Copyright Information

Marea project: UAV Landing procedure on a moving and floating platform

Andrea Nisticò*, Marco Baglietto*, Enrico Simetti*†, Giuseppe Casalino*†, Alessandro Sperindè*†
*DIBRIS, University of Genova, Via Opera Pia 13, 16145 Genova, Italy
†Interuniversity Research Center on Integrated Systems for the Marine Environment
Via Opera Pia 13, 16145 Genova, Italy

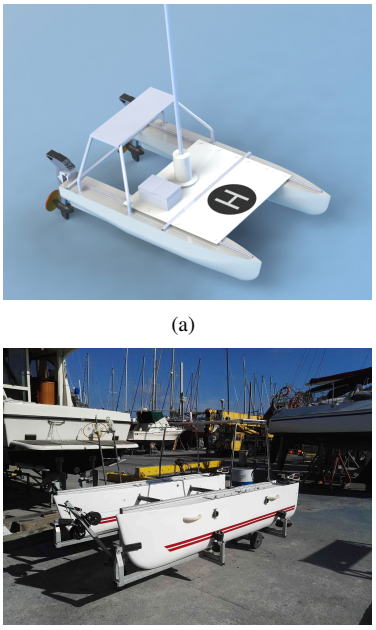
***Abstract*—In this paper we consider the problem of autonomous landing on a horizontally moving platform with vertical unpredictable oscillatory dynamics using a quadrotor system. The quadrotor is equipped with an external Raspberry PI as a companion computer used for communications. The task is divided in two subproblems: tracking and landing. We present the algorithms involved for the entire procedure; a PI regulator is used for the tracking problem while descending is made by controlling relative vertical velocity. A finite state machine approach is chosen to manage multiple robot states and recover from failures. A software framework was developed in order to manage general flight missions and, in this case, the landing assignment. At the end, we performed simulation and real experiments in order to validate the outcome of this work.**

I. Introduction

The framework of this work is MAREA, an Italian research project based on a consortium of university and companies with the purpose of implementing novel strategies for automatic search and rescue operations by exploiting the cooperation between heterogeneous robotic platforms. These demanding tasks are increasingly performed by semi-autonomous vehicles like MAVs (Micro Aerial Vehicles), allowing the exploration of remote areas but increasing the need for automatic management in the entire infrastructure. In particular, during the whole autonomous assignment, the landing procedure is one of the most dangerous and challenging process.

Around 50 % of UAVs (Unmanned Aerial Vehicles) suffer accidents during landing and 70 % of them are attributed to human factors [1]. An automatic or assisted landing procedure could significantly decrease the number of accidents; moreover, with the vehicles constantly decreasing in size, higher precision is required. In the case of MAVs, atmospheric agents such as wind and waves have a very high impact on the dynamics which are hard to handle by human pilots or standard guidance systems.

The main focus of this work is to automatize the entire procedure for a scenario which consists in a commercial quadcopter taking off, searching for the moving target on the ocean surface and then safely landing on the dedicated pad. ISME has been working on marine robotics for search and rescue [2] since many years and thanks to the knowledge obtained in a previous work [3], a boat prototype, shown in figure 1, is been developed and intended to be used in the real application.



(a)

(b)

Fig. 1. (a) boat rendering with target platform; (b) actual vehicle under construction

For practical reasons, we started with the design of a wider software framework capable of dealing with general flight missions by sending the correct commands to the MAV. This architecture, called *mocap2mav* [4], consists of a modular lightweight off-board control station written in C++ where we encapsulated the modules and algorithms for the landing task. Most important features of the software are:

- **Modularity**: modules are independent programs communicating with each other through Lightweight Communications and Marshalling middleware (LCM) [5].
- **Expandability**: consequence of being modular, the architecture is easily expandable.
- **Universality**: the software does not have prior knowledge on the particular quadrotor, this means that through few sw adapter components it is compatible with different systems.

Under this framework, the addition of dedicated modules for the landing procedure is straightforward. The task complexity requires the handling of various situations (approaching,

tracking, landing fail etc.): we expanded previously developed works on similar scenarios [6], [7] by choosing a finite state machine approach and adding vertical compensation induced by waves in the switching logic and algorithm. The considered framework tackles two main jobs separately:

- **Horizontal tracking**: the procedure of aligning with the center of the moving boat on the horizontal plane.
- **Vertical compensation**: the task of getting closer to the target by reducing the height and compensating for vertical oscillation induced by floating dynamics.

This paper will recall briefly the system setup as well as presenting the proposed control algorithms implemented by the software. Some experimental results in a laboratory environment, showing the MAV undergoing the entire procedure, are presented.

II. Flight stack and onboard controllers

Since this application is thought for available commercial drones, development of the flight architecture is not our concern. Usually these pieces of software are designed by specialized teams from the manufacturer company or, as in our case, from the open source community. A flight stack consists in the Firmware which is installed on the autopilot board; the stack manages every hardware communication by running drivers as well as performing high speed control loops and localization. Many of these core architectures are available on the market as they are usually provided with the quadrotor itself. As a consequence, our work aims to be compatible with most architecture types by developing software modules that can be adapted with few effort to the specific platform.

Nevertheless, it is worth presenting briefly the flight control algorithms in order to better understand theoretical and practical aspects of this work, even if they run independently from our system on the autopilot board.

A. State estimation

As a primal assumption, we consider the robot and platform position measure as given. At this stage, position measurement is performed by a Motion Capture (MoCap) system composed by 8 infra red cameras attached to the roof; passive reflective markers are placed on the MAV's body and position feedback is given by the MoCap which calculates markers pose. The calculated values are then injected in the PX4 [8] sensor fusion module which consist in a Kalman Filter: it takes as input different sensors such as GPS, barometer, inertial sensors and vision sensors in order to estimates the robot state (position and orientation). We use the vision sensor channel, which is dedicated to position measures coming from cameras, optical flows or laser scanners, in order to pass the MoCap raw pose to the autopilot.

On the other hand, we directly use the platform raw pose as the real value while velocity is calculated by a discrete derivative between actual and previous measure. A fixed window filter performs the desired smoothing on the velocity signal on each axis.

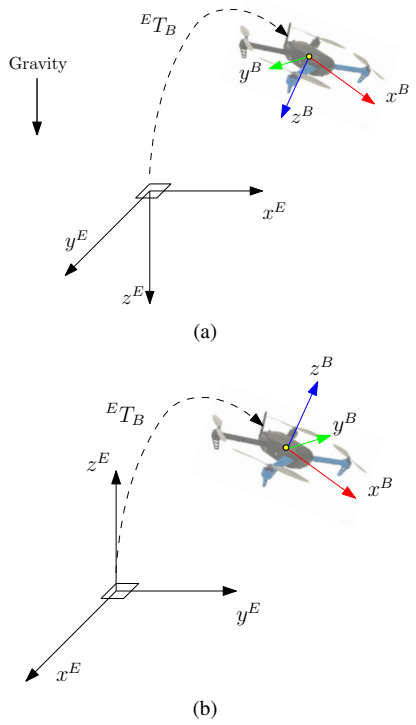


Fig. 2. (a) NED representation; (b) ENU representation

B. PX4 Autopilot control scheme

We chose to work with the PX4 autopilot because it has rapid and active development community, a clear and structured code and it is open-source. PX4 manages the entire robot infrastructure: this section will focus only on the control modules because they represent the interface with our application.

1) Transformation frames: Before presenting the control scheme, we should shortly describe the adopted transformation frames. PX4 uses NED (North-East-Down) coordinate system which consists in having a local frame, ground fixed, with the x axis pointing north, y towards east and z downwards by definition. The angles are represented by roll (rotation on x), pitch (rotation on y) and yaw (rotation on z). NED frame is usually adopted in aeronautics, the downwards z is to achieve the historical definition of a positive pitch when going up. On the other hand, many robotics applications adopts ENU (East-North-Up) frames where x is East, z is up and y accordingly.

In both representation, a body frame is attached to the robot. In NED this will result in x forward and z down while in ENU x forward and z up. The robot pose is described through the transformation $^{E}T_{B}$ between body frame respect to earth frame.

Our application uses ENU frame, the conversion between the two systems is entirely made by PX4 internally. Figure 2 shows both earth and body frames in the two representations.

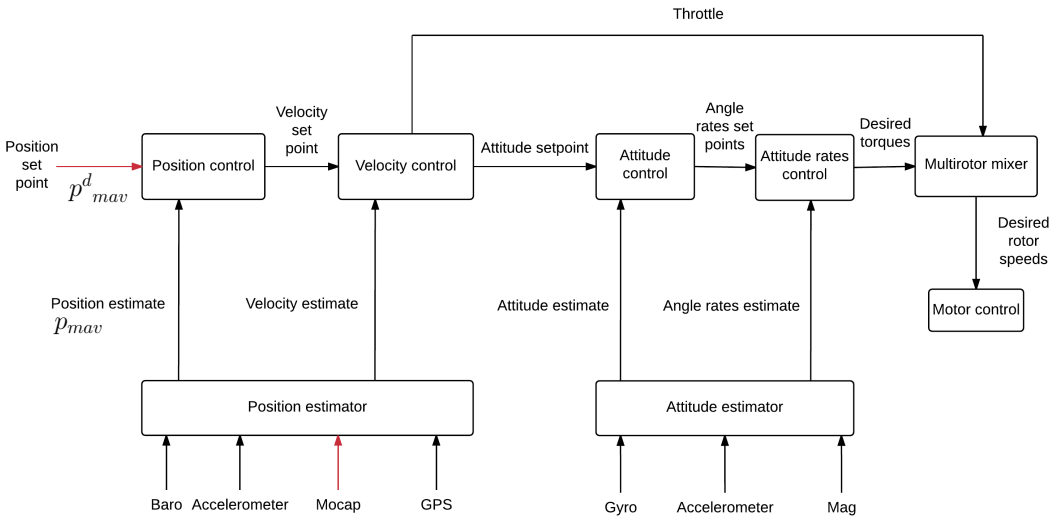


Fig. 3. PX4 Firmware control scheme.

2) Control scheme: Software design is divided in five sub controllers placed in a cascaded fashion one after the other. Each controller generates the input for the next one. From the highest to lowest level we have: *position control*, *velocity control*, *attitude control*, *attitude rate control* and *motor control*. The entire pipeline is shown in figure 3 alongside with the estimation modules, red arrows represent signals which are externally provided namely: **position setpoint** and **position feedback**. On the other hand, being an indoor setup, GPS is not available but depicted for completeness.

The first module, *position control*, implements a proportional law using as reference the provided setpoint and as state the position feedback and generates a velocity setpoint. Formally we have:

$$\begin{aligned} \boldsymbol{e_p} &= \boldsymbol{p^d_{mav}} - \boldsymbol{p_{mav}} \\ \boldsymbol{v^d_{mav}} &= K\boldsymbol{e_p} \end{aligned} \tag{1}$$

where K is the proportional gain; $\boldsymbol{p_{mav}}$, $\boldsymbol{p^d_{mav}}$ and $\boldsymbol{v^d_{mav}}$ are 3-elements vectors and they represent respectively the actual robot position in earth frame, the desired set-point which is externally received and the generated target velocity fed to the next controller. The other cascaded modules are PID regulators, they take as reference the previous module output while state is provided by estimators.

It is worth stressing the fact that our MAV takes as input only position setpoints (which do not coincide with platform pose) and feedback. It will try to track the provided setpoint which will be generated by an external application running on a ground machine. More details are provided in the next sections.

III. Landing Procedure

This section starts describing briefly how we tackled the problem and then it presents the proposed solution from a theoretical point of view.

First, let us recall some of considerations:

- the absolute pose is provided by IR cameras mounted around the flight arena;
- we consider the MAV as closed system, it has its own controllers with their own dynamics;
- the system input is a reference position in space (3-dimensional vector);

With that said, two distinct but dependent phases can be easily recognized during the entire procedure: **horizontal tracking**, which consists in aligning the MAV's center of mass with the landing spot on the horizontal plane (x and y), and **landing** which consists in getting closer to the target by reducing the height and compensating for vertical oscillation.

A. Horizontal tracking

Let us define $\boldsymbol{p^d_{mav,xy}}$ as the desired horizontal position (or the system input) which is sent to the MAV by the mocap2mav architecture and $\boldsymbol{p_{plat,xy}}$ as the platform horizontal position. These quantities are defined as 2-dimensional vectors representing only x and y coordinates. Hence we can define the horizontal error vector as:

$$\hat{\boldsymbol{e}}_{\boldsymbol{p,xy}} = \boldsymbol{p_{plat,xy}} - \boldsymbol{p_{mav,xy}} \tag{2}$$

The system response (between desired and actual position, namely input and output) is comparable to a second order system with a transient and oscillations before reaching the steady state. One could think of assigning the platform position as reference but, in the case of a moving target, convergence will not be achieved in most of the cases. Transient time will induce a delay and the MAV will stay behind the platform with a steady error in position.

The idea is to issue a reference point which is in front of the moving platform, along movement direction, such that the

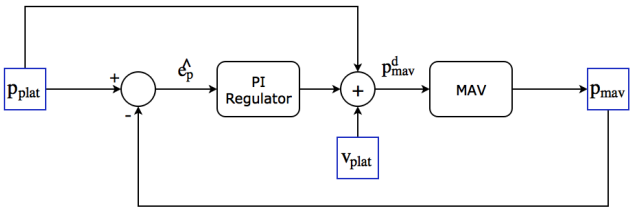


Fig. 4. Tracking control scheme

drone will be aligned with the landing spot. Hence the final reference position sent to the robot will be a PI regulator plus a feedforward component.

We can now introduce the PI regulation on the position error as:

$$\boldsymbol{p^d}_{\boldsymbol{mav,xy}} = \boldsymbol{p}_{\boldsymbol{plat,xy}} + k_p \hat{\boldsymbol{e}}_{\boldsymbol{p,xy}} + k_i \int \hat{\boldsymbol{e}}_{\boldsymbol{p,xy}} + k_f \boldsymbol{v}_{\boldsymbol{plat,xy}} \tag{3}$$

The first term is an offset (starting condition of the integrator) while the other two terms represent the PI controller. Last member is a feedforward on the platform velocity.

Equation 3 calculates the desired setpoint which is the position controller input defined in equation 1. We are basically inverting the PX4 position control in order to send a position reference which will internally generate a desired velocity setpoint. In fact, by substituting equation 3 in (1) and setting $K = 1$ we obtain:

$$\boldsymbol{v^d}_{\boldsymbol{mav,xy}} = (k_p + 1)\hat{\boldsymbol{e}}_{\boldsymbol{p,xy}} + k_i \int \hat{\boldsymbol{e}}_{\boldsymbol{p,xy}} + k_f \boldsymbol{v}_{\boldsymbol{plat,xy}} \tag{4}$$

Equation 4 shows that tracking is performed exploiting the embedded velocity controller by applying an external loop with reference the platform horizontal position and adding a feedforward term. In short, we send over Wi-Fi $\boldsymbol{p^d}_{\boldsymbol{mav,xy}}$ in order to generate internally $\boldsymbol{v^d}_{\boldsymbol{mav}}$. This concept is depicted in figure 4.

B. Landing

In order to cancel vertical oscillatory movement and descend slowly on the platform, our goal is to keep the MAV descending velocity respect to the platform equal to a desired value. Defining $v^{desc}_{mav,z}$ as the desired MAV vertical velocity respect to the platform we have:

$$v^{desc}_{mav,z} = v^{d}_{mav,z} - v_{plat,z} \tag{5}$$

where $v^d_{mav,z}$ and $v_{plat,z}$ are the absolute robot and platform velocities in order to obtain $v^{desc}_{mav,z}$ as relative descendant rate.

Inverting (5) we obtain:

$$v^{d}_{mav,z} = v^{desc}_{mav,z} + v_{plat,z} \tag{6}$$

In order to improve stability, we add a proportional gain on the velocity error; let us define $v^{target}_{mav,z}$ as the final target velocity, we have:

$$v^{target}_{mav,z} = v^{d}_{mav,z} + K_v e_{v,z} \tag{7}$$

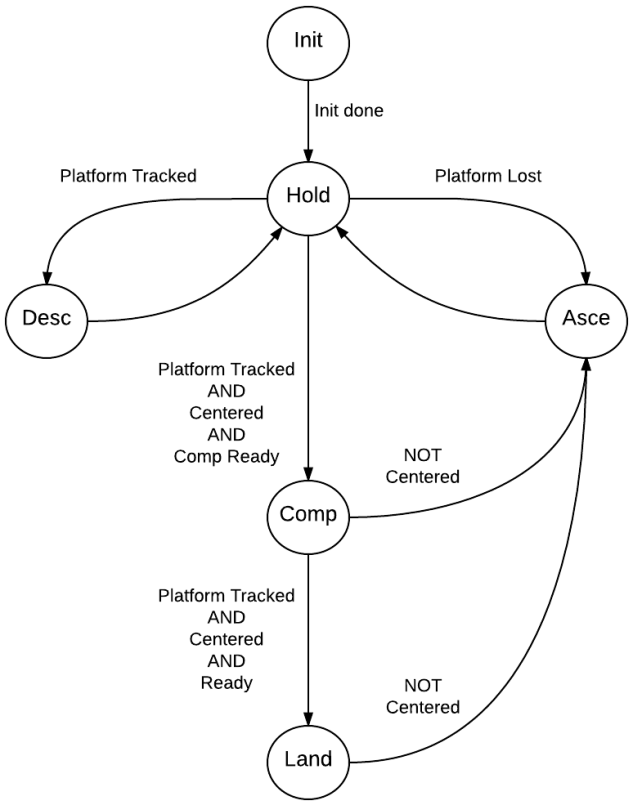


Fig. 5. State machine diagram.

where

$$e_{v,z} = v^{d}_{mav,z} - v_{mav,z} \tag{8}$$

is the difference between desired and actual MAV absolute velocities.

The key step in this procedure is the choice for the desired relative velocity $v^{desc}_{mav,z}$. This quantity is a negative value and the magnitude is calculated with a linear relation respect to the vertical distance $\hat{e}_{p,z}$ from the landing surface:

$$|v^{desc}_{mav,z}| = R\hat{e}_{p,z} \tag{9}$$

This desired value is clamped between a minimum of 0.1 m/s and a maximum of 0.6 m/s.

In order to obtain this behavior, we need to generate a position setpoint. Thus, we will send to MAV the height reference which will generate internally the vertical target velocity calculated in (7). This setpoint will be:

$$p^{d}_{mav,z} = p_{mav,z} + v^{target}_{mav,z} \tag{10}$$

Next step consists in describing how those simple mathematical tools are implemented in order to obtain the full procedure.

IV. Procedure Implementation: Finite State Machine

In section III we present the mathematical equations involved in tracking and descending; the purpose of this part is

to describe how the above equations take part in the overall procedure which not only consists in tracking and descending, but also manages unpredictable states such as losing the platform.

Following a previous work [6], we decided to adopt a similar solution: a finite state machine approach. Therefore, the landing procedure is modeled with different states while the system reacts depending on the actual one.

To each state is mapped a system behavior, an algorithm which is called at each iteration. The following list describes each state:

- *INIT*: initialization state, the procedure entry point. In this state, the robot will fly to its maximum tracking altitude.
- *HOLD*: tracking state. The MAV tracks the platform horizontally using the equations explained in section III-A.
- *DESCEND*: when entering this state, the altitude setpoint is decreased by a small delta, if the minimum tracking altitude h_{min} is not reached; otherwise it is clamped to h_{min}.
- *ASCEND*: in this state, the altitude setpoint is increased by a small delta, if the maximum tracking altitude h_{max} is not reached; otherwise it is clamped to h_{max}.
- COMPENSATION: this states implements the altitude compensation explained in section III-B, combined with tracking algorithm as in *HOLD* state.
- *LAND*: at this stage, motors are set to minimum rate.

On the other hand, signals represent the switching condition for each state: in other words, the event that may occur in order to jump form one particular state to another. Signals are implemented as booleans and consist in the following:

- *Init Done*: true when initialization phase ends, implements a wait function.
- *Platform Tracked*: true when robot horizontal error respect to the platform center is less than half platform size, for N_{hold} consecutive steps.
- *Platform Lost*: true when robot horizontal error respect to the platform center is higher than half platform size, for N_{lost} consecutive steps.
- *Centered*: true when robot horizontal error respect to the platform center is less than a quarter platform size.
- *Comp Ready*: true when the robot reaches the minimum tracking setpoint maintaining that pose for one second.
- *Ready*: true when the robot is 10 cm above platform surface.

V. Components description

This section describes the components involved in the experiments. Real setup is presented as well as the simulated environment, where software components are tested and validated.

The overall setup consists in multiple software and hardware components working simultaneously in order to accomplish the goal. The following parts can be identified:

- **IRIS quadcopter**: platform used in the experiments.

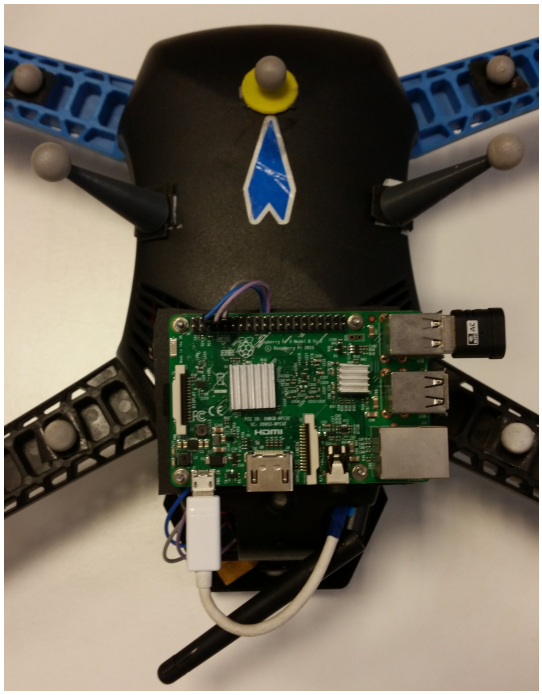


Fig. 6. IRIS drone with IR markers and raspberry

- **PX4 / PiXHawk**: namely the Firmware and the autopilot board on which is running.
- **Raspberry Pi**: the onboard computer implementing Wi-Fi communication and serial interface with PiXHawk.
- **MoCap**: motion capture system.
- **mocap2mav** software: a collection of software modules responsible for generating the correct position setpoints in order to perform a list of predefined task (take off, move, rotate, land on floating boat) defined by the user.

Above listed components can be divided in two groups: offboard and onboard components.

A. Offboard components

These hardware and software units, placed offboard, consist in: motion capture, Motive software and the mocap2mav program. We chose an Optitrack system, as motion capture, composed by 8 "Flex 13" cameras. Each camera has a resolution of 1.3 MP running at 120 frames per second with a 8 ms delay; equipped with IR LEDs, they are able to retrieve the pose of passive reflective markers mounted on the robot's chassis as shown in figure 6 and provide their total center of mass. Motive software is shipped by Optitrack and manages the mocap system; it runs on a Windows machine connected to a LAN network with a Linux computer and a 5 Ghz WiFi router. On the other hand, the mocap2mav software runs on the Linux machine. This architecture is composed by different modules executing in parallel and it was developed for providing a software interface with the IRIS platform, enhancing its features. The main role is to provide a simple solution for allowing a user to execute different tasks with the robot, by task we mean: take off, move, rotate and land.

Through a configuration text file, the user can determine the tasks sequence with their own parameters (e.g. move in a specified location, take off at a specific height) and the robot will execute them one after the other.

The main features are: **Modularity**, **Expandability** and **Universality**. Each module consists in a C++ independent program which is reactive to a specific data stream. Streams are managed by LCM middleware which is responsible for the inter-process communication, it uses a publish-subscribe pattern in which each module can subscribe to a channel (data stream) and read contents from it (C structs). This design choice supports the Modularity and Expandability features. Furthermore, by simply writing an interface module for the target system, we can adapt it to various robots.

The Executioner and Automatic modules represent the application core. Executioner is a program dedicated to mission management, it takes as input the action list and decides whether the actual task is concluded or not, it publishes on the dedicated LCM channel the actual task. On the other hand, the Automatic module, takes as input the actual task and calculates the proper position set-point in order to accomplish the goal. Both modules read form the actual position channel and react to that stream. By reacting we mean that it listens to a specific channel and execute its core loop when data arrive. This means that the specific loop rate is determined by the frequency at which data are streamed. A third module, Lcm2Ros, implements the interface with the specific platform, IRIS in this case. It is written around ROS middleware and simply translates LCM streams in ROS messages and sends them to the robot over WiFi; both middlewares adopt the pub-sub pattern. Using this framework, adding platform landing task was straightforward.

B. Onboard components

These elements represent hardware and software present on the robotic platform, namely IRIS. Hardware components are: PixHawk and RaspberryPI 3. Additionally, software components are the following: PX4 Firmware, Mavros and MocapOptitrack ROS packages.

As previously stated, PixHawk is the onboard autopilot board. It has two co-processors for fail safe, inertial sensors, two barometers, a magnetometer and physical interfaces that are attached to the GPS and motors ESCs (Electronic Speed Controller). The software firmware running on PixHawk is called PX4. Besides implementing the flight stack presented in section II-B, PX4 manages the entire robotic system; it uses Mavlink as a communication protocol streaming more than 20 channels via radio or serial interface such as battery status, actual position or attitude. External communication happens via Mavlink; since radio connection was too weak and unpredictable, we adopted a WiFi communication interface. In order to achieve that, we expanded the computing and hardware capabilities adding a companion computer: a RaspberryPI 3 connected with a serial interface with PixHawk. The OS installed on the Raspberry is Ubuntu MATE 16.06, chosen because it has support for Arm packages and it is very lightweight. Two ROS nodes are executed on the companion computer: mavros and mocapOptitrack. Mavros is a software package written by the community which implements a direct bridge with Mavlink messages sent by PX4.

MocapOptitrack node reads the information coming from Motive and sends the pose to Mavros which redirects it to the autopilot. On the other hand, Mavros receives also the position setpoint coming from the mocap2mav architecture and feeds it to the position controller.

We should remark that LCM was use beside ROS because it is very lightweight and easy to port on different devices, hence we can run the architecture on multiple system types.

Figure 7 depicts the architecture schematics.

C. Overall description

As an example, let us describe the entire procedure. It starts with the MAV grounded and motors at minimum rate, a task list is created by the user which describes the actions sequence to be executed by the robot; as an example, let us define this list as the following: *takeoff* (at a desired altitude), *move* (to a certain point) and *land* (on mobile platform).

The ground control software, namely *mocap2mav*, processes that list and sends the right position reference to the MAV. Starting from take off, a position reference is issued above the robot ground pose at a specific height defined by the user. Reached that position, it is time for the moving action. In this case, as the name implies, a reference is set to a specified position in x, y and z local coordinates. The system checks continuously the ending condition of each action and, if it becomes true, it alerts the Automatic module by providing the new task: that will be landing in this example.

When landing instruction is sent by Executioner to Automatic module, the landing procedure kicks in and the algorithms explained in sections III-A, III-B and IV are executed. Section IV introduces the reader to the finite state machine modeling. The entry state is called *Init*, at this stage h_{max} is sent as height reference. When *init done* becomes true, the machine switches to *Hold* state. In this state, the x and y setpoint are calculated as explained in section III-A while the altitude is maintained constant and clamped between h_{min} and h_{max}. Let us define l as half of the platform size (square): if the horizontal error $|\hat{\boldsymbol{e}}_{\boldsymbol{p}}|$ is less than l, a counter c_{track} is increased by one at each iteration. In the same way, if $|\hat{\boldsymbol{e}}_{\boldsymbol{p}}|$ is greater than l then c_{lost} is increased by one at each iteration. As a remark, this is implemented inside the Automatic module which iterates every time a mocap pose value arrives, approximately at 120 Hz.

At this point if c_{track} becomes greater than a constant N_{hold}, the signal *Platform Tracked* becomes true and the machine jumps to *Descending* state. In this condition, height setpoint is decreased by a small delta while x and y are kept constant. Right after this operation the machine is forced back to *hold* state. This continuous switching is made in order to manage tracking and vertical movement as separate tasks; in the same way, if c_{lost} becomes greater than a constant N_{lost} we switch to *Ascending*: c_{track} is set to zero (and vice versa),

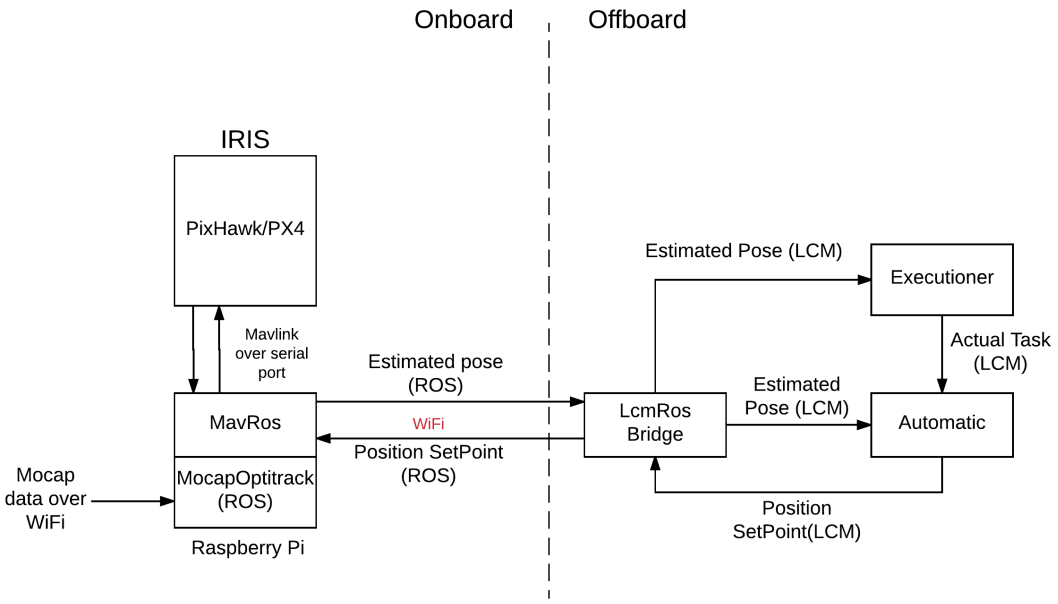


Fig. 7. Software modules

height reference is increased by a delta and state goes back to *Hold*. Both in Hold, Ascending and Descending states, height reference is clamped between h_{min} and h_{max}, they represent respectively: the minimum safest height at which we are sure to avoid collisions, taking into account vertical oscillations, and the maximum altitude for the entire procedure.

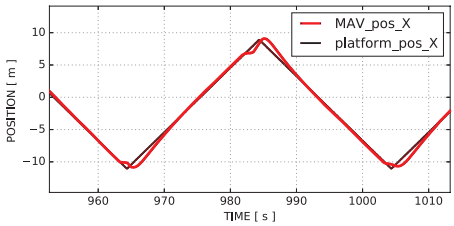
The states modeling is described in figure 5 as well as jumping conditions.

From *Hold* state, *Centered* and *Compensation Ready* signals will become true. The first rises when $|\hat{\boldsymbol{e}}_{\boldsymbol{p}}|$ is less than $l/2$ while the second when the robot hight is around h_{min}. When both are true and the platform is tracked, the machine switches to *Compensation* state. This state is dedicated to height compensation and implements the procedure explained in section III-B beside tracking algorithm.

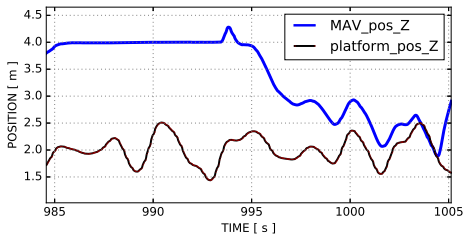
Finally, when *ready* signal rises, meaning that the MAV is 10 cm above landing surface, motors are set to idle performing touchdown. Both *Land* and *Compensate* states can jump to *Ascending*, if the vehicle is not centered, as a failsafe.

D. Simulated environment

When the experimenting phase becomes critical, we need a safer solution than the real robot in order to preserve as much as possible the integrity of the equipment. The Software In The Loop (SITL) simulation is what we need. This kind of environment is able not only to simulate the robot physics, but also to run the actual Firmware. In the simulation, softwares and interfaces are identical to the one explained in figure 7 and the IRIS plus RaspBerry couple is replaced by a physical model inside Gazebo sim.



(a)



(b)

Fig. 8. Simulation results: (a) the x position of the MAV (red) converging to the target moving direction (black); (b) height of the MAV (blue) slowly decreasing until touchdown, while compensating the vertical target movement (black), and taking off again.

VI. Results

In this section we present some trials results reporting the system trajectories and discussing the entire procedure from a performance point of view.

A. Simulation

The preliminary simulation assumes a flat square of dimension 1.2×1.2 meters as target for landing; in order to simulate the floating dynamics, we assume the ocean surface to be modeled with the so called P-M (Pierson-Moskowitz) spectrum [9] and the platform oscillating accordingly.

The following parameters are used in the simulation:

- P-M spectrum frequencies are in the range from [0.5 rad /sec – 4.5 rad /sec].
- Sinusoidal waves frequencies [ω_i with $i = 1, 2 \dots N$] are sampled from that interval with a fixed step.
- For each frequency we generate a random phase and then sum up the N sinusoids in order to obtain the height profile for the oceanic surface.
- Amplitudes are calculated from the P-M spectrum assuming a 7 m/s wind velocity causing wave motion.
- The platform moves back and forth in a straight line along x axis in a segment of 20 meters with a velocity of 1 m/s.
- Final landing phase is performed if the horizontal error respect to the center of the platform is less than 0.5 m.

As previously stated, the procedure is divided in two sub-problems: **horizontal tracking** and **vertical descend**; this simulation was performed in order to test both. As an initial simplifying hypothesis, we assume to have the exact position of the platform at each time step while the on-board position controller works with a simulated GPS as feedback. The MAV autopilot takes position or velocity references processed by the on-board controller [10] coded in the PX4 Firmware.

Figure 8 shows the preliminary tracking performances achieved: in black the target coordinates while in color the MAV coordinates in ENU frame (East-North-UP where x forward and z up). The y plot is omitted since the moving direction is along the x axis. Figure 8(a) shows how the PI regulator converges to the tracked signal, a small overshoot on abrupt platform direction changes appears. On the other hand, figure 8(b) shows the vertical behavior: the MAV compensate the irregular wave by getting closer to the surface until contact, after some seconds it takes off gain. Figure 9 shows snapshots of the entire procedure.

B. Real Experiment

After validating the software in simulation, we performed experiments on the real hardware in order to analyze and compare the system behavior.

We replicated the oscillatory dynamics, for the indoor setup, with a second robot composed by a tracked ground vehicle with a 6 degree of freedom manipulator arm mounted on top of it. On the second link we placed a squared platform of 0.7×0.7 meters as dimensions and a thin foil of foam on it in order to absorb the impact. Figure 10 shows the ground vehicle with the IRIS on top for comparison.

As explained in section VI-A, wave amplitudes are generated through the P-M spectrum according to the procedure explained in [9] and the following parameters are considered:

- P-M spectrum frequencies are in the range from [0.4 rad /sec – 1.3 rad /sec.

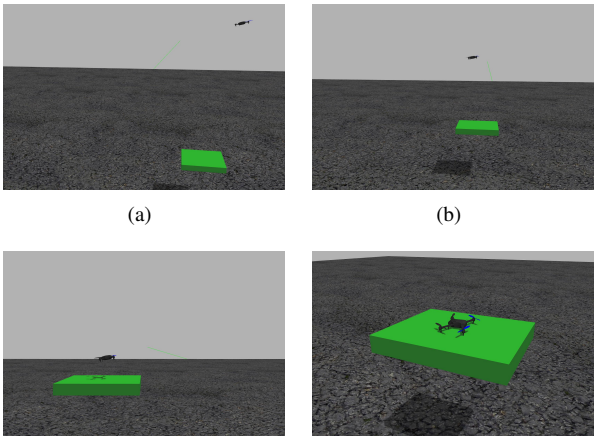


Fig. 9. Landing procedure: (a) tracking, (b) approaching, (c) wave compensation, (d) landing.

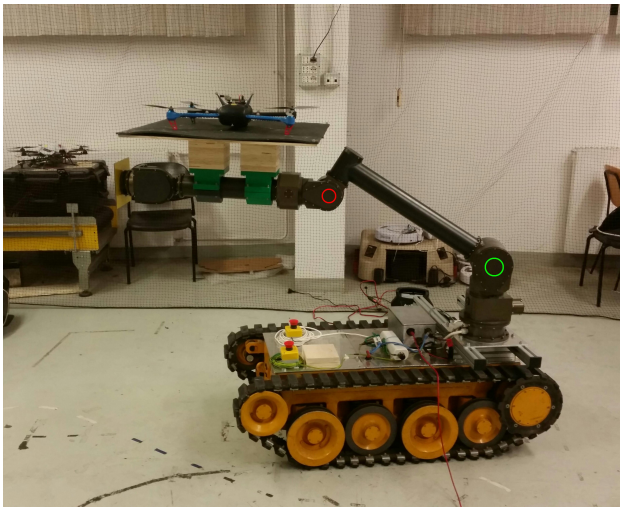
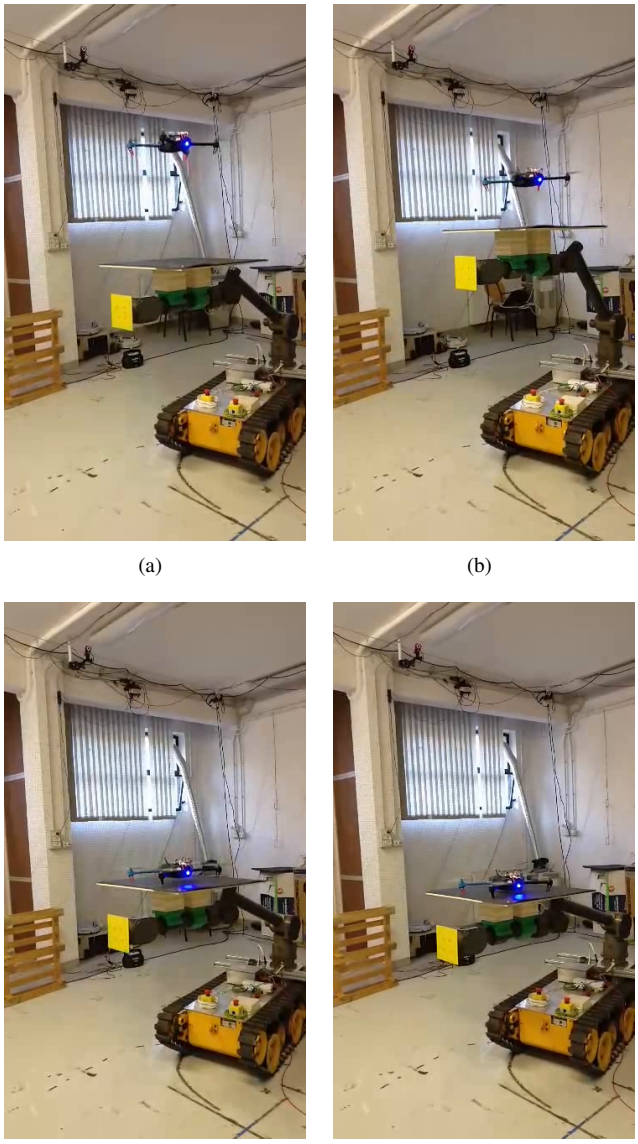


Fig. 10. Ground vehicle with the manipulator placed on top for wave emulation.

- Amplitudes are calculated from the P-M spectrum assuming a 7 m/s wind.
- The wave motion has a maximum amplitude (calculated from the lowest point to the highest) of 0.7 m which correspond to the length of the first arm link.
- PI gains are tuned empirically.

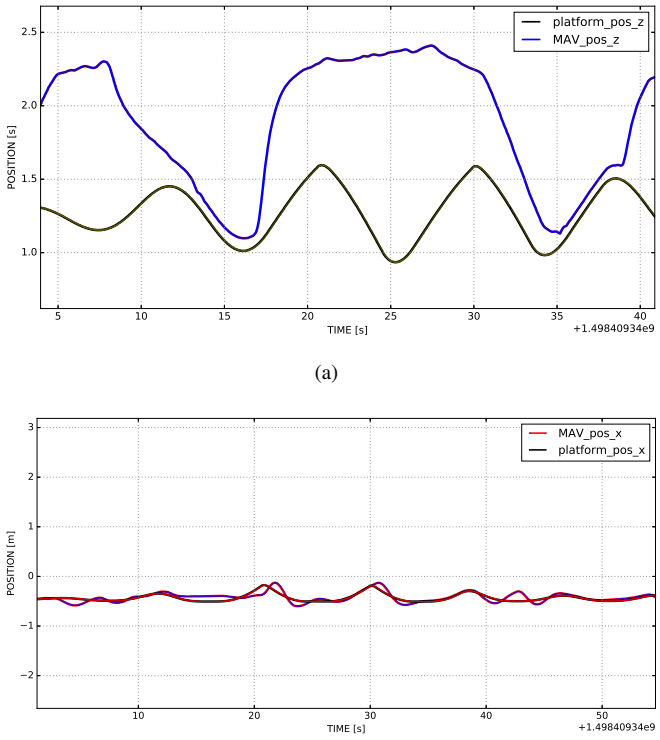
The instantaneous amplitude is generated as a time series and each value is stored in a text file. A program communicates with the arm through a TCP socket sending the right angle references for each joint in order to achieve the desired platform elevation. In particular, from the text file we gather the height value and then we transform it as an angle for the first horizontal pitch arm joint (marked in green in figure 10). Accordingly, we calculate the value for the second pitch joint (marked in red) in order to maintain the platform flat.

The procedure consists in a take off instruction directly with the IRIS placed on the platform and then with a land command; after landing the robot takes off again and the



(a) (b) (c) (d)

Fig. 11. (a)



(a)

(b)

Fig. 12. (a) z coordinate, MAV hight in blue while platform in black; (b) x coordinate, MAV position in red while platform in black;

procedure restarts. Figure 12 shows the evolution of the x and z coordinates both for the robot and platform; the small height difference when landed is due to the quadrotor landing feed which add this offset. Figure 11 shows snapshots of the real experiment.

VII. Conclusions and Future Work

We have considered the problem of autonomous landing on a floating platform, performed by a quadrotor system. We assumed the task of self and relative localization as solved at this stage and we focused our work on the development of the procedure needed in order to tackle the challenge. We prepared a laboratory setup which is easy to use due to the presence of an additional companion computer added to the IRIS robot and the use of standard software tools for communication with the autopilot board.

The developed software framework, namely mocap2mav, offers a simple way to test and implement the different algorithms and modules, as the result of design choices such as: modularity, expandability and universality. Those three pillars are the basis of the entire software development performed in the context of this work giving the opportunity in to integrate more features in the system with few effort in the future.

We proposed a simple but effective method for horizontal tracking by closing the internal controller with an external position loop (with target as reference) including a velocity feed-forward term (with target velocity as reference). On the other hand, descending is performed by forcing the relative vertical velocity to a reference, which is linearly decremented with respect to the distance from the landing surface until touchdown. This method is compatible with the majority of autopilot boards commercially available by simply adjusting the gains for the specific target system.

The entire procedure was modeled with a FSM dividing the problem in different states, this method appeared to be convenient for this task where some unpredictable event may occur. The computational load is very low giving the possibility to run the software on small devices with limited power.

At the end, simulated and real experiments were performed: first on the SITL framework by modeling the wave motion with the P-M Ocean spectrum and after with lab trials on the actual equipment.

Further improvements are: online estimation of the floating

motion and applying a predictive landing maneuver. A vision system would be useful both for P-M spectrum estimate and for providing a more accurate self positioning. Last but not the least, the development of an automatic system identification procedure for the MAV could provide benefits for gain adjustments.

ACKNOWLEDGMENT

This work has been carried out in the hosting laboratory EMAROLab at DIBRIS-Unige in conjunction with the ISME and partially funded by the MIUR/MISE. It has been supported by the Ligurian Technological District SIIT through the MAREA - Monitoring And REscue Automation project.

REFERENCES

[1] W. Kong, D. Zhou, D. Zhang, and J. Zhang, "Vision-based autonomous landing system for unmanned aerial vehicle: A survey," *Processing of 2014 International Conference on Multisensor Fusion and Information Integration for Intelligent Systems*, 2014.

[2] G. Casalino, B. Allotta, G. Antonelli, A. Caiti, G. Conte, G. Indiveri, C. Melchiorri, and E. Simetti, "Isme research trends: Marine robotics for emergencies at sea," in *OCEANS 2016-Shanghai*. IEEE, 2016, pp. 1–5.

[3] E. Simetti, S. Torelli, and A. Sperindè, "Development of Modular USVs for Coastal Zone Monitoring," *Sea Technology*, May 2016.

[4] Mocap2mav software architecture. [Online]. Available: https://github.com/EmaroLab/mocap2mav

[5] A. S. Huang, E. Olson, and D. C. Moore, "LCM: Lightweight Communications and Marshalling," *IEEE/RSJ 2010 International Conference on Intelligent Robots and Systems*, pp. 4057–4062, 2010.

[6] O. Araar, N. Aouf, and I. Vitanov, "Vision Based Autonomous Landing of Multirotor UAV on Moving Platform," pp. 1–16, 2016.

[7] A. Cesetti and E. Frontoni, "Vision-based autonomous navigation and landing of an unmanned aerial vehicle using natural landmarks," *Mediterranean Conference on Control & Automation*, pp. 910–915, 2009.

[8] Mocap2mav software architecture. [Online]. Available: http://px4.io/

[9] Z. H. U. Yinggu, F. A. N. Guoliang, and Y. I. Jianqiang, "Controller design based on T-S fuzzy reasoning and ADRC for a flying boat," *10th IEEE International Conference on Control and Automation*, pp. 1578–1583, 2013.

[10] D. Mellinger, "Minimum Snap Trajectory Generation and Control for Quadrotors," *IEEE International Conference on Robotics and Automation*, pp. 2520–2525, 2011.