CompLACS Quadrotors Simulator

Renzo De Nardi,
University College London
`rdenardi@cs.ucl.ac.uk`

October 1, 2012

Abstract

This document describes the multi-vehicle quadrotor simulator (QRSim) developed at UCL to devise and test algorithms for our flock autonomous helicopters, one of the three application platforms of the CompLACS project. This report is divided into two parts, the first describes the logic and the structure of the simulator while the second presents examples of use. Technical details about the dynamic models used in the simulator and their implementation are reported in the appendix.

A companion to this report 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.

Contents

1 Design and Concepts **2**
1.1 Rationale . . . 2
1.2 Concepts . . . 2
1.2.1 Platforms and Environment Objects . . . 2
1.2.2 State . . . 3
1.2.3 Steppable . . . 3
1.2.4 Task . . . 4
1.2.5 Other Abstract Classes . . . 4
1.3 The QRSim Object . . . 5

2 Installation and Use **7**
2.1 Installation . . . 7
2.1.1 Tested OS . . . 8
2.2 Creating a Task . . . 8
2.3 Example Main . . . 10
2.4 Controllers . . . 13
2.5 Multiple Simulations . . . 13
2.6 TCP Interface (Linux) . . . 13
2.6.1 Client . . . 14
2.6.2 Server . . . 14
2.6.3 Compilation and Installation . . . 15
2.6.4 Using the Client Library in your C++ code . . . 15
2.6.5 Running the Server . . . 16
2.6.6 Testing the Interface . . . 16

A Implementation Details **17**
A.1 Reference Frame and Symbols . . . 17
A.2 Quadrotor Dynamic Model . . . 18
A.3 Sensors . . . 20
A.3.1 Barometric Altimeter . . . 21
A.3.2 Orientation Estimator . . . 21
A.3.3 Accelerometer and Gyroscope . . . 22
A.3.4 GPS Model . . . 22
A.3.5 Wind Model . . . 23

Chapter 1

Design and Concepts

1.1 Rationale

UAVs provide a rich source of control problems some of which (namely the one connected with individual flight attitude control) have been successfully addressed in previous research ([**?**],[**?**],[**?**] to list a few). The CompLACS project focusses instead on the higher level problems of enabling multiple UAVs to achieve a common goal by communicating and cooperating as an autonomous flock.

A simulator that allows to devise and test algorithms aiming at such high level problems must reliably model the effects that are (at least to a certain extent) common to all platforms (e.g. wind or GPS errors) as well as the interaction between them.

At the same time, since the real platform used in this work are already provided with off-the-shelf stabilisation and attitude controllers, the requirement on accurately simulating the very low level dynamic behaviour of the platform can be relaxed. Both these sets of requirements make QRSim substantially different from existing freely available quadrotor simulators.

One of the objectives of CompLACS is to provide a generic interface to specifying learning and control problems, it is therefore important for the tools used in the initial investigation (i.e. the platforms simulators) to not be constrained to a rigid interface. For this reason in QRSim more than providing a single and rigid interface to the application platform, we decided to build a modular and extensible API so to make it possible to redefine the platform interface with relative ease.

Defining concrete application scenarios based on the available platforms and simulators is one of the forthcoming deliverable of CompLACS (milestones 6.2, 7.2), this document aims at aiding this process by providing clear description of QRSim and of its capabilities.

1.2 Concepts

We start by presenting the main conceptual blocks that constitute the simulator.

1.2.1 Platforms and Environment Objects

Within QRSim we make a 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 al 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 a sensible structure to the simulator.

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 classes. Platforms subclass the abstract class `Platform` (defined in `/qrsim/platforms/Platform.m`) and environment objects subclass `EnvironmentObject` (defined in `/qrsim/environment`).

1.2.2 State

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

A handle to a object of this type is returned during the initialisation of the simulator and can be used to reference the various objects part of the simulation. 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 objects is trivial, for instance `state.platforms1.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 2.2) 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`).

1.2.3 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

[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.

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 ODE 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 1.3, 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].

1.2.4 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 for 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 integrated as the task progresses (e.g. this could be used to include a control or state cost). Thiss 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] for defines initial states of the platforms and any other task specific object.

- `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 updates.

We will present how to create a task in section 2.2.

1.2.5 Other Abstract Classes

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

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

[4]Introduced in version 1.2.0.

[5]Introduced in version 1.2.0.

- `AerodynamicTurbulence`
- `Sensors`
- `AHARS`
- `OrientationEstimator`
- `Gyroscope`
- `Altimeter`
- `Accelerometer`. Such abstract classes are used very much like software interfaces and allow to configure specific type of sensor through the task object parameters (see section 2.2), in this way the correct class is loaded at run time.

1.3 The QRSim Object

After a brief introduction of the fundamental building blocks of QRSim we can look at how they are used within the main class of the simulator.

The object `QRSim` allows to initialize, set up and control the simulator as a whole.

`QRSim` exposes three main methods:

- `init('task_name')`, initializes the platforms the environment objects and the task according to what specified by the task `'task_name'` (see section 2.2) and 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.
- `reset()`, resets the simulator to the initial state specified by the task[6]. This is generally called after a learning episode in order to restore the state of the simulator.
- `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 1.1 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 1.2.2. Line 13 shows how if any task specific computation of the platforms controls is defined using the task method `step` (see section 1.2.4), this takes precedence to the controls `U`.
- `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.

[6]This also resets the task reward to zero.

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

Listing 1.1: QRSim step() method

```
function obj=step(obj,U)
  for j=1:obj.simState.task.dt/obj.DT,
    % update time
    obj.simState.t=obj.simState.t+obj.simState.DT;

    % step all the environment objects
    envObjs = fieldnames(obj.simState.environment);
    for i = 1:numel(envObjs)
      obj.simState.environment.(envObjs{i}).step([]);
    end

    % see if the task is the one generating the controls
    UU = obj.simState.task.step(U);
    if(~isempty(UU))
      U = UU;
    end

    % step all the platforms given U
    for i=1:length(obj.simState.platforms)
      obj.simState.platforms{i}.step(U(:,i));
    end
  end

  % update the task reward
  obj.simState.task.updateReward(U);
end
```

Chapter 2

Installation and Use

2.1 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 2.1.1 for details on the configurations tested).

To retrieve the software one can simply clone the git[1] source code repository[2] into a directory of choice:

```
$ git clone http://complacs.cs.ucl.ac.uk/git/qrsim.git
$ cd qrsim
$ git checkout -b lastStable
```

Alternatively, the `lastStable` tag of the software repository is also available as a zip archive from `http://complacs.cs.ucl.ac.uk/complacs/simulator/qrsim-lastStable.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[3]

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

You now have all that is needed to run the basic exampls `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')`[4].

[1]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/)

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

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

[4]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.

2.1.1 Tested OS

The software was tested succesfully on the following setups :

- Ubuntu 11.10 with Matlab R2010b and gcc version 4.6.1[5].
- 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[6].

2.2 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 to adjust continuously 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 2.1) returns a data structure that contains general fields like the task time step `taskparams.dt` (i.e. the timestep at which control are needed)[7] and the seed of the pseudo-random number generator (`taskparams.seed`), fields for each of the environment objects (`taskparams.environment`), and for each of the platforms (`taskparams.platforms`). The fields depend on the number and type of objects, but by convention we name:

- `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.

The number of sensors and parameters present in a platform would make defining several identical platforms quite cumbersome. For this reason all the platform parameters are specified in a single configuration file (see `'pelican_config'`); in this way the same file can be used for more than one platform.

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

The second of the mathods that is necessary to define in a task is the method `reset()`. In such method[8] the user specifies that starting state of each platform and also of any other task specific objects. This method is called after `init()` but also every time `qrsim` itself is reset; it is therefore possible for instance to have different randomly generated position for the platform at every reset of teh task.

As already explained, a task must also define the `updateReward(U)` and `reward()` functions. For the TaskKeepSpot task a straightforward update reward can be defined as a quadratic cost of the control inputs (see listing 2.3) in order to penalize large controls. An appropriate total

[5]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.

[6]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/`

[7]Note that the simulator itself uses an internal timestep of 0.02s.

[8]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.

Listing 2.1: TaskKeepSpot init() method

```
function taskparams=init(obj)
%  taskparams.dt = 1; % % task timestep
 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 %%%%
 % these need to follow the conventions of axis(), they are in m, Z down
 taskparams.environment.area.limits = [-10 20 -10 10 -20 0];
 taskparams.environment.area.type = 'BoxArea';

 % originutmcoords is the location of the RVC (our usual flying site)
 % generally when this is changed gpsspacesegment.orbitfile and
 % gpsspacesegment.svs need to be changed
 [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 gps if 0
 taskparams.environment.gpsspacesegment.dt = 0.2;
 % real satellite orbits from NASA JPL
 taskparams.environment.gpsspacesegment.orbitfile = 'ngs15992_16to17.sp3';
 % simulation start in GPS time, this needs to agree with the sp3 file above,
 % alternatively it can be set to 0 to have a random initialization
 taskparams.environment.gpsspacesegment.tStart = 0;
 % id number of visible satellites, to match the contents of orbitfile
 taskparams.environment.gpsspacesegment.svs=[3,5,6,7,13,16,18,19,20,22,24,29,31];
 % the following model was instead designed to match measurements of real data
 taskparams.environment.gpsspacesegment.type = 'GPSSpaceSegmentGM2';
 taskparams.environment.gpsspacesegment.PR_BETA2 = 4; % time constant
 taskparams.environment.gpsspacesegment.PR_BETA1 =  1.005; % time constant
 taskparams.environment.gpsspacesegment.PR_SIGMA = 0.003; % standard deviation

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

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

Listing 2.2: TaskKeepSpot reset() method

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

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 2.4). Such a function gives higher rewards to a policy that at the end of the task gets the quadrotor close to the starting point.

Listing 2.3: 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 2.4: 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
```

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

2.3 Example Main

To clarify the use of the simulator through the `QRSim` object, we shall present a simple example of use[9]; namely we will use a manually designed PID controller to achieve the behavior required by the `TaskKeepSpot` task[10].

Listing 2.5 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.

[9]Source code in the directory `example`

[10]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.

Listing 2.5: 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);
```

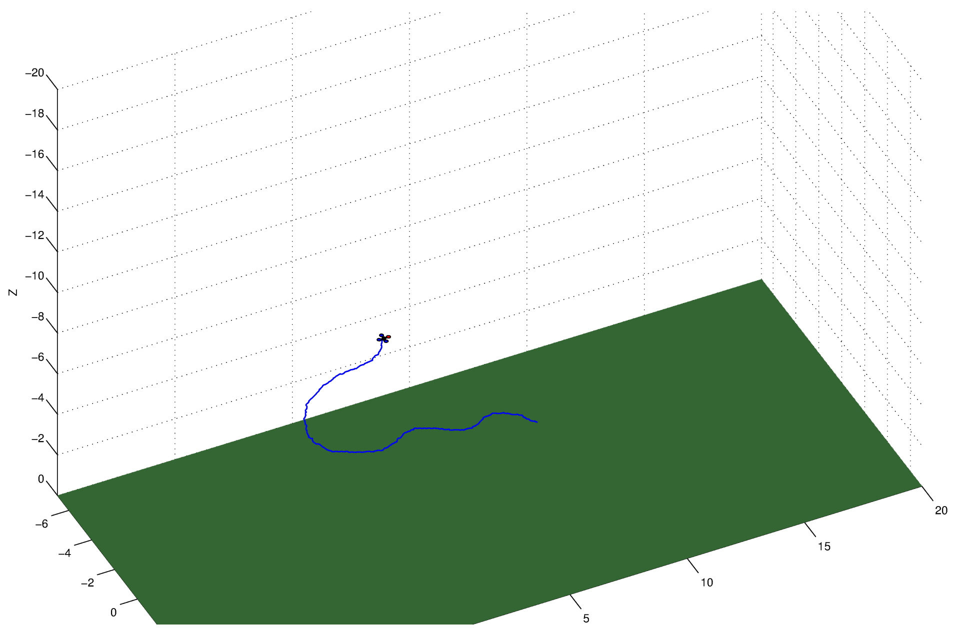

Figure 2.1: 3D visualization

The main for loop (see line 17) 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 and the throttle is the amount required to counterbalance the gravity force[11] 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 command proportionally to the current distance between the helicopter and it 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 since this is where normally the total task reward would be retrieved.

Since the task defines to run the simulation with the 3D visualization, it makes sense to run it in real time, therefore line 30 introduces the necessary pause.

Figure 2.3 shows a snapshot from the 3D visualization of the simulator where both the platform and its past trajectory are visible.

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` directly from a learning algorithm; this is perfectly fine as long as the calls are executed in the same sequence shown in listing 2.5.

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.

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

2.4 Controllers

The directory `/controllers` contain baseline implementations of simple controllers[12] that are often useful in a variety of tasks. Generally such controllers can be used to compute the set of control inputs `U` given the current state of the platform and a waypoint or 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 (NED) position in global frame while maintaining a specified heading.
- `VelocityPID` Controller that computes the pitch, roll and throttle input necessary to maintain a desired velocity in global frame while keeping a specified heading.
- `VelocityHeightPID` Controller that computes the pitch and roll input necessary to achieve a desired 2D (NE) velocity in global frame while maintaining a specified altitude and heading.
- `AnglesHeightPID` Simple controller that computes the throttle and yaw imput necessary to maintain a specified height and heading. The roll and pitch angle control are left to the user.

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

2.5 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[13].

2.6 TCP Interface (Linux)

We also provide a TCP interface to the quadrotor simulator QRSim so that it 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 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[14].

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

[13]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.

[14]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.

2.6.1 Client

The client library exposes a series of convenient methods to interact with the Matlab simulator QRSim; we now present them briefly, however we refer to the header file `QRSimTCPClient.h` and to the QRSim documentation for mare 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 PID in order to reach the specified 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 PID in order to achieve and maintain the velocity command specified.
- `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 command so it is directly passed to the dynamic model of the quadrotors.

2.6.2 Server

The server code is constituted by a Matlab function (`qrsim/tcp-linux/matlab/QRSimTCPServer.m`) that takes care of setting up and executing command in the quadrotor simulator and a Java class (`qrsim/tcp-linux/java/qrsimsrvcli/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[15] and installed (see section 2.6.3 and 2.6.3) on the machine that will run the quadrotor simulation.

2.6.3 Compilation and Installation

The current version of the TCP interface only supports Linux[16], in the following we assume to be working on one of such systems.

Dependencies

The C++ client uses Google protocol buffers[17] for data serialization and the build process uses CMake[18], so both of these tools needs to be available in your Linux machine.

On Ubuntu 12.04 these can be installed as follows[19]:

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

Compilation

To compile the server and client code cd into `qrsim/tcp-linux/build`[20] and run:

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

all should compile cleanly.

CMake is smart enough to figure out if you do not have Java installed in your machine in which case only the C++ client library will be compiled. Note that this is a legitimate scenario since you might be interested in connecting to a remote server which takes care of running QRSim.

Installation

Installing[21] the library is nothing different from any usual Linux library compiled from source:

```
$ sudo make install
```

2.6.4 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 methods

[15]Google protocol buffer serialization tightly depends on the version of the protocol buffer library so shipping a precompiled server library is not a viable option.

[16]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.

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

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

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

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

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

- link against libqrsim `libqrsimtcpclient.so`.

The files `qrsim/tcp-linux/src/exampleclient.cpp` and `qrsim/tcp-linux/src/testclient.cpp` provide two small examples of using the client library.

2.6.5 Running the Server

The server code runs in Matlab; from the Matlab console cd into the directory `qrsim/tcp-linux/matlab` 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.

2.6.6 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:

```
$ ./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]
```

Appendix A

Implementation Details

This appendix describes the models implemented in QRSim in order to replicate a typical quadrotor platform (namely the Ascending Technologies Pelican [?] in use at UCL) complete of its sensors. We primarily aim at giving a basic explanation of our implementation choices to the non expert, for more in depth treatment we refer to the publications referred in each of the following sections.

A.1 Reference Frame and Symbols

For convenience we show the body frame of reference used in many of the computation involving the helicopter dynamics (see figure A.1).

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.

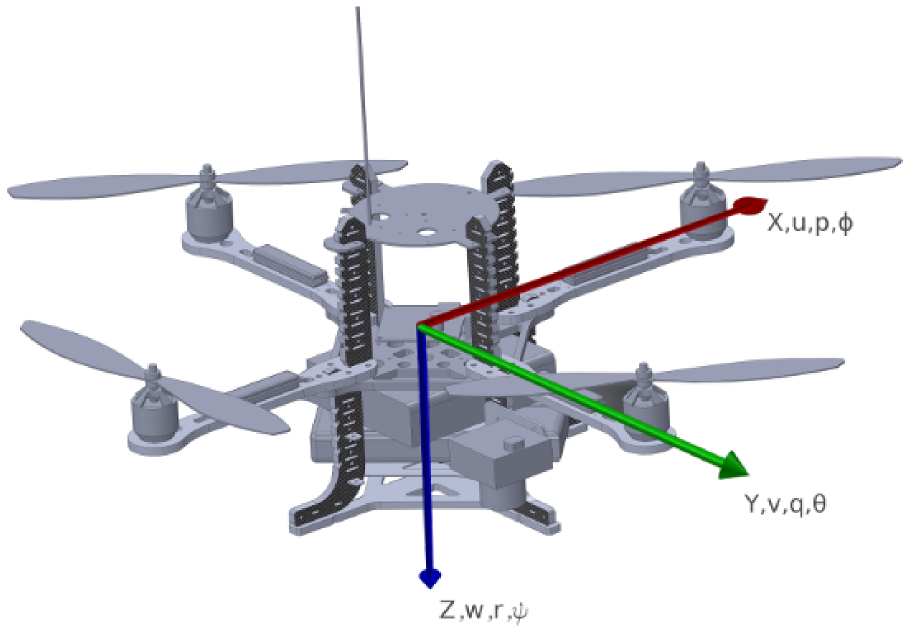

Figure A.1: Body NED frame of reference and state variables.

We also define first several of the variable that are used in the simulator:

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

p_x	x position (NED coordinates)	m
p_y	y position (NED coordinates)	m
p_z	z position (NED coordinates)	m
ϕ	roll attitude in Euler angles right-hand ZYX convention	rad
θ	pitch attitude in Euler angles right-hand ZYX convention	rad
ψ	yaw attitude in Euler angles right-hand ZYX convention	rad
u	linear velocity along x body axis	m/s
v	linear velocity along y body axis	m/s
w	linear velocity along z body axis	m/s
p	rotational velocity around x body axis	rad/s
q	rotational velocity around y body axis	rad/s
r	rotational velocity around z body axis	rad/s
F_{th}	thrust force	N

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

$\tilde{p}_x$	x position estimate from GPS (NED coordinates)	m
$\tilde{p}_y$	y position estimate from GPS (NED coordinates)	m
$\tilde{p}_z$	z position estimate from GPS (NED coordinates)	m
$\tilde{\phi}$	roll attitude in Euler angles right-hand ZYX convention	rad
$\tilde{\theta}$	pitch attitude in Euler angles right-hand ZYX convention	rad
$\tilde{\psi}$	yaw attitude in Euler angles right-hand ZYX convention	rad
$\tilde{p}$	rotational velocity around x body axis from gyro	rad/s
$\tilde{q}$	rotational velocity around y body axis from gyro	rad/s
$\tilde{r}$	rotational velocity around z body axis from gyro	rad/s
$\tilde{a}_x$	linear acceleration in x body axis from accelerometer	m/s^2
$\tilde{a}_y$	linear acceleration in y body axis from accelerometer	m/s^2
$\tilde{a}_z$	linear acceleration in z body axis from accelerometer	m/s^2
h	altitude[1]from altimeter NED	m
$\dot{p}_x$	x velocity from GPS (NED coordinates)	m/s
$\dot{p}_y$	y velocity from GPS (NED coordinates)	m/s
$\dot{h}$	altitude rate from altimeter NED	m/s

- **Platform controls** U $= [u_{pt}; u_{rl}; u_{th}; u_{ya}; V_b]$

u_{pt}	pitch command	rad
u_{rl}	roll command	rad
u_{th}	throttle command	unitless
u_{ya}	yaw command	rad/s
V_b	battery voltage	V

A.2 Quadrotor Dynamic Model

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

[1]It is worth noting that $h = -p_z$ therefore as result of a climb maneuver h increases but p_z decreases.

These type 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 the low level dynamics of the quadrotor is not our primary interest (as explained in section 1.1), 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.

Rotational Dynamics

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 motors and propellers, learning it ensures we tch the characteristics of our platform.

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 ensure to not exceed the sensing and/or control envelope).

We encoded these findings in equations A.1 and A.2 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.

$$\dot{p} = \begin{cases} K_{pq1}(K_{pq0}u_{rl} - \phi) + pK_{pq2} & |p| < p_{max} \\ 0 & |p| \geq p_{max} \end{cases} \tag{A.1}$$

$$\dot{q} = \begin{cases} K_{pq1}(K_{pq0}u_{pt} - \phi) + qK_{pq2} & |q| < q_{max} \\ 0 & |q| \geq q_{max} \end{cases} \tag{A.2}$$

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 A.3.

$$\dot{r} = K_{r0}u_{ya} + rK_{r1}. \tag{A.3}$$

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:

$$F_T = C_{th0} + C_{th1}u_{th} + C_{th2}u_{th}^2 \tag{A.4}$$

this is a typical result ([?],[?]) 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:

$$F_M = C_{vb0} + C_{vb1} V_b. \tag{A.5}$$

As expected due to inertia and drag effects, dynamic tests showed that the thrust can not change instantaneously, but instead it changes at a constant rate[2] 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 (A.4) and (A.4) we obtain:

$$\dot{F}_{th} = \begin{cases} -\tau_1 & \max(F_T, F_M) < F_{th} \\ \tau_1 & \max(F_T, F_M) \leq F_{th}. \end{cases} \tag{A.6}$$

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:

$$\dot{u} = -qw + rv + g_{b_x} + K_{uv}(u - u_w) \tag{A.7}$$

$$\dot{v} = -ru + pw + g_{b_y} + K_{uv}(v - v_w) \tag{A.8}$$

$$\dot{w} = -pv + qu + g_{b_z} - \frac{F_{th}}{m} + K_w(w - w_w). \tag{A.9}$$

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.

Equations (A.1)-(A.3) and (A.7)-(A.9) 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`.

A.3 Sensors

On board a real quadrotor platform a series of sensors allows to produce an estimate of the variables that are needed to control and stabilize the helicopter. A standard suite of sensors (e.g. the ones present on a Pelican) contains a try-axial accelerometer, a set of three gyros, a set of three magnetometers, a pressure sensor and a GPS receiver.

Roughly speaking gyros, accelerometers and magnetometers are combined to estimate the platform attitude (i.e. the angles $\tilde{\phi}, \tilde{\theta}, \tilde{\psi}$), the GPS is used to determine the position coordinates $\tilde{p}_x$ and $\tilde{p}_y$ and their rate of change $\dot{p}_x$ and $\dot{p}_y$, and the pressure sensor (combined with the accelerometer) provides altitude (h) and its rate of change ($\dot{h}$).

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

A.3.1 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 decimeters making them more accurate than a GPS receiver especially when coupled with other sensors.

The main drawback of an altimeter based on air pressure 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 (from the propellers). Following [?] and [?] we model the altimeter measurements as follows:

$$h = -p_z + b_h + \nu_{h_m} \tag{A.10}$$

$$\dot{b} = -\frac{1}{\tau_b} b + \nu_{h_c} \tag{A.11}$$

where h is the of the altimeter, $-p_z$ is true altitude, b is the bias noise which is modelled as first order Gauss-Markov process driven by the white Gaussian noise ν_{h_c} and ν_{h_m} is white Gaussian measurement noise. τ_b is the time constant of the Gauss-Markov process and allows for adjusting the time scale over which the correlation is expressed. 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`.

A.3.2 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 to estimate and compensate 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 allow to replicate the desired time correlation of the attitude estimates while still being a simple model. The resulting noise model is the following:

$$\tilde{\phi} = \phi + b_\phi \tag{A.12}$$

$$\tilde{\theta} = \theta + b_\theta \tag{A.13}$$

$$\tilde{\psi} = \psi + b_\psi \tag{A.14}$$

$$\dot{b}_\phi = -\lambda_\phi b_\phi + \nu_{b_\phi} \tag{A.15}$$

$$\dot{b}_\theta = -\lambda_\theta b_\theta + \nu_{b_\theta} \tag{A.16}$$

$$\dot{b}_\psi = -\lambda_\psi b_\psi + \nu_{b_\psi} \tag{A.17}$$

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`.

A.3.3 Accelerometer and Gyroscope

Several authors have studied the types of errors that commonly affects accelerometers and gyros ([**?**][**?**]) and very accurate, albeit complex, representation of such noise models have been standardised [**?**]. However since in our platform the low level stabilization is performed on board and is not of interest for the type of activity considered in CompLACS, such type of models are not justified by our requirements. At this stage we preferred 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:

$$\begin{aligned} \tilde{p} &= p + \nu_p \quad (A.18) \\ \tilde{q} &= q + \nu_q \quad (A.19) \\ \tilde{r} &= r + \nu_r \quad (A.20) \end{aligned}$$

$$\begin{aligned} \tilde{a}_x &= a_x + \nu_x \quad (A.21) \\ \tilde{a}_y &= a_y + \nu_y \quad (A.22) \\ \tilde{a}_z &= a_z + \nu_z \quad (A.23) \end{aligned}$$

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`.

A.3.4 GPS Model

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 ([**?**], [**?**]). 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 an `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 [**?**] the pseudo-range measurements (ρ) can be written as:

$$\rho = r + \delta_{eph} + \delta_{iono} + \delta_{tropo} - \delta_{clock} + \delta_{mp} + \nu_{rcvr}, \quad (A.24)$$

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[3] 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 2.1), therefore as the simulation time progresses the position of the satellites changes as it would in a real scenario.

As suggested in [?] 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 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:

$$\begin{aligned} \rho &= b_{pr} + \sigma_r \nu_r && \text{(A.25)} \\ \dot{b}_{pr} &= (e^{-\beta T} - 1) b_{pr} + \sigma_s \nu_s, && \text{(A.26)} \end{aligned}$$

wher 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[4]

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`.

A.3.5 Wind Model

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 tends to be prohibitive.

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

[4]The task parameter `svs` (see listing 2.1) 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.

An alternative approach is to use a model that does have knowledge of the physics governing turbulence but that reproduces only its statistical properties. According to references [?] and [?] for low altitudes[5] 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 responses 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 A.27) since, while more approximated, it is more easily implemented computationally.

The Dryden spectrum takes the form:

$$\phi_{u_g}(\Omega) = \sigma_u^2 \frac{2L_u}{\pi} \frac{1}{1+(L_u\Omega)^2} \tag{A.27}$$

$$\phi_{v_g}(\Omega) = \sigma_v^2 \frac{2L_v}{\pi} \frac{1+12(L_v\Omega)^2}{[1+4(L_v\Omega)^2]^2} \tag{A.28}$$

$$\phi_{w_g}(\Omega) = \sigma_w^2 \frac{2L_w}{\pi} \frac{1+12(L_w\Omega)^2}{[1+4(L_w\Omega)^2]^2} \tag{A.29}$$

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 depends on altitude and mean wind magnitude u_{20} as follows:

$$L_u = 2L_v = 2L_w = \frac{h}{(0.177+0.000823h)^{1.2}} \tag{A.30}$$

$$\sigma_w = 0.1u_{20} \tag{A.31}$$

$$\frac{\sigma_u}{\sigma_w} = \frac{\sigma_u}{\sigma_w} = \frac{1}{(0.177+0.000823h)^{0.4}}. \tag{A.32}$$

Roughly speaking equation (A.32) says that the RMS intensity of the turbulence increases at lower altitude, while equation (A.30) express that to lower altitudes are associated shorter length scales.

Following [?] in the time domain (A.27)-(A.29) can be implemented as discrete filters:

$$u_g = (1-a_uT)u_g + \sqrt{2a_uT}\sigma_u\nu_{n_u} \tag{A.33}$$

$$v_g = (1-a_vT)v_g + \sqrt{2a_vT}\sigma_v\nu_{n_v} \tag{A.34}$$

$$w_g = (1-a_wT)w_g + \sqrt{2a_wT}\sigma_w\nu_{n_w} \tag{A.35}$$

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

$$a_u = \frac{V}{L_u} \tag{A.36}$$

$$a_v = \frac{V}{L_v} \tag{A.37}$$

$$a_w = \frac{V}{L_w} \tag{A.38}$$

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

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 vector[6] which is identical for all the quadrotors.

It is well known ([**?**]) 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 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 :

$$u_h = u_{20} \frac{\ln(h/z_0)}{\ln(20/z_0)} \tag{A.39}$$

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.

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