EFLS - Manual

UAV Emergency Forced Landing System

UAV during EFLS landing at a selected safe landing site

Table of Contents

1 Purpose of this document

This document will explain in detail all aspects of the project that has been undertaken. The following points are covered specifically in this document:

- Why the 3-Phase approach was decided.
- Why development of Phase 1 is important.
- Installing EFLS.
- Parameter setting of EFLS.
- EFLS performance and its benefit towards the 3-Phase approach.
- Future development of EFLS

2 Introduction

Unmanned Aerial Vehicles (UAVs) have been increasing in popularity in the military, commercial, and private markets over the past decades. The broad adoption of autonomous technology to a variety of flying platforms has brought new capabilities to many applications which were previously not thought possible. Even through development in UAV technology, there is still one inherent problem affecting the safety towards people and infrastructure; the lack of airmanship and intuitive problem-solving skills of UAVs. These shortfalls have led to UAVs crashing in situations where a human pilot would be able to problem solve and react in a unique way to ensure the safety of people (Williams, 2004).

UAV Emergency Forced Landing (EFL) occurs when the UAV platform is forced to land at an unexpected time or place, due to system failures, external interferences, or a command sent from the UAV controller. Current UAV technology has limited scope to successfully achieve this and industry standard methods require extensive planning, control, and monitoring by human ground control operators (Szabolcsi, 2016). The overall goal is to reduce reliance on human operators through decentralised

autonomous control methods, to increase UAV capability in multi-vehicle application and to ensure safety is maintained during all expected and unexpected situations.

2.1 The 3-Phase approach

Prior research has proven a high success rate for both onboard vision recognition of landing sites, and collision detection and avoidance systems. The high success rate is only for the ideal case for that specific system; when put into a real-world scenario multiple factors become problematic to these systems fundamental operating constraints. The primary constraint being if the landing location can be visually detected with the camera systems.

This aforementioned primary constraint can be counteracted using stored data of satellite images, street maps, and terrain data to enable detection of a landing site without needing line of sight. The main constraint of this method is the use of historical data that could be out of date and additionally does not have any information on temporary or moving objects within the scanned area.

The solution presented is the 3-Phase approach, which combines all three methods to allow for a robust solution as each phase complements the weaknesses of the other phases. Table 1 presents the performance summary of each phase and how they complement each other to create a robust 3-Phase approach. Strengths are highlighted in green and weaknesses highlighted in red, with non-ideal performance represented by orange.

2.1.1 Summary of Phase 1

Uses three different databases that contain information of satellite images, street maps, and terrain data to determine the most suitable landing site within the safe flight range of the UAV system.

Figure 1: Satellite image of area

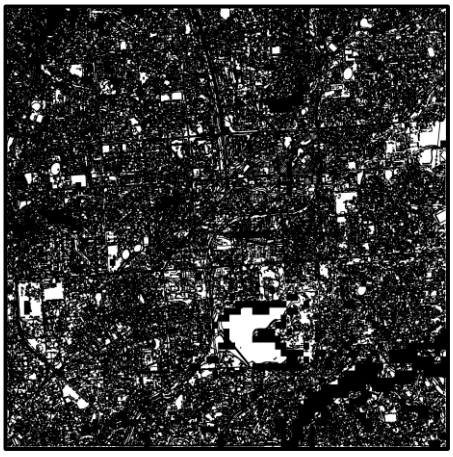


Figure 2: Fused geographic processed data

Figure 3: Selected safe landing site found using EFLS (shown in green)

2.1.2 Summary of Phase 2

Uses onboard camera equipment and scanning laser rangefinders to scan the current area that the UAV system is flying over. The images captured by the camera system can be either visual or infra-red spectrum depending on the application of the UAV system.

Figure 4: Example of Phase 2. Aerial footage take by UAV of a city has detected the water as a safe landing site. Courtesy of Mejias et al. (2009)

2.1.3 Summary of Phase 3

Uses visual/IR cameras, ultrasonic sensors, or scanning laser range finders to avoid collisions with objects on the final approach to the selected landing site as these objects cannot be easily detected while viewing from above.

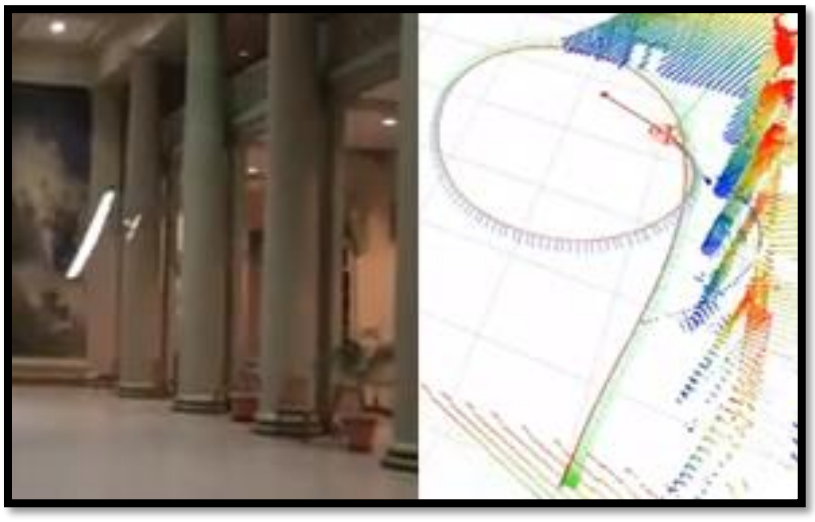


Figure 5: Example of Phase 3. The left side of the image is showing the UAV flying indoors and avoiding obstacles. The right side is the processed LiDAR view used by the UAV. Courtesy of Bry et al. (2012)

Table 1: Summary of Performance for Individual Phases and the 3-Phase Approach

Condition	Phase 1	Phase 2	Phase 3	3-Phase approach
Poor visual conditions (fog, rain, night)	No effect	High impact	Moderate impact	Slight impact
Line of Sight to landing site	Not required	Required	N/A	Not required
Environment has recently changed	High impact	No impact	No impact	No impact
GNSS (GPS) required for operation	Required	Not required	Not required	Not required
Collision detection	Static objects visible only from bird's eye view	Dynamic and static objects visible only from bird's eye view	Dynamic and static objects visible in front of UAV	Dynamic and static objects visible from both bird's eye view and in front of UAV
Human-in-the-loop	Not required	Not required	Not required	Not required

3 Aim of EFLS project

The aim of the EFLS project is to allow a UAV the ability through decentralised autonomous control, to land safely during an emergency situation. This will help future UAVs using this software to avoid harming people or infrastructure.

4 Recommended prereading

To ensure the knowledge base of the reader is sufficient for the detailed operation of EFLS which this manual covers, the following areas of experience are required:

- Software development
- C++ language
- Python language
- ArduPilot, MavProxy, and SITL
- PixHawk or PixHawk 2 understanding
- Linux
- UAV ground control operator experience

These resources can be used to gain experience in the above areas:

- http://ardupilot.org/
- http://ardupilot.org/plane/docs/parameters.html
- http://ardupilot.github.io/MAVProxy/html/index.html
- http://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html
- http://ardupilot.org/copter/docs/common-pixhawk2-overview.html
- http://www.cplusplus.com/doc/tutorial/
- https://www.tutorialspoint.com/python/
- https://www.raspberrypi.org/products/raspberry-pi-3-model-b/
- https://ryanstutorials.net/linuxtutorial/

6 Prototype software

Prototype software was created to test key assumptions prior to full development of the full EFLS code. This also allowed for alternative solutions to be rapidly tested to ensure that the most suitable solution in terms of development time, processing time, and resources was found. The prototype software was developed using MATLAB as it allowed for a quick development cycle to occur.

6.1.1 Testing goals

Through the rapid developing cycle of the prototype software, the following goals will be determined to enhance the direction taken for the optimised software:

1. Determine the accuracy and computation time of Canny edge detection in determining a safe landing site.
2. Determine if adding street map data improves accuracy of finding a safe landing site.
3. Determine potential methods to calculate the landing site based on the Satellite and Street map processed datasets.
4. Determine potential methods for the path planning to the selected landing site.
5. Determine a viable software architecture based on experience with the prototyping software environment.

6.1.2 Canny edge detection

The Canny edge detection algorithm implemented using the MATLAB image processing library, successfully detects buildings, houses, roads, and other manmade structures that are likely to have people located in or near by. The edge detection results are then passed through a line expansion algorithm, which increases the area occupied of the detected unsafe object and hence ensures the selected landing site is not chosen to close to the detected object.

The Canny edge detection algorithm takes on average 0.2s to process a sample image, with the line expansion algorithm taking an additional 0.3s on average to process.

The total time of this processing is sufficiently low and the accuracy of detecting the required objects is at an acceptable level, to prove that its viable to continue with development using Canny edge detection for the processing of the satellite images. Noteworthy bugs from testing on a variety of satellite images, is that buildings with massive roofs such as factories and clouds in the satellite images are detected as valid landing sites due to their uniformity in texture over a large relative area. This is however not an issue for the final UAV EFL Phase 1 software as the combination of street maps and correctly setup landing site parameters will ensure these false positive results are filtered out.

6.1.3 Street maps

Initial testing of the OSM database with MATLAB proved to be near impossible due to the supporting libraries for MATLAB being extremely inefficient. To combat this issue during the prototype stage of development, the Street Map analyser module was implemented in C++ with the Libosmium header library for testing purposes. The Street Map analyser currently can mark roads and other OSM data features as unsafe areas to land. This is achieved through drawing single pixel wide lines representing the data and then applying the line expansion method. The data is then passed through the site selection module and is treated identically to the Satellite Image analyser data during this stage of processing.

Figure 7, shows the Street Map analyser working in parallel with the Satellite Image analyser, and additionally shows the point which the data is combined and the further processing techniques applied to the combined data. Currently as seen in Figure 7, the Street Map analyser does not distinguish between the difference of a closed or open object, thus meaning the closed objects are plotted as lines and not polygons. This will be improved in the final UAV EFL Phase 1 to identify unsafe landing sites based on the closed object data such as airports, military bases, residential areas, and other OSM feature data.

6.1.4 Site selection

The Site Selection module has been developed in MATLAB in conjunction with the Satellite Image analyser code. The Site Selection module consists of three stages:

The first stage is the scanning of the provided data from the analyser modules with a mask representing a suitably sized landing strip. The mask is scanned at 0° and 90° angles to simplify the code, while the final implementation will also scan at ±45°, for increased detection rate.

The second stage is the combination of the data from the analyser modules to create a more accurate data set.

The third stage is the scanning for the most suitable site of the available safe areas. To identify the most suitable site, the “blob” with the largest area is calculate and chosen as the primary candidate. This “blob” is then scanned to see what dimensions the largest rectangle can fit within it with a preference for a 2:1 ratio of length to width for the selected landing site. This rectangle is then confirmed to meet the specifications for the landing distance and is then sequentially submitted to the Path Planner module.

The largest area “blob” is chosen due to two reasons. The significant reduction in computation required to find the largest rectangle within a single “blob” compared to many “blobs”, and the assumption the general area of the largest area “blob” is less cluttered.

6.1.5 Path planning module

The Path Planner module was developed using MATLAB in conjunction with the Satellite Image analyser and the Site Selection module. The Path Planner module will also be converted to C++ code later, however needs to be rewritten for optimisation reasons.

The Path Planner module has the goal of planning a path from the UAVs current location and bearing to the selected landing location. The constraints of the UAVs turning distance, glide slope, and height management above terrain have been considered and solutions implemented; such as dynamically generating a curved turn to align the UAV with the runway, averaging the descent through the entire planned route, and plotting waypoints in Above Ground Level (AGL) using the terrain data. The factor of energy management has been considered but the solution has not been created to ensure a suitable path is planned in a loss of thrust scenario.

The current Path Planning progress can be seen in Figure 6 & 7, where the final stages include a colour image with a red dot representing the current location of the UAV, and the blue dots representing the calculated waypoints which are joined by a blue line until landing at the highlighted selected landing site.

6.1.6 Preliminary results from prototype software

The preliminary results from the prototype UAV EFL Phase 1 module are shown in Figure 6 & 7. These results have provided proof for the assumptions being made that the processing requirements are significant however can be optimised for an onboard companion computer, image processing techniques can be implemented from library functions, Street Map data can be analysed and useful results generated, and system complexity is within the scope of the project. The results have also displayed that the system is quite effective at choosing a safe landing site.

Figure 7 provides a detailed view of all modules completed to date, which is functioning to generate a safe landing site and the path towards it in 3D space. The combination layer can be seen to be smaller than either of the individual analysis modules showing that non-common mode errors have been removed.

Ten different test locations at different zoom levels, terrain, and environments, that have been processed by the Satellite Image analyser, Site Selection module, and the Path Planning module. The first eight were randomly selected from Bing Maps to quantify the performance of the current algorithm, while Location 9 and 10 were chosen to highlight the current bug of detecting clouds and factories as safe landing sites. Locations 1 to 8 all resulted in a safe landing site and path being generated thus resulting in a 100% success rate using this small sample set. Locations 9 and 10 were false-positive results with the cloud being chosen as a safe landing site in both case, however, this has been excluded from the success rate as it's a known fault with a solution planned to be implemented. Location 4 shows how the Canny Edge detection method automatically increases its sensitivity rate based on the average throughout the image, as small differences were classed as edges while the other locations tested would not have returned such a small difference as an edge.

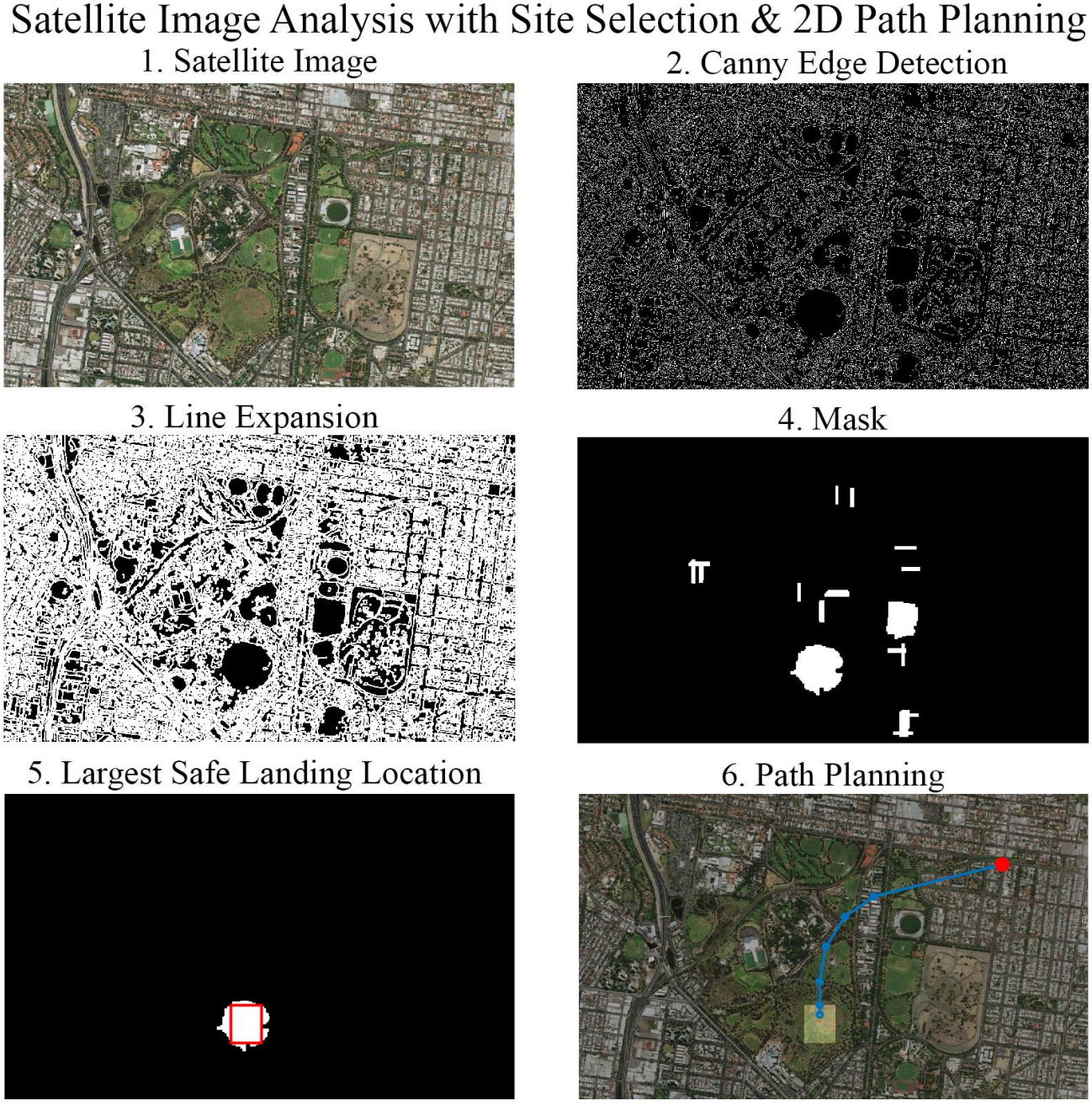


Figure 6: Satellite image analysis with site selection and 2D path planning - Prototype software

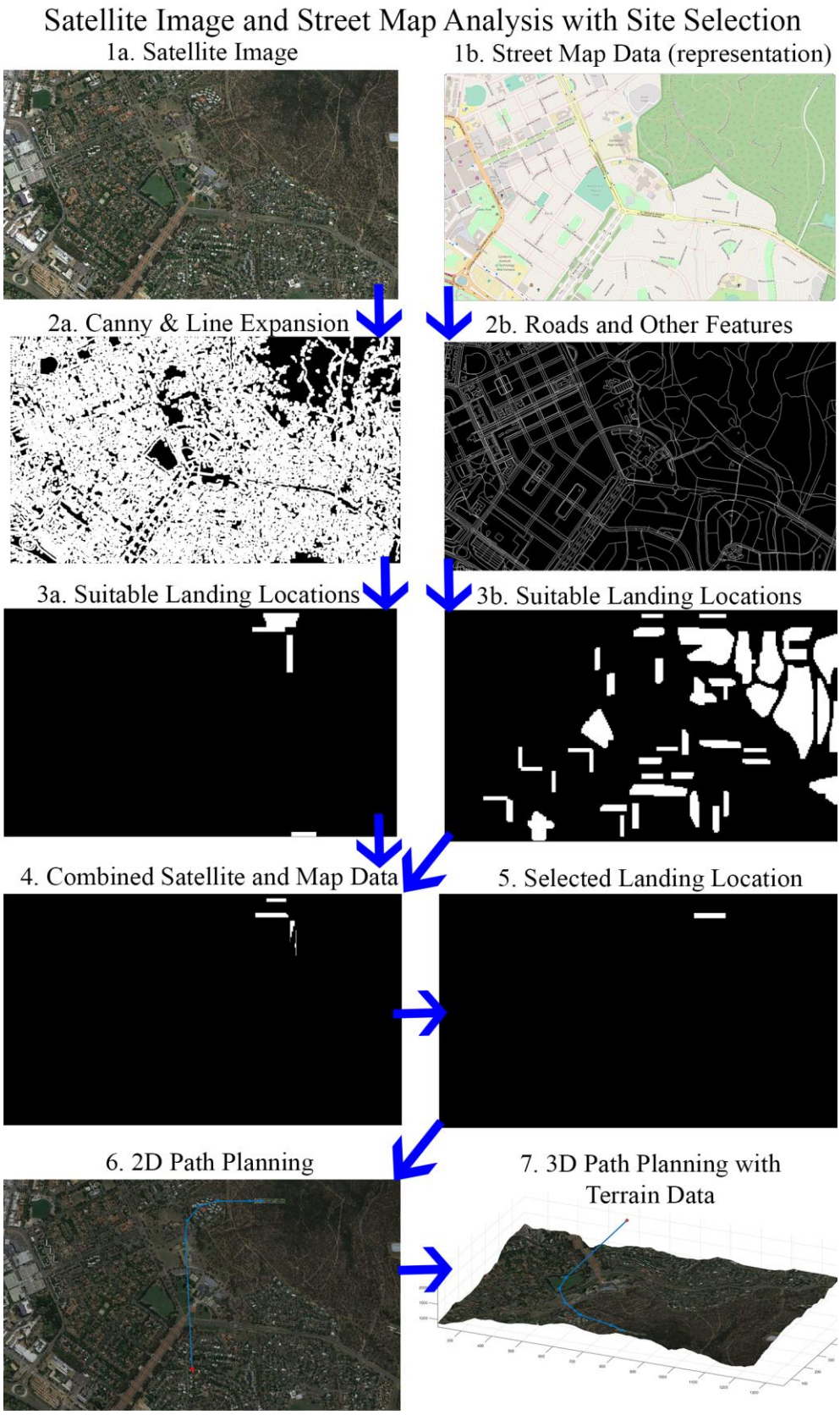


Figure 7: Satellite image and street map analysis with site selection - Prototype software

6.1.7 Performance

The overall performance of the UAV EFL Phase 1 system is currently unsuitable for implementation on a companion computer. The MATLAB code used for the Satellite Image Analyser, Site Selection module, and Path Planning module which resulted in outputs shown in Annex E, on average took 5.7 seconds to find a suitable landing site. Figure 8, shows the time taken to process for each of the locations. Location 4 and onwards increased in resolution by 40% which increased processing time by 50%. Location 8 shows that when many suitable landing areas are found, the rectangle fitting method significantly increases the time of computation.

Performance enhancements will be significant when developed in C++ instead. The Street Map analyser is currently written in C++ as the MATLAB equivalent took 10 minutes to run while the C++ version now used takes under 0.1 seconds to run. Additional performance enhancements can also be made to the custom methods written in MATLAB when rewritten in C++. The ability to use OpenCV in C++ will also allow for GPU processing that will reduce processing time for the Canny edge detection algorithm.

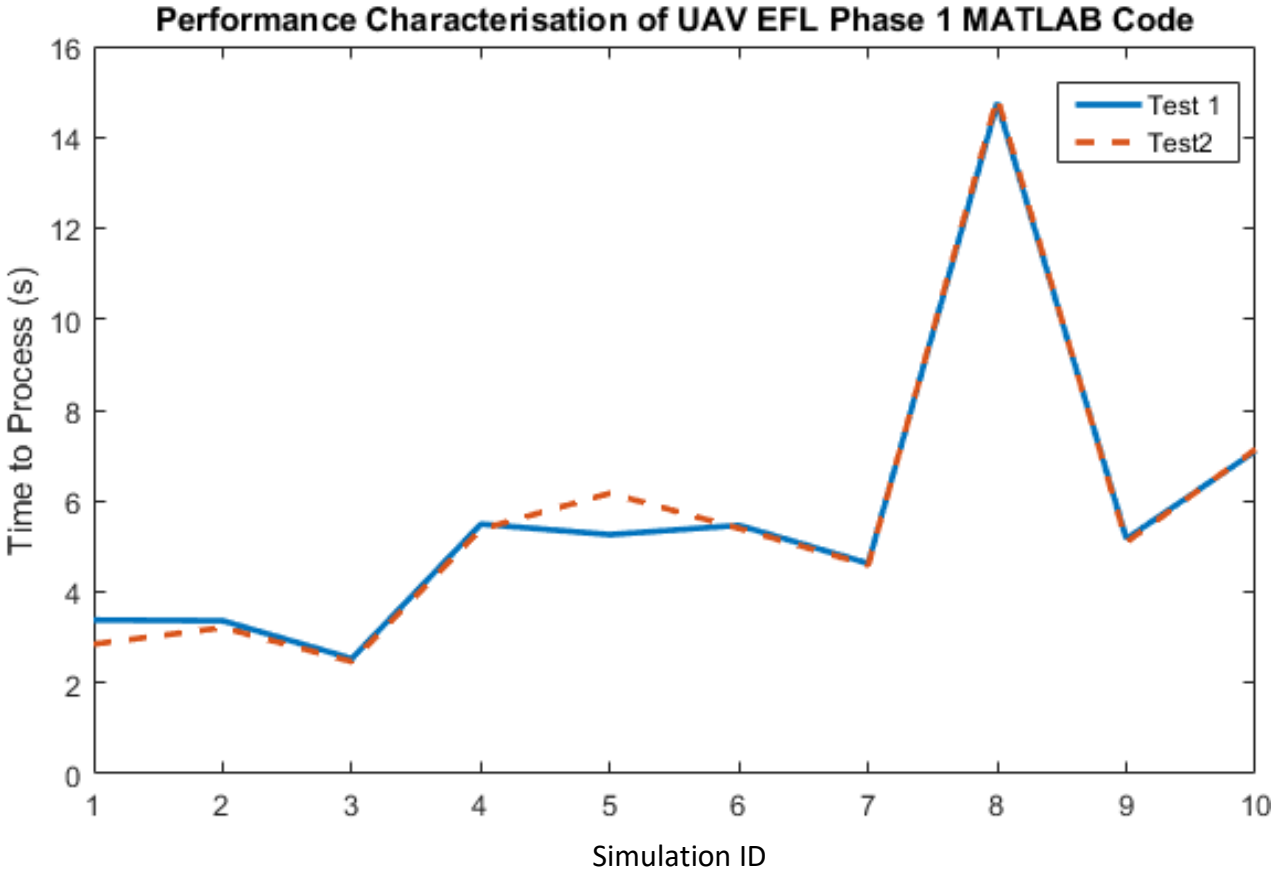


Figure 8: Performance characterisation of UAV EFL Phase 1 MATLAB code

7 Optimised software – EFLS

The optimised software that has been developed has been coined the Emergency Forced Landing System (EFLS). It was developed in the C++ programming language for increased efficiency while running, flexibility of OS and hardware used, and relative ease of development.

This section of the report will explore the details of the EFLS software to increase the readers understanding of how it operates.

7.1 Code architecture

The code architecture was developed with flexibility of expansion and flexibility of which UAV flight controller to use. EFLS was not developed with any specific flight controller constrains and thus it can be integrated with any UAV flight controller quite easily. This is because it uses a custom communication protocol which just needs to be integrated into the flight controller system.

For the purpose of testing and providing a direct use case for EFLS, it has been integrated into the ArduPilot UAV flight controller software. This allows EFLS to be put onto any small scale UAV that uses ArduPilot or MavLink communication protocol.

In the code architecture diagram shown in Figure 9 below, EFLS is internally controlled by the Decision Engine. The decision engine controls the flow of the program and determines the dynamic parameters of the system based upon the current UAV flight data being received. When the decision engine detects an engine failure or EFLS is activated manually, it will use the Satellite image module, street map module, terrain data module, and site selection module, to find a safe landing site within the flight range of the UAV. Further detail of the operation is discussed later in this section.

The Decision engine uses the AircraftLink class to communicate with the Protocol buffer. The protocol buffer is the middle man in the communication system, where the data is stored. No direct link between the EFLS AircraftLink class and the flight controller AircraftLink interpreter is made, its done through the proxy of the protocol buffer. This allows for a simple and robust communication system to be used, with the ease of development for future flight controllers.

The protocol buffer then communicates with the flight controller through an interpreter program written for that specific flight controller. In this case, a MavProxy module has been written to convert the EFLS communication protocol into the language of the UAV flight controller. MavProxy also runs on the same companion computer as EFLS.

MavProxy then communicates through MavLink to the UAV flight controller though a dedicated serial link.

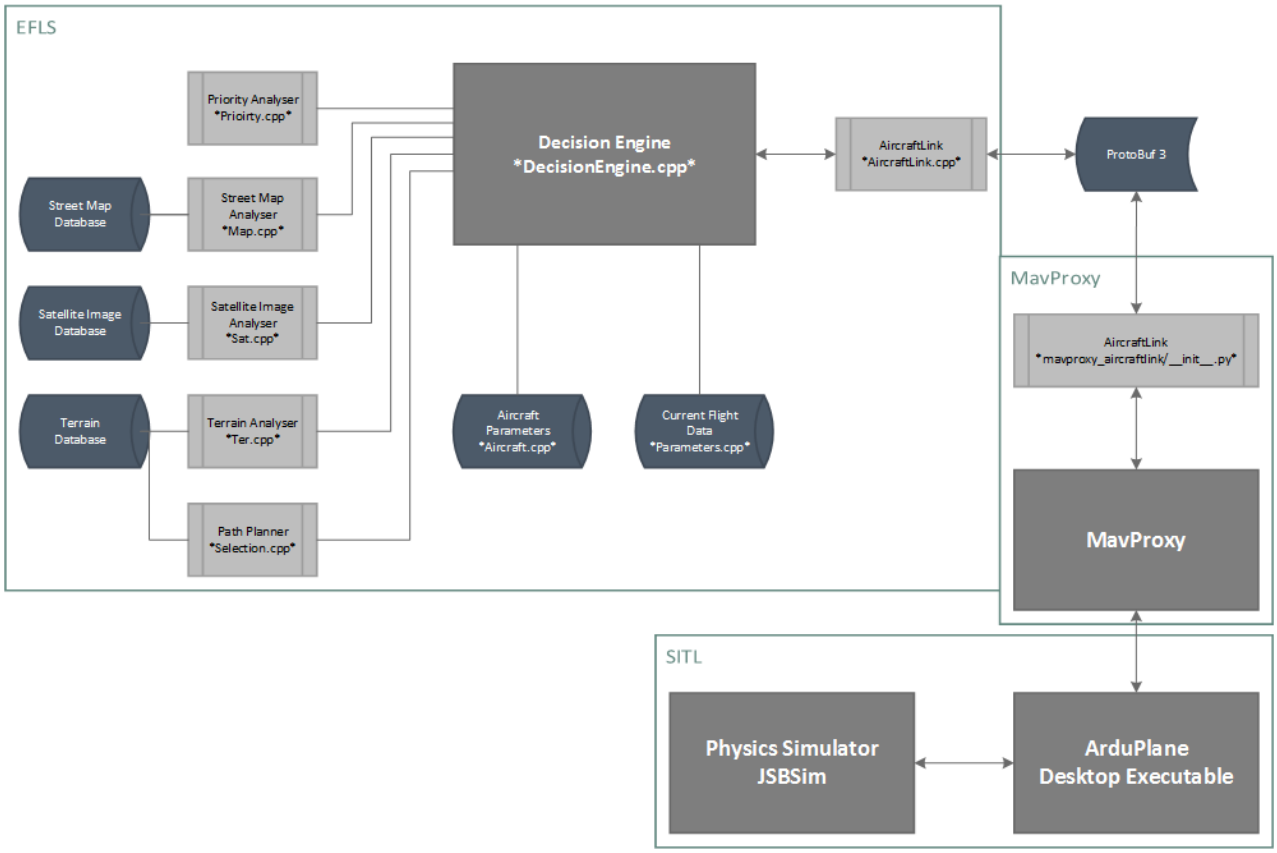


Figure 9: EFLS code architecture diagram

7.2 Decision engine

The decision engine is the core of EFLS. It provides the dynamic selection of a safe landing site by adjusting internal parameters based upon the UAV current flight data. It also controls and managers all other modules and classes of code for EFLS.

The decision engine can be broken up to three components:

1. Emergency monitoring – Checking continuously for an UAV emergency conditions requiring EFLS to be activated.
2. Dynamic adjustment of internal parameters – The internal parameters are adjusted based upon current UAV flight data, to ensure EFLS chooses a suitable landing site that is possible to reach.
3. Executing the code modules – To collect and process data to find a safe landing site within range of the UAV.

The first component, the Decision Engine uses the Estimator class to check for an emergency condition and the available flight range of the UAV. This is explained in detail in section 7.8.

The second component and third components are interweaved during the operation, so the flow of processing will be explained instead as its easier to understand. The below sequence is a basic overview of how the code processes during an emergency scenario.

1. Set internal parameters based on UAV current flight data.
2. If range is greater than *largeSearch_range*, then start the large area search.
    a. If found a suitable landing site, start a close search of the area.
        i. If found a safe landing site, goto 6
3. Close search around the UAV for a safe landing site.
    a. If found a safe landing site, goto 6
4. Do a series of close searches at a random location within the defined boundary.
    a. If found a safe landing site, goto 6
5. If no safe landing sites are found, then goto 1 and reduce the sensitivity of the search.
6. Use the selected safe landing site, to calculate final approach from the UAVs current position, ensuring the planned path is possible to achieve by the UAV.

7.3 Communication protocol

The communication protocol used by EFLS is the Google Protocol buffer (Protobuf 3). This is a method of communicating through binary text files, which is software independent and highly robust. The data sent from both EFLS and the flight controller EFLS interpreter are stored in two different files, which avoids the issue of both programs writing data at the same time, as the system is a write only / read only system.

The protocol message format has been define in the .proto file, and then the specific C++ and python code has been compiled from this message format. The message format current supports the following information being sent between EFLS and the flight controller EFLS interpreter.

- EFLS to flight controller EFLS interpreter
    - List of waypoints containing information of:
        - Latitude of waypoint
        - Longitude of waypoint
        - Altitude of waypoint
        - Type of waypoint

        - Speed or custom data of waypoint

- Flight controller EFLS interpreter to EFLS
    - Aircraft data of:
        - Latitude
        - Longitude
        - Ground trace heading
        - Speed
        - Wind heading
        - Wind speed
        - Engine throttle
        - Engine RPM or current (depending on electric or petrol engine)

More information about the Google protocol buffer can be found at:
https://developers.google.com/protocol-buffers/

7.4 Satellite image module

The satellite image module is the first layer of the multilayer approach implemented in classifying safe landing areas within the designated region. It uses cached satellite images of the search area to identify man-made objects such as houses, buildings, roads, and other surface infrastructure that are potentially populated, therefor need to be avoided.

Satellite image processing of safe landing areas is similar to the processes used for onboard vision base site selection discussed as a part of Phase 2. Mejias et al. (2009) , Faheem et al. (2016), and Lu et al., (2013), have all developed initial image processing techniques utilising the Canny edge detection algorithm for safe landing area detection. While the results provided from their systems don't specifically state the accuracy of the Canny edge detection algorithm, their test results all indicate an average accuracy of 92% in classifying a safe landing site.

EFLS satellite image module implemented the OpenCV Canny edge detection as the primary form of detection for unsafe landing areas. To improve the results of the Canny edge detection a line expansion algorithm was used to expand the unsafe area to the neighbouring pixels, hence creating a border around the potential detected object. The satellite images used were fetched from the Google Maps database using their provided API. The satellite images are cached locally once they are fetched from Google's servers to increase processing speed. Figure 10 shows a satellite image of the example test area, with Figure 11 showing the satellite image processed by EFLS.

A summary of the processes taken by the satellite image module can be summarised below:

1. Save data received from the Decision Engine, such as parameters and aircraft data.
2. Calculate the satellite image required for the location.
3. Check cache for this satellite image and if it exists store it in memory as a OpenCV MAT for quick access.
4. If the cache did not have the required image, download it from the Google satellite image servers. As there is a limit on the size of image that can be download, merge tiles together until the required resolution is reached.
5. Calculate and save the coordinates of the centre of the satellite image for correct calculation later on.
6. Convert to greyscale.
7. Blur the image with a kernel size of 3x3 for improved performance of Canny Edge detection.
8. Conduct Canny Edge detection on the greyscale image.
9. Use the custom line expansion algorithm to create a boundary between the safe areas and the unsafe area.
10. Save the binary array results for later use.

Figure 10: Satellite image of the example area

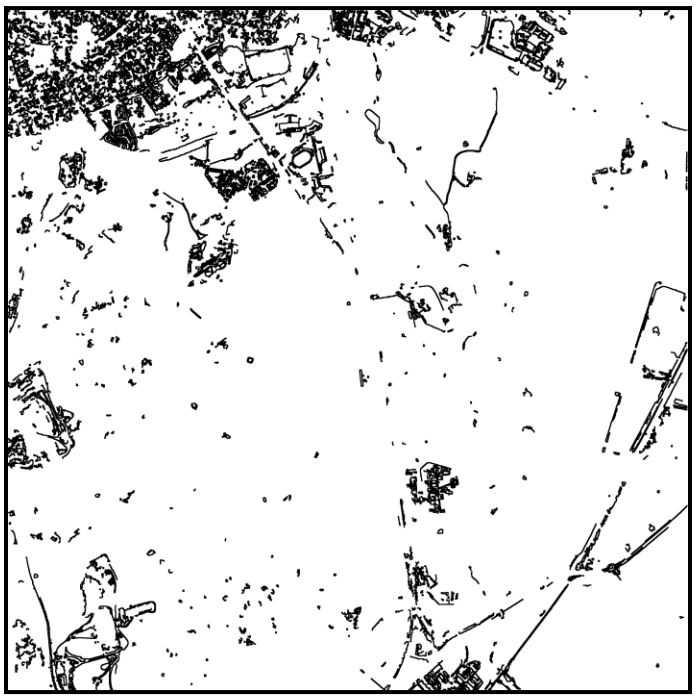


Figure 11: Canny edge detection and line expansion of the example satellite image. With unsafe areas shown in black.

7.5 Street map module

The street map module is the second layer of the multilayer approach used in classifying safe landing areas within the designated region. This layer uses data from the Open Street Map (OSM) database to identify roads, large buildings, airports, and boundaries of different terrain. These man-made objects have the potential to have people within or nearby, which can then be avoided when selecting a safe landing site.

OSM street maps provide an alternative independent data source to satellite images, but after processing results in similar safe and unsafe areas being detected. This increases the reliability and robustness of the Phase 1 approach through design diversity. Not only is additional data provided for the decision of a safe landing site but also if one method fails, the other method can still provide suitable information for a safe site to be selected.

Haklay (2010) presents a review conducted in 2008 on how effective volunteer geographic information is, with OSM as the primary case study. Various quality standards, such as positional accuracy and completeness were used to assess OSM data against maps provided by Ordnance Survey (OS). OS acts as the national mapping agency for Great Britain providing high quality maps throughout England. The OSM positional accuracy was 88% when using a buffer range of approximately 5 meters and a coverage of 57% throughout England. A later review in 2015 conducted by Maron (2015) shows the completeness of the global OSM dataset compared to the total road distance in the CIA World Factbook. Majority of countries had over 90% coverage with the global coverage between 107% and 130% of the stated value

in the CIA World Factbook. This concluded OSM to have a high completeness for global street maps. These studies showed that in general, the OSM dataset was already a good representation of the actual worldwide roads and it shows continuous improvement in completeness due to the contributions made by the active OSM mappers worldwide.

EFLS street map module implemented the use of Libosmium (Topf, 2017), to enable an efficient method of access to the OSM datasets within the C++ language. After extracting the required OSM data for the search area, the Bresenham's line algorithm is used to plot the coordinate data points in a binary array which shows the safe landing areas derived from the OSM data. The lines are expanded by the custom line expansion algorithm to provide a safe boundary between the safe areas detected and the unsafe roads or other elements detected. Figure 12 shows the example area processed with the EFLS street map module.

A summary of the processes of the street map module is show below:

1. Save data received from the Decision Engine, such as parameters and aircraft data.
2. Decode the street maps file and store in memory.
3. Setup location handlers and the custom processing handler.
4. Enter into the MapBuilder class.
5. Rapidly find nodes within the approximate area.
6. Process these nodes for the search area, and save the required nodes in a C++ vector.
7. Plot each way based upon its nodes, using the Bresenham's line algorithm,
8. Return to the Map class.
9. Use the custom line expansion algorithm to create a boundary between the plotted ways and the safe area.
10. Save the generated binary array for later processing.

Figure 12: Processes street maps of the example area

7.6 Terrain data module

The terrain data module is the third and final layer of the multilayer approach used in classifying the safe landing areas. The module uses terrain height data to calculate the gradient of the slope within the landing site area to ensure that the landing site is flat and suitable for both landing and the final approach. While the terrain gradient does not directly protect humans or infrastructure from harm, it helps to protect against secondary effects such as fires from a UAV crash landing into steep terrain.

The terrain height data used was collected from the Shuttle Radar Topography Mission (SRTM) that was launched by NASA in 2001. The mission collected 1 arc second accurate topography data for the globe, with 3 arc second accurate data being released to the public which is approximately equal to sampling every 90m (Farr, et al., 2007).

EFLS terrain data module implemented a custom .HGT file decoder that enables the C++ code to interpret the SRTM encoded data. The decoded data is used to calculate the magnitude of the slope gradient, which forms a binary array showing the safe landing areas based upon the threshold gradient limit set at 5%, which can be changed in the EFLS user parameters. Figure 13 is the processed terrain of the example area, it is seen that the bottom left corner has steep terrain due to the mountain range.

A summary of the processes of the terrain data module is show below:

1. Save data received from the Decision Engine, such as parameters and aircraft data.
2. Search the cache for the terrain data of the search area, and if it exists then load it.
3. If the terrain data is not found, then download it from the ArduPilot servers.
4. Decode the terrain data using the custom .hgt decoder.
5. Limit the terrain data to just the search area.
6. Process the magnitude of the gradient across the search area. If the gradient is steeper than the user set threshold parameter, then class that area as too steep to land.
7. Plot the results into a binary array.
8. Save the generated binary array for later processing.

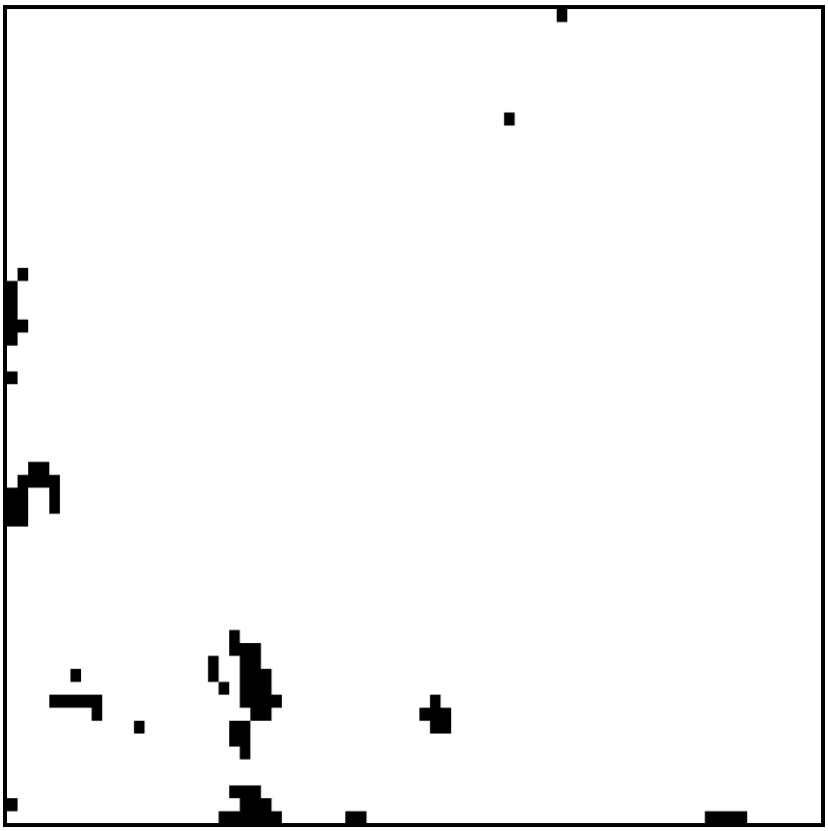


Figure 13: The processed terrain data for the example area

7.7 Site selection module

The site selection module uses the three layers generated in the multilayer approach and fuses them together to create a single binary array of safe landing areas that could be used, this is shown in Figure 14 for the example area. Mejias et al. (2009) and Lu et al. (2013) present the same solution of scanning the binary array with a mask. The mask has the dimensions for a suitable landing site, which when passed over the array generates a new array defining areas which are both safe for landing and are of suitable dimension. The mask is passed over four times at angles of 0°, 45°, 90°, and 135° to ensure all valid landing sites are found.

Within the prototyping stage of code development, the mask method was implemented for preliminary testing as described in Mejias et al. (2009) and Lu et al. (2013). It was found that this method of site selection chooses many valid landing locations, however, does not find a single specific site. Additionally, the landing site would not be directed windward which would affect fixed-wing UAV landings. Alternative methods were explored that used "blob" detection to find the landing site of the largest area and then fit the largest rectangle landing mask to it, however, this was found to be too computationally expensive.

EFLS site selection module implemented a mask based site selection method that has additional features to enable the selection of the single largest landing site that was directed windward. The mask is initially rotated windward to allow the fixed-wing UAV preferred landing characteristics and reduce the required computation since only one set of scans is required. Using the user defined minimum landing site dimensions, the mask is then scanned across the binary array of safe landing sites. If more than one landing site was found then the mask is increased in size and the scan conducted again, if one valid safe landing site is found then it is accepted. If no valid sites are found the decision engine repeats the search with different parameters or at an alternative location, to ensure a safe landing site is eventually found. The found safe landing site for the example area is shown in Figure 15, with an overlay of the satellite image.

Figure 14: Fused data from the processes geographic data

Figure 15: Found safe landing site shown as a red box, for the example area.

The path planning section of EFLS was also coded within the site selection module. The path planning code is used to find a suitable flight path for the UAV from its current position to the selected safe landing site while adapting to the external environment and UAV condition, such as engine failure. For the flight path to adapt to the conditions required, it must account for initial altitude, available range, landing heading, wind conditions, and required flight profile of the airframe.

Eng, Mejias, Walker, and Fitzgerald (2007) conducted an analysis and simulation testing into dynamic path planning of UAV systems. Two different algorithms for dynamic path planning were developed based on processes already implemented by human pilots in forced landing situations. The primary focus for these dynamic path planning algorithms was the management of changing wind conditions, both in magnitude and direction. Algorithm 1 uses a method of generating all waypoints initially and flying to all waypoints only if altitude is at the required level, if not then it diverts to a more direct route to increase its available altitude for the next waypoint. Algorithm 2 initially generates a few key waypoints to define the most direct flight path, then during transit additional waypoints are added perpendicular of the current UAV direction based upon whether the calculated glide slope will exceed the minimum for the direct route. Algorithm 2 outperformed algorithm 1, with an overshoot distance of 200m and a 52% success rate compared to that of 400m and 26% for algorithm 1.

EFLS path planning code implemented a simplified model of algorithm 2 that was presented by Eng, Mejias, Walker, and Fitzgerald (2007). The primary focus of the path planning implemented was on a curved final approach turn, that would allow the UAV to land at the selected safe landing site from any initial direction and altitude. Wind compensation was achieved similarly to algorithm 2 with the worst-case path calculated, however, instead of adding waypoints to reduce altitude the desired altitude is reached by increasing the speed of the UAV. An increase in speed will allow the UAV to increase its drag, hence reducing its overall energy on landing. If the speed on landing is still in excess, the UAV will land with an increase in speed. This method proved to be sufficient, however, requires the accurate setup of the associated parameters to ensure an over speed landing is not possible. Excess altitude prior to the final turn towards the runway is lost through a spiral descent until the desired altitude is reached.

A summary of the processes of the site selection module is show below:

1. Save data received from the Decision Engine, such as parameters and aircraft data.
2. Fuse the three scan results generated by the geographic processing modules.
3. Invert the fused data, as it simplifies the site selection algorithm.
4. Rotate the image with a lossless algorithm, in the direction of the wind. This is so the mask can be scanned vertically but be directed windward in relation to the image. This was done to simplify the code.
5. Scan using a vertical mask for the minimum safe landing area.
6. If no landing site are found, return to the Decision Engine that no safe sites could be found.
7. If more than one landing site is found, increase the mask size by a factor or 1.25 sizes and repeat the scan from 5.

8. If no landing sites are found when the mask has been expanded, select one of the safe landing sites from the mask of the smaller size, accept it and goto 10.

9. If one safe landing site is found, then accept it and goto 10.

10. Save the coordinate location of the centre of the safe landing area.

11. Plot the mask on the binary array and unrotated the image using a lossless algorithm. The mask will now be rotated with the image in its original orientation.

12. Using the saved coordinate location, calculate a suitable path to the selected landing site from the UAVs current position. The algorithm actually calculates it from the selected safe landing site to the UAV current position, this was done as it provided for a simpler coding solution.

    a. Set the landing waypoint as the centre of the safe landing site at an altitude of 0m AGL.

    b. Set the last two waypoints on the landing approach, at the user define intervals and altitudes. These waypoints are set to have the same heading as the runway, to ensure a stable a straight approach.

    c. A custom turn algorithm has been made that plots multiple waypoints for the UAV to fly to in a large arc shape, to change the UAVs initial heading to the heading of the runway. This manoeuvrer also helps to reduce the altitude to the required amount.

    d. Set the first waypoint as a “loiter to alt” waypoint. This will force the UAV to reduce its altitude in a spiral formation down to the desired altitude of the final turn towards the landing site.

13. After calculating the waypoints for the approach to the safe landing site, the waypoints are returned to the decision engine. The decision engine will then send it to the UAV flight controller through the communication protocol.

7.8 Estimator class

The estimator class is used to provide an estimation of the current range of the UAV. The range is estimated on the current altitude of the UAV, wind conditions, UAV current speed, engine failure of the UAV, and the UAVs user defined glide slope value during an engine failure scenario. This estimated range is then reduced by the safety factor value, to ensure the range estimated is within the actual practical range. This value is then returned to the Decision Engine.

Additionally, the estimator class can detect an engine failure. This is done via checking the requested throttle level of the flight controller to a minimum RPM or current reading of the engine/motor of the UAV. If a failure is detected at any point, the Decision Engine uses this information to find the safe landing site.

7.9 Convert class

The convert class provides advance math functions, primarily for the conversion between the xy cartesian plane to the lat/lon coordinate values. The functions are critical to calculate the location based on the images and binary arrays. The below is a summary of each of the functions in the convert class:

- wrapTo180 – Warp the input angle between (-180,180]
- rad2Deg – Converts radians to degrees
- deg2Rad – Converts degrees to radians
- pixelPerMeter – Calculate the pixelPerMeter from the zoom level and the latitude.
- coordinate2Bearing – Calculates the bearing between two provided coordinates.
- coordinateProjection – Calculates the coordinate from a starting coordinate, a distance, and an angle. Uses the Rhumb line theorem.
- haversine – Calculates the distance between two coordinates, using the haversine theorem.
- coordinate2Matrix – Calculates the cartesian location based on the matrix origin coordinate, the required coordinate to find the location of, the matrix dimensions, and the pixels per meter.
- lineExpansion – This is the custom line expansion algorithm. It was placed in the convert class, as it was able to be accesses statically from any function.

A useful resource to understand the complex coordinate transformers can be found at: http://www.movable-type.co.uk/scripts/latlong.html

7.10 File writer class

The file writer class is for analysis of EFLS performance.

The first task it completes, is the custom time profiler. The time profiler writes the time it takes to complete each section of the code in a detailed text file which is human readable for quick analysis. All iterations of the decision engine are also recorded in the profiler, thus it can be seen EFLS performance at all stages of the decision engine. Additionally, if EFLS is run multiple times, the timing profiler will label each different execution of EFLS in its log. This allows for a single file to be analysed that contains potentially thousands of separate executions of EFLS.

The second task the file writer class achieves, is the saving of the detailed results of each of the iteration of the decision engine. The file writer separates each of the results from the decision engine iteration into a separate folder. The results saved is the binary array represented as a black and white image of each of the geographic processing modules, and then the final results generated by the site selection algorithm.

8 EFLS Installation

The installation of EFLS is important to follow closely and correctly to ensure that the software will behaviour and compile as expected. The EFLS installation is required to be compiled from the source code, as the use case for system is diverse and a single hardware and architecture solution would not fit most of the user's requirements.

The installation process will cover all steps required to setup a virtual machine of Ubuntu running on a Windows host machine, with the EFLS configured for use with a ArduPilot based autopilot.

8.1 Virtual Machine

A Linux based operating system is required for the EFLS software to work correctly. In addition to this requirement, MavProxy works best on a Linux based system. The Linux distro recommended is Ubuntu as the software was created in this environment and extensively tested, however other distros should also work but extensive testing should be conducted to ensure reliability.

A virtual machine of Ubuntu was chosen for ease of setup for the EFLS developing environment. For use cases with EFLS operating inside of a UAV companion computer, it is highly suggested to run Linux natively for increased performance and reliability.

8.1.1 VM installation

The recommended VM is Virtual Box as it is free and has been proven to work successfully with EFLS. To run all EFLS and SITL for multiple location simulations, it is recommended to have an Intel Core i5 processor with at least 4GB of RAM dedicated to the VM and 60GB of storage space.

The instructions below are based on the tutorial found at http://www.psychocats.net/ubuntu/virtualbox, with modifications of settings to ensure an ideal setup for the EFLS developing environment.

1. Download the latest VirtualBox install files for Windows from https://www.virtualbox.org/wiki/Downloads.

2. Open the executable file downloaded for the Windows host computer, and follow the prompts to install VirtualBox.

3. After installation is complete, restart and open the VirtualBox manager.

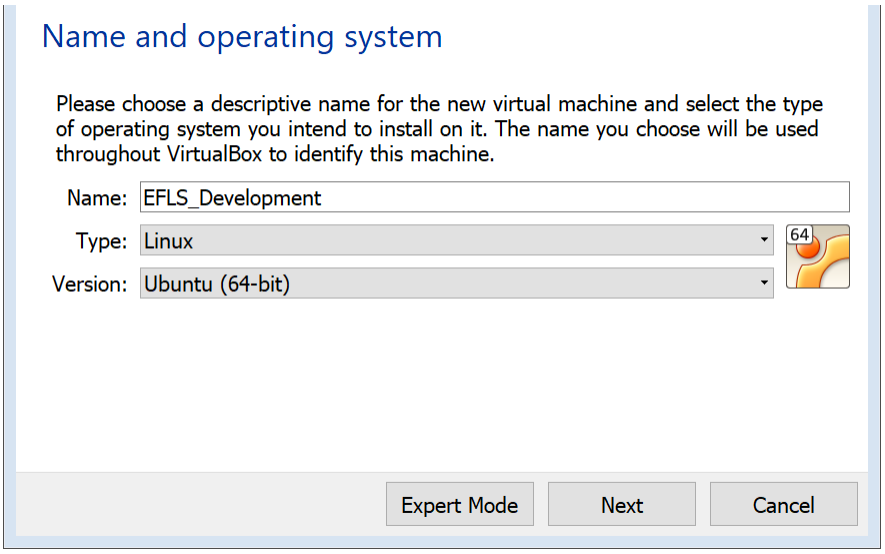
If followed correctly, you have now installed VirtualBox for your PC.

As we have now installed the VirtualBox manager, we now need to create a virtual machine instance of Ubuntu, thus allowing us access to a Linux environment.

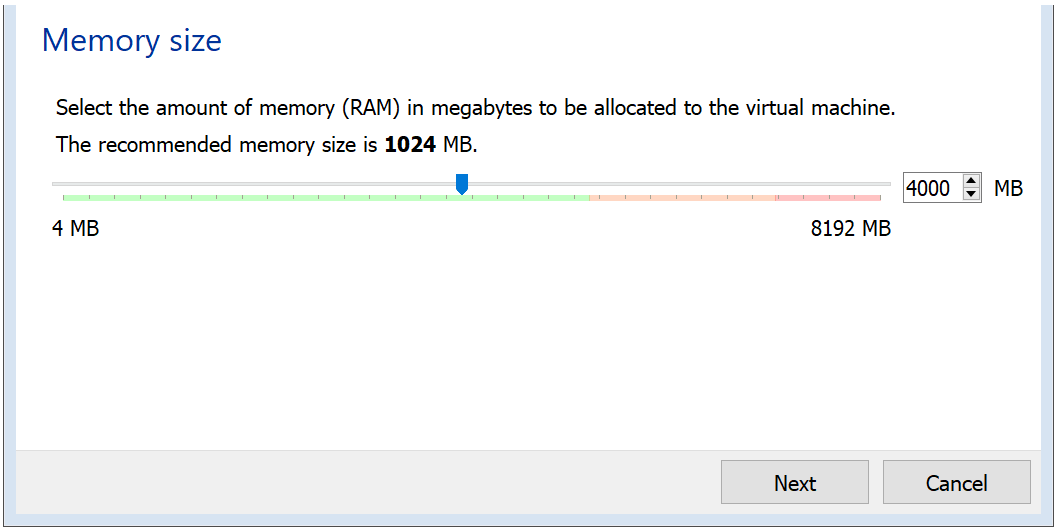
1. Download a copy of the Ubuntu operating system from https://www.ubuntu.com/download/desktop. The recommended version is 16.04.3 LTS, which is a stable release of Ubuntu with long term support for 5 years after its release. Newer versions should work, but have not been tested. Ensure that the 64bit Desktop version is downloaded.

2. Click on the “New” button in the VirtualBox manager to open the setup wizard for a new VM.

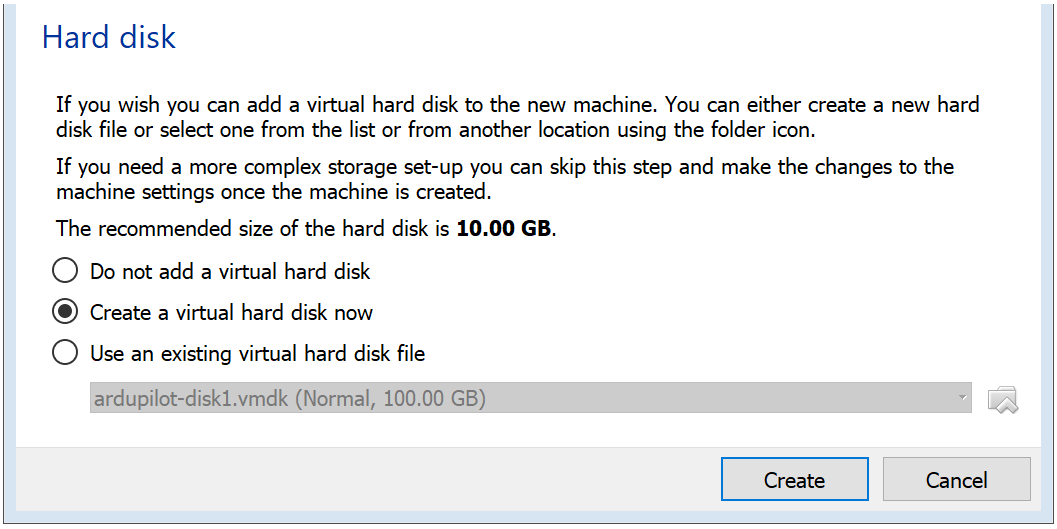
3. Select the Type as “Linux” and the Version as “Ubuntu 64bit”. Add a name to your VM as you feel necessary, and then click next.



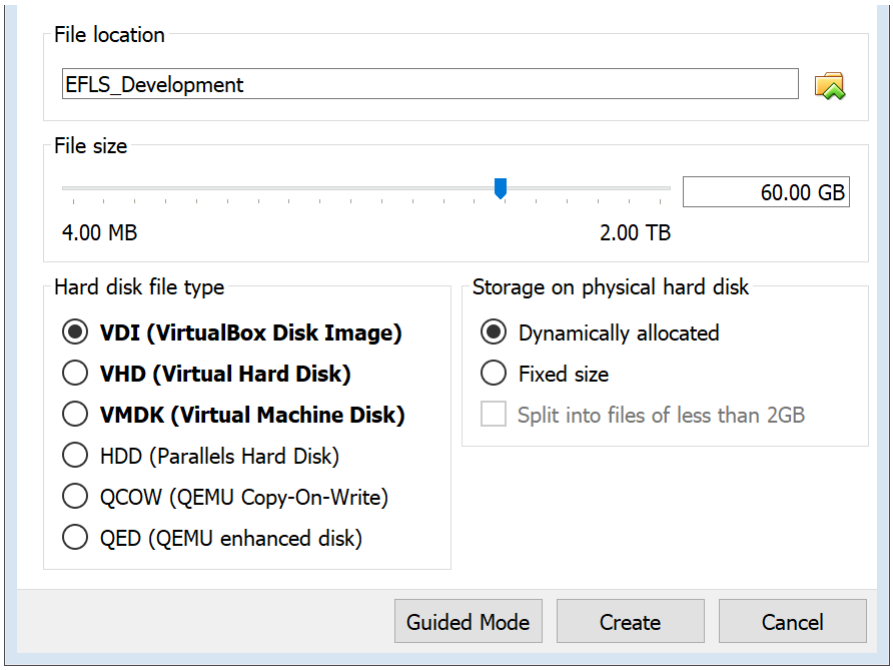
4. Set the memory size to 4000MB or the closest amount your system can safely commit to the VM. The more RAM the better, however we can’t use it all otherwise we affect our host system performance. Click next.



5. Select “Create a virtual hard disk now”, and then click Create.

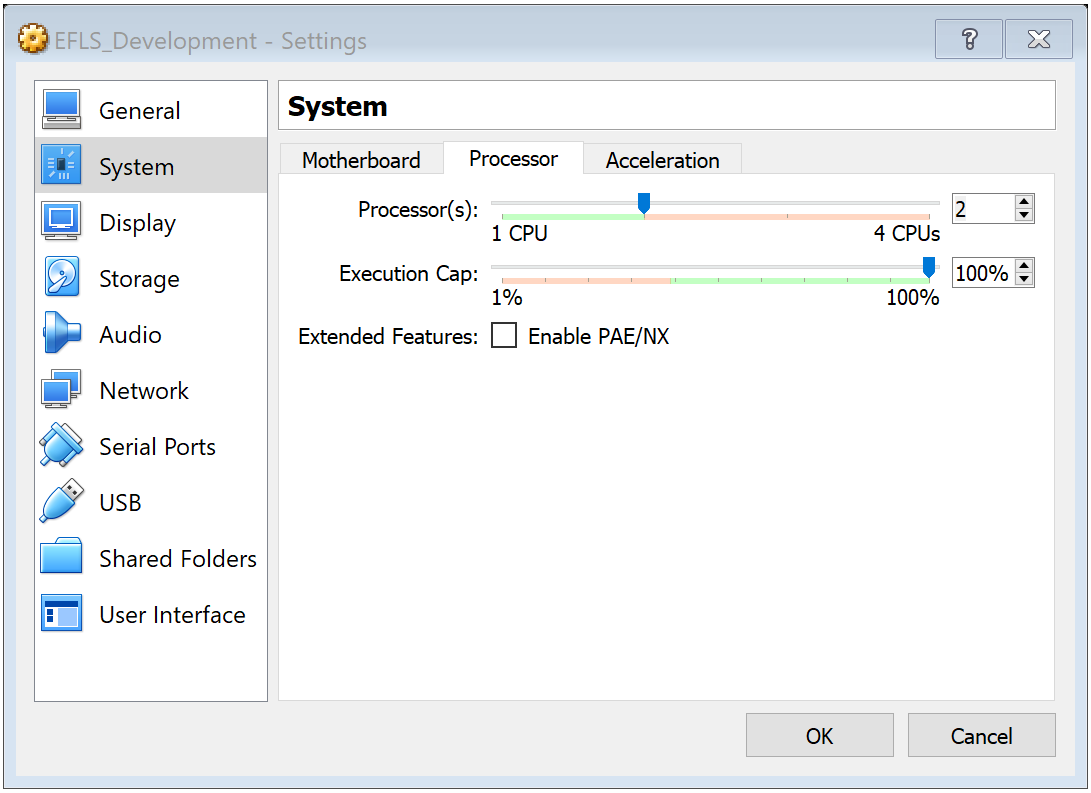


6. Click on Expert Mode, to see the advance menu options. Select a File Location if you would like to change it from the default, set the File size to "60GB", Hard disk file type to "VDI", Storage on physical hard disk to "Dynamically allocated", and then click Create.



7. The VM has not been created, however it needs to have additional settings changed. Select on the VM you just created, and then click on Settings.

8. Under the System>Processor, change the Processor(s) to a minimum of 2. Increasing the Display video memory and enabling 3D acceleration may be required if your system is performing slowly later on. Click OK to save these changes.



9. Double click on the VM you created before, this should start up the VM and open up additional configuration windows as this is the first boot.

10. Before selecting your start-up disk, we need to link the Ubuntu iso file previously downloaded. This is done by selecting Devices>Optical_Drives and then finding the .iso file of Ubuntu. After its selected, click start on the Select start-up disk. This will start the installation process.

11. Select Install Ubuntu.

12. Select “Download updates while installing Ubuntu” and “Install third-party software…”.

13. Select “Erase disk and install Ubuntu”. As long as you are running a VM, it will not affect your actual host PC disk. Click Install now to continue.

14. Proceed with the following account settings setup.

15. Restart after the installation, and then login to your new VM.

16. To ensure that all packages are updated to the latest when downloading, use the following commands:

    sudo apt-get update
    sudo apt-get install

8.2 MavProxy

MavProxy is a command line based ground control station (GCS) for MAVLink based autopilots, specifically ArduPilot. It provides a lot of usefully features as a GCS however it also has a lot of benefits to run on a companion computer on the UAV, as it acts as a MAVLink packet interpreter. As MavProxy allows the additions of third-party modules, the EFLS utilises this to communicate to the UAV through MavProxy, thus meaning the complicated MAVLink packets do not need to be decoded/encoded by EFLS.

There are two options for installation of the required MavProxy module, this could be by copying it over to a current master build of MavProxy or just downloading MavProxy directly with these added features. For simplicity of this tutorial, the second method will be elaborated upon.

8.2.1 MavProxy Installation with built in EFLS Support

The below installation process for MavProxy is based upon http://ardupilot.github.io/MAVProxy/html/getting_started/download_and_installation.html and http://ardupilot.github.io/MAVProxy/html/development/mavdevenvlinux.html.

1. MavProxy prerequisites packages are required to be installed first. For a Debain based system (eg. Ubuntu) the following can be used:

   sudo apt-get install python-dev python-opencv python-wxgtk3.0 python-pip python-matplotlib python-pygame python-lxml
   sudo apt-get install python-setuptools python-future

   For Raspberry Pi (Raspian) systems, the following command is also required:

   sudo apt-get install libxml2-dev

2. Git will need to be installed, so we can clone the repository:

   sudo apt-get install git

3. After git has been installed, the preinstalled EFLS MavProxy repository can be copied. Ensure your current working folder is set to your user accounts Home.

   git clone https://github.com/dell-o/MAVProxy.git

4. These following settings might be required, to set the system path for MAVProxy and uder permissions:

   echo "export PATH=$PATH:$HOME/.local/bin" >> ~/.bashrc
   sudo adduser <username> dialout

5. Due to a permission problem with this system setup, you must manually change the permission of the .local/lib folder to allow MAVProxy access. Ensure you are in your Home folder.

   sudo chmod 777 .local/lib -R

6. For the first time and every time after changing a MAVProxy module, the following setup script is required to be run. The setup script is located inside of MAVProxy folder.

   sudo python setup.py build install --user

7. MAVProxy has now been successfully installed. If your running SITL simulation only, then no further commands are required as SITL will automatically open MAVProxy. If your using this in the UAV, then MAVProxy will need to be opened via the command of:

   mavproxy.py

   Additional configurations and advanced user information is available at http://ardupilot.github.io/MAVProxy

8.3 SITL

Software In The Loop (SITL) simulation of the ArduPilot code base allows a real time simulation of the ArduPilot aircraft, with the exact source code that would be used on the UAV. The flight model is provided by the JSBsim, and the GCS used in this case is MAVProxy which also allows the EFLS software to communicate with the simulation.

SITL provides the ability to test EFLS with the UAV software in many different test locations before implementing on an actual UAV. This can help configure the EFLS parameters and see there affects in practice, additionally test for bugs prior to flight.

8.3.1 SITL Installation

The following tutorial is based on http://ardupilot.org/dev/docs/setting-up-sitl-on-linux.html.

1. A copy of the ArduPilot git repository is required, this can be downloaded with the commands as shown below:

   git clone git://github.com/ArduPilot/ardupilot.git
   cd ardupilot
   git submodule update --init --recursive

2. JSBSim is required to be installed to allow a physics model for ArduPlane, this can be installed with the following commands in your Home dictionary:

   cd
   git clone git://github.com/tridge/jsbsim.git
   sudo apt-get install libtool libtool-bin automake autoconf libexpat1-dev

3. If an error says you require a newer version of JSBSim, then it can be updated with the following commands:

   cd jsbsim

git pull
./autogen.sh --enable-libraries
make

4. For a Debian based system, the following prerequisites are required:

   sudo apt-get install python-matplotlib python-serial python-wxgtk3.0 python-wxtools python-lxml
   sudo apt-get install python-scipy python-opencv ccache gawk git python-pip python-pexpect
   //sudo pip install future pymavlink MAVProxy

5. Some directories are required to be added to the search path. Se a text editor to add them to the .bashrc file located in the Home directory.

   export PATH=$PATH:$HOME/jsbsim/src
   export PATH=$PATH:$HOME/ardupilot/Tools/autotest
   export PATH=/usr/lib/ccache:$PATH"

6. The PATH can reloaded with the “dot” command:

   . ~/.bashrc

7. SITL can be scripted to run automatically as described later, however a single simulation can be performed with the following command:

   cd ardupilot/ArduPlane
   sim_vehicle.py -w

   Following runs of the SITL only require the command below:

   sim_vehicle.py --console --map --aircraft test

For information about SITL and advance uses can be found at http://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html.

8.4 EFLS

The EFLS software is another git hub repository that is required to be downloaded to your computer. The EFLS software is a C++ based program that uses the GCC compiler configured with a CMake file. The majority of the prerequisites are already apart of the Ubuntu package or hopefully your Linux distro. Other prerequisites have already been installed such as MAVProxy and SITL. A few more are however required to be installed, for full functionality.

8.4.1 CURL - Installation

Curl is a download tool used by EFLS to fetch data from the internet when it's not available within the cache folders. The installation is shown below and was adapted from https://curl.haxx.se/docs/install.html.

1. Download curl from the source archives located at https://curl.haxx.se/download.html in the tar.gz format. Version 7.55.1 has been tested and confirmed to work.

2. Extract the downloaded file with the following command:

   cd Downloads
   tar -xvzf curl-7.55.1.tar.gz

3. Enter into the extracted folder to continue with the build process for curl:

   cd curl-7.55.1

4. Build curl and install with the following commands:

   ./configure
   make
   make test
   make install

More information about curl and advanced configuration can be found at https://curl.haxx.se/.

8.4.2 Google Protocol Buffer

Google's protocol buffer (ProtoBuf), is used to allow simple and efficient communication between the EFLS software and MAVProxy. As both of these programs run in different programming languages, a language neutral communication protocol is required and hence why Protobuf was chosen.

ProtoBuf works by communicating through a common binary file between the two programs, using the simple technique of reading and writing to the file. For EFLS to be compiled and work, ProtoBuf must be installer for both C++ (EFLS) and Python (MAVProxy). For additional flexibility, the instructions will cover the full installation process to allow protoc to be installed hence allowing the ProtoBuf message definition to be updated as required.

The below instructions have been adapted from https://github.com/google/protobuf/blob/master/src/README.md and

https://github.com/google/protobuf/blob/master/python/README.md. Version 3.4.1 has been tested and proven to work successfully.

1. Download the tar.gz folder for Python at https://github.com/google/protobuf/releases/tag/v3.4.1.

2. Unzip the folder with:

   cd Downloads
   tar -xvzf protobuf-python-3.4.1.tar.gz

3. Install prerequisites for protoc installation:

   sudo apt-get install autoconf automake libtool curl make g++ unzip

4. Generate configure script:

   cd protobuf-python-3.4.1
   ./autogen.sh

5. To build and install protoc, use the following commands. This takes a significant amount of time to compile.

   ./configure
   make
   make check
   sudo make install
   sudo ldconfig # refresh shared library cache.

6. Test the installation has worked successfully by accessing protoc with the below command:

   protoc --help

7. The next step is to install the Python side of ProtoBuf. This is much quicker than the prior install.

   cd python
   python setup.py build
   python setup.py test

8. If this installation does not work with MAVProxy and shows an error saying that the google.protobuf module could not be loaded, then run the following command followed by a restart:

   sudo pip install protobuf

This should have now installed both C++ and Python sides of the Protocol Buffer, and thus allowing the EFLS software to communicate to MAVProxy. For more information about ProtoBuf refer to https://developers.google.com/protocol-buffers/.

8.4.3 OpenCV – Installation

OpenCV is the image recognition library used throughout the EFLS software. It provides a large amount of image recognition methods and additionally provides convenient methods of handling the data in a structured format.

For EFLS to be compiled, OpenCV must be installed. The recommended version is the current latest release of 3.3.0. Future release should also be compatible however due to the core nature of this library to EFLS recognition, different versions should be thoroughly tested to ensure stability, accuracy, and performance. The below installation process was based on http://docs.opencv.org/2.4/doc/tutorials/introduction/linux_install/linux_install.html:

1. Download the source code (tar.gz) from https://github.com/opencv/opencv/releases/tag/3.3.0.
2. Install prerequisites for OpenCV:

   sudo apt-get install build-essential
   sudo apt-get install cmake git libgtk2.0-dev pkg-config libavcodec-dev libavformat-dev libswscale-dev
   sudo apt-get install python-dev python-numpy libtbb2 libtbb-dev libjpeg-dev libpng-dev libtiff-dev libjasper-dev libdc1394-22-dev

3. Extract the files from the downloaded source code:

   cd Download
   tar -xvzf opencv-3.3.0.tar.gz

4. To build OpenCV, the following commands are required to generate the required make files:

   cd opencv-3.3.0
   mkdir release
   cd release
   cmake -D CMAKE_BUILD_TYPE=RELEASE -D CMAKE_INSTALL_PREFIX=/usr/local ..

5. As the make files have been generated, now located and enter into the created *cmake_binaray_dir*:

   make
   sudo make install

8.4.4 Libosmium – Install

Libosmium is a header library used to read and process OSM data (street map data), in the EFLS software. While a header library does not require to be installed, it does have some specific prerequisites required to be installed.

1. Install prerequisites:

   sudo apt-get install cmake cmake-curses-gui make libexpat1-dev zlib1g-dev libbz2-dev
   sudo apt-get install libsparsehash-dev libboost-dev libgdal-dev libproj-dev doxygen graphviz

8.4.5 Git hub LFS – Install

Git hub LFS is required to download some large files within the repository. These files could be added later if LFS cannot be installed, however for ease its recommended installing it.

1. Go to https://git-lfs.github.com/ and click on the download button.
2. After downloading, the following commands can be used to extract and install Git LFS:

   cd Downloads
   tar -xvzf git-lfs-linux-amd64-2.3.1.tar.gz
   cd git-lfs-2.3.1
   sudo git ./install.sh
   git lfs install

8.4.6 Clone EFLS

EFLS can be cloned from the Git hub repository. Cloning is the recommended method of downloading, as new updates can be quickly applied to the copy with the Git pull command.

1. Git hub and Git hub LFS must be installed first. This should already have been completed with a previous section of this tutorial.

2. The following command can be used to clone EFLS into your Home directory:

   git clone https://github.com/dell-o/EFLS

This should download the EFLS source code to your computer ready for building and/or developing.

Please note that due to EFLS using the GPL-3.0 license, all development using the EFLS software must be then uploaded back to the master EFLS or released as open source code.

8.4.7 Build EFLS

Once EFLS has been cloned from the Git hub repository, the build files can be generated and then EFLS can be built. This is done by using CMake and Make.

1. Enter into the EFLS folder:

   cd EFLS

2. Make and enter into the build location:

   mkdir build
   cd build

3. Generate the Make files using the CMake script provided:

   cmake ..

4. Compile the C++ code using Make:

   make

The above commands should have successfully compiled the EFLS source code into the required C++ code for your system. For future changes to the EFLS C++ code, only the *make* command will be required to run to recompile the code, unless you have added additional C++ files to the project.

8.4.8 Running EFLS

The EFLS code will be automatically run through the MAVProxy module "efls" when required or by the SITL simulation script for multiple simulations. EFLS can also be run separately for testing if required, this is shown below:

1. Run SITL and the MAVProxy module "efls" to generate some data inside of the protocol buffer files. Without this data, EFLS will not know any information about the aircraft and will fail to run.

2. Run the EFLS code by entering into the build folder and then running the following command:

   ./EFLS

This will run EFLS with a verbose message output, updating you on the current item being processed.

8.5 Using an IDE for editing and compiling

Eclipse is a useful IDE that can be used to edit and modify the EFLS source code. Eclipse was used to construct the project and has been confirmed to work successfully with the entire project except for the chrono time library used in the FileWriter class for performance analysis, nevertheless this is only an inconvenience while editing that particular method.

8.5.1 Eclipse - Installation

Eclipse Luna has been tested and proven to work successfully with EFLS. Eclipse can be installed with the following commands:

1. Download the tar.gz install files from http://www.eclipse.org/downloads/packages/eclipse-ide-cc-developers/heliossr2.
2. The Java jdk is required to be installed as a prerequisite for Eclipse, this can be done with the following command:

   sudo apt-get install openjdk-8-jdk

3. Exact the Eclipse install files:

   cd Downloads
   tar -xvzf eclipse-cpp-luna-R-linux-gtk-x86_64.tar.gz

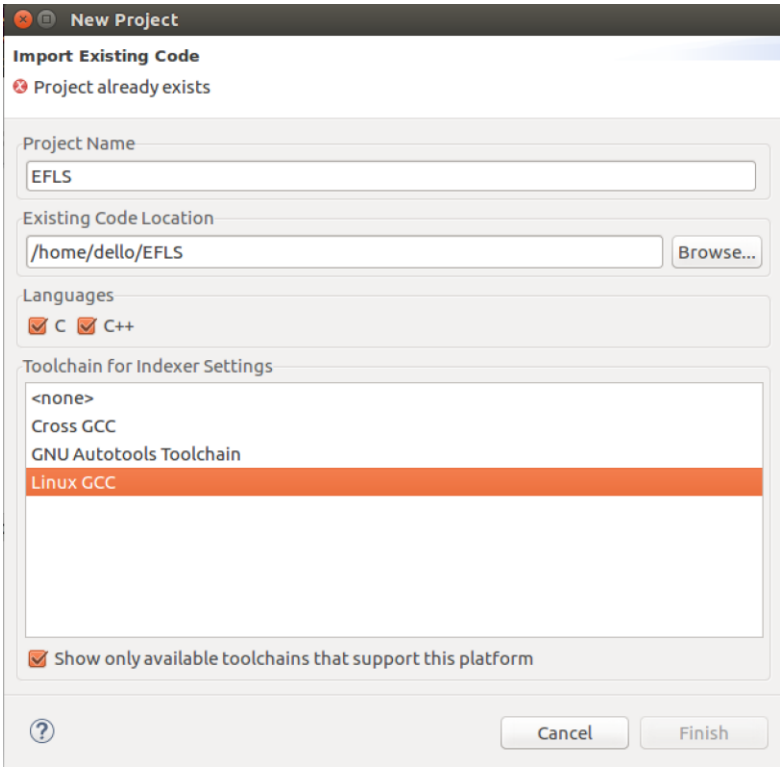
4. To open Eclipse use the following command:

   cd eclipse
   ./eclipse

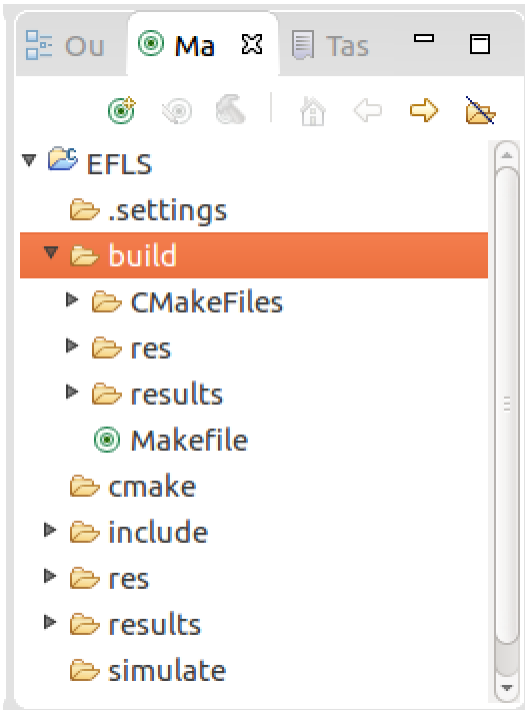
8.5.2 Eclipse - EFLS setup

The setup for EFLS with Eclipse is quite simple if using the make file generated by cmake. Using this method is easier to setup, however if any additional .cpp files are added or advanced cmake scripting is required then the cmake file will have to be manually changed and then rerun to generate the updated make file.

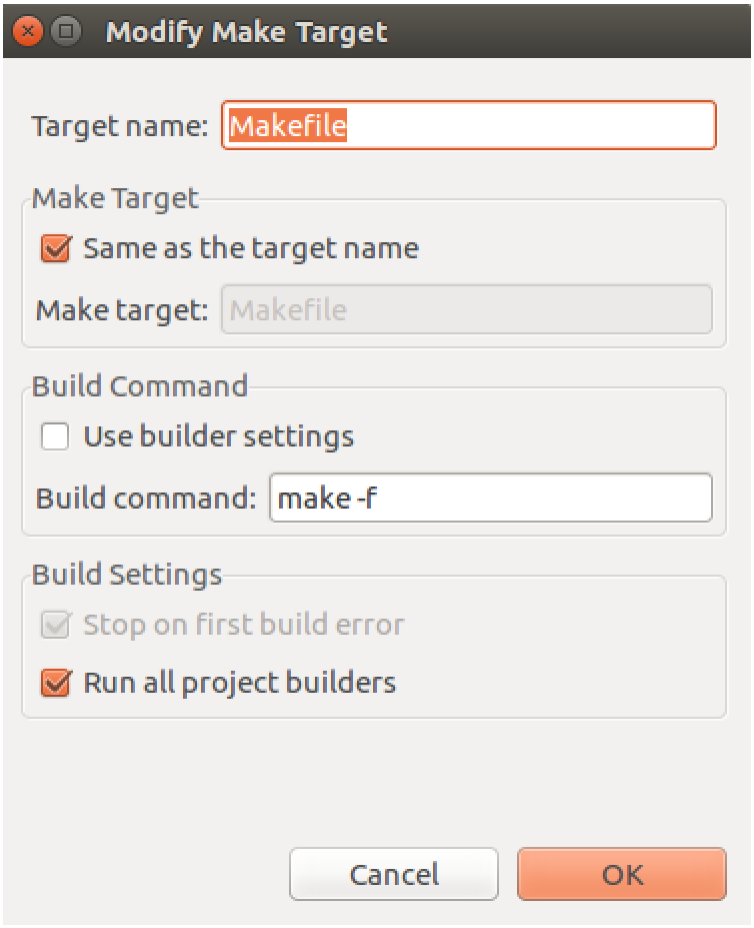
1. Ensure the cmake file has generated the make file located at EFLS/build/Makefile.
2. Open eclipse.
3. Go to File >> New >> Makefile Project with Existing Code
4. Enter into the wizard the project name, code location, and choose the Linux GCC compiler option. Click finish, and this should bring the project into Eclipse. Let the indexer finish before continuing.



5. Enter into the build folder located in the Make Target window. Click on the build folder, and then click on the New Make Target button.



6. In the New Make Target wizard, set the Target Name to Makefile. Deselect the Use builder settings and write the build command of *make -f.* Click OK to save your settings.



7. To build the EFLS code, just click on this build target

8.6 Raspberry Pi - Installation

Using a Raspberry Pi 3 is a recommended hardware solution for running the EFLS software on a UAV platform. This is due to the Raspberry Pi 3 being light weight, low power consumption, small physical size, and offering one of the highest computation ability in its class.

The installation process for the Raspberry Pi is very similar to the approach taken to install EFLS on the virtual machine, as cross-compiling is not used.

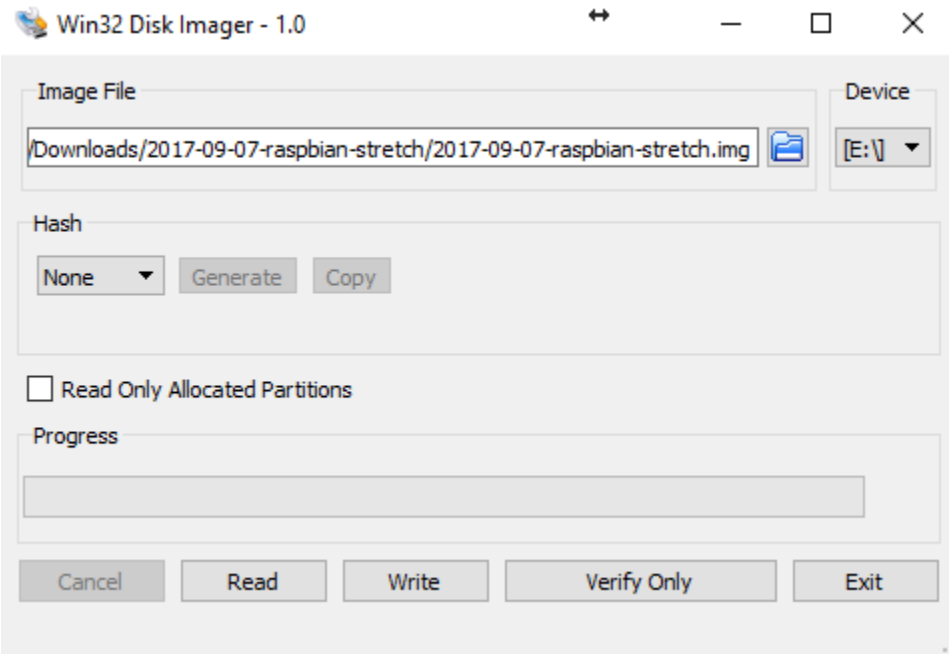
8.6.1 Install the OS

Raspberry Pi OS installation instructions have been based from the information provided at https://www.raspberrypi.org/documentation/installation/installing-images/README.md. A Windows operating system has been assumed for the following instructions.

1. Download the RASPBIAN OS with Desktop, located at https://www.raspberrypi.org/downloads/raspbian/.

2. Download WIN32 Disk Imager, located at https://sourceforge.net/projects/win32diskimager/.

3. Install WIN32 Disk Imager, by double clicking the downloaded file. Follow the install wizard.
4. Locate the downloaded RASPBIAN OS which should be in a ZIP folder. Unzip the files within this folder.
5. Insert a Micro SD card into your computer. Ensure that the Micro SD card has no files on it as they will be erased during the imaging process.
6. Open WIN32 Disk Imager program. Under Image File, select your extracted ISO of RASPBIAN that was just extracted. Carefully select your Micro SD card in the device drop down. WARNING: Make sure you have selected the correct device, if not then you could delete your whole computer!!!



7. Double check that you have selected your Micro SD card and not another device.
8. Click the “Write” button to write the OS image.

After this is process has been completed, we will now setup headless mode for the Raspberry Pi which will allow direct SSH after powerup.

9. Open up the Micro SD Card from the file explorer. It should be name “boot”.
10. Create a file name with “ssh” with NO file extension. This will tell the Raspberry Pi on first bootup to enable SSH.

If completed successfully, the SD card is now ready for the Raspberry Pi.

11. Insert the Micro SD card into the Raspberry Pi and connect a ethernet cable to a network that will allow you SSH access to the Raspberry Pi, for example connect it to your home router.

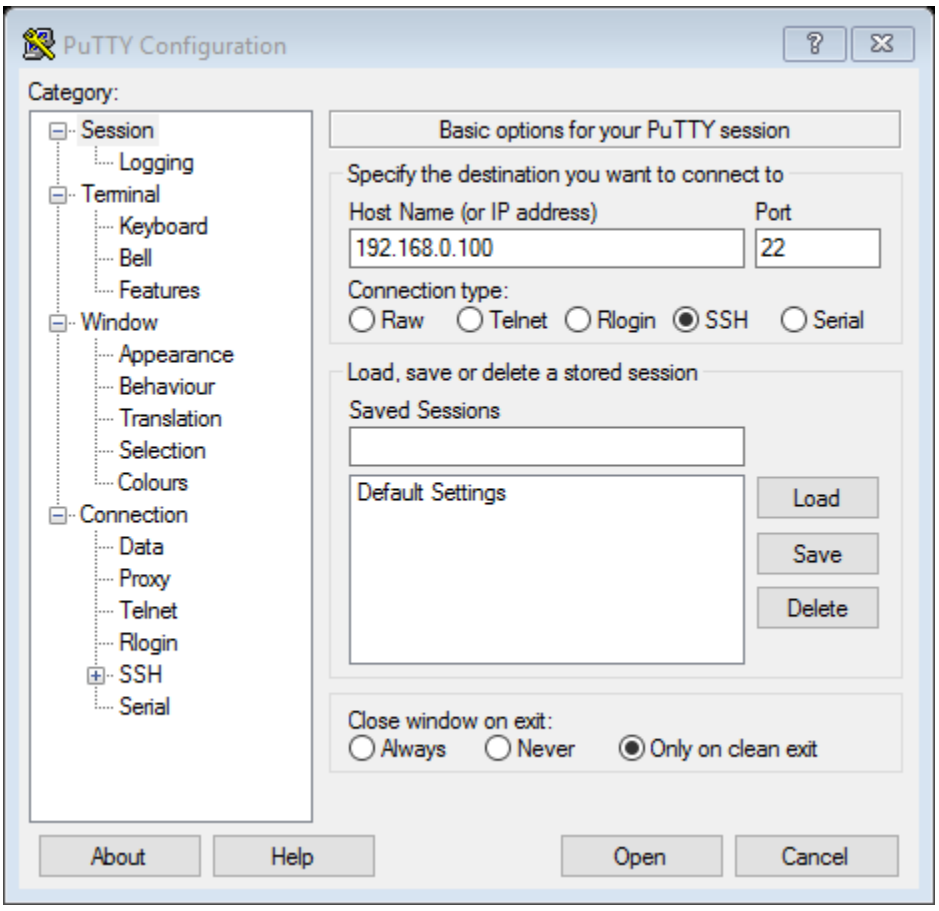
12. Powerup the Raspberry Pi using the Micro USB cable connected to a 5V 2.1A or greater power supply.

13. Wait for 5 minutes before proceeding. This will ensure the first-time boot process has been completed. The green status light should also stop flash regularly.

14. Open up your home router settings and find a recently added device under the DHCP client list named “raspberrypi”. Note down the IP address of this device.

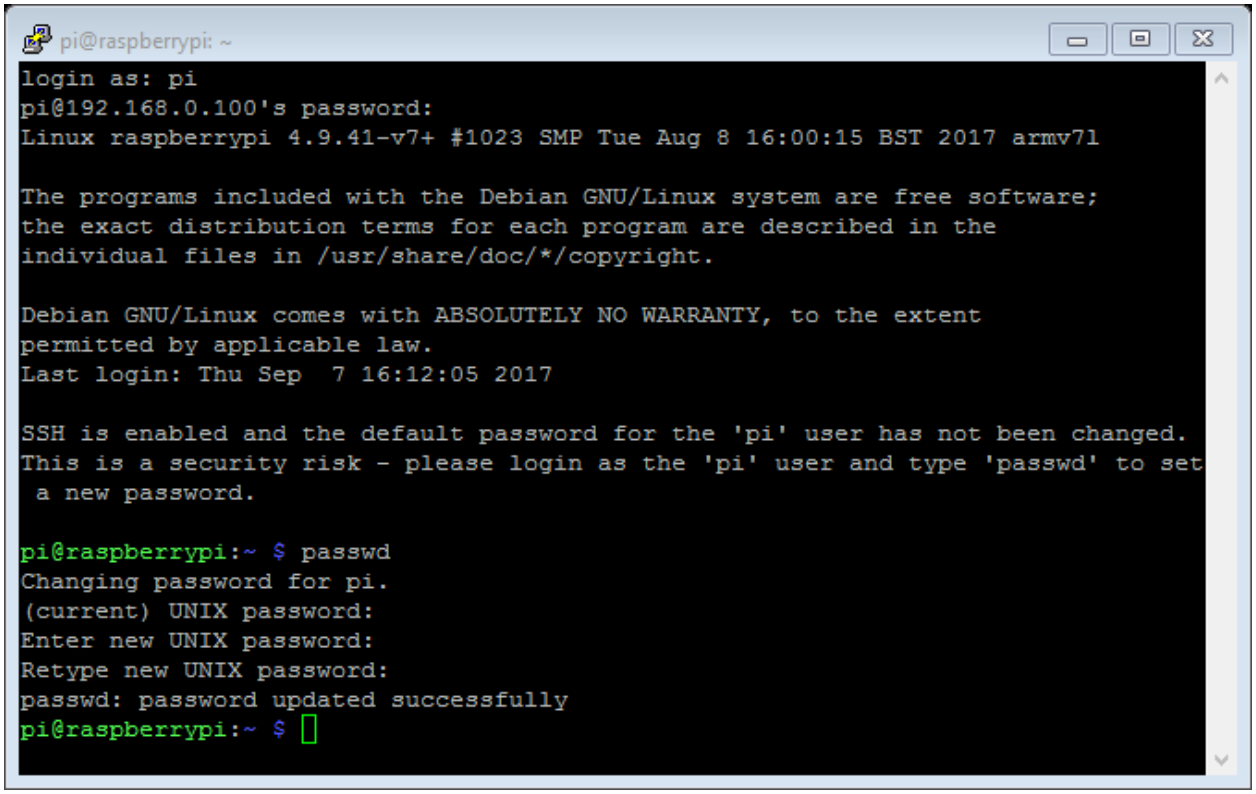
15. Download and install a SSH client to access the Raspberry Pi SSH host, such as PuTTY. https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html

16. Open PuTTY and type in the IP address previously found as the “Host Name”. Click “Open” to open the connection.



17. A terminal window should now be open with the word “login as:” at the top of the screen. The account username is “pi” and the password is “raspberry”. Make sure to change the password after first login.

18. The terminal should look like the image below, if so you have been successful.



8.6.2 MAVProxy & EFLS Install for the Raspberry Pi

The above sections of (8.2 – MAVProxy Install) and (8.4 – EFLS Install) can be followed. It is not recommended to install SITL or a IDE on the Raspberry Pi as it has not enough computation power to run smoothly.

8.6.3 EFLS Install for the Raspberry Pi

EFLS can be installed solely on the Raspberry Pi to remove the overheads generated by MAVProxy running as the MAVLink interpreter. This is recommended if it is feasible to have multiple companion computers on the UAV or alternatively a larger companion computer such as a NUC could be used. This setup will only be explained in theory as no current testing has been conducted in this configuration.

The setup requires the two companion computers to share a network folder. An easy method is to connect the two ethernet ports of the companion computers and setup an internal network. Once the network and shared folders have been established, the protocol buffer message files of “aircraftLink_medium_aircraft” and “aircraftLink_medium_waypoints” should be moved into the shared folder. Both the MAVProxy EFLS module and the EFLS software will require to be changed to point to this new location of the shared folder. It should also be noted, that for the current version of EFLS an internet connection will be required for both companion computers which can be achieved though forwarding it from one companion computer to the other.

8.6.4 Other Installation Notes for the Raspberry Pi

As it is recommended above to only use a SSH connection to configure the Raspberry Pi, a few issues occur due to this however there are a few simple solutions to handle these problems. Additionally, there are some other issues with the Raspberry Pi installation that also need to be handled.

- The instructions for MAVProxy and EFLS require downloading some compressed source files from the internet. Instead of using a web browser the "wget" console command can be used to fetch the required files. For example:

  wget "http://download.geofabrik.de/australia-oceania/australia-latest.osm.pbf"

- If using the turbo_sim simulation programs for EFLS testing, it seems that the Raspberry Pi python installation does not come with the pexpect module by default. This can be fixed by downloading and installing it with the following command:

  sudo pip install pexpect

- The Git LFS installation process for a Raspberry Pi is very convoluted and painful. It might be easier in most cases to download the OSM maps via "wget" and putting them into the correct resource folder and renaming it to the required name.

The hardware that can be used with EFLS is only limited by if it can run Linux and if it has some form of method to communicate with the autopilot (eg. Serial or USB). The performance of EFLS allows it to be used on pretty much any type of computer. The Raspberry Pi 3 has been used for testing and is the minimum recommended companion computer when integrating EFLS into a UAV platform. While more powerful companion computers would be useful to run EFLS faster, the Raspberry Pi 3 has a great community with a lot of support thus allowing for ease in problem shooting.

The Raspberry Pi 3 connects to the Pixhawk via its GPIO header pins 14 and 15 shown in the diagram. The Ground (GND) should also be connected between the companion computer and the PixHawk. This connects to any of the PixHawks serial ports that are available.

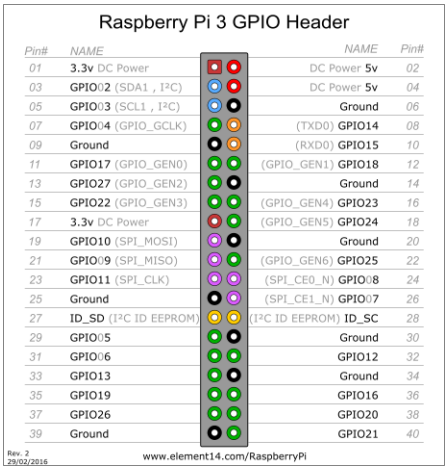


Figure 16: Courtsy of Element14

Figure 17: PixHawk 2 - Courtsy of ProfiCNC

There is come slight software configuration required on both devices to ensure the serial link is setup correctly and robust. The Raspberry Pi 3 requires the following commands to setup the serial link:

sudo nano /boot/config.txt
Add the line at the bottom:
enable_uart=1
Save and exit.

sudo systemctl stop serial-getty@ttyS0.service
sudo systemctl disable serial-getty@ttyS0.service
sudo nano /boot/cmdline.txt
remove the line of console=serial0,115200.
Save, exit, and then reboot.

sudo nano /boot/config.txt
Add the line at the bottom:
dtoverlay=pi3-miniuart-bt
Save, exit, and reboot.

For more information on the Raspberry Pi 3 serial setup:
https://spellfoundry.com/2016/05/29/configuring-gpio-serial-port-raspbian-jessie-including-pi-3/

The PixHawk 2 flight controller can now be setup with its serial configuration using the below commands. These parameters can either be changed with your ground control station of your choosing, or with MavProxy with the command *param set parameter_name parameter_value*. The # is replaced with your chosen serial number on the PixHawk 2.

SERIAL#_BAUD = 115
SERIAL#_PROTOCOL = 1

This serial link should now be tested with just opening MavProxy on the companion computer. MavProxy should automatically detect the serial port and connect to it with the correct baud rate.

More information can be found about setting up a companion computer with ArduPilot (PixHawk) at:
http://ardupilot.org/dev/docs/raspberry-pi-via-mavlink.html

10 EFLS Parameter Configuration

EFLS contains various parameters for configuration on the specified UAV platform. These parameters are based upon multiple factors such as the UAV flight envelope, its required landing area, and other factors. The below section will provide a helpful resource of understanding each parameters purpose and how to configure the parameter.

Warning: It is highly recommended that a simulation is conducted with either SITL or TurboSim as discussed in Section 11, to ensure that the parameters set provides suitable performance.

10.1 Scan Parameters

The following parameters are related to the scanning algorithms of the EFLS code.

10.1.1 Resolution of scan

The resolution parameters are used to define the resolution of the satellite image that will be processed. This size is then used for the street map and terrain data, to convert their data to a binary array of the same size of the satellite image. The resolution will affect the processing time of EFLS, as a larger image will require more time to process. 3000 is the default value for both x and y, as it contains enough coverage. When setting this parameter also consider *search_zoom.*

- resolution.x = 3000;
- resolution.y = 3000;

10.1.2 Zoom of scan

The zoom value is used to adjust the amount of area covered by a single scan, however, the trade-off being the detail of the satellite image. The zoom of the scan is defined by the equation of:

$$zoom = \log_2\left(\frac{PixelPerMeter \times 2\pi \times re \times \cos(latitude)}{256}\right)$$

As seen in the equation, the zoom level is defined by the earth radius (re), the latitude, and the Pixels Per Meter. The only value which the user change decided is the pixel per meter value, which is recommended to be around 0.5 pixels per meter or greater. The zoom value must be a integer between the range of 0 and 18.

- search_zoom = 16;

10.1.3 Land site parameters

The land site parameters define the minimum dimensions of the landing site. This is determined by the UAVs landing profile and is specific to each UAV. It should be noted, that the decision engine will automatically try to find the largest site possible, this value just determines the minimum accepted landing site. The UAV will attempt to touch down in the middle of this landing site. The units of measurement are in meters.

- runway_length_init = 300;
- runway_width_init = 100;

10.1.4 Aircraft general parameters

The aircraft general parameters allow EFLS to account for the UAVs flight envelope and additional factors. The glide slope parameter should be set to the UAVs defined glide slope with an engine failure, the glide safety factor is used to allow EFLS to account for some minor factors that will reduce the glide range of the UAV, and the estimated_processing_time is the mean processing time of EFLS to find a suitable landing site as EFLS needs to know the amount of time lost in altitude during processing.

- glide_slope = 8;
- glide_safetyFactor = 1.1;
- estimated_process_time = 60;

10.1.5 Landing final approach parameters

The landing final approach parameters are used to define the last stage of the final approach in both distance from selected landing site and altitude. The two points which can be configured with these parameters is the last two waypoints which are directly inline with the landing site. The other points can be changed using the *Landing final approach arc parameters*.

- landFinalApproachDistance = 300;
- landFinalApproachAltitude = 20;
- landEndTurnDistance = 200;
- landEndTurnAltitude = 30;

10.1.6 Landing final approach arc parameters

The arc parameters are used to calculate the desired turn to align the UAV with the runway. The *view* parameters define the angle between two consecutive waypoints. The *distance* parameters define the distance between each consecutive waypoint. The *partial accept* parameters are used to accept a landing arc that overlaps with the runway. The *rate of descent* parameter is used to calculate the altitude of the waypoints during the turn with the ratio of the glide slope desired. The *min waypoint* sets the minimum number of waypoints the algorithm should plot for the arc, this fixes the issue of incorrect decent profile when the UAV is already aligned with the runway.

- landArcViewTurn = 70;
- landArcViewFinal = 40;
- landArcDistance = 150;
- landArcDistanceFinal = 200;
- landArcPartialAcceptDistance = 100;
- landArcPartialAcceptAngle = 30;

- landArcRateOfDescent = 2;
- landArcMinWaypoint = 3;

10.1.7 Engine failure parameters

These parameters are used to define if an engine failure has occurred. This is calculated by if the throttle is above the set threshold value, then the current of the motor should not be below the minimum value. If it is below the minimum current value, then an engine failure has been detected.

- throttle_threshold = 30;
- current_min = 3.0;

10.1.8 Decision engine parameters – Random search

The random search method of the decision engine defined using these parameters. The *attempts* define the number of random searches should be conducted for every iteration of the decision engine. The *max and min distance* define where the random search will find a random location. The *min range* defines if the random search should be used, dependent on the estimated range of the UAV.

- randomLoc_attempts = 1;
- randomLoc_maxDistance = 10000;
- randomLoc_minDistance = 3000;
- randomLoc_minRange = 5000;

10.1.9 Decision engine parameters – Large search

The large search method of the decision engine is defined using these parameters. The *zoom* parameter is used to set the satellite image size, which needs to be set smaller than the value used for the regular search. The *runway* parameters define the landing site dimensions in meters, and since this is search a large area with low pixel per meter value, this site is recommended to be quite large. The *min range* parameters defines if this search method should be used, based upon the current estimated range of the UAV.

- largeSearch_minRange = 20000;
- largeSearch_zoom = 14;
- largeSearch_runwayLength = 1000;
- largeSearch_runwayWidth = 1000;

11 EFLS Simulation

The simulation component of EFLS is important to its safe setup and operation. The simulations ensure that EFLS is operating correctly on the installed hardware and the parameters used provide the desired outcome.

There are two different simulation types for EFLS. The first is TurboSim which is aimed at rapid testing of EFLS at various locations and the second is SITL which is aimed at stimulating EFLS with an entire UAV platform.

11.1 TurboSim

TurboSim is a custom-made simulator specifically designed for EFLS. It uses the python pexcept module to allow a scripting functionality. TurboSim is aimed at rapidly testing EFLS with a large amount of different locations without the overhead of the UAV software. The output from these simulations are from EFLS itself, with its inbuilt timing profiler and saved processed images and selected locations.

11.1.1 Running TurboSim

TurboSim comes within the download of EFLS from GitHub and is located in the *simulate* folder.

The locations for EFLS to simulate are located in the locations.csv file. The only information required for each location is the ID number, latitude, and longitude. The current list contains 300 different locations, evenly split-up into different settlement sizes of city, town, and country areas. Additional values can be easily added though the use of Excel to read and edit the .csv file, and the use of https://www.findlatitudeandlongitude.com/click-lat-lng-list/ to quickly get a large list of coordinate locations.

There are three different TurboSim applications available for use, with slightly different functions. The first is the original, turbo_sim_EFLS.py, which simulate all the locations within the locations.csv file. The second, turbo_sim_EFLS_distribute.py, is used to test a defined number of locations multiple times, which is useful for working out the processing time distribution for the same set of locations. The third, turbo_sim_EFLS_range.py, is used to test a defined number of locations over a range of different altitudes, which is useful in determining EFLS behaviour dependent on the estimate range of the UAV.

To run TurboSim, the following python command can used:

```
cd EFLS/simulate
python turbo_sim_EFLS.py
```

11.1.2 Results analysis

TurboSim uses the results automatically generated by EFLS, and then saves them into a test folder for later viewing. Additionally, the custom time profiler built into EFLS automatically runs and the results are also saves with the other results. The results can be located in the EFLS/build/results/ folder.

The time profiler generates a text document with a detailed analysis on the time taken to complete each part of the code during each simulation. These results can then be analyses within MATLAB with a custom text file reader that decodes the time profiler document. The time profiler document has been designed to be human readable for ease of quick analysis.

The landing sites that EFLS has selected can be viewed directly in the EFLS/build/results/sites folder, with every simulation having a respective safe landing site image. The image showing the safe landing site is the used satellite image, with the selected landing site shown as a brighter area. This can then be viewed to check if EFLS has successfully found a safe landing site.

If a more detailed understanding of why EFLS selected a specific site is required, the TurboSim can be configured to save all detailed results generated by each EFLS simulation. This does use a significant amount of hard drive space, with 100 simulations estimated to be in excess of 10GB of data hence why its not enabled by default. It can be activated by uncommenting out the code within the TubroSim program. These results show the original satellite image and thee binary arrays generated by the satellite image module, map module, terrain module, and the site selection module. If the decision engine runs more than a single search per simulation, the other searches are also saved within their separate folders.

11.2 SITL

SITL simulation allows the full UAV flight controller software to be simulated with a flight model of the UAV. This thus allows an identical test of EFLS under the same operating conditions while on the UAV. This is important as it confirms the communication of EFLS to the flight controller is suitable and also if the desired performance and flight path can be achieved and practical. The setup for SITL is shown in the instruction manual section of this document.

11.2.1 Running of SITL

The locations for SITL to simulate at are defined in the ardupilot/Tools/autotest folder. It is recommended to expand the list with additional locations, to copy the locations.txt file located in the EFLS/simulate folder to the autotest folder. This is necessary if running the SITL scripting simulations that are later discussed.

The manual SITL simulation can be activated with the commands below:

cd ardupilot/ArduPlane
sim_vehicle.py --console --map --aircraft test

If the location can be changed with the use of the -L command, which then you can define the ID of another location within the location.txt list to simulate at.

As SITL simulates the UAV directly, to operate the UAV and EFLS the instructions in Section 10 need to be followed.

For more information on SITL: http://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html

11.2.2 Results analysis

The best way of analysing the results, is to confirm that the desired UAV performance is being achieved with the outputs of the ground control station in real time of the simulated UAV. This will not only show if the desired path and flight profile on final decent is accurate, but it will also show if any miss configured parameters are causing issues to the EFLS selected sites and desired flight profile.

If the selected landing location results need to be analysed, they can be found in the EFLS/build/results folder.

11.2.3 SITL scripted simulation

SITL can also be scripted using the Python pexcept module, which has been done in the sitl_script.py program located in the EFLS/simulate folder. The script activates SITL automatically with the listed locations in the locations.txt files with the ID prefix of *Test*. This is ideal to test the UAV at multiple different locations with an accurate UAV model, however the overhead of the UAV to simulate extends these simulations so it's not feasible to test more than 50 within a reasonable time frame. The altitude of engine failure success rate can also be determined using the SITL script, which allows for an accurate understanding of EFLS performance during an engine failure based on the altitude of the UAV.

The SITL script also has a special functionality of teleporting the waypoints of a mission to the new location without any user steps required. It moves the waypoints based on the 0 waypoint location, with the new 0 waypoint location being the location that SITL is simulating at.

12 EFLS Practical Operation

As EFLS has only been integrated into ArduPlane currently, this section will focus on its operation with ArduPilot with a basic tutorial on operation of ArduPilot and MavProxy.

12.1 Activate EFLS prior to UAV flight

As EFLS is on the companion computer of the UAV, it is necessary to activate EFLS prior to each flight. An alternative method would be to write an automatic script to run MavProxy and EFLS when the companion computer boots up on the UAV, however this section will focus on the activate of EFLS manually.

The first step in the activation and checks of EFLS while onboard the UAV companion computer, is to start-up MavProxy with the command listed below:

mavproxy.py

This will automatically connect MavProxy to the serial link setup to communicate with the PixHawk autopilot. After connecting to the PixHawk, MavProxy will indicate its current status and the PixHawks current status. Ensure for increased performance not to load any other modules such as map, console, or missioneditor, as they are not required and will slow down the system performance. To load EFLS communication module, the following command can be used within the MavProxy terminal:

module load EFLS

EFLS should not report it has opened and has established a link to the communication protocol buffer files. It should be noted at this point that EFLS is not running at this point, only the MavProxy communication protocol to EFLS is running. To run the EFLS program, the following commands can be used in a separate terminal window:

cd EFLS/build
./EFLS

After running the above command, EFLS will now be running and continuously checking for an engine failure condition.

If EFLS does detect an engine failure conditions the following is the expected response of the system:

1. The EFLS program will display a warning that an engine failure condition has been detected.
2. The EFLS program will then provide details on its current progress on finding a safe landing site based on the satellite images, street maps, and terrain data. Once completed, it will list the waypoints it has sent to the EFLS communication protocol and state it has successfully found a safe landing site.
3. The MavProxy terminal window will now display that the EFLS communication protocol module has detected receiving new waypoints. The waypoints will then be listed in the terminal while also being sent to the UAV flight controller. The progress of the waypoints being sent to the

flight controller is also displayed, however since the flight controller is connected via a serial link it will be near instantons to transfer the waypoints.

4. The final response expected, is that the ground control station map of the flight path will now show the EFLS planned flight path toward the safe landing site.

13 EFLS performance analysis

The aim of the testing conducted is to confirm that EFLS works to the required specification. The same test methods used in this section can also be replicated to determine if the setup of EFLS with different parameters also provides suitable performance. The testing conducted as a part of this section is done with all default parameters and the default SITL aircraft model.

13.1.1 Different test method

There are three different test methods that can be used to identify that EFLS is working correctly and to analysis its performance. The methods are listed below:

1. The first of these test methods is the custom-made simulator TurboSim, which rapidly test EFLS with a large range of locations characterised by settlement size of city (greater than 300000 people), town (between 1000 – 300000 people), and country areas (less than 1000 people). This testing method is ideal for large quantity simulation without the worry of overheads generated by the UAV software.

2. SITL was used as the second test method. The aim of SITL is to simulate on a computer the identical software that runs on the UAV flight controller in real time, with the UAV flight being simulated within JSBsim. This allows for a near replica of the real UAV system, which is ideal for integration testing with the flight controller software and small quantities of simulations at various locations.

3. The final test method is running EFLS on an actual UAV system and conducting a test flight, however, this is a time-consuming process. A test flight is used to prove the simulated results within SITL are applicable to real-world cases, hence it is expected there is a high confidence of bug-free software and EFLS accuracy prior to a test flight.

13.1.2 EFLS reliability

Reliability was tested to ensure EFLS could operate when the UAV needs it during an emergency. The definition of reliability was, once activated EFLS would produce an output result that was failure free. This means that EFLS would produce its designated output, however, it would be irrelevant if this output was safe as this is determined in the accuracy testing. The reliability data was collected from the simulations conducted as a part of the testing for processing time and accuracy. This meant that three different test methods could be analysed for their specific reliability.

The first test method was with the TurboSim, where 900 different simulations were conducted on a range of settlement sizes. From the 900 simulations, zero failed to produce a suitable output, thus the mean reliability was 100%. The second test method used the SITL simulator, utilising 41 simulations conducted at a variety of settlement sizes. From the conducted simulations, zero failed to produce a suitable output, thus the mean reliability was 100%. The third test method was the use of the UAV with EFLS onboard. Over the three test flights conducted, zero failed to produce a successful landing based on the EFLS output, thus the mean reliability was 100%.

Over the three different tests methods utilised, it has been proven that the mean reliability is 100% with no failures during the entire test process. This indicates that EFLS is highly reliable at producing a successful output, however, all complex software has some form of bugs which decreases the reliability from the ideal performance as observed, but this was not observed within the current test population. To determine the mean with a confidence interval, continual testing must take place until a single failure is observed.

13.1.3 EFLS processing time

The processing time for EFLS to find a suitable landing site is an important test, as this amount of time the UAV could potential have an engine failure and would be continuously descending. The time used by EFLS to process a safe landing site, causes a reduction in the available flight range of the UAV, hence the shorter processing time the higher probability that a landing site is within range.

A control group was created to test the processing time on a consistent platform. The control group consisted of two different computers that are discussed in detail in Section 13.1.3.2. The control group had a mean performance difference of 57.0% with a standard deviation of 13.2%. All processing time testing was compared between the two control group computers and if abnormal results were observed outside of bounds defined by the mean and standard deviation, then a retest was commenced to ensure the results were consistent and not affected by other processes running simultaneously on the operating systems. This was only observed in three cases during the testing and repeating the tests showed that it was an isolated event that caused the processing times between the control groups to diverge. For testing where only one of the control group samples were required, the Surface Pro 4 processing time data was used.

13.1.3.1 EFLS Processing Bottleneck

As EFLS is aimed at processing as quickly as possible, a bottleneck in performance will exist. Identifying the current bottleneck will allow for future optimisation to be made to the software more effectively. The two initial test platforms were a Surface Pro 4 with Intel Core i5-6300U, 8GB RAM, and 256GB SSD and a custom built Gaming PC with Intel Core i7-3930K, 12GB RAM, and 256GB SSD. The two computers were compared using the UserBenchmark software, and it was found that the Gaming PC had a significantly faster CPU, however, the Surface Pro 4 had a faster SSD, with both having similar RAM performance.

EFLS was then tested on both computers using SITL at 41 different locations. It was found that the mean processing time for the Surface Pro 4 was 57.0% greater than the Gaming PC with a standard deviation of 13.2%. This result indicates that the current bottleneck is in CPU performance, as the Gaming PC CPU is approximately 75% faster than the Surface Pro 4 CPU.

Using the identification of the CPU as the bottleneck for EFLS, future optimisations can look into optimising the number of loop iterations that EFLS makes, thus reducing the demand on CPU and achieving the largest performance increase for the amount of development time.

By conducting 41 simulations using SITL at different locations and settlement sizes, as well as the use of the custom time profiler, it was determined that the mean execution time was 4.17s with a standard deviation of 2.51s. It was additionally found that there was a correlation between settlement size and processing time. For the town and country areas no noticeable difference in processing time was observed, however, the city case had a 115% increase in processing time which is shown in Figure 18. This increase in processing time was found to be contributed heavily to the time required to process the street maps, as a city contains a lot more streets for a given area.

The total amount of processing time can be split up into six key groups as shown in the pie chart of Figure 19. While optimisation has been applied to the current EFLS code where possible, the map and site selection modules have had the least amount of optimisation. As seen in Figure 19 these two contribute to 84% of the total time required to process. By identifying the map and site selection modules as key areas for future optimisation, the most improvement can be achieved for the amount of development time. The overhead was calculated to be 10% of the total processing time, caused by writing large amounts of data to the hard disk for later results analysis. If operating EFLS without the need for detailed results analysis of the processed data, a 9% increase in performance can then be made.

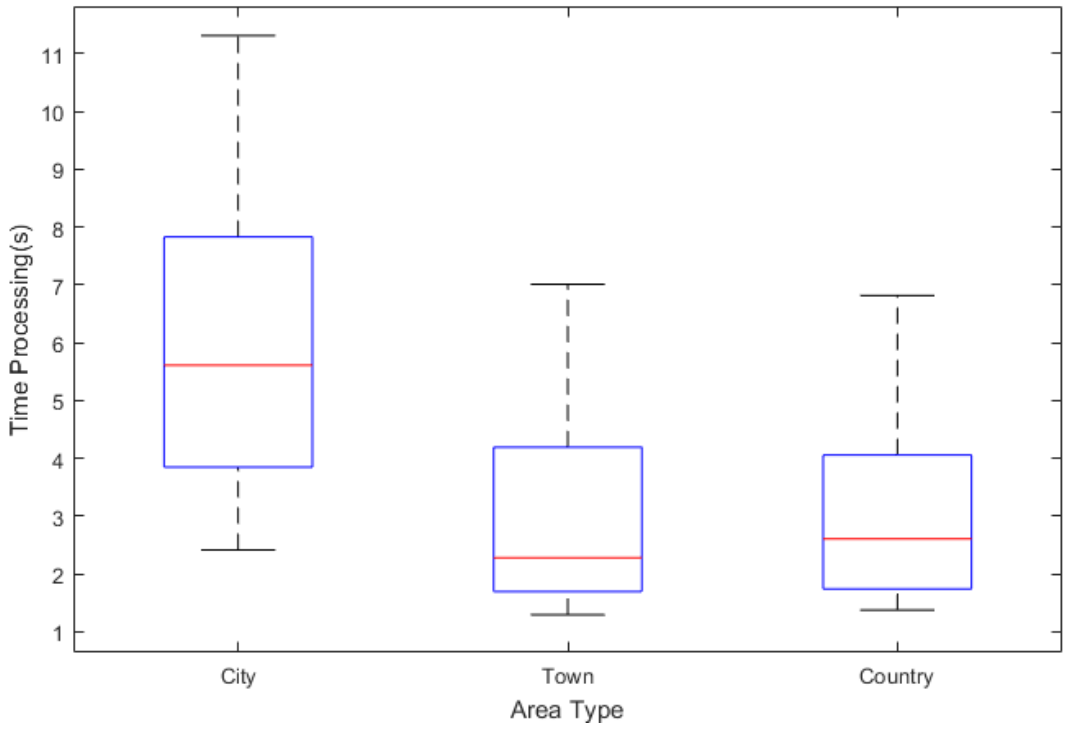


Figure 18: Processing time comparison between different settlement size

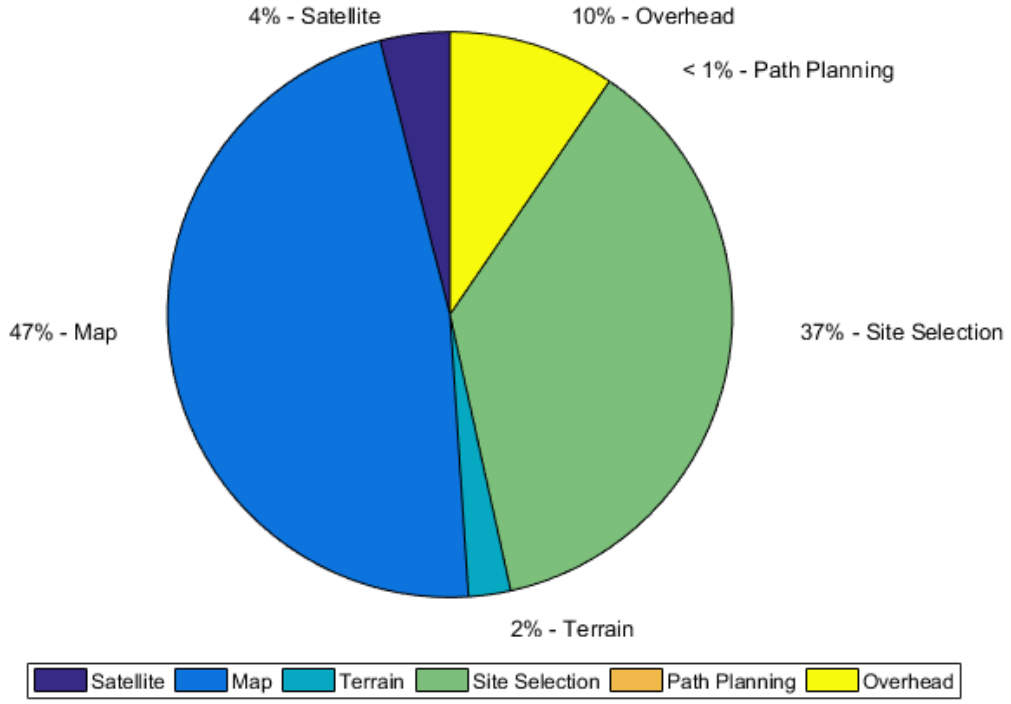


Figure 19: Percentage of total processing time for each module

13.1.3.3 EFLS Performance on a Raspberry Pi 3

The Raspberry Pi 3 has been set as the minimum standard required for EFLS to operate. This is due to extremely small physical size and weight, hence allowing its integration into any fixed-wing UAV or the majority of rotary wing UAVs currently available. The Raspberry Pi 3 however has significantly lower performance compared with any PC, hence it will be expected to be significantly slower than the control group.

The EFLS was installed on the Raspberry PI 3 identically as it was in the control group, allowing for a performance comparison between the control group. It was found that the Raspberry Pi 3 had a mean processing time of 10.9 times greater than the control group with a standard deviation of 3.45 times greater. This is equivalent to the Raspberry Pi 3 mean processing time of 45.5s. The difference in processing time was found to differ between the modules as represented in Figure 20. This indicates that different performance bottlenecks were observed for different segments of the code. The estimated performance difference between the control group and the Raspberry Pi 3 is 1000% slower CPU, 2000% slower RAM, and 500% slower hard drive. From these estimates it can be determined that the bottleneck for the satellite, map, and terrain module is expected to be the RAM, with the CPU limiting the performance of the site selection and path planning module, with the overhead bottlenecked by the hard drive. This information can improve future optimisation to the Raspberry Pi 3 platform as well as alternative hardware selection with increased specifications of CPU and RAM performance.

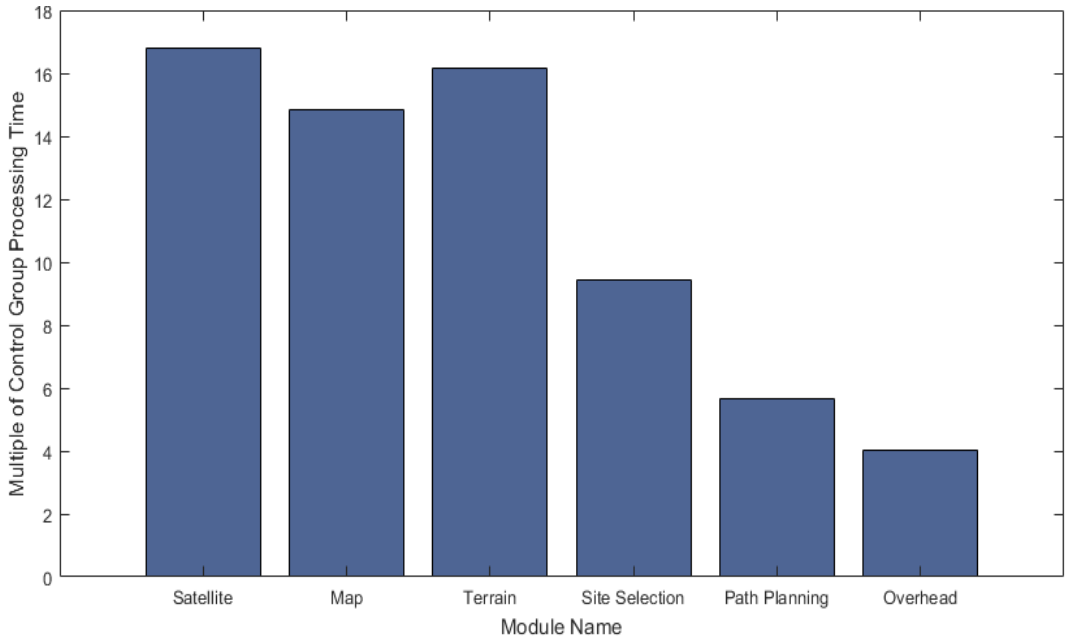


Figure 20: Comparison of processing time of the Raspberry Pi 3 and the control group for each module

13.1.4 EFLS accuracy

The accuracy of EFLS is the fundamental proof that needs to be shown, as if accuracy cannot be guaranteed to be high then there is no purpose of having Phase 1 within the 3-Phase approach. The accuracy had been determined through the use of two different test cases. The first test case being with unrestricted range showing that if no range restriction is applied to the EFLS system then it can find a suitable landing site within a large area. The second test case shows the probability of a safe landing with a restricted range, such as from an engine failure at a set altitude.

13.1.4.1 EFLS Range Unrestricted

The range unrestricted case shows that if the UAV does not have a limited range (range below 25km for the default settings) then its likelihood of finding a suitable landing site can be predicted. This can be alternatively stated as EFLS providing a false positive result for a safe landing site detected, thus it has made a mistake and could potentially injure people.

A null hypothesis test was conducted to ensure the results found are not based upon the random chance of selecting a correct landing site. Three different settlement sizes were used for a comparative test for accuracy, due to different random chance of selecting a correct landing site. The settlement sizes used were city, town, and country areas with a mean chance of finding a landing site correctly, calculated by the total amount of scanned area divided by the amount of safe area using the minimum landing site size specification in EFLS as a minimum requirement for the safe area. For the city, town, and country areas tested it was found to have a mean random safe landing site positive result of 9.00%, 55.4%, and 93.0% respectively. The null hypothesis was defined as the accuracy of finding a valid landing site is not greater than random chance of selecting one within the specified settlement size, with the alternative hypothesis being the contrary case. The simulations conducted to test the hypothesis were

completed using the TurboSim, where 300 different locations per settlement size category were simulated. By reviewing each of the selected landing sites and identifying if the area had the potential for human inhabitants or infrastructure through the satellite images, it was found that only one failure was detected for the town and country areas with no failure for the city areas. With a P-value set to 0.01 and a one-sample single tail Z-test conducted on the found results, it could be determined that the EFLS results had a higher accuracy than random chance in all three settlement size cases. Thus, the null hypothesis was rejected and the alternative hypothesis was accepted. This showed that the mean accuracy of 99.7% with a confidence interval of 0.73% was not based upon the random chance of selecting a correct landing site however it was based on the correct design and execution of EFLS.

Analysing the failed results, one failed in a country area and one failed in a town area. This is shown in Figure 21 – 24. The first set of figures show that EFLS selected a safe landing site that it thought was a forest area, however, when zooming into the picture it is seen that two houses are located within this selected safe area. While it is seen in Figure 22 that the houses are mostly covered over by trees, it does show that EFLS has failed to detect an area that has human inhabitants.

The second set of figures show that EFLS failed to detect a farm house within its selected safe landing area. This was determined to be because the canny edge detection could not make a distinction between the brown ground and the brown roof at the set search resolution. Additionally, there were no street maps in the OSM database of the privately maintained road that connected the house to the main road. This lead to EFLS believing the area was free when in fact it has human inhabitants.

In both of the cases of failure, it is even hard for a human to determine that the area is unsafe to land at the original resolution of the satellite image. This indicates that EFLS is quite good at find safe landing areas, however, in very difficult circumstances it may fail to produce the desired result. This could be improved by increasing the pixelPerMeter of the satellite image but this means increases storage of the larger files. Alternatively, the canny edge detection sensitivity could be increased to notice more subtle edges but this could increase the rate of false negative results and decrease the EFLS site selection time. This trade-off needs to be made for the users specific computer processing power, storage, and required accuracy.

Figure 21: Town area failed case

Figure 22: Zoomed in version of Figure 21

Figure 23: Country area failed case

Figure 24: Zoomed in version of Figure 23

13.1.4.2 EFLS Range Restricted Case

The range restricted case is used to show the relationship between the available flight range of the UAV to the probability of landing safely. This can be alternatively stated as EFLS responsiveness to an emergency event and its false negative rate rejecting potential landing spots within reach and attempting to land at a further location than possible.

The test conducted involved two case studies based on location. Case study 1 was when the UAV was over a densely populated city (Sydney, NSW, Australia) and case study 2 was when the UAV was over a small town (Queanbeyan, NSW, Australia). The simulation conducted utilised the SITL UAV simulator which was used to replicate an engine failure at the same location but different altitudes, and as a fixed-wing UAV has a glide-slope during an unpowered descent the range can be estimated. EFLS was operated on a Raspberry Pi 3 to show the effects of increased computation time on the probability of landing the UAV safely. The simulation was repeated five times per altitude sample to determine the approximate probability of successfully landing the UAV. The UAV was classed as successfully landing if it was able to land within the specified landing site area by EFLS. From the simulations conducted, the cases where the UAV failed to land successfully were due to not finding a suitable landing site in enough time, not enough flight range to get to the location, or crashing into high terrain.

Figure 25 shows the relationship that was observed from the simulated results using SITL. A linear trend was observed when the altitude was less than the critical altitude value, defined by when a 100% success rate was achieved. Above the critical altitude value, a 100% success rate was maintained. The linear trend fitted to this data when below the critical altitude value had an R-squared value of 0.92 and 0.98 respectively for the two cases showing a suitable fit was achieved using the applied linear trend.

The simulation model was then used to test a human UAV pilot's ability to land a UAV in the same scenario, the pilot could successfully land the UAV in case study 1 when the altitude was above 200m and in case study 2 when the altitude was above 100m. This can then be compared to the performance of EFLS in the same scenario, which shows that EFLS has no chance of reaching the site at the same distance and is seen being able to reach the site between 20-40% of the time at the altitudes of 300m and 200m respectively. This offset of 100m between the human and EFLS achieving the site was due to two factors. The first factor is due to EFLS taking a significant amount of time to process on a Raspberry Pi 3 hence accounting for 80m of altitude loss. The second factor is since EFLS may not take the most direct approach to the landing site and will prefer a wider turn to increase stability on landing which approximately accounts for the remaining 20m of altitude loss. To achieve the full 100% success rate for EFLS another 300-400m of altitude is required. This was determined to be due to variations in time processing the EFLS algorithm, the UAV losing too much altitude in the altitude reduction maneuverer and by random variations in the simulation of the UAV.

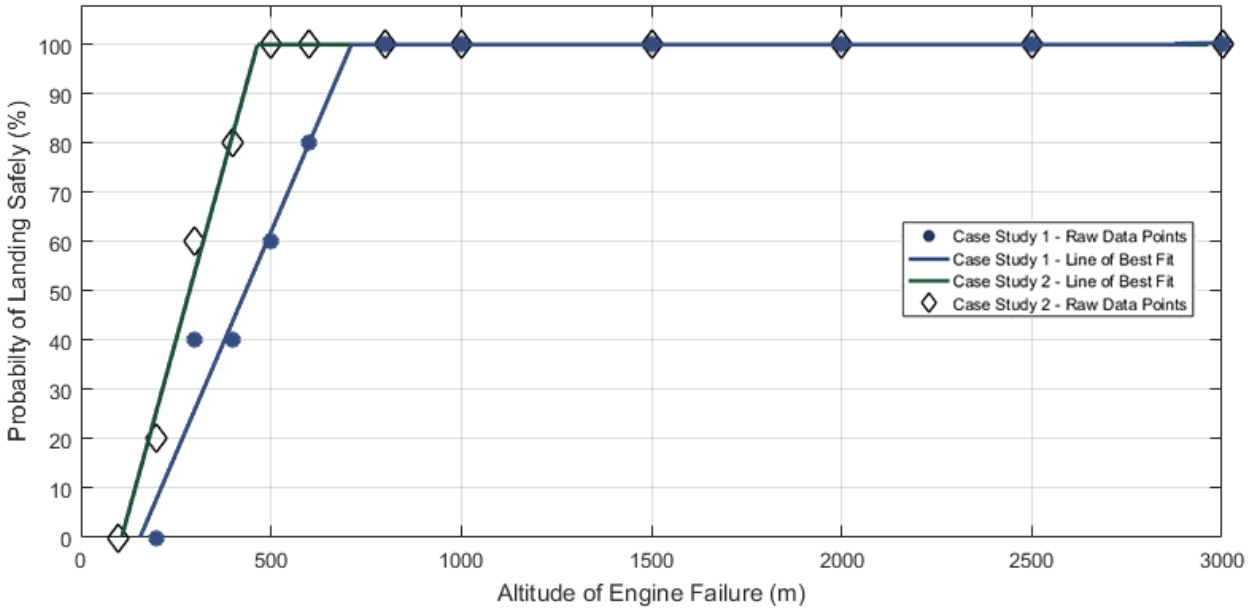


Figure 25: Two case studies showing the reliability of EFLS to land a UAV from an altitude with no thrust

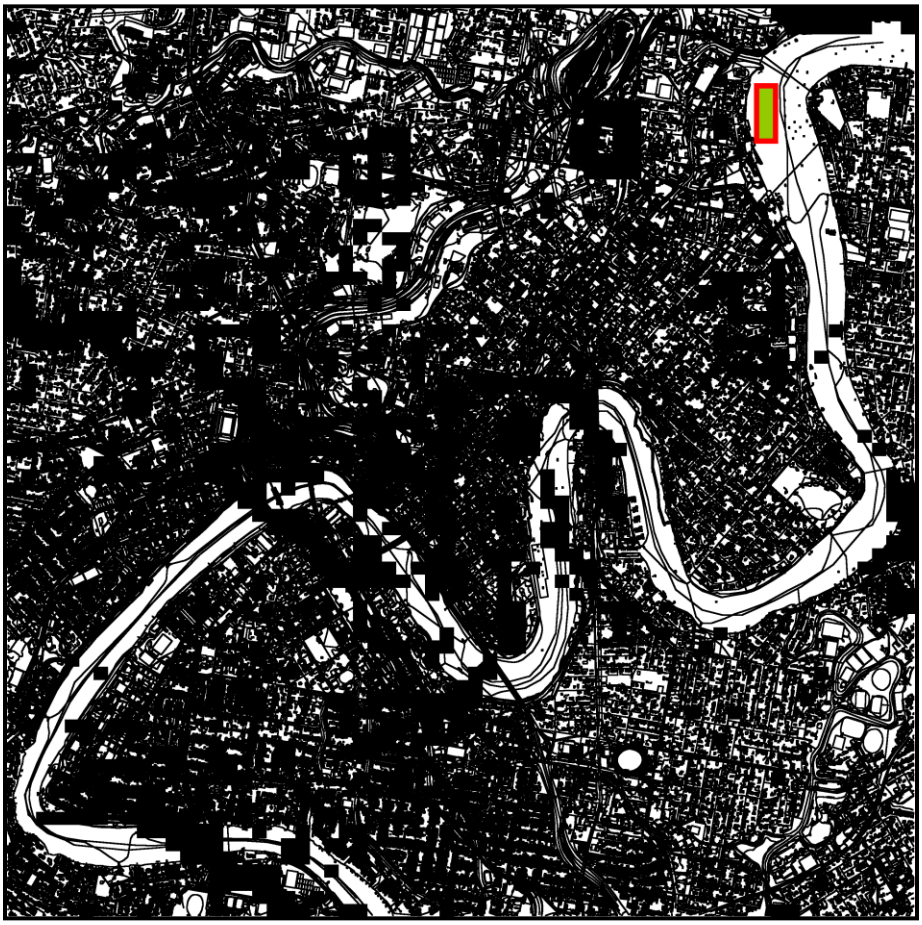


Figure 26: Case study one, example of selected safe landing site in a dense city settlement

13.1.5 EFLS UAV test flight

As EFLS has been integrated into SITL, the step towards a test flight on an actual UAV system is very small as its requirements are identical except for hardware integration into the UAV. The test flight conducted with the UAV was aimed at proving that the simulations conducted both with SITL and TurboSim are representative of the actual performance when used on a real UAV. The test flight was conducted at the CMAC model flying club located in Canberra, Australia. Figure 27 was a photo taken of the UAV during its final approach in an emergency landing. The entire flight was autonomous from take-off, the designated mission flight, and the simulated engine failure.

When the UAV was flying its predefined mission, an engine failure was simulated and EFLS was achieved. Once EFLS was activated it was able to find a suitable landing site which it then planned a desired path towards and sent this path to the UAV flight controller. Due to safety and insurance reasons on the day, a landing was only allowed at the flying club's runway. As EFLS automatically detects nearby buildings and the runway as unsafe areas to land due to it being man-made, it finds an alternative location. To ensure EFLS both found a suitable landing site and the UAV would land at the airfield, the selected site was overridden within the software. This achieves the requirement of landing at the airfield, while still being identical if the actual selected site was landed at. The UAV then proceeded on the flight path as specified by EFLS and shown in Figure 28 from waypoint 9 to 19. The UAV landed safely within 1m of the selected landing location. This same test flight was repeated three times to ensure it would successfully select the same safe landing site and land successfully each time.

The SITL simulation was also conducted for the same location and mission profile as the actual UAV flew during its testing. The simulated model was found to overshoot the landing location by 50m, however, other performance were proven to be identical and both identical landing locations were found between the simulated and actual UAV flights.

Figure 27: UAV on final approach to safe landing site selected by EFLS

Figure 28: Ground control operators view of UAV while on EFLS created flight path to the safe landing site

14 EFLS Feasibility

It was aimed to determine if the concept for Phase 1 was viable, through the testing of a developmental version of the Phase 1 concept. EFLS was developed to test the key performance requirements of reliability, processing time, and accuracy. The reliability of Phase 1 has been determined to be 100% given the 941 simulations conducted. The processing time was found to be sufficiently low enough for the control group testing, however for the Raspberry Pi 3 case optimisation should be made specifically in the map module for decrease processing time. The accuracy was sufficient with a mean of 99.7% and a confidence interval of 0.73%. A restricted range showed the critical altitude for accurate landing using EFLS to be 700m or approximately 3.5km minimum flight range. All of the performance criteria have been achieved except for the Raspberry Pi 3 processing time, thus it can be determined that Phase 1 is viable towards the 3-Phase approach, but optimisation is required for the Raspberry Pi 3. Additionally, due to Phase 1 high accuracy when compared to the other phases, it can be said it will be the integral component of the 3-Phase approach that increases its overall performance to human pilot safety levels in emergency scenarios in future development.

It should be noted that while EFLS Phase 1 has only be completed, that a majority of the foundation work for the entire 3-Phase approach has been created. This includes its coding foundation in C++, a basic dynamic decision engine, the establishment of an EFLS communication protocol for integration into any flight controller, and full integration into ArduPilot.

15 EFLS Development Future

This section will cover future areas of develop that is necessary for EFLS.

15.1 Current bugs

Current bugs are known issues that need to be resolved. While EFLS still works effectively as shown in the successful results section, these bugs are known weak spots of EFLS that need to be fixed. The majority of these issues has arisen due to the short amount of development time available.

1. EFLS satellite image cache should be optimised to store the satellite images more effectively. The current method stores 3000x3000 pixel image with an approximate reuse area of 25%. This means that a majority of the stored cached data is duplicated across another image. To fix this, an industry standard method of image cache for maps should be implemented.

2. Terrain data is not used when EFLS needs to access more than one terrain data file. This is because a simple method of merging two datasets could not be quickly implemented, thus if the error occurs it deactivates the terrain data from the analysis.

3. Street map processing time is currently unoptimised. The street map data is all contained within the country file and each node is checked to be within the selected area every time. A optimised version would be to split the country file into smaller sections for quicker processing.

4. The site selection module is currently unoptimised and takes too long to process the required safe landing site. An improve method is to use “blob” detection to find the largest enclose landing area, then conduct the landing mask search on that. Alternative algorithms should also be investigated and compared in processing time.

5. The wind compensation algorithm is not currently suitable and is a very rough estimate. It should be updated with an improved wind model accounting for turns in wind and increased downwind range.

15.2 Future improvements

The below list details the required improvements to enhance the current EFLS software and make it easier to use or improve its functionality.

1. Preloading cache feature should be added, to allow the simple download of a large amount of cache data. This will reduce the dependency of EFLS on downloading new data while in the air.

2. Continuous cache updater feature show be added, to allow the option for the UAV to continuously update its cache data while in the air when flying over a new area. The current

method only downloads what it needs when required, however, this has associate delays. To ensure a landing site is found quickly, the data should be predownloaded when flying over new area not in the current cache data.

3. Black and white list feature should be added. This will allow the user to predefine locations that are and are not suitable for landing. This is useful if EFLS could identify a safe landing area that is highly undesired to land at, thus EFLS can be made aware of this prior to flight.

4. EFLS live parameter updating. EFLS was designed to support live parameter updating without affecting its internal operations. While it is safe to change the parameters, no method was created to change the parameters without recompiling the code. This should be changed with EFLS reading parameters from a text file within its running dictionary.

5. Terrain data accuracy should be improved. The current terrain data used is the 3 arc-second SRTM terrain data. This should be replaced with the 1 arc-second terrain data where available, which will improve the gradient detection accuracy.

6. EFLS should be made easier to install. This could be done through changing the way C++ operates with its decencies thus allowing for the simple *apt-get* command to install the suitable libraries and dependencies.

7. The EFLS communication protocol should be expanded to allow for more features, such as EFLS setting different waypoint modes and EFLS receiving more detailed data from the UAV.

8. The EFLS software and the EFLS module within MavProxy currently do not check if they are connected. They just assume a connection is present. This does not affect the code, however, it can cause confusion for an operator thinking that they have activated EFLS while they have only opened the EFLS communication module.

9. The EFLS module within MavProxy should have a feature that allows automatic activation of the EFLS software, thus the user does not need to start two different processes.

10. The path planning algorithm currently implemented is sufficient, however, it could be improved through the addition of a dynamic path planning algorithm. This will allow the UAV to avoid specific areas while on the final approach, determine the path in 3D space accurately, and being able to tell the decision engine that it cannot plan a suitable and/or safe path to the landing site.

11. Current information of airfields and airports is provided within the OSM street maps. While there has been no simulated case of EFLS choosing an airfield or airport as a safe landing site, it does have the potential to have aircraft flying nearby. Therefor an exclusion zone should be made around all airfields and airports to ensure the UAV does not hit another aircraft.

15.3 Future expansion

Future expansions are additional items that should be added to the EFLS software, which add whole new functionality.

1. Implement the full 3-Phase approach. Currently only Phase 1 has been implemented as it was the unknown which needed to be proven. Since the results of Phase 1 have proven its viability towards the 3-Phase approach, the continuation in development towards the full 3-Phase approach can now be started. Phase 2 and 3 have been previously researched and reported on, however, the software was not made public available, so it may be necessary to develop all new C++ code implementing these concepts. Lessons can however be learnt from the previous research and implementations as this has been documented.

2. The 3-Phase approach has a weakness towards aircraft collisions as it does not look and avoid other aircraft. While this has purposely been not specified within the 3-Phase approach, it should be noted that the addition of ADS-B avoidance systems and TCAS will be beneficial however active collision detection is the role of the UAV. This data could be sent from the flight controller to EFLS, providing additional information on areas to avoid.

3. The decision engine has already been made to react dynamically to the input of current UAV flight data, however, this could potentially be improved with the use of a neural network, that learns based on simulations, the best way to search for a landing site using the tools available of satellite image module, street map module, terrain data module, and the site selection module. Each of these modules have configurable parameters that can be changed by the decision engine, thus allowing the neural network to learn the best parameters to change.

15.4 Future testing plans

The current testing methods available through the use of SITL and TurboSim offer a wide range of testing opportunities that are very valuable. To improve on this, one type of testing that was conducted manually was the probability of landing the UAV based upon the altitude of engine failure. This requires extensive amounts of manual SITL simulations that very time consuming. This meant that a low number of data points were found to derive a conclusion from. To improve this, a SITL script could be written to simulate the engine failure and then record the distance away from the landing location chosen. The script can then use this distance to determine automatically if the landing has been successfully, which would save a considerable amount of time to collect the data.

16 EFLS Licensing

EFLS is released under the open-source license of GPLv3. This allows for download, use, modification, redistribution of the EFLS software. The main benefit to this license is that all development on the original idea and source code must also be made available as open-source and brought back to the original project. This allows the efforts of others to help in furthering this project into the future.

17 References

Andrew, M. (2004). *Understanding Open Source and Free Software Licensing.* St. Laurent: O'Reilly Media inc.

Andrews, T. (2012). Computation Time Comparison Between Matlab and C++ Using Launch Windows. *California Polytechnic State University*, 1-6.

ArduPilot - License. (2017). *License GPLv3 - Dev Documentation*. Retrieved from ArduPilot: http://ardupilot.org/dev/docs/license-gplv3.html

ArduPilot - MAVProxy. (2017). *MAVProxy 1.6.1 Documentation*. Retrieved from ArduPilot: http://ardupilot.github.io/MAVProxy/html/index.html

ArduPilot. (2017). *Open Source AutoPilot*. Retrieved from ArduPilot: http://ardupilot.org/

Barry, A., & Tedrake, R. (2015). Pushbroom Stereo for High-Speed Navigation in Cluttered Environments. *2015 IEEE International Conference on Robotics and Automation (ICRA)* (pp. 3046-3052). Seattle: IEEE. doi:10.1109/ICRA.2015.7139617

Bry, A., Bachrach, A., & Roy, N. (2012). State Estimation for Aggressive Flight in GPS-Denied Environments Using Onboard Sensing. *2012 IEEE International Conference on Robotics and Automation* (pp. 1-8). Saint Paul: IEEE. doi:10.1109/ICRA.2012.6225295

Crump, M., & Williams, P. (2012). Intelligent Landing System for Landing UAVs at Unsurveyed Airfields. *28th Congress of The International Council of The Aeronautical Sciences* (pp. 1-19). Brisbane: ICAS.

Dade, S. (2016, August 20). *ArduPilot/MAVProxy*. Retrieved from GitHub: https://github.com/ArduPilot/MAVProxy/blob/master/MAVProxy/modules/mavproxy_map/mp_elevation.py

Eagle Tree Systems. (2017). *Vector Flight Controller*. Retrieved from Eagle Tree Sytems: http://www.eagletreesystems.com/index.php?route=product/product&product_id=136

Eng, P., Mejias, L., Walker, R., & Fitzgerald, D. (2007). Simulation of a Fixed-wing UAV Forced Landing with Dynamic Path Planning. *Australasian Conference on Robotics and Automation (ACRA 2007)* (pp. 1-9). Brisbane: QUT.

Faheem, R. M., Aziz, S., Khalid, A., Bashir, M., & Yasin, A. (2016). UAV Emergency Landing Site Selection System using Machine Vision. *Journal of Machine Intelligence, 1*(1), 1-8. doi:10.21174

Farr, T. G., Rosen, P. A., Caro, E., Crippen, R., Duren, R., Hensley, S., . . . Burbank, D. (2007, June). The Shuttle Radar Topography Mission. *Reviews of Geophysics, 45*(2), 1-33. doi:10.1029/2005RG000183

Fetching tiles for Offline Map. (2011, September 26). Retrieved from The Mobile Galaxy: http://go2log.com/2011/09/26/fetching-tiles-for-offline-map/

Gallant, J., Dowling, T., Read, A., Wilson, N., Tickle, P., & Inskeep, C. (2011, October). *1 Second SRTM Derived Products User Guide.* Retrieved from Geoscience Australia: www.ga.gov.au/topographic-mapping/digital-elevation-data.html

GNU. (2007, June 29). *The GNU General Public License v3.0*. Retrieved from GNU Operating System: http://www.gnu.org/licenses/gpl.html

Haklay, M. (2010). How Good is Volunteered Geographical Information? A Comparative Study of OpenStreetMap and Ordnance Survey Datasets. *Environment and Planning B: Planning and Design, 37*, 682-703. doi:10.1068/b35097

Hulens, D., Verbeke, J., & Goedeme, T. (2016). Choosing the Best Embedded Processing Platform for On-Board UAV Image Processing. *VISIGRAPP 2015. 598.* Cham: Springer. doi:10.1007/978-3-319-29971-6_24

Lu, A., Ding, W., & Li, H. (2013). Multi-Information based Safe Area Step Selection Algorithm for UAV'S Emergency Forced Landing. *Journal of Software, 8*(4), 995-1002. doi:10.4304/jsw.8.4.995-1002

Luis Mejias, Daniel Fitzgerald, Pillar Eng and Xi Liu. (2009). *Forced Landing Technologies for Unmanned Aerial, Aerial Vehicles.* (T. M. Lam, Ed.) Queensland, Australia: InTech.

Maron, M. (2015, November 19). *How complete is OpenStreetMap?* Retrieved from mapbox: https://www.mapbox.com/blog/how-complete-is-openstreetmap/
Matuska, S., Hudec, R., & Benco, M. (2012). The Comparison of CPU Time Consumption for Image Processing Algorithm in Matlab and OpenCV. *2012 ELEKTRO*, (pp. 75-78). Rajeck Teplice. doi:10.1109/ELEKTRO.2012.6225575
Microsoft Corporation. (2017, May). *Bing Maps API Terms*. Retrieved from Microsoft: https://www.microsoft.com/maps/product/terms.html
OpenStreetMap - License. (2017). *Copyright*. Retrieved from OpenStreetMap: http://www.openstreetmap.org/copyright/
OpenStreetMap - Statistics. (2017, January 30). *Stats*. Retrieved from Wiki OpenStreetMap: http://wiki.openstreetmap.org/wiki/Stats
ProfiCNC. (2017). *PixHawk2*. Retrieved from ProfiCNC: http://www.proficnc.com/
px4 autopilot - NUC. (2017). *Intel NUC*. Retrieved from Open Source for Drones: https://pixhawk.org/peripherals/onboard_computers/intel_nuc
px4 autopilot. (2017). *Open Source for Drones*. Retrieved from px4: http://px4.io/
Szabolcsi, R. (2016). A New Emergency Landing Concept For Unmanned Aerial Vehicles. *Review of the Air Force Academy*(2), 32-40. doi:10.19062/1842-9238.2016.14.2.1
Topf, J. (2017). *Osmium Library*. Retrieved from osmcode: osmcode.ord/libosmium
Tridge, A. (2017, January 19). *ArduPilot/MAVProxy*. Retrieved from GitHub: https://github.com/ArduPilot/MAVProxy/blob/master/MAVProxy/modules/mavproxy_map/mp_tile.py
Williams, K. W. (2004). A Summary of Unmanned Aircraft Accident/Incident Data : Human Factors Implications. Washington: Federal Aviation Administration.