C++, the file `CMakeLists.txt` and `package.xml` must have the following lines:

File *CMakeLists.txt*

```
find_package(catkin REQUIRED
COMPONENTS
roscpp
behavior_process
droneMsgsROS
)

catkin_package(
INCLUDE_DIRS ${BEHAVIOR_TAKE_OFF_INCLUDE_DIR}
CATKIN_DEPENDS
roscpp
behavior_process
droneMsgsROS
)
```

File *package.xml*

```
<build_depend>behavior_process</build_depend>
<build_depend>droneMsgsROS</build_depend>
```


Pages 37

Introduction

- What is Aerostack?
- Getting Started
- License
- List of distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Catalog of Behaviors
- Memory of Beliefs

Mission Plan Specification

- TML Language
- Behavior tree
- Python Language

Tutorials [Beginners]

- Prepare the Mission:
  - Configure Map
  - Specify Mission Tasks
  - Configure Launcher
- Execute the Mission:
  - Launch Real Flight
  - Practical Tips for Real Flights
- Mission examples:
  - Simulated Quadrotor
  - AR Drone and Visual Markers

Tutorials [Advanced]

- How to program a new behavior
- Setup the Communication Network
- Wi Fi Communication with Multiple Drones
- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

More information

- Publications
- Vision & Aerial Robotics:
  - First prize at IMAV 2013
  - Stack pre-release 2013

Clone this wiki locally

```
<run_depend>behavior_process</run_depend>
<run_depend>droneMsgsROS</run_depend>
```

You can find an example of the `package.xml` and `CMakeLists.txt` files at the end of the tutorial.

Step 3. Create the header and implementation files

To use the class BehaviorProcess the header file needs to have the following structure:

```
class BehaviorTakeOff : public BehaviorProcess
{
...
};
```

Then, the following functions should be implemented:

```
private: //BehaviorProcess
void ownSetUp();
void ownStart();
void ownRun();
void ownStop();
std::tuple<bool, std::string> ownCheckSituation();
```

These functions give a behavior the ability to:

- Prepare the execution (set up)
- Start the execution (start)
- Stop the execution (stop)
- Supervise the execution (run)
- Verify if the behavior can be executed in that situation (check situation)

Step 4. Include the behavior in the catalog

The behavior catalog is the file `behavior_catalog.yaml`. The catalog includes information about each behavior to describe how to use it. This information includes the allowable arguments of each behavior, the compatibility with other behaviors, etc.

It is important to include the the information of a new behavior in the catalog in order to be able to check its format and to coordinate its execution with the execution of other behaviors.

There are some important aspects to consider in this file:

- NO tabs should be used. Spaces are used instead (two spaces).
- Capabilities represent a group of processes that need to be activated before the behavior starts.
- Incompatibilities represent inconsistencies among behaviors.

Some examples of the behavior catalog:

- An example of a behavior with no capabilities associated and no incompatibilities is written in the following lines:

```
- behavior: TAKE_OFF
```

- An example of a behavior with no capabilities associated and some incompatibilities is written in the following lines:

https://github.com/Vision4U
Clone in Desktop

```
behavior_descriptors:
  - behavior: TAKE_OFF
    incompatibilities: motion

  - behavior: LAND
    incompatibilities: motion

behavior_lists:
  - name: motion
    behaviors:
      - TAKE_OFF
      - LAND
```

- An example of a behavior with capabilities associated and some incompatibilities is written in the following lines:

```
behavior_descriptors:
  - behavior: TAKE_OFF
    incompatibilities: motion

  - behavior: LAND
    incompatibilities: motion

  - behavior: GO_TO_POINT
    capabilities: TRAJECTORY

behavior_lists:
  - name: motion
    behaviors:
      - TAKE_OFF
      - LAND

capability_descriptors:
  - capability: TRAJECTORY
    process_sequence: [droneTrajectoryController]
```

NOTE: Every process written inside the tag process_sequence should be the name of the service that starts the process. For example, for the process `droneTrajectoryController`, the service `/droneX/droneTrajectoryController/start` has to be called. So, inside the catalog you don't have to write the whole path.

Step 5. Launch the behavior within Aerostack

After the behavior is described inside the catalog file, you just need to add the behavior created inside the launcher script. For that reason, go to the launchers directory and edit the script `simulated_quadrotor_basic.sh` (if you want to use the simulation) or `ardrone_basic.sh` (if you want to fly the real drone) with the following lines:

```
`#-----------------------------------------------------------------------------------------
`# Behavior TakeOff
`#-----------------------------------------------------------------------------------------
--tab --title "Behavior TakeOff" --command "bash -c \"
roslaunch behavior_take_off behavior_take_off.launch --wait \
    drone_id_namespace:=drone$NUMID_DRONE \
    drone_id:=$NUMID_DRONE \
    my_stack_directory:=${AEROSTACK_STACK};
exec bash\"" \
```

When Aerostack is loaded, if you want to start the behavior created, just call the ROS service `activate_behavior` with the name of the behavior. An example can be found below these lines.

```
rosservice call /drone1/activate_behavior "behavior:
name: 'TAKE_OFF'
arguments: ''"
```

One important point is the name of the behavior, it needs to be the same as the one written in the catalog and without the `behavior_`.

Example behaviors

The following lines represents the definition file, the implementation file of a behavior, the package file and the CMakeLists file.

header file:

```
#ifndef TAKE_OFF_H
#define TAKE_OFF_H

//System
#include <string>
#include <tuple>
// ROS
#include <ros/ros.h>
#include "std_srvs/Empty.h"
#include <geometry_msgs/Vector3Stamped.h>

// Aerostack msgs
#include <droneMsgsROS/BehaviorEvent.h>
#include <droneMsgsROS/dronePose.h>
#include <droneMsgsROS/droneCommand.h>
#include <droneMsgsROS/setInitDroneYaw_srv_type.h>
#include<droneMsgsROS/ConsultBelief.h>
//Aerostack libraries
#include <behavior_process.h>


class BehaviorTakeOff:public BehaviorProcess
{
  //Constructor
public:
  BehaviorTakeOff();
  ~BehaviorTakeOff();

private:
  ros::NodeHandle node_handle;

  //Congfig variables
  std::string drone_id;
  std::string drone_id_namespace;
  std::string my_stack_directory;
  std::string behavior_name_str;
  std::string estimated_pose_str;
  std::string controllers_str;
  std::string rotation_angles_str;
  std::string initialize_yaw_str;
  std::string execute_query_srv;

  //Subscriber---
  ros::Subscriber estimated_pose_sub;
  ros::Subscriber rotation_angles_sub;
  ros::Publisher controllers_pub;
  ros::ServiceClient initialize_yaw_cli;
  ros::ServiceClient query_client;
  //Message
  droneMsgsROS::dronePose estimated_pose_msg;
  droneMsgsROS::dronePose static_pose;
  geometry_msgs::Vector3Stamped rotation_angles_msg;
  //Timer staticity_timer;
  bool first_position;
  bool is_finished;

private: //BehaviorProcess
  void ownSetUp();
```

```
  void ownStart();
  void ownRun();
  void ownStop();
  std::tuple<bool, std::string> ownCheckSituation();

private:
  void endingImplementation();
  bool monitor();

public: //Callbacks
  void estimatedPoseCallBack(const droneMsgsROS::dronePose&);
  void rotationAnglesCallback(const geometry_msgs::Vector3Stamped&);
};


#endif
```

And for the implementation file:

```
#include "../include/behavior_take_off.h"

BehaviorTakeOff::BehaviorTakeOff() : BehaviorProcess()
{

}

BehaviorTakeOff::~BehaviorTakeOff()
{

}


void BehaviorTakeOff::ownSetUp()
{
  node_handle.param<std::string>("drone_id", drone_id, "1");
  node_handle.param<std::string>("drone_id_namespace", drone_id_namespace, "drone"+drone_
  node_handle.param<std::string>("my_stack_directory", my_stack_directory,
            "~/workspace/ros/quadrotor_stack_catkin/src/quadrotor_stack");

  estimated_pose_str = "estimated_pose";
  node_handle.param<std::string>("controllers_topic", controllers_str, "command/high_leve
  node_handle.param<std::string>("rotation_angles_topic", rotation_angles_str, "rotation_
  node_handle.param<std::string>("initialize_yaw_srv", initialize_yaw_str,
  "droneOdometryStateEstimator/setInitDroneYaw");
  node_handle.param<std::string>("consult_belief",execute_query_srv,"consult_belief");
}

void BehaviorTakeOff::ownStart()
{
  /*Initialize topics*/
  estimated_pose_sub = node_handle.subscribe(estimated_pose_str, 1000, &BehaviorTakeOff::
  rotation_angles_sub = node_handle.subscribe(rotation_angles_str, 1000, &BehaviorTakeOff
  controllers_pub = node_handle.advertise<droneMsgsROS::droneCommand>(controllers_str, 1,
  initialize_yaw_cli = node_handle.serviceClient<droneMsgsROS::setInitDroneYaw_srv_type>(
  query_client = node_handle.serviceClient <droneMsgsROS::ConsultBelief> (execute_query_s

/*behavior implementation*/
  std_msgs::Header header;
  header.frame_id = "behavior_take_off";

  droneMsgsROS::droneCommand msg;
  msg.header = header;
  msg.command = droneMsgsROS::droneCommand::TAKE_OFF;
  controllers_pub.publish(msg);
  estimated_pose_msg.z = 0;
  first_position = false;
  is_finished = false;
}

void BehaviorTakeOff::ownRun()
```

```
{
  if(!is_finished){
    monitor();
  }
}

//ownRun
void BehaviorTakeOff::ownStop()
{
  std::cout << "ownStop" << std::endl;
  std_msgs::Header header;
  header.frame_id = "behavior_land";

  droneMsgsROS::droneCommand msg;
  msg.header = header;
  msg.command = droneMsgsROS::droneCommand::HOVER;
  controllers_pub.publish(msg);

  estimated_pose_sub.shutdown();
  rotation_angles_sub.shutdown();
  controllers_pub.shutdown();
  initialize_yaw_cli.shutdown();
}

std::tuple<bool,std::string> BehaviorTakeOff::ownCheckSituation()
{
  droneMsgsROS::ConsultBelief query_service;
  std::ostringstream capturador;
  capturador << "battery_level(self,LOW)";
  std::string query(capturador.str());
  query_service.request.query = query;
  query_client.call(query_service);
  if(query_service.response.success)
  {
    return std::make_tuple(false,"Error: Battery low, unable to perform action");
  }
  std::ostringstream capturador2;
  capturador2<<"flight_state(self,FLYING)";
  std::string query2(capturador2.str());
  query_service.request.query = query2;
  query_client.call(query_service);
  if(query_service.response.success)
  {
    return std::make_tuple(false,"Error: Already flying");
  }

  return std::make_tuple(true,"");
}

//Private functions
bool BehaviorTakeOff::monitor()
{
  double precision_take_off = 0.1;

  //Check achievement
  if(std::abs(std::abs(estimated_pose_msg.z) - 0.7) < precision_take_off){
    BehaviorProcess::setFinishEvent(droneMsgsROS::BehaviorEvent::GOAL_ACHIEVED);
    BehaviorProcess::setFinishConditionSatisfied(true);
    endingImplementation();
    is_finished = true;
    return true;
  }

  //Check timeout
  if(timerIsFinished()){
    BehaviorProcess::setFinishEvent(droneMsgsROS::BehaviorEvent::TIME_OUT);
    BehaviorProcess::setFinishConditionSatisfied(true);
    endingImplementation();
    is_finished = true;
    return true;
  }
```

```
  return false;
}

void BehaviorTakeOff::endingImplementation()
{
  droneMsgsROS::setInitDroneYaw_srv_type init_yaw_msg;
  init_yaw_msg.request.yaw_droneLMrT_telemetry_rad = (rotation_angles_msg.vector.z)*(M_PI
  initialize_yaw_cli.call(init_yaw_msg);

  droneMsgsROS::droneCommand msg;
  msg.command = droneMsgsROS::droneCommand::HOVER;
  controllers_pub.publish(msg);
}

//Custom topic Callbacks
  void BehaviorTakeOff::estimatedPoseCallBack(const droneMsgsROS::dronePose& message){
  if(!first_position)
    static_pose = message;
  estimated_pose_msg = message;
}

void BehaviorTakeOff::rotationAnglesCallback(const geometry_msgs::Vector3Stamped& message
  rotation_angles_msg = message;
}
```

Finally, the main file should look like the one below:

```
#include "../include/behavior_take_off.h"

int main(int argc, char** argv){
  ros::init(argc, argv, ros::this_node::getName());

  std::cout << ros::this_node::getName() << std::endl;

  BehaviorTakeOff behavior;
  behavior.setUp();
  ros::Rate rate(10);
  while(ros::ok()){
    behavior.run();
    ros::spinOnce();
    rate.sleep();
  }
  return 0;
}
```

In order to compile and to launch the program, the following file have to be inside the behavior created.

package.xml:

```
<?xml version="1.0"?>
<package>
  <name> behavior_take_off </name>
  <version>0.1.0</version>
  <description>The behavior_take_off package</description>

  <author email="test@test.com">Author</author>
  <maintainer email="test@test.com">Maintainer</maintainer>

  <license>GPLv3</license>

  <!-- The *_depend tags are used to specify dependencies -->
  <!-- Dependencies can be catkin packages or system dependencies -->
  <!-- Examples: -->
  <!-- Use build_depend for packages you need at compile time: -->
  <!--   <build_depend>message_generation</build_depend> -->
  <!-- Use buildtool_depend for build tool packages: -->
  <!--   <buildtool_depend>catkin</buildtool_depend> -->
```

```
  <!-- Use run_depend for packages you need at runtime: -->
  <!--   <run_depend>message_runtime</run_depend> -->
  <!-- Use test_depend for packages you need only for testing: -->
  <!--   <test_depend>gtest</test_depend> -->
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>std_msgs</build_depend>
  <build_depend>drone_process</build_depend>
  <build_depend>behavior_process</build_depend>
  <build_depend>droneMsgsROS</build_depend>
  <build_depend>geometry_msgs</build_depend>

  <run_depend>roscpp</run_depend>
  <run_depend>std_msgs</run_depend>
  <run_depend>drone_process</run_depend>
  <run_depend>behavior_process</run_depend>
  <run_depend>droneMsgsROS</run_depend>
  <run_depend>geometry_msgs</run_depend>

  <!-- The export tag contains other, unspecified, tags -->
  <export>
    <!-- Other tools can request additional information be placed here -->

  </export>
</package>
```

CMakeLists.txt:

```
cmake_minimum_required(VERSION 2.8.3)
project(behavior_take_off)

### Use version 2011 of C++ (c++11). By default ROS uses c++98
#see: http://stackoverflow.com/questions/10851247/how-to-activate-c-11-in-cmake
#see: http://stackoverflow.com/questions/10984442/how-to-detect-c11-support-of-a-
compiler-with-cmake
add_definitions(-std=c++11)

# Directories definition
set(BEHAVIOR_TAKE_OFF_SOURCE_DIR
  src/source
)

set(BEHAVIOR_TAKE_OFF_INCLUDE_DIR
  src/include
)

# Files declaration
set(BEHAVIOR_TAKE_OFF_SOURCE_FILES
  ${BEHAVIOR_TAKE_OFF_SOURCE_DIR}/behavior_take_off.cpp
  ${BEHAVIOR_TAKE_OFF_SOURCE_DIR}/behavior_take_off_main.cpp
)

set(BEHAVIOR_TAKE_OFF_HEADER_FILES
  ${BEHAVIOR_TAKE_OFF_INCLUDE_DIR}/behavior_take_off.h
)

### Find catkin macros and libraries
## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)
## is used, also find other catkin packages
find_package(catkin REQUIRED
  COMPONENTS
  roscpp
  std_msgs
  behavior_process
  droneMsgsROS
  geometry_msgs
)

####################################
## catkin specific configuration ##
```

```
#####################################
## INCLUDE_DIRS: uncomment this if you package contains header files
## LIBRARIES: libraries you create in this project that dependent projects also need
## CATKIN_DEPENDS: catkin_packages dependent projects also need
## DEPENDS: system dependencies of this project that dependent projects also need
catkin_package(
  INCLUDE_DIRS ${BEHAVIOR_TAKE_OFF_INCLUDE_DIR}
  CATKIN_DEPENDS
  roscpp
  std_msgs
  drone_process
  behavior_process
  droneMsgsROS
  geometry_msgs
)

###########
## Build ##
###########
include_directories(
  ${BEHAVIOR_TAKE_OFF_INCLUDE_DIR}
  ${BEHAVIOR_TAKE_OFF_SOURCE_DIR}
)
include_directories(
  ${catkin_INCLUDE_DIRS}
)

add_library(behavior_take_off_lib ${BEHAVIOR_TAKE_OFF_SOURCE_FILES}
${BEHAVIOR_TAKE_OFF_HEADER_FILES})
add_dependencies(behavior_take_off_lib ${catkin_EXPORTED_TARGETS})
target_link_libraries(behavior_take_off_lib ${catkin_LIBRARIES})

add_executable(behavior_take_off
${BEHAVIOR_TAKE_OFF_SOURCE_DIR}/behavior_take_off_main.cpp)
add_dependencies(behavior_take_off ${catkin_EXPORTED_TARGETS})
target_link_libraries(behavior_take_off behavior_take_off_lib)
target_link_libraries(behavior_take_off ${catkin_LIBRARIES})
```

behavior_take_off.launch:

```
<!-- Launches: behavior_take_off -->
<launch>

  <!-- Basic data -->
  <arg name="drone_id_namespace"                    default="drone1" />
  <arg name="drone_id"                              default="1" />
  <arg name="my_stack_directory"                    default="$(env AEROSTACK_STACK)" />
  <arg name="behavior_name"                         default="TAKE_OFF" />
  <arg name="behavior_mode"                         default="" />

  <!-- Custom topics & services-->
  <arg name="estimated_pose_topic"                default="estimated_pose" />
  <arg name="controllers_topic"                     default="command/high_level" />
  <arg name="rotation_angles_topic"                 default="rotation_angles" />
  <arg name="initialize_yaw_srv"                    default="droneOdometryStateEstimator/
  <arg name="consult_belief"                         default="consult_belief"/>

  <group ns="$(arg drone_id_namespace)">

    <!-- Basic data -->
    <param name="~drone_id_namespace"                   value="$(arg drone_id_namespace)
    <param name="~drone_id"                             value="$(arg drone_id)" type="st
    <param name="~my_stack_directory"                   value="$(arg my_stack_directory)
    <param name="~behavior_name"                        value="$(arg behavior_name)" typ
    <param name="~behavior_mode"                        value="$(arg behavior_mode)" typ

    <!-- Custom topics & services -->
    <param name="~estimated_pose_topic"                 value="$(arg estimated_pose_topi
    <param name="~controllers_topic"                    value="$(arg controllers_topic)"
    <param name="~rotation_angles_topic"                value="$(arg rotation_angles_top
```

```
    <param name="~initialize_yaw_srv"                    value="$(arg initialize_yaw_srv)
    <param name="~consult_belief"                         value="$(arg consult_belief) "

    <node name="behavior_take_off" pkg="behavior_take_off" type="behavior_take_off" outpu

    <!--<test test-name="SequencerTest" pkg="behavior_take_off" type="SequencerTest"/>-->

  </group>

</launch>
```

Contact

We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page.

You can also contact the team support at the following address: aerostack.vision4uav@gmail.com

How to Program a New Process

Martin Molina edited this page on 8 Nov 2017 · 1 revision

Any process (ROS node) that belongs to Aerostack must be correctly supervised by the process monitor. In order to facilitate this supervision, the process must be subclass of the class RobotProcess and it must be programmed following certain conventions. This section describes with an example how to program a new process for Aerostack in C++ language.

Example Description

The purpose of this example is to illustrate the common steps that the developer must perform to build a new process. So, the example process is a very simplified case. The example illustrates certain details that must be taking into account by the programmer such as how to handle error messages or other aspects related to the communication with the supervision system. We do not describe here specific algorithms for this process.

In the example, we make the following assumptions:

- The process is called **my process**.
- The process is part of the **planning system**.
- The process uses as input a topic called **my topic**. This topic includes a robot position (coordinates x, y).
- The specific algorithms for this process are externally encapsulated in a separate class called **my class**. The algorithms included in this class are independent of ROS. This division is a good practice that separates ROS dependent code from ROS independent code and, therefore, promotes the reusability of ROS independent code in other programming contexts.

1. Folder Organization

The folder organization corresponds to catking packages (see creating ROS packages) within an system of the Aerostack architecture. In our example, the folder organization is the following:

```
planning_system/                        -- System to which our process belongs
  my_process/                           -- Name of my process
    my_class/
      CMakeLists.txt                    -- CMakeLists.txt file
      package.xml                       -- Package manifest
      src/
        include/
          my_class.h                    -- Header file
        source/
          my_class.cpp                  -- C++ file
        test/
          my_class_test.cpp             -- C++ program to test the class
          ...                           -- Other files (xml, csv, etc.) to test the
class
    my_process_process/
      CMakeLists.txt                    -- CMakeLists.txt file
      package.xml                       -- Package manifest
      README.md                         -- Process description (with ROS topics and
services)
      launch/
        my_process_process.launch       -- Launcher
      msg/                              -- Messages and service types
      src/
        include/
```


Pages 37

Introduction
- What is Aerostack?
- Getting Started
- License
- List of distributions

Components
- Aerostack Architecture
- Graphical User Interface
- Catalog of Behaviors
- Memory of Beliefs

Mission Plan Specification
- TML Language
- Behavior tree
- Python Language

Tutorials [Beginners]
- Prepare the Mission:
  - Configure Map
  - Specify Mission Tasks
  - Configure Launcher
- Execute the Mission:
  - Launch Real Flight
  - Practical Tips for Real Flights
- Mission examples:
  - Simulated Quadrotor
  - AR Drone and Visual Markers

Tutorials [Advanced]
- How to program a new behavior
- Setup the Communication Network
- Wi Fi Communication with Multiple Drones
- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

More information
- Publications
- Vision & Aerial Robotics:
  - First prize at IMAV 2013
  - Stack pre-release 2013

Clone this wiki locally

```
          my_process_process.h         -- Header file (with the suffix "process")
        source/
          my_process_process.cpp       -- ROS dependent C++ program (with the suffix
"process")
          my_process_process_main.cpp  -- Program with the main function (with the
suffix "main")
        test/
          my_process_process_test.cpp  -- C++ program to test the process (with the
suffix "test")
          ...                          -- Other files (xml, csv, etc.) to test the
process
```

2. Header File

This section shows the header file corresponding to the MyProcess class. The complete content of the header file is the following:

```
#ifndef MY_PROCESS
#define MY_PROCESS

#include "robot_process.h"
#include "my_class.h"
#include <cvg_stack_msgs/position.h>

/*!*****************************************************************************
 *  \class     MyProcess
 *
 *  \brief     This is an example that shows how a process should be programmed.
 *
 *  \details   This process receives a position message through an input topic.
 *             If either the x or y values are negative, an error is reported.
 *             This process uses algorithms defined by the MyClass class.
 *
 *  \authors   Martin Molina, Enrique Ortiz
 *
 *  \copyright Copyright 2016 Universidad Politecnica de Madrid (UPM)
 *
 *     This program is free software: you can redistribute it and/or modify
 *     it under the terms of the GNU General Public License as published by
 *     the Free Software Foundation, either version 3 of the License, or
 *     (at your option) any later version.
 *
 *     This program is distributed in the hope that it will be useful,
 *     but WITHOUT ANY WARRANTY; without even the implied warranty of
 *     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 *     GNU General Public License for more details.
 *
 *     You should have received a copy of the GNU General Public License
 *     along with this program. If not, see http://www.gnu.org/licenses/.
 *****************************************************************************/
class MyProcess: public RobotProcess
{
public:
  //! Constructor. \details Same arguments as the ros::init function.
  MyProcess();

  //! Destructor.
  ~MyProcess();

/*!*****************************************************************************
 * \details The function ownSetUp() is executed to setup everything that
 *          the process needs at the beginning such as read
 *          configuration files, read ROS parameters, etc.
 *****************************************************************************/
  void ownSetUp();

/*!*****************************************************************************
 * \details The function ownStart() prepares the process to start running.
```

https://github.com/Vision4U

Clone in Desktop

```
 *          Connects topics and services and set the appropiate values.
 ***************************************************************************/
  void ownStart();

/*!***************************************************************************
 * \details The function ownStop() perform shutdown of topics and services and
 *          reset values of variables.
 ***************************************************************************/
  void ownStop();

/*!***************************************************************************
 * \details The function ownRun() the main body of execution to perform in the
 *          running state. Only used when the process is synchronous.
 ***************************************************************************/
  void ownRun();

private:
  //! ROS NodeHandler used to manage the ROS communication
  ros::NodeHandle n;

  //! ROS subscriber handler used to receive messages.
  ros::Subscriber my_topic_subscriber;

  //! Topic name used to subscribe
  std::string my_topic;

  int x_position;   //!< Position x
  int y_position;   //!< Position y

  //! Ros independent class
  MyClass my_class;

  //! Function launched when this process receives a message from my_topic
  void myTopicCallback(const cvg_stack_msgs::position::ConstPtr& position);

  //! This function will be called in the myTopicCallback
  void myFunction(int* x, int* y);

  //! This function is called periodically, between callbacks.
  void myOtherFunction();
};
#endif
```

The process is programmed using a C++ class called *MyProcess*. This class must be subclass of RobotProcess to inherit the required functionality to be monitored by the supervision system.

The following paragraphs describe in more detail the different parts of the header file.

2.1. Dependence on Header Files

At the beginning of the file the following sentences are written:

```
#include "robot_process.h"
#include "my_class.h"
#include <cvg_stack_msgs/position.h>
```

Since MyProcess is subclass of RobotProcess, the header file robot_process has to be included. The my_class.h header class is also included to be able to use the class MyClass. The cvg_stack_msgs/position header file allows our class to receive position data.

2.2. Class Declaration, Constructor and Destructor

The declaration of the MyProcess class as subclass of RobotProcess, with its constructor and destructor functions is the following:

```
class MyProcess: public RobotProcess
```

```
{
  MyProcess();
  ~MyProcess();
  ...
}
```

The constructor and destructor functions are always needed in C++ language.

2.3. Own Functions

The declaration of the specific functions (own functions) for setUp(), start() and stop() is the following:

```
void ownSetUp();
void ownStart();
void ownStop();
void ownRun();
```

These functions must be defined at every subclass of RobotProcess. The body of these functions with their algorithms are written in the cpp file.

2.4. Node Handle

The use of RobotProcess class abstracts certain details of the use of ROS. However, it is still needed that the programmer declares a node handle in the header file. The ROS handler is defined in the following way:

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

2.5. Topic Subscription

The following part of the header file is required for topic subscription:

```
 //! ROS subscriber handler used to receive messages.
ros::Subscriber my_topic_subscriber;

//! Topic name used to subscribe
std::string my_topic;

int x_position;   //!< Position x
int y_position;   //!< Position y

//! Function launched when this process receives a message from my_topic. The message i
void myTopicCallback(const cvg_stack_msgs::position::ConstPtr& position);
```

2.6. Function Definition

Different functions are declared.

```
//! This function will be called in the myTopicCallback
void myFunction(int* x, int* y);

//! This function is called periodically, between callbacks.
void myOtherFunction();
```

3. The CPP File

This section shows the CPP file corresponding to MyProcess class. The complete content of the CPP file is the following:

```
#include "my_process_process.h"

//Constructor
MyProcess::MyProcess():RobotProcess()
{
  MyClass my_class();

  if (!n.getParam("my_topic_name", my_topic))
    my_topic = "my_topic";
}

//Destructor
MyProcess::~MyProcess() {}

//myTopicCallback
void MyProcess::myTopicCallback(const cvg_stack_msgs::position::ConstPtr& position)
{
  //Input data validation
  if (position->x < 0 || position->y < 0)
  {
    notifyError(InvalidInputData,0,
                "myTopicCallback()",
                "Positive values expected for coordinates x and y");
  }
  else
  {
    ROS_INFO("Received from %s: x=%d, y=%d",
             my_topic.c_str(), position->x, position->y);
    x_position = position->x;
    y_position = position->y;

    try
    {
      myFunction(&x_position, &y_position);
    }
    catch (std::logic_error& exception)
    {
      //This is how the exception is catched.
      notifyError(SafeguardFatalError,0,"myFunction()",exception.what());
      //This is a fatal error that can not be recovered.
      std::exit(EXIT_FAILURE);
    }
  }
}

//myFunction - any function for illustration purposes
void MyProcess::myFunction(int* x, int* y)
{
  ...
}

void MyProcess::myOtherFunction()
{
  ...
}

//ownSetUp()
void MyProcess::ownSetUp()
{
  x_position = 0;
  y_position = 0;
}

//ownStart()
void MyProcess::ownStart()
{
  my_topic_subcriber = n.subscribe(my_topic, 1000, &MyProcess::myTopicCallback, this);
}

//ownStop()
```

```
void MyProcess::ownStop()
{
  x_position = 0;
  y_position = 0;
  my_topic_subscriber.shutdown();
}

//ownRun()
void MyProcess::ownRun()
{
  myOtherFunction();
}
```

3.1. Constructor and Destructor

The code of the body for both constructor and destructor functions is the following:

```
//Constructor
MyProcess::MyProcess():RobotProcess()
{
  MyClass my_class();

  if (!n.getParam("my_topic_name", my_topic))
    my_topic = "my_topic";
}

//Destructor
MyProcess::~MyProcess() {}
```

We use the ROS parameter server with the function getParam() to obtain the drone id value, as well as the name of the topic used to receive position messages.

3.2. Input Data Validation

The code to validate input data with the corresponding message error is the following:

```
  //Input data validation
  if (position->x < 0 || position->y < 0)
  {
    notifyError(InvalidInputData,0,
                "myTopicCallback()",
                "Positive values expected for coordinates x and y");
  }
```

The myTopicCallback function, which has the main logic of the process, checks the received position. If this position is noticed to be invalid, an invalid input data error is sent to the performance monitor.

3.3. Safeguard Error

The following example shows how the process detects and notifies a safeguard error, when the error cannot be recovered. To illustrate how to catch this type of error, we have the following piece of code:

```
    try
    {
      myFunction(&x_position, &y_position);
    }
    catch (std::logic_error& exception)
    {
      //This is how the exception is catched.
      notifyError(SafeguardFatalError,0,"myFunction()",exception.what());
      //This is a fatal error that can not be recovered.
```

```
    std::exit(EXIT_FAILURE);
  }
```

In this case, the error cannot be solved during process execution. That is why the type of the error notified is SafeguardFatalError. To detect these errors the exceptions are also used and, because the program cannot recover, it stops by using the std::exit() function.

3.4. Own Functions

The following piece of code illustrates how to write own functions:

```
//ownSetUp()
void MyProcess::ownSetUp()
{
  x_position = 0;
  y_position = 0;
}

//ownStart()
void MyProcess::ownStart()
{
  my_topic_subcriber = n.subscribe(my_topic, 1000, &MyProcess::myTopicCallback, this);
}

//ownStop()
void MyProcess::ownStop()
{
  x_position = 0;
  y_position = 0;
  my_topic_subscriber.shutdown();
}

//ownRun()
void MyProcess::ownRun()
{
  myOtherFunction();
}
```

The ownSetUp() function is used to initialize the variables.

The ownStop() function resets the values of variables and disconnect from the topic using shutdown.

The ownStart() function connects to the topic my_topic.

The ownRun() function is called periodically when the process is running.

4. The Main File

This section shows the CPP file corresponding to the main function. A possible way to write a main function is:

```
#include "my_process.h"

int main(int argc, char **argv)
{
  ros::init(argc, argv, ros::this_node::getName());
  MyProcess my_process;
  my_process.setUp();
  try
  {
    my_process.start();
  }
  catch (std::exception& exception)
  {
    notifyError(SafeguardRecoverableError,0,"ownStart()",exception.what());
```

```
    my_process.stop();  // The process will have to be started by a service call from ano
  }

  ros::Rate loop_rate(10);
  while(ros::ok())
  {
    ros::spinOnce();
    my_process.run();
    loop_rate.sleep();
  }

  return 0;
}
```

Other different execution modes can be programmed. For example, we can distinguish between two different execution modes:

- *Immediate start*. The process starts automatically after set up, without the intervention of any other process.
- *Subordinated start*. After set up, this process waits until it is started by another process (using a ROS service).

The previous example illustrates a process with immediate start. A process with subordinated start (and synchronous execution) has the following body for the main() function:

```
int main(int argc, char **argv)
{
  ros::init(argc, argv, ros::this_node::getName());
  MyProcess my_process;
  my_process.setUp();

  ros::Rate loop_rate(10);
  while(ros::ok())
  {
    ros::spinOnce();
    my_process.run();
    loop_rate.sleep();
  }

  return 0;
}
```

A process with subordinated start (and asynchronous execution) has the following body for the main() function:

```
int main(int argc, char **argv)
{
  ros::init(argc, argv, ros::this_node::getName());
  MyProcess my_process;
  my_process.setUp();

  ros::spin();   // Waits for the service start

  return 0;
}
```

Contact

We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page.

You can also contact the team support at the following address: aerostack.vision4uav@gmail.com

The Class RobotProcess

Edit New Page

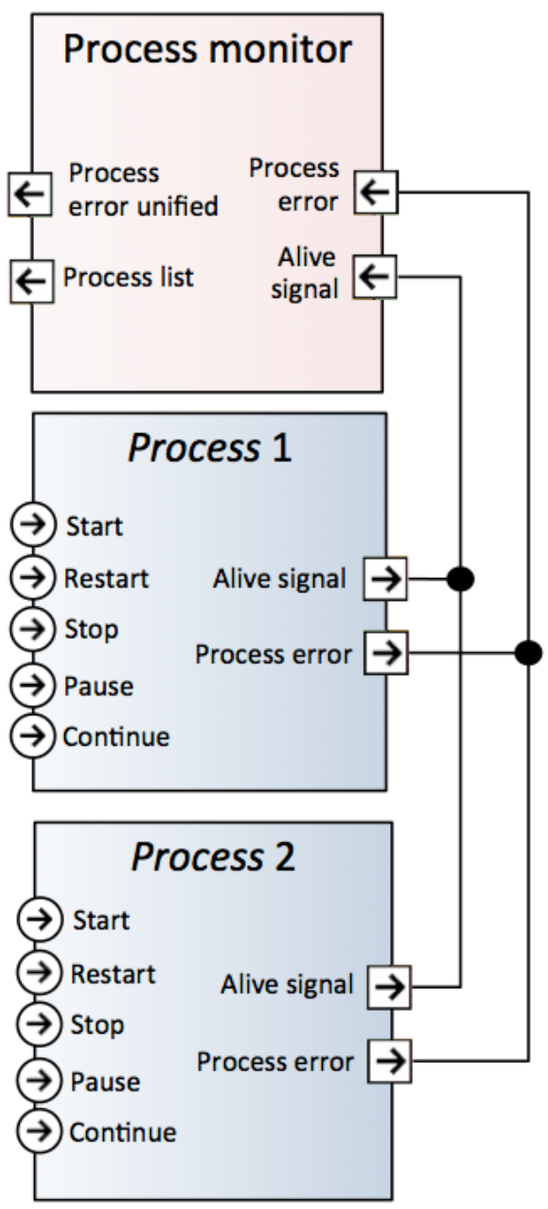
Martin Molina edited this page on 8 Nov 2017 · 3 revisions

The class RobotProcess has been designed to support the development of processes in Aerostack. The following sections describe the main features provided by this class.

1. Alive Signal for Process Monitoring

The supervision system in Aerostack monitors the correct execution processes. To facilitate this, each process generates automatically a periodic alive signal that allows the supervision system to detect unexpected stops.

The class RobotProcess provides the mechanism to generate the alive signal. This is done by creating a thread that is responsible of the periodical alive signal sending. By using this thread, the generation of the alive signals does no produce undesired delays in the implemented algorithm of the process. Every process sends alive signals at the same frequency, making more reliable a process stop detection.



2. Controlling the Process Execution

The class RobotProcess provides a set of services that allow to control the execution of the process by other processes (start, restart, stop, pause and continue).

In general, the execution of a process has the following characteristics:

- *Set up*. Before the process can be executed, the process must be configured with an initial set of parameter values and/or configuration files.
- *Interprocess communication*. When the process starts its execution it is connected to the data channels for inter-process communication in a network of processes through an external interface of input/output variables (ROS topics) and ROS services.

Pages 37

Introduction
- What is Aerostack?
- Getting Started
- License
- List of distributions

Components
- Aerostack Architecture
- Graphical User Interface
- Catalog of Behaviors
- Memory of Beliefs

Mission Plan Specification
- TML Language
- Behavior tree
- Python Language

Tutorials [Beginners]
- Prepare the Mission:
  - Configure Map
  - Specify Mission Tasks
  - Configure Launcher
- Execute the Mission:
  - Launch Real Flight
  - Practical Tips for Real Flights
- Mission examples:
  - Simulated Quadrotor
  - AR Drone and Visual Markers

Tutorials [Advanced]
- How to program a new behavior
- Setup the Communication Network
- Wi Fi Communication with Multiple Drones
- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

More information
- Publications
- Vision & Aerial Robotics:
  - First prize at IMAV 2013
  - Stack pre-release 2013

Clone this wiki locally

https://github.com/Vision4U

Clone in Desktop

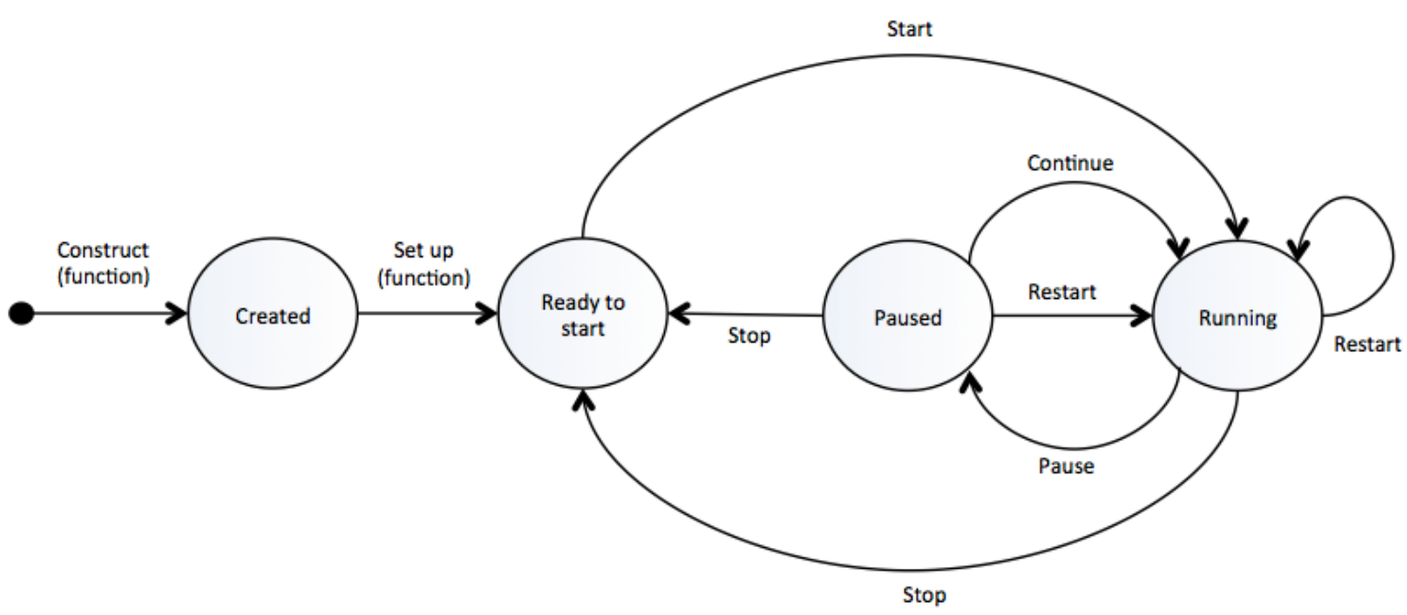
- *Synchronous/asynchronous execution*. During the execution, the process develops a series of data transformations. The execution can be asynchronous if data transformations are performed in response to changes in inputs (new messages in subscribed ROS topics) or in response to service calls from other processes (ROS services). The execution can be synchronous if data transformations are periodically performed with a certain frequency (e.g., frequency of 10 hertzs). As a result of its execution, the process generates output data to be used by other processes (new published messages in ROS topics or responses to ROS services).

- *Sequential logic*. Some processes follow a sequential logic (as opposite of combinatorial logic). This means that the outputs depend not only on the present value of the input but also on the input history. In this case, the process has memory, represented with memory states with a set of variables. At the beginning of the execution of a process, the memory has an initial state (with certain initial values for the variables) and, then, the memory state changes with the execution progress.

- *Termination*. In any moment, the normal execution can be forced to stop, i.e., finish the execution and return to the initial state of the memory. This is useful to cancel the execution of certain processes that can be in conflict with others. This can be done, for example, by a process that controls the correct execution of groups of processes to give an adequate and efficient support to certain actions and skills. The operator can also terminate a process to solve an operational problem or to check certain functionalities. When the execution is finished, the process transits to a state that ignores changes in inputs and service calls (other than certain special services, such as “start” again the execution).

- *Pause*. The normal execution of the process can be paused, i.e., stop the execution keeping the current state of the memory. When the execution is paused, the process transits to a state that ignores changes in inputs and service calls (other than certain special services, such as “continue” the execution).

- Other possible changes of the flow execution (not implemented yet). The normal execution of the process can also be changed with the following types of actions: (1) pause a period of time, i.e., stop the execution keeping the current state of the memory and, after a period of time, continue the execution, (2) recover fault, i.e., recover from a detected fault, (3) update memory state, i.e., set a given state as the current memory state.



The states of a process are the following:

- *Created*: The process has been created but it is not set up yet (it is not configured and it is disconnected to the network of processes).

- *Ready to start*: The process has been set up so it is ready to start running. The memory has its initial state. The process is not executing yet, so it is not reading and writing any message (only certain reserved messages related to its supervision). The process does not respond to any service call (except execution control services like “start”).

- *Running*: The process is running. The process reads and writes messages and responds to service calls.

- *Paused*: The process execution is paused indefinitely. The memory keeps its current state. The process is not executing, so it is not reading and writing any message (only execution control messages related to its supervision). The process does not respond any service call (except execution control services like “start”).

The class RobotProcess uses this set of states that are common for all processes. The services provided by the class RobotProcess to control the execution of processes are the following:

- START: Initiates the normal execution of a process that is ready to start.
- STOP: Cancel the execution of a process that is running or paused, and reset variables, i.e., the variables are changed to their initial values (initial state of the memory).
- RESTART: Restart the normal execution of a process that is running or paused. The process starts again from the initial state of memory.
- PAUSE: Pause the process execution, keeping the state of the memory.
- CONTINUE: Continue the normal execution from the paused state.

To support these services, the class RobotProcess includes a set of predefined functions:

- *constructor*: This function performs the following steps: (1) construct the process with default values for some variables and (2) set the process state to the initial value (created).
- *setUp*(): This function must be called in the main program and performs the following steps: (1) subscribe the ROS topics related to execution control, (2) call the function ownSetUp(), and (3) set the process state to the value ready to start.
- *start*(): This function implements the service START. For immediate execution of a process, this function can be called in the main program of the process, just after the function setUp(). The function performs the following steps: (1) set the process state to the value running, (2) call the function ownStart().
- *stop*(): This function implements the service STOP. This function performs the following steps: (1) call the function ownPause(), (2) call the function ownReset() and (3) set the process state to the value ready to start.
- *restart*(): This function implements the service RESTART. This function perform the following steps: (1) call the function stop() and (2) call the function start().
- *pause*(): This function implements the service PAUSE. This function performs the following steps: (1) call the function ownPause() and (2) set the process state to the value paused.
- *continue*(): This function implements the service CONTINUE. This function performs the following steps: (1) set the process state to the value running, and (2) call the function ownContinue().
- *run*(): This is a function that can be called from the main function to be executed periodically. This function performs the following: if the current state is running then call the function ownRun().

The class RobotProcess defines also a set of specific functions, with the prefix *own*, that the programmer must complete for each particular process:

- *ownSetUp*(): This function includes specific actions to set up the process such as read ROS parameters or read configuration files (e.g., XML files). This function can initiate ROS topics and services related to static information that do not depend on the execution.
- *ownStart*(): This function includes specific actions to start running the process such as initiate ROS topics and services specific for this process whose values depend on the execution.

- *ownRun*(): This function develops the main body of execution to perform in the running state. This does not include the tasks to be done as an answer to the service calls or topic messages. If the process execution is asynchronous, i.e., it only reacts to ROS topics and services, then the function ownRun() is empty.

- *ownPause*(): This function pauses the process execution by doing shutdown of ROS topics and services (specific for this process that depend on the execution).

- *ownReset*(): This function reset the memory state of the process.

- *ownContinue*(): This function continue the process execution by subscribing again ROS topics and services (specific for this process).

3. Centralized Error Notification

The class RobotProcess provides the function notifyError() to notify errors to the supervision system. This function needs four parameters: *error_type*, *error_reference_code*, *error_location* and *error_description*. For example:

```
notifyError(InvalidInputData,32,"determineTrajectory()",exception.what());
```

Please notice the last function attribute. It is using the exception's function what() which returns the description of a captured exception. Using exceptions is an appropriate way to notice errors during execution and thus the recommended way for handling them.

To handle invalid input data errors, each process should include a validation procedure for the input data and the corresponding notification message. For example, let us consider a process that receives as input a degree value of a robot joint, using the variable KneeDegree. Because this joint can only bend from 0 to 90 degrees, the process receiving this data must check if the value of KneeDegree is within this interval, before using this value in the corresponding procedure. The following C++ code example illustrates how to do this validation and the corresponding notification:

```
void kneePositionCallback(const KneeDegree::ConstPtr& degree)
{
  if (degree < 0 || degree >90)
  {
     notifyError(InvalidInputData,0,"kneePositionCallback()",
                 "Invalid knee bend value");
     return;
  }
  //Functional code
  ...
}
```

To handle safeguard errors, C++ exceptions should be used. An exception brings the ability of advertising an error (throw an exception) and dealing with it at any other part of the process code, where the error may actually be solved (catching an exception). For instance, let us consider a process with the function positionOperation() that can throw an exception in the case something goes wrong. The particular error that can detect the function positionOperation() is that some of the position values have a negative value, which is against the logic of the program. The try code block checks if an exception is thrown. If that is the case and the exception is the one defined at the catch block (std::logic_error), the error is handled by the catch code block. In this particular example, the error is notified to the performance monitor using the function notifyError(). The corresponding C++ code is the following:

```
    try
    {
      positionOperation(&x_position, &y_position);
    }
    catch (std::logic_error& exception)
```

```
    {
      notifyError(SafeguardRecoverableError,0,
                  "positionOperation",exception.what());
      ...
      return;
    }
```

4. The class DroneModule

The class DroneModule has a similar purpose as the class RobotProcess. However the class DroneModule is a simplified and previous version that was implemented for old processes. New processes are now implemented as subclasses of RobotProcess. The class DroneModule is still part of the software architecture to provide compatibility with old processes that keep their original C++ implementation.

One of the consequences of this is that these processes do not have the same capabilities to be supervised by the performance monitor. In particular, in contrast to the supervised processes that are subclass of RobotProcess, the old processes only have the following states:

- Started (same as running): The process has started and it is running. It is activated with the function start().
- NotStarted (same as ready to start): The process has not started yet. It is activated with the function stop().

These processes also send alive signals (AliveSignal message), so that the performance monitor is able to verify their state.

Contact

We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page.

You can also contact the team support at the following address: aerostack.vision4uav@gmail.com

Setup the Communication Network

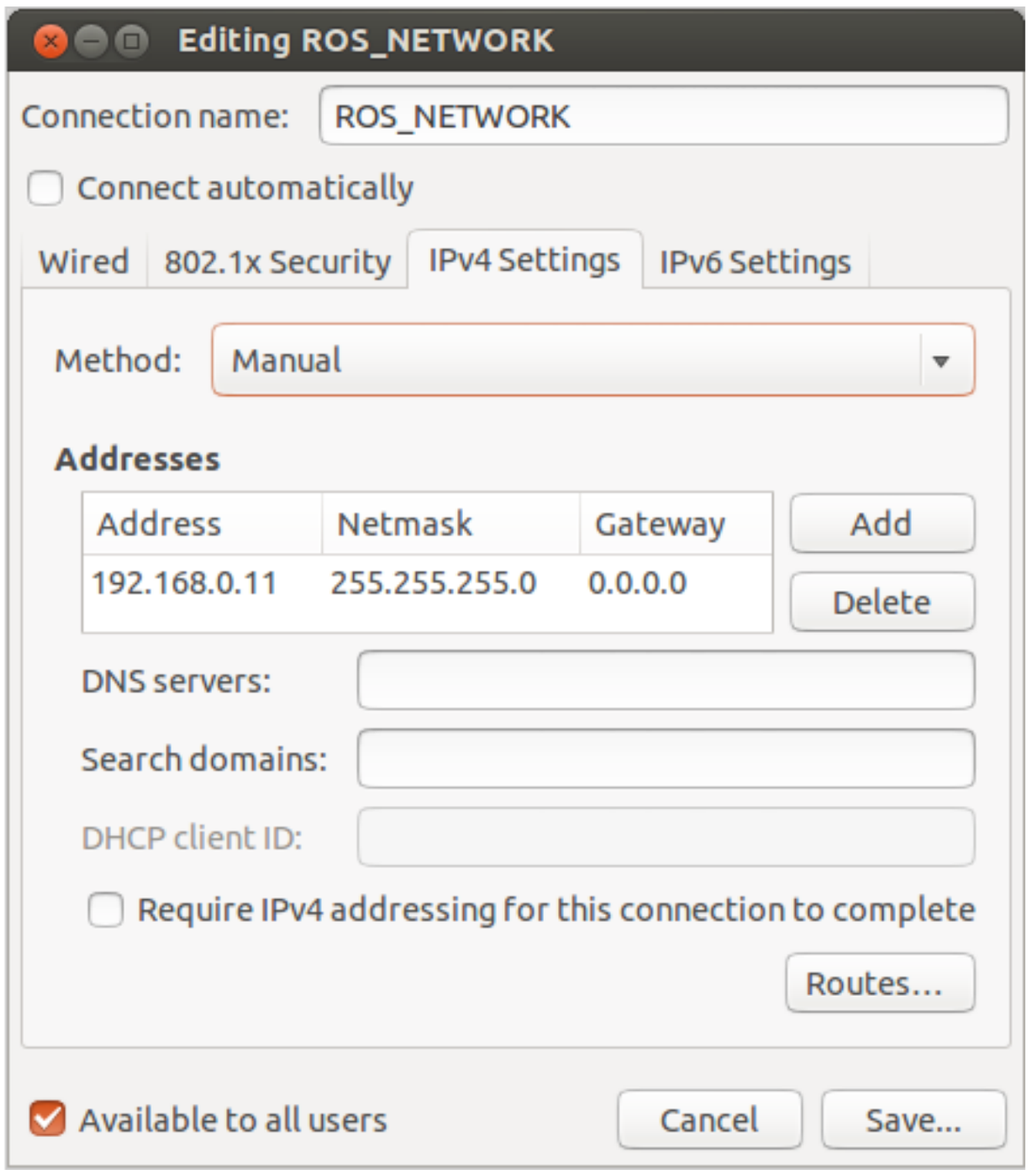
Martin Molina edited this page on 26 Oct 2017 · 2 revisions

To setup the communication network, please read before the related ROS tutorial.

> NOTE: There is no need of a network setup when using an ARDrone as it can be connected directly to the groundstation using WIFI.

Then, follow these steps.

Configure a LAN network connection to communicate ground stations with the rest using ROS. The following image shows one of the used configurations:



Edit the hosts file, all the computers in the network have to be included in it. The IPs have to be the same that is specified in the configuration of each computer's ROS_NETWORK configuration. To edit the hosts file, run:

```
$ sudo gedit /etc/hosts
```

An example of the content of this file is the following:

```
127.0.0.1       localhost
127.0.1.1       jespestana-XPS

# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters

# ROS_NETWORK, etc.
192.168.1.1     ardrone
192.168.0.10    CVG-JL-Toshiba
192.168.0.10    ROS_10
```


Pages 37

Introduction

- What is Aerostack?
- Getting Started
- License
- List of distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Catalog of Behaviors
- Memory of Beliefs

Mission Plan Specification

- TML Language
- Behavior tree
- Python Language

Tutorials [Beginners]

- Prepare the Mission:
  - Configure Map
  - Specify Mission Tasks
  - Configure Launcher
- Execute the Mission:
  - Launch Real Flight
  - Practical Tips for Real Flights
- Mission examples:
  - Simulated Quadrotor
  - AR Drone and Visual Markers

Tutorials [Advanced]

- How to program a new behavior
- Setup the Communication Network
- Wi Fi Communication with Multiple Drones
- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

More information

- Publications
- Vision & Aerial Robotics:
  - First prize at IMAV 2013
  - Stack pre-release 2013

Clone this wiki locally

```
192.168.0.11    ROS_11
192.168.0.12    carrio-GT70
192.168.0.12    ROS_12
192.168.0.13    SATELLITE-P850
192.168.0.13    ROS_13
192.168.0.20    joselu-Aspire-E1-571
192.168.0.20    ROS_20
192.168.0.21    joselu-Aspire-E1-571-01
192.168.0.21    ROS_21
192.168.0.30    joselu-Q330
192.168.0.30    ROS_30
192.168.0.40    CVG-Desktop01
192.168.0.40    ROS_40
192.168.0.41    CVG-Desktop02
192.168.0.41    ROS_41
```

Finally, restart your network for the configuration changes to take place.

Contact

We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page.

You can also contact the team support at the following address: aerostack.vision4uav@gmail.com

https://github.com/Vision4U

Clone in Desktop

Wi Fi Communication with Multiple Drones

Martin Molina edited this page on 30 Jan 2017 · 4 revisions

This section describes how to proceed to establish an adequate communication using Wi-Fi when there are multiple drones. There are two alternative options:

- One to one (each drone to one computer)
- Many to one (several drones to one computer)

One to one

In this case, you must have the same (or more) number of computers as drones. Each computer connects to a drone's Wi-Fi hotspot. Also, you must have a way of connecting computers to each other, for example a switch or a router. Keep in mind that there are two different subnets, one for the connection to the drone and another for the connection among computers.

After the computers are connected, you must edit the /etc/hosts file to add each computers hostname and their respective IPs.

NOTE: To avoid interferences make sure that the drones' hotspots are in different Wi-Fi channels. It also helps if there aren't too many other hotspots in those channels. To change a drone's channel you can use the following script:

```
SSID=$(grep ssid_single_player /data/config.ini | awk -F "=" '{print $2}')
ifconfig ath0 down
iwconfig ath0 mode master essid $SSID channel $2
iwconfig ath0 commit
ifconfig ath0 $1 netmask 255.255.255.0 up
```

To run this script login to the drone via telnet, create the script and execute it.

```
telnet 192.168.1.1
vi bin/set_IP_ch.sh #Copy the script and close vi
chmod +x bin/set_IP_ch.sh
set_IP_ch.sh 192.168.1.1 <CHANNEL>
```

NOTE: Since the drones are MAC-locked by default, when connecting a new computer to a drone, you should press the reset button on the drone to erase the previous computer's MAC address. The button is located under the battery, and you will need a paper clip or similar to press it.

Many to one

In this case, you should use a router or a computer with a wifi hostpot to which the drones will connect to. To do this, you have to edit the drone's configuration to connect to your router. For detailed instructions see: Multiple AR Drones

1. Edit the drone's pairing setup

First connect to the drone's hotspot, and login via telnet:

```
telnet 192.168.1.1
```

Now edit the following script:

```
vi bin/pairing_setup.sh
```


Pages 37

Introduction
- What is Aerostack?
- Getting Started
- License
- List of distributions

Components
- Aerostack Architecture
- Graphical User Interface
- Catalog of Behaviors
- Memory of Beliefs

Mission Plan Specification
- TML Language
- Behavior tree
- Python Language

Tutorials [Beginners]
- Prepare the Mission:
  - Configure Map
  - Specify Mission Tasks
  - Configure Launcher
- Execute the Mission:
  - Launch Real Flight
  - Practical Tips for Real Flights
- Mission examples:
  - Simulated Quadrotor
  - AR Drone and Visual Markers

Tutorials [Advanced]
- How to program a new behavior
- Setup the Communication Network
- Wi Fi Communication with Multiple Drones
- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

More information
- Publications
- Vision & Aerial Robotics:
  - First prize at IMAV 2013
  - Stack pre-release 2013

Clone this wiki locally

https://github.com/Vision4U
Clone in Desktop

Look for this line and comment it:

```
iptables -A INPUT -m mac --mac-source $MAC_ADDR -j ACCEPT
```

Then change

```
iptables -P INPUT DROP
```

into

```
iptables -P INPUT ACCEPT
```

2. Add and execute the wifi setup script

Create the following file

```
vi bin/drone_wifi_config.sh
```

Add these lines

```
killall udhcpd
ifconfig ath0 down
iwconfig ath0 mode managed essid <ESSID>
iwconfig ath0 commit
ifconfig ath0 <IP_ADDRESS> netmask <NETMASK> up
```

Now make the script executable

```
chmod +x bin/drone_wifi_config.sh
```

Finally, exit your telnet session and execute the following command in your computer

```
echo drone_wifi_setup.sh | telnet 192.168.1.1
```

Your drone should now be connected to your router

NOTE: To insert text in vi, press 'i' before writing some text. To save and exit the file in vi, introduce the following sequence `ESC shift+z shift+z`

3. Edit the launchers

Edit the file $AEROSTACK_STACK/launchers/ardrone_indoors.launch and add `args="-ip $(arg drone_ip)"` to the first `<node>` tag. Now look for the ArDrone Autonomy section of your launcher and add the following parameter: `drone_ip:=$DRONE_IP`.

Ultrasound frequency

In order to avoid the drones repelling each other when they are in proximity, the following parameter should be modified in the *ardrone_indoors.launch* file located in *launchers/ardrone_launch*

```
<param name="ultrasound_freq" value="8" />
```

This parameter can be set to either 7 or 8. Drones with this parameter set to different values won't repel each other.

Contact

We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page.

You can also contact the team support at the following address: aerostack.vision4uav@gmail.com

Calibrate Controllers

Martin Molina edited this page on 18 Jul 2017 · 2 revisions

Calibrating controllers become very important step if user has added more payloads to the UAV or a UAV has been added which with the Aerostack was not used before. There are several controllers which are used the Aerostack and they are arranged in several layers. The first layer is the low level controller which is inside the autopilot of the UAV system. This low level controller can be tuned according to the instructions given by the people providing the autopilot system.

Above the low-level controller comes the mid-level controller which is present inside the Aerostack. This controller is used for take off, landing and hovering of the UAV. (Note: In case of the ARdrone, the mid-level controller provided by the ARdrone manufactures is being used and hence the mid-level controller provided by the Aerostack is not used). Above the mid-level controller is the high-level controller, using which the user can give speed commands, position commands as well as trajectories to the UAV.

Calibrating the Mid-Level Controller

As mentioned above the mid-level controller is used for take-off, landing and hovering of the UAV, and hence it is one of the critical controllers. If the mid-level controller is not tuned properly, there are high chances of the UAV crashing. Hence it is very important that is mid-level is tuned perfectly.

The mid-level controller is a combination of three different controllers namely:

- Altitude controller
- Hover controller
- Yaw controller

As the names suggest, the altitude controller is the one which is used for controlling the altitude of the UAV. The hover controller takes in input as vertical speeds and is used for making the UAV hover at a particular position. The yaw controller controls the heading of the UAV maintaining the current heading of the UAV.

The mid-level controller tuning parameters are stored inside $DRONE_STACK/configs/drone. The drone id (drone0, drone1, etc.) depends on the drone which the user is using. For example, the mid-level controller tuning file for Asctec Pelican UAV can be found inside $DRONE_STACK/configs/drone0, as drone0 is being used for Pelican. The name of the file which contains the tuning parameters is midlevel_autopilot.xml. Below is a step by step explanation of the midlevel_autopilot.xml.

```
<midlevel_autopilot>
  <ON_GROUND_ALTITUDE>  0.25  </ON_GROUND_ALTITUDE> <!-- m -->

  <TAKE_OFF>
    <TAKEOFF_SPEED>  0.6  </TAKEOFF_SPEED> <!-- m/s -->

    <TAKEOFF_FINAL_ALTITUDE> 1.1 </TAKEOFF_FINAL_ALTITUDE> <!-- m   -->

    <TAKINGOFF_ALT_SPEED_COS_GAMMA> 2.15 </TAKINGOFF_ALT_SPEED_COS_GAMMA>

    <MAX_TAKINGOFF_TIME_S> 5.0 </MAX_TAKINGOFF_TIME_S> <!-- s   -->
  </TAKE_OFF>

  <LANDING>
    <LANDING_SPEED> 0.40 </LANDING_SPEED> <!-- m/s -->
```


Pages 37

Introduction

- What is Aerostack?
- Getting Started
- License
- List of distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Catalog of Behaviors
- Memory of Beliefs

Mission Plan Specification

- TML Language
- Behavior tree
- Python Language

Tutorials [Beginners]

- Prepare the Mission:
  - Configure Map
  - Specify Mission Tasks
  - Configure Launcher
- Execute the Mission:
  - Launch Real Flight
  - Practical Tips for Real Flights
- Mission examples:
  - Simulated Quadrotor
  - AR Drone and Visual Markers

Tutorials [Advanced]

- How to program a new behavior
- Setup the Communication Network
- Wi Fi Communication with Multiple Drones
- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

More information

- Publications
- Vision & Aerial Robotics:
  - First prize at IMAV 2013
  - Stack pre-release 2013

Clone this wiki locally

```
  <ALTITUDE_FOR_LANDING_WITH_TIME> 0.40   </ALTITUDE_FOR_LANDING_WITH_TIME>  <!-- m   -

  <LANDING_WITH_TIME_MAX_SECONDS>     1.00 </LANDING_WITH_TIME_MAX_SECONDS>  <!-- s   -

  <LANDING_WITH_TIME_FIXED_THRUST_KG> 0.32 </LANDING_WITH_TIME_FIXED_THRUST_KG>  <!-- k
</LANDING>
```

- ON_GROUND_ALTITUDE is the on ground altitude measured by the distance sensor when the UAV is landed.

The next parameters influence the take off of the UAV.

- TAKEOFF_SPEED influences how fast the UAV will take off. Ranging from 0.1 to 1.0. It is generally advised to keep to the takeoff speed parameter to 0.6 for a good take off.
- TAKEOFF_FINAL_ALTITUDE influences the final altitude the UAV is maintain after it taking off. In case PX4flow sensor is being used as the distance sensor the range is between 0.5 to 3.0.
- MAX_TAKINGOFF_TIME_S parameter defines the maximum time the midlevel controller will maintain its status at taking off after which it will change its status to hovering. In other words if the max taking off time is 5 secs, the maximum time the midlevel controller will allow the UAV to reach its final altitude is 5 secs after which the UAV will start hovering evern if it has not reached the final altitude.

The next set of parameters are for landing

- LANDING_SPEED parameter defines how fast the UAV will land. Advisable to keep it around 0.4 m/s.
- ALTITUDE_FOR_LANDING_WITH_TIME parameter defines the altitude at which the controller will give the final thrust for llanding the UAV. For example is the the parameter is 0.4 m the controller will give the final thrust after the UAV reaches 0.4 m.
- LANDING_WITH_TIME_MAX_SECONDS parmeter defines the time is seconds the controller will wait before give the final thrust.
- LANDING_WITH_TIME_FIXED_THRUST_KG parameter is the value of final thrust given in kgs for the UAV to touch the ground. It is influenced by the weight of the UAV, lighter UAV's around 2 kgs can have a this parameter around 0.32kgs and for heavy UAV's this parameter can be around 1.32 kgs. For more information on how to choose the landing thrust value user can refer to configs/drone0/midleve_autopilot.xml.

The next set of parameters are for setting the limits for the controller as can be seen in the table below

```
<MAX_ALTITUDE>
  <MAX_ALTITUDE>   3.00 </MAX_ALTITUDE> <!-- m   -->

  <MAX_ALTITUDE_DESCENDING_SPEED> 0.20 </MAX_ALTITUDE_DESCENDING_SPEED> <!-- m/s -->

  <MAX_ALTITUDE_TO_CONSIDER_UNKNOWN_STATE_AS_LANDED> 0.40 </MAX_ALTITUDE_TO_CONSIDER_UNKN
</MAX_ALTITUDE>


<CONTROLLER_SATURATIONS>
  <ML_AUTOPILOT_MAX_ALLOWED_YAW_SPEED> 30.0</ML_AUTOPILOT_MAX_ALLOWED_YAW_SPEED> <!-- deg

  <ML_AUTOPILOT_MAX_ALLOWED_ALTITUDE_SPEED>0.5 </ML_AUTOPILOT_MAX_ALLOWED_ALTITUDE_SPEED>
</CONTROLLER_SATURATIONS>
```

- MAX_ALTITUDE parameter as the the name says defines the maximum altitude that can be achieved by the UAV. Note: when using the px4flow sensor for altitude measurement, the height is always limited to 3 meters.

https://github.com/Vision4U
Clone in Desktop

- MAX_ALTITUDE_DESCENDING_SPEED parameter defines the maximum descending speed which a UAV can achieve while landing.
- MAX_ALTITUDE_TO_CONSIDER_UNKNOWN_STATE_AS_LANDED parameter publishes the unknown state landed below a given altitude. The midlevel controller always publishes the states of the UAV where it is landed, taking off, flying etc, if in case the state is unknown the and the altitude is below.
- MAX_ALTITUDE_TO_CONSIDER_UNKNOWN_STATE_AS_LANDED parameter the midlevel controller will publish the state as landed.
- ML_AUTOPILOT_MAX_ALLOWED_YAW_SPEED parameter defines the maximum yaw speed for the UAV, recommended to keep it at 30 deg/s.
- ML_AUTOPILOT_MAX_ALLOWED_ALTITUDE_SPEED defines the maximum allowed altitude speed, right now the midlevel controller does not work properly with altitude speed more that 0.5 m/s.

The next set of parameters define the maximum and minimum tilt value, change in yaw value and the thrust value. This values remain constant to a specific autopilot of the UAV, which can obtained from the manufacturer of the UAV. In the below case the values are for the Asctec Pelican UAV obtained from the astec manufacturers.

```
<LL_AUTOPILOT>

  <LL_AUTOPILOT_COMMAND_TILT_MIN_VALUE>-51.175</LL_AUTOPILOT_COMMAND_TILT_MIN_VALUE> <!--

  <LL_AUTOPILOT_COMMAND_TILT_MAX_VALUE>+51.175</LL_AUTOPILOT_COMMAND_TILT_MAX_VALUE> <!--

  <LL_AUTOPILOT_COMMAND_DYAW_MIN_VALUE>-254.760</LL_AUTOPILOT_COMMAND_DYAW_MIN_VALUE> <!-

  <LL_AUTOPILOT_COMMAND_DYAW_MAX_VALUE>+254.760</LL_AUTOPILOT_COMMAND_DYAW_MAX_VALUE> <!-

  <LL_AUTOPILOT_COMMAND_THRUST_MIN_VALUE>0.000</LL_AUTOPILOT_COMMAND_THRUST_MIN_VALUE> <!

  <LL_AUTOPILOT_COMMAND_THRUST_MAX_VALUE>+32.000</LL_AUTOPILOT_COMMAND_THRUST_MAX_VALUE>

</LL_AUTOPILOT>
```

The next set of values define the altitude controller which is the first controller inside the midlevel controller as mentioned above.

```
<altitude_speed_controller>

  <FEEDFORWARD_M>  2.11  </FEEDFORWARD_M> <!-- kg  -->

  <THRUSTER_GAIN> 0.3065625 </THRUSTER_GAIN> <!-- 1p1/kg, pelican: 1/(32/9.81)-->

  <GAIN_P>        0.10      </GAIN_P> <!-- Pelican: 0.1, 1p1/(m/s)-->

  <GAIN_I>        0.08      </GAIN_I> <!-- Pelican:(GAIN_P / 1.25),1p1/(s*m/s)-->

  <GAIN_D>        0.00      </GAIN_D> <!-- Pelican:(GAIN_P * 0.00),1p1*s/(m/s)-->


  <ALT_SPEED_MEASURE_SATURATION_FACTOR> 3.0</ALT_SPEED_MEASURE_SATURATION_FACTOR>

  <COMMAND_UPPER_LIMIT>  0.33  </COMMAND_UPPER_LIMIT>  <!-- Pelican:(          450.0 / 4

  <COMMAND_LOWER_LIMIT> -0.11  </COMMAND_LOWER_LIMIT>  <!-- Pelican: ( (-1.0) *  150.0 /

  <SUBSAMPLING_DIVIDER>  6      </SUBSAMPLING_DIVIDER> <!-- Pelican: 6 -->

</altitude_speed_controller>
```

- FEEDFORWARD_M parameter is in kilograms the value at which the UAV can manitain its altitude. It can be calculated as follows. The user first must know the value in the terms units of the radio at which the UAV holds altitude, this can be found out manaully flying the UAV. For example in case of pelican it was found that with the current payload it holds altitude at 2560 counts using the radio, and hence FEEDFORWARD_M is: 2650 * (32/9.81) * (1/4095) = 2.11 kgs, where 32: LANDING_WITH_TIME_FIXED_THRUST_KG, 4095: The maximum thrust in units of the radio (counts), 9.81: Acceleration due to gravity.

- THRUSTER_GAIN parameter can be calculated as follows: 1/(32/9.81) where 32: LANDING_WITH_TIME_FIXED_THRUST_KG, 9.81: Acceleration due to gravity.

- GAIN_P parameter is the most important parameter, should be started with a smaller value like 0.01, until the UAV does not oscillate in the altitude.

- GAIN_I parameter should be increased from a smaller value like 0.01 until the UAV reaches its setpoint which is 1.1 meters in this case.

- GAIN_D parameter influences how fast the UAV reaches the given the setpoint. Higher values can make it very responsive and can be kept to very less value or zero as in case of pelican.

- ALT_SPEED_MEASURE_SATURATION_FACTOR parameter deprecated.

- COMMAND_UPPER_LIMIT and COMMAND_LOWER_LIMIT set the maximum and minimum saturations for the altitude controller.

- SUBSAMPLING_DIVIDER currently not used.

The next set of parameters define are used for tuning the hover controller.

```
<hover_controller>

  <GAIN_P>     9.00 </GAIN_P>      <!-- [deg / (m/s)]  -->

  <GAIN_I>     1.80 </GAIN_I>      <!-- Ti = 5s, (GAIN_P / 5.0)  -->

  <GAIN_D>     0.50 </GAIN_D>      <!-- -->

  <MAX_OUTPUT> 12.00 </MAX_OUTPUT> <!-- [deg] -->

  <ALTITUDE_FOR_NOMINAL_D> 0.70 </ALTITUDE_FOR_NOMINAL_D> <!-- [m]   -->

</hover_controller>
```

- GAIN_P parameters as for all the PID controllers is most important, higher the gain_p the more the oscillations UAV about a given position. Gain_p parameter should be set a value where the UAV is not too sluggish or responsive at a desired position.

- GAIN_I parameter makes the UAV maintain its desired setpoint position. In this case the gain_I should be gain_p/5.

- GAIN_D parameter parameter influences how fast the UAV can reach the desired attitude, hence higher values tend to make the UAV more responsive.

- MAX_OUTPUT parameter defines the minimum and maximum saturations for the hover controller.

- ALTITUDE_FOR_NOMINAL_D is the altitude value upto which the gain_d is considered by the controller after which the gain_d becomes zero.

The next set of values are for the yaw controller which is currently a PD controller.

```
<yaw_controller>

  <GAIN_P> 5.00 </GAIN_P> <!-- [deg/s / (deg)]  -->
```

```
  <GAIN_I> 0.00 </GAIN_I> <!--  -->

  <GAIN_D> 0.50 </GAIN_D> <!-- (0.1 * GAIN_P) -->

</yaw_controller>
```

- GAIN_P parameter can be tuned the same way as in the above controllers.
- GAIN_D parameter is 0.1*gain_p

Contact

We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page.

You can also contact the team support at the following address: aerostack.vision4uav@gmail.com

Add a New Camera

Martin Molina edited this page on 26 Oct 2017 · 3 revisions

Aerostack supports all types of ueye cameras with usb. There are ROS packages in the Aerostack for the ueyecameras which take the data from the camera and then publishes the camera data using the suitable config files provided by the user. Adding a new camera to the Aerostack is relatively easy and can be done in the following steps.

1. Installing the ueyecamera driver
2. Getting the camera_id
3. Generating a suitable config file
4. Generating a camera calibration file
5. Launching the ueyecamera node

1. Installing the Ueyecamera Driver

The most important thing in order to add a new camera to the stack is installing the drivers for the ueyecamera. Note that installing the ueyecamera drivers is mandatory when installing the Aerostack because without the ueyecamera drivers the Aerostack will never be build. But just in case if the ueyecamera drivers are not installed here is how it can be done.

The link to download the ueyecamera can be found here

https://en.ids-imaging.com/download-ueye-lin64.html

The user has to install the drivers for linux. The installing instructions for installing the ueyecamera drivers is properly mentioned in a text file which comes along with the drivers.

2. Getting the camera_id

After installing the ueyecamera drivers, the user has to check the camera_id of the connected ueyecamera. Which can be done as follows. Open the Gnome terminal and type the command ueyecameramanager and a window will open showing all the connected cameras to the computer.

The figure below shows an example of the ueyecameramanager connected with one camera and with all its properties. The camera_id can then be seen under the cam ID section.

Pages 37

Introduction

- What is Aerostack?
- Getting Started
- License
- List of distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Catalog of Behaviors
- Memory of Beliefs

Mission Plan Specification

- TML Language
- Behavior tree
- Python Language

Tutorials [Beginners]

- Prepare the Mission:
  - Configure Map
  - Specify Mission Tasks
  - Configure Launcher
- Execute the Mission:
  - Launch Real Flight
  - Practical Tips for Real Flights
- Mission examples:
  - Simulated Quadrotor
  - AR Drone and Visual Markers

Tutorials [Advanced]

- How to program a new behavior
- Setup the Communication Network
- Wi Fi Communication with Multiple Drones
- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

More information

- Publications
- Vision & Aerial Robotics:
  - First prize at IMAV 2013
  - Stack pre-release 2013

Clone this wiki locally

https://github.com/Vision4U
Clone in Desktop

3. Generating a Suitable Config File

The config file is an important file, using which the user can define all the properties of the camera, for example the parameters like the brightness, frame rate, exposure and many more. In order to set these parameters the camera needs to be opened first. It can be done by right clicking on the available camera in the screen shown in the above figure.

After opening the camera, all the camera parameters can be set as desired by the user. After setting the desired camera parameters, the user then has to save the config file with the desired parameters. The config file has to be saved inside DRONE_STACK/Configs/drone_id. The number of the drone_id depends on the drone which the user is using. For example if the user is using the camera with Asctec Pelican the config file needs to be saved in drone0, as drone0 is currently being used for the Astec Pelican.

Also there are some default config files for the ueye cameras which are already there inside DRONE_STACK/Configs/drone_id. small_camera_RGB24_HUB.ini is one such config file which is already being generated with certain camera parameters. It completely depends on the user to choose as to which camera config is the best.

4. Generating a Camera Calibration File

The camera calibration file contains the necessary camera calibration data like the focal length, projection matrix etc of the camera. The camera calibration file can be generated using the ROS camera calibration package which can be found in the link below:

http://wiki.ros.org/camera_calibration

After generating the suitable calibration file for the camera it can then be saved inside the same folder DRONE_STACK/Configs/drone_id. Where again drone_id depends on the drone the user is using. An example of the calibration file can be found below.

```
image_width: 640
image_height: 360
camera_name: front
camera_matrix:
  rows: 3
  cols: 3
  data: [466.6953647276067, 0, 306.515467461629, 0, 466.1550228548235,
169.6544700845251, 0, 0, 1]
distortion_model: plumb_bob
```

```
distortion_coefficients:
  rows: 1
  cols: 5
  data: [-0.3088667172761168, 0.1060937937543472, -0.001639357143514502,
0.0004536735112699787, 0]
rectification_matrix:
  rows: 3
  cols: 3
  data: [1, 0, 0, 0, 1, 0, 0, 0, 1]
projection_matrix:
  rows: 3
  cols: 4
  data: [391.9715881347656, 0, 301.7997751755956, 0, 0, 442.0516967773438,
167.6912864963015, 0, 0, 0, 1, 0]
```

Note that the format of the calibration file should be same as the above file or the ueye package in the Aerostack will fail to start. The camera calibration file has to be saved with the same camera name which is will be used by the user in the launch file to launch the camera. For example, if the name of the camera is front, the name of the calibration file has to be front.yaml.

5. Launching the Ueyecamera Node

The user has to make a suitable launch file in order to launch the ueyecamera with the proper config file and the calibration file as well as with the proper camera name and camera_id. An example of such a launch file can be found below.

```
<!-- launches: ueye_cvg node -->
<launch>
    <arg name="camera_id_num" default="1"/>
    <arg name="config_file" default="small_camera_RGB24.ini"/>
    <arg name="camera_name_str" default="front" />

    <arg name="drone_id_int" default="0" />
     <arg name="my_stack_directory" default="$(env DRONE_STACK)" />

    <group ns="drone$(arg drone_id_int)">
     <node name="cvg_ueye_cam$(arg camera_id_num)_$(arg camera_name_str)" pkg="driver_cam
      <param name="camera_ident_number" value="$(arg camera_id_num)" type="int"/>
      <param name="configuration_file" value="$(arg my_stack_directory)/configs/drone$(ar
      <param name="camera_name"   value="$(arg camera_name_str)" type="string"/>
      <param name="topic_name" value="camera/$(arg camera_name_str)"  type="string"/>
      <param name="droneId" value="$(arg drone_id_int)" type="int"/>
      <param name="stackPath" value="$(arg my_stack_directory)" type="string"/>
      <param name="cam_intr_filename"  value="$(arg my_stack_directory)/configs/drone$(ar
     </node>
   </group>
</launch>
```

The above launch file launches the ueyecamera node in the Aerostack with parameters and files as explained below:

- The first argument in the launch file is the camera_id where the user inputs the camID seen from the ueyecameramanager. In this example the camera_id is 1.
- The second argument defines the name of the config file, which is small_camera_RGB24.ini in this case above.
- The third argument is the camera name which is front in the above case. The fourth and the fifth argumets defines the drone_id being used and the environment for the Aerostack.

After all the arguments the ueyecamera ROS node is launched with the name "cvg_ueye_cam$(arg camera_id_num)_$(arg camera_name_str)". Hence here in the above case the name of the ueyenode would be cvg_ueye_cam1_front.

After launching the node, the parameters for the node are set:

- The first parameter is the camera_id.
- The second parameter is the assigns the camera config file, which takes the value "$(arg my_stack_directory)/configs/drone$(arg drone_id_int)/$(arg config_file)". In the above it is
- $DRONE_STACK/configs/drone0/small_camera_RGB24.ini
- The third parameter is the name of the camera, in the above case the camera name is front.
- The fourth parameter defines the topic name which contains the camera data, which takes the camera/$(arg camera_name_str). In the above case it is camera/front.
- The fifth and the sixth parameters are the drone_id and the stack_path.
- The seventh parameter is the assigns the camera calibration file, which takes the value "$(arg my_stack_directory)/configs/drone$(arg drone_id_int)/$(arg camera_name_str).yaml". In the above case the calibration file is assigned as follows
  $DRONE_STACK/configs/drone0/front.yaml"

The above launch file can be launched using the roslaunch command assigning all the camera arguments. For example:

```
roslaunch driver_camera_ueye_ROSModule driver_camera_ueye_ROSNode
--wait camera_id_num:=2
         camera_name_str:="front"
         config_file:="small_camera_RGB24_HUB.ini"
```

Contact

We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page.

You can also contact the team support at the following address: aerostack.vision4uav@gmail.com

Add a GPS Sensor

Martin Molina edited this page on 26 Oct 2017 · 3 revisions

This section describes with an example how to add a new sensor to a UAV platform using the Aerostack. There might be several sensors that can be added to the Aerostack according to its usage, the important thing to keep in mind is that when adding a new sensor to the stack the sensor should publish its data in ROS and there should be an appropriate translator/interface that receives this sensor data. This translator or interface then converts the data in the units and conventions required by the stack and then publishes it to the appropriate module of the stack. In the following sections an example of adding a GPS sensor to the Aerostack is presented.

This example illustrates adding a particular type of GPS sensor shown in the figure below. GPS is the most commonly used sensor with UAV's, since it can give a good position estimation of the UAV with reference to a global frame of reference, the only drawback of GPS is that is cannot be used indoors or in cluttered environments for example in places with a lot of trees or buildings.

The position estimated by the GPS always has a drift, hence GPS data has to be given to the extended kalman filter in the Aerostack in order to obtain a better pose estimation of the UAV with respect to a given frame of reference.

The following steps have to performed in order to add a GPS sensor in the Aerostack

1. A GPS sensor with USB connection sending NMEA sentences (shown in figure below).
2. A ROS package that listens to this NMEA sentences and then converts them to latitude and longitude values.
3. A ROS package which converts this latitude and longitude values into proper x, y and z coordinates in Universal Transverse Mercator coordinate system (UTM coordinate system).
4. A ROS package to convert these x, y and z values in UTM coordinate system to appropriate coordinate system required by the Aerostack. This package is called the translator or the interface for the GPS sensor.
5. Adding few changes in the extended kalman filter package to subscribe to the x, y, z values published by the GPS interface package.

A detailed explanation of all this steps is given below step by step.

1. Choosing a GPS Sensor

The first and the most important part in choosing a GPS is that it should publish NMEA sentences, because all the packages developed in ROS for GPS modules assume that the GPS module sends NMEA sentences.

One can also use the GPS data given by a GPS which is attached to the autopilot of UAV. In this case, the user has to hack the autopilot to publish the appropriate GPS data for the following package which will be subscribing to this data. The figure below shows the USB GPS being used in this example which can be bought from a local electronics store at a low cost.

Pages 37

Introduction

- What is Aerostack?
- Getting Started
- License
- List of distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Catalog of Behaviors
- Memory of Beliefs

Mission Plan Specification

- TML Language
- Behavior tree
- Python Language

Tutorials [Beginners]

- Prepare the Mission:
  - Configure Map
  - Specify Mission Tasks
  - Configure Launcher
- Execute the Mission:
  - Launch Real Flight
  - Practical Tips for Real Flights
- Mission examples:
  - Simulated Quadrotor
  - AR Drone and Visual Markers

Tutorials [Advanced]

- How to program a new behavior
- Setup the Communication Network
- Wi Fi Communication with Multiple Drones
- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

More information

- Publications
- Vision & Aerial Robotics:
  - First prize at IMAV 2013
  - Stack pre-release 2013

Clone this wiki locally

https://github.com/Vision4U
Clone in Desktop

2. ROS Module to Get the Data from the GPS Sensor

There is already a package available in ROS which listens to the NMEA sentences given by the GPS sensor and converts them into appropriate latitude and longitude values. The ROS package can be found under this link:

https://github.com/ros-drivers/nmea_navsat_driver

A launch file can be made to launch the node of this package which is as follows:

```
<!-- Launches: nmea_navsat_driver_node -->
<launch>

  <arg name="drone_id_namespace"  default="drone0" />
  <arg name="drone_id_int"                default="0" />
  <arg name="my_stack_directory"     default="$(env DRONE_STACK)" />

  <group ns="$(arg drone_id_namespace)">

        <node name="nmea_navsat_driver_node" pkg="nmea_navsat_driver" type="nmea_serial_d

        <param name="port"    value="/dev/ttyUSB0"  type="string" />
        <param name="baud"   value="4800"               type="int"   />
        </node>
 </group>
</launch>
```

This file launches the 'nmea_serial_driver' node with the name 'nmea_navsat_driver_node' from the package 'nmea_navsat_driver'. The param port defines the port at which the GPS is connected and the baud which should always be kept at 4800 in case of this GPS.

3. ROS Module Which Provides Position in UTM Coordinates

This is also a standard ros package which can be downloaded from github:

https://github.com/ktossell/gps_umd/tree/master/gps_common

This package converts the latitude and longitude values provided by the ROS Module in the previous section into x, y and z position in UTM coordinates.

A launch file can be made to launch the node of this package which is as follows:

```
<!-- Launches: gps_umd_node -->
<launch>

  <arg name="drone_id_namespace"  default="drone0" />
  <arg name="drone_id_int"                default="0" />
  <arg name="my_stack_directory"     default="$(env DRONE_STACK)" />

   <group ns="$(arg drone_id_namespace)">
```

```
        <node name="gps_umd" pkg="gps_common" type="utm_odometry_node" output="screen" res

        </node>
    </group>
  </launch>
```

This file launches the ‘utm_odometry_node’ with a name ‘gps_umd’ from the package ‘gps_common’.

4. ROS Interface/Translator for the Received GPS Data

This package converts the position data given by the utm_odometry_node into data which is required by the Aerostack. At first, for the Aerostack the X axis should always be pointing towards the north and the Y axis towards the east but in case of utm_odometry_node the X axis points towards the east and Y axis points towards the north. Hence when this package receives data from the utm_odometry_node it first changes the axis as shown in the following code below. This piece of code is a callback function in this package which is called when this package receives the data from the utm_odometry_node.

```
void GPSOdomROSModule::GPSOdomCallback(const nav_msgs::Odometry &msg)
{
        //Asynchronous module with only one callback!
        if(!run())
        return;

        GPS_pose.x = msg.pose.pose.position.y;
        GPS_pose.y = msg.pose.pose.position.x;
        GPS_pose.z = msg.pose.pose.position.z;

        ConvertToLocal(&GPS_pose);

        publishGPSValue();
        return;
}
```

Where:

- GPS_pose.x is Pose in x axis which will be published to the Aerostack,
- msg.pose.pose.position.y is Pose in y axis which was published by the utm_odometry_node,
- The convention goes similar for other axis.

The function ConvertToLocal(&GPS_pose) takes the current pose given utm_odometry_node and substracts it from the pose at the initial point (Initial pose is the posewhich was sent by the GPS was got connected). For example, if the GPS is connected at a particular point the utm_ododmetry_node publishes data in x = 44545 y = 4482, z = 650, but since this being the initial point the data should always be x = 0, y = 0 and z = 0, and hence the function ConvertToLocal(&GPS_pose) converts all the start point data to zero.

```
bool GPSOdomROSModule::GlobalToLocal(droneMsgsROS::dronePose *current)
{
        static droneMsgsROS::dronePose map_origin;
        static bool first_pose_received = false;

        bool initial_pose = !first_pose_received;

        if(initial_pose)
        {
        map_origin   = *current;
        map_origin.x = map_origin.x;
        map_origin.y = map_origin.y;
        map_origin.z = map_origin.z;

        first_pose_received = true;
```

```
        ROS_INFO("INITIAL data (%.3f, %.3f, %.3f), map origin (%.3f, %.3f, %.3f)",
        current->x, current->y, current->z,
        map_origin.x, map_origin.y, map_origin.z);
        }

        current->x -= map_origin.x;
        current->y -= map_origin.y;
        current->z -= map_origin.z;

        return initial_pose;
}
```

The variable map_origin has the data when the GPS started sending data. The current data is then subtracted from the map_origin data to get distances with respect to the start point.

5. Publishing the GPS Data to Aerostack

The translated data given by a sensor now has to be processed by a suitable module in the Aerostack. In case of the GPS data the x, y and z positions need to be given to EKF_Module to get a better pose estimate from it.

But before giving the gps data to the EKF it first needs to be converted to the frame of reference used by the EKF. The frame of reference used by the EKF can be seen inside documention/Coordinate_frames/General_Frames_Field.svg. As can be seen from the figure the EKF takes measurements in Fdrone__Lmrt wrt to FEKF Frame. This fixed EKF frame always aligns itself to the heading of the UAV when the EKF is started and then the heading of the fixed EKF frame remains constant.

The GPS Frame of reference is always fixed with x pointing towards the north and y pointing towards the east, and hence by using the yaw_degree given by the magnetometer the GPS fixed frame can be aligned to the EKF fixed frame as soon as the EKF is started. Below is a code showing how this reference frame conversion is being done.

```
        float x_drone,y_drone;

        GlobalFrame(0,0) = x_gps;
        GlobalFrame(1,0) = y_gps;
        GlobalFrame(2,0) = z_gps;

        RotationMat(0,0) = cos(yaw_drone);
        RotationMat(1,0) = -sin(yaw_drone);
        RotationMat(2,0) = 0;

        RotationMat(0,1) = sin(yaw_drone);
        RotationMat(1,1) = cos(yaw_drone);
        RotationMat(2,1) = 0;

        RotationMat(0,2) = 0;
        RotationMat(1,2) = 0;
        RotationMat(2,2) = 1;

        BodyFrame = RotationMat*GlobalFrame;

        x_drone = BodyFrame(0,0);
        y_drone = BodyFrame(1,0);
```

The yaw_drone parameter is taken only once when the EKF is started or it is reset, and during rest of time it remains constant with the first value when EKF was started. The x_drone and y_drone are positions converted with respect to EKF frame of reference and are then provided to the EKF and if everything else is correct the user should obtain a pose from the EKF which is fused with the GPS data. Note that in the example above only x and y data from the GPS are given to the EKF the z data is not used as it has a lot of noise.

Contact

We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page.

You can also contact the team support at the following address: aerostack.vision4uav@gmail.com

Use Aerostack for a New Physical Platform

Martin Molina edited this page on 19 May 2017 · 2 revisions

Adding a new physical platform basically means adding a new autopilot system which is not supported by the Aerostack. Sometimes the autopilot comes with a fixed platform like the Ardrone or sometimes it is only the autopilot and the user has to build the platform for the autopilot like Pixhawk. The following example demonstrates how the Pixhawk autopilot was added to the Aerostack.

1. Basic Steps for Setting up Communication Between Aerostack and Pixhawk

When adding a new physical platform the most important thing to have a ROS package of the autopilot being added to the Aerostack. This ROS Package is used for communication between the Autopilot with the Aerostack. As mentioned before the Aerostack is modular and hence in order to communicate with the Aerostack the Autopilot ROS package needs an Interface in order to convert the data coming from the Autopilot in units required by the Aerostack and also to convert the commands sent by the Aerostack in units required by the Autopilot. If the user is successful is making an interface for the Autopilot, the rest of the code for the controllers remains the same, and then its just a matter of tuning the controllers to fly the Autopilot with the Aerostack.

In case of Pixhawk, the ROS package used for communication is the Mavros which is a MAVLink extendable communication node for ROS. An Interface called the driverPixhawkROSModule is then made for the Mavros in order for it to communicate with the Aerostack.

The basic things that the Aerostack requires from any autopilot are the IMU data (rotation angles) and the battery voltage data and for sending commands from the Aerostack, the autopilot should except roll, pitch, dyaw (change in yaw) and throttle commands.

2. Getting the Rotation Angles from the Pixhawk Using Macros

The Aerostack accepts rotation angles in NED (North, East and Down) frame of reference for more information about the frame of reference used in Aerostack please refer to documentation/coordinate_frame/Multirotor_coordinate_frames.txt. It requires the rotations in euler angles and the angles should be in degrees and not in radians.

Mavros on the other hand uses ENU (East, North and Up) frame of reference. Please refer to the following wiki article for more information https://en.wikipedia.org/wiki/Axes_conventions. Also mavros publishes rotations in quaternion.

Hence the driverPixhawkROSModule converts the rotations published by mavros into suitable units required by the Aerostack. Below is a piece of code on how it is done

```
        //deg,  mavwork reference frame
        geometry_msgs::Vector3Stamped RotationAnglesMsgs;

        tf::Quaternion q(msg.orientation.x, msg.orientation.y,
                         msg.orientation.z, msg.orientation.w);

       //make a matrix m of quaternion q
       tf::Matrix3x3 m(q);


       //convert quaternion to euler angels
        m.getEulerYPR(yaw, pitch, roll);
```


Pages 37

Introduction

- What is Aerostack?
- Getting Started
- License
- List of distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Catalog of Behaviors
- Memory of Beliefs

Mission Plan Specification

- TML Language
- Behavior tree
- Python Language

Tutorials [Beginners]

- Prepare the Mission:
  - Configure Map
  - Specify Mission Tasks
  - Configure Launcher
- Execute the Mission:
  - Launch Real Flight
  - Practical Tips for Real Flights
- Mission examples:
  - Simulated Quadrotor
  - AR Drone and Visual Markers

Tutorials [Advanced]

- How to program a new behavior
- Setup the Communication Network
- Wi Fi Communication with Multiple Drones
- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

More information

- Publications
- Vision & Aerial Robotics:
  - First prize at IMAV 2013
  - Stack pre-release 2013

Clone this wiki locally

```
RotationAnglesMsgs.vector.x = (double)(+1)*(roll)*180/M_PI;
RotationAnglesMsgs.vector.y = (double)(-1)*(pitch)*180/M_PI;
RotationAnglesMsgs.vector.z = (double)(-1)*(yaw)*180/M_PI;

//converting the yaw which is in the range from -phi to +phi to 0 to 360
degrees
if( RotationAnglesMsgs.vector.z < 0 )
RotationAnglesMsgs.vector.z += 360.0;

publishRotationAnglesValue(RotationAnglesMsgs);
```

The above code takes the rotations given by mavros in quaternion converts them into roll, pitch and yaw angles. This angles are then converted from ENU frame of reference to NED frame of reference and also converted from radians to degrees. The yaw angle which is in the range of -180 to +180 degrees is then converted in the range of 0 to 360 degrees. These angles are then finally published to the Aerostack.

3. Getting the Battery Voltage from the Pixhawk Using Macros

Mavros provides the current battery voltage using which the current battery percentage can be calculated. This percentage is then displayed on the user interface used for flying, which can be very important during a mission or normal flight to determine if the mission or flight can be carried out with a given percentage of battery. The code below determines how to convert the battery voltage to percentage required by the Aerostack.

```
droneMsgsROS::battery BatteryMsgs;

// Read Battery from navdata
BatteryMsgs.header   = msg.header;
// minimum voltage per cell used by the "digital battery checker" is
3.55 V
// msg.voltage is in V
float battery_voltage = (msg.voltage);

//Battery used in this case is 4 cell battery
BatteryMsgs.batteryPercent = ((battery_voltage - 14.8)/(16.8-14.8))*100;

publishBatteryValue(BatteryMsgs);
```

The above code is used to convert the battery voltage to battery percentage for a 4 cell battery and hence the lower voltage limit is 14.8V and higher voltage limit is 16.8V. If using a 3 cell battery the lower voltage limit should be 11.0V and higher voltage limit should be 12.6V.

4. Sending Commands to the Pixhawk Using Macros

There are several ways of sending commands to the pixhawk using mavros, by giving a certain position using position setpoints or directly setting the attitude and altitude using attitude setpoints.

This example uses roll, pitch, yaw and throttle commands from the Aerostack and sends to the pixhawk using mavros using setpoint_raw/attitude topic.

The Aerostack produces commands for any autopilot in terms of roll, pitch, dyaw (change in yaw) and throttle and in NED frame of reference.

The mavros on the other hand accepts commands in terms of roll, pitch, yaw and throttle but the roll, pitch and yaw have to be in quaternion and in ENU frame of reference. The throttle also has to be in the range from 0 to 1 counts. The driverPixhawkROSModule does all this conversions and finally sends the commands to the pixhawk using mavros.

The code below shows how commands are sent using mavros to the pixhawk.

https://github.com/Vision4U
Clone in Desktop

```
//message type for sending commands to mavros
 mavros_msgs::AttitudeTarget attitude_msg;

 //adding the current yaw angle to the dyaw command
 yaw_command = yaw_angle + msg.dyaw;

 //convert the values in radians and from NED frame(Aerostack) to ENU
frame (Mavros)
 roll_command  = (+1)*msg.roll*(M_PI/180.0);
 pitch_command = (-1)*msg.pitch*(M_PI/180.0);
 yaw_command   = (-1)*yaw_command*(M_PI/180.0);

 //convert euler angles to quaternion but in eigen form
 quaterniond =  mavros::UAS::quaternion_from_rpy(roll_command,
                                  pitch_command, yaw_command);

 //convert quaternion in eigen form to geometry_msg form
 tf::quaternionEigenToMsg(quaterniond, orientation);

 attitude_msg.orientation = orientation;

 attitude_msg.thrust = msg.thrust * THRUST_SCALE;

 drone_CtrlInput_publisher.publish(attitude_msg);
```

In the above code the dyaw command is first converted into yaw by adding the current yaw angle to it because the mavros needs commands in yaw and not in dyaw. The code then converts rotation angles from degrees to radians and then the angles to quaternions.

Then code then converts the throttle in the range defined in the Aerostack to the range of 0 to +1 required by the mavros by using the parameter THRUST_SCALE. The value of the THRUST_SCALE depends on the throttle range defined in midlevel controller in the Aerostack.

For example if the throttle is in the range of 0 to 32 Newtons in the Aerostack (midlevel controller), then the THRUST_SCALE has to be 1/32 in order to convert the throttle command in the range of 0 to +1 for macros.

Contact

We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page.

You can also contact the team support at the following address: aerostack.vision4uav@gmail.com

Publications

Edit New Page

Martin Molina edited this page 5 days ago · 13 revisions

Details about the functionality of Aerostack can be found in the following papers.

Core Papers

- J.L. Sanchez-Lopez, M. Molina, H. Bavle, C. Sampedro, R. A. Suarez-Fernandez, P. Campoy. (2017). A Multi-Layered Component-Based Approach for the Development of Aerial Robotic Systems: The Aerostack Framework. Journal of Intelligent & Robotic Systems, 1-27.
- J. L. Sanchez-Lopez, R. A. Suarez-Fernandez, H. Bavle, C. Sampedro, M. Molina, J. Pestana, P. Campoy (2016). AEROSTACK: An Architecture and Open-Source Software Framework for Aerial Robotics. ICUAS 2016, Arlington, USA.
- J.L. Sanchez-Lopez, J. Pestana, P. de la Puente, P. Campoy (2015): "A reliable Open-source System Architecture for the Fast Designing and Prototyping of Autonomous Multi-UAV Systems: Simulation and Experimentation". Journal of Intelligent and Robotic Systems. Online: Oct. 2015. Springer Netherlands. Online ISSN: 0921-0296. DOI: 10.1007/s10846-015-0288-x.

Other related papers:

- J. L. Sanchez-Lopez, J. Pestana, P. de la Puente, R. Suarez-Fernandez, P. Campoy. A System for the Design and Development of Vision-based Multi-robot Quadrotor Swarms. 2014 International Conference on Unmanned Aircraft Systems (ICUAS'14). Orlando (FL, USA). May 27-30, 2014. Pp: 640-648. DOI: 10.1109/ICUAS.2014.6842308.

Mission Planning

- M. Molina, R. A. Suarez-Fernandez, C. Sampedro, J. L. Sanchez-Lopez, P. Campoy, (2017): TML: a language to specify aerial robotic missions for the framework Aerostack. International Journal of Intelligent Computing and Cybernetics, Vol. 10 Issue: 4, pp.491-512.
- M. Molina, A. Diaz-Moreno, D. Palacios, R. A. Suarez-Fernandez, J.L. Sanchez-Lopez, C. Sampedro, H. Bavle, P. Campoy (2016): Specifying Complex Missions for Aerial Robotics in Dynamic Environments. International Micro Air Vehicle Conference and Competition, IMAV 2016. Beijing, China.
- C. Sampedro, H. Bavle, J. L. Sanchez-Lopez, R. A. Suarez-Fernandez, A. Rodriguez-Ramos, M. Molina, and P. Campoy (2016): A Flexible and Dynamic Mission Planning Architecture For UAV Swarm Coordination. ICUAS 2016, Arlington, USA.

Controllers

- J. Pestana, I. Mellado-Bataller, C. Fu, J. L. Sanchez-Lopez, I. F. Mondragon, P. Campoy. A General Purpose Configurable Controller for Indoors and Outdoors GPS-Denied Navigation for Multirotor Unmanned Aerial Vehicles. Journal of Intelligent and Robotic Systems. Jan. 2014. Vol 73, Issue 1-4, pp. 387-400. Springer Netherlands. Print ISSN: 0921-0296. Online ISSN: 1573-0409. DOI: 10.1007/s10846-013-9953-0
- J. Pestana, I. Mellado-Bataller, J. L. Sanchez-Lopez, C. Fu, I. F. Mondragon, P. Campoy. Floor Optical Flow Based Navigation Controller for Multirotor Aerial Vehicles. Robot 2013: First Iberian Robotics Conference (ROBOT 2013). Madrid (Spain). Nov. 28-29, 2013. Book title: Robot 2013: First Iberian Robotics Conference. Book subtitle: Advances in intelligent Systems and Computing. Vol. 253. Published year: 2014. Pp: 91-106. DOI: 10.1007/978-3-319-03653-3_8. Print ISBN: 978-3-319-03652-6. Online ISBN: 978-3-319-03653-3. Publisher: Springer International Publishing.

Pages 37

Introduction

- What is Aerostack?
- Getting Started
- License
- List of distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Catalog of Behaviors
- Memory of Beliefs

Mission Plan Specification

- TML Language
- Behavior tree
- Python Language

Tutorials [Beginners]

- Prepare the Mission:
  - Configure Map
  - Specify Mission Tasks
  - Configure Launcher
- Execute the Mission:
  - Launch Real Flight
  - Practical Tips for Real Flights
- Mission examples:
  - Simulated Quadrotor
  - AR Drone and Visual Markers

Tutorials [Advanced]

- How to program a new behavior
- Setup the Communication Network
- Wi Fi Communication with Multiple Drones
- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

More information

- Publications
- Vision & Aerial Robotics:
  - First prize at IMAV 2013
  - Stack pre-release 2013

Clone this wiki locally

https://github.com/Vision4U

Clone in Desktop

- J. Pestana, I. Mellado-Bataller, C. Fu, J. L. Sanchez-Lopez, I. Fernando Mondragon, P. Campoy. A General Purpose Congurable Navigation Controller for Micro Aerial Multirotor Vehicles. 2013 International Conference on Unmanned Aircraft Systems (ICUAS'13). Atlanta, Georgia (USA). May 28-31, 2013. DOI: 10.1109/ICUAS.2013.6564733. Print ISBN: 978-1-4799-0815-8. Publisher: IEEE. Pages: 557 - 564

Human Robot Interaction

- M. Molina, P. Frau, D. Maravall, J.L. Sanchez-Lopez, H. Bable, P. Campoy (2017): Human-Robot Cooperation in Surface Inspection Aerial Missions. The International Micro Air Vehicle Conference and Competition, IMAV 2017, Toulouse, France.

- R.A. Suarez Fernandez, J. L. Sanchez-Lopez, C. Sampedro, H. Bavle, M. Molina, and P. Campoy (2016): Natural User Interfaces for Human-Drone Multi-Modal Interaction. ICUAS 2016, Arlington, USA.

Sensoring and estimation

- H. Bavle, J. L. Sanchez-Lopez, A. Rodriguez-Ramos, C. Sampedro, P. Campoy, "A flight altitude estimator for multirotor uavs in dynamic and unstructured indoor environments", International Conference on Unmanned Aircraft Systems (ICUAS) 2017.

- Jose Luis Sanchez-Lopez, Victor Arellano-Quintana, Marco Tognon, Pascual Campoy, Antonio Franchi. Visual Marker based Multi-Sensor Fusion State Estimation. IFAC World Congress, Jul 2017, Toulouse, France. Proceedings of the 20th IFAC World Congress, 6p., 2017.

- A. Rodríguez-Ramos, C. Sampedro, A. Carrio, H. Bavle, R. A. Suarez-Fernandez, Z. Milosevic, P. Campoy, P. Z. Peng, D. F. Lin, "A monocular pose estimation strategy for uav autonomous navigation in gnss-denied environments", International Micro Air Vechicle Competition and Conference 2016, pp. 22-27, 2016.

Visual Servoing

- J. Pestana, J. L. Sanchez-Lopez, S. Saripalli, P. Campoy. Computer Vision Based General Object Following for GPS-denied Multirotor Unmanned Vehicles. 2014 American Control Conference (ACC 14). Portland (OR, USA). Jun. 4-6, 2014. Pp: 1886-1891. ISSN: 0743-1619. DOI: 10.1109/ACC.2014.6858831.

- J. Pestana, J. L. Sanchez-Lopez, S. Saripalli, P. Campoy. Vision based GPS-denied Object tracking and Following for Unmanned Aerial Vehicles. The 2013 (11th) IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR 2013). Linköping (Sweden). Oct 21-26, 2013. Best Paper Award Finalist. Pages: 1-6. Print ISBN: 978-1-4799-0879-0. DOI: 10.1109/SSRR.2013.6719359

Competitions

IMAV 2016:

- C. Sampedro, H. Bavle, A. Rodriguez-Ramos, A. Carrio, R. A. Suárez Fernández, J. L. Sanchez-Lopez, P. Campoy, "A fully-autonomous aerial robotic solution for the 2016 international micro air vehicle competition", Unmanned Aircraft Systems (ICUAS) 2017 International Conference on, 2017.

IARC 2014:

- J. L. Sanchez-Lopez, J. Pestana, J.-F. Collumeau, R. Suarez-Fernandez, P. Campoy, M. Molina. A Vision Based Aerial Robot solution for the Mission 7 of the International Aerial Robotics Competition. Unmanned Aircraft Systems (ICUAS), 2015 International Conference on. Denver, (CO, USA). 9-12 June 2015. Pp: 1391 – 1400. Print ISBN: 978-1-4799-6009-5. DOI: 10.1109/ICUAS.2015.7152435

IMAV 2013:

- J. Pestana, J. L. Sanchez-Lopez, P. de la Puente, A. Carrio, P. Campoy. A Vision-based Quadrotor Multi-Robot Solution for the Indoors Autonomy Challenge of the 2013 International Micro Air Vehicle Competition. Journal of Intelligent and Robotic Systems.

- J. Pestana, J. L. Sanchez-Lopez, P. de la Puente, A. Carrio, P. Campoy. A Vision-based Quadrotor Swarm for the participation in the 2013 International Micro Air Vehicle Competition. 2014 International Conference on Unmanned Aircraft Systems (ICUAS'14). Orlando (FL, USA). May 27-30, 2014. Pp: 617-622. DOI: 10.1109/ICUAS.2014.6842305.

- J. L. Sanchez-Lopez, J. Pestana, P. de la Puente, A. Carrio, P. Campoy. Visual Quadrotor Swarm for the IMAV 2013 Indoor Competition. Robot 2013: First Iberian Robotics Conference (ROBOT 2013). Madrid (Spain). Nov. 28-29, 2013. Book title: Robot 2013: First Iberian Robotics Conference. Book subtitle: Advances in intelligent Systems and Computing. Vol. 253. Published year: 2014. Pp: 55-63. DOI: 10.1007/978-3-319-03653-3_5. Print ISBN: 978-3-319-03652-6. Online ISBN: 978-3-319-03653-3. Publisher: Springer International Publishing.

Contact

We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page.

You can also contact the team support at the following address: aerostack.vision4uav@gmail.com

Aerostack Virtual Machine

Edit New Page

Martin Molina edited this page on 26 Oct 2017 · 3 revisions

In the following link you can download a Virtual Machine (Virtual Box VM) with Aerostack 1.0 (Gregale) installed: https://drive.google.com/open?id=0BziAte9fn_WzQk41cTJaYk85dlE

Features:

- Architecture: x64
- OS: Ubuntu 14.04
- ROS: Jade
- Aerostack

Host:

- Architecture: x64
- OS: Linux Ubuntu 14.04

User and password:

- User: aerostack
- Pass: paerostack

Pages 37

Introduction

- What is Aerostack?
- Getting Started
- License
- List of distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Catalog of Behaviors
- Memory of Beliefs

Mission Plan Specification

- TML Language
- Behavior tree
- Python Language

Tutorials [Beginners]

- Prepare the Mission:
  - Configure Map
  - Specify Mission Tasks
  - Configure Launcher
- Execute the Mission:
  - Launch Real Flight
  - Practical Tips for Real Flights
- Mission examples:
  - Simulated Quadrotor
  - AR Drone and Visual Markers

Tutorials [Advanced]

- How to program a new behavior
- Setup the Communication Network
- Wi Fi Communication with Multiple Drones
- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

More information

- Publications
- Vision & Aerial Robotics:
  - First prize at IMAV 2013
  - Stack pre-release 2013

Clone this wiki locally

https://github.com/Vision4U

Clone in Desktop

Contact

We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page.

You can also contact the team support at the following address: aerostack.vision4uav@gmail.com