Exercise 0:
Installing qrsim and testing the installation

- download qrsim from
  `http://complacs.cs.ucl.ac.uk/complacs/simulator/qrsim-lastStable.zip`
- unzip the archive in a directory of your choice
- start matlab and navigate to the directory in which the qrsim archive was unpacked
- In the Matlab console, add the sim directory to path and run example/main.m

```
>> addpath sim
>> cd example
>> main
```

If the simulator is working correctly a matlab figure (see figure) should appear and you should see a helicopter moving about. In this task the helicopter is attempting to maintain the position it had at the start of the run, due to air turbulnce and estimation error the uav tends to drift and therefore corrective actions are needed from time to time.

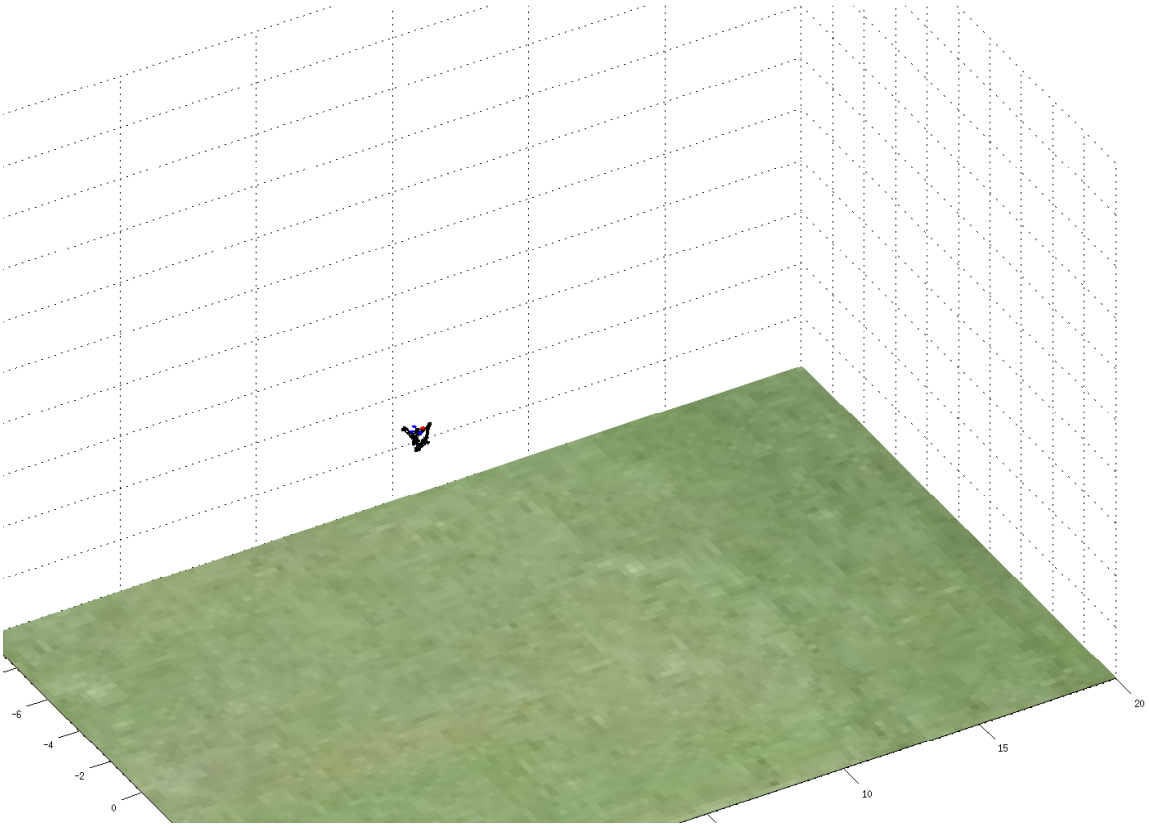

Figure 1: Typical main output figure

Exercise 1:
Creating a new project

A new qrsim simulation is usually composed of at least three files:

- a main scripts that control the execution of the simulation;
- a task file that define the configuaration for the environment objects;
- a platform configuration file that defines the setting for the platform sensors.

Create a new directory in a location of your choice[1] and copy into it the files:

- `example/main.m`
- `sim/tasks/TaskKeepSpot.m`
- `sim/platforms/pelican_config.m`

Since we will be modifying these files in the following exercises it makes sense to rename them to something more meaningful; let's rename `TaskKeepSpot.m` to `TaskTutorial.m` and `pelican_config.m` to `pelican_config_tutorial.m`.

Of course now we need to update the files that reference souch scripts, this will give us the chance to see what references what, within one of our projects. We start by editing `main.m`:

- line 8: update `addpath('../sim:../controllers');` to point to the right directories in your machine. Note that is is the only thing needed to make use of qrsim in your project.
- line 14: update `state = qrsim.init('TaskKeepSpot');` to reference the renamed task `TaskTutorial.m`. Note that this is the only location in which a task is referenced.

Now, since `TaskTutorial.m` is a Matlab class, its filename needs to match the class name[2], we therefore need open `TaskTutorial.m` in the file editor and do the following:

- line 1: replace `classdef TaskKeepSpot<Task` with `classdef TaskTutorial<Task`
- line 28: replace `obj = TaskKeepSpot(state)` with `obj = TaskTutorial(state)`.

Finally if we look at the bottom of the method `init()` defined in `TaskTutorial.m` (line 102) we can see that the script `pelican_config` is indicated as the configuration file that should be used for platform 1. Obviously we want to change this to `pelican_config_tutorial` so that the configuration file that we have in the current project is used instead.

At this point, we are done with creating a new project. You can now test that the modifications we just applied did not break the project; simply run `main.m` and you should see the same output you got in Exercise 0.

[1] Using qrsim within a project only requires to have the `sim` directory in the class path therefore you have complete freedom of choosing where to keep your projects.

[2] If you are not familiar with object-oriented programming in Matlab you can find more informations here: `http://www.mathworks.co.uk/help/techdoc/matlab_oop/ug_intropage.html`

Exercise 2:
Familiarise with `QRSim` and platforms objects

We now have a look in more detail at the file `main.m`; open it in your matlab editor. This script implements what is one of the simplest way of running a simulation using qrsim; nevertheless it shows the three foundamental steps that are always needed when running a simulation:

- line 11: `qrsim = QRSim()` creation of a simulation object;
  this instruction takes care of initializing the path variables correctly and creates (but does not initialize) the data structure that maintains the simulation state.
- line 14: `state = qrsim.init('TaskTutorial')` initialisation of the simulation;
  the setting specified in the task are used to instantiate the required platform and environment objects, and subsequently to initialised them to the required state. After calling `init` the simulator is in a valid state and the simulation time `state.t` is equal to zero.
- line 35: `qrsim.step(U)` stepping the simulator;
  this command first updates the enviroment objects and then applies to each platform the given input `U` (where `U` is a matrix with a number of columns equal to the number of platforms[3]). At completion the simulation time had advanced of `state.DT`.

From the description just given one can understand that calling `step` only makes sense after the simulation is created and initialized to a valid state. In our example the simulator is stepped within a for loop since we want to run the task for `N` steps but is rather obvious that this does not need to be the case; once the simulation object is initialized the step method can be called in whatever fashion is more convenient to the user.

Now that is more clear what the `main.m` script is doing, modify it in order to produce a time plot of the true and of the estimated platform position. Use a 3 by 1 subplot so that you can plot each of the three (x,y,z) components independently. In a real platform the altitude estimated using the barometric altimeter is often more accurate than the one produced by the GPS receiver for this reason you will plot the former instead of p_x in your plots. Once you obtained the plots have a good look at the true and estimated variables, what do you notice?

The following methods defined in the class `Pelican` (accessible as `state.platforms1.getX()`) will be useful to carry out the above:

- `getEX()` returns the estimated state (noisy) state;
- `getX()` returns the true state (noiseless) state;
- `getEXasX()` returns the estimated state (noisy) formatted as the noiseless state;
- `isValid()` returns true if the state is valid; the state can become not valid if the uav exits the flying area (e.g. hits the ground) or if two uavs collide;

use the command `doc Pelican` to have more details about the format of the returned data[4].

[3] Use `doc Pelican` to see details about meaning and ranges of the contrul inputs.

[4] The commands `getEX(), getX()` and `getEXasX()` all accept intervals as argument (e.g. `getEX(1:3)`); this provides an easy way to query only for the variables of interest.

Note: in order to update the figure containing the 3D visualization the simulator needs to redefine the current window, this might interfere with your plot. In order to avoid that you can store the handles of your plots before the beginning of the for loop and then update the plot inside the loop by setting the `YData` property of each plot.

Exercise 3:
Modify task and platform settings

So far we kept unchanged the settings of both the environment and of the platforms in our task, in this exercise we will explore the effects of changing such parameters.

Open the file `TaskTutorial.m` and look at the function `init()`, you will see that the parameters are divided into sections corresponding to the various environment objects. Also note how we use a nested naming scheme in order to specify the properties of each object. Most of the objects have the property `on` which allow to enable or disable the module in question and some have a parameter `dt` that defines the update rate.

Try out some of the following changes in turn, and look at the resulting behaviour of the helicopter (remember to save the task file before re-running `main.m`):

- set `taskparams.seed` to a number different from 0 and run the simulation twice. What do you notice about the flight path of the helicopter?
- set `taskparams.display3d.on` to 0; the simulation will run withoud 3D display. Did you notice any change in simulation speed?
- change the values of `taskparams.environment.area.limits`, you will see the 3D area changing accordingly[5].
- try to set `taskparams.environment.gpsspacesegment.on` to 0, what happens? We will see later why this is the case.
- set `taskparams.environment.wind.on` to 1, and run the simulation twice with different values of `taskparams.environment.wind.direction`. You should see a clear effect on the helicopter flight pattern.
- experiment with different initial values for the platform state `taskparams.platforms(1).X`, note that you can also define the initial velocities by specifying an initial state of lenght 12.

Let's now continue by experimenting with the platform settings and therefore editing the file `pelican_config_tutorial.m`. Immediately one can see that the file uses the same naming convention that we have seen for the environment object parameters. Even in this case is intructive to make changes to the setting and compare the output of the simulator. We suggest to try some of the following:

[5] If you change the area so that the starting position of the helicopter is outside the flying volume the simulation won't run!

- set `sensors.gpsreceiver.on` to 0 and run the simulation; the GPS receiver noise is now off. What happens to the plots of estimated and true position that we set up in Excercise 2?
- if you now go back to `TaskTutorial` and retry setting `environment.gpsspacesegment.on` to 0; you should now be able to run the simulations without any error.
- set `sensors.ahars.on` to 0 to turn off any noise in the inertial sensors; this should be clearly visible if you plot true and estimated angles (or angular velocities).
- set `aerodynamicturbulence.on` to 0; is the flight behaviour changed?
- set `aerodynamicturbulence.on` and `sensors.gpsreceiver.on` both to 0; do you see a drastic improvement in the capability of the helicopter in keeping station?

Exercise 4:
Set and reset the platform state

We have seen how changing the initial platform state in the `pelican_config_tutorial.m` is very straightforward but is often necessary to be able to save, set and reset the platform state at runtime.

set / reset state

Exercise 5:
Using predifined PID controllers

controllers

Exercise 6:
Defining a task reward

A task is defined not only by the configuration of platforms and environment objects but also by the objective that needs to be achieved.

Is important to point out that while defining rewards within a task is very handy, this is by no means something necessary to perform simulations using qrsim. In facts the `updateReward()` and `reward()` methods of a task can simply be left empty if not needed.

Exercise 7:
Working with several helicopters

Up until now we only considered one single helicopter, however qrsim can readily deal with more than one platform. Let's try to modify the task file and `main.m` to handle 10 different

helicopters; to do so you will need to:

- modify the platforms section of `TaskTutorial.m` to define `configfile` and initial state `X` for each of the platform. You can easily do this with a for loop but make sure that the initial position of the platform are sufficiently far apart otherwise they would be deemed in collision.
- modify `main.m` so that ten different waypoints and ten different pid controller are created instead of only one of each. In the for loop use each of the ten pids to compute the control input for the respective helicopter. Remember to bundle up the control commands of each platform as columns of the matrix `U` so you can pass it to the step method of qrsim.

If you get stuck you can have a look in the directory `example`, the file `main10.m` shows a possible solution to this excercise.

Now that you have several helicopters in your simulation you can play around with the parameter `collisionDistance` in `pelican_config_tutorial.m` and verify that if two paltforms are closser than the collision distance both will be deemed had having an invalid state.