Vision4UAV / Aerostack

Code | Issues 11 | Pull requests | Actions | Wiki | Security | Insigh

Home

Edit | New Page

Martin Molina edited this page on 23 Mar · 52 revisions

Latest news:

Pages 38

Getting started

- What is Aerostack
- Install Aerostack
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interfac
- Mission Plan Language
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

- Example Projects
- Program New Behavior
- Add New Hardware
- Setup Communications

International competitions

- 3rd place at MBZIRC 2(
- 2nd prize at IMAV 2017
- 4th place at IMAV 2016
- 1st prize at IMAV 2013

More information

- Publications
- Research Group CVAR

[March 5, 2020] We are very proud to announce that our group CVAR (Computer Vision and Aerial Robotics) from the "Universidad Politecnica de Madrid" won the third place of the Grand Challenge in the International Robotic Competition MBZIRC 2020 that took place in February 25, 2020. The Aerostack software framework was successfully used for building the system architecture of aerial robots that participated in the competition as it described in this publication.

Our group CVAR participated as the coordinator of SkyTeam, a combined team that also integrated members from Universidad Pablo Olavide and Poznan University of Technology. The group SkyTeam was in charge of Challenge 1 and Challenge 3. The group French National Center for Scientific Research (CNRS) was in charge of Challenge 2. Congratulations to all the members of these groups!

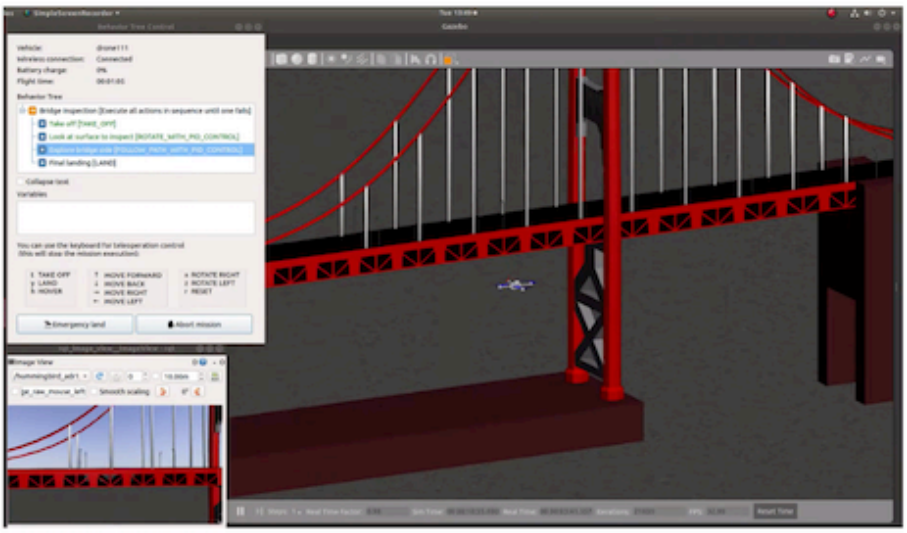
[July 30, 2019] We are pleased to announce the release of Aerostack **Version 3.0 Distribution Sirocco**. The development of this new version has received funding from the European Union's Horizon 2020 research and innovation programme under the project ROSIN. Compared to previous versions, it includes the following main features:

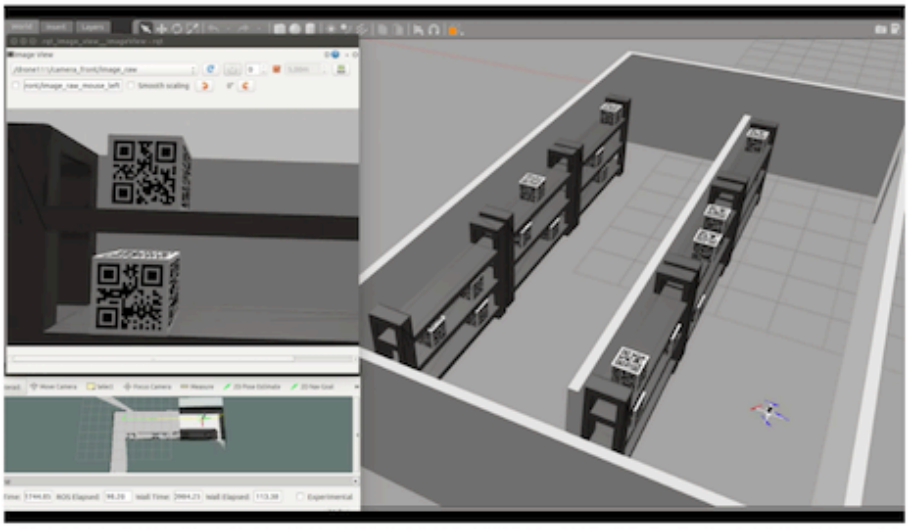
- **Standard methods:** Components of Aerostack have been updated to use standard ROS message types. Aerostack has been also updated to operate with ROS Melodic Morenia and Ubuntu 18.04.

Clone this wiki locally

```
https://github.com/Visi
```

- **New license and improved portability:** The software license of Aerostack has been changed to BSD-3. The installation method has also been improved using projects that install subsets of Aerostack packages.
- **Improved efficiency:** The performance efficiency and maintainability of the executive system has been improved with a more efficient design. The graphical user interface has been divided into separate more efficient and simpler ROS nodes.
- **New capabilities for navigation and communication:** New behaviors for positioning and navigation based on lidar sensors have been developed reusing existing ROS packages, and Aerostack has been extended to operate with Gazebo simulator (RotorS). New behaviors have been developed for communication between the robot and the operator and between the robot and other robots.
- **Learning documentation:** New test scenarios with increasing complexity have been developed. These scenarios have used to create new learning documentation to facilitate the use of Aerostack.





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.upm@gmail.com

The content of the Aerostack wiki is licensed under Creative Commons license CC BY 4.0

What is Aerostack

Martin Molina edited this page 3 days ago · 18 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, motion 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 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.

Pages 38

Getting started

- What is Aerostack
- Install Aerostack
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interfac
- Mission Plan Language
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

- Example Projects
- Program New Behavior
- Add New Hardware
- Setup Communications

International competitions

- 3rd place at MBZIRC 2(
- 2nd prize at IMAV 2017
- 4th place at IMAV 2016
- 1st prize at IMAV 2013

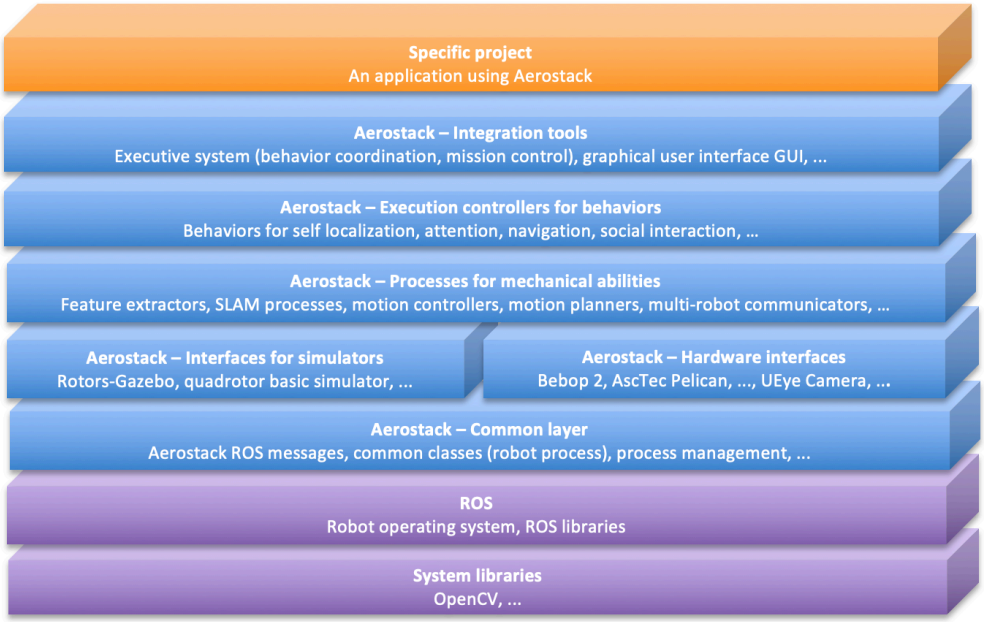
More information

- Publications
- Research Group CVAR

Clone this wiki locally

```
https://github.com/Visi
```

- **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
    - Parrot Bebop 2
    - DJI Matrice 100



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 (mainly programmed in C++). The components are 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 uses a reference architecture with a combination scheme, based on the composition of configurable building blocks, to develop architectures for specific aerial robotic systems.
- **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**. The license of the Aerostack kernel is BSD-3.

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.

Example of search and rescue mission using Aerostack (click to see video)

Type 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.
- 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.

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

- 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 a 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 algorithm for localization and mapping). The developer can reuse Aerostack but she/he might need to substitute the core algorithm used by the previous component.
- 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.upm@gmail.com

The following list includes Aerostack contributors:

Aerostack administrators:

- Hriday Bavle
- Alberto Rodelgo
- Pablo Santamaría

Research professors:

- Pascual Campoy
- Martin Molina
- Paloma de la Puente

Research assistants (current):

- Adrián Alvarez
- Abraham Carrera
- Adrián Carrio
- Miguel Fernández
- Lu Liang
- Alberto Rodelgo
- Alejandro Rodríguez-Ramos
- Javier Rodríguez-Vazquez
- Carlos Sampedro
- Jose Luis Sánchez-Lopez

- Ramon Suárez-Fernandez

Research assistants (past):

- Alberto Camporredondo
- Jean-François Collumeau
- Jesus Pestana Puerta
- Pablo Santofimia

Software developers:

- Javier Melero [2020], John Bradley [2020]
- Pablo Santamaría [2019-2020], Javier Cabrera [2019-2020], Adrián Cabrera [2019-2020], Marcos Tirado [2019], Laura Colomer [2019]
- Daniel del Olmo [2018-2019], José Martín-Antequera [2018-2019], Guillermo Echegoyen [2018], Raúl Cruz [2018-2019]
- Rafael Artiñano [2017-2018], Jorge Pascual [2017], Daniel Rabasco [2017], Jorge Sánchez [2017], Pedro Frau Amar [2017]
- Guillermo de Fermín [2016-2017], Carlos Valencia [2016-2017], Germán Quintero [2016]
- Adrian Díaz-Moreno [2015-2016], David Palacios [2015-2016], Enrique Ortiz [2015], Adrián Navarro [2015], Angel Luis González [2015], Laura García [2015], Oscar Fernández [2015]
- Yolanda de la Hoz [2014-2015], Marcos Bernal [2014-2015]
- Jorge Martin [2013-2014]

Funding support

The research work and development related to Aerostack has received the following public funding support:

- 2019-2021: Project "Comcise: Complex coordinated inspection and security missions by UAVs in cooperation with UGV". This project has received funding from the Ministry of Science, Innovation and Universities of Spain, reference RTI2018-100847-B-C21.

- 2018-2019: Focused technical project (FTP) "Aerostack 3.0: Aerial robotics framework for the industry". This FTP has received funding from the European Union's Horizon 2020 research and innovation programme under the project ROSIN with the grant agreement No 732287
- 2015-2017: Project "VA4UAV: Visual autonomy for UAV in Dynamic Environments". This project has received funding from the Ministry of Economy and Competitiveness of Spain, reference DPI2014-60139-R.

Besides public funding, Aerostack is being developed with the help of own funding of the research group CVAR-UPM.

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.upm@gmail.com

The content of the Aerostack wiki is licensed under Creative Commons license CC BY 4.0

Install Aerostack

Martin Molina edited this page on 8 Aug 2019 · 10 revisions

Hardware and software requirements

To install Aerostack, you need to meet these requirements:

- A non-ARM architecture computer.
- 2 GB of free disk space.
- (Good) internet connection.
- Linux Ubuntu 18.04 with ROS Melodic or Linux Ubuntu 16.04 with ROS Kinetic.
- The code requires a compiler that is compatible with the C++11 standard.

License agreement

Before the installation, you must read and accept the conditions defined by the software license

Installation method of a specific project

This installation method is appropriate to install a project developed using Aerostack. This method only installs the set of modules of Aerostack that are needed for the project. During the installation, the user has to provide the name of the project. Please, consult the following link to know the available example projects:

- Example projects

To install the project, perform the following steps:

1. Download the installation files:

Pages 38

Getting started

- What is Aerostack
- Install Aerostack
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interfac
- Mission Plan Language
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

- Example Projects
- Program New Behavior
- Add New Hardware
- Setup Communications

International competitions

- 3rd place at MBZIRC 2(
- 2nd prize at IMAV 2017
- 4th place at IMAV 2016
- 1st prize at IMAV 2013

More information

- Publications
- Research Group CVAR

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

2. Run the following installation script to install the project called `teleoperation_drone_simulator`, which is located in the folder `projects` (you can install any other available project of example projects):

```
$ ~/temp/install_project.sh projects/teleoperation_drone_simulator
```

Update the project

In order to update the modules of a project, execute the following script:

```
$ ./$AEROSTACK_STACK/update_project_modules.sh
```

Installation method (full version)

This method downloads and builds all the components of Aerostack.

1. Download the installation script file:

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

2. Execute the installation script:

```
$ ~/temp/install_full_aerostack.sh
```

3. Re-open the Terminal and the Aerostack is ready to be used.

Update Aerostack

In order to update Aerostack, execute the following commands:

Clone this wiki locally

```
https://github.com/Visi
```

```
    $ cd $AEROSTACK_STACK
    $ git pull origin master
    $ git submodule sync
    $ git submodule update --init
    $ cd $AEROSTACK_STACK/installation &&
./install_dependencies.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.upm@gmail.com

The content of the Aerostack wiki is licensed under Creative Commons license CC BY 4.0

License

Martin Molina edited this page on 25 Jul 2019 · 11 revisions

BY INSTALLING, COPYING OR OTHER USES OF SOFTWARE YOU ARE DECLARING THAT YOU AGREE WITH THE TERMS AND CONDITIONS OF THIS LICENSE AGREEMENT. IF YOU DO NOT AGREE WITH THE TERMS AND CONDITIONS, DO NOT INSTALL, COPY OR USE THE SOFTWARE

License agreement: 3-Clause BSD License

Aerostack - A Software Framework for Aerial Robotic Systems

Copyright (C) 2016-2019 Universidad Politécnica de Madrid

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

Pages 38

Getting started

- What is Aerostack
- Install Aerostack
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interfac
- Mission Plan Language
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

- Example Projects
- Program New Behavior
- Add New Hardware
- Setup Communications

International competitions

- 3rd place at MBZIRC 2(
- 2nd prize at IMAV 2017
- 4th place at IMAV 2016
- 1st prize at IMAV 2013

More information

- Publications
- Research Group CVAR

3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

IMPORTANT: Safety Warning and Disclaimer

It is important to note that this software is provided 'as is' without warranty of any kind. You are using this software at your own risk. The authors and their institutions decline any responsibility for any harm or damage of any kind (e.g., personal injuries and/or property damage) caused by the use of this software or by following these instructions.

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

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.

Clone this wiki locally

```
https://github.com/Visi
```

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.upm@gmail.com

The content of the Aerostack wiki is licensed under Creative Commons license CC BY 4.0

List of distributions

Edit New Page

Martin Molina edited this page on 6 Sep 2019 · 21 revisions

Distribution	Release date	Main features	Tested platforms	Requir
Version 3.0 Sirocco Distribution	July 2019	BSD license. New behaviors (e.g., navigation based on lidar and occupancy grids). Efficient execution control system. Modular graphical user interface. Installation of Aerostack using projects.	DJI Matrice, Pixhawk, Parrot Bebop 2.0, Parrot AR Drone 2.0	Ubuntu ROS M
Version 2.1 Levant Distribution	February 2018 PDF wiki	Improvements and extensions of graphical user interface (e.g., new control panel, first person	Parrot AR Drone 2.0, Pixhawk, Parrot Bebop 2 (with	Ubuntu ROS Ki

Pages 38

Getting started

- What is Aerostack
- Install Aerostack
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interfac
- Mission Plan Language
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

- Example Projects
- Program New Behavior
- Add New Hardware
- Setup Communications

International competitions

- 3rd place at MBZIRC 2(
- 2nd prize at IMAV 2017
- 4th place at IMAV 2016
- 1st prize at IMAV 2013

More information

- Publications
- Research Group CVAR

		view) and new behaviors	SLAM-dunk).	
Version 2.0 Levant Distribution	October 2017 PDF wiki	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 ROS Ki
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 ROS Ja

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.upm@gmail.com

Clone this wiki locally

https://github.com/Visi

The content of the Aerostack wiki is licensed under Creative Commons license CC BY 4.0

Aerostack Architecture

Martin Molina edited this page on 25 Jul 2019 · 12 revisions

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

Main components

The next figure shows the main components of the Aerostack architecture. In the figure, circles represent data processing units (or processes in short) that are implemented as ROS nodes. Processes are organized in three main layers: interface, functional and executive.

The interface layer includes processes that help interact with the world. They include processes that receive sensor data or command robot actuators, besides communication processes with the human operator and other robots.

The other layers (functional and executive) are common components of multilayer architectures of autonomous systems. The Aerostack architecture also uses a standard communication channel, which is implemented with ROS message types that are common in aerial robotics. This channel facilitates process interoperability and also, it helps reuse components of the functional and the executive layers across different types of aerial platforms.

Pages 38

Getting started

- What is Aerostack
- Install Aerostack
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interfac
- Mission Plan Language
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

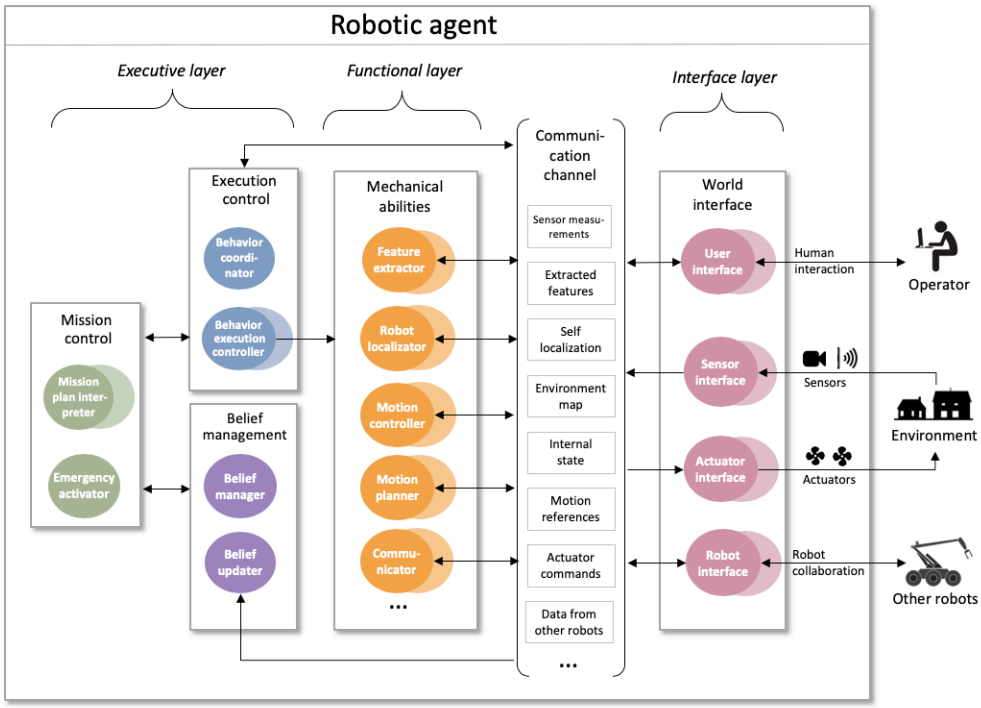
- Example Projects
- Program New Behavior
- Add New Hardware
- Setup Communications

International competitions

- 3rd place at MBZIRC 2(
- 2nd prize at IMAV 2017
- 4th place at IMAV 2016
- 1st prize at IMAV 2013

More information

- Publications
- Research Group CVAR



The functional layer of the Aerostack architecture includes processes for mechanical abilities of aerial robots. Aerostack provides a software library to implement these processes and the developer can use and combine them to build a particular robotic system architecture. Examples of these components are:

- feature extractors that read simple states of sensors or implement vision recognition algorithms (e.g., recognizer of ArUco markers or QR codes),
- motion controllers (e.g., PID controllers for pose control or speed control and trajectory controllers),
- processes that perform self-localization and mapping (SLAM),
- motion planners that generate obstacle-free paths to reach destination points, and
- methods for communicating with other agents (other robots or human operators).

The third layer of the Aerostack architecture is the executive layer and includes the processes that make up the executive system. The main function of these processes is to execute the mission plan by activating and monitoring the execution of the mechanical abilities of the functional layer.

Clone this wiki locally

```
https://github.com/Visi
```

The executive layer in Aerostack creates a logical interface that helps specify mission plans. This layer relieves the human operator from having to specify unnecessary execution details that are common for similar tasks and provides user error protection by verifying the correct syntax and semantics of written commands. The executive layer also verifies that a mission is executed as expected. For example, the executive system checks in advance that each task to be executed is consistent with the situation of the environment, preventing the robot from performing wrong or dangerous behaviors. The executive system also detects the presence of unexpected events and coordinate multiple decisions about what the robot must perform (e.g., deliberative or reactive decisions).

Architecture organization in processes

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). Each process of Aerostack is implemented as a ROS node.

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

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.upm@gmail.com

The content of the Aerostack wiki is licensed under Creative Commons license CC BY 4.0

Graphical User Interface

Abraham Carrera edited this page on 26 Jul 2019 · 19 revisions

Graphical user interface

This page describes the graphical user interface (GUI) of Aerostack. The GUI is used by the operator to (1) supervise the behavior of the aerial robot from the ground station and (2) configure details of the mission to be executed by the robot. The GUI presents user-friendly graphical views and visual aids to facilitate the realization of these tasks.

The GUI of Aerostack is provided as a collection of individual applications for different tasks. Each of these applications is implemented as a separate ROS node that can be launched individually. This separation is useful, for example, to minimize the computational consumption during the execution of a mission, because only the different parts of the GUI are launched by the user when they are needed.

The following sections present these interface applications separated into three groups:

1. Individual viewers
2. Mission control windows
3. Integrated viewer
4. Graphical editors

Besides the GUI, Aerostack also provides an alphanumeric user interface to monitor basic parameters and to send commands to the robot using the keyboard.

1. Individual viewers

Pages 38

Getting started

- What is Aerostack
- Install Aerostack
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interfac
- Mission Plan Language
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

- Example Projects
- Program New Behavior
- Add New Hardware
- Setup Communications

International competitions

- 3rd place at MBZIRC 2(
- 2nd prize at IMAV 2017
- 4th place at IMAV 2016
- 1st prize at IMAV 2013

More information

- Publications
- Research Group CVAR

This section shows the list of individual viewers, i.e., applications that present a window to monitor certain features of the robot behavior. Each application may be launched using the ROS command:

```
 $ roslaunch <package-name> <launch-filename> --wait
[args]
```

Vehicle dynamics viewer

This window is used to consult the values of the pose and speed parameters.

To launch the vehicle dynamics viewer (for example, for the robot namespace `drone7`), the command is:

```
 $ roslaunch vehicle_dynamics_viewer
vehicle_dynamics_viewer.launch --wait
robot_namespace:=drone7
```

Execution viewer

This window shows two kinds of components: the memory of beliefs, and a list of the behaviors currently executing accompanied by the arguments that have been passed as a parameter at the time of defining the mission. It also offers the option of modifying the existing beliefs and behaviors by right-clicking on them.

Clone this wiki locally

https://github.com/Visi

To launch the execution viewer (for example, for the robot namespace `drone7`), the command is:

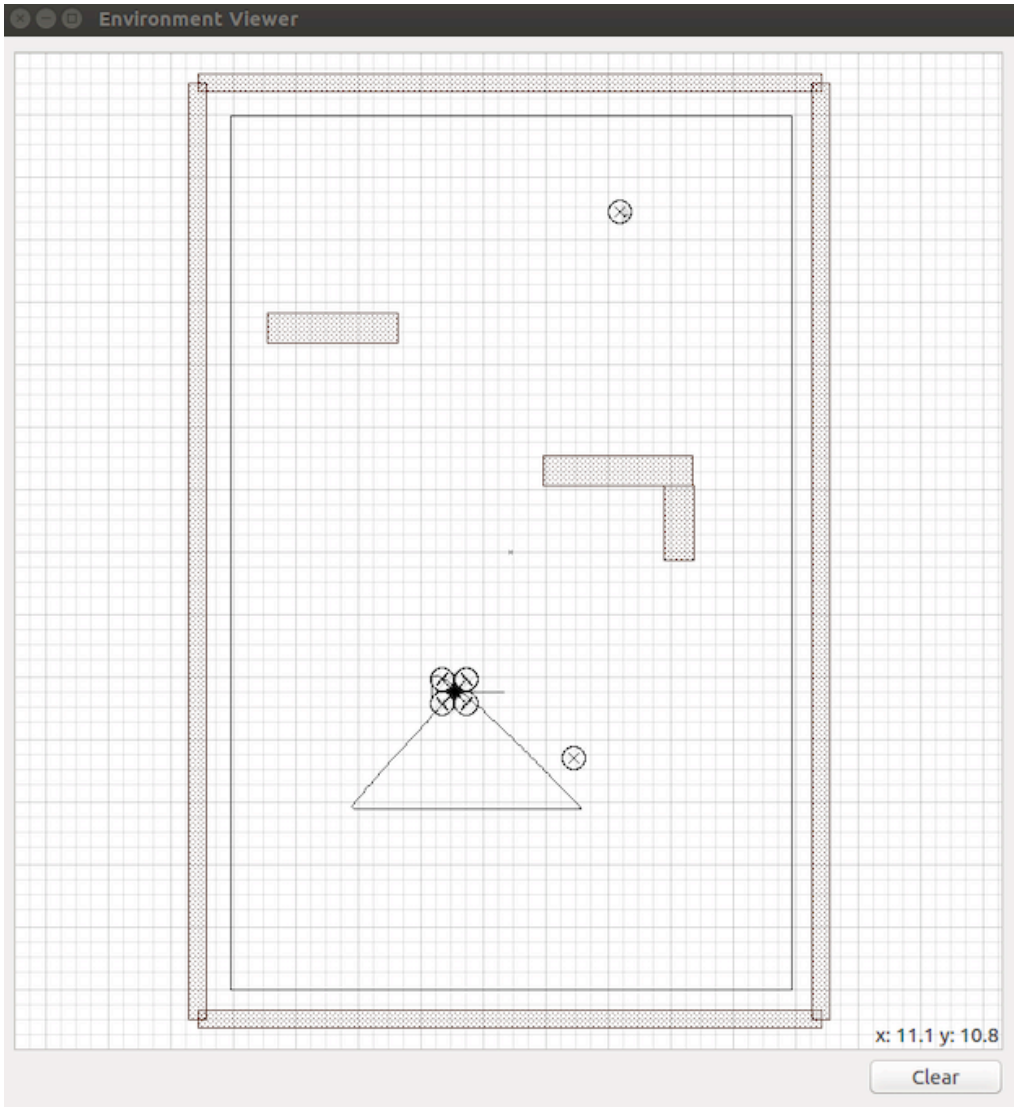
```
$ roslaunch execution_viewer execution_viewer.launch --wait robot_namespace:=drone7
```

Environment Viewer

2D representation of the elements in the scene of the mission (obstacles, poles, visual markers...) and the drone, showing its trajectory updated in real time. To clean up the path followed by the vehicle press the button clear.

To launch the environment viewer (for example, for the robot namespace `drone1`), the command is:

```
$ roslaunch environment_viewer environment_viewer.launch --wait robot_namespace:=drone1 robot_config_path:=$AEROSTACK_STACK/configs/drone1
```

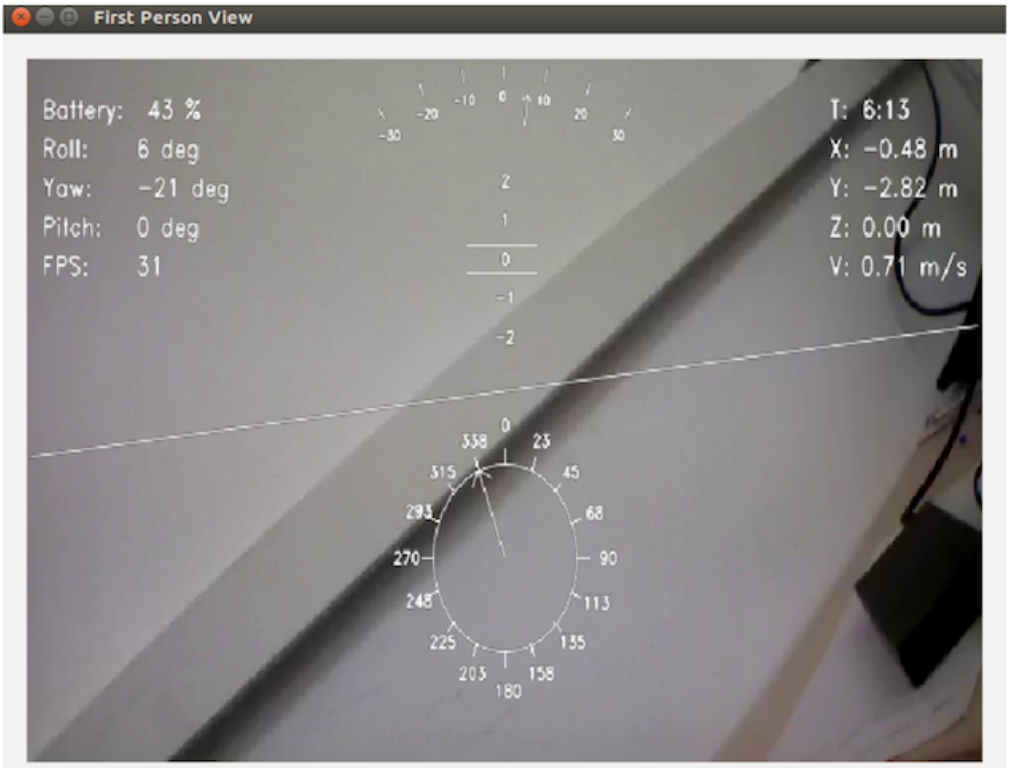


First Person Viewer

Head-up visualization of the images captured by the frontal camera, with some dynamic parameters over it. You can press the button in the upper left corner to switch to extended mode.

To launch the first person viewer (for example, for the robot namespace `drone7`), the commands are:

```
 $ roslaunch first_view_process first_view.launch --wait
robot_namespace:=drone7
 $ roslaunch first_person_view_process
first_person_view.launch --wait robot_namespace:=drone7
```

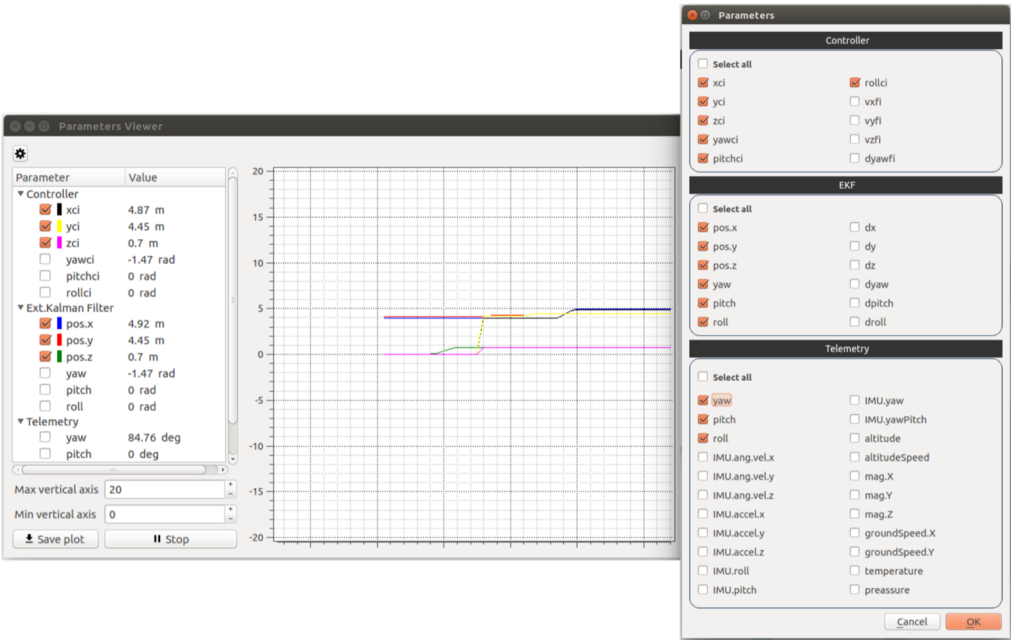


Parameters Viewer

Through this component, the user can monitor those variables that s/he has previously selected in its side menu. They are shown in the form of a line chart updated in real time. It also shows the value of the complete set of parameters. To make it easier to read, it uses a legend in which a different color is automatically assigned to each of the variables. It allows the operator to stop the representation as well as export the graphic built up to that moment in different formats. It is very useful to verify that the course of the mission is happening as planned.

To launch the parameters viewer (for example, for the robot namespace `drone7`), the command is:

```
 $ roslaunch parameters_viewer parameters_viewer.launch --wait robot_namespace:=drone7
```

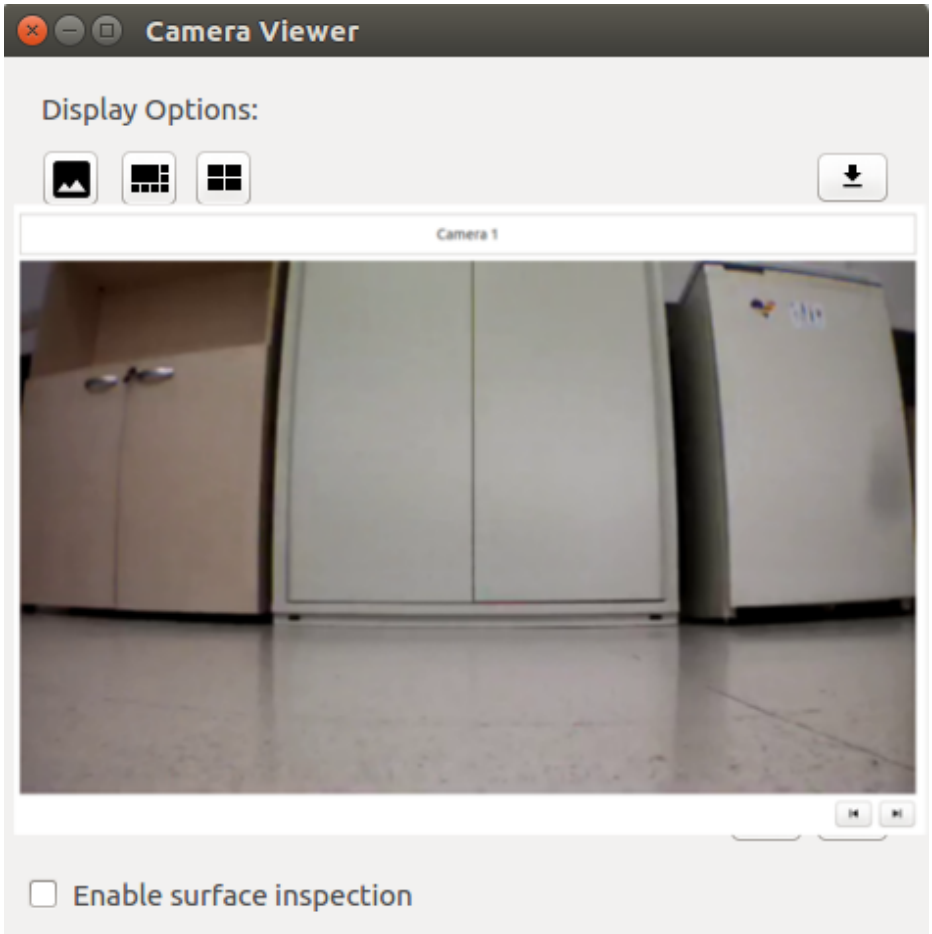


Camera Viewer

This widget shows images and videos captured by all the cameras installed in the vehicle. We can visualize different cameras at the same time by using one of the display options in the upper left part. We can also save these images in a file.

To launch the camera viewer (for example, for the robot namespace `drone7`), the command is:

```
 $ roslaunch camera_viewer camera_viewer.launch --wait
robot_namespace:=drone7
```



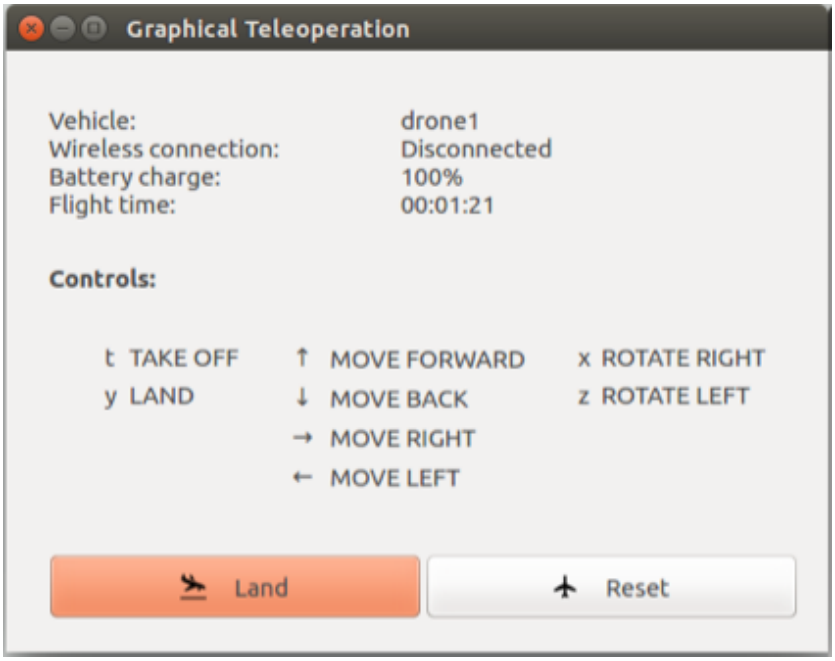
2. Mission control windows

Teleoperation control window

Window to control the operation of the vehicle using the keyboard. The behavior of the robot is fully controlled by the keys pressed by the user.

To launch the teleoperation control window (for example, for the robot namespace `drone1`), the command is:

```
 $ roslaunch teleoperation_control_window
teleoperation_control_window.launch --wait
robot_namespace:=drone1
```

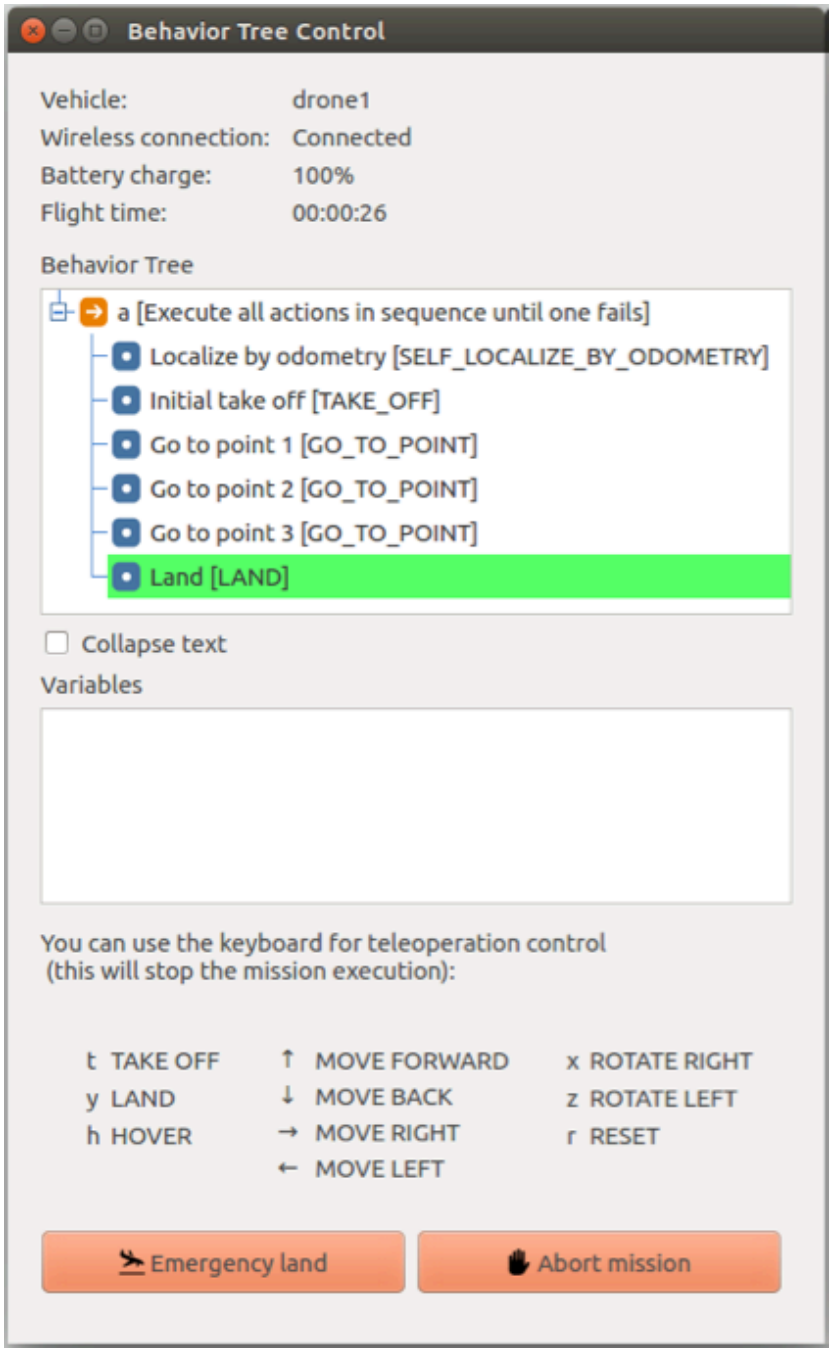


Behavior tree control window

Aerostack presents graphically the behavior tree using a hierarchy browser. To execute the behavior tree, the interpreter loads the yaml file with the mission plan and it follows its structure to generate a sequence of activations and deactivations of robot behaviors. During the execution, the control window shows graphically the dynamic evolution of the execution. The control window also presents the current values of variables used by the behavior tree. The interpreter also facilitates preemptive interaction between the user and the robot, i.e., the user can interrupt the mission at any point and continue the execution in another node of the behavior tree.

To launch the window to control a mission plan written as behavior trees (for example, for the robot namespace `drone7` and the mission configuration folder $AEROSTACK_STACK/projects/configs/mission), the command is:

```
 $ roslaunch behavior_tree_interpreter
behavior_tree_interpreter.launch --wait
robot_namespace:=drone7
mission_configuration_folder:=$AEROSTACK_STACK/projects/con
```

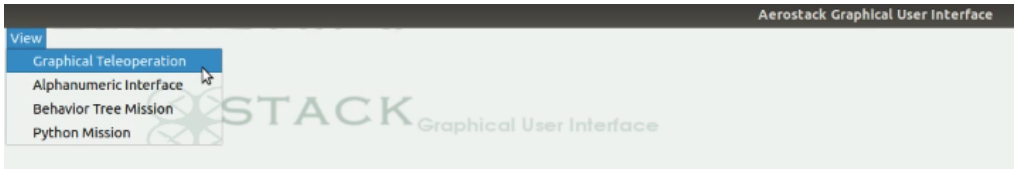


3. Integrated viewer

The integrated viewer may be used to show layouts that present prefixed groups of windows. The integrated viewer is launched with the following ROS command:

```
$ roslaunch integrated_viewer integrated_viewer.launch
```

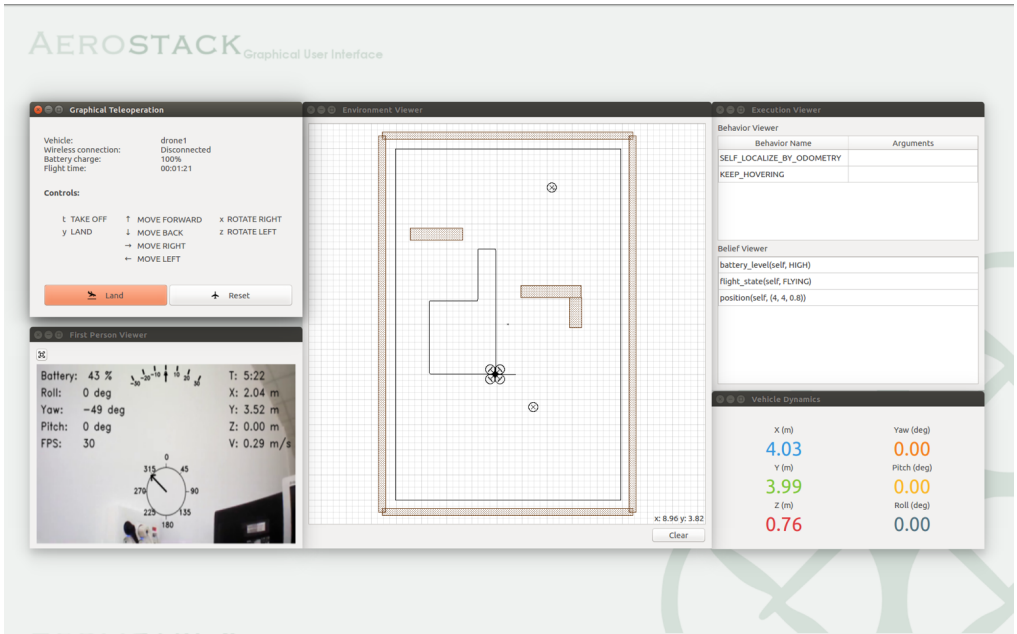
Then, a screen background is presented with a menu bar in the upper part.



The option **View** shows a menu to launch the different layouts that are described below.

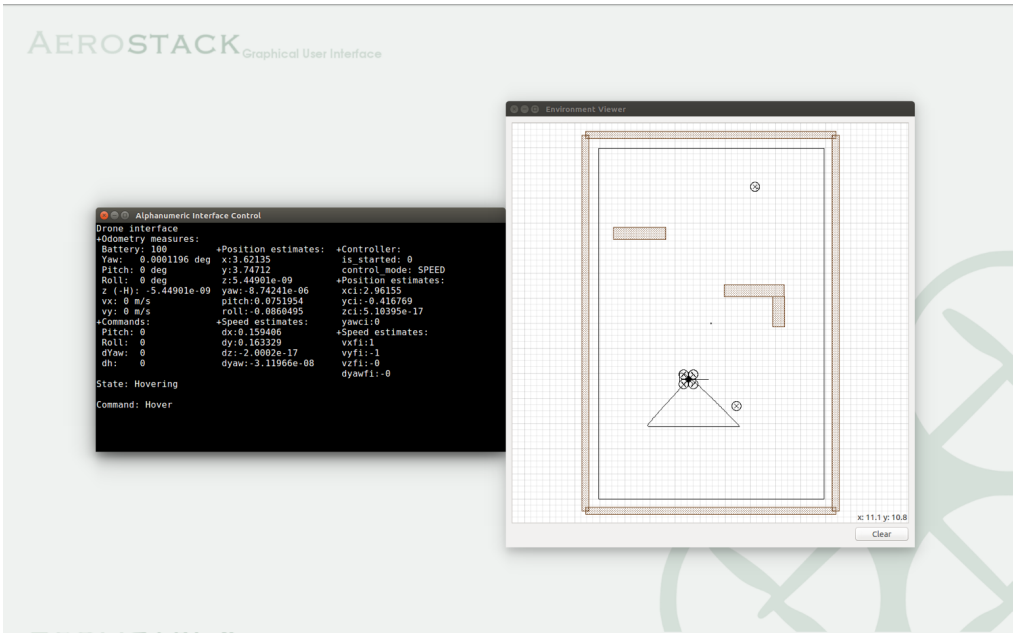
Graphical teleoperation

Layout oriented to control the operation of the vehicle using the keyboard. The behavior of the robot is fully controlled by the keys pressed by the user while the Graphical Teleoperation **window is active**. Besides that dialog, this layout also includes the Vehicle Dynamics Viewer, the Execution Viewer, the Environment Viewer and the First Person Viewer.



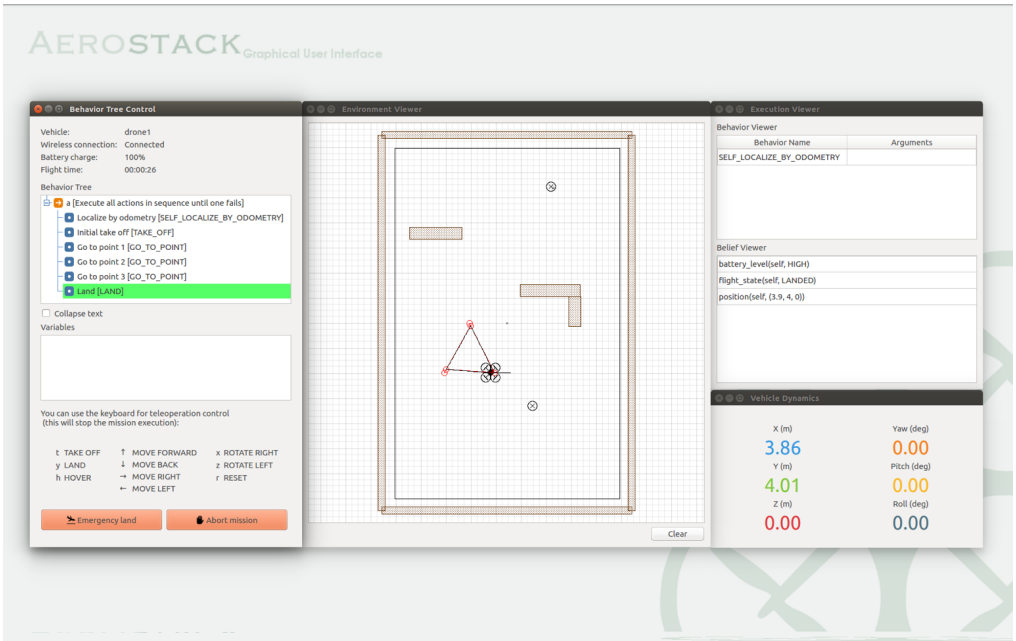
Alphanumeric interface

As we have mentioned before, the Alphanumeric Interface is integrated inside the Graphical User Interface as a prefixed layout. It shows the console and the Environment Viewer. The console accepts commands through the keyboard and sends them to the actuators of the aerial robot.



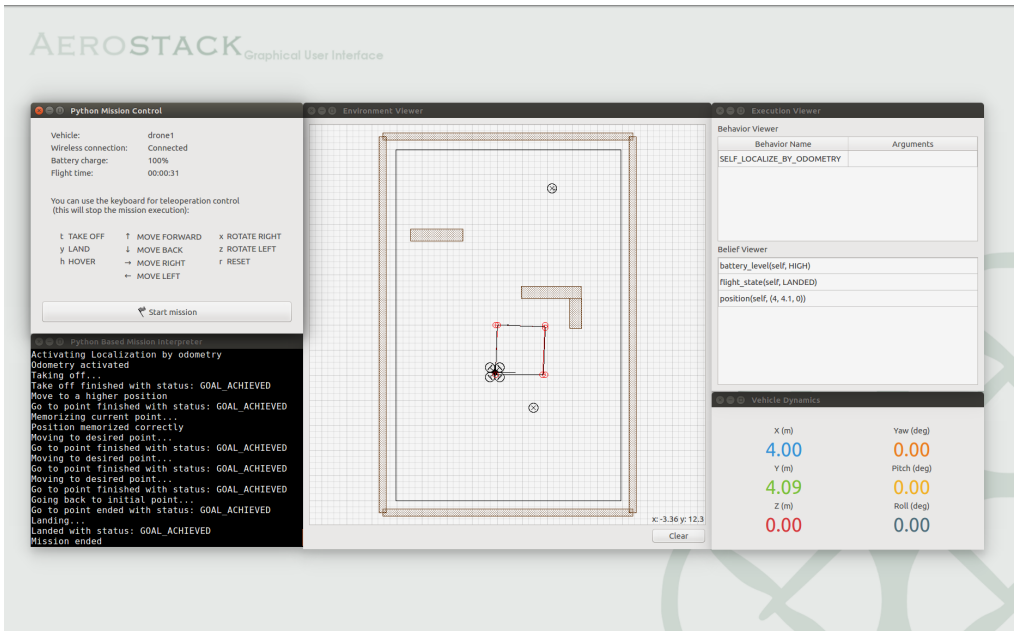
Behavior tree mission

The user executes the behavior tree previously charged using Edit Behavior Tree. This widget also captures commands through the keyboard but their use **stops the execution of the mission**. The mission can be started from any node right-clicking on the desired one. It is composed of the Behavior Tree Control, the Environment Viewer, the Execution Viewer and the Vehicle Dynamics. For more information visit the Behavior Tree Control section.



Python mission

With this layout, the operator starts and controls the execution of the selected Python Mission. He can also use the keyboard interrupting the execution of the mission. It shows the Python Mission Control, the Pỳthon Based Mission Interpreter (feedback of the mission), the Environment Viewer, the Execution Viewer and the Vehicle Dynamics Viewer.



4. Graphical editors

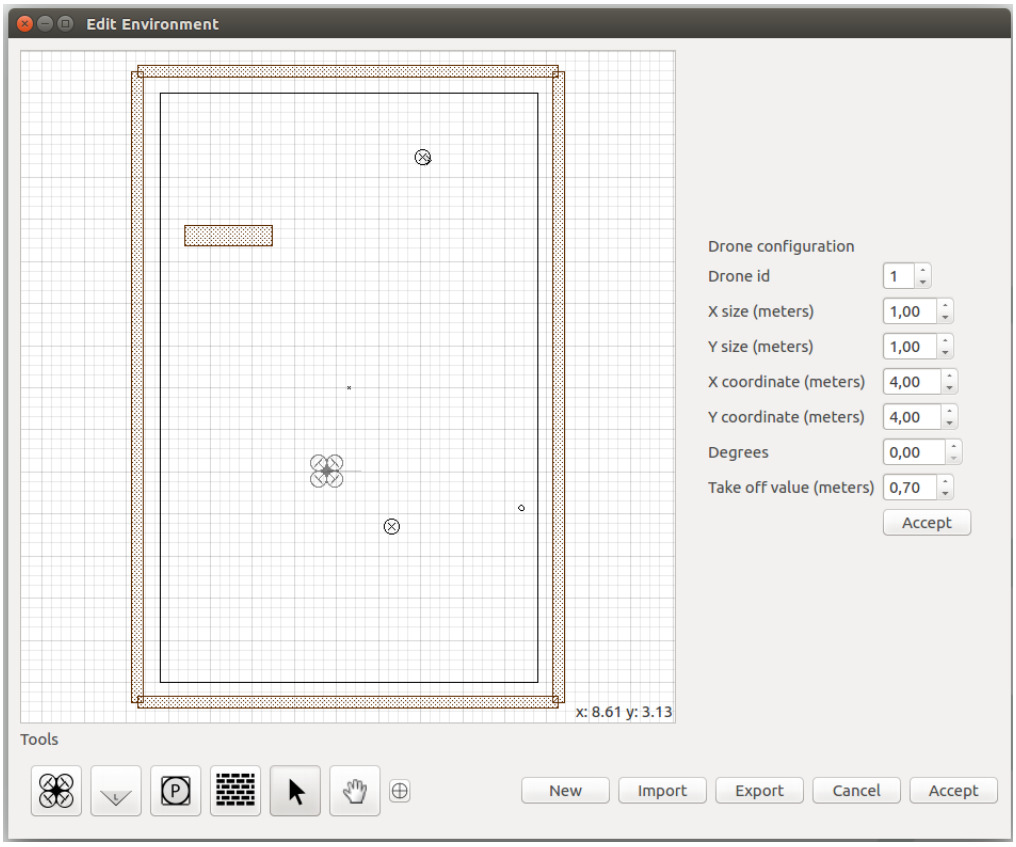
The editors may be used by the operator to change the configuration of the mission.

Environment map editor

The user can use this editor to design a new map or import and change an existing one.

To launch the environment map editor (for example, for the robot namespace `drone1`), the command is:

```
 $ roslaunch environment_editor environment_editor.launch
--wait  robot_namespace:=drone1
robot_config_path:=$AEROSTACK_STACK/configs/drone1
```

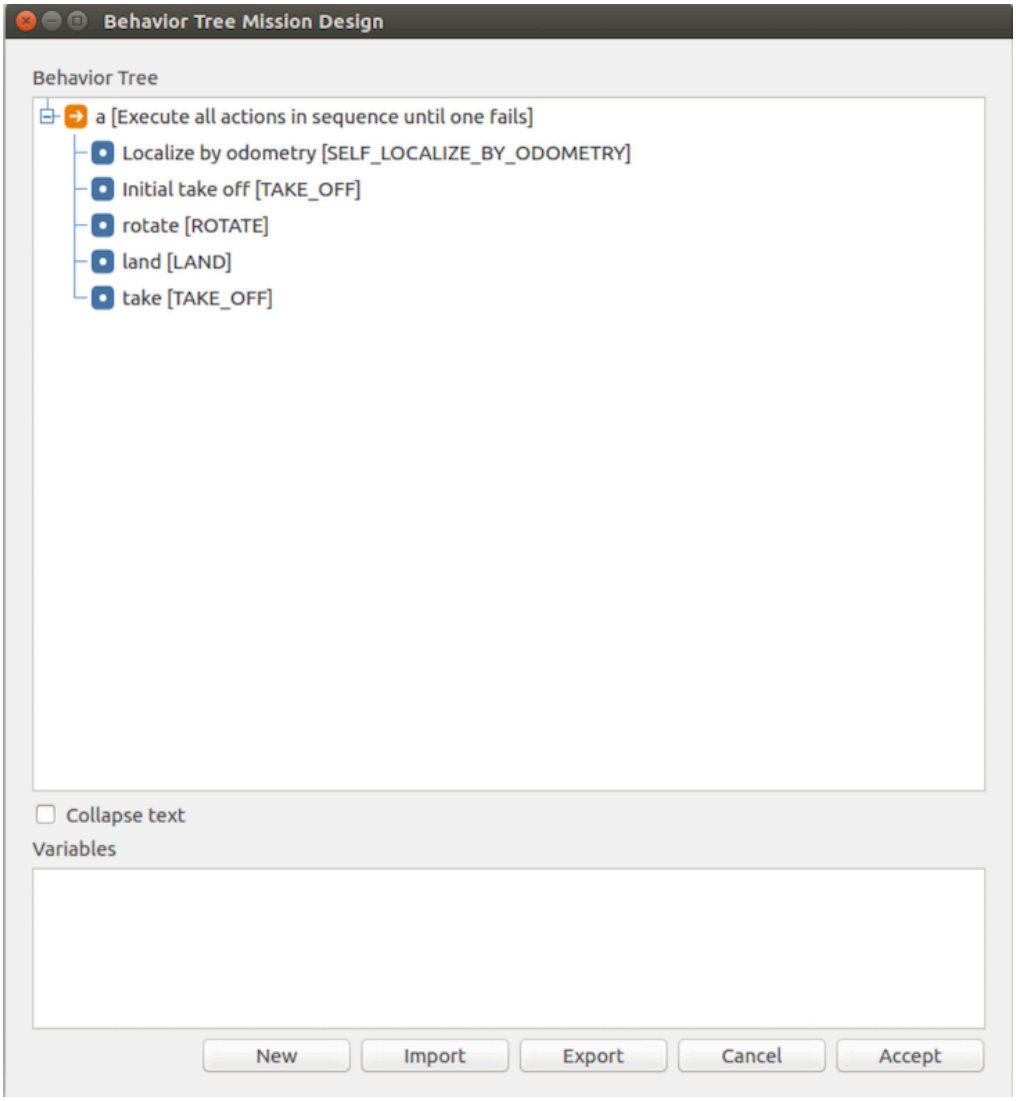


See more details in the following page:

- Configure the Environment Map

Behavior tree editor

The operator can use this editor to create or import and change a mission plan represented as a behavior tree.



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.upm@gmail.com

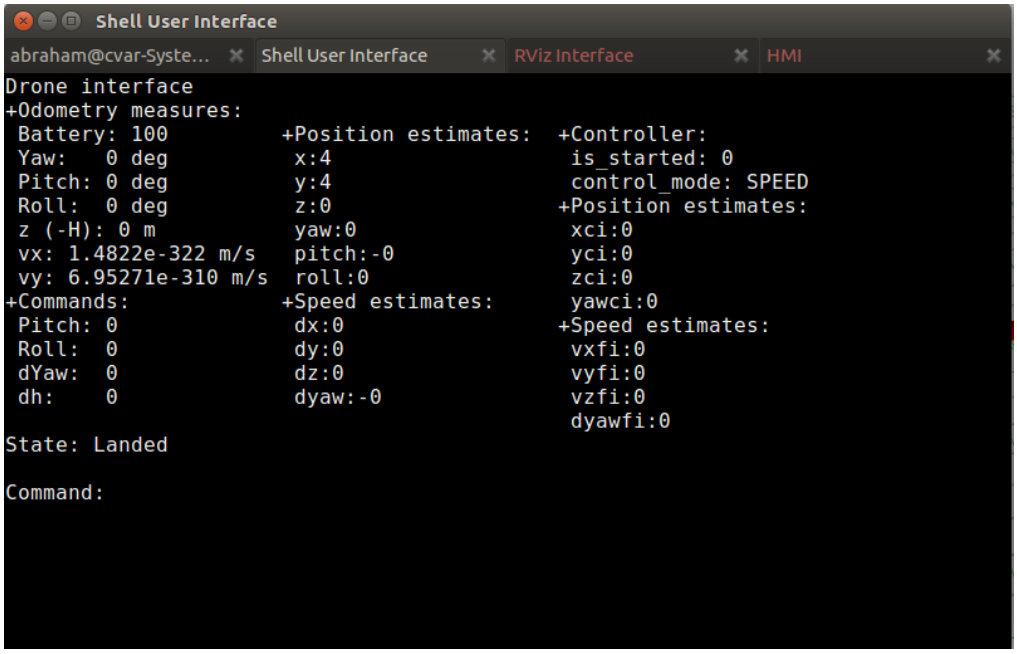
The content of the Aerostack wiki is licensed under Creative Commons license CC BY 4.0

Alphanumeric User Interface

Martin Molina edited this page on 22 Feb 2018 · 1 revision

The alphanumeric interface is an efficient process that presents information about the robot behavior and can be used to teleoperate the robot.

This user interface is presented in a window terminal (with the tab called *Shell User Interface*) after launching Aerostack. You can see several parameters that give information about the situation of the drone and its state.



The operator can send different commands using the following keys:

Key	Command
t	*take off*: take off
y	*land*: hover > land > stop controller
space	*emergency stop:* emergency stop > stop controller

Pages 38

Getting started

- What is Aerostack
- Install Aerostack
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interfac
- Mission Plan Language
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

- Example Projects
- Program New Behavior
- Add New Hardware
- Setup Communications

International competitions

- 3rd place at MBZIRC 2(
- 2nd prize at IMAV 2017
- 4th place at IMAV 2016
- 1st prize at IMAV 2013

More information

- Publications
- Research Group CVAR

bar	
h	*hover*: hover > stop controller
m	*move*: move (set flying mode)
s	*set commands to zero*: [open loop] hover, [closed loop] (non internal - with controller) hover
q	*move upwards*: [open loop] upwards speed command; [speed, position control] move altitude reference upwards
a	*move downwards*: same as before but downwards
z	*(yaw) turn counter-clockwise*: [open loop] turn with speed command; [any controller] move yaw reference counter-clockwise
x	*(yaw) turn clockwise*: same as before but clockwise
****	*set yaw reference to 0* (look parallel to x-axis)
right arrow	*move right, or to[+]y_axis:* [open loop] or [speed command] or [position command], ignored during trayectory control
left arrow	*move left: [open loop] or [speed command] or [position command]*, ignored during trayectory control
up arrow	*move frontwards; [open loop] or [speed command] or [position command]*, ignored during trayectory control
down arrow	*move backwards: [open loop] or [speed command] or [position command]*, ignored during trayectory control
j	*reset EKF*: stop controller > reset EKF
k	*stop EKF:* stop controller > stop EKF
l	*start EKF*: start EKF and setInitDroneYaw
u	*reset controller*
i	*stop controller*: stop controller > drone_hover
o	*start controller*: start EKF > start controller
7	*start speed controller*: if controller started > drone_move > start controller (speed)

Clone this wiki locally

```
https://github.com/Visi
```

8	*start position controller*: if controller started > drone_move > start controller (position)
9	*start trajectory controller* (3D quadrilateral) using rel_trajectory_ref channel

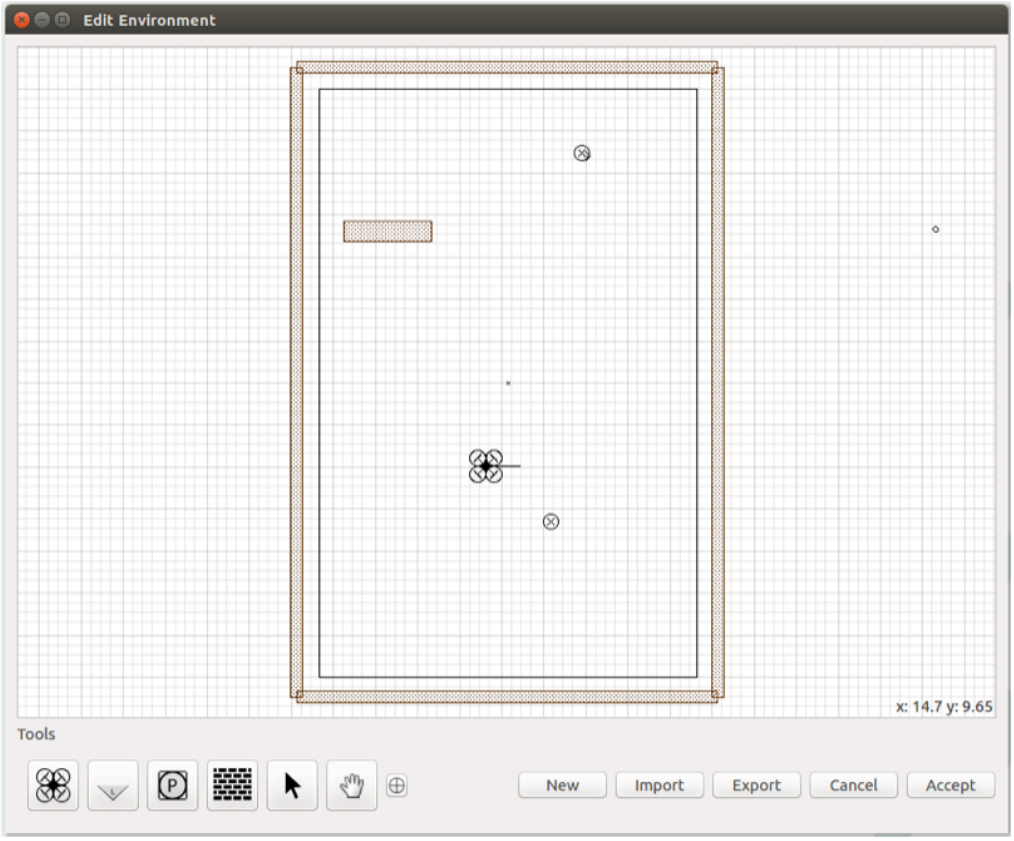
Contact: We thank and welcome any suggestion or comment about Aerostack. For any question or bug report you can read and/or write at the issues page. You can also contact the team support at the following address: aerostack.upm@gmail.com

The content of the Aerostack wiki is licensed under Creative Commons license CC BY 4.0

Configure the Environment Map

Martin Molina edited this page 15 seconds ago · 4 revisions

This page describes how the operator can create the environment map for a mission by using the graphical user interface. 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' button in the upper bar and then select the option 'Environment Map'. It will pop-up a window as the following one.



Pages 38

Getting started

- What is Aerostack
- Install Aerostack
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interfac
- Mission Plan Language
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

- Example Projects
- Program New Behavior
- Add New Hardware
- Setup Communications

International competitions

- 3rd place at MBZIRC 2(
- 2nd prize at IMAV 2017
- 4th place at IMAV 2016
- 1st prize at IMAV 2013

More information

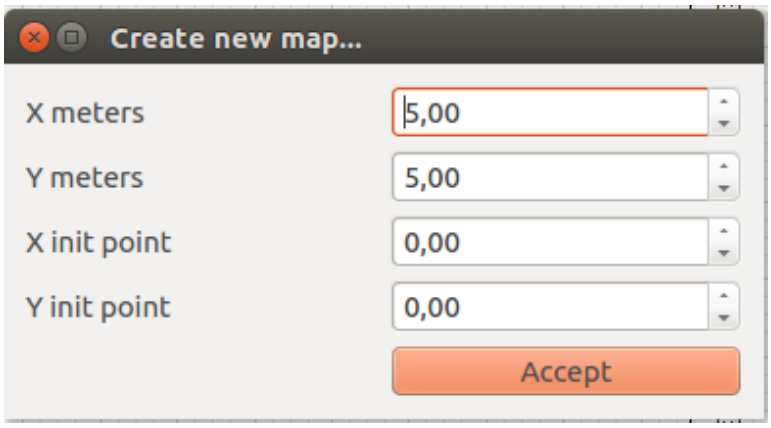
- Publications
- Research Group CVAR

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

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 new 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.

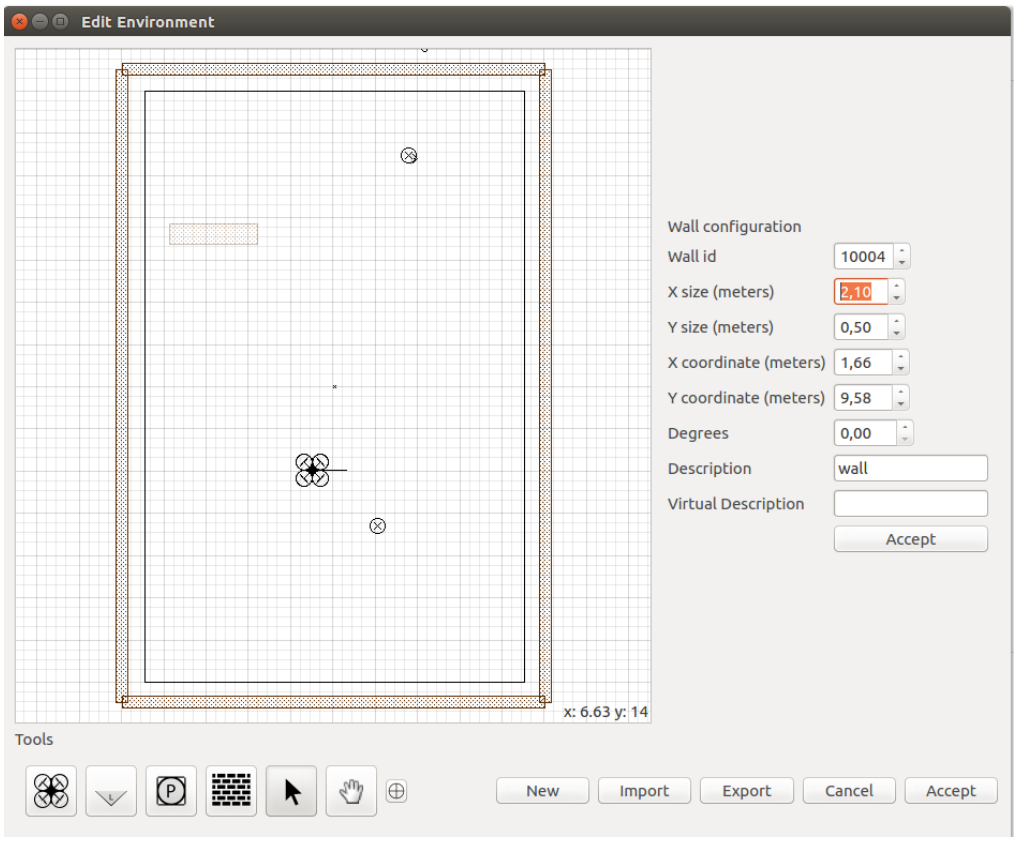
Editing an existing object

Clone this wiki locally
https://github.com/Visi

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.



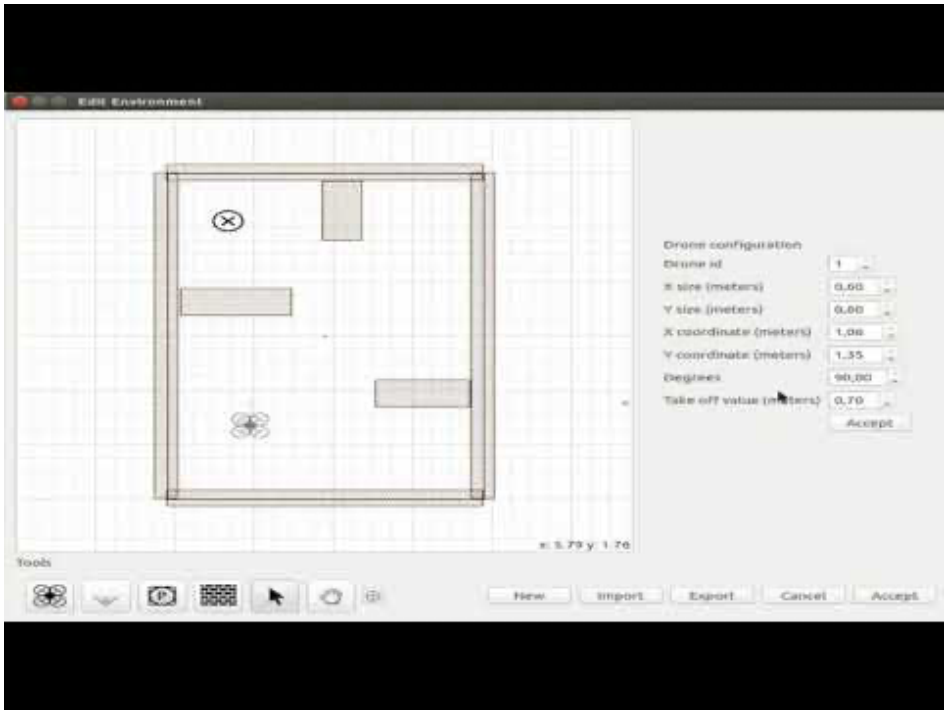
When you're finished editing your new Environment Map you can click the 'Accept' button. The new map will be ready to be used.

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.

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



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.upm@gmail.com

The content of the Aerostack wiki is licensed under Creative Commons license CC BY 4.0

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).

Pages 38

Getting started

- What is Aerostack
- Install Aerostack
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interfac
- Mission Plan Language
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

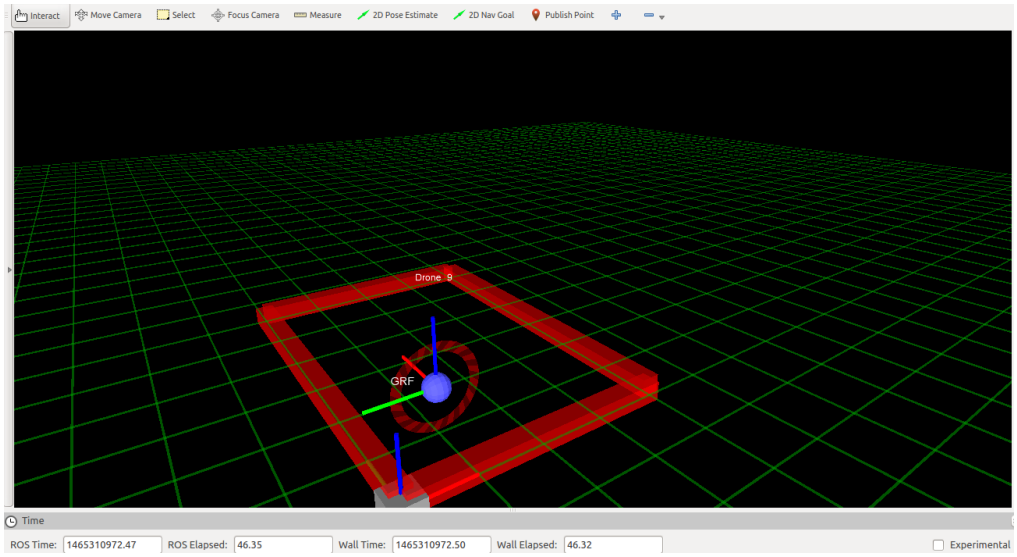
- Example Projects
- Program New Behavior
- Add New Hardware
- Setup Communications

International competitions

- 3rd place at MBZIRC 2(
- 2nd prize at IMAV 2017
- 4th place at IMAV 2016
- 1st prize at IMAV 2013

More information

- Publications
- Research Group CVAR



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

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>
```

Clone this wiki locally

https://github.com/Visi

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>

<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>`.

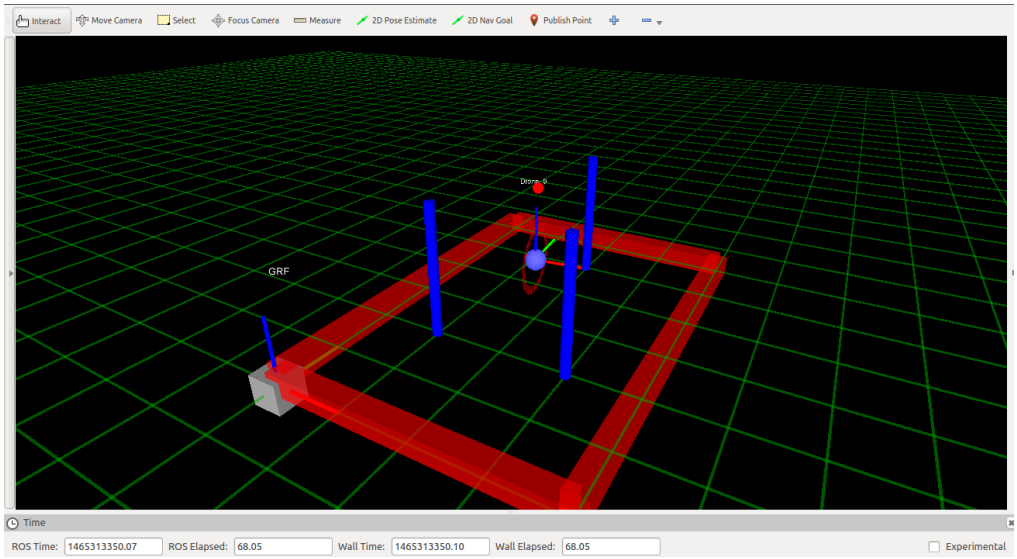
Also in this file the user can specify the fixed visual markers in the map.

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"
<Aruco id="10" x="2.80" y="2.0"  z="1.3" roll="0" pitch="0"
<Aruco id="11" x="3.0"  y="2.20" z="1.3" roll="0" pitch="0"
<Aruco id="12" x="3.20" y="2.0"  z="1.3" roll="0" pitch="0"

<!-- Pole 2 -->
<Aruco id="13" x="1.0"  y="1.80" z="1.3" roll="0" pitch="0"
<Aruco id="14" x="0.80" y="2.0"  z="1.3" roll="0" pitch="0"
<Aruco id="15" x="1.0"  y="2.20" z="1.3" roll="0" pitch="0"
<Aruco id="16" x="1.20" y="2.0"  z="1.3" roll="0" pitch="0"

<!-- Pole 3 -->
<Aruco id="25" x="2.0"  y="4.80" z="1.3" roll="0" pitch="0"
<Aruco id="26" x="1.80" y="5.0"  z="1.3" roll="0" pitch="0"
<Aruco id="27" x="2.0"  y="5.20" z="1.3" roll="0" pitch="0"
<Aruco id="28" x="2.20" y="5.0"  z="1.3" roll="0" pitch="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" y
<Aruco id="62" x="0.0" y="4.11" z="1.3" roll="0" pitch="0" y
```

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" y
<Aruco id="70" x="3.11" y="6.0" z="1.3" roll="0" pitch="0" y
```

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" y
<Aruco id="62" x="0.0" y="4.11" z="1.3" roll="0" pitch="0" y
```

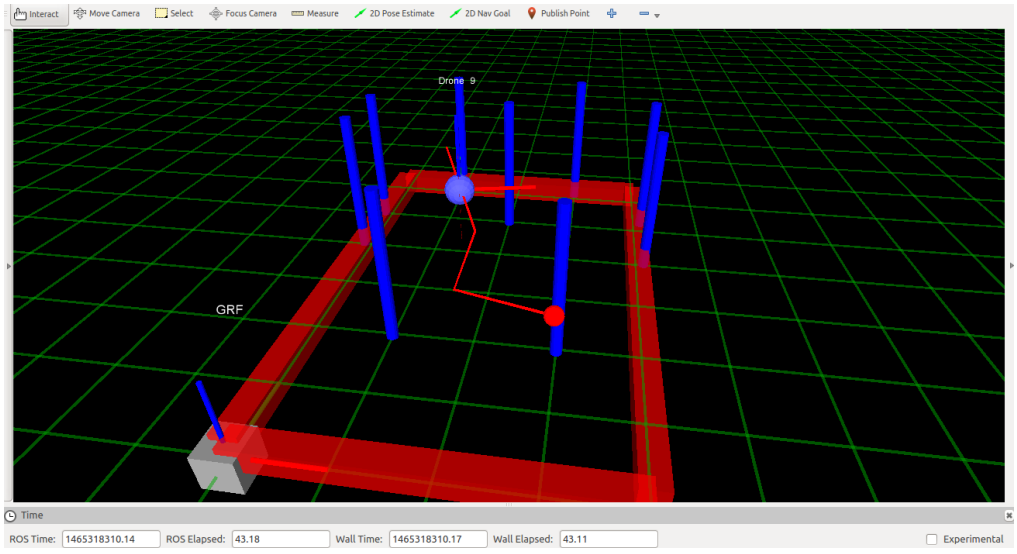
```
<!-- Landmark 2 -->
<Aruco id="63" x="0.0" y="4.89" z="1.3" roll="0" pitch="0" y
<Aruco id="64" x="0.0" y="5.11" z="1.3" roll="0" pitch="0" y

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

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

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

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

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.upm@gmail.com

The content of the Aerostack wiki is licensed under Creative Commons license CC BY 4.0

Vision4UAV / **Aerostack**

Code Issues 11 Pull requests Actions **Wiki** Security Insigh

Mission Plan Languages

Edit New Page

Martin Molina edited this page on 24 Jul 2019 · 2 revisions

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

- Behavior tree. In this case, a graphical editor can be used to specify missions using a visual modeling language.
- Python language. Python language can be used (with a library of functions) to specify a mission plan.

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.upm@gmail.com

The content of the Aerostack wiki is licensed under Creative Commons license CC BY 4.0

Pages 38

Getting started

- What is Aerostack
- Install Aerostack
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interfac
- Mission Plan Language
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

- Example Projects
- Program New Behavior
- Add New Hardware
- Setup Communications

International competitions

- 3rd place at MBZIRC 2(
- 2nd prize at IMAV 2017
- 4th place at IMAV 2016
- 1st prize at IMAV 2013

More information

- Publications
- Research Group CVAR

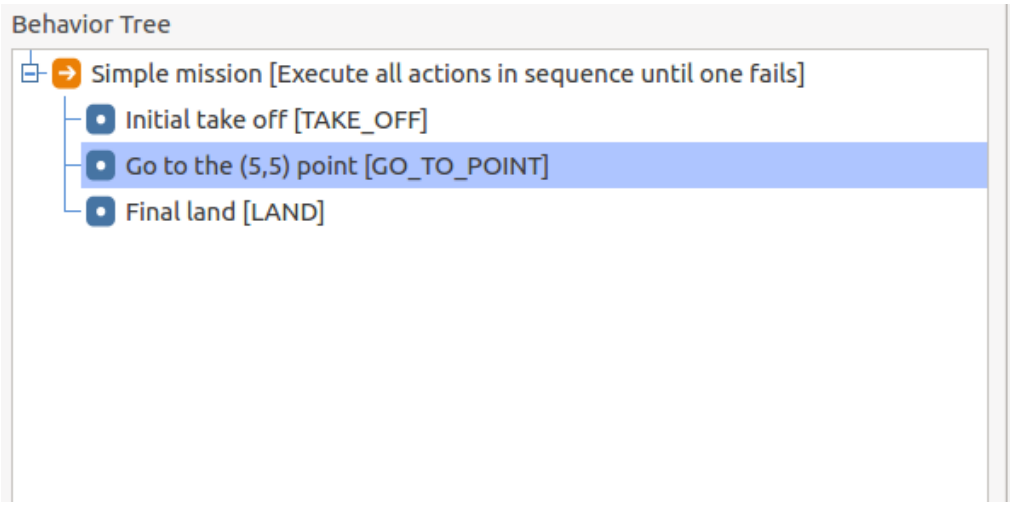
Clone this wiki locally

```
https://github.com/Visi
```

Behavior Tree

Martin Molina edited this page on 1 Feb · 12 revisions

A behavior tree is a visual modelling language that uses a graphical notation to represent the behavior of a system. Aerostack provides a graphical editor to create a mission plan as a behavior tree.



Node types

Each behavior tree is represented with a hierarchy of executable nodes. Nodes can succeed or fail during the execution of the mission plan. 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

Pages 38

Getting started

- What is Aerostack
- Install Aerostack
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interfac
- Mission Plan Language
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

- Example Projects
- Program New Behavior
- Add New Hardware
- Setup Communications

International competitions

- 3rd place at MBZIRC 2(
- 2nd prize at IMAV 2017
- 4th place at IMAV 2016
- 1st prize at IMAV 2013

More information

- Publications
- Research Group CVAR

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 of the tree establish the control regime (e.g., a sequence, a loop, etc.). The types of intermediate nodes are the following:

- Sequence. This node executes the child nodes in sequence and succeeds when all the children succeed. Otherwise it fails.
- Selector. This node executes the child nodes in sequence and succeeds when one of the children succeeds. If none of them succeeds, it fails.
- Parallel. This node executes 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 N - S.
- Repeat. 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 sequential execution of child nodes until a child node fails. This node always succeeds.
- 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.

Launching the behavior tree editor

Clone this wiki locally

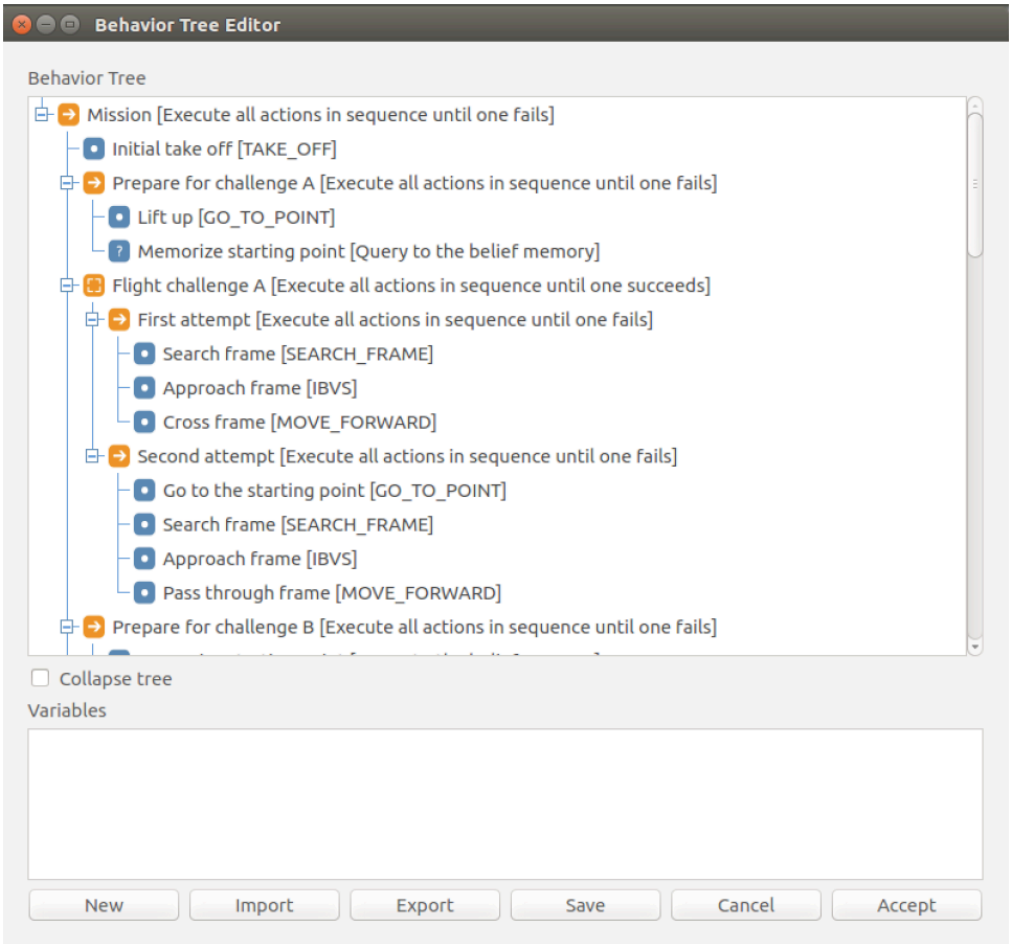
https://github.com/Visi

To launch the behavior tree editor, you can execute a script file with the following content (for example, for the drone drone111 and the project `basic_behavior_tree_drone_simulator`):

```
#!/bin/bash

NUMID_DRONE=111
export
AEROSTACK_PROJECT=${AEROSTACK_STACK}/projects/basic_behavio


. ${AEROSTACK_STACK}/setup.sh

gnome-terminal \
--tab --title "Executive Coordinator" --command "bash -c
\"
roslaunch behavior_coordinator_process
behavior_coordinator_process.launch --wait \
  drone_id_namespace:=drone$NUMID_DRONE \

behavior_catalog_path:=${AEROSTACK_PROJECT}/configs/mission/
 \
  drone_id:=$NUMID_DRONE \
  my_stack_directory:=${AEROSTACK_PROJECT};
exec bash\"" \
--tab --title "Behavior Tree Editor" --command "bash -c
\"
roslaunch behavior_tree_editor
behavior_tree_editor.launch --wait \
  robot_namespace:=drone$NUMID_DRONE \
  drone_id:=$NUMID_DRONE \
  my_stack_directory:=${AEROSTACK_PROJECT} \

mission_config_path:=${AEROSTACK_PROJECT}/configs/drone$NUM

exec bash\""  &
```

Editing a behavior tree

The operator can specify a mission plan using the behavior tree editor of Aerostack. Aerostack presents graphically the behavior tree using a hierarchy browser. The editor uses standard edition mechanisms to allow users create a behavior tree by adding, modifying or deleting nodes of the tree. The editor provides guidance and assistance to users presenting valid options on menu bars and checking the presence of user errors in texts describing parameter values or belief expressions. The created behavior tree is stored in yaml file (with the name behavior_tree_mission_file.yaml).



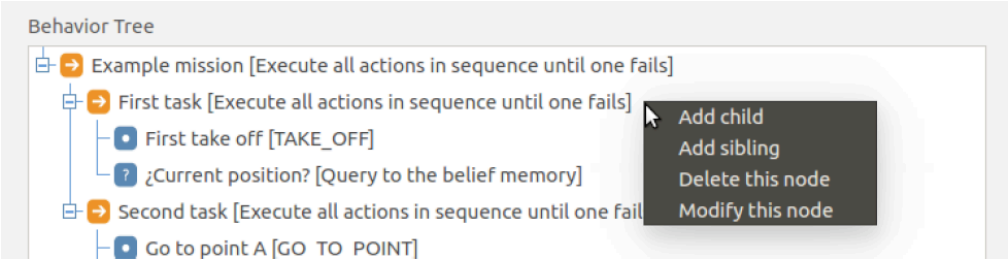
The editor window has a set of buttons at the bottom with the following functionality:

- New: Creates the root node of a new behavior tree with a name given by the user.
- Import: Imports a behavior tree defined in a yaml file selected by the user.
- Export: Stores the behavior tree in a yaml file selected by the user.
- Save: Saves the behavior tree in the file "behavior_tree_mission_file.yaml" (in the configuration folder).

- Cancel: Finishes the edition.
- Accept: Saves the behavior tree in the file "behavior_tree_mission_file.yaml" (in the configuration folder) and the edition is finished.

When the operator clicks the right mouse button on any of the nodes, a context menu is displayed showing a list of possible actions:

- Add child: Adds a new child node.
- Add sibling: Adds a new sibling node.
- Delete this node: Deletes a node from the tree.
- Modify this node: Allows to modify the characteristics of the node.



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]
```

Behavior trees in Aerostack can use variables to communicate information between leaf nodes. For example, the user can create a query node called "Identify current position" with the belief expression:

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

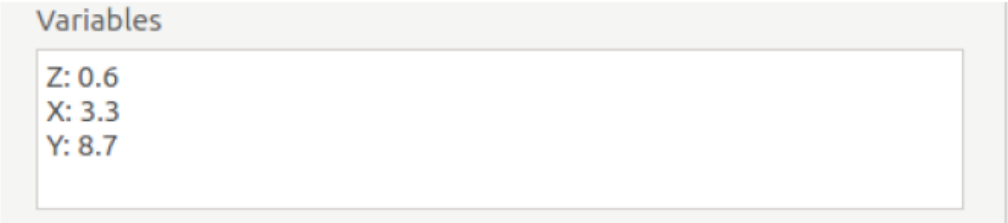
In this expression, X, Y, and Z are preceded by a question mark (?) to represent that they are variables. When this node is executed, the expression matches the corresponding predicate in the belief memory. For example, if the belief memory has the predicate `position(self,(3.3, 8.7, 0.6))`, the variables get the values X = 3.3, Y = 8.7, Z = 0.6.

The values of these variables can be used by other nodes of the mission plan. For instance, there may be an operation node in another place of the same mission plan that uses the behavior 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 will be substituted during the execution by the value that the variable has in this moment. Considering the previous example, this means that the robot will requested to go to a destination with the coordinates (3.3, 8.7, 0.6).

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.upm@gmail.com

The content of the Aerostack wiki is licensed under Creative Commons license CC BY 4.0

Python Language

Edit New Page

Martin Molina edited this page on 25 Jul 2019 · 5 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

Pages 38

Getting started

- What is Aerostack
- Install Aerostack
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interfac
- Mission Plan Language
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

- Example Projects
- Program New Behavior
- Add New Hardware
- Setup Communications

International competitions

- 3rd place at MBZIRC 2(
- 2nd prize at IMAV 2017
- 4th place at IMAV 2016
- 1st prize at IMAV 2013

More information

- Publications
- Research Group CVAR

Clone this wiki locally

https://github.com/Visi

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 deactivate a recurrent behavior, use the function `deactivateBehavior()`. When a behavior needs to be deactivated, you should set the Unique identifier (uid) as the parameter of the function. The UID is given when a behavior is activated by using the `activateBehavior()` function.

```
import executive_engine_api as api

def runMission():
  ack, uid = api.activateBehavior('SELF_LOCALIZE_BY_ODOMETR

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

  api.deactivateBehavior(uid)
```

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: `queryBelief()` 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

  success, unification = api.queryBelief('my_belief(?X, ?),
  if success: # Always check if query was successful before
    my_string = unification['X']

    api.removeBelief('my_belief(%s, something_else)' % my_s
    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():
  ack, uid = api.activateBehavior('SELF_LOCALIZE_BY_ODOMETR
  api.executeBehavior('TAKE_OFF')

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

  api.inhibitBehavior(uid)

  ack, uid = api.activateBehavior('SELF_LOCALIZE_BY_VISUAL_
  objectFound = False
  # Start searching for object
  while not objectFound:
    nextPoint = [10*random.random(), 10*random.random(), 0.
    api.executeBehavior('GO_TO_POINT', coordinates=nextPoin
    objectFound = api.trueBelief('position(object_1, ?)')

  success, unification = api.queryBelief('position(object_1
  if success:
    objectPosition = [unification['X'], unification['Y'], u

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

    api.executeBehavior('GO_TO_POINT', coordinates=targetPo

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

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

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

```
api.deactivateBehavior(uid)

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.upm@gmail.com

The content of the Aerostack wiki is licensed under Creative Commons license CC BY 4.0

Python Language

Martin Molina edited this page on 25 Jul 2019 · 5 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

Pages 38

Getting started

- What is Aerostack
- Install Aerostack
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interfac
- Mission Plan Language
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

- Example Projects
- Program New Behavior
- Add New Hardware
- Setup Communications

International competitions

- 3rd place at MBZIRC 2(
- 2nd prize at IMAV 2017
- 4th place at IMAV 2016
- 1st prize at IMAV 2013

More information

- Publications
- Research Group CVAR

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 deactivate a recurrent behavior, use the function `deactivateBehavior()`. When a behavior needs to be deactivated, you should set the Unique identifier (uid) as the parameter of the function. The UID is given when a behavior is activated by using the `activateBehavior()` function.

```
import executive_engine_api as api

def runMission():
  ack, uid = api.activateBehavior('SELF_LOCALIZE_BY_ODOMETR

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

  api.deactivateBehavior(uid)
```

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: `queryBelief()` 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

  success, unification = api.queryBelief('my_belief(?X, ?),
  if success: # Always check if query was successful before
    my_string = unification['X']

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

Clone this wiki locally

`https://github.com/Visi`

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():
  ack, uid = api.activateBehavior('SELF_LOCALIZE_BY_ODOMETR
  api.executeBehavior('TAKE_OFF')

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

  api.inhibitBehavior(uid)

  ack, uid = api.activateBehavior('SELF_LOCALIZE_BY_VISUAL_
  objectFound = False
  # Start searching for object
  while not objectFound:
    nextPoint = [10*random.random(), 10*random.random(), 0.
    api.executeBehavior('GO_TO_POINT', coordinates=nextPoin
    objectFound = api.trueBelief('position(object_1, ?)')

  success, unification = api.queryBelief('position(object_1
  if success:
    objectPosition = [unification['X'], unification['Y'], u

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

    api.executeBehavior('GO_TO_POINT', coordinates=targetPo

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

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

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

```
api.deactivateBehavior(uid)

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.upm@gmail.com

The content of the Aerostack wiki is licensed under Creative Commons license CC BY 4.0

Catalog of Behaviors

Alberto Rodelgo edited this page on 2 Dec 2019 · 28 revisions

This catalog shows the available behaviors provided by Aerostack. They are presented according to different categories:

- Basic motion behaviors:
  - TAKE_OFF
  - LAND
  - WAIT
- Behaviors using motion controllers:
  - KEEP_HOVERING_WITH_PID_CONTROL
  - KEEP_MOVING_WITH_PID_CONTROL
  - ROTATE_WITH_PID_CONTROL
  - FOLLOW_PATH_WITH_PID_CONTROL
- Navigation behaviors:
  - GENERATE_PATH_WITH_OCCUPANCY_GRID
  - GO_TO_POINT_WITH_OCCUPANCY_GRID
- Behaviors for attention control:
  - PAY_ATTENTION_TO_QR_CODE
  - PAY_ATTENTION_TO_ROBOT_MESSAGES
- Behaviors for self localization:
  - SELF_LOCALIZE_BY_ODOMETRY
  - SELF_LOCALIZE_AND_MAP_WITH_LIDAR
- Behaviors for communication:
  - INFORM_OPERATOR
  - INFORM_ROBOTS
  - REQUEST_OPERATOR_ASSISTANCE

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

Pages 38

Getting started

- What is Aerostack
- Install Aerostack
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interfac
- Mission Plan Language
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

- Example Projects
- Program New Behavior
- Add New Hardware
- Setup Communications

International competitions

- 3rd place at MBZIRC 2(
- 2nd prize at IMAV 2017
- 4th place at IMAV 2016
- 1st prize at IMAV 2013

More information

- Publications
- Research Group CVAR

- **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_QR_CODE).

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

- Program new behaviors

Basic motion behaviors

Behavior `TAKE_OFF`

Description: The robot takes off vertically from a static surface. This behavior ends when the robot reaches a default altitude.

Type: Goal-based behavior.

Behavior `LAND`

Description: The robot lands vertically in the current position. This behavior assumes that the ground is static.

Type: Goal-based behavior.

Behavior `WAIT`

Description: The robot waits until a certain condition is satisfied (e.g., duration).

Type: Goal-based behavior.

Arguments	Format	Example
duration	number(seconds)	duration: 30

Behaviors using motion controllers

Clone this wiki locally

https://github.com/Visi

Behavior `KEEP_HOVERING_WITH_PID_CONTROL`

Description: 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_WITH_PID_CONTROL`

Description: 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_WITH_PID_CONTROL`

Description: The robot rotates left or right a certain number of degrees (angle) on the vertical axis (yaw). The number of degrees can be expressed as an absolute value (argument “angle”) or relative to the robot (argument “relative_angle”).

Type: Goal-based behavior.

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

Behavior `FOLLOW_PATH_WITH_PID_CONTROL`

Description: The robot tries to follow a path that is defined as a sequence of points. The robot assumes that there are not moving or unknown obstacles.

Type: Goal-based behavior.

Arguments	Format	Example
path:	sequence of 3D points (x,y,z) in meters	path: [[1.1,1.1,0.7], [2.2,2.2,0.7], [3.3,3.3,2.5]]

Navigation behaviors

Behavior `GENERATE_PATH_WITH_OCCUPANCY_GRID`

Description: This behavior generates a path free of obstacles between the current position of the robot and the given destination. This behavior uses a representation of the environment as an occupancy grid. The result of this task is stored in the belief memory in the predicates: object(X, path) and path(X, Y), where Y is a list of 3D points. For example: object(32, path), path(32, ((1.1,1.1,0.7), (2.2,2.2,0.7), (3.3,3.3,0.7))). Note that the height value of each point (z value) is a constant value because the path is planned on a horizontal surface.

Type: Goal-based behavior.

Arguments	Format	Example
destination	Tuple of 3 numbers x, y, z (meters)	destination: [3.3,3.3,3.3]

Note: This behavior is especially useful when used in companion with the Behavior SELF_LOCALIZE_AND_MAP_WITH_LIDAR

Behavior `GO_TO_POINT_WITH_OCCUPANCY_GRID`

Description: The robot goes to a destination point avoiding obstacles. The point can be expressed in absolute coordinates (argument "coordinates") or coordinates relative to the robot (argument "relative coordinates"). To reach the target point, the robot vertically changes the altitude to be at the same altitude as the target point and, then, follows a two-dimensional (2D) movement in the horizontal plane avoiding existing obstacles. During the movement, the robot can dynamically change the planned path if a new obstacle (e.g., another robot) is detected.

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 attention control

Behavior `PAY_ATTENTION_TO_QR_CODE`

Description: The robot pays attention to QR codes. Each recognized code is stored as a belief with the following format: object(X, qr_code), code(X, Y), visible(X) where X is the object identifier (generated automatically) and Y is the recognized QR code (string). The predicate visible(X) is deleted when the QR code is not observed. We assume that the objects with QR codes are static (they don't move in the environment). We assume that there are not different objects with the same QR code.

Type: Recurrent behavior.

Behavior `PAY_ATTENTION_TO_ROBOT_MESSAGES`

Description: This behavior pays attention to messages sent by other robots. When the name of the message's receiver is the name of the own robot, the message is added to the belief memory. Each message must be written as a logic predicate according to the format of the belief memory.

Type: Recurrent behavior

Behaviors for self localization

Behavior `SELF_LOCALIZE_BY_ODOMETRY`

Description: The robot self localizes using odometry. To obtain good trajectories with visual odometry it is necessary to fly in a controlled environment (e.g., good illumination without reflections and a non visually uniform ground).

Type: Recurrent behavior.

Behavior `SELF_LOCALIZE_AND_MAP_WITH_LIDAR`

Description: The robot self localizes and maps the environment using lidar. The resulting map is represented with an occupancy grid.

Type: Recurrent behavior.

Behaviors for communication

Behavior `INFORM_OPERATOR`

Description: This behavior presents through a pop-up window a text message that the robot sends to the operator.

Type: Goal-based behavior.

Name	Format	Example
message	String with the message to show	message: "Sensors detect a low level of light intensity in the environment where the aerial robot is flying."

Behavior `INFORM_ROBOTS`

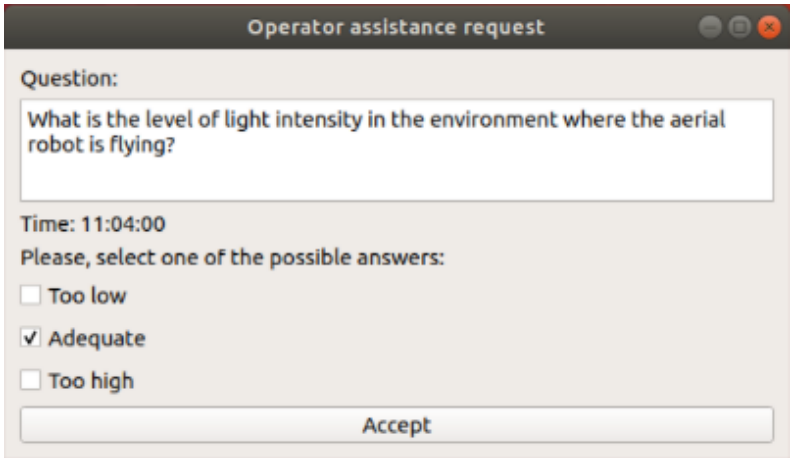
Description: This behavior sends a text message to another robot. The other robot must have activated the behavior PAY_ATTENTION_TO_ROBOT_MESSAGES. The text message must be a logic predicate according to the format of the belief memory.

Type: Goal-based behavior.

Arguments	Format	Example
source	String with the name of the sender	source: "drone1"
destination	String with the name of the receiver	destination: "drone2"
text	String with the text of the message as a logic predicate	text: "found(item07)"

Behavior `REQUEST_OPERATOR_ASSISTANCE`

Description: This behavior shows a pop-up window with a question for the operator and several options to be selected as the answer. The robot will save the selected answer in its belief memory.



Type: Goal-based behavior.

Name	Format	Example
question	String with the question for the operator	question: "What is the level of light intensity in the environment where the aerial robot is flying?"
belief_predicate_name	String with the name of the predicate to be saved in the belief memory	belief_predicate_name: "light_intensity"
options	Array of strings with the possible answers to the question	options: ["Too low","Adequate","Too high"]

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.upm@gmail.com

The content of the Aerostack wiki is licensed under Creative Commons license CC BY 4.0

Memory of Beliefs

Martin Molina edited this page on 12 Feb 2018 · 5 revisions

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). These predicates are always present in the memory with their corresponding values:

Predicate	Description
flight_state(self, x)	The flight state of the aerial robot is x (e.g., LANDED or FLYING)
position(self, x)	The aerial robot is at the spatial position x (in 3D coordinates). For example, x = (4.9, 5.1, 0).
battery_level(self,x)	The level of the battery charge is x (e.g., HIGH or LOW)

Pages 38

Getting started

- What is Aerostack
- Install Aerostack
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interfac
- Mission Plan Language
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

- Example Projects
- Program New Behavior
- Add New Hardware
- Setup Communications

International competitions

- 3rd place at MBZIRC 2(
- 2nd prize at IMAV 2017
- 4th place at IMAV 2016
- 1st prize at IMAV 2013

More information

- Publications
- Research Group CVAR

Clone this wiki locally

https://github.com/Visi

The examples presented in the next table are for illustration purposes (i.e., not all of them they are not implemented in the current version of Aerostack):

Predicate	Description
name(x, y)	The name of object x is y
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
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))
```

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.

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.upm@gmail.com

The content of the Aerostack wiki is licensed under Creative Commons license CC BY 4.0

Example projects

Martin Molina edited this page on 30 Jul 2019 · 13 revisions

Here you can find a list of example projects that show how to use Aerostack for different missions. To install a particular project, please follow the instructions described in the following page using the name of the desired project:

- Install Aerostack

Project name	Aerial platform	Link
teleoperation_drone_simulator	Drone-simulator	Teleoperation with "Drone simulator"
basic_mission_drone_simulator	Drone-simulator	Basic mission with "Drone simulator"
basic_behavior_tree_drone_simulator	Drone-simulator	Behavior tree mission with "Drone simulator"
teleoperation_rotors_simulator	Rotors simulator	Teleoperation with Rotors simulator
basic_mission_rotors_simulator	Rotors simulator	Basic mission with Rotors simulator
basic_lidar_rotors_simulator	Rotors simulator	Basic use of lidar with Rotors

Pages 38

Getting started

- What is Aerostack
- Install Aerostack
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interfac
- Mission Plan Language
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

- Example Projects
- Program New Behavior
- Add New Hardware
- Setup Communications

International competitions

- 3rd place at MBZIRC 2(
- 2nd prize at IMAV 2017
- 4th place at IMAV 2016
- 1st prize at IMAV 2013

More information

- Publications
- Research Group CVAR

		simulator
warehouse_rotors_simulator	Rotors simulator	Warehouse inventory with Rotors simulator
bridge_inspection_rotors_simulator	Rotors simulator	Bridge inspection with Rotors simulator

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.upm@gmail.com

The content of the Aerostack wiki is licensed under Creative Commons license CC BY 4.0

Clone this wiki locally

```
https://github.com/Visi
```

Teleoperation with "Drone simulator"

Martin Molina edited this page on 25 Jul 2019 · 1 revision

Teleoperation with Drone-simulator

Drone-simulator is a program that simulates the dynamics of a quadrotor. With this project, you can launch Aerostack with Drone-simulator and teleoperate the drone using the keyboard. After installing this project, perform the following steps:

1. Execute the ROS master node:

```
$ roscore
```

2. Open another terminal and change your current directory to the folder that contains the launcher of this project:

```
$ cd
$AEROSTACK_STACK/projects/teleoperation_drone_simulator
```

3. Execute the script that launches the Aerostack components for this project:

```
$ ./main_launcher.sh
```

An ArViz window will pop up showing a 3D image with a drone.

Pages 38

Getting started

- What is Aerostack
- Install Aerostack
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interfac
- Mission Plan Language
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

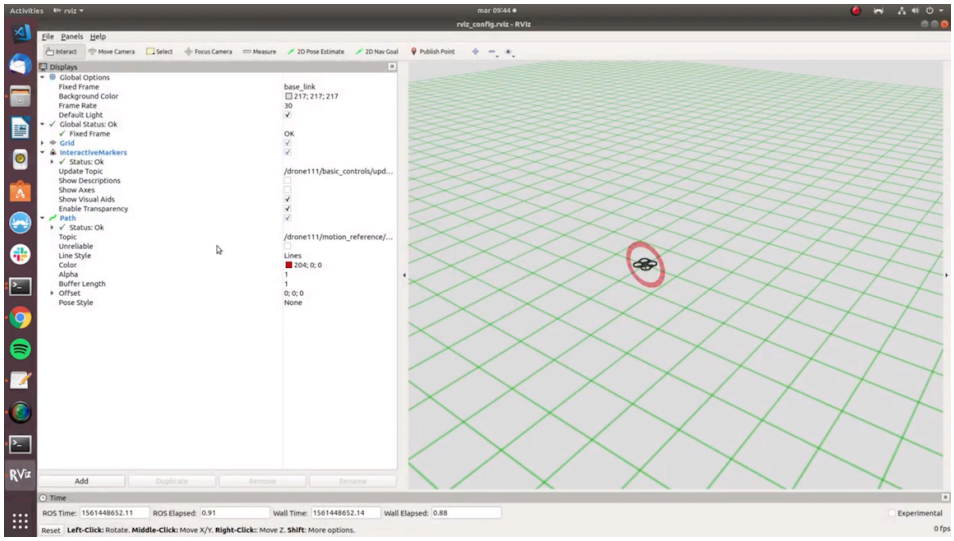
- Example Projects
- Program New Behavior
- Add New Hardware
- Setup Communications

International competitions

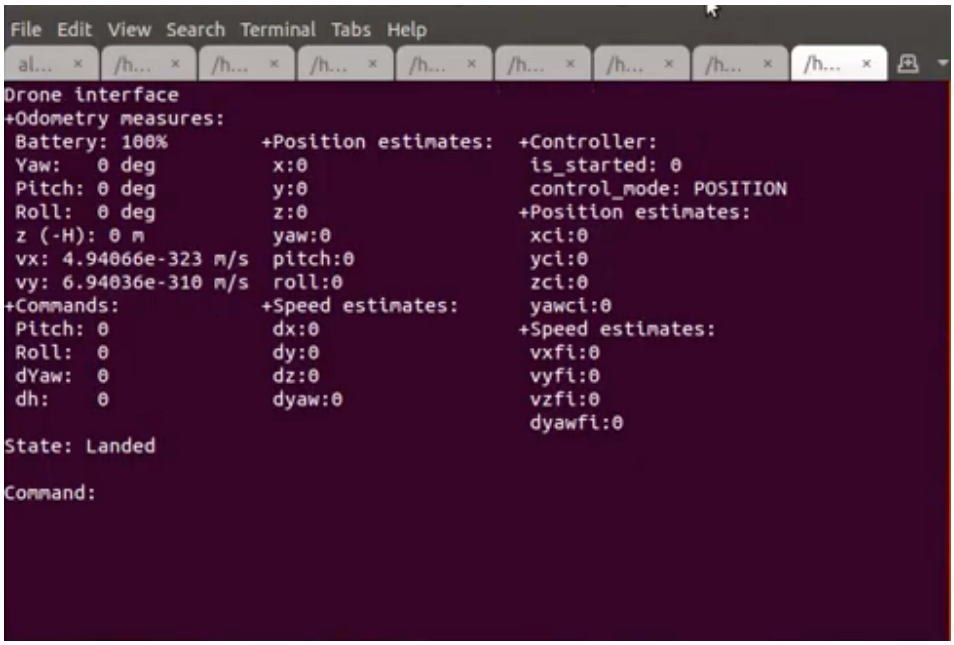
- 3rd place at MBZIRC 2(
- 2nd prize at IMAV 2017
- 4th place at IMAV 2016
- 1st prize at IMAV 2013

More information

- Publications
- Research Group CVAR



A set of terminals with different tabs is presented. Select the tab corresponding to the drone interface:



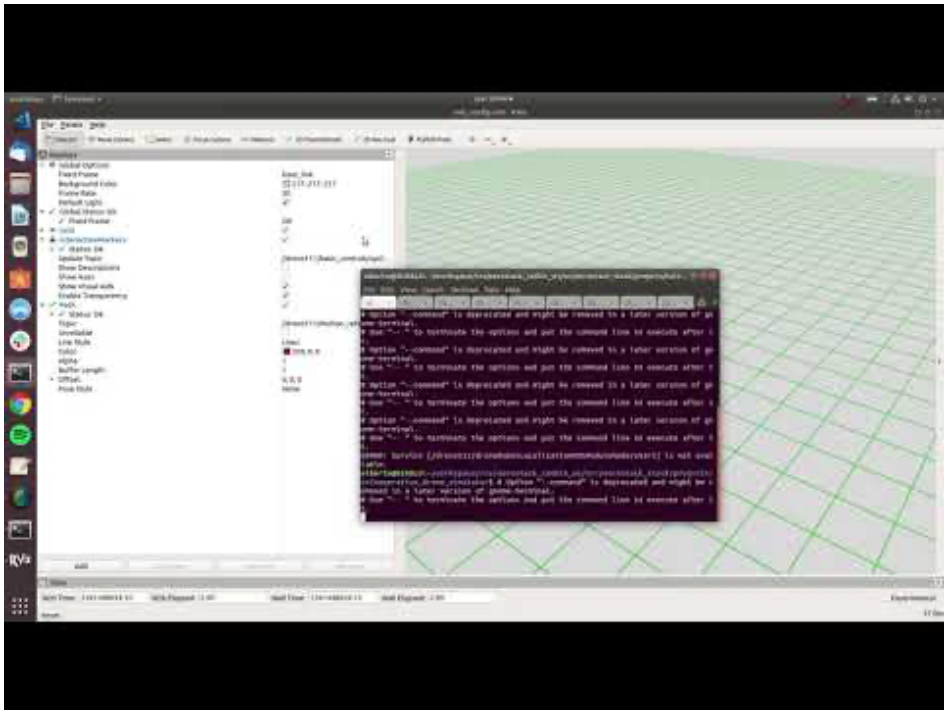
Aerostack now is ready to operate the simulated drone. Teleoperate the drone with the alphanumeric interface using the appropriate keys. Here you can see how to use the keyboard:

- Alphanumeric drone interface

Here there is a video that shows the execution of this project:

Clone this wiki locally

```
https://github.com/Visi
```



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.upm@gmail.com

The content of the Aerostack wiki is licensed under Creative Commons license CC BY 4.0

Basic mission with "Drone simulator"

Martin Molina edited this page on 25 Jul 2019 · 1 revision

With this project, you can launch Aerostack with Drone-simulator and execute a mission that is written in Python. The mission performs simple actions (e.g., take off, follow a path, land, etc.) by executing the corresponding behaviors. After installing this project, perform the following steps to use this project:

1. Execute the ROS master node:

```
$ roscore
```

2. Open a terminal and change your current directory to the folder that contains the launcher of this project:

```
$ cd
$AEROSTACK_STACK/projects/basic_mission_drone_simulator
```

3. You can see the tasks of the mission plan in the Python file. For example, write the following command to see the content of the file with the mission plan:

```
$ more configs/mission/mission.py
```

4. Execute the script that launches the Aerostack components for this project:

```
$ ./main_launcher.sh
```

An ArViz window will pop up showing a 3D image with a drone.

Pages 38

Getting started

- What is Aerostack
- Install Aerostack
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interfac
- Mission Plan Language
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

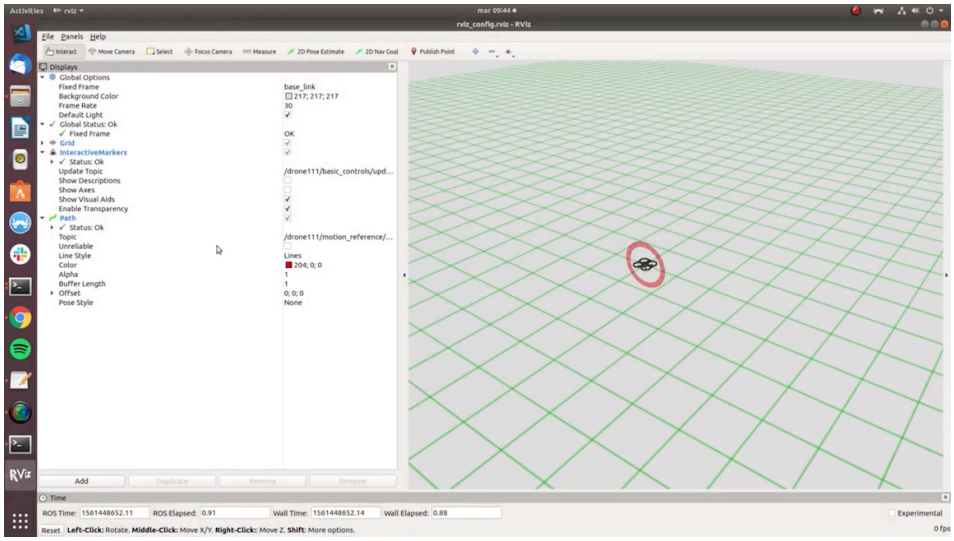
- Example Projects
- Program New Behavior
- Add New Hardware
- Setup Communications

International competitions

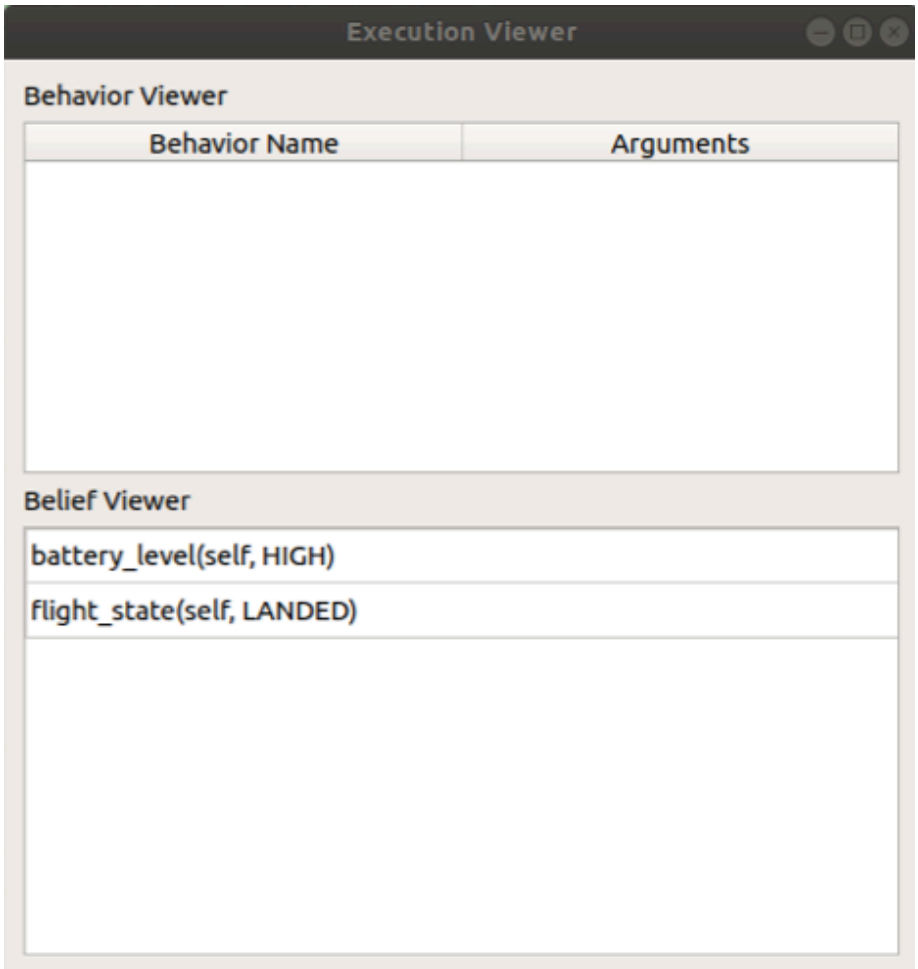
- 3rd place at MBZIRC 2(
- 2nd prize at IMAV 2017
- 4th place at IMAV 2016
- 1st prize at IMAV 2013

More information

- Publications
- Research Group CVAR



The window of the execution viewer is presented. This helps to monitor the behaviors that are executed during the mission execution:



5. Write the following command to execute the mission:

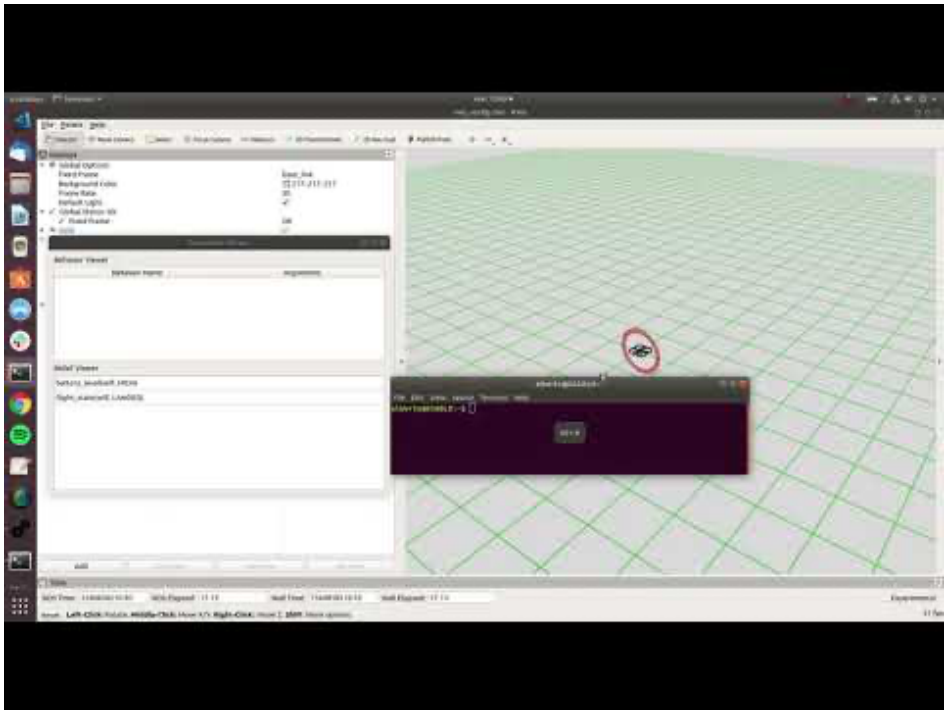
```
 $ rosservice call
/drone111/python_based_mission_interpreter_process/start
```

Clone this wiki locally

```
https://github.com/Visi
```

After this, you should see the drone moving in the simulated environment, performing the mission specified in the mission file. You can change the Python file to add, modify or remove tasks of the mission plan, and execute again the mission with these changes.

Here there is a video that shows the execution of the 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.upm@gmail.com

The content of the Aerostack wiki is licensed under Creative Commons license CC BY 4.0

Behavior tree mission with "Drone simulator"

Martin Molina edited this page on 30 Jul 2019 · 2 revisions

With this project, you can launch Aerostack with Drone-simulator and execute a mission using a behavior tree. The mission performs simple actions (e.g., take off, follow a path, land, etc.) by executing the corresponding behaviors. After installing this project, perform the following steps to use this project:

1. Execute the ROS master node:

```
$ roscore
```

2. Open a terminal and change your current directory to the folder that contains the launcher of this project:

```
 $ cd
$AEROSTACK_STACK/projects/basic_behavior_tree_drone_simu
```

3. Execute the script that launches the Aerostack components for this project:

```
$ ./main_launcher.sh
```

An ArViz window will pop up showing a 3D image with a drone.

Pages 38

Getting started

- What is Aerostack
- Install Aerostack
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interfac
- Mission Plan Language
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

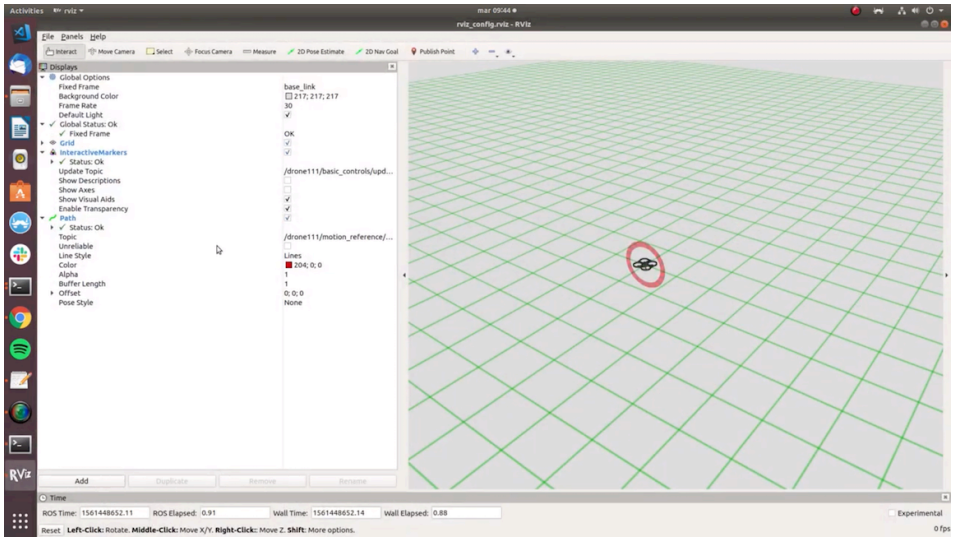
- Example Projects
- Program New Behavior
- Add New Hardware
- Setup Communications

International competitions

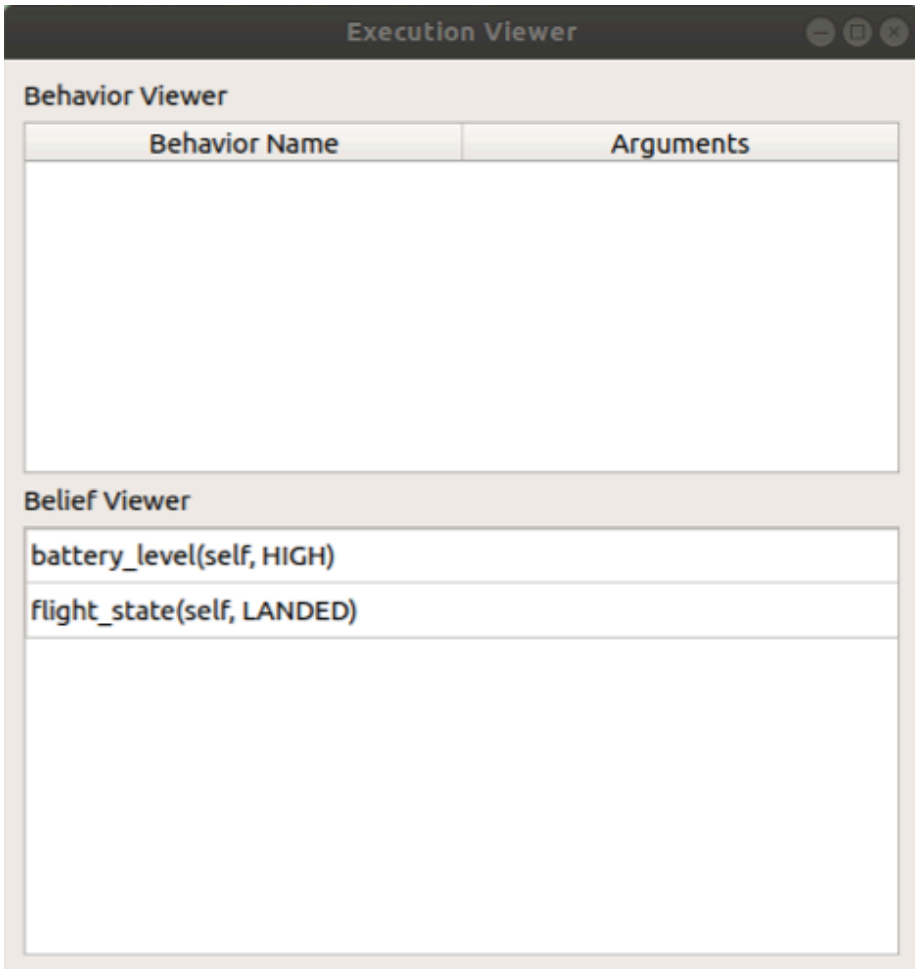
- 3rd place at MBZIRC 2(
- 2nd prize at IMAV 2017
- 4th place at IMAV 2016
- 1st prize at IMAV 2013

More information

- Publications
- Research Group CVAR



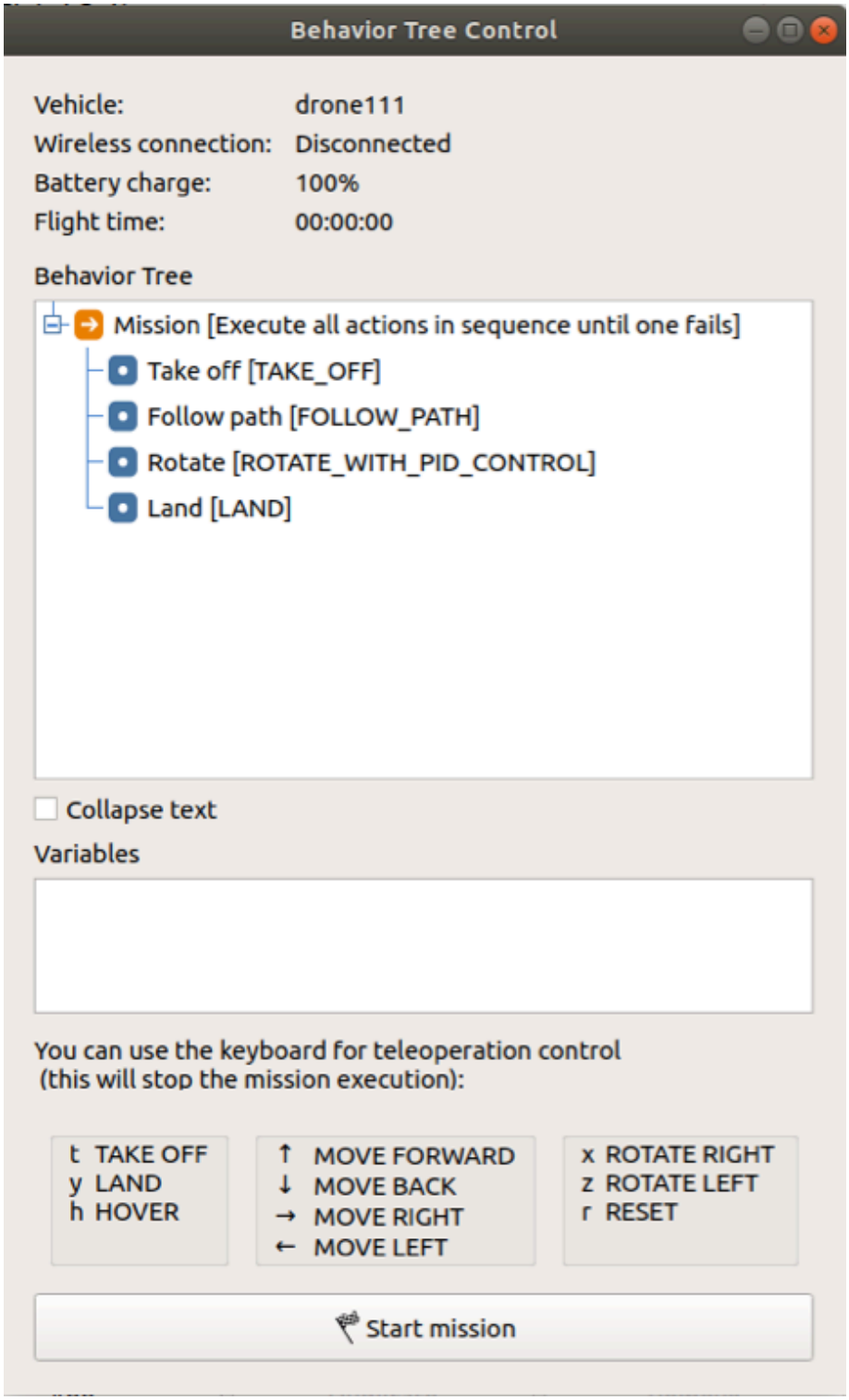
The window of the execution viewer is presented. This helps to monitor the behaviors that are executed during the mission execution:



The window of behavior tree viewer is also presented, showing the mission specification:

Clone this wiki locally

```
https://github.com/Visi
```



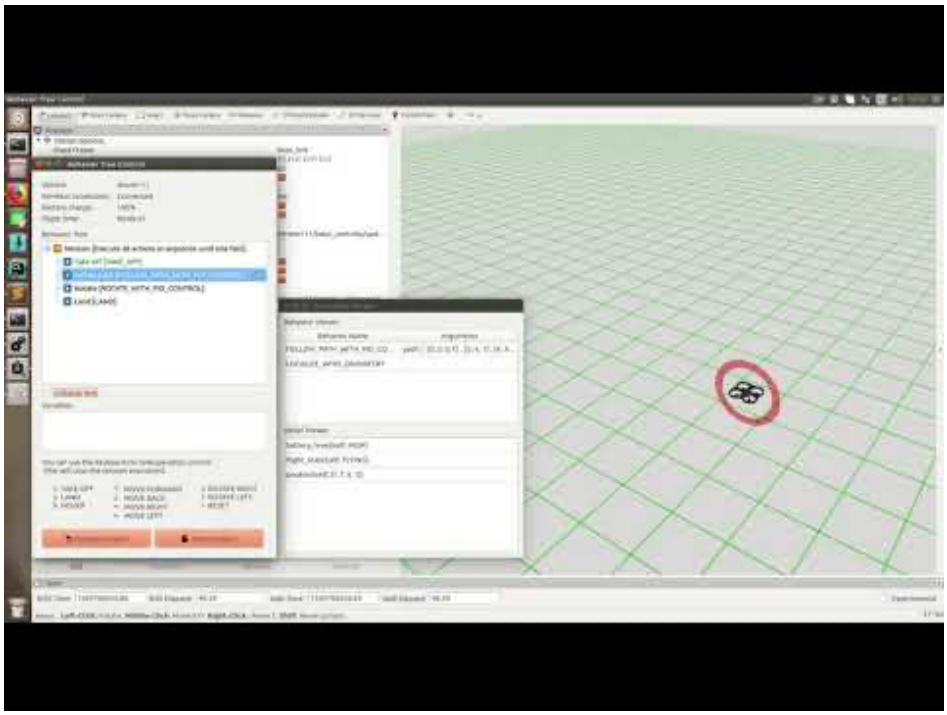
4. Press the `Start mission` button to start executing the mission.

After this, you should see the drone moving in the simulated environment, performing the mission specified in the behavior tree.

You can change behavior tree by executing the behavior tree editor with the following script:

```
$ ./mission_editor_launcher.sh
```

Here there is a video that shows the execution of the 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.upm@gmail.com

The content of the Aerostack wiki is licensed under Creative Commons license CC BY 4.0

Teleoperation with Rotors simulator

Martin Molina edited this page on 25 Jul 2019 · 1 revision

Rotors is a computer program (based on the simulator Gazebo) that simulates the dynamics of a quadrotor. With this project, you can launch Aerostack with Rotors and teleoperate the drone using the keyboard. After installing this project, perform the following steps:

1. Execute the ROS master node:

```
$ roscore
```

2. Open another terminal and change your current directory to the folder that contains the launcher of this project:

```
$ cd
$AEROSTACK_STACK/projects/teleoperation_rotors_simulatic
```

3. Execute the script that launches Gazebo for this project:

```
$ ./launcher_gazebo.sh
```

A Gazebo window is presented with a simulated drone.

Pages 38

Getting started

- What is Aerostack
- Install Aerostack
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interfac
- Mission Plan Language
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

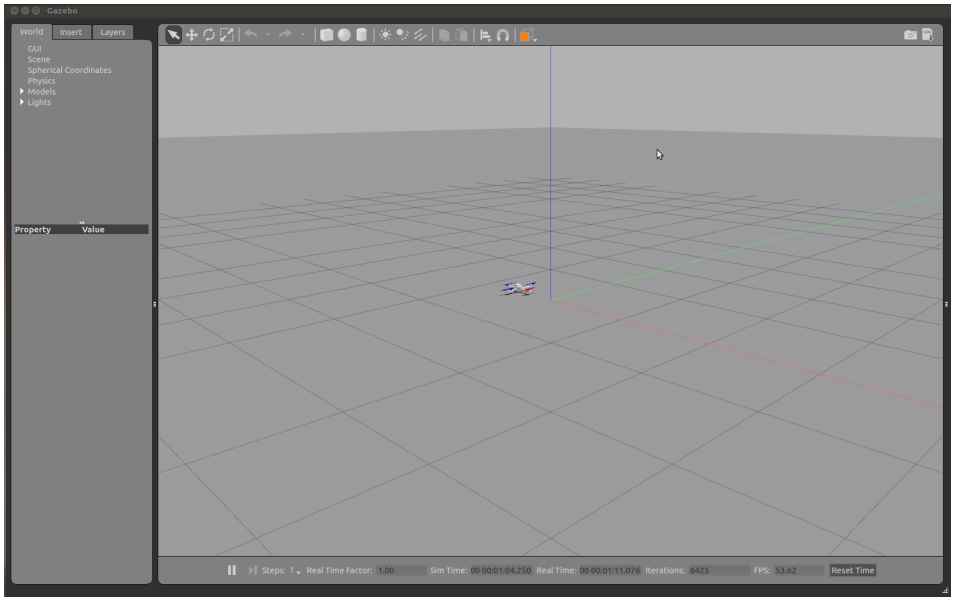
- Example Projects
- Program New Behavior
- Add New Hardware
- Setup Communications

International competitions

- 3rd place at MBZIRC 2(
- 2nd prize at IMAV 2017
- 4th place at IMAV 2016
- 1st prize at IMAV 2013

More information

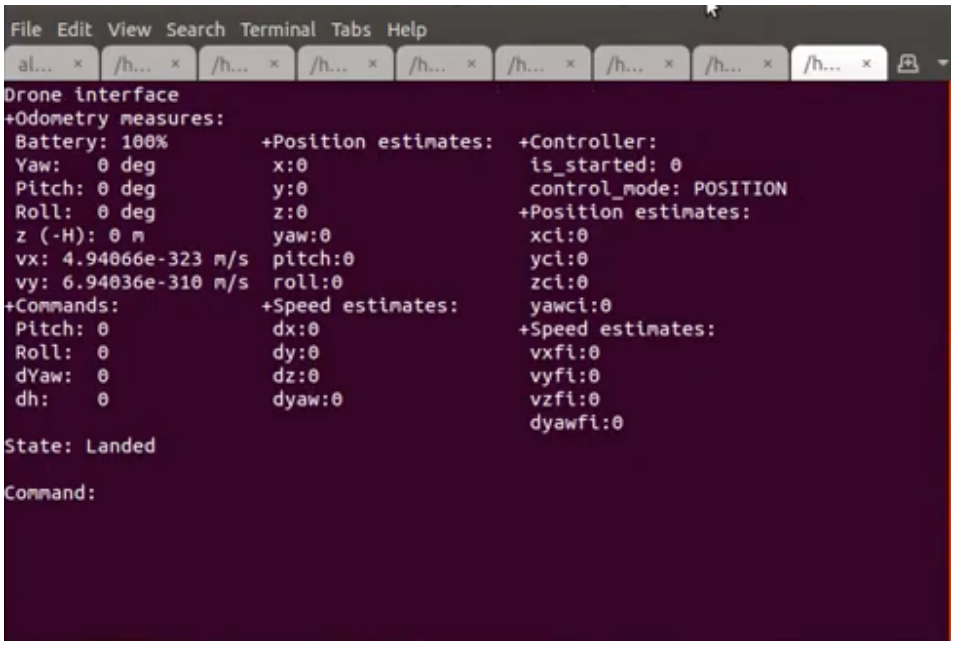
- Publications
- Research Group CVAR



4. Execute the script that launches the Aerostack components for this project:

```
$ ./main_launcher.sh
```

A set of terminals with different tabs is presented. Select the tab corresponding to the drone interface:



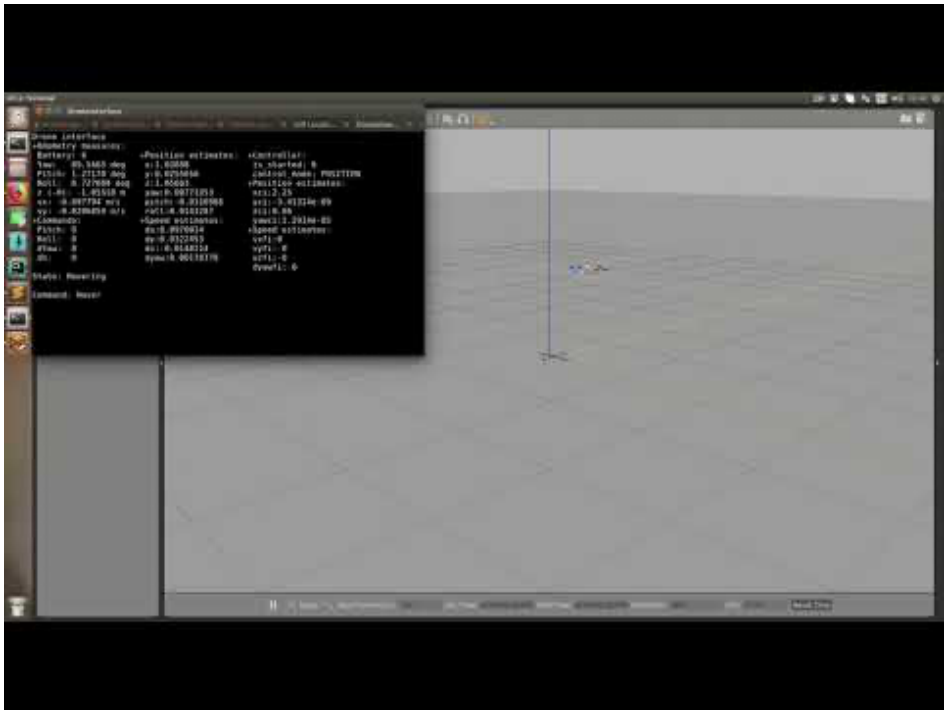
Aerostack now is ready to operate the simulated drone. Teleoperate the drone with the alphanumeric interface using the appropriate keys. Here you can see how to use the keyboard:

- Alphanumeric drone interface

Clone this wiki locally

```
https://github.com/Visi
```

Here there is a video that shows the execution of this project:



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.upm@gmail.com

The content of the Aerostack wiki is licensed under Creative Commons license CC BY 4.0

Basic mission with Rotors simulator

Martin Molina edited this page on 25 Jul 2019 · 1 revision

Rotors is a computer program (based on the simulator Gazebo) that simulates the dynamics of a quadrotor. With this project, you can launch Aerostack with Rotors and execute a mission written in Python language. After installing this project, perform the following steps:

1. Execute the ROS master node:

```
$ roscore
```

2. Open a terminal and change your current directory to the folder that contains the launcher of this project:

```
$ cd
$AEROSTACK_STACK/projects/basic_mission_rotors_simulator
```

3. Execute the script that launches Gazebo for this project:

```
$ ./launcher_gazebo.sh
```

A Gazebo window is presented with a simulated drone.

Pages 38

Getting started

- What is Aerostack
- Install Aerostack
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interfac
- Mission Plan Language
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

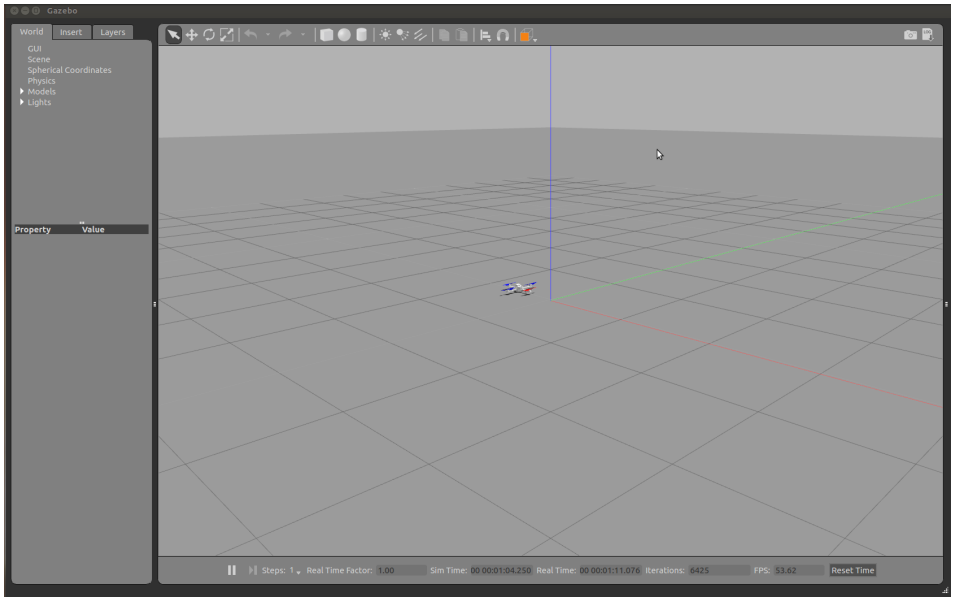
- Example Projects
- Program New Behavior
- Add New Hardware
- Setup Communications

International competitions

- 3rd place at MBZIRC 2(
- 2nd prize at IMAV 2017
- 4th place at IMAV 2016
- 1st prize at IMAV 2013

More information

- Publications
- Research Group CVAR



4. You can see the tasks of the mission plan in the Python file. For example, write the following command to see the content of the file with the mission plan:

```
$ more configs/mission/mission.py
```

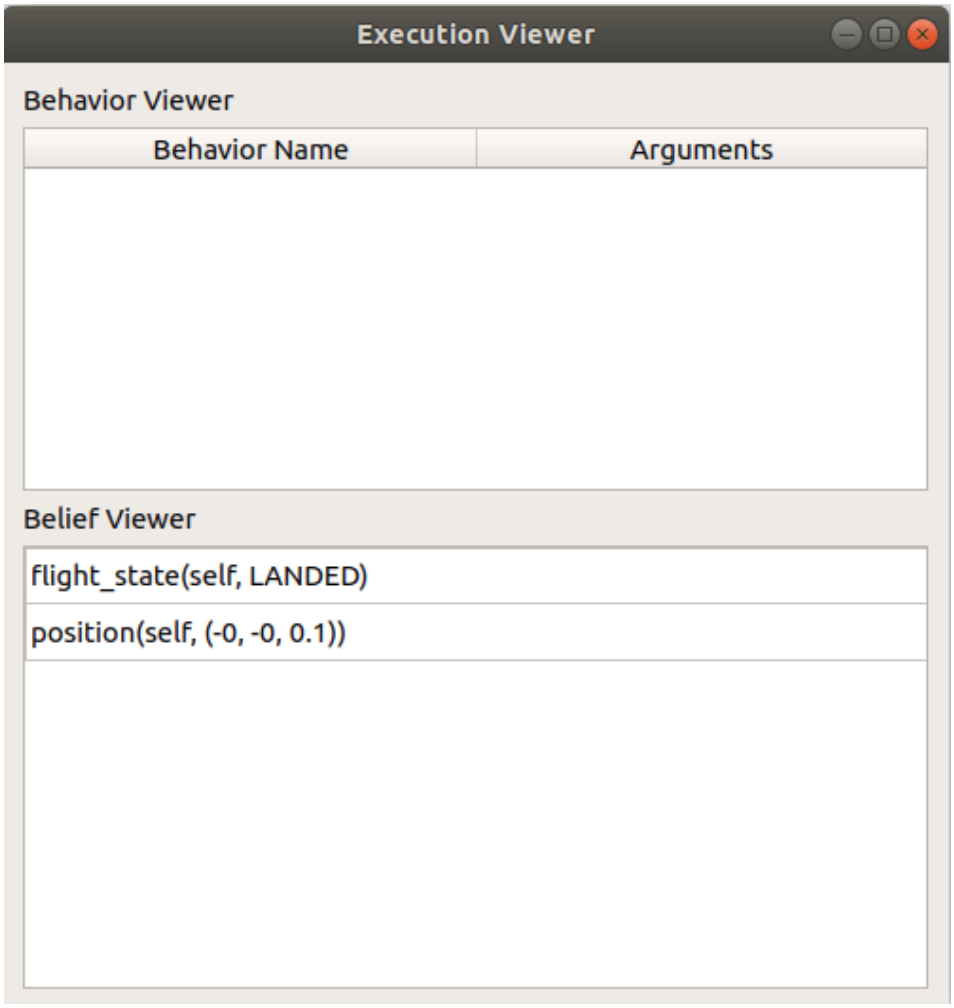
5. Execute the script that launches the Aerostack components for this project:

```
$ ./main_launcher.sh
```

The window of the execution viewer is presented. This helps to monitor the behaviors that are executed during the mission execution:

Clone this wiki locally

```
https://github.com/Visi
```

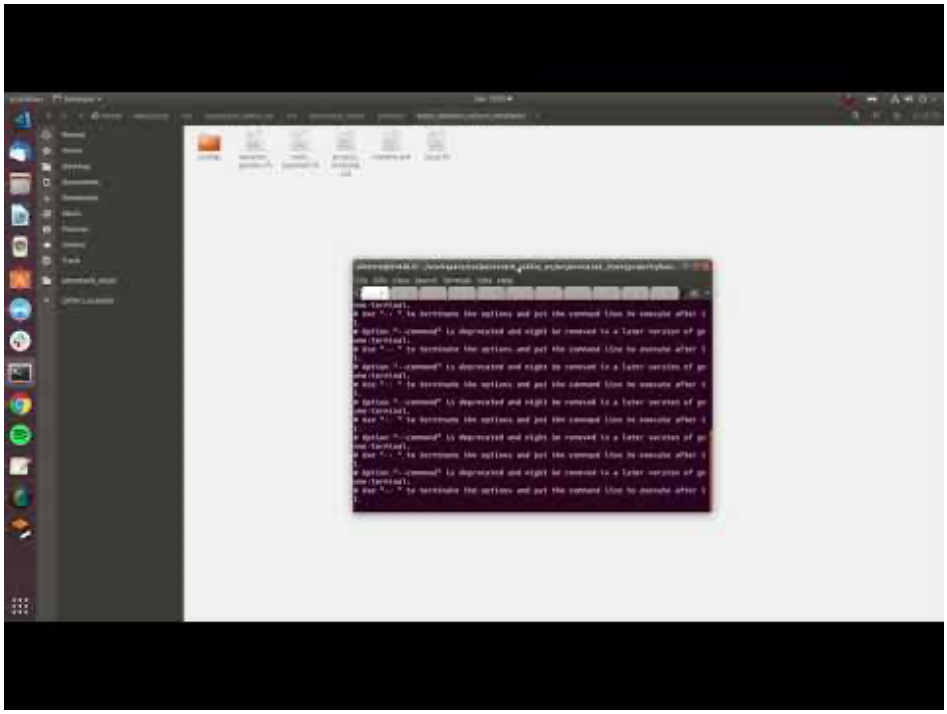


6. Write the following command to execute the mission:

```
 $ rosservice call
/drone111/python_based_mission_interpreter_process/start
```

After this, you should see the drone moving in the simulated environment, performing the mission specified in the mission file. You can change the Python file to add, modify or remove tasks of the mission plan, and execute again the mission with these changes.

Here there is a video that shows the execution of the 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.upm@gmail.com

The content of the Aerostack wiki is licensed under Creative Commons license CC BY 4.0

Basic use of lidar with Rotors simulator

Martin Molina edited this page on 25 Jul 2019 · 1 revision

With this project, you can launch Aerostack with Rotors simulator and execute a basic mission (written in Python) that uses a lidar sensor. The mission performs simple movements in a scenario with some obstacles.

> IMPORTANT: This mission has only been tested in Ubuntu 16.04 with ROS Kinetic.

After installing this project, perform the following steps to use this project:

1. Open a terminal and change your current directory to the folder that contains the launcher of this project:

```
$ cd $AEROSTACK_STACK/projects/basic_lidar_rotors_simulator
```

2. This project uses "move_base" and "hector_slam" provided by the ROS library. To install "move_base" and "hector_slam", write the following commands (in this example we are using the ROS distribution ros-melodic, but you can use another distribution instead):

```
$ sudo apt-get install ros-melodic-move-base*

$ sudo apt-get install ros-melodic-hector*

$ sudo apt-get install ros-melodic-amcl*
```

3. Install Rotors components of this project to be used by Rotors simulator:

Pages 38

Getting started

- What is Aerostack
- Install Aerostack
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interfac
- Mission Plan Language
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

- Example Projects
- Program New Behavior
- Add New Hardware
- Setup Communications

International competitions

- 3rd place at MBZIRC 2(
- 2nd prize at IMAV 2017
- 4th place at IMAV 2016
- 1st prize at IMAV 2013

More information

- Publications
- Research Group CVAR

```
$ ./lidar_instalation.sh
```

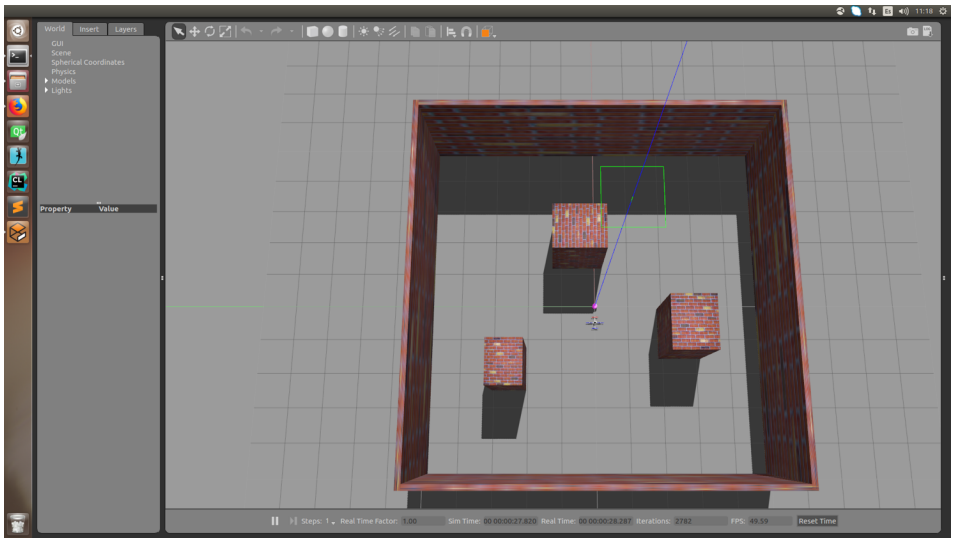
4. Execute the ROS master node:

```
$ roscore
```

5. Execute the script that launches Gazebo for this project:

```
$ ./launcher_gazebo.sh
```

A Gazebo window is presented with a simulated drone.



6. You can see the tasks of the mission plan in the Python file. For example, write the following command to see the content of the file with the mission plan:

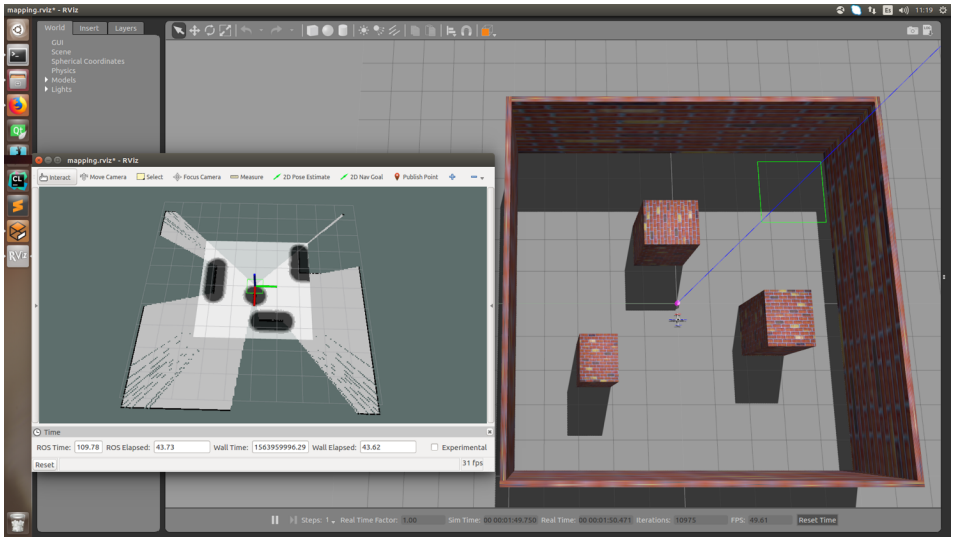
```
$ more configs/mission/mission.py
```

7. Execute the script that launches the Aerostack components for this project:

```
$ ./main_launcher.sh
```

Clone this wiki locally

```
https://github.com/Visi
```



8. Write the following command to execute the mission:

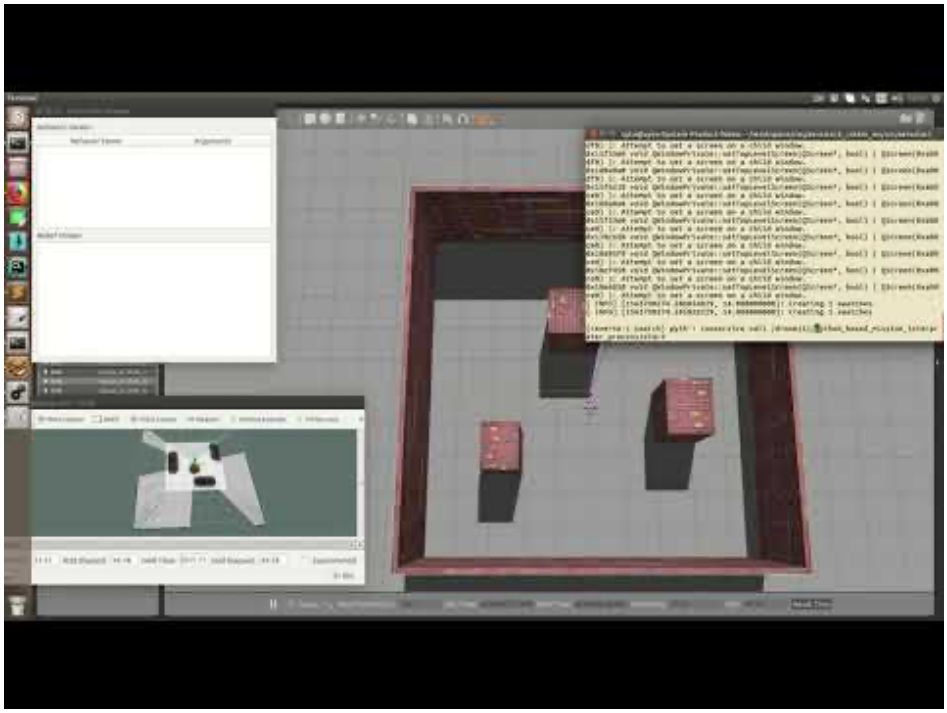
```
 $ rosservice call
/drone111/python_based_mission_interpreter_process/start
```

9. Write the following command to execute the mission:

```
$ rosservice call
/drone111/python_based_mission_interpreter_process/start
```

After this, you should see the drone moving in the simulated environment, performing the mission specified in the mission file. You can change the Python file to add, modify or remove tasks of the mission plan, and execute again the mission with these changes.

Here there is a video that shows the execution of the 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.upm@gmail.com

The content of the Aerostack wiki is licensed under Creative Commons license CC BY 4.0

Warehouse inventory with Rotors simulator

Martin Molina edited this page on 25 Jul 2019 · 1 revision

With this project, you can launch Aerostack with Rotors simulator and execute a mission to perform the inventory of a warehouse. The mission is written in Python and the drone uses a lidar sensor and a QR recognizer.

> IMPORTANT: This mission has only been tested in Ubuntu 16.04 with ROS Kinetic.

After installing this project, perform the following steps to use this project:

1. Open a terminal and change your current directory to the folder that contains the launcher of this project:

```
$ cd $AEROSTACK_STACK/projects/warehouse_rotors_simulation
```

2. This project uses "move_base" and "hector_slam" provided by the ROS library. To install "move_base" and "hector_slam", write the following commands (in this example we are using the ROS distribution ros-melodic, but you can use another distribution instead):

```
$ sudo apt-get install ros-melodic-move-base*

$ sudo apt-get install ros-melodic-hector*

$ sudo apt-get install ros-melodic-amcl*
```

Pages 38

Getting started

- What is Aerostack
- Install Aerostack
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interfac
- Mission Plan Language
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

- Example Projects
- Program New Behavior
- Add New Hardware
- Setup Communications

International competitions

- 3rd place at MBZIRC 2(
- 2nd prize at IMAV 2017
- 4th place at IMAV 2016
- 1st prize at IMAV 2013

More information

- Publications
- Research Group CVAR

3. Install Rotors components of this project to be used by Rotors simulator:

```
$ ./lidar_instalation.sh
```

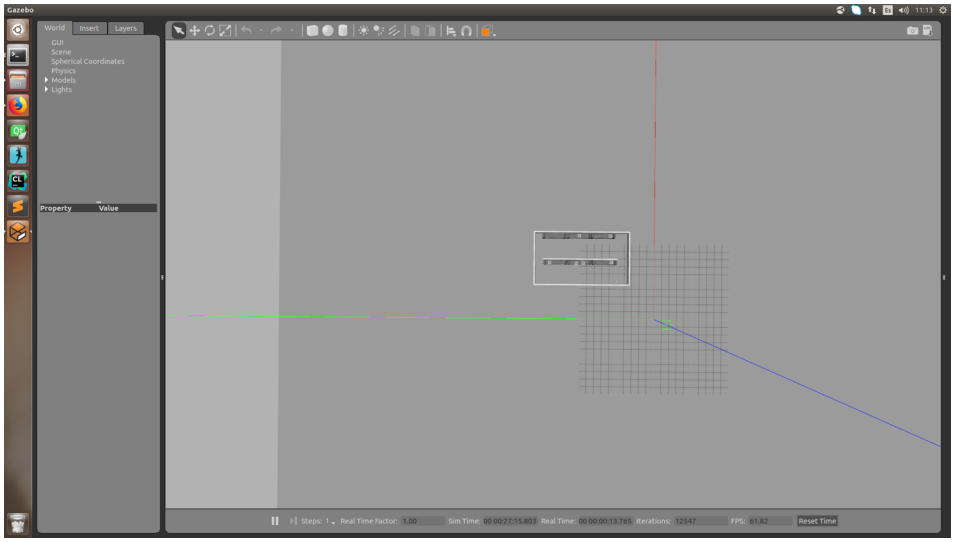
4. Execute the ROS master node:

```
$ roscore
```

5. Execute the script that launches Gazebo for this project:

```
$ ./launcher_gazebo.sh
```

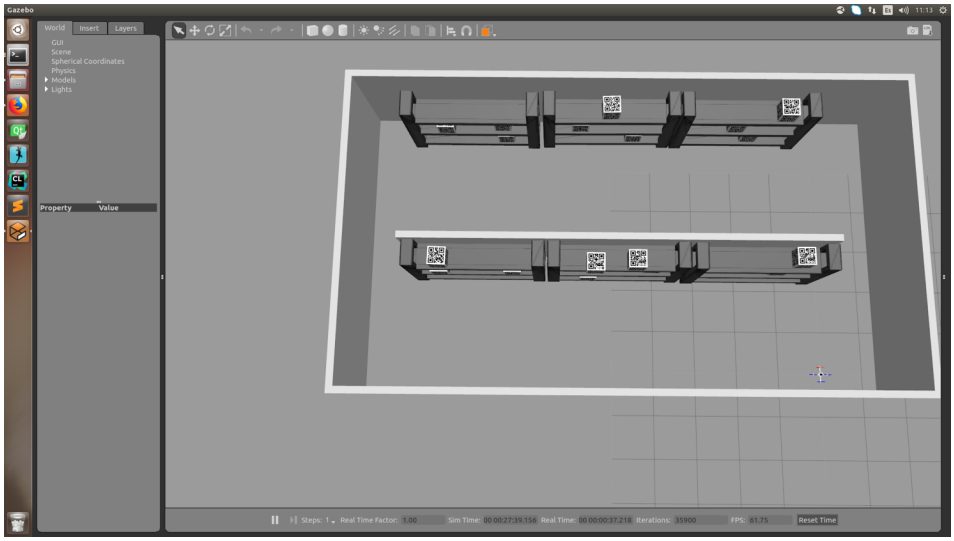
A Gazebo window is presented with a simulated drone.

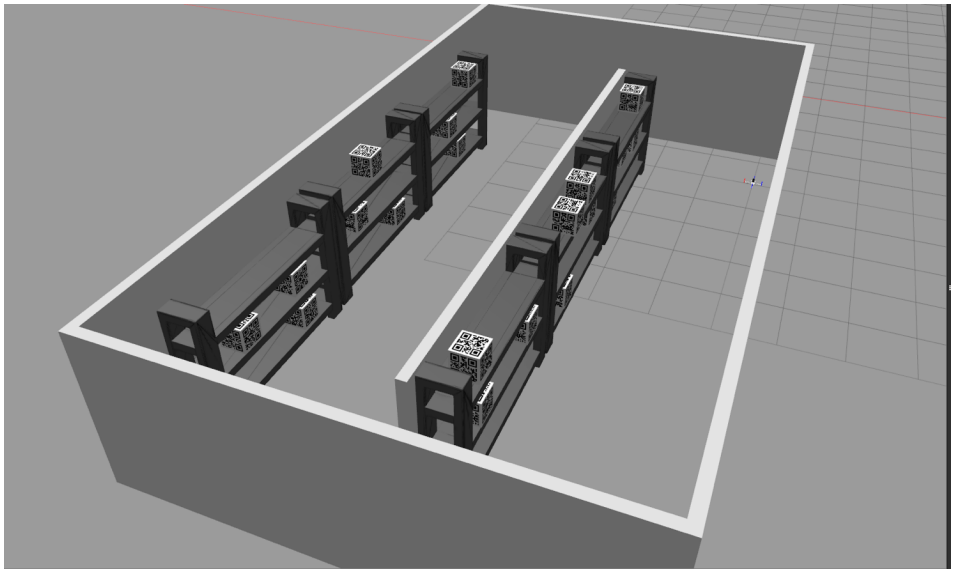


6. You can zoom and rotate the image of the warehouse using the mouse.

Clone this wiki locally

```
https://github.com/Visi
```





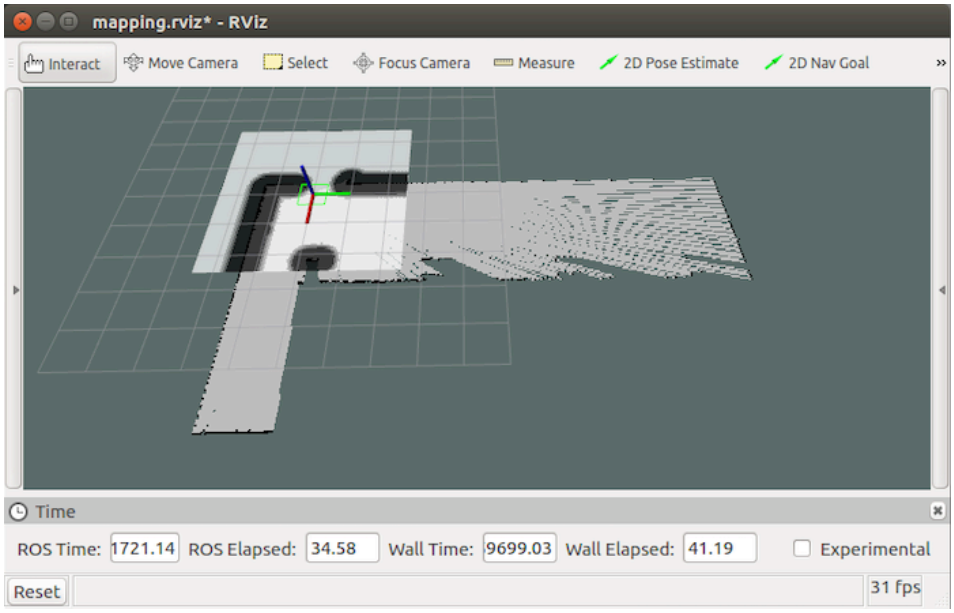
7. You can see the tasks of the mission plan in the Python file. For example, write the following command to see the content of the file with the mission plan:

```
$ more configs/mission/mission.py
```

8. Execute the script that launches the Aerostack components for this project:

```
$ ./main_launcher.sh
```

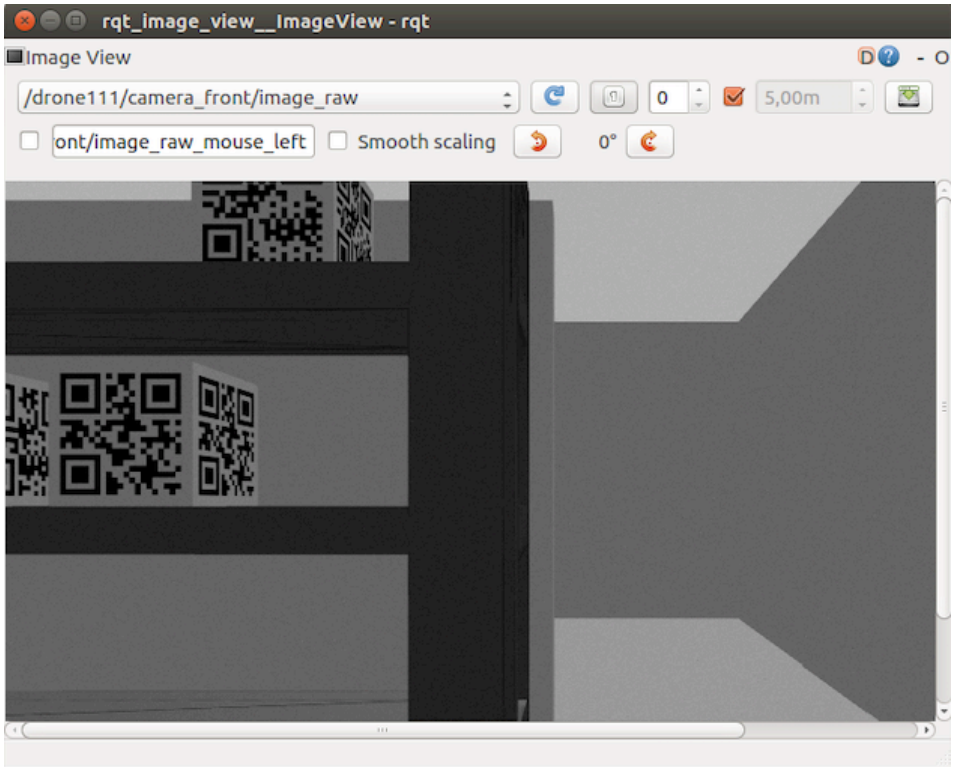
The window showing the map is presented:



9. Write the following command to show the camera image:

```
$ rqt_image_view
```

A window showing the camera image:



10. Write the following command to execute the mission:

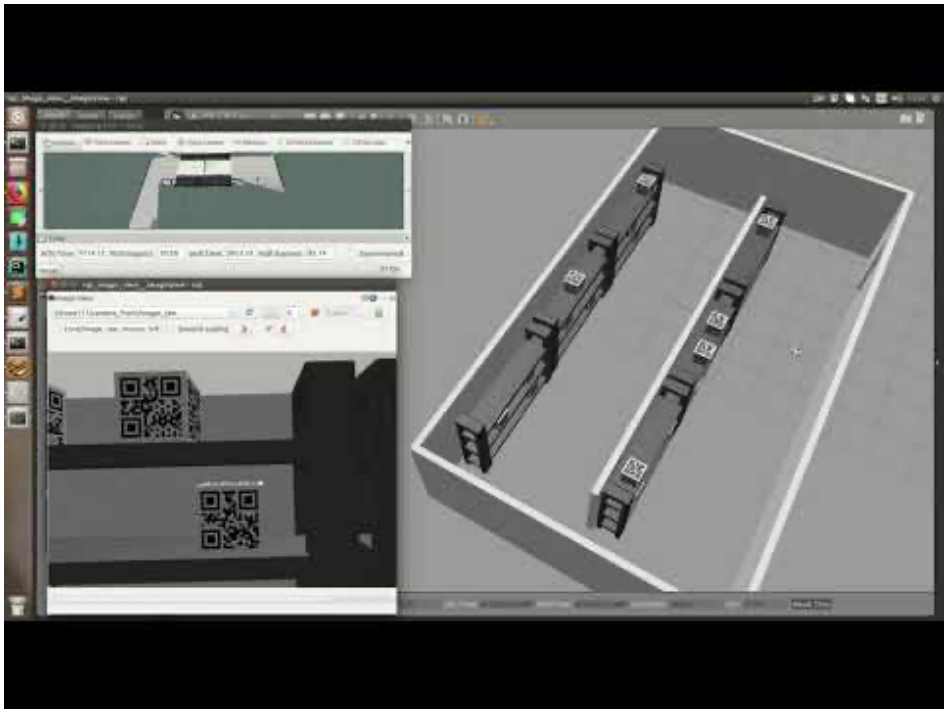
```
$ rosservice call
/drone111/python_based_mission_interpreter_process/start
```

11. Write the following command to execute the mission:

```
$ rosservice call
/drone111/python_based_mission_interpreter_process/start
```

After this, you should see the drone moving in the simulated environment, performing the mission specified in the mission file. You can change the Python file to add, modify or remove tasks of the mission plan, and execute again the mission with these changes.

Here there is a video that shows the correct execution of the test:



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.upm@gmail.com

The content of the Aerostack wiki is licensed under Creative Commons license CC BY 4.0

Bridge inspection with Rotors simulator

Martin Molina edited this page on 31 Jul 2019 · 4 revisions

With this project, you can launch Aerostack with Rotors simulator and execute a mission to perform the inspection of a bridge. The mission is specified as a behavior tree.

After installing this project, perform the following steps to use this project:

1. Open a terminal and change your current directory to the folder that contains the launcher of this project:

```
$ cd
$AEROSTACK_STACK/projects/bridge_inspection_rotors_simul
```

2. Execute the ROS master node:

```
$ roscore
```

3. Execute the script that launches Gazebo for this project:

```
$ ./launcher_gazebo.sh
```

A Gazebo window is presented with a simulated drone.

Pages 38

Getting started

- What is Aerostack
- Install Aerostack
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interfac
- Mission Plan Language
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

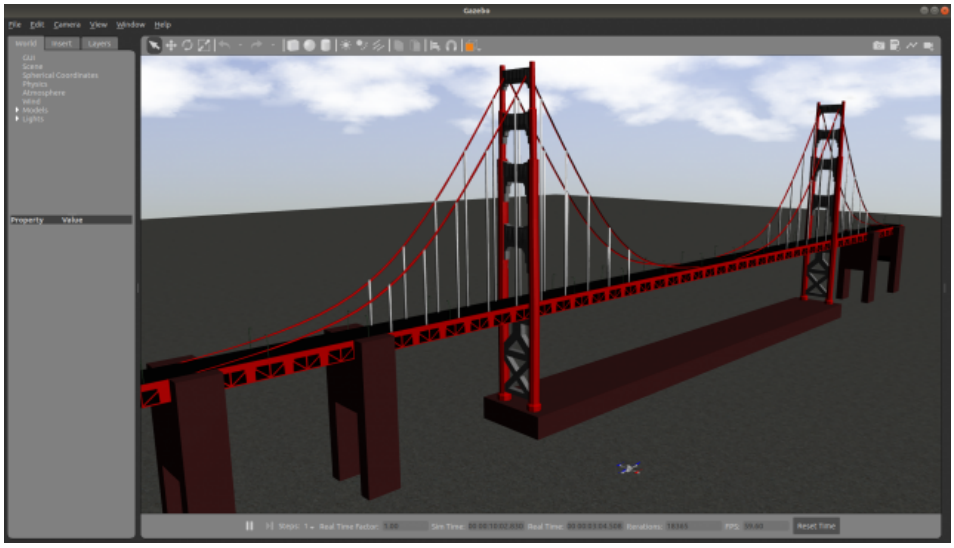
- Example Projects
- Program New Behavior
- Add New Hardware
- Setup Communications

International competitions

- 3rd place at MBZIRC 2(
- 2nd prize at IMAV 2017
- 4th place at IMAV 2016
- 1st prize at IMAV 2013

More information

- Publications
- Research Group CVAR



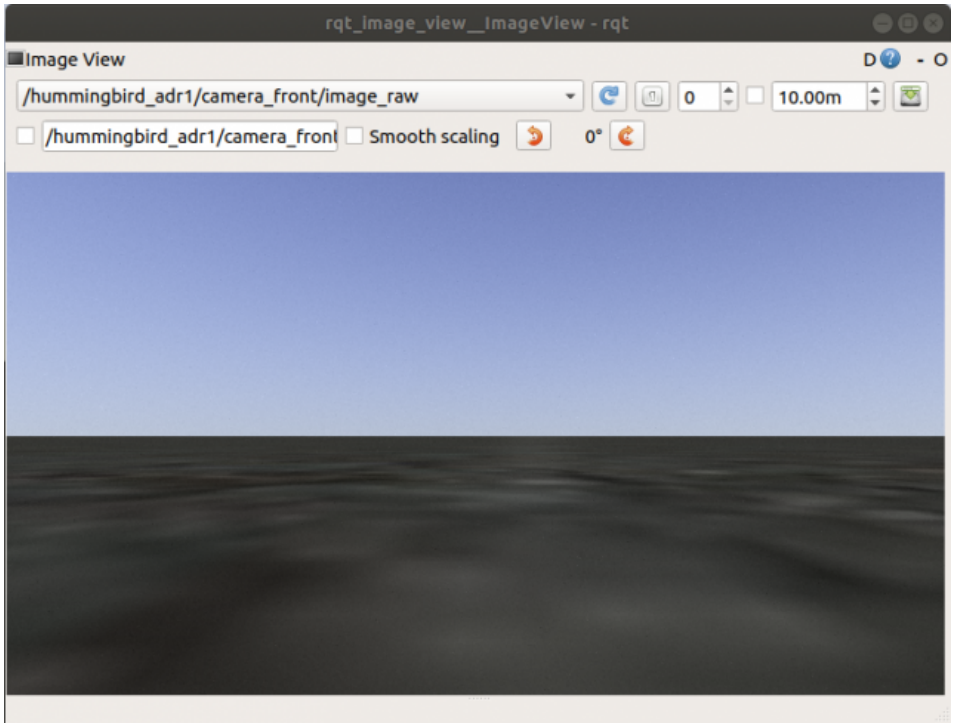
Clone this wiki locally

```
https://github.com/Visi
```

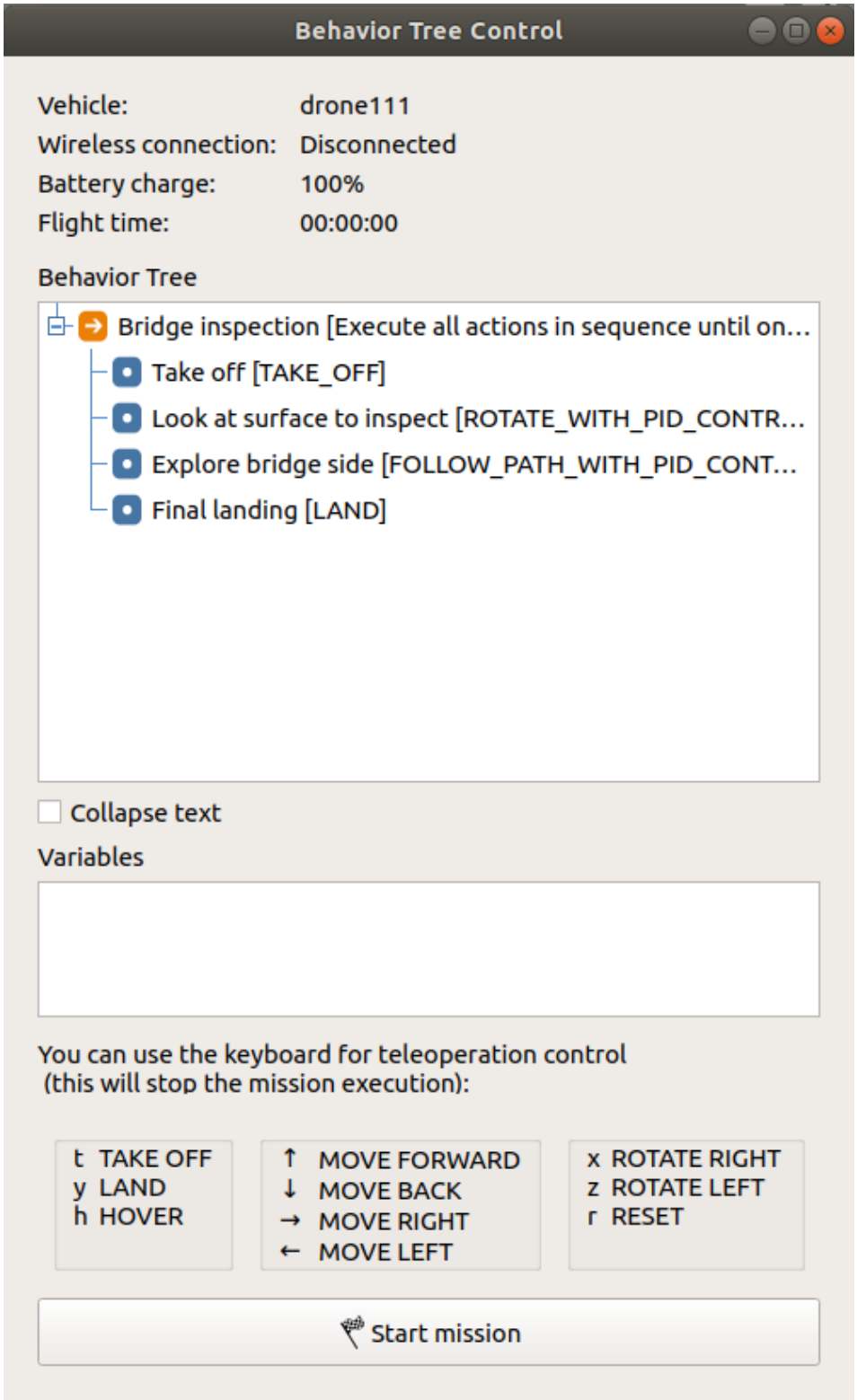
4. Execute the script that launches the Aerostack components for this project:

```
$ ./main_launcher.sh
```

A window showing the camera view is presented:



A window showing the behavior tree is presented:



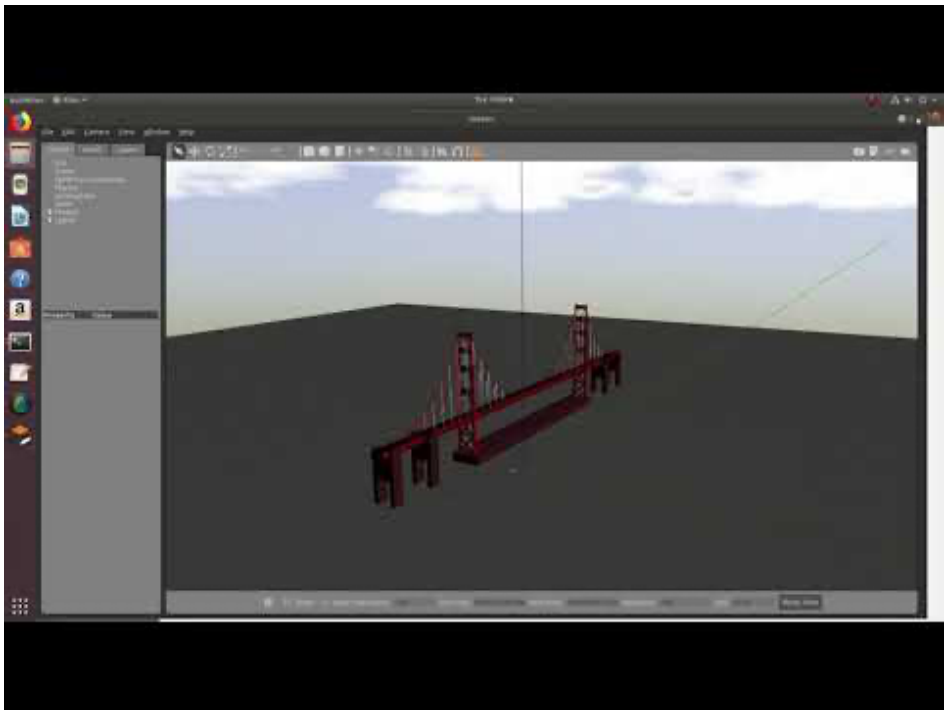
5. Click on the button `Start mission` to initiate the mission execution.

After this, you should see the drone moving in the simulated environment, performing the mission specified in the behavior tree.

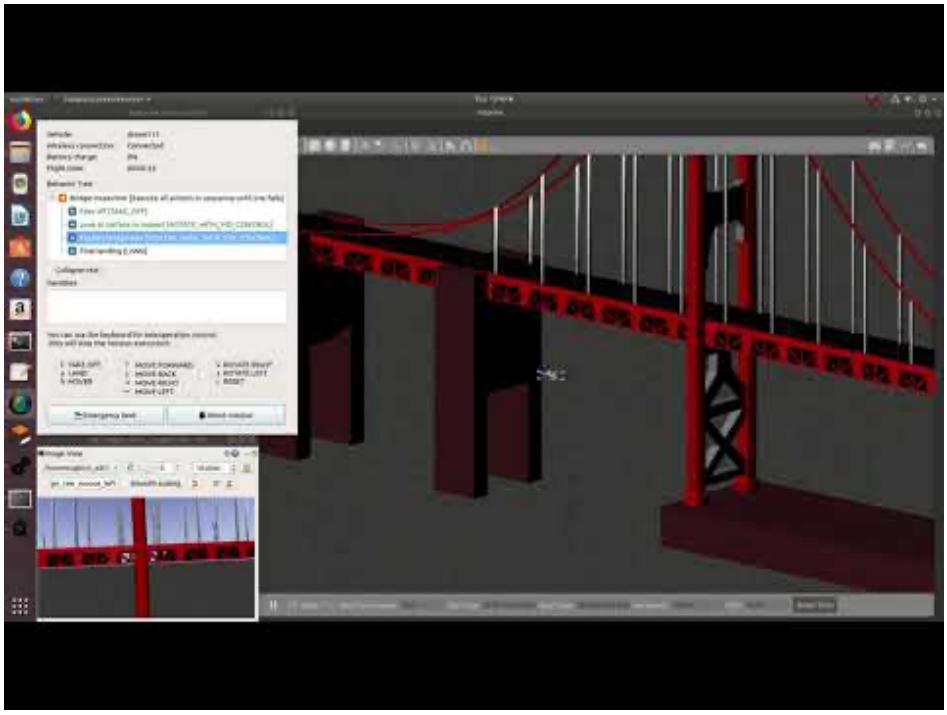
You can change behavior tree by executing the behavior tree editor with the following script:

```
$ ./mission_editor_launcher.sh
```

Here there is a video that shows how to launch the project:



Here there is a video that shows the mission execution:



Credits

The original model of the bridge (format .obj) was created by Vishnu Srinivas (nikname mishukutti) and it is available at Thingiverse. The model was modified for this Aerostack project and adapted to be used with the Gazebo simulator.

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.upm@gmail.com

The content of the Aerostack wiki is licensed under Creative Commons license CC BY 4.0

Program New Behaviors

Abraham Carrera edited this page on 26 Jul 2019 · 2 revisions

Aerostack includes a behavior-based execution system that simplifies the configuration of robot mission plans. Each behavior is implemented as a nodelet inside a behavior system that groups behaviors with common uses. Those behaviors are also a subclass of BehaviorExecutionController class.

Steps to create a new behavior

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

The name of the package should be the name of the behavior system. As an example, the system for the behavior `TAKE_OFF`, and therefore the package, is called `basic_quadrotor_behaviors`.

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

```
basic_cuadrotor_behaviors/              -- Name of the
package
      CMakeLists.txt                    -- CMakeLists.txt
file
      package.xml                       -- Package
manifest
      basic_quadrotor_behaviors.xml     -- File to define
the nodelets as a plugin
      launch/
        basic_quadrotor_behaviors.launch   -- Launcher
      src/
        include/
          behavior_take_off.h           -- Header file
        source/
          behavior_take_off.cpp         -- C++ file
```

Pages 38

Getting started

- What is Aerostack
- Install Aerostack
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interfac
- Mission Plan Language
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

- Example Projects
- Program New Behavior
- Add New Hardware
- Setup Communications

International competitions

- 3rd place at MBZIRC 2(
- 2nd prize at IMAV 2017
- 4th place at IMAV 2016
- 1st prize at IMAV 2013

More information

- Publications
- Research Group CVAR

Clone this wiki locally

https://github.com/Visi

Step 2. Edit the files CMakeLists and package.xml

In order to develop a new behavior, the class `BehaviorExecutionController` must be inherited, as well as the `Nodelet` class . 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_execution_controller
nodelet
aerostack_msgs
geometry_msgs
std_msgs
)

catkin_package(
INCLUDE_DIRS ${BEHAVIOR_TAKE_OFF_INCLUDE_DIR}
CATKIN_DEPENDS
roscpp
behavior_execution_controller
nodelet
aerostack_msgs
geometry_msgs
std_msgs
)
```

File *package.xml*

```
<build_depend>roscpp</build_depend>
<build_depend>behavior_execution_controller</build_depend>
<build_depend>nodelet</build_depend>
<build_depend>aerostack_msgs</build_depend>
<build_depend>geometry_msgs</build_depend>
<build_depend>std_msgs</build_depend>

<run_depend>roscpp</run_depend>
<run_depend>behavior_execution_controller</run_depend>
<run_depend>nodelet</run_depend>
<run_depend>aerostack_msgs</run_depend>
<run_depend>geometry_msgs</run_depend>
<run_depend>std_msgs</run_depend>
```

It is also important to include the tag to specify the xml file used to export the nodelets:

```
<export>

<nodelet  plugin="${prefix}/basic_quadrotor_behaviors.xml"

</export>
```

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

Step 3. Edit the <package_name>.launch file

As you can see, there is a directory called `launch` which contains a file with the definition of topics names or other parameters. It is important to include the following line in each behavior system to launch its nodelet manager:

```
<node  pkg="nodelet"  type="nodelet"  args="manager"  name="
```

And also include a line like the following one for each behavior in the behavior system:

```
<node  pkg="nodelet"  type="nodelet"  args="load basic_quad
```

You can find an example of the `basic_quadrotor_behaviors.launch` file at the end of the tutorial.

Step 4. Edit the <package_name>.xml file

Now we have to edit the <package_name>.xml file to correctly define the nodelet as a plugin. To do so, in the first line indicate the library to which the plugin will be exported:

```
<library  path="lib/libbasic_quadrotor_behaviors">
```

Then include a tag with the following format for each behavior included in the system :

```
<class  name="basic_quadrotor_behaviors/BehaviorTakeOff"  ty
base_class_type="nodelet::Nodelet">
    <description>
      BehaviorTakeOff.
    </description>
</class>
```

Step 5. Create the header and implementation files

To use the class BehaviorExecutionController the header file needs to have the following structure, using as namespace the name of the behavior system:

```
#include <behavior_execution_controller.h>
namespace  basic_quadrotor_behaviors
{
class  BehaviorTakeOff : public  BehaviorExecutionControlle
{
(...)
};
}
```

Then, the following functions should be implemented:

```
private:
// BehaviorExecutionController
void  onConfigure();
void  onActivate();
void  onDeactivate();
void  onExecute();
bool  checkSituation();
void  checkGoal();
void  checkProgress();
void  checkProcesses();
```

These functions give a behavior the ability to:

- Prepare the execution (onConfigure)
- Start the execution (onActivate)
- Stop the execution (onDeactivate)
- Develop the execution (onExecute)
- Verify if the behavior can be executed in that situation (checkSituation)

- Verify if the behavior has achieved its goal (checkGoal)
- Supervise the progress of the execution (checkProgress)
- Supervise the state of the necessary processes (checkProcesses)

The implementation file must include the following line to export the class to the nodelet manager:

```
PLUGINLIB_EXPORT_CLASS(basic_quadrotor_behaviors::BehaviorTa
```

Step 6. 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).
- The `default_values` tag indicates values to be used when no explicit specification for the parameter is found.
- There are two categories of behaviors: goal_based and recurrent.
- Incompatibilities represent inconsistencies among behaviors.
- Precedences represent behaviors that a requirement to carry out the activation.
- It is necessary to indicate the behavior system each behavior belongs to.
- It is possible to specify a timeout for the execution of the behavior.
- If a behavior is specified as default, it will continuously be intended to be activated with low priority

Here there is an example of behavior catalog:

```
#-----------------------------------------------------
# Default values for behavior attributes.
#-----------------------------------------------------
default_values:
  category: goal_based
```

```
  default: no
  timeout: 15
#-----------------------------------------------------
# LIST OF BEHAVIOR DESCRIPTORS
#-----------------------------------------------------
behavior_descriptors:
#-----------------------------------------------------
# LOCALIZE_WITH_ODOMETRY
#-----------------------------------------------------
- behavior: LOCALIZE_WITH_ODOMETRY
   category: recurrent
   default: yes
   system: basic_quadrotor_behaviors
#-----------------------------------------------------
# TAKE_OFF
#-----------------------------------------------------
- behavior: TAKE_OFF
    system: basic_quadrotor_behaviors
#-----------------------------------------------------
# LAND
#-----------------------------------------------------
- behavior: LAND
    system: basic_quadrotor_behaviors
#-----------------------------------------------------
# PRECEDENCE CONSTRAINTS
#-----------------------------------------------------
precedence_constraints:
- active:
  - LOCALIZE_WITH_ODOMETRY
  before:
  - TAKE_OFF

  - LAND
#-----------------------------------------------------
# EXCLUSIVITY CONSTRAINTS
#-----------------------------------------------------
exclusivity_constraints:
#-----------------------------------------------------
# Motion behaviors are mutually exclusive
#-----------------------------------------------------
- mutually_exclusive:
  - TAKE_OFF
  - LAND
```

Step 7. Launch the behavior within Aerostack

After the behavior is included in the catalog file, you just need to add the created behavior system inside the launcher script. In your launcher include (following the example with `basic_quadrotor_behaviors`) the next lines:

```
`#-----------------------------------------------------------
`# Basic Quadrotor Behaviors
`#-----------------------------------------------------------
--tab --title "Basic Quadrotor Behaviors" --command "bash -c
roslaunch basic_quadrotor_behaviors basic_quadrotor_behavio
  namespace:=drone$NUMID_DRONE \
  my_stack_directory:=${AEROSTACK_STACK};
exec bash\""\
```

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

```
rosservice call /drone1/request_behavior_activation "behavio
name: 'TAKE_OFF'
arguments: ''
priority: 2"
```

Example

As an example, we include all the files necessary for the execution of the behavior `TAKE_OFF`:

- behavior_take_off.h:

```
#ifndef TAKE_OFF_H
#define TAKE_OFF_H

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

// Aerostack msgs
```

```
#include <aerostack_msgs/BehaviorActivationFinished.h>
#include <droneMsgsROS/ConsultBelief.h>
#include <droneMsgsROS/droneCommand.h>
#include <droneMsgsROS/setInitDroneYaw_srv_type.h>
#include <geometry_msgs/PoseStamped.h>
// Aerostack libraries
#include <behavior_execution_controller.h>

namespace basic_quadrotor_behaviors
{
class BehaviorTakeOff : public BehaviorExecutionController
{
  // Constructor
public:
  BehaviorTakeOff();
  ~BehaviorTakeOff();

private:
  // ros::NodeHandle node_handle;

  // Congfig variables
  std::string estimated_pose_str;
  std::string controllers_str;
  std::string rotation_angles_str;
  std::string initialize_yaw_str;
  std::string execute_query_srv;

  // Communication variables
  ros::Subscriber estimated_pose_sub;
  ros::Subscriber rotation_angles_sub;
  ros::Publisher controllers_pub;
  ros::ServiceClient initialize_yaw_cli;
  ros::ServiceClient query_client;

  // Messages
  geometry_msgs::PoseStamped estimated_pose_msg;
  geometry_msgs::Vector3Stamped rotation_angles_msg;

  // Timer staticity_timer;
  bool first_position;

private:
  // BehaviorExecutionController
  void onConfigure();
  void onActivate();
  void onDeactivate();
  void onExecute();
  bool checkSituation();
  void checkGoal();
  void checkProgress();
```

```
  void checkProcesses();



public: // Callbacks
  void estimatedPoseCallBack(const geometry_msgs::PoseStampe
  void rotationAnglesCallback(const geometry_msgs::Vector3S
};
}

#endif
```

- behavior_take_off.cpp:

```
#include "../include/behavior_take_off.h"
#include <pluginlib/class_list_macros.h>

namespace basic_quadrotor_behaviors
{
BehaviorTakeOff::BehaviorTakeOff() : BehaviorExecutionContr
{
  setName("take_off");
}

BehaviorTakeOff::~BehaviorTakeOff() {}

void BehaviorTakeOff::onConfigure()
{
  ros::NodeHandle nh = getNodeHandle();
  nh.param<std::string>("estimated_pose_topic", estimated_po
  nh.param<std::string>("controllers_topic", controllers_st
  nh.param<std::string>("rotation_angles_topic", rotation_an
  nh.param<std::string>("initialize_yaw_srv", initialize_yaw
                                "droneOdometryStateEstimato
  nh.param<std::string>("consult_belief", execute_query_srv_
  query_client = nh.serviceClient<droneMsgsROS::ConsultBelie
}

void BehaviorTakeOff::onActivate()
{
  ros::NodeHandle nh = getNodeHandle();
  std::string nspace = getNamespace();
  // Activate communications
  estimated_pose_sub = nh.subscribe("/" + nspace + "/" + est
                                            &BehaviorTakeOf
  rotation_angles_sub = nh.subscribe("/" + nspace + "/" + ro
                                             &BehaviorTake
  initialize_yaw_cli =
```

```
      nh.serviceClient<droneMsgsROS::setInitDroneYaw_srv_typ
  controllers_pub = nh.advertise<droneMsgsROS::droneCommand>

  // 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.pose.position.z = 0;
  first_position = false;
}

void BehaviorTakeOff::onDeactivate()
{
  std_msgs::Header header;
  header.frame_id = "behavior_take_off";

  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();
}

void BehaviorTakeOff::onExecute()
{

}

bool BehaviorTakeOff::checkSituation()
{
  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)
  {
    setErrorMessage("Error: Battery low, unable to perform a
    return false;
  }
```

```
  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)
  {
    setErrorMessage("Error: Already flying");
    return false;
  }

  return true;
}

void BehaviorTakeOff::checkGoal()
{
  double precision_take_off = 0.1;
  // Check achievement
  if (std::abs(std::abs(estimated_pose_msg.pose.position.z)
  {
    BehaviorExecutionController::setTerminationCause(aerosta
    droneMsgsROS::setInitDroneYaw_srv_type init_yaw_msg;
    init_yaw_msg.request.yaw_droneLMrT_telemetry_rad = (rota
    initialize_yaw_cli.call(init_yaw_msg);

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

  }

}

void BehaviorTakeOff::checkProgress()
{

}


void BehaviorTakeOff::checkProcesses()
{

}

// Custom topic Callbacks
void BehaviorTakeOff::estimatedPoseCallBack(const geometry_m
{
  if (!first_position)

    estimated_pose_msg = message;
```

```
}

void BehaviorTakeOff::rotationAnglesCallback(const geometry_
{
  rotation_angles_msg = message;
}
}
PLUGINLIB_EXPORT_CLASS(basic_quadrotor_behaviors::BehaviorTa
```

- package.xml:

```
<?xml  version="1.0"?>
<package>
<name>basic_quadrotor_behaviors</name>
<version>0.1.0</version>
<description>Basic quadrotor system of behaviors</descriptio
<maintainer  email="abraham.carreragrob@alumnos.upm.es">Abra

<license>BSD-3</license>

<buildtool_depend>catkin</buildtool_depend>
<build_depend>roscpp</build_depend>
<build_depend>nodelet</build_depend>
<build_depend>std_msgs</build_depend>
<build_depend>drone_process</build_depend>
<build_depend>yaml-cpp</build_depend>
<build_depend>behavior_execution_controller</build_depend>
<build_depend>droneMsgsROS</build_depend>
<build_depend>geometry_msgs</build_depend>
<build_depend>aerostack_msgs</build_depend>

<run_depend>roscpp</run_depend>
<run_depend>nodelet</run_depend>
<run_depend>std_msgs</run_depend>
<run_depend>drone_process</run_depend>
<run_depend>behavior_execution_controller</run_depend>
<run_depend>droneMsgsROS</run_depend>
<run_depend>geometry_msgs</run_depend>
<run_depend>yaml-cpp</run_depend>
<run_depend>aerostack_msgs</run_depend>

<export>
<nodelet  plugin="${prefix}/basic_quadrotor_behaviors.xml"

</export>
</package>
```

- CMakeLists.txt:

```
cmake_minimum_required(VERSION 2.8.3)
project(basic_quadrotor_behaviors)

### 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(BASIC_QUADROTOR_BEHAVIORS_SOURCE_DIR
  src/source
)

set(BASIC_QUADROTOR_BEHAVIORS_INCLUDE_DIR
  src/include
)

set(BASIC_QUADROTOR_BEHAVIORS_HEADER_FILES

${BASIC_QUADROTOR_BEHAVIORS_INCLUDE_DIR}/behavior_take_off.h


${BASIC_QUADROTOR_BEHAVIORS_INCLUDE_DIR}/behavior_localize_w


${BASIC_QUADROTOR_BEHAVIORS_INCLUDE_DIR}/behavior_land.h

${BASIC_QUADROTOR_BEHAVIORS_INCLUDE_DIR}/behavior_wait.h
)

set(BASIC_QUADROTOR_BEHAVIORS_SOURCE_FILES

${BASIC_QUADROTOR_BEHAVIORS_SOURCE_DIR}/behavior_take_off.cp


${BASIC_QUADROTOR_BEHAVIORS_SOURCE_DIR}/behavior_localize_wi


${BASIC_QUADROTOR_BEHAVIORS_SOURCE_DIR}/behavior_land.cpp

${BASIC_QUADROTOR_BEHAVIORS_SOURCE_DIR}/behavior_wait.cpp
)
## 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
  nodelet
  std_msgs
  behavior_execution_controller
  droneMsgsROS
  geometry_msgs
  aerostack_msgs
)

###################################
## catkin specific configuration ##
###################################
## The catkin_package macro generates cmake config files
for your package
## Declare things to be passed to dependent projects
## 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 ${BASIC_QUADROTOR_BEHAVIORS_INCLUDE_DIR}
  LIBRARIES ${PROJECT_NAME}
  CATKIN_DEPENDS
  roscpp
  nodelet
  std_msgs
  drone_process
  behavior_execution_controller
  droneMsgsROS
  geometry_msgs
  aerostack_msgs
  DEPENDS yaml-cpp
)

###########
## Build ##
###########

## Specify additional locations of header files
## Your package locations should be listed before other
locations
# include_directories(include)
include_directories(
```

```
  ${catkin_INCLUDE_DIRS}
)

## Declare a C++ library
add_library(basic_quadrotor_behaviors
  src/source/behavior_take_off.cpp
  src/source/behavior_localize_with_odometry.cpp
  src/source/behavior_land.cpp
  src/source/behavior_wait.cpp
)
target_link_libraries(basic_quadrotor_behaviors
${catkin_LIBRARIES})
target_link_libraries(basic_quadrotor_behaviors yaml-cpp)
if(catkin_EXPORTED_LIBRARIES)
  add_dependencies(basic_quadrotor_behaviors
${catkin_EXPORTED_LIBRARIES})
endif()
```

- basic_quadrotor_behaviors.xml:

```
<library path="lib/libbasic_quadrotor_behaviors">
        <class name="basic_quadrotor_behaviors/BehaviorTake(
                base_class_type="nodelet::Nodelet">
                <description>
                        BehaviorTakeOff.
                </description>
        </class>
        <class name="basic_quadrotor_behaviors/BehaviorLoca
                base_class_type="nodelet::Nodelet">
                <description>
                        BehaviorLocalizeWithOdometry.
                </description>
        </class>
        <class name="basic_quadrotor_behaviors/BehaviorLand'
                base_class_type="nodelet::Nodelet">
                <description>
                        BehaviorLand.
                </description>
        </class>
        <class name="basic_quadrotor_behaviors/BehaviorWait'
                base_class_type="nodelet::Nodelet">
                <description>
                        BehaviorWait.
                </description>
        </class>
</library>
```

- basic_quadrotor_behaviors.launch:

```
                          default="drone1" />
havior_srv"               default="activate_behavior" />
behavior_srv"             default="deactivate_behavior" />
ation_conditions_srv"     default="check_activation_conditions'
finished_topic"           default="behavior_activation_finishe
ose_topic"                default="self_localization/pose" />
_topic"                   default="command/high_level" />
gles_topic"               default="rotation_angles" />
yaw_srv"                  default="droneOdometryStateEstimator/
ief"                      default="consult_belief" />
stem"                     default="basic_quadrotor_behaviors" /
r"                        default="change_self_localization_mo
ace)/$(arg behavior_system)">
~namespace"                           value="$(arg namespace)'
~activate_behavior_srv"               value="$(arg activate_b
~deactivate_behavior_srv"             value="$(arg deactivate_
~check_activation_conditions_srv"     value="$(arg check_acti
~estimated_pose_topic"                value="$(arg estimated_
~controllers_topic"                   value="$(arg controller
~rotation_angles_topic"               value="$(arg rotation_a
~initialize_yaw_srv"                  value="$(arg initialize_
~consult_belief"                      value="$(arg consult_be
~behavior_system"                     value="$(arg behavior_s
~odometry_str"                        value="$(arg odometry_s

 type="nodelet" args="manager" name="basic_quadrotor_behavior
 type="nodelet" args="load basic_quadrotor_behaviors/Behavior
e_off" output="screen"/>
 type="nodelet" args="load basic_quadrotor_behaviors/Behavior
alize_with_odometry" output="screen"/>
 type="nodelet" args="load basic_quadrotor_behaviors/Behavior
d" output="screen"/>
 type="nodelet" args="load basic_quadrotor_behaviors/Behavior
t" output="screen"/>
```

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.upm@gmail.com

The content of the Aerostack wiki is licensed under Creative Commons license CC BY 4.0

Vision4UAV / **Aerostack**

How to Program a New Process

Edit New Page

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 53

Getting started

- What is Aerostack
- Install Aerostack
- Launch Real Flight
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Mission Plan Languages
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

- Execute Missions
- Program New Behaviors
- Add New Hardware
- Setup Communications

International competitions

- 1st prize at IMAV 2013
- 4th place at IMAV 2016
- 2nd prize at IMAV 2017

More information

- Publications
- Research Group CVAR

Clone this wiki locally

https://github.com/Visior

```
        include/
          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.
 *          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 NodeHandle 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 me
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, th
}

//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, th
}

//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 f
  }

  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

Vision4UAV / **Aerostack**

The Class RobotProcess

Edit New Page

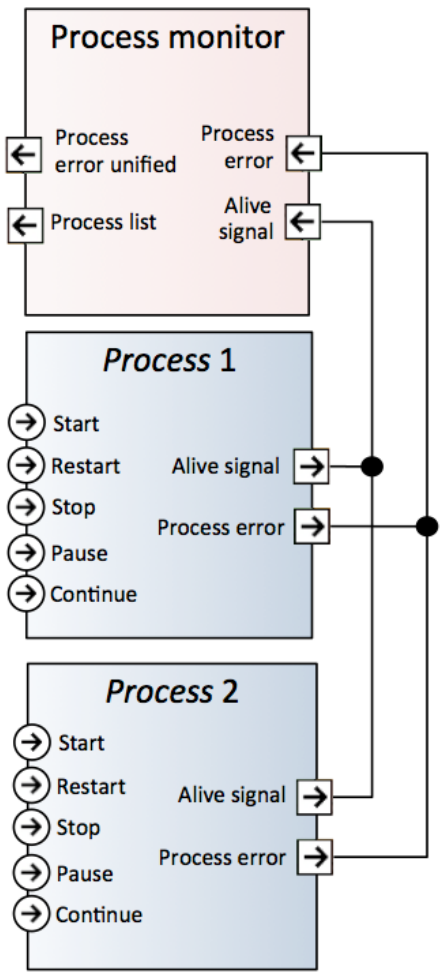
Martin Molina edited this page on 8 Nov 2017 · 3 revisions

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 53

Getting started

- What is Aerostack
- Install Aerostack
- Launch Real Flight
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Mission Plan Languages
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

- Execute Missions
- Program New Behaviors
- Add New Hardware
- Setup Communications

International competitions

- 1st prize at IMAV 2013
- 4th place at IMAV 2016
- 2nd prize at IMAV 2017

More information

- Publications
- Research Group CVAR

Clone this wiki locally

https://github.com/Visior

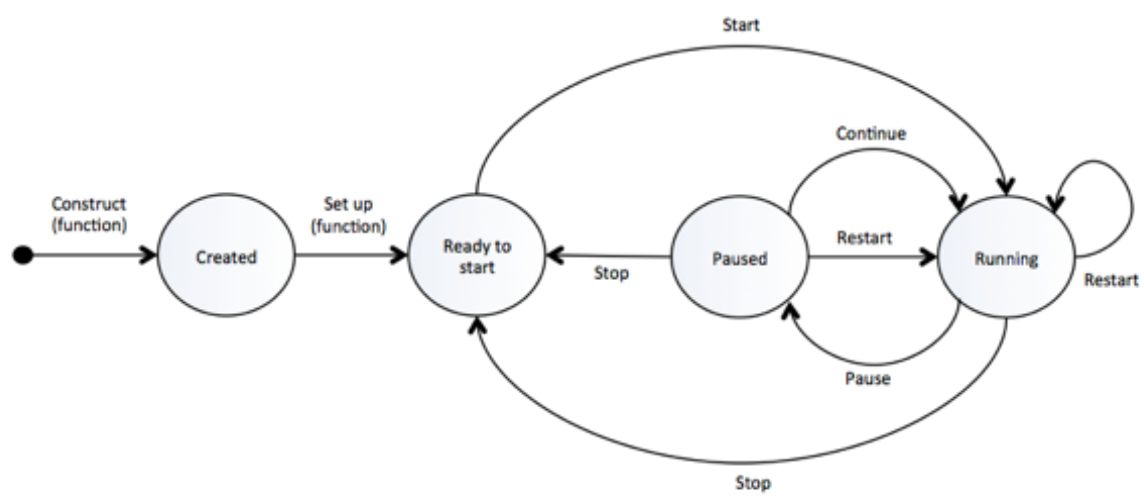
- *Synchronous/asynchronous execution*. During the execution, the process develops a series of data transformations. The execution can be asynchronous if data transformations are performed in response to changes in inputs (new messages in subscribed ROS topics) or in response to service calls from other 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

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
```

```
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

Vision4UAV / **Aerostack**

Add new hardware

Edit New Page

Martin Molina edited this page on 23 Feb 2018 · 1 revision

Here you can find how to do the following tasks:

- Calibrate Controllers
- Add a New Camera
- Add a GPS Sensor
- Use Aerostack for a New Physical Platform

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

Pages 53

Getting started

- What is Aerostack
- Install Aerostack
- Launch Real Flight
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Mission Plan Languages
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

- Execute Missions
- Program New Behaviors
- Add New Hardware
- Setup Communications

International competitions

- 1st prize at IMAV 2013
- 4th place at IMAV 2016
- 2nd prize at IMAV 2017

More information

- Publications
- Research Group CVAR

Clone this wiki locally

https://github.com/Visior

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 53

Getting started

- What is Aerostack
- Install Aerostack
- Launch Real Flight
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Mission Plan Languages
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

- Execute Missions
- Program New Behaviors
- Add New Hardware
- Setup Communications

International competitions

- 1st prize at IMAV 2013
- 4th place at IMAV 2016
- 2nd prize at IMAV 2017

More information

- Publications
- Research Group CVAR

Clone this wiki locally

https://github.com/Visior

```
  <ALTITUDE_FOR_LANDING_WITH_TIME> 0.40   </ALTITUDE_FOR_LANDING_WITH_TIME>  <!-

  <LANDING_WITH_TIME_MAX_SECONDS>     1.00 </LANDING_WITH_TIME_MAX_SECONDS>  <!-

  <LANDING_WITH_TIME_FIXED_THRUST_KG> 0.32 </LANDING_WITH_TIME_FIXED_THRUST_KG>
</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_CONSID
</MAX_ALTITUDE>


<CONTROLLER_SATURATIONS>
  <ML_AUTOPILOT_MAX_ALLOWED_YAW_SPEED> 30.0</ML_AUTOPILOT_MAX_ALLOWED_YAW_SPEED> <

  <ML_AUTOPILOT_MAX_ALLOWED_ALTITUDE_SPEED>0.5 </ML_AUTOPILOT_MAX_ALLOWED_ALTITUDE
</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.
- 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_VALU
  <LL_AUTOPILOT_COMMAND_TILT_MAX_VALUE>+51.175</LL_AUTOPILOT_COMMAND_TILT_MAX_VALU
  <LL_AUTOPILOT_COMMAND_DYAW_MIN_VALUE>-254.760</LL_AUTOPILOT_COMMAND_DYAW_MIN_VAL
  <LL_AUTOPILOT_COMMAND_DYAW_MAX_VALUE>+254.760</LL_AUTOPILOT_COMMAND_DYAW_MAX_VAL
  <LL_AUTOPILOT_COMMAND_THRUST_MIN_VALUE>0.000</LL_AUTOPILOT_COMMAND_THRUST_MIN_VA
  <LL_AUTOPILOT_COMMAND_THRUST_MAX_VALUE>+32.000</LL_AUTOPILOT_COMMAND_THRUST_MAX_
</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:(          45
  <COMMAND_LOWER_LIMIT> -0.11  </COMMAND_LOWER_LIMIT>  <!-- Pelican: ( (-1.0) *  1
  <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

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

Vision4UAV / **Aerostack**

Add a New Camera

Edit New Page

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 53

Getting started

- What is Aerostack
- Install Aerostack
- Launch Real Flight
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Mission Plan Languages
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

- Execute Missions
- Program New Behaviors
- Add New Hardware
- Setup Communications

International competitions

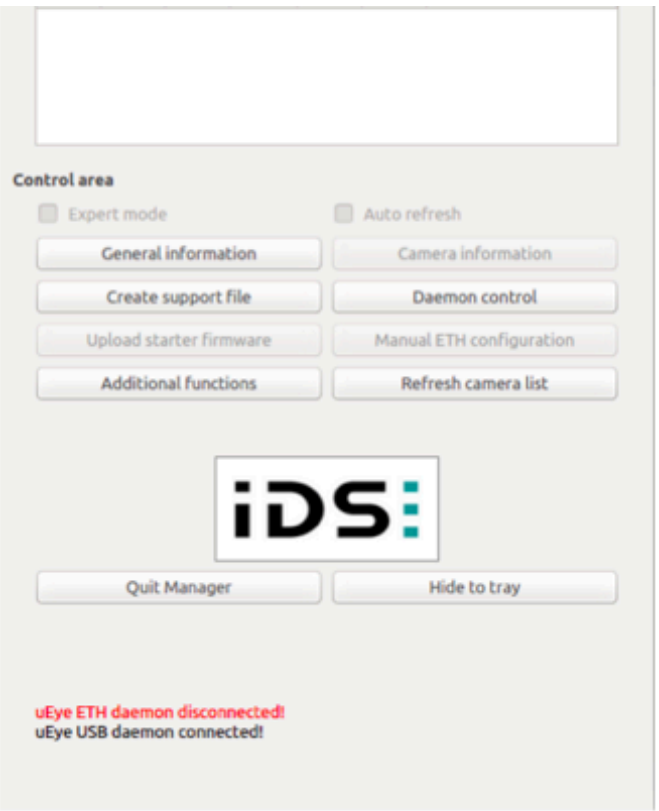
- 1st prize at IMAV 2013
- 4th place at IMAV 2016
- 2nd prize at IMAV 2017

More information

- Publications
- Research Group CVAR

Clone this wiki locally

https://github.com/Visior



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)_$(arg camera_name_str)" pkg="dri
      <param name="camera_ident_number" value="$(arg camera_id_num)" type="int"/>
      <param name="configuration_file" value="$(arg my_stack_directory)/configs/dr
      <param name="camera_name"   value="$(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/dr
     </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

Vision4UAV / **Aerostack**

Add a GPS Sensor

Edit New Page

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 53

Getting started

- What is Aerostack
- Install Aerostack
- Launch Real Flight
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Mission Plan Languages
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

- Execute Missions
- Program New Behaviors
- Add New Hardware
- Setup Communications

International competitions

- 1st prize at IMAV 2013
- 4th place at IMAV 2016
- 2nd prize at IMAV 2017

More information

- Publications
- Research Group CVAR

Clone this wiki locally

https://github.com/Visior

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_s

        <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="scre

       </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);
```

```
        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

Vision4UAV / **Aerostack**

Setup Communications

Edit New Page

Martin Molina edited this page on 23 Feb 2018 · 1 revision

Here you can find how to do the following tasks:

- Setup the Communication Network
- Wi Fi Communication with Multiple Drones

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

Pages 53

Getting started

- What is Aerostack
- Install Aerostack
- Launch Real Flight
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Mission Plan Languages
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

- Execute Missions
- Program New Behaviors
- Add New Hardware
- Setup Communications

International competitions

- 1st prize at IMAV 2013
- 4th place at IMAV 2016
- 2nd prize at IMAV 2017

More information

- Publications
- Research Group CVAR

Clone this wiki locally

https://github.com/Visior

Vision4UAV / **Aerostack**

Setup the Communication Network

Edit New Page

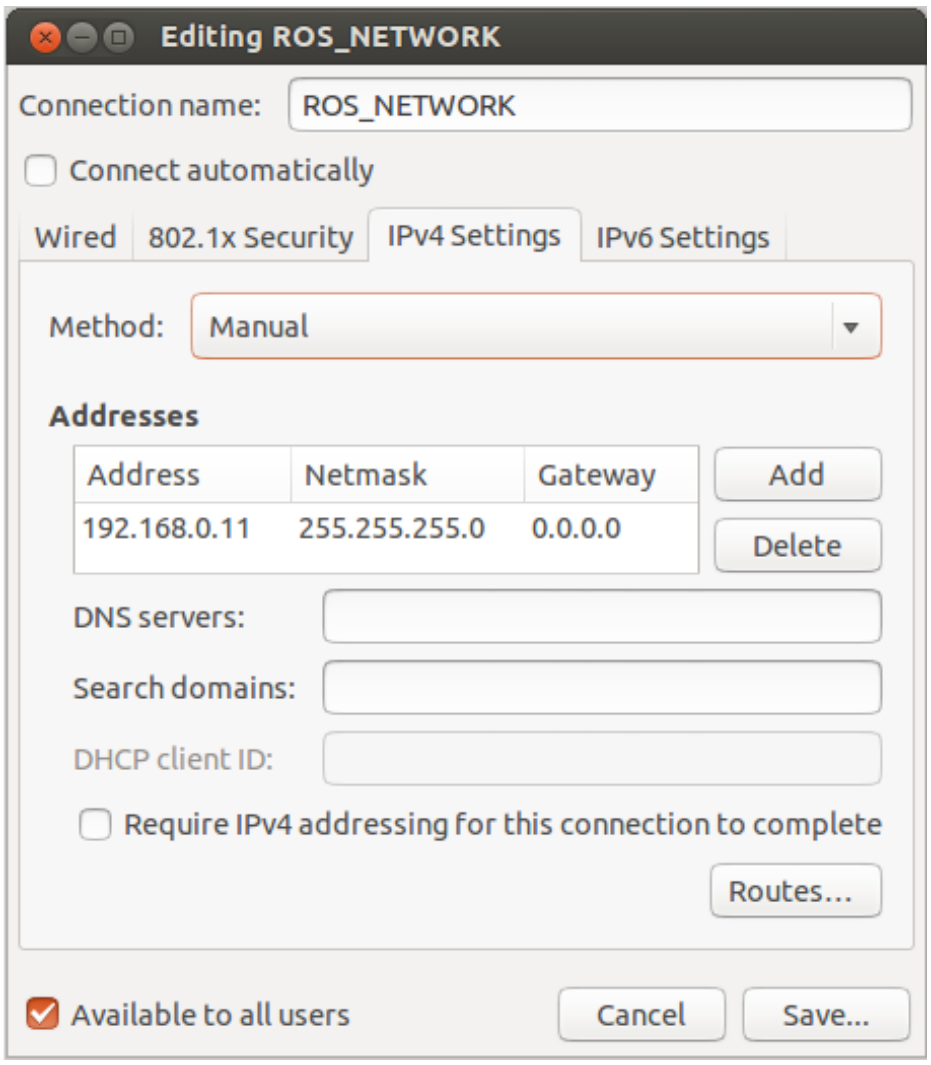
Martin Molina edited this page on 26 Oct 2017 · 2 revisions

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 53

Getting started

- What is Aerostack
- Install Aerostack
- Launch Real Flight
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Mission Plan Languages
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

- Execute Missions
- Program New Behaviors
- Add New Hardware
- Setup Communications

International competitions

- 1st prize at IMAV 2013
- 4th place at IMAV 2016
- 2nd prize at IMAV 2017

More information

- Publications
- Research Group CVAR

Clone this wiki locally

https://github.com/Visior

```
192.168.0.10    ROS_10
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

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 53

Getting started

- What is Aerostack
- Install Aerostack
- Launch Real Flight
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Mission Plan Languages
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

- Execute Missions
- Program New Behaviors
- Add New Hardware
- Setup Communications

International competitions

- 1st prize at IMAV 2013
- 4th place at IMAV 2016
- 2nd prize at IMAV 2017

More information

- Publications
- Research Group CVAR

Clone this wiki locally

https://github.com/Visior

```
vi bin/drone_wifi_setup.sh
```

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

Use Aerostack for a New Physical Platform

Martin Molina edited this page on 24 Jul 2019 · 5 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 in 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.

Pages 38

Getting started

- What is Aerostack
- Install Aerostack
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interfac
- Mission Plan Language
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

- Example Projects
- Program New Behavior
- Add New Hardware
- Setup Communications

International competitions

- 3rd place at MBZIRC 2(
- 2nd prize at IMAV 2017
- 4th place at IMAV 2016
- 1st prize at IMAV 2013

More information

- Publications
- Research Group CVAR

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 accept roll, pitch, dyaw (change in yaw) and throttle commands.

2. Getting the Rotation Angles from the Pixhawk Using Mavros

The Aerostack accepts rotation angles in NED (North, East and Down). 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
                 msg.orientation.z, msg.orientation.

//make a matrix m of quaternion q
tf::Matrix3x3 m(q);


//convert quaternion to euler angels
 m.getEulerYPR(yaw, pitch, roll);

 RotationAnglesMsgs.vector.x = (double)(+1)*(roll)*1
 RotationAnglesMsgs.vector.y = (double)(-1)*(pitch)*
 RotationAnglesMsgs.vector.z = (double)(-1)*(yaw)*18

 //converting the yaw which is in the range from -ph
degrees
 if( RotationAnglesMsgs.vector.z < 0 )
 RotationAnglesMsgs.vector.z += 360.0;

 publishRotationAnglesValue(RotationAnglesMsgs);
```

Clone this wiki locally

https://github.com/Visi

The above code takes the rotations given by mavros in quaternion converts them into roll, pitch and yaw angles. These 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 Mavros

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 the percentage required by the Aerostack.

```
droneMsgsROS::battery BatteryMsgs;

// Read Battery from navdata
BatteryMsgs.header   = msg.header;
// minimum voltage per cell used by the "digital ba
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

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 Mavros

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 these conversions and finally sends the commands to the pixhawk using mavros.

The code below shows how commands are sent using mavros to the pixhawk.

```
        //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(
        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 f
         quaterniond =  mavros::UAS::quaternion_from_rpy(rol
                                        pitch_command, yaw

         //convert quaternion in eigen form to geometry_msg
         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 mavros.

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.upm@gmail.com

The content of the Aerostack wiki is licensed under Creative Commons license CC BY 4.0

3rd place at MBZIRC 2020

Edit New Page

Martin Molina edited this page on 23 Mar · 5 revisions

Our group CVAR (Computer Vision and Aerial Robotics) from the "Universidad Politecnica de Madrid" won the third place of the Grand Challenge in the International Robotic Competition MBZIRC 2020 that took place in February 25, 2020. The Aerostack software framework was successfully used for building the system architecture of aerial robots that participated in the competition as it is described in this publication.

Our group CVAR participated as the coordinator of SkyTeam, a combined team that also integrated members from Universidad Pablo Olavide and Poznan University of Technology. The group SkyTeam was in charge of Challenge 1 and Challenge 3. The group French National Center for Scientific Research (CNRS) was in charge of Challenge 2.

Pages 38

Getting started

- What is Aerostack
- Install Aerostack
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interfac
- Mission Plan Language
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

- Example Projects
- Program New Behavior
- Add New Hardware
- Setup Communications

International competitions

- 3rd place at MBZIRC 2(
- 2nd prize at IMAV 2017
- 4th place at IMAV 2016
- 1st prize at IMAV 2013

More information

- Publications
- Research Group CVAR

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.upm@gmail.com

The content of the Aerostack wiki is licensed under Creative Commons license CC BY 4.0

Clone this wiki locally

```
https://github.com/Visi
```

Vision4UAV / **Aerostack**

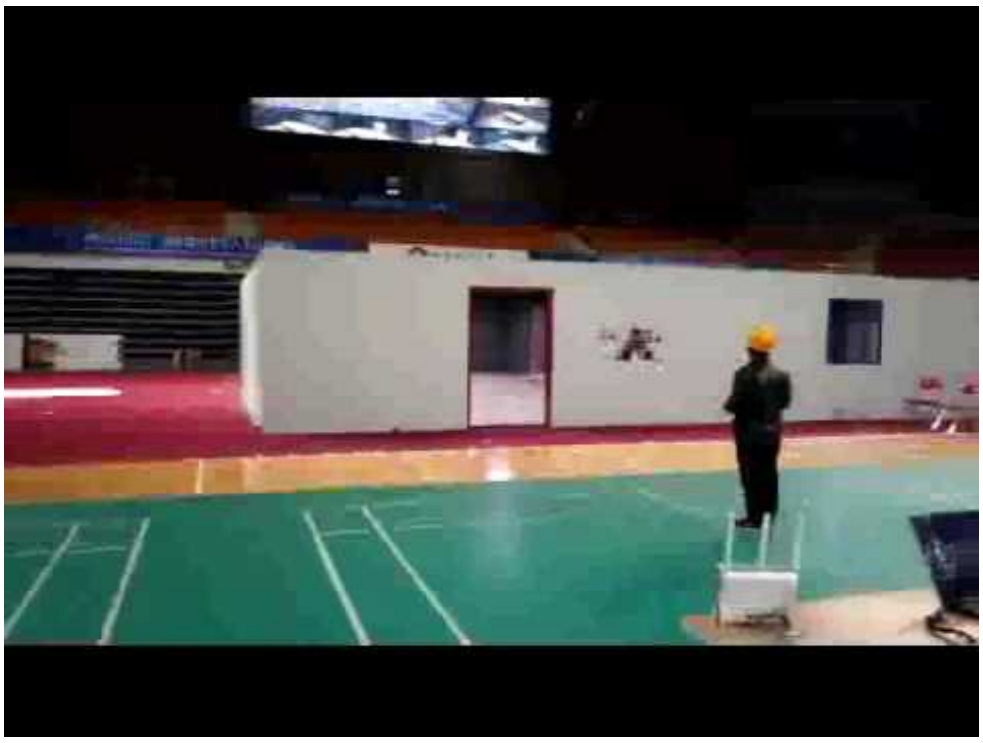
4th place at IMAV 2016

Edit New Page

Martin Molina edited this page on 18 Mar 2018 · 1 revision

This page shows some videos corresponding to the International Competition at IMAV 2016, where our group achieved the 4th place.

Entering the house:



Autonomous landing:

In the house:

Pages 53

Getting started

- What is Aerostack
- Install Aerostack
- Launch Real Flight
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interface
- Mission Plan Languages
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

- Execute Missions
- Program New Behaviors
- Add New Hardware
- Setup Communications

International competitions

- 1st prize at IMAV 2013
- 4th place at IMAV 2016
- 2nd prize at IMAV 2017

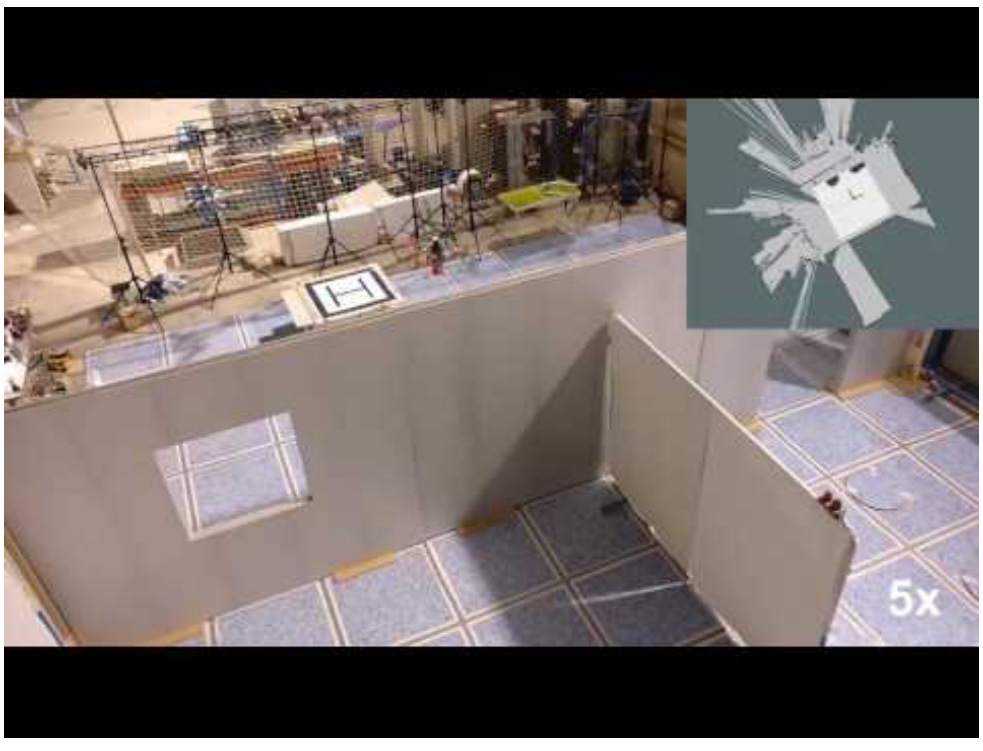
More information

- Publications
- Research Group CVAR

Clone this wiki locally

https://github.com/Visior

Experimental flights:



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

Martin Molina edited this page 4 days ago · 27 revisions

If you are using Aerostack within the research for your publication, please cite:

- 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, 88, 683–709.

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, 88, 683–709.
- 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.

Mission execution control

- M. Molina, A. Carrera, A. Camporredondo, H. Bavle, A. Rodriguez-Ramos, P. Campoy (2020): Building the executive system of autonomous aerial robots using the Aerostack open-source framework. International Journal of Advanced Robotic Systems. 17(3) : 1-20.

Pages 38

Getting started

- What is Aerostack
- Install Aerostack
- License
- List of Distributions

Components

- Aerostack Architecture
- Graphical User Interfac
- Mission Plan Language
- Catalog of Behaviors
- Memory of Beliefs

Tutorials

- Example Projects
- Program New Behavior
- Add New Hardware
- Setup Communications

International competitions

- 3rd place at MBZIRC 2(
- 2nd prize at IMAV 2017
- 4th place at IMAV 2016
- 1st prize at IMAV 2013

More information

- Publications
- Research Group CVAR

- M. Molina, A. Camporredondo, H. Bavle, A. Rodriguez-Ramos, P. Campoy (2019): An execution control method for the Aerostack aerial robotics framework. Frontiers of Information Technology & Electronic Engineering. 20(1) : 60-75.

- 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.

Motion planning

- 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.

- Sanchez-Lopez, J. L., Pestana, J., & Campoy, P. (2017, June). A robust real-time path planner for the collision-free navigation of multirotor aerial robots in dynamic environments. In 2017 International Conference on Unmanned Aircraft Systems (ICUAS) (pp. 316-325). IEEE.

Human Robot Interaction

- M. Molina, P. Frau, D. Maravall (2018). A Collaborative Approach for Surface Inspection Using Aerial Robots and Computer Vision. Sensors, 18(3), 893; doi:10.3390/s18030893

- 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.

Clone this wiki locally

```
https://github.com/Visi
```

- 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.

Motion control

- Sampedro, C., Bavle, H., Rodriguez-Ramos, A., de la Puente, P., & Campoy, P. (2018, October). Laser-Based Reactive Navigation for Multirotor Aerial Robots using Deep Reinforcement Learning. In 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (pp. 1024-1031). IEEE.

- Rodriguez-Ramos, A., Sampedro, C., Bavle, H., Moreno, I. G., & Campoy, P. (2018, October). A Deep Reinforcement Learning Technique for Vision-Based Autonomous Multirotor Landing on a Moving Platform. In 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (pp. 1010-1017). IEEE.

- 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

- 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.

- 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.

Sensoring and estimation

- Bavle, H., Manthe, S., de la Puente, P., Rodriguez-Ramos, A., Sampedro, C., & Campoy, P. (2018, October). Stereo Visual Odometry and Semantics based Localization of Aerial Robots in Indoor Environments. In 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (pp. 1018-1023). IEEE.

- Bavle, H., Sanchez-Lopez, J., de la Puente, P., Rodriguez-Ramos, A., Sampedro, C., & Campoy, P. (2018). Fast and Robust Flight Altitude Estimation of Multirotor UAVs in Dynamic Unstructured Environments Using 3D Point Cloud Sensors. Aerospace, 5(3), 94.

- 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.

Applications

- Sampedro, C., Rodriguez-Ramos, A., Bavle, H., Carrio, A., de la Puente, P., Campoy, P. (2018). A fully-autonomous aerial robot for search and rescue applications in indoor environments using learning-based techniques. Journal of Intelligent & Robotic Systems 95: 601–627.

Competitions

MBZIRC 2020:

- R. A. Suarez Fernandez, A. Rodriguez-Ramos, A. Alvarez, J. Rodriguez-Vazquez, H. Bavle, L. Liang, M. Fernandez, A. Rodelgo, J.A. Cobano, D. Alejo, D. Acevedo, R. Rey, S. Martinez-Rozas, M. Molina, L. Merino, F. Caballero, P. Campoy (2020). The Skyeye Team Participation in the 2020 Mohamed Bin Zayed International Robotics Challenge. Mohammad Bin Zayed International Robotics Competition (MBZIRC) Simposium. Abu Dhabi.

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.

Other related papers:

- 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.
- 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.

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.upm@gmail.com

The content of the Aerostack wiki is licensed under Creative Commons license CC BY 4.0