Research Note
RN/13/08

The QRSim Quadrotors Simulator

March 25, 2013

Renzo De Nardi

Abstract

Examples of quadrotor helicopters models described in the literature (e.g. [6], [4], [17]) tend to focus on reproducing only the dynamic aspects the aerial platform and their primary use is in the domain of closed loop flight control. When the aim is simulating more general higher level tasks that involve multiple platforms which sense and react in their environment, the usefulness of such models is limited.

This document describes the multi-vehicle simulator software QRSim developed to devise and test algorithm that allow a set of UAVs to communicate and cooperate to achieve common goals. In addition to realistically simulate the dynamic of the aerial platform (learned from flight tests), the software simulates the sensors suite that typically equip a UAV (i.e. GPS, IMU, camera) and the endogenous and exogenous sources of inaccuracies that characterize them. Environmental effects that affect directly (e.g. wind) or indirectly (e.g. the plume dispersion) the task are also part of the simulation.

To offer a well structured and challenging set of control and machine learning problems we present a number of non trivial task scenarios and their implementations.

1 Introduction

UAVs provide a rich source of control problems spanning a whole spectrum, from flight attitude control to multi-platform high level collaborative behaviour.

Past research efforts ([12],[22],[6] to list a few) made considerable advancement in the understanding of the flight dynamics and control of a quadrotor platform to the point that such helicopters are now readily available on the market ([25],[15],[14]). Such platforms are generally able to fly autonomously to specified waypoints but still rely on an operator to specify meaningful obstacle free paths. Recent research has shown promising result in the coordination of multiple quadrotor UAVs (e.g.[16],[5],[20]) but more work needs to be done to tackle more useful applications as well as the difficulties posed by unstructured indoor and outdoor environments.

The QRSim software we describe herein is designed to be a useful tool to test and develop algorithms for complex outdoor quadrotor tasks, tasks involving both sensing and control. To this aim the simulator includes realistic models of the helicopter dynamics learned from flight data, probabilistic models of inertial, localization and vision sensors. Similarly, the simulator includes environmental effects such as wind and GPS propagation errors that affect all the platforms in a correlated fashion. The combination of such features makes QRSim substantially different from existing freely available quadrotor simulators.

This research note is divided in four parts, the first two sections (i.e. section 2 and 3) explain respectively the conceptual framework of the simulator and its use, section 4 presents three application scenarios and finally section 5 covers in detail the models that the simulator implements.

A companion to this note is the documentation that Matlab can automatically generate from the source code (i.e. using the command `doc`), the reader is referred to it for all the more specific API details.

2 Concepts

In the following sections we start by presenting the main conceptual blocks that constitute the simulator, we then present the simulator core in section 2.6.

2.1 Platforms and Environment Objects

Within QRSim we make a clear logical distinction between two types of objects:

- platforms,
- environment objects.

To the first type belong the description of the quadrotor dynamics but also the models of sensors and other phenomena that are platform specific (e.g. aerodynamic turbulence). The second class comprehends all the phenomena that are not platform specific and have direct or indirect impact on several platforms (e.g. the flight area or the satellite vehicles of the GPS system). In many ways this distinction is rather natural and in QRSim we are simply exploiting it in order to provide an intuitive structure for to the simulator code and its API.

In general the platforms are the only objects that can be controlled by means of actions, of course those might eventually have an indirect effect on the environment. We must underline that this does not impose for any of the two classes to be time invariant; in fact both types of objects might change during the course of a simulation as result of actions or simply as result of the time passing.

To the extent allowed by Matlab we used object-oriented programming concepts in order to map physical objects directly into the corresponding software classes. Platforms subclass the abstract class `Platform` (defined in `/qrsim/platforms/Platform.m`) and environment objects subclass `EnvironmentObject` (defined in `/qrsim/environment`).

2.2 State

At any point in time the simulator must maintain the state of all the objects taking that are part of the simulation; this is accomplished with a data structure defined by a Matlab class called (rather obviously) `State`.

A handle to an object of this type is returned during the initialisation of the simulator and can be used to reference the various simulation objects. For instance the fields `environment` and `platforms` store respectively the handles to all the environment objects and to the cell array of platforms objects. Given such a structure accessing the state of a specific object is trivial, for instance `state.platforms{1}.getX()` gives access to the state variables of the first platform, while `state.environment.area.getLimits()` returns the size of the flying area.

When meaningful, the `Platform` and `EnvironmentObject` classes define methods to retrieve the object state (e.g. `getX()` in the case of a platform[1]), to reset the state to its initial value defined by the task (see section 3.2) and to set the object state (e.g. `setX(X)` in the case of a platform)[2].

The `State` structure stores also other important variables:

- the simulator time (field `t`), used to ensure synchronization between all the environment and platforms objects;
- the independent pseudo-random number generator streams (field `rStreams`), used by any of the objects that requires any form of random numbers (e.g. to simulate noise samples);
- the simulator time step (field `DT`), the time granularity with which the simulator time is incremented;
- the handle to the 3D graphics visualization (field `display3d`).

[1]In addition to `getX()` platforms define `getEX()` to return the estimated state and `getEXasX()` which returns the estimated state with the same format of `getX()`; we refer to the Matlab API for more details.

[2]In addition to overwriting the object's state these calls also make sure that any other internal variable (e.g. noise model states) is appropriately set. It is worth stressing that simply using `setX(X)` will not allow to produce identical trajectories even when a fix seed for the pseudorandom number generator is specified, this is because the platform dynamics depends also on the environment objects. Refer to section 2.6 for how to produce identical runs.

2.3 Steppable

Since the platforms and environment that we are simulating are representation of physical objects, one simple way to think about their evolution in time is to consider time discretized into steps and to "step forward" the object's state at each time step. In the case of a quadrotor for instance, stepping forward is nothing more than integrating the ODEs that describe its dynamics; for other objects stepping forward might mean instead triggering an event associated with a specific time. In our implementation every object that evolves with time is derived from a common class (i.e. `Steppable`) which defines an abstract `update()` method and a time step property `dt`.

The method `update()` propagates the object's state forward to the current time (field `t` of the `State` data structure), the object's field `dt` specifies with what (time) granularity `update()` should be called.

As we will see in section 2.6, the `update()` method is never called directly, instead the `Steppable` class exposes the method `step()` which in turns only calls `update()` if a time equal to `dt` has passed since the last update[3].

2.4 Task

By combining different types of environment objects and platforms QRSim allows to define a variety of single and multiple helicopter scenarios as well as many different objectives to be accomplished by the platforms.

The abstract class `Task` provides a way to derive task objects that specify both a scenario and an activity to be learned; the class defines four required and one optional methods:

- `init()` allows the user to define the type of each of the environment objects (i.e. the class name) and of each platform and sensor. Along with the object type, a specification of all the class specific parameters is also needed.
- `updateReward(U)` allows for defining an instantaneous reward for the task which is accumulated as the task progresses (e.g. this could be used to include a control or state cost). This method is called by QRSim after a step; its content depends on how the task is defined.
- `reward()` allows for defining the total reward for the task (e.g. the sum of the integrated instantaneous reward plus a final reward). The design of a reward function is again very task specific but most often the user will rely on the `state` data structure in order to compute such a reward.
- `reset()`[4] defines the initial state of the platforms and of any other task specific object.

[3]To simplify the implementation the time step `dt` of any object must be a multiple of the simulator time step (`DT`); in practise this is not a restrictive requirement.

[4]Introduced in version 1.2.0.

- `step(U)`[5] *(optional)* allows for defining a method that handles inputs in a format specific to the task (i.e. not the standard U=$[u_{pt}; u_{rl}; u_{th}; u_{ya}]$), or for triggering any other task specific update.

We will discuss in detail how to create a task in section 3.2.

2.5 Other Abstract Classes

At the aim of making the code easy to extend, the software API defines also several other abstract classes:

- `AerodynamicTurbulence`
- `Sensors`
- `AHARS`
- `OrientationEstimator`
- `Gyroscope`
- `Altimeter`
- `Accelerometer`
- `Camera`.

Such abstract classes are used very much like software interfaces and allow to load at run time the suite of sensors needed for the task (see section 3.2).

2.6 The QRSim Object

After introducing of the fundamental building blocks of QRSim we can look at how they are used within the main class of the simulator called `QRSim`.

The object `QRSim` allows to initialize, setup and control the simulator as a whole; it exposes three main methods:

- `init('task_name')`, initializes the platforms the environment objects and the task according to what specified in the file `'task_name'` (see section 3.2) additionally it also creates the `state` data structure. This is generally the first command called soon after the creation of the `QRSim` object and must be called only once.

- `qrsim.reset()` resets the simulator to the initial state specified by the task. This calls the method `reset()` of the task which takes care of re-initializing the platforms state, it also resets the task reward to zero. This method is generally called after a learning episode in order to restore the state of the simulator. Note however that `reset()` will not reinitialize the random number generator[6].

[5]Introduced in version 1.2.0.

[6]The rationale for this is that in general is useful to generate statistically independent runs with the same initial state as opposed to identical runs.

- `qrsim.resetSeed()` resets the seed of the pseudorandom number generator to the value specified in the task (or to a random value if in the task the seed is set to 0). After calling `resetSeed()` is necessary to call `reset()` in order to reinitialise all the platforms and environment objects. Running repeatedly a task with a fixed seed followed by the two calls `resetSeed()` and `reset()` will lead to identical runs.

- `step(U)`, steps forward in time all the environment objects and all the platforms. This command accepts a matrix of control inputs `U` (i.e. one column array for each platform) and calls the `step()` method for each of the objects. Listing 1 shows the implementation of the method.

  The sequence of operations executed during a step is straightforward, first the simulation time `obj.simState.t` is updated[7], then all the environment objects are propagated and only later each of the platforms is stepped forward. At this point the new state of the platforms can be accessed as described in section 2.2. Line 13 shows how, if any task specific computation of the platforms controls is defined using the task method `step` (see section 2.4), this takes precedence to the controls `U`.

Listing 1: QRSim step() method

```
1  function obj=step(obj,U)
2    for j=1:obj.simState.task.dt/obj.DT,
3      % update time
4      obj.simState.t=obj.simState.t+obj.simState.DT;
5
6      % step all the environment objects
7      envObjs = fieldnames(obj.simState.environment);
8      for i = 1:numel(envObjs)
9        obj.simState.environment.(envObjs{i}).step([]);
10     end
11
12     % see if the task is the one generating the controls
13     UU = obj.simState.task.step(U);
14     if(~isempty(UU))
15       U = UU;
16     end
17
18     % step all the platforms given U
19     for i=1:length(obj.simState.platforms)
20       obj.simState.platforms{i}.step(U(:,i));
21     end
22   end
23
24   % update the task reward
25   obj.simState.task.updateReward(U);
26 end
```

[7]This is the only statement that updates the simulation time.

- `reward()`, returns the instantaneous reward defined by the current task. This is simply a pass through call to the task `reward()` method and is meant to be called after stepping the simulator.

3 Installation and Use

3.1 Installation

The simulator is entirely written in Matlab and does not depend on any additional toolbox; the only requirement is a recent version of Matlab (i.e. version number greater than 7.6; see section 3.1.1 for details on the configurations tested).

To retrieve the software one can simply clone the git[8] source code repository[9] into a directory of choice:

```
$ git clone https://github.com/UCL-CompLACS/qrsim.git
$ cd qrsim
```

Alternatively, the master branch of the software repository is also available as a zip archive from `https://github.com/UCL-CompLACS/qrsim/archive/master.zip`. In this case the archive needs to be downloaded and unpacked into a directory of choice.

Once we have the software in a local directory (let's assume it to be `~/qrsim`), in order to use it is necessary to add the absolute path to the `sim` directory to the current Matlab search path. This can be done using the menu File > Set Path from the Matlab toolbar. Alternatively this can be done from the Matlab console: after navigating to the local simulator directory (`~/qrsim` in our example) one can issue the command[10]

```
>> addpath([pwd,filesep,'sim']);
```

You now have all that is needed to run the basic examples `main.m` and `main10.m`. To do so navigate to the `example` directory (i.e. `~/qrsim/example`) and then simply run the `main.m` script. If `main.m` fails with the message

```
??? Undefined function or variable 'QRSim'
```

the path was not set correctly; use the toolbar menu File > Set Path to confirm that the absolute path of the directory `sim` was added correctly.

At the aim of improving performance, some of the most computationally expensive functions in the simulator have been written also as MEX function, these can be easily compiled using the function `mexify('compile')`[11].

[8] The git version control software is available for Linux (usually from the package manager), OSX (http://code.google.com/p/git-osx-installer/) and Windows (http://code.google.com/p/msysgit/)

[9] Please note that currently the git repository is read only.

[10] Note that this command depends on the location from which it is executed.

[11] Depending on your system setup you might need to configure the MEX compiler using `mex -setup`; please refer to the Matlab documentation for more details.

3.1.1 Tested OS

The software was tested succesfully on the following setups :

- Ubuntu 11.10 with Matlab R2010b and gcc version 4.6.1[12].
- Windows XP SP3 with Matlab 2010a.
- Windows Server 2008 R2 with Matlab 2011a and Visual Studio 2008.
- OSX 10.5 with Matlab 2010a and Xcode gcc version 4.2[13].

3.2 Creating a Task

As a first example of use we look at a single platform task which requires to maintain the quadrotor hovering at the position it has when the task starts; the solution requires continuous adjustment of the controls since the helicopter is affected by time varying wind disturbances. We call this task TaskKeepSpot.

To implement such a task, we start by creating a new class that extends the abstract class `Task` and implements the `init()`, `updateReward(U)` and `reward()` methods.

The `init()` method (see listing 2) returns the `state` data structure presented in section 2.2. The environment and platform objects within the `state` structure exposes the following standard fields:

- `on`, the flag that allows to enable and disable the object in question,
- `dt`, the time step of the object,
- `type`, the class name of the object.

In addition depending on the object type other fields and methods will be available.

The number of sensors and parameters present in a platform makes the process of defining several identical platforms programatically very cumbersome; for this reason all the platform parameters are specified in a single configuration file (see `'pelican_config'`). In this way the same settings can be loaded for more than one platform.

Lastly we see the `taskparams.display3d` parameters which allow specifying if the 3D visualization is active and the size of the window used for its rendering.

The second of the methods that is necessary to define in a task is the method `reset()`. In such method[14] the user specifies that starting state of each platform and also of any other task specific object. This method is called after `init()` but also every time `qrsim` resets; it is therefore possible, for instance, to design the init method to have different randomly generated position of the platform at every task reset.

[12]With gcc 4.6.1 mex compilation warns the user that gcc 4.6.1 is not a supported compiler, in our experience this warning can be safely ignored.

[13]We removed the option `-isysroot` from the `CFLAGS` and `CXXFLAGS` in the file `mexopts.sh`. For newer versions of Xcode see http://www.mathworks.co.uk/support/solutions/en/data/1-FR6LXJ/

[14]Prior to version 1.2.0 the syntax `taskparams.platforms(1).X` was used within the init method to set the platform initial state; such syntax is now meaningless and if used a runtime error will prompt the user to conform to the new API.

Listing 2: TaskKeepSpot init() method

```
function taskparams=init(obj)
 taskparams.dt = 0.02; % task time-step
 taskparams.seed = 0; % if 0 the seed depends on the system time

 %%%% visualization %%%%
 % 3D display parameters
 taskparams.display3d.on = 1;
 taskparams.display3d.width = 1000;
 taskparams.display3d.height = 600;

 %%%% environment %%%%
 % limits need to follow the conventions of axis(), in m, Z down
 taskparams.environment.area.limits = [-10 20 -10 10 -20 0];
 taskparams.environment.area.type = 'BoxArea';

 % utm origin depends on gpsspacesegment.orbitfile
 [E N zone h] = lla2utm([51.71190;-0.21052;0]);
 taskparams.environment.area.originutmcoords.E = E;
 taskparams.environment.area.originutmcoords.N = N;
 taskparams.environment.area.originutmcoords.h = h;
 taskparams.environment.area.originutmcoords.zone = zone;
 taskparams.environment.area.graphics.type = 'AreaGraphics';

 % GPS - the space segment of the gps system
 taskparams.environment.gpsspacesegment.on = 1; % noiseless if 0
 taskparams.environment.gpsspacesegment.dt = 0.2;
 taskparams.environment.gpsspacesegment.orbitfile=...
       'ngs15992_16to17.sp3'; % real satellite orbits from NASA JPL
 % simulation start in GPS time, needs to agree with file above
 taskparams.environment.gpsspacesegment.tStart = 0;
 % id number of visible satellites, matches the contents of orbitfile
 taskparams.environment.gpsspacesegment.svs=[3,5,7,13,16,19,20,22,29];
 taskparams.environment.gpsspacesegment.type = 'GPSSpaceSegmentGM2';
 taskparams.environment.gpsspacesegment.PR_BETA2 = 4; % t const
 taskparams.environment.gpsspacesegment.PR_BETA1 = 1.005; % t const
 taskparams.environment.gpsspacesegment.PR_SIGMA = 0.003; % sd

 % Wind - a steady homogeneous wind common to all helicopters
 taskparams.environment.wind.on = 0;
 taskparams.environment.wind.type = 'WindConstMean';
 askparams.environment.wind.direction=degsToRads(45); %mean direction
 taskparams.environment.wind.W6 = 0.5; % vel at 6m from ground in m/s

 %%%% platforms %%%%
 % Configuration for each of the platforms
 taskparams.platforms(1).configfile = 'pelican_config';
end
```

Listing 3: TaskKeepSpot reset() method

```
function reset(obj)
  % defines the platform initial state
  obj.simState.platforms{1}.setX([0;0;-10;0;0;0]);
end
```

A task must also define the `updateReward(U)` and `reward()` functions. For TaskKeepSpot a straightforward update reward can be defined as a quadratic cost of the control inputs (see listing 4) in order to penalize large controls. An appropriate total reward can be defined adding the current reward to a final reward computed as the square of the distance to the quadrotor initial position (see listing 5). Such a function gives higher rewards to a policy that at the end of the task gets the quadrotor close to the starting point.

The complete listing for this task can be found in the file `TaskKeepSpotWithReward.m`.

Listing 4: TaskKeepSpotWithReward updateReward(U) method

```
function updateReward(obj,U)
  % updates the reward based on control costs
  for i=1:size(U,2)
    u = (U(:,i)-obj.U_NEUTRAL);
    obj.currentReward = obj.currentReward ...
                              - ((obj.R*u)'*(obj.R*u))*obj.dt;
  end
end
```

Listing 5: TaskKeepSpot reward() method

```
function r=reward(obj)
  % returns the total reward for this task
  if(obj.simState.platforms{1}.isValid())
    e = obj.simState.platforms{1}.getX(1:12);
    e = e(1:3)-obj.initialX(1:3);
    % control cost so far plus end cost
    r = obj.currentReward - e' * e;
  else
    r = - obj.PENALTY;
  end
end
```

3.3 Example Main

To clarify the use of the simulator by means of the `QRSim` object, we shall present a simple example of use; namely we will use a manually designed PID controller to achieve the

Listing 6: main script

```

% create simulator object
qrsim = QRSim();

% load task parameters and return hanle to simulator state
state = qrsim.init('TaskKeepSpot');

% create a PID object
pid = WaypointPID(state.DT);

% number of steps we run the simulation for
N = 3000;

% initial position of the helicopter
wp = [state.platforms{1}.getX(1:3)',0];

tstart = tic;

for i=1:N,
    tloop=tic;

    % compute controls based on the ESTIMATED state
    U = pid.computeU(state.platforms{1}.getEX(),wp);

    % step simulator
    qrsim.step(U);

    % wait so to run in real time
    wait = max(0,state.task.dt-toc(tloop));
    pause(wait);
end

% get reward
% r = qrsim.reward();

elapsed = toc(tstart);
```

required behaviour[15]. Listing 6 shows the code of this bare bone example.

Before anything else we are required to instantiate the `QRSim` object, this will make sure that all the functions and classes of the simulator are reachable. Next we load all the task parameters by calling `qrsim.init()`, such function returns a handle to the simulator state.

[15]We want to clarify that the PID controller was chosen exclusively for its simplicity, in facts it is a very conservative controller and its performance are suboptimal. This said it might still be useful as a baseline for comparisons.

The main for loop (see line 19) drives the simulation forward by calling in sequence the PID controller and the `qrsim.step(U)` method. Given the current state of the platform, the PID computes the new control action `U` which is then used to step the dynamics forward in time.

A quadrotor requires four control inputs U=$[u_{pt}; u_{rl}; u_{th}; u_{ya}]$ respectively pitch roll throttle and yaw. In hover condition pitch and roll are zero, the throttle is the amount required to counterbalance the gravity force[16] while yaw controls the direction in which the helicopter is pointing. Any pitch or roll rotation away from the level attitude will produce a corresponding translational acceleration. To move forward is sufficient a small amount of pitch (negative pitch to be precise), and a roll motion allows to move left and right.

The PID controller computes the required pitch, roll, yaw and throttle commands proportionally to the current distance between the helicopter and the selected waypoint.

Since we are using a PID controller and no learning is involved, after running the predefined number of iterations we do not make use of the reward returned by the task. In the listing (line 34) we show the call commented out to give an idea of when normally the total task reward would be retrieved.

The task defines to run the simulation with the 3D visualization so it makes sense to run it in real time; line 30 introduces the necessary pause.

Figure 3.3 shows a snapshot from the 3D visualization of the simulator where both the platform and its past trajectory are visible.

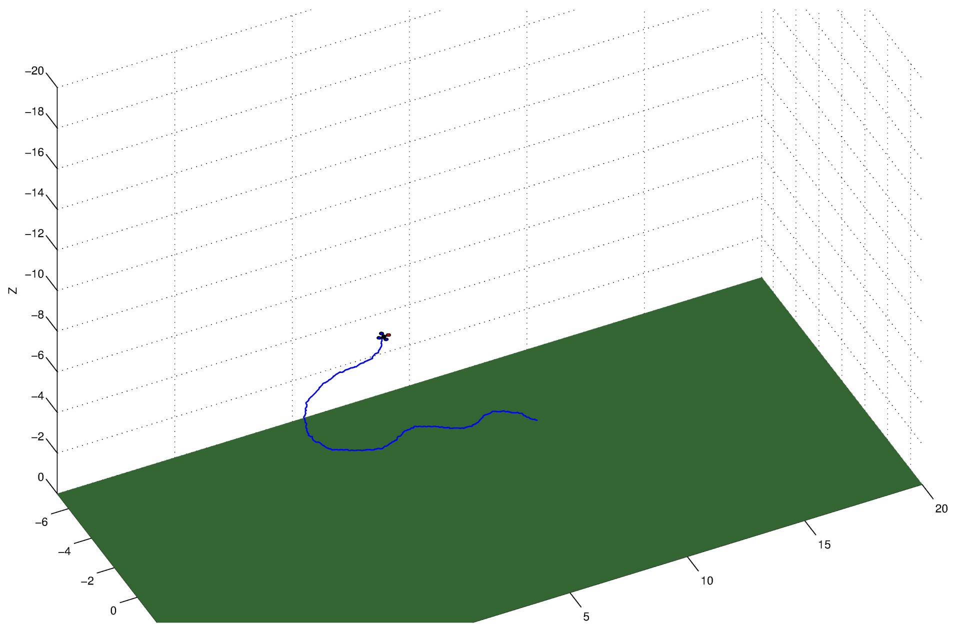


Figure 1: 3D visualization

There is obviously no need to define a specific main script like the one presented above, in certain application for instance it might make more sense to call the `QRSim`

[16]With our dynamic model and a mass of $1.68Kg$ the required throttle control is 0.59

directly from a learning algorithm; this is perfectly fine as long as the calls are executed in the same sequence shown in listing 6.

An additional example called `main10.m` is also present in the example directory and shows how to generalize the `TaskKeepSpot` to a group of ten helicopters.

3.4 Controllers

The directory `/controllers` contains baseline implementations of simple controllers[17] that are often useful in a variety of tasks. Such controllers can be used to compute the set of control inputs `U` given the current platform state and a waypoint or a velocity that the UAV is meant to reach.

Currently the following controllers are available:

- `WaypointPID` Controller that computes the pitch, roll and throttle input necessary to achieve a desired 3D position (NED) while maintaining a specified heading.
- `VelocityPID` Controller that computes the pitch, roll and throttle input necessary to maintain a desired velocity (NED) while keeping a specified heading.
- `VelocityHeightPID` Controller that computes the pitch and roll input necessary to achieve a desired 2D (NE) velocity while maintaining a specified altitude and heading.
- `AnglesHeightPID` Simple controller that computes the throttle and yaw input necessary to maintain a specified height and heading. The roll and pitch angle controls are left to the user.

To use one or more of such controllers simply add the directory `controllers` to the Matlab path.

3.5 Multiple Simulations

Since the implementation of the simulator is completely object oriented, it is possible (and sometimes very useful) to instantiate multiple and completely independent simulations within the same Matlab workspace.

The script `/example/multisim.m` shows an example in which one simulator is used to sample control actions the best of which (in terms of reward) is then carried out on the second simulator[18].

3.6 TCP Interface (Linux)

We also provide a TCP interface to the quadrotor simulator QRSim so that its functionalities can by easily accessed by a second software. The interface is constituted of two parts: a Matlab (Java) server that is responsible for listening to client connection and

[17]We believe these controllers to be qualitatively similar to the ones implemented in the Pelican helicopter, however at present no quantitative comparison has been carried out.

[18]This replicates what one might do in reality where the simulation is used to evaluate control action that are then carried out on a real platform.

managing the QRSim simulation and a client library that allows a client to communicate with such server. Currently the client library is written in C++ for Linux but very similar libraries could be developed easily for other languages[19].

3.6.1 Client

The client library exposes a series of convenient methods to interact with the Matlab simulator QRSim. We present them briefly below and refer to the header file `QRSimTCPClient.h` and to the QRSim API documentation for more details about the meaning of the parameters.

- `bool connectTo(string ip, int port)`
  Connects to the Matlab server of QRSim listening at the specified ip address and port.

- `bool init(string task,vector<vector<double>>& X,vector<vector<double>> & eX,double& tStep,int& nUAVs,bool realTime)`
  Send to the server the command to initialize the simulator by loading the specified task file; the initial state is returned;

- `bool reset()`
  Sends a reset command to the server, this will cause the simulator to set itself to the initial state defined in the task.

- `bool disconnect()`
  Disconnects from the server without turning off the simulator.

- `bool quit()`
  Disconnects from the server and turns off the simulator.

- `bool setState(const vector<vector<double>>& X)`
  Send to the server the command to set the UAV noiseless states to the values provided. The state must be within the limits specified in the task.

- `bool stepWP(double dt,const vector<vector<double>>& WP, vector<vector<double>>& X,vector<vector<double>>& eX)`
  Step forward the simulator giving to each of the UAVs the specified waypoint as input. The simulator uses a `WaypointPID` controller to reach the waypoint, once the waypoint is achieved the UAV will remain stationary at the waypoint.

- `bool stepVel(double dt, const vector<vector<double>>& vel, vector<vector<double>>& X,vector<vector<double>>& eX)`
  Steps forward the simulator giving to each of the UAVs the specified velocity commands as input. The simulator uses a `VelocityPID` in order to achieve and maintain the velocity command specified.

[19]The data serialization is based on the Google protocol buffers library, so writing a client for any of the languages supported by protocol buffer is pretty straightforward.

- `bool stepCtrl(double dt, const vector<vector<double>>& ctrl,
              vector<vector<double>>& X,vector<vector<double>>& eX)`
  Steps forward the simulator giving to each of the UAVs the specified control commands as input. The input is expressed in terms of angles and throttle commands so it is directly passed to the platforms.

3.6.2 Server

The server code[20] is constituted by the Matlab function `QRSimTCPServer.m` which executes the incoming commands in the quadrotor simulator and the Java class `QRSimTCPServer.java` which manages the TCP connection and the data serialization.

In general a user of the client library does not need to modify the server code, but this needs to be compiled[21] and installed (see section 3.6.3 and 3.6.3) on the machine that will run the quadrotor simulation.

3.6.3 Compilation and Installation

The current version of the TCP interface only supports Linux[22], in the following we assume to be working on one of such systems.

Dependencies:

The C++ client uses Google protocol buffers[23] for data serialization and CMake[24] to manage the build process, so both of these tools needs to be available in your Linux machine. On Ubuntu 12.04 these can be installed as follows[25]:

```
$ sudo apt-get install cmake
$ sudo apt-get install libprotoc-dev libprotobuf7 libprotobuf-lite7
$ sudo apt-get install libprotobuf-java
```

Compilation:

To compile the server and client code cd into `qrsim/tcp-linux/build`[26] and run:

```
$ cmake ..
$ make
```

all should compile cleanly.

CMake is configured to compile and install only the C++ client if Java is not installed in your machine. Note that this is a legitimate scenario since you might be interested in connecting to a remote server which takes care of running QRSim.

[20]Under the directory `qrsim/tcp-linux/`.

[21]Google protocol buffer serialization has strong dependencies on the specific version of the protocol buffer library so shipping a pre-compiled server library is not a viable option.

[22]Currently we only tested the code on Ubuntu 10.04LTS and Ubuntu 12.04LTS, but (assuming that all the dependencies are satisfied) the code should compile in any other recent Linux distribution.

[23]https://developers.google.com/protocol-buffers/

[24]http://www.cmake.org/

[25]On a different distro the protocol buffer library might have a different name.

[26]If the directory `build` does not exist, create it.

Installation:

Installing[27] the library is nothing different from any usual Linux library compiled from source:

```
$ sudo make install
```

3.6.4 Using the Client Library in your C++ code

Now that the client library is compiled and installed in your system, using it in your custom C++ code is straightforward. As you commonly do with other shared library installed in your system:

- include the file `QRSimTCPClient.h` in you source code to have access to the library definitions and function prototypes
- link against libqrsim `libqrsimtcpclient.so`.

Two small examples of using the client library are provided with the files `qrsim/tcp-linux /src/exampleclient.cpp` and `qrsim/tcp-linux/src/testclient.cpp`.

3.6.5 Running the Server

The server code runs in Matlab; reach the directory `qrsim/tcp-linux/matlab` from the Matlab console and run:

```
>> QRSimTCPServer(10000)
```

where `10000` is the number of the TCP port you want to use. As the program starts you should see the following message:

```
>> QRSimTCPServer(10000)
Waiting for client to connect to this host on port : 10000
```

indicating that the server is running correctly and is awaiting for connecting from the client.

3.6.6 Testing the Interface

We provide a simple binary to test the TCP interface, after the compilation this should be present in the directory `qrsim/tcp-linux/bin`. Once the server is up and running you can run the client as follows:

```
$ testclient 127.0.0.1 10000
```

where `127.0.0.1` and `10000` are the IP address and TCP port of the server. If the tests run successfully you should obtain the following output:

[27]The library is installed in `/usr/local/lib` and the header files in `/usr/local/include/qrsimtcpclient/`.

```
$ ./testclient 127.0.0.1 10000
QRSIM init test[PASSED]
QRSIM reset test [PASSED]
QRSIM stepWP test [PASSED]
QRSIM stepCtrl test [PASSED]
QRSIM stepVel test [PASSED]
QRSIM disconnect test [PASSED]
QRSIM quit test [PASSED]
```

4 Scenarios

This section describes the use of QRSim to implement three types of scenarios devised to develop and test learning and control algorithm. The three scenarios are explicitly chosen to expose different types of challenges that occur in the domain of multi-platform aerial robotics so to provide a variety of opportunities for investigation.

For each of the three scenarios; we give a formal description of the scenario in terms of its setup, its objective, and its variations as well as handy code examples.

4.1 Scenario 1: Cats and Mouse Game

The first of the scenarios is designed to focus primarily on the challenges encountered in the coordinated control of multiple UAVs; the associated problems of sensing and state estimation are somewhat simplified by the choice of task, environment and platform sensors.

4.1.1 Description

The task to be accomplished in this scenario is in the form of a team game in which N helicopters (cats) have to surround and effectively trap (i.e. get close to) another helicopter (mouse) at the end of the allotted time.

For simplicity the task is assumed to take place in an area devoided of obstacles so that helicopters can fly freely. Getting in contact with the ground or another UAV will however produce a collision. The platforms are equipped with noisy sensors so only observation of the vehicle state are available. Depending on the circumstances wind and other aerodynamic disturbances that affect the flight behaviour might be present in the flying area.

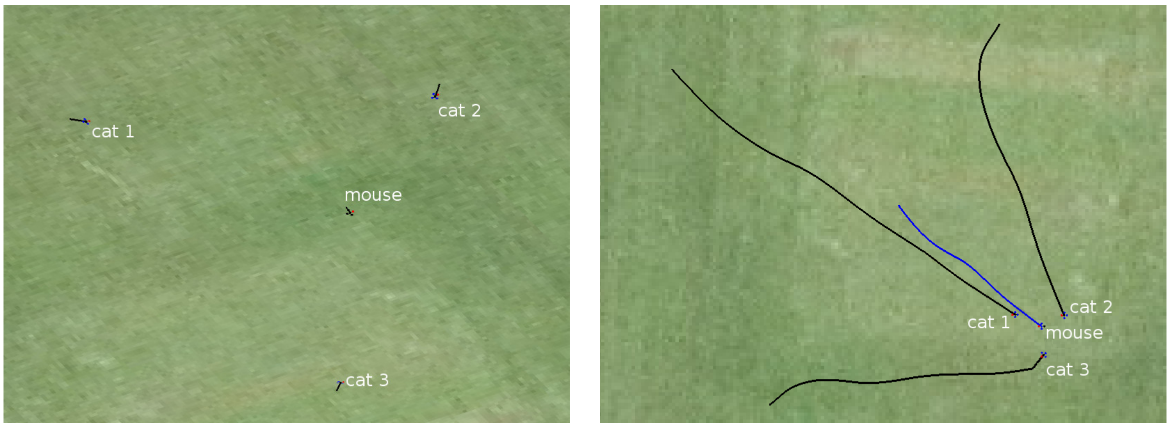
Snapshots of the initial ($t = 0$) and terminal ($t = T$) configurations from a typical successful run are visible in figures 2(a) and 2(b) respectively.

In the next section we give a more formal description of the problem.

4.1.2 MDP

The system underlying the task is modelled as a discrete time, finite horizon MDP on a continuous state space. The system runs from $t = 0$ to $t = T$ with a time step equivalent to $1s$ of simulated time[28].

[28]The update rate is user-configurable with default value of 1Hz.



(a) $t = 0$ (b) $t = T$

Figure 2: Typical successful run: start(a) and end(b) configurations.

State: The state $s_t = (x_t^1, ..., x_t^N, x_t^m)$ at time t comprises of the state vectors of all the cats $(x^1, ..., x^N)$ and of the mouse (x^m) helicopters. Each platform state contains in turn the position $([p_x, p_y, p_z]^\intercal)$, velocity $([u, v, w]^\intercal)$, orientation $([\phi, \theta, \psi]^\intercal)$ and rotational velocity $([p, q, r]^\intercal)$ of the platform:

$$x = [p_x, p_y, p_z, \phi, \theta, \psi, u, v, w, p, q, r]^\intercal . \tag{1}$$

All of these are continuous variables (see section 4.1.4 for more details).

The environment is itself three dimensional although we will see (section 4.1.2) that the helicopters are assumed to fly at a fixed altitude.

Initial State: At the beginning of the task the mouse is placed at the origin of the flight space[29] and the cats are positioned randomly around the mouse.

The initial positions of cats are generated as:

$$\begin{bmatrix} p_x^i \\ p_y^i \\ p_z^i \end{bmatrix}_0 = \begin{bmatrix} p_x^m \\ p_y^m \\ 0 \end{bmatrix} + \begin{bmatrix} d_i \cos(\alpha_i) \\ d_i \sin(\alpha_i) \\ -h_{fix} \end{bmatrix} \qquad i \in 1..N$$

where $\alpha_i \sim \mathcal{U}(0, 2\pi)$, $d_i \sim \mathcal{U}(D_m, D_M)$ and $\mathcal{U}(a, b)$ indicates a uniform distribution over the interval $[a, b]$. The minimum and maximum distance from the mouse D_m, D_M and the altitude h_{fix} are user-configurable. An additional parameter D_{c2c} allows to define a minimum distance between any two cats; this prevent the occurrence of an initial state in which two cats are too close.

At $t = 0$ all the helicopters are stationary (i.e. their velocities are zero).

Actions: A real quadrotor of the type considered in our task generally presents four continuous control inputs (i.e. pitch, roll, yaw angles and throttle) which needs to be updated at a rate of $50Hz$. Even with a moderately long time horizon the control space becomes rather large.

[29]Without loss of generality since the control problem depends on the relative positions of mouse and cats as long as the flying area is sufficiently large.

To limit the space of control inputs, in our setup each UAV is equipped with a closed loop PID controller that accepts 2D linear velocity commands $a_t^i = [v_x^i, v_y^i]^\intercal$ (in global coordinates) and combines them with the estimated platform velocity to produce the necessary attitude and throttle commands for the platform. The PID accepts commanded velocity at a rate of $1Hz$ and provides the platform controls at $50Hz$. To additionally reduce the control space the PID takes also care of maintaining a constant altitude and heading[30].

The reduced action space a_t at time t comprises of the 2D linear velocity commands for each of the cats helicopters:

$$a_t = (a_t^1, ..., a_t^N). \tag{2}$$

Dynamics: The task has a Markovian transition dynamics defined by $P(s_{t+1}|s_t, a_t)$ which denotes the conditional density of state s at time $t+1$ given state-action $(s_t, a_t) = (s, a)$ at time t.

In the case of the cats helicopters the transition dynamics is defined by the combination of PID velocity controllers, platforms dynamics, sensor and wind dynamics (since these in turn effect the quadrotor) all of which are part of the QRSim simulator[31].

For the mouse helicopter the transition dynamics is not only based on the platform dynamics but also on the way the mouse moves in response to the cats.

As the task starts the mouse helicopter tries to escape the cats by moving at a constant (max) speed[32] and choosing a direction that prioritize escaping from the closer cats. In specific the 2D velocity action for the mouse at time t is computed as:

$$a_t^m = V_M \frac{\mathsf{v}_t}{\|\mathsf{v}_t\|} \quad \text{where} \quad \mathsf{v}_t = \sum_{i=1}^N \frac{\begin{bmatrix} p_x^i \\ p_y^i \end{bmatrix}_t - \begin{bmatrix} p_x^m \\ p_y^m \end{bmatrix}_t}{\left\| \begin{bmatrix} p_x^i \\ p_y^i \end{bmatrix}_t - \begin{bmatrix} p_x^m \\ p_y^m \end{bmatrix}_t \right\|^2} \tag{3}$$

and V_M is the (user-configurable) maximum speed that the mouse can achieve.

Different control strategies could be considered for the mouse; in practice the one considered in equation 3 is both simple and has shown to be sufficient to provide for a challenging task.

Observations The helicopter state x_t^i is observed directly although depending on the specific task variation (see Section 4.1.3) the state variables might be affected by stochastic noise. For more details about the noise models used by QRSim we refer the reader to section 5.

Reward: The cats are successful if they are able to trap the mouse at the end of the task; a meaningful final reward can be defined as the sum of the squared (2D) distances[33]

[30]The use of a PID controller in addition to reducing the number of control inputs makes the task closer to what is possible to implement and test using real quadrotors.

[31]We refer to the QRSim manual for details.

[32]While the control law that governs the quadrotor attempts to maintain a fix maximum speed, in practice the true speed will not be constant due to sensor noise wind disturbance and dynamic effects.

[33]$d_{2D}(x_T^i, x_T^m) = \|[p_x^i, p_y^i]_T^\intercal - [p_x^m, p_y^m]_T^\intercal\|$.

between the cats and the mouse at time T:

$$r_T = -\sum_{i=1}^{N} d_{2D}(x_T^i, x_T^m)^2.$$

A large negative reward[34] is returned if any of the helicopters (including the mouse) goes outside of the flying area or if any collision happens during the task.

4.1.3 Task Variations

The difficulty of the considered control problem depends on the level of sensor noise and wind disturbance; so we define three versions of the task with increasing level of realism (and consequently of difficulty):

- ***1A*** *noiseless:* the dynamics of the quadrotors is deterministic and the state returned is the true platform state;
- ***1B*** *noisy:* the dynamics of the quadrotors is stochastic and the state returned is a noisy estimate of the platform state (i.e. with additional correlated noise);
- ***1C*** *noisy and windy:* the dynamics of the quadrotors is stochastic and affected by wind disturbances (following a wind model), the state returned is a noisy estimate of the platform state (i.e. with additional correlated noise).

4.1.4 Simulation Code

The cat and mouse scenario can be easily implemented on top of QRSim by means of dedicated task classes (see section 2.4). In specific we make available one task class for each of the three scenario variations just introduced:

- *1A*: `TaskCatsMouseNoiseless.m`,
- *1B*: `TaskCatsMouseNoisy.m`,
- *1C*: `TaskCatsMouseNoisyAndWindy.m`.

Commenting the code in details is beyond the scope of this note but it is useful to briefly outline which task methods are responsible for handling the various task specific components:

- `init()`: defines all the platforms and sensor parameters;
- `reset()`: defines the UAVs starting condition;
- `step(U)`: defines the mouse evasion strategy and uses the PID controllers to transform velocity commands into angle command to the helicopters;
- `reward()`: defines the task reward.

Listing 7: main_catsmouse.m

```
qrsim = QRSim();

% load task parameters and do housekeeping
state = qrsim.init('TaskCatsMouseNoisyAndWindy');

U = zeros(2,state.task.Nc);

% run the scenario and at every time step generate a control
% input for each of the uavs
for i=1:state.task.durationInSteps,

  % get the mouse position (note id state.task.Nc+1)
  mousePos = state.platforms{state.task.Nc+1}.getEX(1:2);

  % quick and easy way of computing velocity controls for each cat
  for j=1:state.task.Nc,
    collisionDistance = state.platforms{j}.getCollisionDistance();

    % vector to the mouse
    u = mousePos - state.platforms{j}.getEX(1:2);

    % add a weighted velocity ("predict" where the mouse will be)
    u = u+(norm(u)/2)*state.platforms{state.task.Nc+1}.getEX(18:19);

    % keep away from other cats if closer than 2*collisionDistance
    for k = 1:state.task.Nc,
      if(state.platforms{k}.isValid())
        d = state.platforms{j}.getEX(1:2)
              - state.platforms{k}.getEX(1:2);
        if((k~=j)&&(norm(d)<2*collisionDistance))
          u = u+(1/(norm(d)-collisionDistance()))*(d/norm(d));
        end
      end
    end

    % scale by the max allowed velocity
    U(:,j) = state.task.velPIDs{j}.maxv*(u/norm(u));
  end

  % step simulator
  qrsim.step(U);
end

% get final reward
fprintf('final reward: %f\n',qrsim.reward());
```

Listing 7 (file `main_catsmouse.m`) shows the set of steps that are necessary to run a scenario task.

After the desired task is initialized (line 4), a basic for loop is executed for the number of time steps specified by the task. Within the loop the 2D velocity control is computed for each helicopter and passed to the corresponding PID (line 37). In our listing we show a simple (and suboptimal) scheme in which the velocity direction of each cat is chosen as a combination of a component directly towards the future mouse position mouse (line 20-23) and a component that keeps the UAV away from neighbouring cats (line 26-33). The helicopter control inputs produced by the PID controllers are then used to step the simulator (line 41). Finally after the execution of the task is concluded, the final reward for the task can be retrieved (line 45).

We remind the reader that within the simulator the helicopter state x_t^i is denoted as `eX`:

$$\texttt{eX} = [\tilde{p}_x, \tilde{p}_y, \tilde{p}_z, \tilde{\phi}, \tilde{\theta}, \tilde{\psi}, 0, 0, 0, \tilde{p}, \tilde{q}, \tilde{r}, 0, \tilde{a}_x, \tilde{a}_y, \tilde{a}_z, h, \dot{p}_x, \dot{p}_y, \dot{h}]^\intercal$$

while the actions a_t^i are denoted as controls `u`:

$$\texttt{u} = [v_x, v_y]^\intercal$$

with the variables defined in section 5.1.

The task, the configurations and the example main files are in the directory `scenarios/catsmouse` within the QRSim simulator.

4.2 Scenario 2: Search and Rescue

The second scenario is designed explicitly to expose the complex interplay between sensing and acting typical in autonomous robotics task. To solve the task the agent/s has to perform inference about the state of the environment given some observations and take actions based on its belief.

4.2.1 Description

The task to be accomplished in this scenario is in the form of a wilderness search and rescue mission; several targets (people) are lost/injured on the ground in a landscape and need to be located and rescued.

In addition to its navigation sensors each helicopter agent taking part in the search is equipped with a camera/classification module that allows it to detect the presence of targets in its field of view. Rather than raw images the camera module provides higher-level data in the form of log likelihood ratios for the current observation conditioned on the presence or absence of a target. The quality of detection depends upon the ground type and the geometry between helicopter and ground (e.g. the distance).

The search area is limited but its extent is so that a trivial lawn mower pattern of search will not allow covering all the area in the allotted time. The landscape has different types of terrain; persons are more likely to be present on some class of terrain than others. The persons are assumed to not move during the task.

[34]The default value of the negative reward is -1000 but it can be configured by the user.

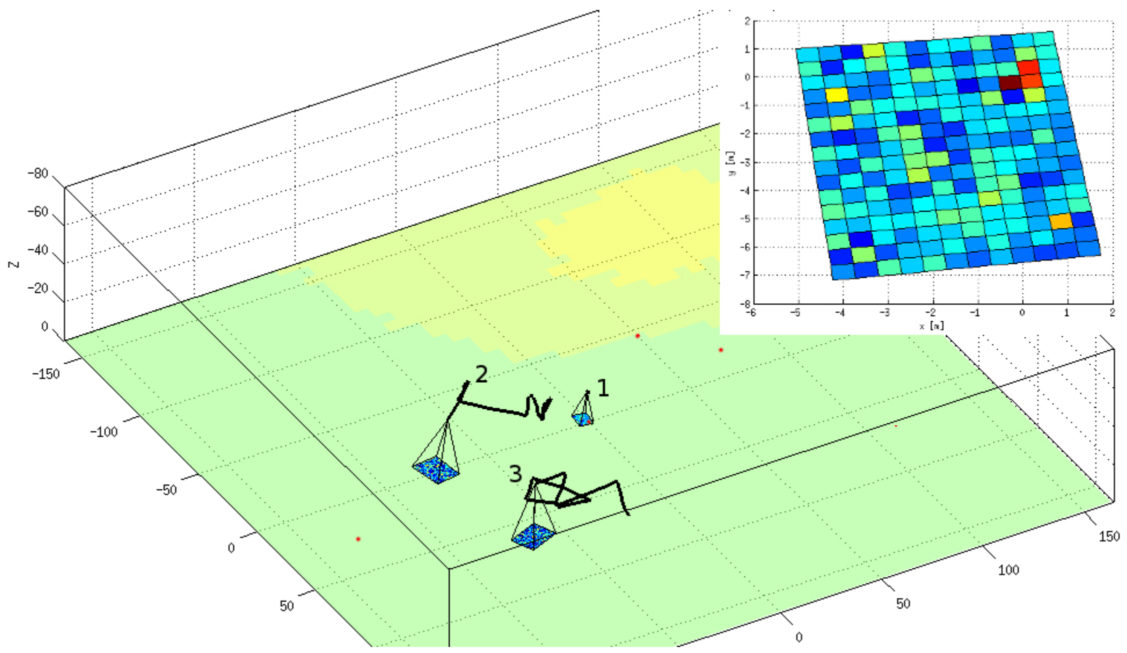


Figure 3: Search and rescue run with three helicopters; note the different terrain types (denoted by different colors) and the persons (red dots). The insert shows the camera observation from UAV 1 that happens to be over a person.

Additionally we assume that the flying area is free of obstacles so that the UAVs can move freely in the 3D space. Getting in contact with the ground or another UAV will however produce a collision.

A snapshot from a typical run is shown in figure 3 along with an example of the observation returned by the camera/classifier model.

4.2.2 MDP

The system underlying the task is modelled as a discrete time, finite horizon MDP on a continuous state space. The system runs from $t = 0$ to $t = T$ with a time step equivalent to $1s$ of simulated time[35].

States: The state $s_t = (x_t^1, ..., x_t^N, b_t^1, ..., b_t^P)$ at time t comprises the helicopters state vectors x_t^i and the location of the P targets b_t^j.

Each platform state contains in turn the position ($[p_x, p_y, p_z]^\intercal$), velocity ($[u, v, w]^\intercal$), orientation ($[\phi, \theta, \psi]^\intercal$) and rotational velocity ($[p, q, r]^\intercal$) of the platform:

$$x = [p_x, p_y, p_z, \phi, \theta, \psi, u, v, w, p, q, r]^\intercal. \quad (4)$$

The environment is itself three dimensional and the helicopter can freely move in three dimensions.

The person's location is expressed in terms of its coordinate w.r.t. the global reference frame $b^j = [p_x^j, p_y^j, 0]^\intercal$; the person's z coordinate is zero since we assume that the ground is flat and that the targets are located on the ground.

[35]The update rate is user-configurable with default value of 1Hz.

Initial State: At the beginning of the task a new terrain map is generated based on the number of terrain classes and split between class types specified by the user[36]. The extent of the flying area is scaled accordingly to the number of platforms and time horizon in order to ensure that the task is nontrivial.

Subsequently a number of persons is randomly placed in the search area. The user can specify the total number of persons as well as control where persons are placed by specifying the probability of a target to be located on a specific terrain class[37].

Finally the helicopters are located randomly (with uniform probability) around the flying area at the nominal flight height[38].

Actions: At each time step the agent specifies an action a^i_t for each of the UAVs taking part in the task; the action is expressed in terms of a 3D velocity vector in global NED coordinates:

$$a^i_t = [v^i_x, v^i_y, v^i_z]^\intercal_t \qquad\qquad i \in [1..N]. \tag{5}$$

Similarly to what explained for in section 4.1.2 the actions are nothing more than set points to a PID controller that attempts to fly the UAV at the requested velocity. Due to delays and disturbances mismatches between the commanded and the actual velocity of the UAV have to be expected.

Dynamics: For the UAVs the transition dynamics is defined by the combination of PID velocity controllers, platforms dynamics, sensor and wind dynamics (since these in turn effect the quadrotor) all of which are specified by the QRSim simulator. Such transition dynamics are Markovian and defined by $P(x_{t+1}|x_t, a_t)$ which denotes the conditional density of state x at $t+1$ given $(x_t, a_t) = (x, a)$ at time t. The dynamics of the UAVs is independent of the targets b^j_t.

Targets b^j are stationary unless a helicopter hovers (i.e. its speed is low) sufficiently close to it, in which case the target is considered rescued and removed from the environment. More formally:

$$\begin{array}{ll} if\ \exists\ i,j\ :\ d_{3D}\left(x^i_t, b^j_t\right) < \delta\ \wedge\ \parallel [u^i, v^i, w^i]^\intercal_t \parallel < \epsilon & \\ \Rightarrow \quad P = P - 1 & i \in \{1,...,N\}, j \in \{1,...,P\} \end{array} \tag{6}$$

where d_{3D} is the standard Euclidean distance in three dimensions[39] and (ϵ, δ) are user specified thresholds[40].

Observations The helicopter state x_t is observed directly although depending on the specific task variation (see Section 4.2.3) the state variables might be affected by stochastic noise.

[36]The user specifies what percentage of the total area belongs to each class.

[37]Each person's location is generated independently.

[38]The initial altitude can be configured by the user but is set to a $25m$ default value.

[39]Defined as

$$d_{3D}(x_t, b^j_t) = \parallel b^j_t - [p_x, p_y, p_z]^\intercal_t \parallel .$$

[40]The default values for the distance threshold and the speed threshold are $\delta = 5m$ and $\epsilon = 0.1m/s$

The position of the targets b_t^j is not known, but observations o_t are provided by the camera at each time step.

Following standard object detection techniques we assume that the incoming image (at time t) is split into M_t windows $\{w_t^k\}_{k=1}^{M_t}$ which are then analysed for targets[41]. Given the current UAV pose x_t and the fixed camera parameters the set of windows projects on the ground to a set of M patches $\{g_t^k\}_{k=1}^{M_t}$. More precisely this assumes that a map is available and that the mapping

$$\{g_t^k\}_{k=1}^{M_t} \mapsto \{w_t^k\}_{k=1}^{M_t}$$

is known, but does not have to be considered explicitly by the agent.

We assume that some form of person detection algorithm is run on each of the image windows w_t^k. As a result it is possible to evaluate the probability that such image patch was originated by a person (at the corresponding ground location g_t^k) versus the probability that w_t^k was originated by clear ground at location g_t^k; what is commonly called a likelihood ratio.

To generate likelihood ratios we use a model learned from the scores obtained by running an off-the-shelf person classifier on aerial images collected with the quadrotor UAV in use at UCL. We refer to section 5.8 for more details about such model.

The observation o_t is simply the collection of the log likelihood ratios obtained for each of the image windows w_t^k (and therefore also for the corresponding ground patches g_t^k):

$$o_t = \left\{ \log \left(\frac{\Pr(\text{image at time } t \mid \text{target in } g_t^k, \text{ UAV at } x_t)}{\Pr(\text{image at time } t \mid \text{no target in } g_t^k, \text{ UAV at } x_t)} \right) \right\}_{k=1}^{M_t}.$$

An example of the observation returned by our simulated scenario is visible in the insert of figure 3. A higher value of the ratio (red colour) is noticeable for the ground patches that are at the person location; also note how the patches are conveniently expressed in ground coordinates.

Reward: The reward r_t for the task is designed to prize the UAVs for quickly rescuing targets. In this spirit r_t at time t is defined to be 1 when a helicopter hovers sufficiently close to a target and a small negative value otherwise. More formally:

$$r_t = \begin{cases} 1 & if\ \exists\ i,j\ :\ d_{3D}\left(x_t^i, b_t^j\right) < \delta \wedge \parallel [u,v,w]_t^\intercal \parallel < \epsilon \quad i \in \{1,...,N\}, j \in \{1,...,P\} \\ -\frac{1}{T} & otherwise \end{cases} \tag{7}$$

where T is the task duration T and ϵ, δ are respectively the distance and velocity thresholds that define a person as rescued (see section 4.2.2). A large negative reward[42] is returned if the helicopter goes outside of the flying area or if any collision happens during the task.

[41]The window size is informed by the current altitude and an assumed fixed size for the person.

[42]The default value of the negative reward is -1000 but it can be configured by the user.

4.2.3 Task Variations

The difficulty of solving the task depends on the number of platforms that are employed as well as on the level of sensor noise and wind disturbance; we provide four versions of the task with increasing level of difficulty:

- ***2A** single helicopter noiseless:* only one helicopter is used for the search, its dynamic is deterministic and the state returned is the true platform state;
- ***2B** single helicopter noisy:* only one helicopter is used for the search, its dynamics is stochastic and the state returned is a noisy estimate of the platform state (i.e. with additional correlated noise);
- ***2C** multiple helicopters noiseless:* several helicopters are used for the search, their dynamics is deterministic and the state returned is the true platform state;
- ***2D** multiple helicopters noisy and windy:* several helicopters are used for the search, their dynamics is stochastic and affected by wind disturbances (following a wind model), the state returned is a noisy estimate of the platform state (i.e. with additional correlated noise).

4.2.4 Simulation Code

A simulated version of each of the variations of this scenario is made available in the form of a task class that can be readily used within the QRSim helicopter simulator. More specifically the four task variations presented in section 4.2.3 are named:

- *2A*: `TaskSearchRescueSingleNoiseless.m`,
- *2B*: `TaskSearchRescueSingleNoisy.m`,
- *2C*: `TaskSearchRescueMultipleNoiseless.m`
- *2D*: `TaskSearchRescueMultipleNoisyAndWindy.m`.

Commenting the code in details is beyond the scope of this node but it is useful to briefly outline which task methods are responsible for handling the various task specific components:

- `init()`: defines all the platforms, sensor and environment parameters;
- `reset()`: defines the task starting condition for UAVs and persons;
- `step(U)`: uses the PID controllers to transform velocity commands into angle command to the helicopters, checks if a target was located and removes it;
- `reward()`: returns the reward at the current time step;
- `getNumberOfPersons()`: returns the number of persons present at the beginning of the task.

Listing 8: main_searchrescue.m

```
% create simulator object
qrsim = QRSim();

% load task parameters and do housekeeping
state = qrsim.init('TaskSearchRescueSingleNoiseless');

% create a 3 x helicopters matrix of control inputs column i
% will contain the 3D NED velocity [vx;vy;vz] for helicopter i
U = zeros(3,state.task.numUAVs);

% number of persons to locate in this task
np = state.task.getNumberOfPersons();

% run the scenario and at every time step generate a control
% input for each of the uavs
for i=1:state.task.durationInSteps,
  % random search policy in which the helicopter(s) moves around
  % at a fixed velocity changing direction every once in a while
  if(rem(i-1,10)==0)
    for j=1:state.task.numUAVs,
      if(state.platforms{j}.isValid())
        % random velocity direction
        u(:,j) = rand(2,1)-[0.5;0.5];
        % fixed velocity 0.5 times max allowed velocity
        U(:,j) = [0.5*state.task.velPIDs{j}.maxv*...
                          (u(:,j)/norm(u(:,j)));0];
        % if the uav is going astray we point it back to the center
        p = state.platforms{j}.getEX(1:2);
        if(norm(p)>100)
          U(:,j) = [-0.8*state.task.velPIDs{j}.maxv*p/norm(p);0];
        end
      end
    end
  end
  % step simulator
  qrsim.step(U);

  % camera measurements, a simple structure with fields:
  % llkd      log-likelihood difference for each ground patch
  % wg        list of corner points for the ground patches
  % gridDims  dimensions of the grid of measurements
  for j=1:state.task.numUAVs,
    m = state.platforms{j}.getCameraOutput();
  end
  % reward (1 as soon as we hovering close enough to a person)
  r = qrsim.reward();
end
```

We also provide the example file `main_searchrescue.m` (listing 8) which briefly shows how to initialize and run any of the search and rescue tasks. The layout of the code is very similar to what we have seen for the cats and mouse scenarios; after a preliminary initialization (lines $2-9$), a for loop takes care of stepping the simulation (line 36) for the predefined number of time steps. For simplicity in this example, the UAVs move in the space at a fix velocity changing their direction randomly every 10 time steps (lines $19-34$). For each UAV the camera measurement can be retrieved after stepping the simulator with a new action (line 43). In a more interesting policy than the random one we are showing, an agent would make use of this new observation (and of the previous) to decide its next action. Lastly line 46 shows how to retrieve the reward at the current time step.

In some situations it might be useful to know the total number of targets present in the environment, line 12 shows how to do that using the `getNumberOfPersons()` method.

We remind the reader that within the simulator the helicopter state x_t^i is denoted as `eX`:

$$\texttt{eX} = [\tilde{p}_x, \tilde{p}_y, \tilde{p}_z, \tilde{\phi}, \tilde{\theta}, \tilde{\psi}, 0, 0, 0, \tilde{p}, \tilde{q}, \tilde{r}, 0, \tilde{a}_x, \tilde{a}_y, \tilde{a}_z, h, \dot{p}_x, \dot{p}_y, \dot{h}]^\intercal$$

while the actions a_t^i are denoted as controls `u`:

$$\texttt{u} = [v_x, v_y, v_z]^\intercal$$

with the variables defined in section 5.1.

The task, the configurations and the example main files are in the directory `scenarios/searchrescue` within the QRSim simulator.

4.3 Scenario 3: Plume Modelling

In the third and last scenario we design tasks in which sensing and estimation of the target environment play a crucial role. In specific the actions of the agent are aimed at generating reliable predictions about the flying area.

4.3.1 Description

In this scenario we envisage the situation in which a plume is dispersed in the flying area and we are interested in using one or more UAVs to estimate its concentration[43].

Depending on the task settings (see section 4.3.3) one or more sources might be present in the environment and the plume distribution might evolve over time. Any wind affects both the UAV flight behaviour and the plume distribution. Each UAV is equipped with noisy navigation sensors that allow observing its location, attitude and velocity and also with a sensor that produces noisy observation of the plume concentration.

The plume concentration follows a known model but with unknown parameter values; the task objective is to provide a concentration estimate $\hat{c}_T$ at some pre-specified time T. For this scenario we selected three main classes of concentration models; they were chosen for being widely adopted ([24]) while remaining relatively simple:

[43] Hereby we simply refer to "concentration", in a real environment this would be a property of the plume that can be realistically measured e.g. CO concentration.

- *Gaussian*: when no wind is present in the flying area and the source emits plume with a constant rate, the plume disperses around the source and its concentration can be described by a three dimensional Gaussian (see section 5.9.1). Since the source emits at constant rate the resulting dispersion pattern is static, however to make the model not completely trivial we assume that the dispersion is not necessarily isotropic. Figure 4(a) depicts an example of type of concentration pattern produced by this model.
- *Gaussian Dispersion*: when wind is present in the flying area, and the source emits plume with a constant rate, the plume is blown downwind and disperses as it gets farther from the source. The concentration takes a cone like shape which expands as the distance from the source increases (see section 5.9.2). In the case in which the mean wind velocity and direction do not change with time[44], the resulting concentration is also time invariant. An example of the resulting dispersion is visible in Figure 4(b).
- *Gaussian Puff Dispersion* when wind is present in the flying area but the source emits plume for short time intervals (i.e. short puffs), each puff travels downwind expanding as it gets farther from the source. At every time instant the resulting concentration is the superposition of the many puffs (see section 5.9.2); the concentration is therefore time variant. Figure 4(c) shows an example of such dispersion model.

For each type of dispersion model the case in which more than one source is present in the environment can be considered simply by superimposing the effects of several individual sources; superposition gives origin to substantially more complex concentration patterns.

As in the previous tasks we assume that the flying area is free of obstacles so that the UAVs can move freely in the 3D space. Getting in contact with the ground or another UAV will however produce a collision.

Note: Since the dispersion models are known, one possible way to solve the tasks is to estimate the model parameters (or a distribution over them). While this is an appropriate solution we emphasize that the agent is not required to solve the task in this way and so model agnostic way of producing concentration estimations are equally appropriate.

4.3.2 MDP

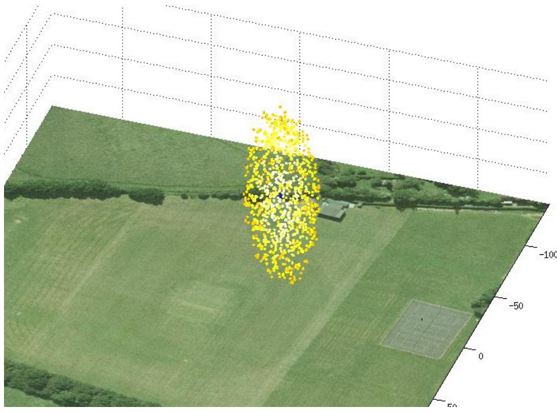
The system underlying the task is modelled as a discrete time Markov process on a continuous state space. The system is updated at a frequency of 1Hz[45] and the task has a duration T.

States: the state $s_t = (x_t^1, .., x_t^N, c_t)$ at time t comprises the helicopter/s state x_t^i and the plume concentration c_t. Each platform state contains in turn the position ($[p_x, p_y, p_z]^\intercal$), velocity ($[u, v, w]^\intercal$), orientation ($[\phi, \theta, \psi]^\intercal$) and rotational velocity ($[p, q, r]^\intercal$) of the platform:

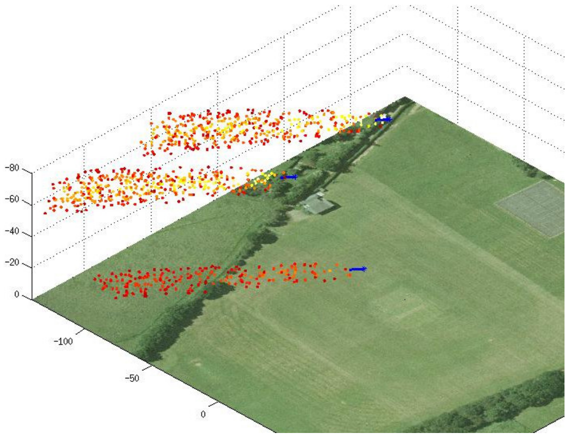
$$x = [p_x, p_y, p_z, \phi, \theta, \psi, u, v, w, p, q, r]^\intercal. \tag{8}$$

[44]This will be the case in our settings.

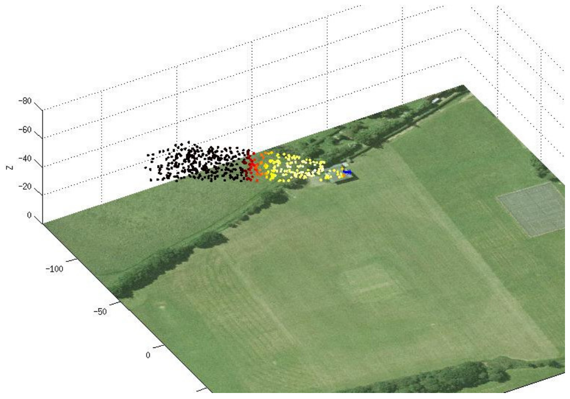
[45]The update rate is user-configurable with default value of 1Hz.



(a) Single source Gaussian concentration.



(b) Multiple sources Gaussian dispersion.



(c) Single source Gaussian puff dispersion.

Figure 4: Plume models; darker markers indicate lower concentration.

The environment is itself three dimensional and the helicopter can freely move in three dimensions.

The plume concentration is defined at every 3D location over the whole flight volume.

Initial State: At the beginning of the task a new set of source locations within the flying area is created with their associated emission rates so that given the current wind, the complete plume distribution can be computed. The user can control the number, height and emission rate of sources through task parameters.

At $t = 0$ the helicopters are located randomly (with uniform probability) in the fly area at the starting flight height[46].

Actions: At each time step the agent specifies an action a_t^i for each of the UAV taking part in the task; the action is expressed in terms of a 3D velocity vector in global NED coordinates:

$$a_t^i = [v_x^i, v_y^i, v_z^i]_t^\intercal \qquad\qquad i \in [1..N]. \tag{9}$$

[46]The initial altitude can be configured by the user but is set to a $25m$ default value.

The actions are nothing more than set points to a PID controller that attempts to fly the UAV at the requested velocity. Due to delays and disturbances mismatches between the commanded and the actual velocity of the UAV have to be expected.

Dynamics: For the UAVs the transition dynamics is defined by the combination of PID velocity controllers, platforms dynamics, sensor and wind dynamics (since these in turn effect the quadrotor) all of which are specified by the QRSim simulator. Such transition dynamics are Markovian and defined by $P(x_{t+1}|x_t, a_t)$ which denotes the conditional density of state x at $t+1$ given $(x_t, a_t) = (x, a)$ at time t.

In the case of non puff-like dispersion models the plume distribution is stationary and therefore does not evolve during the task. For plume distributions defined by the Gaussian puff dispersion model puffs are emitted at random intervals drawn from an exponential distribution. Each puff travels downwind at the mean wind speed and simultaneously expands as specified by equations 58 and 61.

For simplicity any effect that a UAV might have on the plume concentration is neglected and the evolution in time of the plume is assumed to be independent of the helicopter actions and state $P(c_{t+1}|c_t, x_t, a_t) = P(c_{t+1}|c_t)$.

Observations: The helicopter state x_t is observed directly although depending on the specific task variation (see Section 4.3.3) the state variables might be affected by stochastic noise.

The plume concentration c_t is not known; at each time step a noisy observations o_t at the position in which the helicopter is located is returned by a concentration sensor. At present the sensor reading are corrupted by additive Gaussian noise.

Reward: The task objective is to provide a concentration estimate $\hat{c}_T$ at some pre-specified time T (i.e. at the end of the task).

For the tasks in which the concentration is stationary (i.e. tasks 3A, 3B, 3C and 3D), the agent must provide a set of concentration estimates $\{\hat{c}^j_T\}^M_{j=1}$ at a series of M spatial locations specified at the beginning of the task. Given the $\{\hat{c}^j_T\}^M_{j=1}$ the task reward is simply computed as (minus) the square error between the true concentrations and the estimates provided by the agent:

$$r_T = -\sum_{j=1}^{M} |c^j_T - \hat{c}^j_T|^2.$$

For tasks in which the concentration evolves over time (i.e. tasks 3E, 3F and 3G), the concentration at each location $\hat{c}^j_T$ can be thought of as a random variable with an associated probability distribution $\Pr(\hat{c}^j_T)$. The better the agent is at approximating such distribution, the higher should be its reward. A simple way to compute the reward is in the form of (minus) the KL divergence between true concentration distribution $\Pr(c^1_T, .., c^M_T)$ and the estimate provided by the agent[47] $\Pr(\hat{c}^1_T, .., \hat{c}^M_T)$:

$$r_T = -KL(\Pr(c^1_T, .., c^M_T) \| \Pr(\hat{c}^1_T, .., \hat{c}^M_T)).$$

[47]In practice in order to enable the empirical computation of the reward the agent will be asked to return several samples of the concentration at each of the M locations specified by the task.

4.3.3 Task Variations

The complexity of the estimation problem changes substantially depending on the type of dispersion model followed by the plume, and on the number of helicopters used to tackle the task hence we provide several version of the task:

- ***3A*** *single source static Gaussian concentration:* only one helicopter is used for the sampling, its dynamic is deterministic, the navigation sensors return the true platform state, the plume concentration is static and has the form or a three dimensional Gaussian centred at the source (see equation 55).
- ***3B*** *single source static Gaussian dispersion model:* only one helicopter is used for the sampling, its dynamics is stochastic and affected by wind disturbances (following a wind model), the navigation sensors return a noisy estimate of the platform state (i.e. with additional correlated noise) and the plume concentration is static and has the form specified by what is commonly called a Gaussian dispersion model (see equation 56).
- ***3C*** *multiple sources static Gaussian dispersion model:* only one helicopter is used for the sampling, its dynamics is stochastic and affected by wind disturbances (following a wind model), the navigation sensors return a noisy estimate of the platform state (i.e. with additional correlated noise) and the plume concentration is static and has the form specified by the superposition of several sources each of which follows a Gaussian dispersion model (see equation 57).
- ***3D*** *multiple helicopters multiple sources static Gaussian dispersion model :* as above but multiple helicopters are used for the sampling.
- ***3E*** *single source time-varying Gaussian puff dispersion model:* only one helicopter is used for the sampling, its dynamics is stochastic and affected by wind disturbances (following a wind model), the navigation sensors return a noisy estimate of the platform state (i.e. with additional correlated noise) and the plume concentration is time-varying and has the form specified by what is commonly called a Gaussian puff dispersion model (see equation 58).
- ***3F*** *multiple sources time-varying Gaussian puff dispersion model:* only one helicopter is used for the sampling, its dynamics is stochastic and affected by wind disturbances (following a wind model), the navigation sensors return a noisy estimate of the platform state (i.e. with additional correlated noise) and the plume concentration is static and has the form specified by the superposition of several sources each of which follows a Gaussian puff dispersion model (see equation 61).
- ***3G*** *multiple helicopters multiple sources time-varying Gaussian puff dispersion model :* as above but multiple helicopters are used for the sampling.

4.3.4 Simulation Code

As for the previous scenarios, we provide an implementation of a simulated version of each of the scenario variations in the form of a task class that can be readily used within the QRSim helicopter simulator. The seven variations of the scenario are named:

- *3A*: `TaskPlumeSingleSourceGaussian.m`,
- *3B*: `TaskPlumeSingleSourceGaussianDispersion.m`,
- *3C*: `TaskPlumeMultiSourceGaussianDispersion.m`,
- *3D*: `TaskPlumeMultiHeliMultiSourceGaussianDispersion.m`,
- *3E*: `TaskPlumeSingleSourceGaussianPuffDispersion.m`,
- *3F*: `TaskPlumeMultiSourceGaussianPuffDispersion.m`,
- *3G*: `TaskPlumeMultiHeliMultiSourcePuffDispersion.m`.

Commenting the code in details is beyond the scope of this note but it is useful to briefly outline which task methods are responsible for handling the various task specific components:

- `init()`: defines all the platforms, sensor and plume parameters;
- `reset()`: defines the task starting condition for UAVs and plume;
- `step(U)`: uses the PID controllers to transform velocity commands into helicopter controls;
- `reward()`: returns the final reward;
- `getLocations()`: returns the list of 3D locations at which the agent is expected to return predictions;
- `getSamplesPerLocation()`: returns the number samples the agent needs to return at each location, this will be larger than 1 only if the concentration is time variant (i.e. for puff models);
- `setSamples(samples)`: allows the agent to return the predictions at each of the 3D locations so that the task can compute a reward.

We provide the example file `main_plume.m` (listing 9) which shows how to initialize and run one of such tasks.

After the usual initializations (lines 1-7) we show how to retrieve the list of 3D locations at which the agents has to return predictions (line 10).

In the main loop (lines 16-40) for each simulation step we generate a velocity command for each of the UAVs; in this example we follow a random search strategy that consist in flying to a new direction every 10 time steps. After stepping the simulator (line 35) the concentration measurement for the location at which the helicopters are flying can be retrieved (line 38).

Finally after the task time is elapsed, we are required to provide the simulator with the concentration estimates (line 45); in this case for simplicity we set those to randomly generated values (line 43). At line 47 we obtain the reward associated with the provided estimates.

Listing 9: main_plume.m

```
% create simulator object and load task parameters
qrsim = QRSim();
state = qrsim.init('TaskPlumeSingleSourceGaussianDispersion');
U = zeros(3,state.task.numUAVs);

% allocate up a matrix to store all the concentration measurements
plumeMeas = zeros(state.task.numUAVs,state.task.durationInSteps);

% get the list of locations the agent needs to return predictions at
positions = state.task.getLocations();

% predictions to be returned at each location, >1 only for puff models
samplesPerLocation = state.task.getSamplesPerLocation();

% run the scenario and at every time step generate uav controls
for i=1:state.task.durationInSteps,
  % random exploration policy in which the helicopter(s) moves around
  % at a fixed velocity changing direction every once in a while
  if(rem(i-1,10)==0)
    for j=1:state.task.numUAVs,
      if(state.platforms{j}.isValid())
        % random velocity direction scaled by the max allowed velocity
        u(:,j) = rand(2,1)-[0.5;0.5];
        U(:,j) = [0.5*state.task.velPIDs{j}.maxv...
                          *(u(:,j)/norm(u(:,j)));0];
        % if the uav is going astray we point it back to the center
        p = state.platforms{j}.getEX(1:2);
        if(norm(p)>100)
          U(:,j) = [-0.8*state.task.velPIDs{j}.maxv*p/norm(p);0];
        end
      end
    end
  end
  % step simulator
  qrsim.step(U);
  % get plume measurements
  for j=1:state.task.numUAVs,
    plumeMeas(j,i)=state.platforms{j}.getPlumeSensorOutput();
  end
end
% matrix containing all the predictions made by the agent
% for the purpose of illustration we generate those randomly
samples = randn(samplesPerLocation,size(positions,2));
% pass the sample to the task so that a reward can be computed
state.task.setSamples(samples);
% get final reward (note: this works only after setSamples() ).
r = qrsim.reward();
```

The task, the configurations and the example main files are in the directory `scenarios/plume` within the QRSim simulator.

We remind the reader that within the simulator the helicopter state x_t^i is denoted as `eX`:

$$\texttt{eX} = [\tilde{p}_x, \tilde{p}_y, \tilde{p}_z, \tilde{\phi}, \tilde{\theta}, \tilde{\psi}, 0, 0, 0, \tilde{p}, \tilde{q}, \tilde{r}, 0, \tilde{a}_x, \tilde{a}_y, \tilde{a}_z, h, \dot{p}_x, \dot{p}_y, \dot{h}]^\intercal$$

while the actions a_t^i are denoted as controls `u`:

$$\texttt{u} = [v_x, v_y, v_z]^\intercal$$

with the variables defined in section 5.1.

5 Simulator Implementation Details

This section explains in detail the models used in QRSim in order to simulate the quadrotor dynamics, its sensors and the various elements of the environment.

5.1 Reference Frame and Symbols

For convenience we start by showing show the body frame of reference used in many of the computation involving the helicopter dynamics (see figure 5).

The global frame (not shown in the figure) is defined as the local tangent plane to the earth surface at the origin. By definition the body and global frame coincide when the quadrotor is at coordinates $(0, 0, 0)$ and its attitude is zero in all three angles.

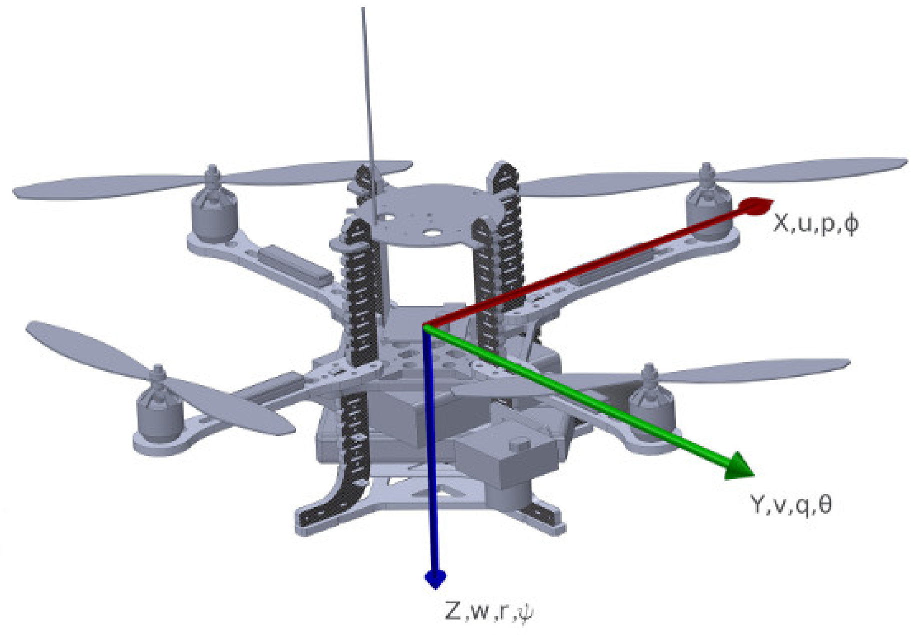


Figure 5: Body NED frame of reference and state variables.

For completeness we report a list of the various symbols that appear throughout this note:

- **True platform state**

  X = $[p_x; p_y; p_z; \phi; \theta; \psi; u; v; w; p; q; r; F_{th}]$

- **Estimated platform state**

  eX = $[\tilde{p}_x; \tilde{p}_y; \tilde{p}_z; \tilde{\phi}; \tilde{\theta}; \tilde{\psi}; 0; 0; 0; \tilde{p}; \tilde{q}; \tilde{r}; 0; \tilde{a}_x; \tilde{a}_y; \tilde{a}_z; h; \dot{p}_x; \dot{p}_y; \dot{h}]$

- **Platform controls**

  U = $[u_{pt}; u_{rl}; u_{th}; u_{ya}; V_b]$

t	time	s
T	task duration	s
s_t	task state at time t	
x_t^i	state vector of UAV i at time t	
a_t^i	action input to UAV i at time t	
N	number of UAVs taking part in the task	
b^j	target/person j state at time t	
P	number of person/targets in the search area	
w_t^k	window k in input image at time t	
g_t^k	ground patch k at time t	
M_t	number of windows in image frame at time t	
o_t	observation at time t	
c_t	plume concentration at time t	
M	number of locations for which the agent should provide estimates	
$\hat{c}_t$	estimated plume concentration at time t	
r_t	reward at time t	
p_x	UAV true x position (NED coordinates)	m
p_y	UAV true y position (NED coordinates)	m
$\tilde{p}_x$	UAV x position estimate from GPS (NED coordinates)	m
$\tilde{p}_y$	UAV y position estimate from GPS (NED coordinates)	m
$\tilde{p}_z$	UAV z position estimate from GPS (NED coordinates)	m
$\tilde{\phi}$	UAV roll attitude in Euler angles right-hand ZYX convention	rad
$\tilde{\theta}$	UAV pitch attitude in Euler angles right-hand ZYX convention	rad
$\tilde{\psi}$	UAV yaw attitude in Euler angles right-hand ZYX convention	rad
$\tilde{p}$	UAV rotational velocity about x body axis from gyro	rad/s
$\tilde{q}$	UAV rotational velocity about y body axis from gyro	rad/s
$\tilde{r}$	UAV rotational velocity about z body axis from gyro	rad/s
$\tilde{a}_x$	UAV linear acceleration in x body axis from accelerometer	m/s^2
$\tilde{a}_y$	UAV linear acceleration in y body axis from accelerometer	m/s^2
$\tilde{a}_z$	UAV linear acceleration in z body axis from accelerometer	m/s^2
h	UAV altitude from altimeter NED	m
$\dot{p}_x$	UAV x velocity from GPS (NED coordinates)	m/s
$\dot{p}_y$	UAV y velocity from GPS (NED coordinates)	m/s

$\dot{h}$	UAV altitude rate from altimeter NED	m/s
v_x	UAV desired x velocity control (NED coordinates)	m/s
v_y	UAV desired y velocity control (NED coordinates)	m/s
v_z	UAV desired z velocity control (NED coordinates)	m/s

5.2 Quadrotor Dynamic Model

Several dynamic models of quadrotor helicopters based on first principles have been published in the literature ([6], [4], [11] to name a few); such examples consider the forces and moments acting on the helicopter in order to obtain the rotational and translational dynamics.

These types of models replicate only the "mechanics" and the aerodynamics of the flying machine but do not include the action of the low level control system necessary to stabilise the rotational dynamics. Since the manufacturer of our quadrotor does not make public the design of the stabilization system, simulating it is not a straightforward task.

Given that close loop flight control of the quadrotor is not our primary interest, for the rotational dynamics we prefer to use an equivalent model that replicates directly the combined effect of both the quadrotor and of its controller. We learn the parameter of such a model directly from flight data in order to match as accurately as possible the characteristics of our flight machine.

For the translational dynamics we use a more standard model that consider the forces acting on the center of mass of the flight machine but we learn from data the relationship between the throttle control input and the generated thrust force. This is known to be very much dependent on the type of motor and propeller, learning it ensures we capture the characteristics of our platform.

Rotational Dynamics

In the case of the pitch and roll motion the Pelican platform accepts as control input the angles that the platform should maintain (u_{pt} and u_{rl} respectively); our model needs to map such controls into the rate of change of the corresponding rotational velocities q and p. From flight tests data we deemed that a second order model can provide a good representation of the dynamic response of the state variable θ and ϕ to changes in control. It was also evident that the low level controller drives the propellers so to limit the maximum rotational velocity (a common precaution that ensures to not exceed the sensing and/or control envelope).

We encoded these findings in equations 10 and 11 where K_{pq0}, K_{pq1} and K_{pq2} are constant factors that are obtained from flight data while $p_{max} = q_{max}$ are derived from the platform firmware settings.

$$\dot{p} = \begin{cases} K_{pq1}(K_{pq0}u_{rl} - \phi) + pK_{pq2} & |p| < p_{max} \\ 0 & |p| \geq p_{max} \end{cases} \tag{10}$$

$$\dot{q} = \begin{cases} K_{pq1}(K_{pq0}u_{pt} - \phi) + qK_{pq2} & |q| < q_{max} \\ 0 & |q| \geq q_{max} \end{cases} \tag{11}$$

Perhaps quite obviously the same parameters are used for the pitch and roll dynamics since the platform is symmetric.

In the case of the yaw motion the control input to the pelican platform takes the for of the desired yaw speed u_{ya}. Even in this case a second order model provides a good fit to the flight data and leads to equation 12.

$$\dot{r} = K_{r0}u_{ya} + rK_{r1}. \tag{12}$$

Again the constants were obtained from flight data.

A Gaussian white noise component is added to each of the angular velocity derivatives in order to account for those effects that are not captured by the model.

Translational Dynamics

In the case of the translational dynamics the first relationship to capture is the one between the throttle control input (u_{th}) and the thrust force (F_{th}) experienced by the quadrotor. Through a series of static tests emerged that the relationship between throttle and thrust can be represented by a second order polynomial:

$$F_T = C_{th0} + C_{th1}u_{th} + C_{th2}u_{th}^2 \tag{13}$$

this is a typical result ([19],[11]) in accordance with basic lift theory which predicts a lift proportional to the square of the blades speed. Full throttle tests also evidenced that the maximum thrust is proportional to the current battery voltage (V_b) and decreases as the battery depletes:

$$F_M = C_{vb0} + C_{vb1}V_b. \tag{14}$$

As expected due to inertia and drag effects, dynamic tests showed that the thrust cannot change instantaneously, but instead it changes at a constant rate[48] τ_1. Exception to this are very slow speed ranges for which behaviour compatible with a first order system is exhibited instead. Combining this dynamic behaviour with the maximum thrust defined by the lowest between (13) and (14) we obtain:

$$\dot{F}_{th} = \begin{cases} -\tau_1 & \max(F_T, F_M) < F_{th} \\ \tau_1 & \max(F_T, F_M) \leq F_{th}. \end{cases} \tag{15}$$

Given F_{th}, the gravity force in body coordinates g_b and the wind velocity in body coordinates $[u_w, v_w, w_w]$, the resulting accelerations in body coordinates can be computed as:

$$\begin{aligned} \dot{u} &= -qw + rv + g_{b_x} + K_{uv}(u - u_w) && (16) \\ \dot{v} &= -ru + pw + g_{b_y} + K_{uv}(v - v_w) && (17) \\ \dot{w} &= -pv + qu + g_{b_z} - \frac{F_{th}}{m} + K_w(w - w_w). && (18) \end{aligned}$$

Where m is the mass of the quadrotor and the the terms $K_{uv}(u - u_w)$,$K_{uv}(v - v_w)$ and $K_w(w - w_w)$ are the effects of the aerodynamic drag. The first two terms on the right hand side of the equations are needed to take into account the fact that the linear velocities u, v, w are expressed in a rotating frame.

[48]We believe that this is due to the speed control loop present in the electronic speed controllers governing the motors.

Equations (10)-(12) and (16)-(18) provide the state update equations which are numerically integrated in order to simulate the time evolution of the state variables.

Similarly to the rotational dynamics even for the translational velocity derivatives we consider an additive Gaussian white noise component in order to account for those effects that are not captured by the model.

The interested reader can find the implementation of the model presented in this section in the file `pelicanODE.m`.

The behaviour of real quadrotor platform depends on environmental conditions as well as on the nature of its on-board sensors; the following sections explain in detail how both sensors and environment are modelled in the simulator.

5.3 Barometric Altimeter

Pressure base altimeter constitutes a light weight and economical way of recovering the altitude of a flying machine. Their precision can reach the order of decimetres making them more accurate than a GPS receiver especially when coupled with other sensors.

The main drawback of a barometric altimeter is that the air pressure tends to change slowly with time due to meteorological effects. Its noise process is therefore time correlated. Obviously the measured pressure is also affected by the local air turbulence (produced both by wind and the propellers). Following [3] and [21] we model the altimeter measurements as follows:

$$\begin{aligned} h &= -p_z + b_h + \nu_{h_m} && (19) \\ \dot{b} &= -\frac{1}{\tau_b} b + \nu_{h_c} && (20) \end{aligned}$$

where h is altimeter output, $-p_z$ is the true altitude, b is the bias noise and ν_{h_m} is white Gaussian measurement noise. b is modelled as a first order Gauss-Markov process with time constant τ_b driven by the white Gaussian noise ν_{h_c}. It is worth noting that $-p_z$ is due to our choice of following the aeronautic convention and adopting a NED body frame.

The interested reader can find the implementation of the model presented in this section in the file `AltimeterGM.m`.

5.4 Orientation Estimator

The orientation of the flying machine is generally estimated on-board using primarily the input from gyros accelerometers and magnetometers, but in some cases might also include other sensors (i.e. the GPS and barometer). Since all these sensors are combined using an estimator (e.g. a Kalman filter), designing a noise model for the orientation measurement that correctly takes into account the estimation effects is all but trivial. In our case matters are further complicated by the fact that the manufacturer does not make available any clear information about the attitude estimator.

Pragmatically we decide to go for a much simpler noise model that attempts to capture the main characteristic of the errors in the estimated attitude. Since the primary component of the estimated attitude is the integration of the gyros readings, the orientation noise tends to be correlated due to uncorrected gyro bias errors. Additional correlation might be introduced also during high acceleration manoeuvres due to the simplification often adopted in the mechanization equation of the estimator process model. Errors are mitigated by the use of acceleration and magnetometer measurements which allow estimating and compensating for such biases.

To replicate such behaviour of reverting back to zero, we choose to model the noise on each each of the three estimated angles a zero mean Ornstein–Uhlenbeck process. This allows to replicate the desired time correlation of the attitude estimates while still being a simple model. The resulting noise model is the following:

$$\begin{aligned}\tilde{\phi} &= \phi + b_\phi && (21)\\ \tilde{\theta} &= \theta + b_\theta && (22)\\ \tilde{\psi} &= \psi + b_\psi && (23)\\ \dot{b}_\phi &= -\lambda_\phi b_\phi + \nu_{b_\phi} && (24)\\ \dot{b}_\theta &= -\lambda_\theta b_\theta + \nu_{b_\theta} && (25)\\ \dot{b}_\psi &= -\lambda_\psi b_\psi + \nu_{b_\psi} && (26)\end{aligned}$$

where $\lambda_\phi, \lambda_\theta, \lambda_\psi$ are mean reversion speeds and $\nu_{b_\phi}, \nu_{b_\theta}, \nu_{b_\psi}$ are white Gaussian noises.

The interested reader can find the implementation of the model presented in this section in the file `OrientationEstimatorGM.m`.

5.5 Accelerometer and Gyroscope

Several authors have studied the types of errors that commonly affect accelerometers and gyros ([23][13]) and very accurate, albeit complex, representation of such noise models have been standardised [2]. For the type of high level tasks we consider acceleration and rotational velocity measurements are of secondary importance. At this stage we prefer to go for a simpler model that considers only the main white noise component of the standardized model. The modular nature of the simulator will allow to easily upgrade to a more complex noise model if this is deemed necessary.

For both accelerometer and gyros reading (respectively $[\tilde{a}_x; \tilde{a}_y; \tilde{a}_z]$ and $[\tilde{p}; \tilde{q}; \tilde{r}]$) the noise is considered additive:

$$\begin{aligned}\tilde{p} &= p + \nu_p && (27)\\ \tilde{q} &= q + \nu_q && (28)\\ \tilde{r} &= r + \nu_r && (29)\end{aligned}$$

$$\begin{aligned}\tilde{a}_x &= a_x + \nu_x && (30)\\ \tilde{a}_y &= a_y + \nu_y && (31)\\ \tilde{a}_z &= a_z + \nu_z && (32)\end{aligned}$$

where $\nu_p, \nu_q, \nu_r, \nu_x, \nu_y, \nu_z$ are white Gaussian processes.

The interested reader can find the implementation of the model presented in this section in the file `AccelerometerG.m` and `GyroscopeG.m`.

5.6 GPS

GPS is complex and errors arise from a wide variety of sources with differing characteristics. The quality of the computed position depends on errors affecting the satellite vehicles (sv from now on), the propagation of the GPS signal through the atmosphere, the number and configuration of svs used to compute a solution and the receiver itself.

If instead of concentrating on a noise model at the level of computed receiver position, we work at the level of the single pseudo-range measurement provided by each of the svs our task is greatly simplified since accurate models of the typical pseudo-range errors are available ([18], [7]). Given such measurements we will then need to compute the position solution taking into account the receiver characteristics.

From an implementation perspective we split the GPS model into a `GPSSpaceSegmentGM` (this is an `environmentObject`) which models the pseudo-range noise affecting the svs, and a `GPSreceiverG` object that models the receiver present on each of the quadrotors. Having a single `GPSSpaceSegmentGM` implies that the noise affecting the position measurement of quadrotors using the same satellites will be to a certain extent correlated. This is exactly what happens in practice when quadrotor are flying in the same geographical location and is one of the aspects that we necessarily want to capture with our model.

Following [18] the pseudo-range measurements (ρ) can be written as:

$$\rho = r + \delta_{eph} + \delta_{iono} + \delta_{tropo} - \delta_{clock} + \delta_{mp} + \nu_{rcvr}, \tag{33}$$

where:

- r, is the true range,
- δ_{eph}, is the satellite ephemeris error,
- δ_{iono}, is the ionosphere error,
- δ_{tropo}, is the troposphere error,
- δ_{clock}, is the receiver clock error,
- δ_{mp}, is the multipath error,
- ν_{rcvr}, is the receiver measurement noise.

The true range r is simply the distance between the current (simulated) position of the receiver[49] and the position of the satellite in question. The position the satellite is obtained from a log of actual satellite positions (see parameter `orbitfile` in listing 2), therefore as the simulation time progresses the position of the satellites changes as it would in a real scenario.

As suggested in [18] we model the ephemeris, ionosphere, troposphere and multipath errors as Gauss-Markov processes. These processes have an exponential autocorrelation function with variance, σ_s^2 and time constant β; the receiver measurement noise term, is

[49]For simplicity we assume the position of the receiver coincides with the origin of the body frame of reference.

the accuracy with which the code can be tracked and is modelled as Gaussian white noise (with variance σ_r^2). A single pseudo-range measurement takes the form:

$$\begin{aligned} \rho &= b_{pr} + \sigma_r \nu_r && (34) \\ \dot{b}_{pr} &= (e^{-\beta T} - 1) b_{pr} + \sigma_s \nu_s, && (35) \end{aligned}$$

where ν_r and ν_s are unit variance Gaussian noise source and T is the time discretization interval.

The missing part of the model is the simulation of the receiver, which is formed by two main steps, defining which satellites are visible (i.e. which measurement can be used) and computing the actual solution.

The number of satellites visible by a receiver depends on the location, the time of the day, the antenna and the landscape surrounding the receiver (i.e. the obstacles). Since it is usually difficult to tease apart the contribution of such effects we prefer to use data from a representative flight test to establish what satellites are visible. At run time we initialize each receiver with a potentially slightly different set of svs[50]

Given the set of svs associated to a receiver, for each of which we have a pseudo-range measurement, simulating a new GPS position measurement is simply a matter of solving a least squares problem.

The interested reader can find the implementation of the model presented in this section in the files `GPSSpaceSegmentGM.m` and `GPSreceiver.m`.

5.7 Wind

The effects of wind and aerodynamic turbulence on a flying machine are notoriously difficult to model and even when accurate techniques based on the physical processes governing air flow are available (e.g. CFD) the level of computation involved tends to be prohibitive.

An alternative approach is to use a model that does not have knowledge of the physics governing turbulence but that reproduces only its statistical properties. According to references [1] and [9] for low altitudes[51] turbulence can be modelled as a stochastic process defined by its velocity spectra. The turbulence field is assumed to be "frozen" in time and space (i.e. time variations are statistically equivalent to distance variations in traversing the turbulence field). This assumption implies that the turbulence-induced response of the aircraft is result only of the motion of the aircraft relative to the turbulent field. Under the "frozen field" assumption, the turbulence can be modelled as a one-dimensional field that involves just the three orthogonal velocity components taken at a single point (namely the aircraft centre of gravity).

For each component the PSD of the stochastic process is defined and the turbulence can be generated using the corresponding time formulation. The von Karman and Dryden spectrum are two of the most standard form of specifying the PSD of the turbulence, we prefer the second (see equations 36) since, while more approximated, it is more easily implemented computationally.

[50]The task parameter `svs` (see listing 2) defines the ids of the satellites that are potentially visible, a random number of these defined by the platform parameter `minmaxnumsv` will be chosen and associated with the receiver.

[51]We report here the form of the model valid for altitudes lower than $1000ft$ ($304.8m$).

The Dryden spectrum takes the form:

$$\begin{aligned}\phi_{u_g}(\Omega) &= \sigma_u^2 \frac{2L_u}{\pi} \frac{1}{1+(L_u\Omega)^2} && (36)\\ \phi_{v_g}(\Omega) &= \sigma_v^2 \frac{2L_v}{\pi} \frac{1+12(L_v\Omega)^2}{\left[1+4(L_v\Omega)^2\right]^2} && (37)\\ \phi_{w_g}(\Omega) &= \sigma_w^2 \frac{2L_w}{\pi} \frac{1+12(L_w\Omega)^2}{\left[1+4(L_w\Omega)^2\right]^2} && (38)\end{aligned}$$

where L_u, L_v and L_w are the turbulence scale length and σ_u, σ_v and σ_w are their intensities. Length scales and intensities depend on altitude and mean wind magnitude u_{20} as follows:

$$\begin{aligned}L_u &= 2L_v = 2L_w = \frac{h}{(0.177+0.000823h)^{1.2}} && (39)\\ \sigma_w &= 0.1u_{20} && (40)\\ \frac{\sigma_u}{\sigma_w} &= \frac{\sigma_u}{\sigma_w} = \frac{1}{(0.177+0.000823h)^{0.4}}. && (41)\end{aligned}$$

Roughly speaking equation (41) says that the RMS intensity of the turbulence increases at lower altitude, while equation (39) express that to lower altitudes are associated shorter length scales.

Following [1] in the time domain (36)-(38) can be implemented as discrete filters:

$$\begin{aligned}u_g &= (1-a_uT)u_g + \sqrt{2a_uT}\sigma_u\nu_{n_u} && (42)\\ v_g &= (1-a_vT)v_g + \sqrt{2a_vT}\sigma_v\nu_{n_v} && (43)\\ w_g &= (1-a_wT)w_g + \sqrt{2a_wT}\sigma_w\nu_{n_w} && (44)\end{aligned}$$

where ν_{n_u}, ν_{n_v} and ν_{n_w} are unit variance Gaussian noise source and T is the time discretization interval,

$$\begin{aligned}a_u &= \frac{V}{L_u} && (45)\\ a_v &= \frac{V}{L_v} && (46)\\ a_w &= \frac{V}{L_w} && (47)\end{aligned}$$

and V is the magnitude of the platform airspeed.

The turbulence model that we just presented provides a way of simulating time varying turbulences at the level of a single quadrotor, however it is often the case that a predominant "mean" wind field is also present in the area and this affects all the platforms.

To model this in addition to the turbulence model we also have a constant velocity field[52] which is identical for all the quadrotors.

It is well known ([1]) that even in the presence of a constant wind the airspeed depends logarithmically on the distance from the ground, an effect that is commonly called wind

[52]Magnitude at 6m from the ground and direction can be defined in the task configuration parameters.

shear. Following standard practise we define the magnitude of the mean wind field by specifying its intensity at $20ft$ ($6.096m$) from the ground, the magnitude u_h at altitude h can then be obtained from :

$$u_h = u_{20}\frac{\ln(h/z_0)}{\ln(20/z_0)} \tag{48}$$

where $z0 = 0.15ft$.

The interested reader can find the implementation of the turbulence and mean wind models presented in this section in the files `AerodynamicTurbulenceMILF8785.m` and `WindConstMean.m` respectively.

5.8 Person Classifier

In the search and rescue scenario we assume the UAV platform to be equipped with a camera capable of observing the landscape in which the targets are located. Additionally we assume that each camera frame is analysed by a classifier algorithm in order to scout for persons.

Despite the latest advancement in the domain of computer vision ([10]), relying on first principles to simulate camera images for generic outdoor scenes, is still very challenging and computationally expensive. Constructing an accurate first principle model of the accuracy of an image classifier is also similarly difficult.

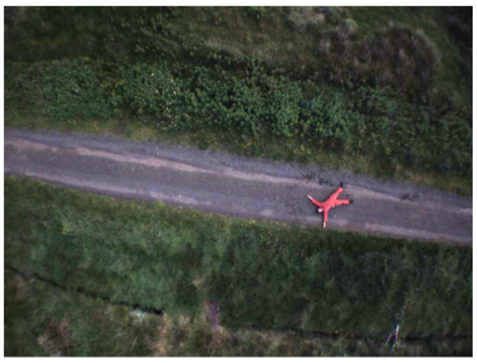
For these reasons we choose to rely on experimental data to build a probabilistic model of the joint camera and classifier system. In the first phase of devising the model we apply an off-the-shelf image classification algorithm to real world imagery collected with our quadrotor helicopters; in the second phase we devise an analytical model that relates the scores returned by the classification algorithm to the task variables that have a primary effect on the classification performance.

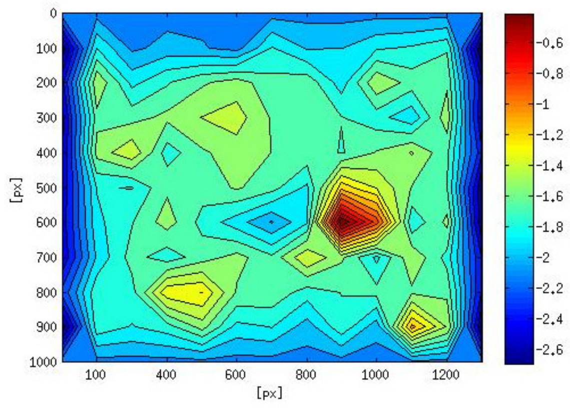
Our image dataset consists of several thousand images collected during multiple outdoor flights. Images were taken at different flight altitudes (up to $\sim 50m$) and over a variety of terrains (different types of grass, bushes, rocks). While the helicopter was flying one or more persons were lying on the ground on a variety of poses so to be visible in at least some of the images. An example of the type of images in our dataset is visible in figure 6(a).

To perform person detection we used the histogram of gradients classifier (HOG) described in [8]. We have chosen such classifier since it is effective, well know in the image detection community and more importantly because the type of gradient descriptors it uses are found at the core of many of the latest more sophisticated image classifiers. Ultimately in our simulation we are interested in modelling a real world classifier more than modelling the latest or best classifier.

We trained the classifier on a manually labelled subset of our data using a descriptor size of 100 by 100 pixels. We then proceeded by running the classifier on the reminder of our dataset. As it is commonly done in the object detection literature, images were analysed for target at different locations (windows) and at different scales. The highest scores among scales were then aggregated for each image; figure 6(b) shows the classification result for the image in figure 6(a).

The classifier model needs to be queried multiple times for each of the simulated image observation, so limiting its computational complexity is crucial to a successful integration


(a) Camera image



(b) HOG classifier scores

Figure 6: Person detection from aerial images.

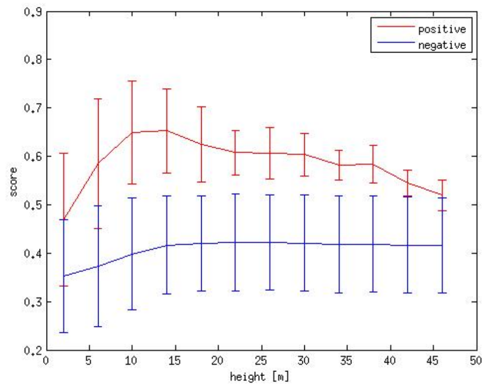
into our simulations. The UAV flying altitude has a direct effect on the size that a person in the field of view has on the camera image, therefore is the primary variable that affects the classifier ability to perform detections. For these two reasons at present we focus on modelling the dependency between the flying altitude and the score returned by the classifier. In specific we model the scores as belonging to a Gaussian distribution for which with mean and standard deviation are simple functions of height.

If we plot the scores obtained by applying the HOG classifier to our dataset against altitude and we distinguish between positive (i.e. the person was in the window considered by the classifier) and negative examples (i.e. there was no person in the window), we obtain the graph of figure 7(a).

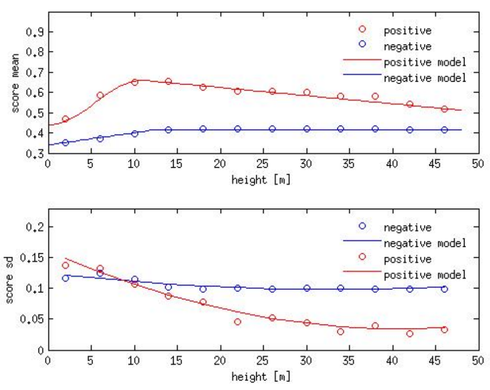
At low altitudes ($< 5m$) positive and negative example generate similar scores; at such altitudes only a part of the person is typically visible in the frame and this is not sufficient to elicit a strong classifier response since the HOG template we use is based on a whole body view of a person. As the altitude increases, positive examples are characterized by higher scores than negative ones as one would expect. At altitudes above $40m$ again the size of the person comes into play and simply the resolution is not sufficient to distinguish the person reliably from the background.

Is interesting to note also how the variation in scores decreases with altitude for the positive examples while it is almost independent from altitude for the negative examples. The latter can at least in part be explained by the multi-scale nature of the image features commonly present in a natural environment.

The relationship between scores and altitude can be easily captured and simple analytic models can be devised for both the mean and the standard deviation of positive



(a) Classifier scores as function of height, bars indicate 1σ.



(b) Mean and standard deviation score as function of height, experimental vs. model.

Figure 7: Scores as function of height.

(+) and negative (−) scores:

$$m_s^+ = \begin{cases} C_{m0}^+ + C_{m1}^+ \cos(\pi(1 + C_{m2}^+ h)) & h < h_c \\ C_{m3}^+ + C_{m4}^+(h - h_c) + C_{m5}^+ & h \geq h_c \end{cases} \tag{49}$$

$$\sigma_s^+ = C_{\sigma 0}^+ + C_{\sigma 1}^+ h + C_{\sigma 2}^+ h^2 \tag{50}$$

$$m_s^- = \begin{cases} C_{m0}^- + C_{m1}^- h & h < h_c \\ C_m^- & h \geq h_c \end{cases} \tag{51}$$

$$\sigma_s^- = C_{\sigma 0}^- + C_{\sigma 1}^- h + C_{\sigma 2}^- h^2 \tag{52}$$

where h is the UAV altitude and $C_{m0}^+, ..., C_{m5}^+, C_{\sigma 0}^+, ..., C_{\sigma 2}^+, C_{m0}^-, C_{m1}^-, C_{\sigma 0}^-, ..., C_{\sigma 2}^-$ are obtained by data fitting.

Within the simulator the UAV pose and the target locations are known, so it is possible to compute in which windows of the image frame the target will appear. Since the UAV height is known, to generate an observed score for a window we sample from a Gaussian distribution with mean and variance defined by equations 49-52. We choose the parameters for positive scores if the target appears in the window, for negative scores otherwise.

Given the sampled score, the likelihood ratio for each window is easily obtained since the positive and negative models are Gaussians distributions with known means and standard deviations. In practice for numerical reason we compute log likelihoods and therefore we return the difference between the positive and negative log likelihoods

5.9 Plume Propagation

This section gives more details about the dispersion models used for the plume modelling scenario; before starting it is useful to introduce some nomenclature:

x, y, z	coordinates w.r.t. the global NED frame of reference	m
x', y', z'	coordinates w.r.t. the wind frame of reference	m

X_s, Y_s	coordinates of the source w.r.t. the global NED frame	m
$x\prime_s, y'_s$	coordinates of the source w.r.t. the wind frame of reference	m
Q_s	emission rate of source s	Kg/s
H_s	equivalent height of source s	m
u	constant magnitude of the wind speed	m/s
S	number of sources	
a	diffusion parameter	
b	diffusion parameter	
α_w	wind direction (clockwise from north)	$rads$
I_s	total number of puff for source s	
T^i_s	time at which puff i of source s was emitted	s
Q^i_s	total amount of plume emitted by source s at time T^i	Kg
$\mathbf{\Sigma}$	Gaussian concentration covariance matrix	

Several of the following models are more easily formulated in a frame of reference with origin in the global frame and aligned with the wind direction (wind frame). We denote such coordinate with $'$ and the transformation between global and wind frame is simply:

$$x' = x\cos(\alpha_w) \quad (53)$$

$$y' = y\sin(\alpha_w) \quad (54)$$

where α_w is the wind direction.

5.9.1 Gaussian Concentration Model

As a starting point for the plume modelling scenario discussed in section 4.3 it is very useful to define a concentration model that has a small number of parameter and that is straightforward to estimate. To this aim we chose a simple and familiar three dimensional Gaussian model for the concentration c at a generic location x, y, z:

$$c(x,y,z) = \exp\left(-\frac{1}{2}\begin{bmatrix} x - X_s \\ y - Y_s \\ z - H_s \end{bmatrix}^T \mathbf{\Sigma}^{-1} \begin{bmatrix} x - X_s \\ y - Y_s \\ z - H_s \end{bmatrix}\right) \quad (55)$$

where $\mathbf{\Sigma}$ is the usual covariance matrix which define the ellipsoid dimensions and principal axes and X_s, Y_s, H_s is the source location.

In practical terms such concentration pattern would be produced by a source emitting at a constant rate in an environment where wind is not present; in such situation wind advection can be neglected and only the diffusive contribution from turbulent eddy motion in the atmosphere is considered. When specifying a covariance matrix which has eigenvectors of different magnitude we implicitly define that the diffusivity in the 3D space is non isotropic.

5.9.2 Gaussian Dispersion Models

In a more realistic situation than the one described in the previous section wind should be considered since it generally has a strong influence on the plume dispersion.

Under the following assumptions:

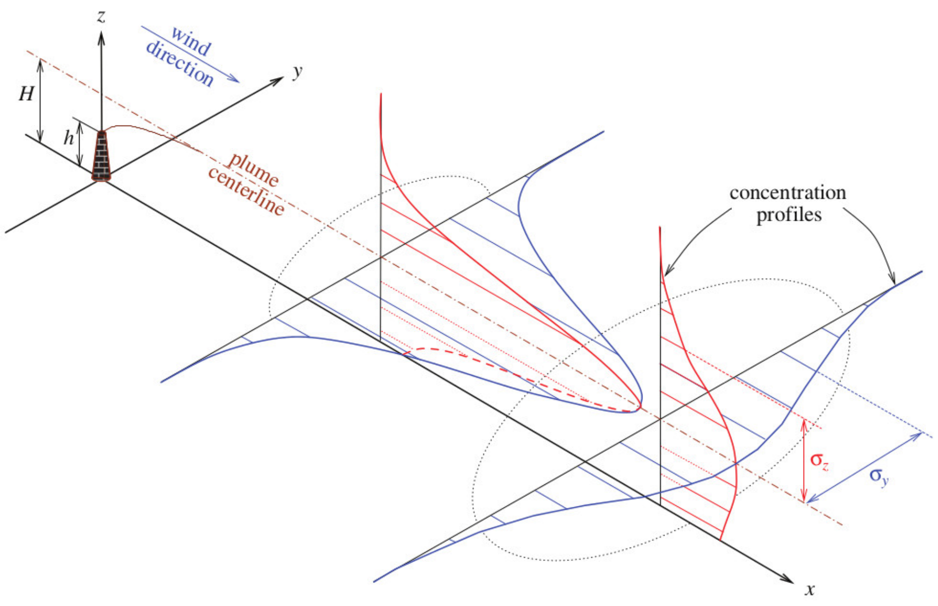


Figure 8: Gaussian plume dispersion driven by wind (from[24]).

- constant wind (with magnitude u);
- source with constant emission rate Q;
- sufficiently long time scale of interest;
- isotropic diffusion;
- negligible ground penetration;
- negligible variations in topography;

the advection-diffusion flux equation that distinguish how the plume disperse in the environment has a closed form solution and the plume concentration can be computed in closed form (see [24]). The resulting concentration has a cone-like shape starting at the plume source and expanding downwind. Profile concentrations are symmetric about the plume centerline and have the shape of a Gaussian (see figure 8).

Analytically such a concentration is more easily written in the wind frame of reference (note the use of x' and y') as:

$$\begin{aligned} c(x',y',z) =& \frac{Q}{2\pi u a(x'-X'_s)^b} \exp\left(-\frac{(y'-Y'_s)^2}{2a(x'-X'_s)^b}\right) \\ & \left[\exp\left(-\frac{(z-H_s)^2}{2a(x'-X'_s)^b}\right) + \exp\left(-\frac{(z+H_s)^2}{2a(x'-X'_s)^b}\right)\right]. \end{aligned} \tag{56}$$

where a and b are diffusion coefficients that regulate how the center line concentration evolves as the distance from the source increases.

Multiple Sources Gaussian Dispersion Model

In the case of a S number of sources the total concentration can be computed by superposition of the effects of each of the single source:

$$c(x', y', z) = \sum_{s=1}^{S} c(x', y', z; X'_s, Y'_s, H_s, Q_s). \tag{57}$$

where $c(x', y', z; X'_s, Y'_s, H_s, Q_s)$ is nothing more equation 56 in which X'_s, Y'_s, H_s is the location of source s and Q_s is its emission rate. We assume that the wind acting on each source is the same and therefore the wind frame of reference is unique.

Single Source Gaussian Puff Dispersion Model

The Gaussian dispersion model we just presented looks at the plume distribution over long time scales and assumes a constant emission rate from the source.

It is fairly common however the case in which the source emits plume intermittently, the blob-like plume puffs are blown by the wind creating a time varying plume concentration. A simple way of modelling this puff like dispersion is to assume that each puff i generated by source s has an emission time T^i_s following which it moves downwind at the wind speed u. At a location x', y', z and time t the full concentration model is simply the superposition of all the puffs generated so far:

$$\begin{aligned} c(x', y', z, t) = \sum_{i=1}^{I} \Bigg\{ & \frac{Q^i_s}{8(\pi a(x' - X'_s)^b)^{3/2}} \\ & \exp\left(-\frac{(x' - X'_s - u(t - T^i_s))^2 + (y' - Y'_s)^2}{2a(x' - X'_s)^b} \right) \\ & \left[\exp\left(-\frac{(z - H_s)^2}{2a(x' - X'_s)^b} \right) + \exp\left(-\frac{(z + H_s)^2}{2a(x' - X'_s)^b} \right) \right] \Bigg\} \end{aligned} \tag{58}$$

where a and b are the dispersion coefficients that we already encountered in equation 56 and Q^i_s is the amount of plume emitted with puff i by source s. We sample Q^i_s from a uniform distribution:

$$Q^i_s \sim \mathcal{U}(q_m, q_M). \tag{59}$$

To fully define the model, the emission times need to be specified; in our settings we sample the inter-emission time τ^i from a exponential distribution with rate λ:

$$T^i_s = T^{(i-1)}_s + \tau^i \qquad \tau^i \sim Exp(\lambda), \quad T^0_s = 0. \tag{60}$$

Multiple Sources Gaussian Puff Dispersion Model

Even in the case of a time varying dispersion model, multiple sources can be considered. The total concentration can be computed by superposition:

$$c(x', y', z, t) = \sum_{s=1}^{S} c(x', y', z, t; X'_s, Y'_s, H_s). \tag{61}$$

where $c(x', y', z, t; X'_s, Y'_s, H_s)$ is equation 58 in which X'_s, Y'_s, H_s is the location of source s and Q^i_s and T^i_s are respectively sampled from an exponential and uniform distribution (i.e. equations 59-60). Again we implicitly assume that the wind acting on each source is the same and therefore the wind frame of reference is unique.

6 Acknowledgements

The author wants to thank Guy Lever, Nicolas Hees, Simon Julier, John Showe-Taylor, David Silver, Stephen Hailes and Luke Teacy for the fruitful discussing about the application scenarios and the classifier model. This work was carried out with the support of the European Research Council #FP7-ICT-270327 (CompLACS).

References

[1] Military specification, flying qualities of piloted airplanes. MIL-F-8785C.

[2] Standard specification format guide and test procedure for single-axis laser gyros. IEEE Std 647-1995, 2006.

[3] J. Stanley Ausman. A Kalman filter mechanization for the baro-inertial vertical channel. In *Proceedings of the 47th Annual Meeting of The Institute of Navigation*, pages 153 – 159, june 1991.

[4] C. Balas. Modelling and linear control of a quadrotor. Master's thesis, School of Engineering, Cranfield University, 2007.

[5] B. Bethke, J. How, and J. Vian. Multi-uav persistent surveillance with communication constraints and health management. In *AIAA Guidance Navigation and Control Conference*, 2009.

[6] S. Bouabdallah. *Design and control of quadrotors with application to autonomous flying*. PhD thesis, EPFL, 2007.

[7] Justin David Carlson. *Mapping Large, Urban Environments with GPS-Aided SLAM*. PhD thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, August 2010.

[8] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *IEEE Conference on Computer Vision & Pattern Recognition (CVPR)*, volume 2, pages 886–893, 2005.

[9] Stacey Gage. Creating a unified graphical wind turbulence model from multiple specifications table 1 : Dryden longitudinal spectra. *Simulation*, (August), 2003.

[10] A. Handa, R. A. Newcombe, A. Angeli, and A. J. Davison. Real-time camera tracking: When is high frame-rate best? In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2012.

[11] G. M. Hoffmann, H. Huang, S. L. Waslander, and C. J. Tomlin. Quadrotor helicopter flight dynamics and control: Theory and experiment. In *Proceedings of the AIAA Guidance, Navigation and Control Conference (GNC)*, 2007.

[12] G. M. Hoffmann, S. L. Waslander, and C. J. Tomlin. Quadrotor helicopter trajectory tracking control. In *Proceedings of the AIAA Guidance, Navigation and Control Conference (GNC)*, New York, NY, USA, 2008.

[13] H. Hou and N. El-Sheimy. Inertial sensors errors modeling using Allan Variance. In *Proceeding of ION GPS/GNSS 2003*, pages 2860–2867, 2003.

[14] Ayeron Labs Inc. ayeron scout.
`http://www.aeryon.com/products.html`, March 2013.

[15] Draganfly Innovations Inc. draganflyer x6es.
`http://www.draganfly.com/uav-helicopter/draganflyer-x6es/index.php`, March 2013.

[16] Alex Kushleyev, Daniel Mellinger, and Vijay Kumar. Towards a swarm of agile micro quadrotors. *Robotics: Science and Systems*, July 2012.

[17] P. Pounds, R. Mahony, and P. Corke. Modelling and control of a quad-rotor robot. In *ACRA*, 2006.

[18] J Rankin. An error model for sensor simulation GPS and differential GPS. In *Proceedings of the Position Location and Navigation Symposium*, page 260–260. IEEE, 1994.

[19] De Nardi Renzo. *Automatic Design of Controllers for Miniature Vehicles through Automatic Modelling*. PhD thesis, University of Essex, 2010.

[20] Mac Schwager, Brian J. Julian, and Daniela Rus. Optimal coverage for multiple hovering robots with downward facing cameras. In *ICRA*, pages 3515–3522, 2009.

[21] Jaewon Seo, Jang Gyu Lee, and Chan Gook Park. Bias suppression of GPS measurement in inertial navigation system vertical channel. In *Proceedings of the Position Location and Navigation Symposium, 2004. PLANS 2004*, pages 143 – 147, april 2004.

[22] D. H. Shim, H. J. Kim, and S. Sastry. A flight control system for aerial robots: Algorithms and experiments. In *Proceedings of the IFAC Control Engineering Practice Conference*, 2003.

[23] Han Songlai and Wang Jinling. Quantization and colored noises error modeling for inertial sensors for GPS/INS integration. *IEEE Sensors Journal*, 11(6), June 2011.

[24] J. M. Stockie. The mathematics of atmospheric dispersion modeling. *SIAM Review*, 53(2):349–372, May 2011.

[25] Ascending Technologies. Pelican.
`http://www.asctec.de/asctec-pelican-3/`, March 2013.