Algorithms for controlling and tracking UAVs in indoor scenarios

Andrea Nisticò

Supervised by: Marco Baglietto, Fulvio Mastrogiovanni
Co-supervised by: Tommaso Falchi Delitalia

DIBRIS - Department of Informatics, Bioengineering, Robotics, and Systems Engineering
University of Genova

Degree in *Robotics Engineering*

September 18, 2015

Overview

The quadrotor

The number of applications in which UAVs (Unmanned Aerial Vehicles) are involved is exponentially increasing, especially in indoor scenarios.

Problem statement

Goal

We want to design a set of algorithms and SW components enabling a quadrotor to achieve stability, through position feedback given from a motion capture system, and perform different tasks in an indoor scenario.

The work of this thesis produced:

- A correct integration between the on board autopilot and the mocap system
- A functioning experimental setup mounted in the new EMARO laboratory
- A software architecture capable of letting the robot execute tasks sequentially from a list
- A procedure for landing on a moving platform

IRIS

The IRIS quadcopter is flying robot manufactured by 3DRobotics and commercially available.

Main features

- Relatively cheap
- High quality and robust
- Ready to fly
- Compatible with Android tablets

Main components

- Four 850 KV motors (PWM)
- One 11.1 V LiPo battery
- Control electronics
- Auto pilot software

IRIS

IRIS - PixHawk / PX4

The brain of the robot is the **PixHawk** board.

- Two embedded processors
- IMU, barometer, magnetometer and GPS receiver
- ESCs (Electronic Speed Controllers)
- RC Radio interface
- Telemetry module through MAVLink protocol

Featuring the **PX4** open source autopilot.

PX4 runs on NuttX, an open source RTOS, and includes different modules executing in parallel.

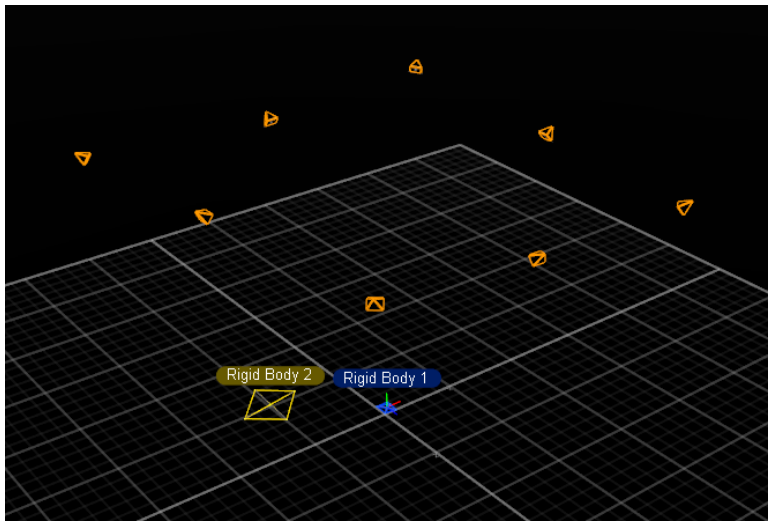
Modules implement: estimation and control, low level management, failsafe.

Motion Capture

Responsible for the localization of the robot.

It relies on eight OptiTrack cameras detecting IR beams reflected by passive markers.

Provides 6D (translation and rotation) position of rigid bodies defined by at least three markers. Position is provided by a local frame predefined by the user.



Adapting reference frames

Mocap reference frame

- X and Z axis on the horizontal plane
- Y axis vertical pointing up

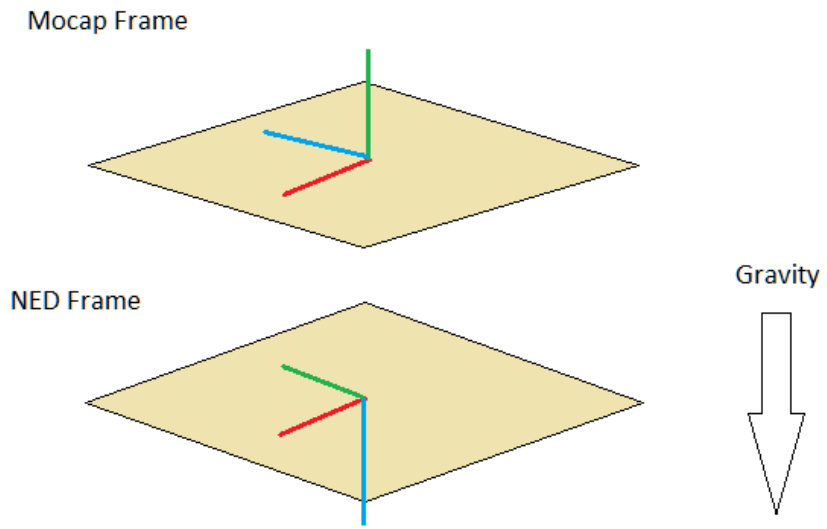
Earth reference frame

- X and Y axis on the horizontal plane
- Z axis vertical pointing down
- When mag is on, X is North, Y East and Z is down

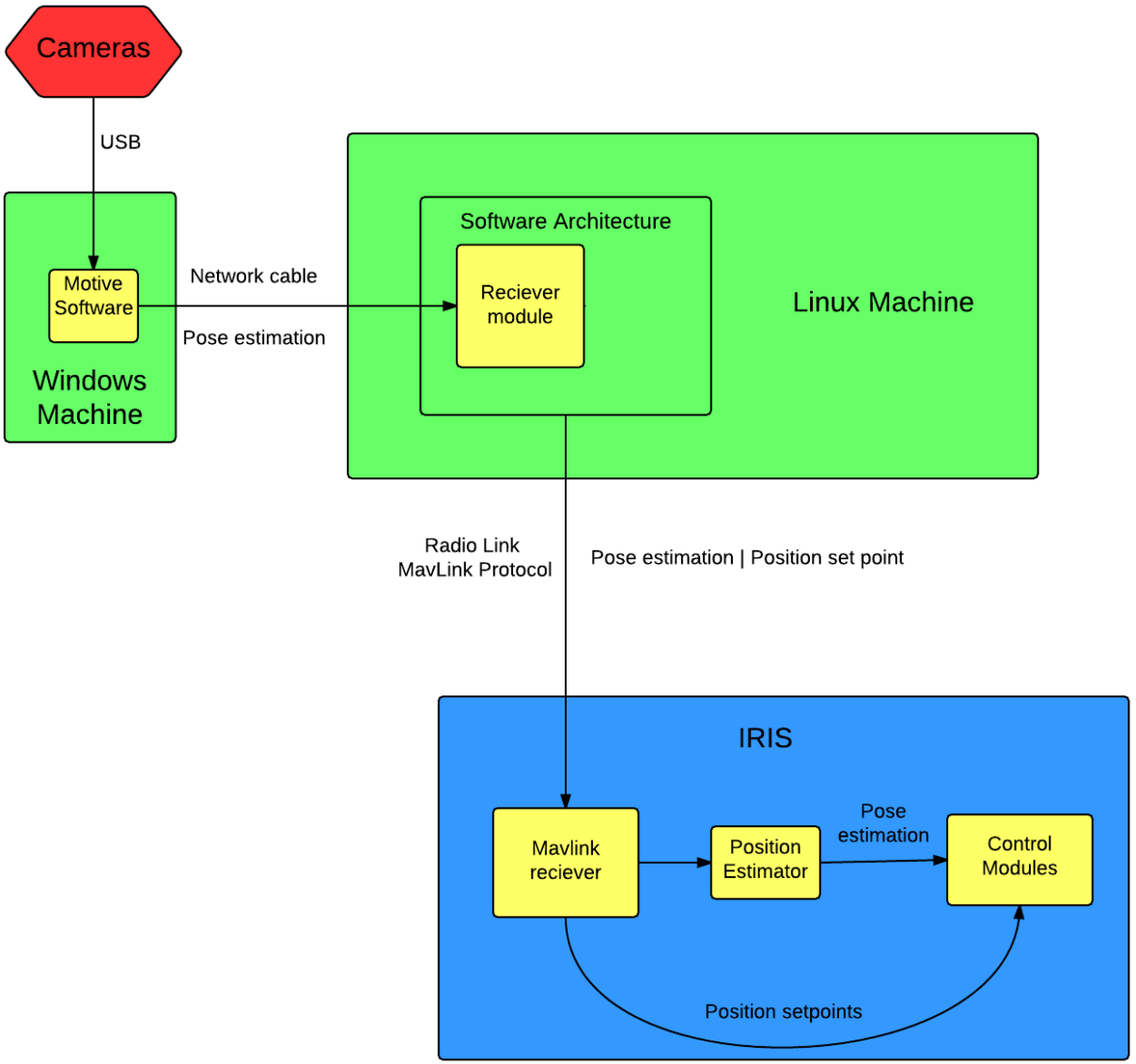
$$P^{\mathrm{E}} = \begin{bmatrix} x & z & -y \end{bmatrix}^T$$

$$Q^{\mathrm{E}} = \begin{bmatrix} w & x & z & -y \end{bmatrix}^T$$

with $[x, y, z]$ and $[w, x, y, z]$ in mocap frame

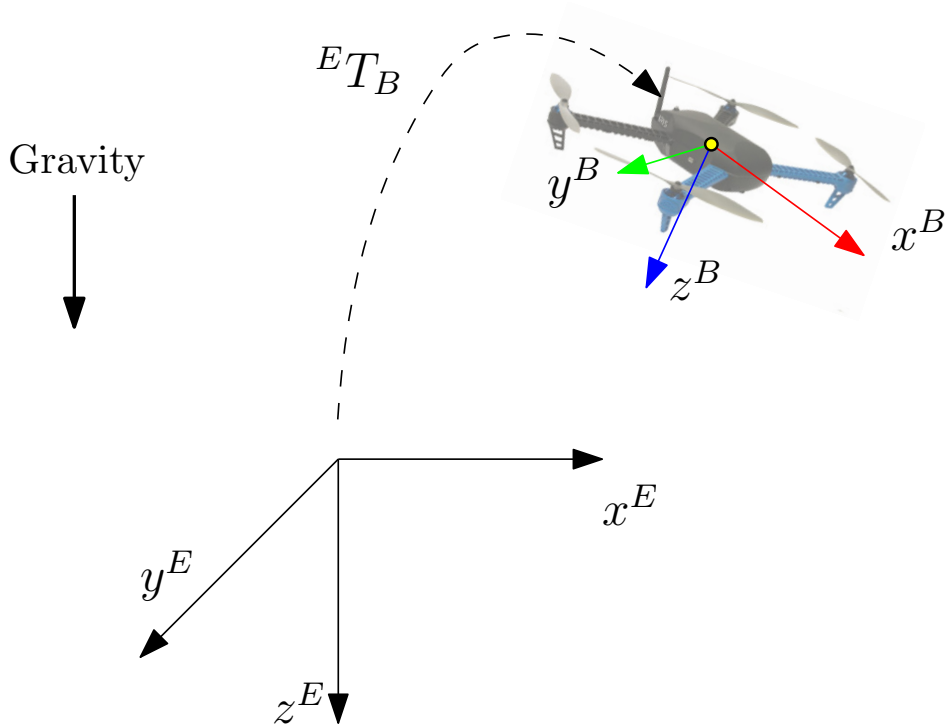


Overall integration



Reference frames

- $\Re^B$ body frame, attached to the robot center of mass
- $\Re^E$ earth frame, placed on the floor at the center of the room



${}^E T_B$ defines the transformation of $\Re^B$ respect to $\Re^E$ as base frame.

Propulsion

Each rotor generates:

- A force $f_i = k\omega_i^2$
- A torque $\tau_i = k_d\omega_i^2$

Performing forces and torques balance:

$$\begin{bmatrix} T^B \\ \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \begin{bmatrix} k & k & k & k \\ kd_iS(30) & -kd_iS(30)k & -d_iS(30) & d_iS(30) \\ -kd_iC(30) & -kd_iC(30)k & d_iC(30) & d_iC(30) \\ k_d & -k_d & -k_d & k_d \end{bmatrix} \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ \omega_4 \end{bmatrix}^2$$

T^B: thrust along z^B

τ : torques on the three axis

Rigid body equations

The model is described by the well known rigid body equations of motion:

$$\begin{aligned} \dot{\mathbf{r}} &= \mathbf{v} \\ m\dot{\mathbf{v}} &= \mathbf{F} \\ \dot{R} &= R\Omega_{\times} \\ I\dot{\mathbf{\Omega}} &= -\mathbf{\Omega} \times I\mathbf{\Omega} + \boldsymbol{\tau} \end{aligned}$$

Define the input vector as : $\mathbf{U} = \begin{bmatrix} T^B & \tau_\phi & \tau_\theta & \tau_\psi \end{bmatrix}^T$

then: $\dot{\mathbf{v}} = \frac{1}{m}\left(\begin{bmatrix} 0 \\ 0 \\ mg \end{bmatrix}^E + R\begin{bmatrix} 0 \\ 0 \\ U_1 \end{bmatrix}^B\right)$ and $\dot{\mathbf{\Omega}} = I^{-1}\left(\begin{bmatrix} U_2 \\ U_2 \\ U_4 \end{bmatrix} - \mathbf{\Omega} \times I\mathbf{\Omega}\right)$

State estimation

Position estimator: Fixed gain observer

- Accelerometers (input)
- GPS
- Barometer
- Mocap

Attitude estimator: Extended Kalman Filter

- Accelerometer
- Gyroscope
- Magnetometer

Position estimator model:

$$\mathbf{x_k}(\mathbf{x_{k-1}}, dt, \mathbf{a}) = \mathbf{x_{k-1}} + \mathbf{v_k} dt + \frac{1}{2}\mathbf{a} dt^2$$

$$\mathbf{v_k}(\mathbf{v_{k-1}}, \mathbf{a}, dt) = \mathbf{v_{k-1}} + \mathbf{a} dt$$

The magnetometer gives the value of the magnetic field in robot frame along three axis. The yaw is normally calculated respect to the North-South line.

Yaw correct estimation

Measured mag. field in body frame:

$$\mathbf{mag}^B = \begin{bmatrix} mag_x \\ mag_y \\ mag_z \end{bmatrix}$$

Fake mag. field in earth frame:

$$\mathbf{mag}^{\mathbf{E}} = \begin{bmatrix} 1 \\ 0 \\ 0.4 \end{bmatrix}^T$$

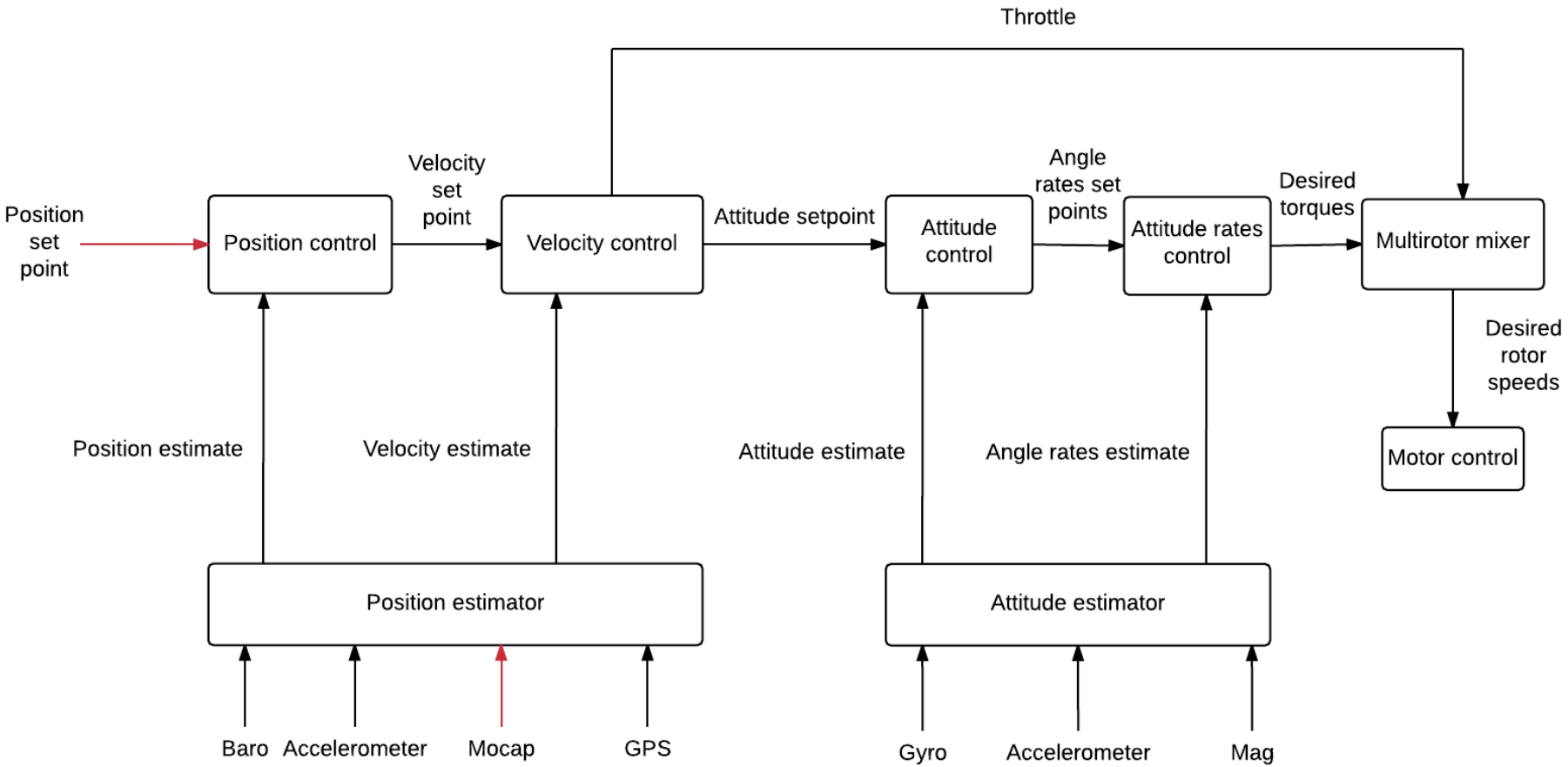
Then we cut of the magnetometer sensor and replace $\mathbf{mag}^B$ with:

$$\mathbf{mag}^{\mathbf{B}} = R^T \mathbf{mag}^{\mathbf{E}} \tag{1}$$

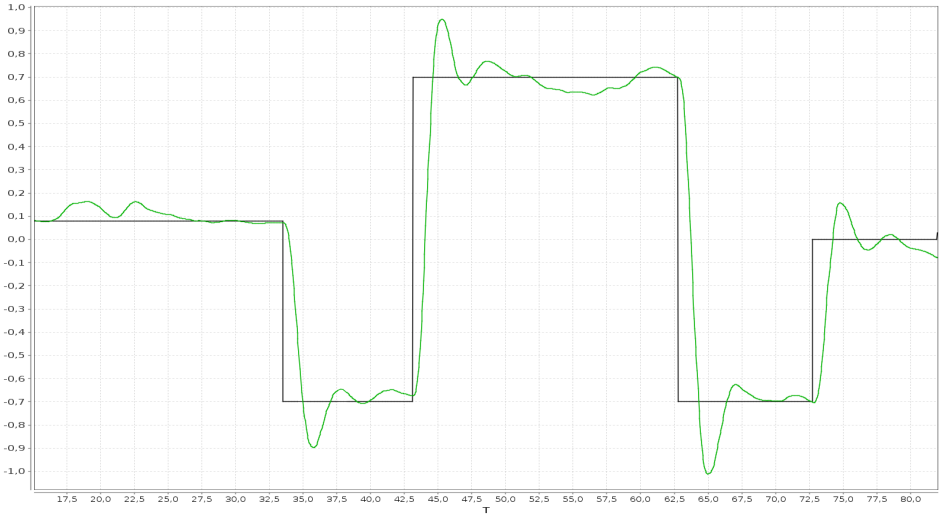
Roll and *Pitch* angles are estimated by the EKF. By passing the value of the *Yaw* calculated by the mocap we are able to construct the R matrix and inject the value of $\mathbf{mag}^{\mathbf{B}}$ inside the estimator.

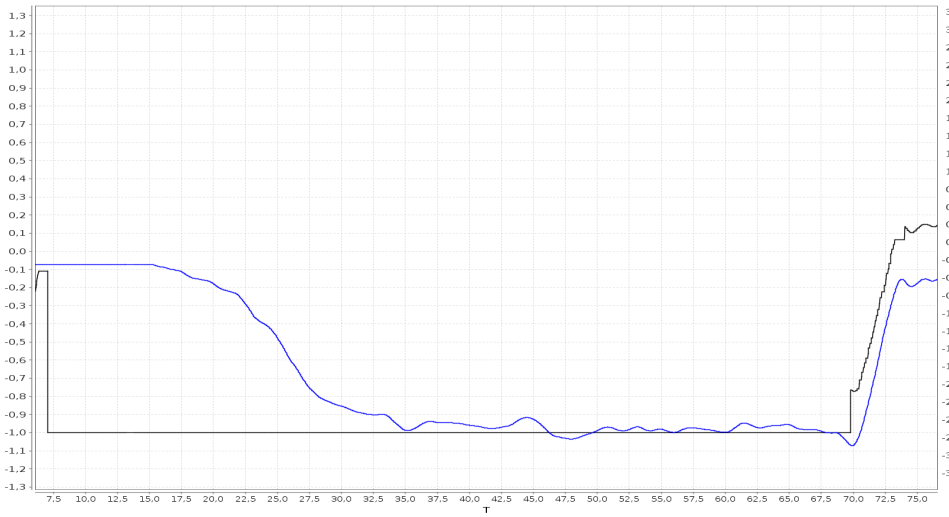
Controller structure

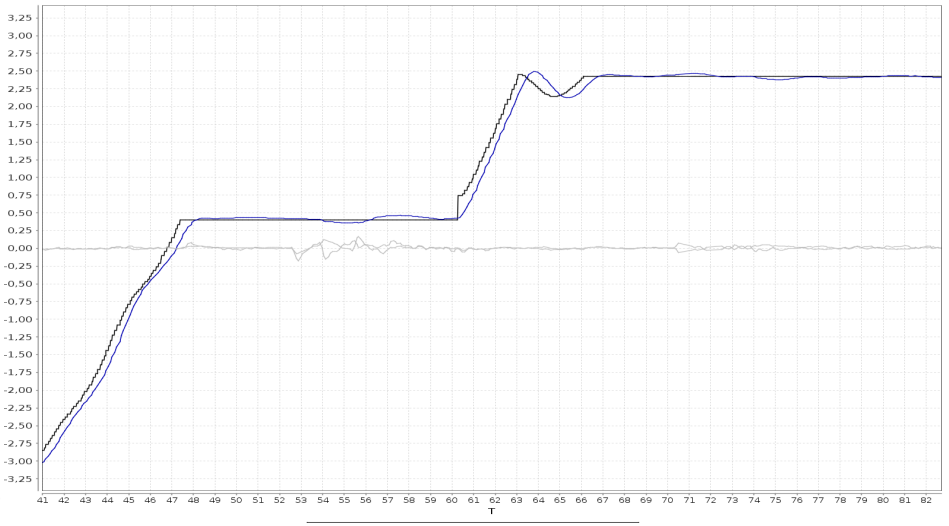
Two modules implement a cascade of P / PID controllers: **position control** module and **attitude control** module.



Results







Definition

Definition

A software architecture is an abstract view of a software system distinct from the details of implementation, algorithms, and data representation.

It is represented by a block diagram. Each block is independent with known in/out relation.

A well written software architecture should:

- Provide flexibility and adaptability
- Allow for interoperability
- Provide control on the system
- Reduce maintenance time and cost
- Help developers improving the software

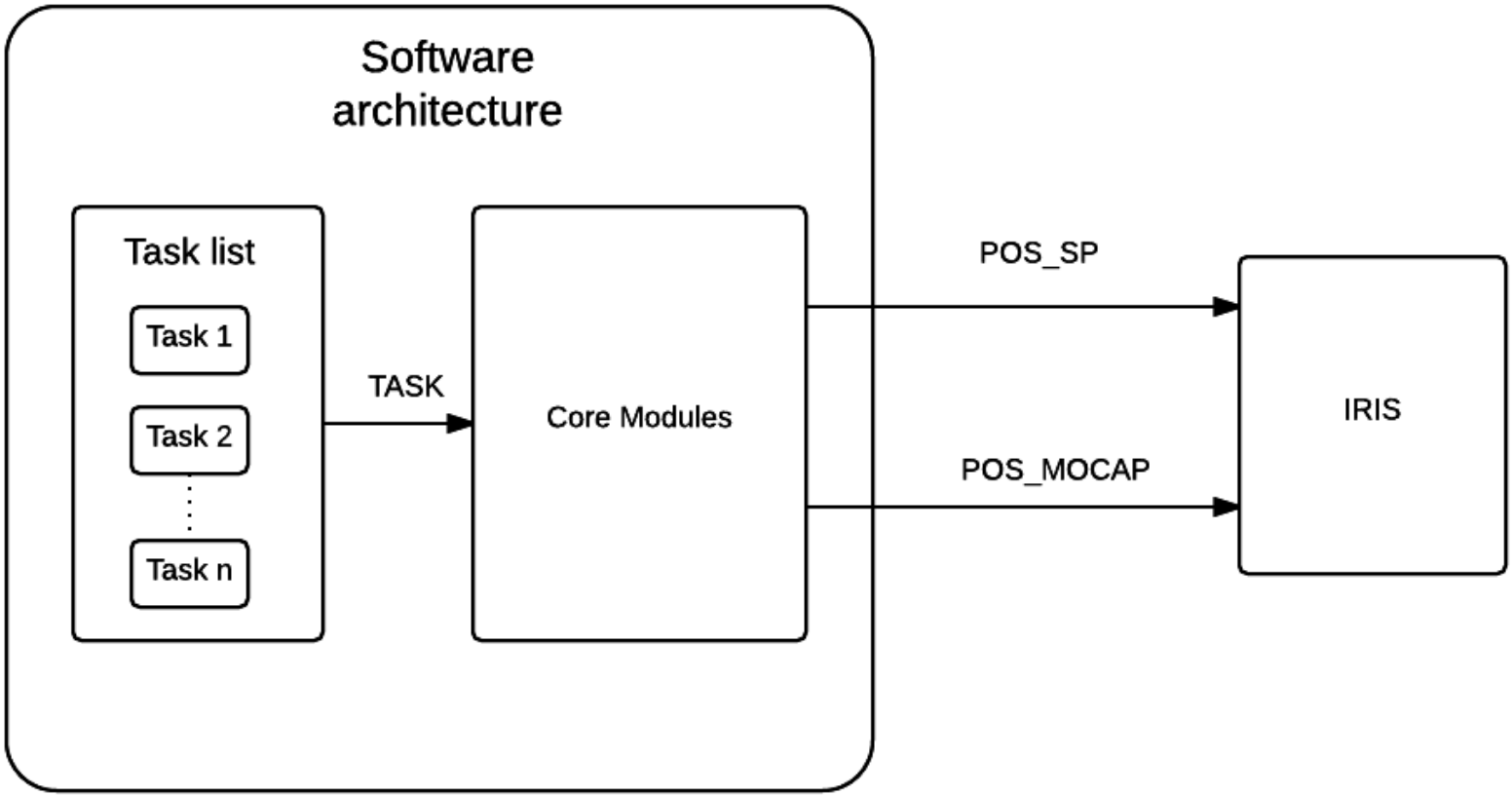
Goal

In particular the SW must:

- Be able to control the robot by sending position feedback and set point
- Execute sequentially a list of tasks predefined by the user

Note: Position feedback and set point are vectors $P = \begin{bmatrix} x \\ y \\ z \\ yaw \end{bmatrix}$

Scheme


Software
architecture
Task list
Task 1
Task 2
Task n
TASK
Core Modules
POS_SP
POS_MOCAP
IRIS

Design Patterns

How do we design the core modules?

Design Pattern

The pattern is a description or template for how to solve a problem that can be used in many different situations.

The Pattern is the architectural style chosen among the available ones depending on the problem.

A **Behavioral design** is used for organizing each SW component.

Behavioral design

Inspired from animals but largely applied in robotics

```
if(TAKE_OFF){ takeOff(); }
```

```
if(LAND){ land(); }
```

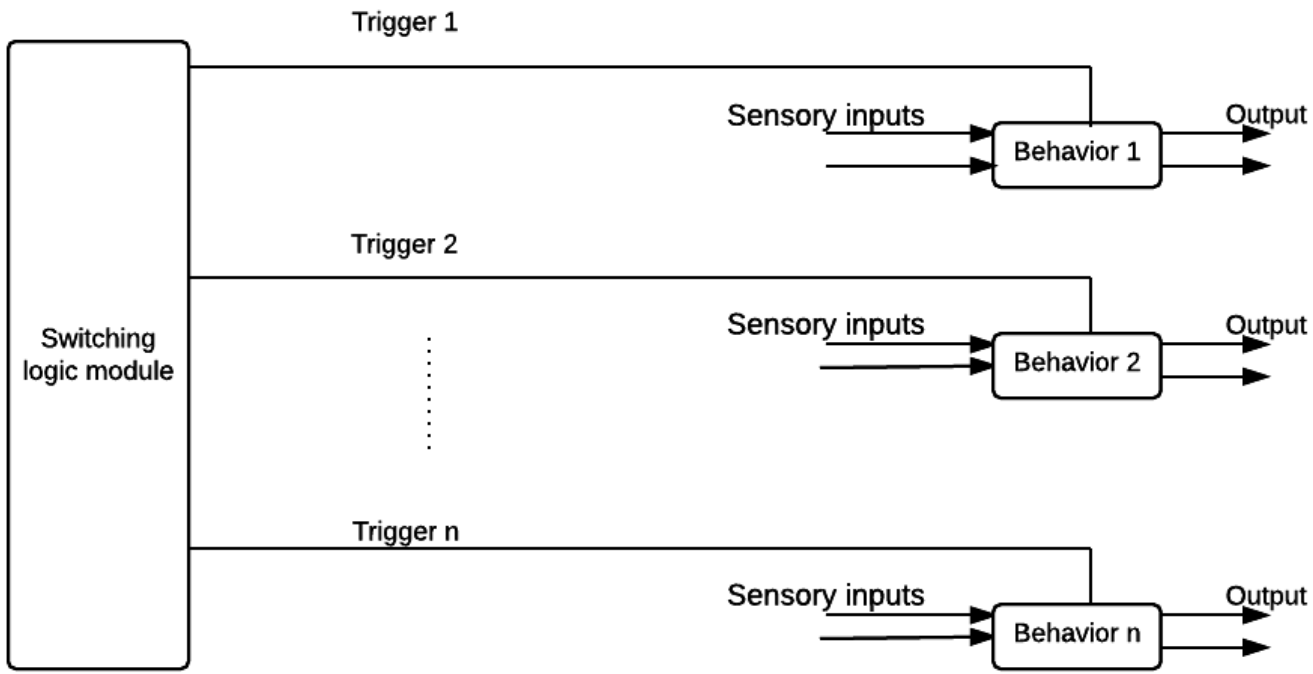
```
if(MOVE){ move(); }
```

```
if(FOLLOW_T){ followTraj();
              rotate(); }
```

TAKE_OFF, *LAND*, *MOVE*, *FOLLOW_T* are boolean signals of the so called **switching logic**.

takeOff(), *land()*, *move()*, *followTraj()* are C++ functions and they are called **behaviors**.

Behaviors and switching logic



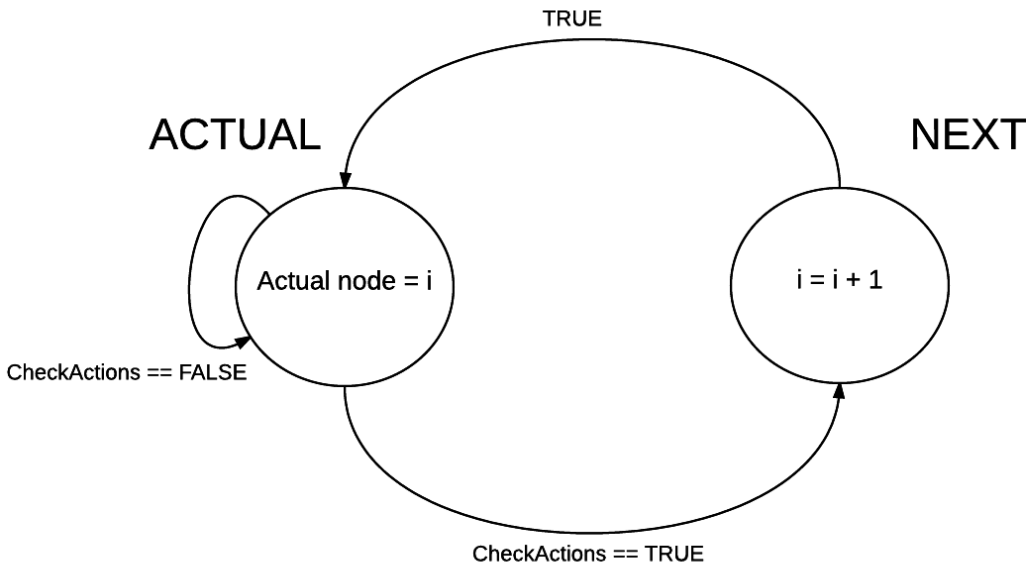
Implemented behaviors

Actual task	Activated behaviors
take off	take_off
land	land
move	move
follow trajectory	follow_traj ; rotate
rotate	rotate
land on a mobile platform	plat_land

Table : Switching logic mapping.

Tasks and switching

- Tasks are executed sequentially.
- Tasks activate different behaviors.
- Behaviors can be concurrent if they act on different dynamics.
- Tasks can be personalized by parameters

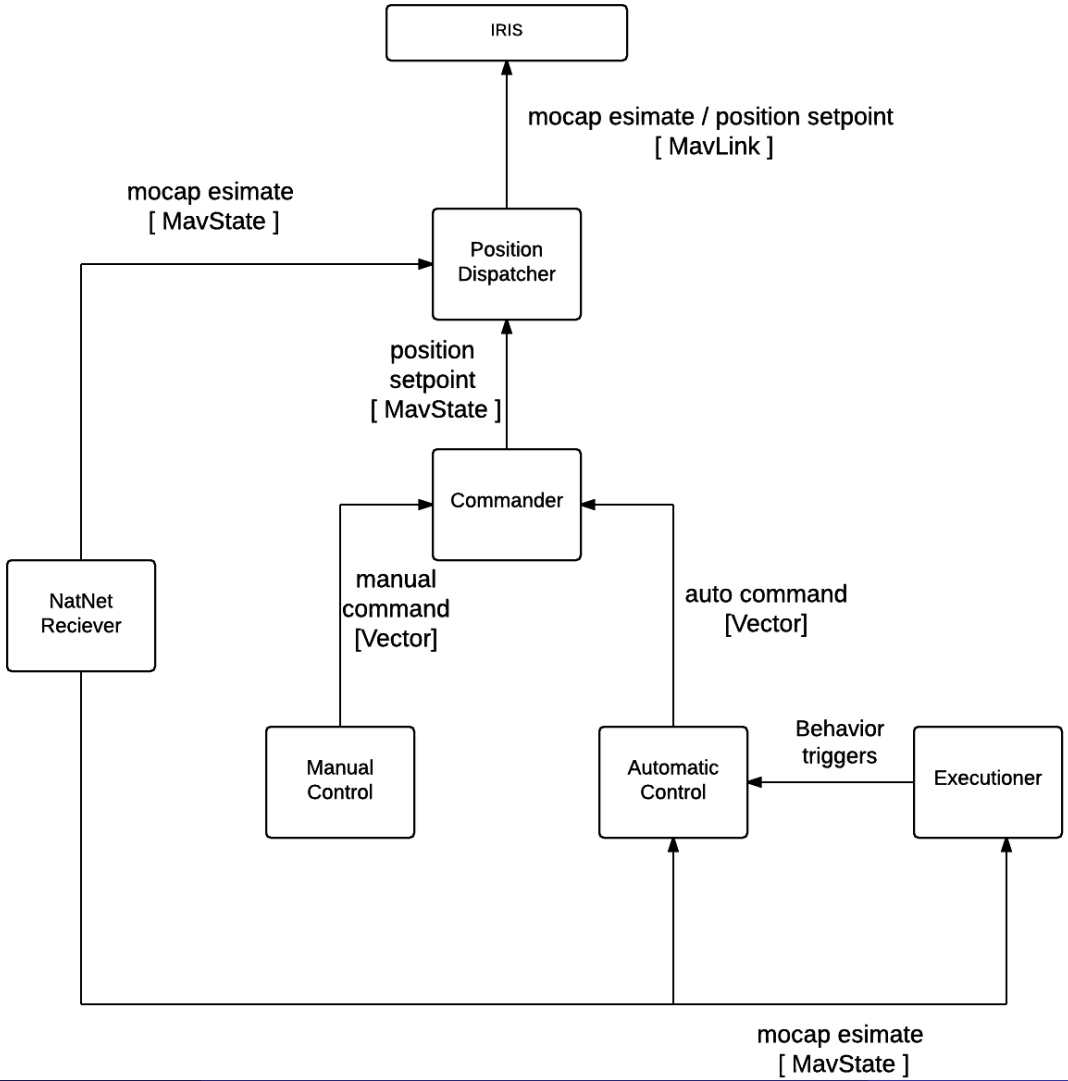


List Example

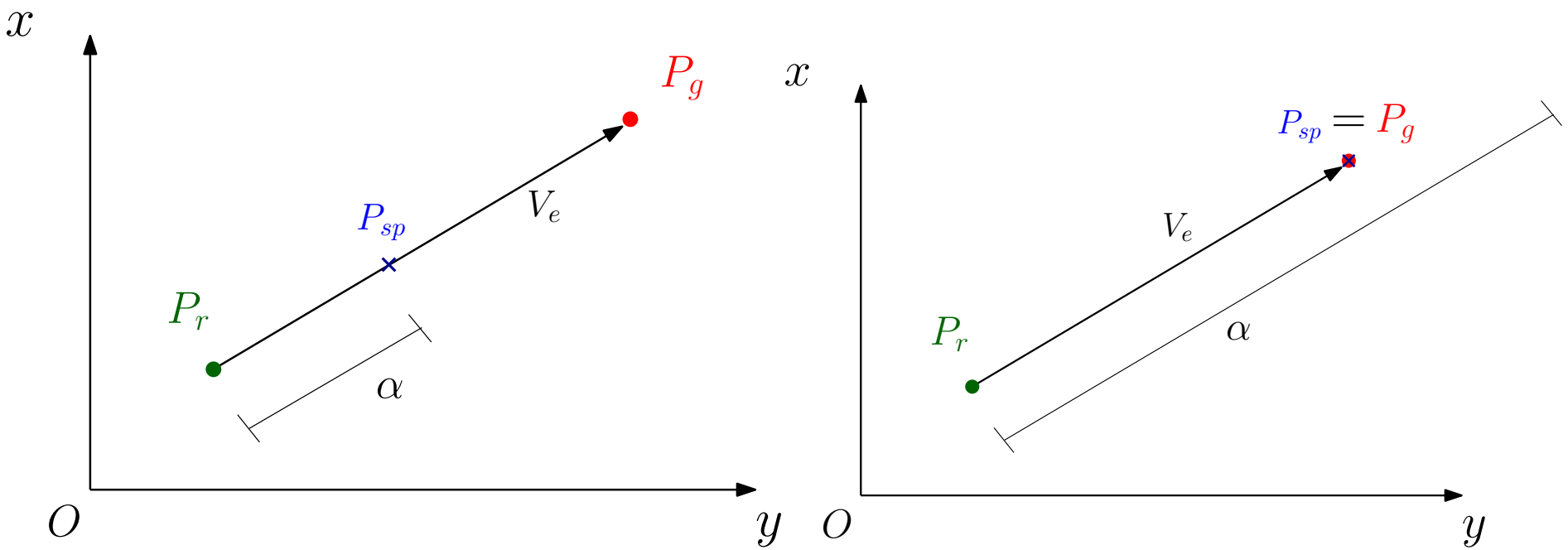
```
// Fill Node list

node takeOff;
takeOff.action.type = 't';
takeOff.action.params[0] = -1; //height
nodeList.push_back(takeOff);

node rotate;
rotate.action.type = 'r';
rotate.action.params[0] = 1; // Angle / directonal vector
rotate.position.yaw = PI;
nodeList.push_back(rotate);

node move;
move.action.type = 'm';
move.position.x = 1.22;
move.position.y = 1.14;
move.position.z = -1.4;
move.position.params[0] = 0.6; //Scaling factor
move.position.params[1] = 3;   // Hovering time on target
nodeList.push_back(move);

node land;
land.action.type = 'l';
land.action.params[0] = 4; //height velocity
land.action.params[1] = 0; // offset
nodeList.push_back(land);
```

Overall structure

Each module runs on a different thread

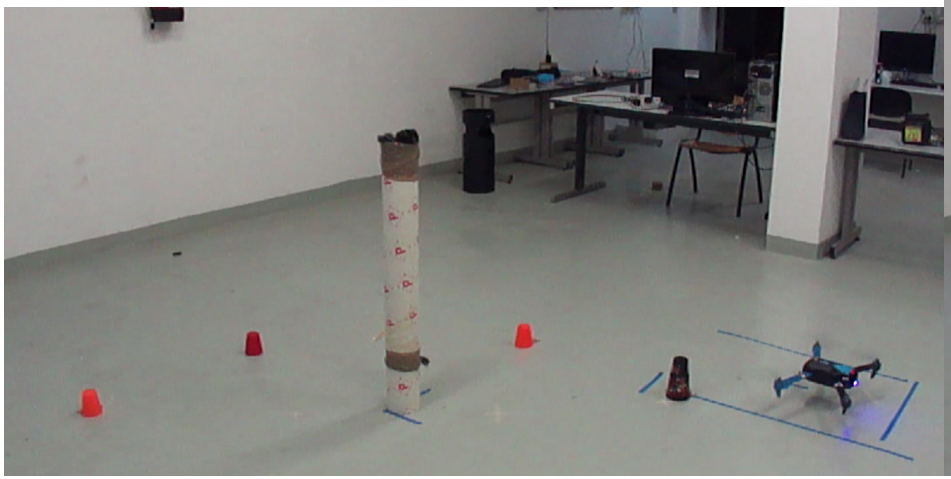


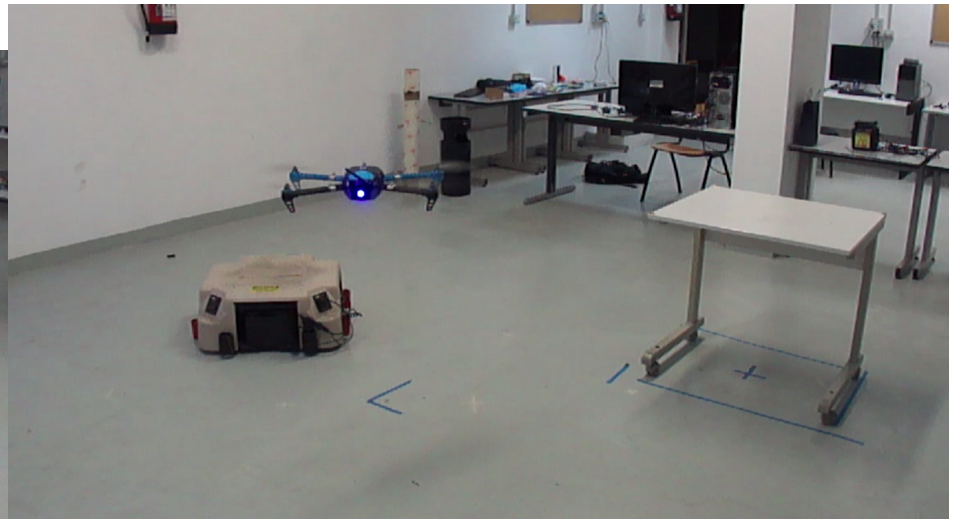
Move algorithm



$$\mathbf{P_{sp}} = \begin{cases} \mathbf{P_g}, & \text{if } \|\mathbf{V_e}\| \leq \alpha \\ \mathbf{P_r} + \alpha \mathbf{u_e}, & \text{if } \|\mathbf{V_e}\| > \alpha \end{cases}$$

Demo





Remember the first video?

Motivations

Manual landing on a mobile platform could be difficult, even with some help. Landing maneuver must be automated.

Applications:

- Cooperative robotics
- Land on moving recharge stations
- Land on air carriers

This is still an open problems with various solutions.

Problem statement

It is possible, through position control, to develop an algorithm which is able to guide IRIS along a landing maneuver on a mobile platform?

Assumptions

- Only the position of the platform is know, velocity is not used in the controller.
- The trajectory of the platform is assumed to be mostly linear, no sudden changes in direction are taken into account.
- The velocity of the platform is constant.
- The extremities of the trajectory are roughly known.

Tackling the problem

We divide the problem in two subproblems, in fact vertical and horizontal dynamics are decoupled and treated separately.

Tracking - horizontal dynamics

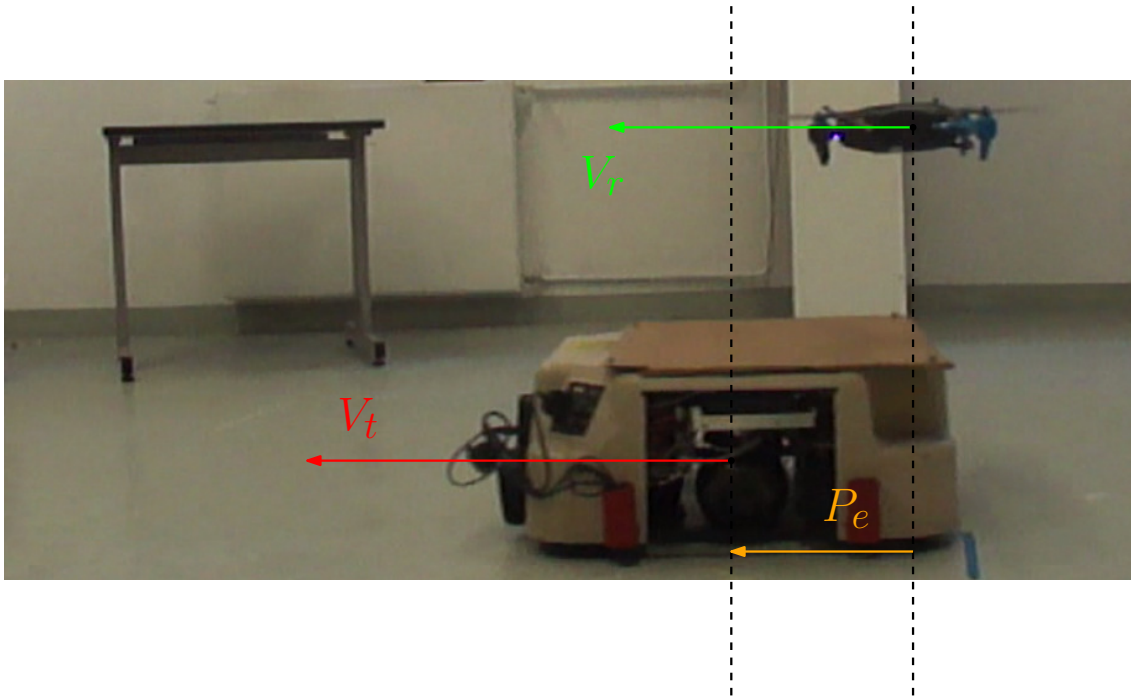
Forcing the IRIS horizontal position to a reference signal (platform position) and minimizing the error between the two. Done by changing x and y set points.

Landing - vertical dynamics

Assuring the descent maneuver to be performed at the proper time in order to finally perform the touchdown. Done by changing altitude set point.

Tracking

Let us first issue the horizontal set point (x , y) equal to the platform position: $\mathbf{P_{sp}} = \mathbf{P_r}$



We observe a static error $\mathbf{P_e} = \mathbf{P_t} - \mathbf{P_r}$

Tracking

We need to place the position setpoint in front of the platform.

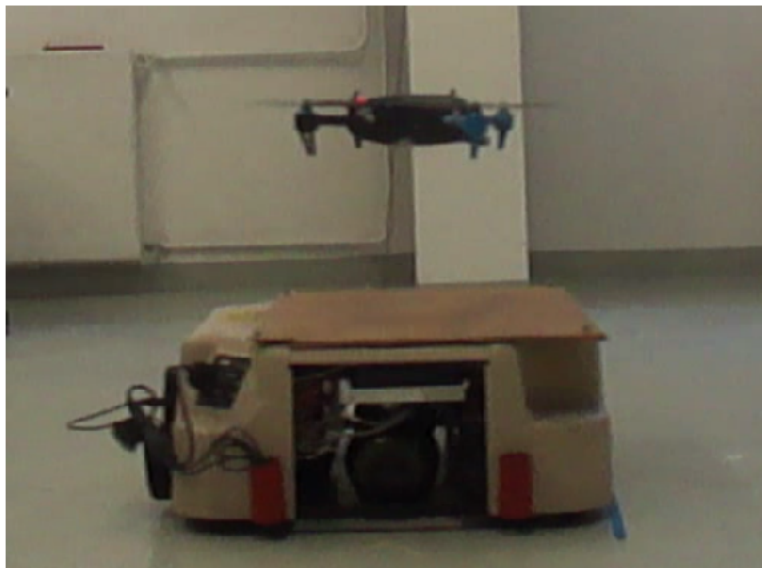
$$\mathbf{P_{sp}} = \mathbf{P_t} + K_p \mathbf{P_e} \tag{2}$$

A big improvement but still wrong, when the error goes to zero the proportional action disappears. Finally:

$$\mathbf{P_{sp}} = \mathbf{P_t} + K_p \mathbf{P_e} + K_i \int \mathbf{P_e} \tag{3}$$

where the integral forces the static error to zero.

Tracking comparison



Landing

Landing is performed in a simple but effective way. The variable that is calculated in this section is the z component of the position set point.

Let $\|\mathbf{P_e}\| = d$

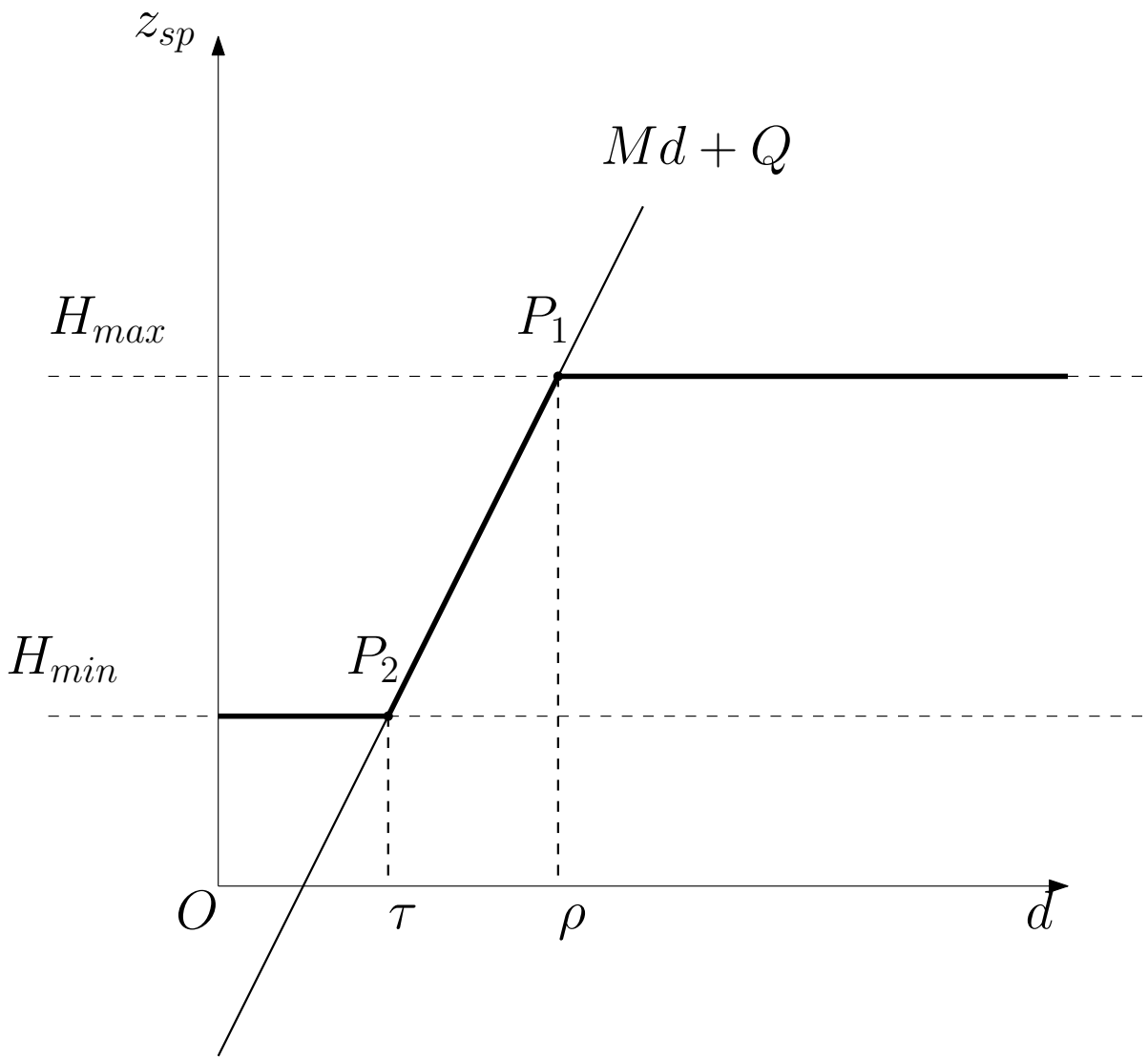
then a linear relation between z_{sp} and d is chosen:

$$z_{sp} = Md + Q \tag{4}$$

Line parameters are calculated following a procedure:

1. Choose the maximum height H_{max}
2. Define a minimum positive height H_{min}
3. Define the radius of influence ρ
4. Define the tolerance τ

Landing profile



Thank you