Implementing the LPS-ROS system

The ROS module for positioning with the LPS is available on Github: https://github.com/bitcraze/lps-ros. It is developed and tested on ROS Kinetic running on Ubuntu 16.04. As of now ROS is the only software environment in which the system has been tested.

To get started from a newly installed Ubuntu 16.04:

1. Install ROS (full desktop): http://wiki.ros.org/kinetic/Installation/Ubuntu
2. Install the ros joystick node with
   `sudo apt-get install ros-kinetic-joy`
3. Create your workspace: http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment#Create_a_ROS_Workspace
4. Clone https://github.com/bitcraze/lps-ros and https://github.com/whoenig/crazyflie_ros in the workspace src folder
5. Run *catkin_make* to build the packages
6. Source the workspace *devel/setup.bash* again
7. Run *rosdep install bitcraze_lps_estimator* to install dependencies

The positions of the anchors has to be entered in a configuration file. Create a file at *catkin_ws/src/lps-ros/data/anchor_pos.yaml* and enter your anchor coordinates. An example is present in the data folder:

anchor_pos.yaml

```
n_anchors: 6
anchor0_pos: [  0,   0, 1.85]
anchor1_pos: [  0,   2, 1.85]
anchor2_pos: [  4,   2, 1.85]
anchor3_pos: [  4,   0, 1.85]
anchor4_pos: [1.3,   1, 1.85]
anchor5_pos: [2.6,   1, 1.85]
```

The positions are in meters, [x, y, z], from the origin.

When this file is configured, connect an XBox 360 (or compatible) gamepad, start the Crazyflie 2.0 and launch:

```
$ roslaunch bitcraze_lps_estimator dwm_loc_ekf_hover.launch
uri:=radio://0/80/250K x:=3 y:=2.3 z:=1.0
```

Change the URI with your copter radio configuration. You will see the anchor position in RED so that you can verify there position make sense. The Crazyflie position will be shown in green, before flying you can verify that the position is roughly correct.

The X,Y,Z parameter will be the hovering point, make sure they make sense in your setup and it is better to place the Crazyflie just under this first setpoint to start with. The gamepad mapping is (this is for an X-Box gamepad):

Button	Action
X	Take-off
A	Land
B	Emergency stop (this is the red button ;)
Left joystick	X/Y position
Right joystick	Altitude/Yaw setpoint

Note that the YAW setpoint is not handled yet.

At first we were unsure what the correct system setup was, so it took some time to find out what the previous group had used to get the system working previously. We figured out that it was the LPS-ROS system, which led us to these instructions. Because we had already built another system, we reinstalled the computer completely to get a fresh start at tackling these instructions.

We followed all instructions and were unable to get the system beyond step 5: "Run *catkin_make* to build the packages". After running the catkin_make command, some packages were not found, so I began to manually install the packages that were missing. Even after installing some of the packages manually, we found ourselves in the same place. We consistently got messages stating "invoking catkin_make failed". Which is why we had to make the decision to go with a simulator to simulate the actions of the real system.

Link to instructions:
https://wiki.bitcraze.io/doc:lps:ros

User Manual:

Initialization

1. Start 4 terminals in the directory home/catkin_ws
2. Put each required file into the directory home/catkin_ws/src/beginner_tutorials/scripts
3. Navigate to the home/catkin_ws/src/beginner_tutorials/scripts directory in a terminal and use the command "`chmod 777 *.py`" in order to set permissions allowing for catkin to use each of the files
4. Navigate to the home/catkin_ws directory
5. In each terminal, initialize the workspace with the command `source "./devel/setup.bash"`
6. Using the command "`rosrun beginner_tutorials *file*.py`" run each of the below commands in one of the terminals
    - backend2.py
    - visros.py
    - simulator.py
    - Monitor.py

    You should now see an empty visualization of the flight area, a text window with an end flight button, and a GUI for entering flight information.

Entering flight information

After all windows are open, you can move on to entering flight information into the GUI.

Flight Paths Section

This section is used to add flight path files to the flight routine. When upload is clicked, it will open a file browser from which the desired flight path file can be chosen.

Flight Path Format

- Must be a python script
- Can have any name
- Must contain a function named "flightPath" that takes in a list of all drone locations associated with the drones assigned to that flight path
- Must return a list of destinations, one for each of the drones associated with that flight path

Drones Section

This section is used to add drones to the flight routine. Drones can then either be assigned a list of destinations to fly to in sequence or be assigned to one of the imported flight paths.

In order to add a drone, enter a name and id for the drone. Then select either coordinates or flight path in the drop down.

If coordinates is chosen, enter each coordinate in sequence in the format of “X,Y,Z”. Click add for each coordinate and once all coordinates are added, click done.

If flight path is chosen, select the flight path you wish to associate that drone with and click done.

Obstacles Section

This section is used to add obstacles to the flight area.

To add an obstacle, add a name and two points to define the shape. Obstacles are defined by two opposite points on a rectangular prism. These prisms are made to be on a perfect 90 degree orientation. From the two opposite points the other six points that make up the shape are calculated.

Visualization Configuration File

This section gives options for customizing the visualization display. By selecting various options you can turn these portions of the visualization on or off.

<u>Visualization</u>

Controls the visualization as a whole.

<u>Drone Locations</u>

Controls the display of the icons representing the current location of each drone in the routine.

<u>Drone Paths Expected</u>

This displays the expected flight path of each drone in the flight routine.

<u>Drone Paths Flown</u>

Controls the display of a trail left behind each drone as it flies through its own path.

Sensors

Controls the display of the 8 sensors at each corner of the flight area. Does not affect presence of boundary for drones flying out of bounds.

Objects

Controls the display of all obstacles added the flight area. Does not affect actual presence of the obstacles for purposes of collision.

Logging Configuration File

This section gives options for customizing the logging to file performed during the flight. By selecting various options you can turn these portions of the logging on or off.

Logging

Controls the logging as a whole.

Drones Flown

Controls the logging of which drones are used in the flight.

Drone Locations

Controls the logging of drone locations during the flight. The drone locations will be logged periodically by reporting each drone and its most recent reported location.

Frequency

Controls the frequency at which drone locations are logged to file. Currently tied to the refresh rate of the simulator.

Objects

Controls the logging of the simulated obstacles present during the flight.

Flight Paths

Controls the logging of the flight paths imported into the flight path

Coordinates

Controls the logging of the drone flight paths entered by the user

Major Events

Controls the logging of major events. Major events currently include drones flying out of bounds, drones flying too close to one another, and drones flying within the boundaries of obstacles.

Operation

Ensure that all windows are open and all relevant flight information has been entered. Then click the “Start Flight” button and pay attention to the visualization.

To stop the flight prematurally, click on the end flight button on the monitor window. This will tell each drone to fly to their initial positions.

When all flight have stopped, run the ctrl^c command in each terminal to end each process.

Known Shortcomings

The monitor node currently does not accept events. The monitor node must be multithreaded to accept new input.

The system currently does not end after the flight path files have ended their routines, or when the manually inputted flight paths run out of coordinates. If the end flight button is pressed or a drone error occurs, the system does not end when all drones have returned to their initial positions.

Sometimes packet loss occurs between different ROS nodes. This can result in needing to press the end flight button numerous times, some drone coordinates being lost or skipped, and rarely the initial setup information being lost. Measures have been taken to assure the initial information is sent properly.

If the need to switch to Python 3 ever arises, the code should mostly transfer. Some print statements will need to be modified to fit Python 3 standards.

The expected flight path is currently not implemented.

Troubleshooting

Our project does have some potential issues. This section will cover these issues and how they can be minimized or solved.

Packet Loss within ROS

Occasionally, initial flight information from the backend does not reach the simulator or the visualization correctly. This results in either an error ending that run of the simulator or failing to update the locations of the obstacles or sensors respectively.

We minimized the incidence of this error through rewriting the backend and have not experienced the error since. However, because of the unknown nature of this error's cause, we elected to include it in this section despite our belief that it is largely if not entirely fixed.

Our only known solution is to restart all sections of the project that may have been altered by an occurance of the error. This includes visros.py, backend2.py, and simulator.py. We believe this error to be largely fixed but it may reoccur.

Incorrect Order of Node Initialization

The nodes of the project must be started in a particular order or packets sent between various nodes will not arrive on time. There is a certain degree of leeway in this order. In general, the backend needs to be started last as it starts and manages all communications between the nodes in the system. By starting the backend last we ensure that all other nodes are in place to receive information when it is started.