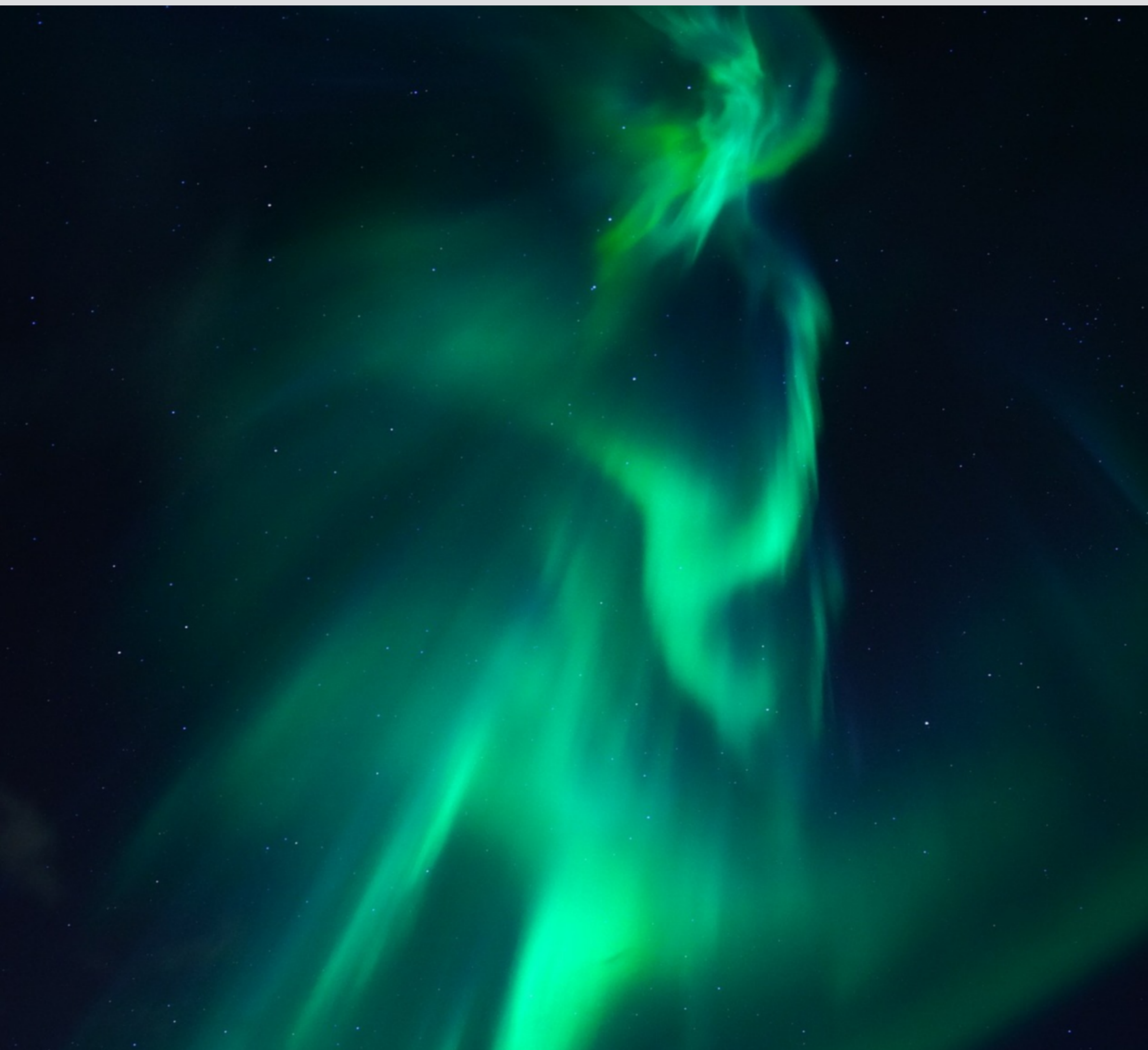
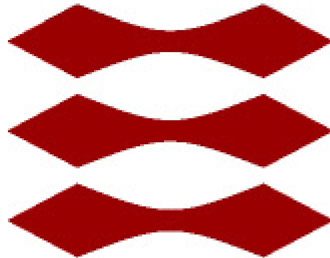
DTU

DTU Space

Development of mapping software for real-time plotting of sensor data values from a UAV survey

Andrés Gómez Álvarez, June 2019





TECHNICAL UNIVERSITY OF DENMARK

BSC PROJECT

Development of mapping software for real-time plotting of sensor data values from a UAV survey

Authors:
Andrés Gómez Álvarez

Student Number:
s182089

Supervisors:
Mick Emil Kolster
Eduardo Lima Simões da Silva
Arne Døssing Andreasen, Ph.D.

17th June 2019

Abstract

Geomagnetic field variations are affected by the composition and morphology of Earth's surface. By isolating these contributions, topology mapping as well as magnetic anomaly detection can be achieved. Commonly, UAV-based surveys are used to collect readings over a large area. Traditional techniques rely on post-processing of the field values to generate high-precision maps. They are often faced with the problem of having little to no feedback during the data gathering. As a result, errors are difficult to detect and no information is available until the lengthy processing is finished. This document presents *LiveSensing*, a real-time mapping software meant to aid during the surveying process. Rapid visualization of the measured data is achieved by implementing simplified processing techniques that allow a dynamic plot of the magnetic field, which can be interacted with through a GUI. Remote control of the gathering process is made possible by a command system. The application is presented as a modular platform designed for simple expandability. Small, light systems are targeted for usability at the survey site, making performance and resource utilization an important concern that lead to a heavily multi-processed implementation. Recreated real-world scenarios show the existing possibilities for real-time techniques. These prove that radio links can be used for long-range communication with a surveying aircraft, while light algorithms are able to yield respectable real-time results. Limitations compared to traditional methods are detected and linked to the simplified approaches. However, they are considered small enough to make real-time processing techniques a potentially invaluable assisting tool.

Contents

1 Introduction **1**
1.1 Geomagnetic sources . . . 1
1.2 Traditional surveying techniques . . . 1
1.3 Real-time implications . . . 3
1.4 Project objectives . . . 4

2 Implementation **4**
2.1 Application core . . . 5
2.1.1 Gathered Data thread . . . 6
2.1.2 Processed Data thread . . . 7
2.1.3 Graph logic . . . 7
2.2 Data Gatherers . . . 8
2.2.1 File Data Gatherer . . . 10
2.2.2 Fifo Data Gatherer . . . 10
2.2.3 Replay Data Gatherer . . . 11
2.3 Data Processing . . . 11
2.3.1 Interpolation methods . . . 13
2.3.2 Inverse Squared Distance interpolator . . . 14

3 Results and capabilities **17**
3.1 File-based analysis . . . 17
3.2 Radio link results . . . 22

4 Using the tool **25**
4.1 Installation and dependencies . . . 26
4.2 User guide . . . 26
4.2.1 Start screen . . . 26
4.2.2 Main screen . . . 28
4.2.3 Managing Gatherers . . . 29

5 Expanding the tool **30**
5.1 Creating a new gatherer . . . 30
5.2 Creating a new interpolator . . . 32
5.3 Future work . . . 33

6 Discussion **34**

7 Conclusion **36**

References **37**

Appendix: Source code **38**

1 Introduction

An aeromagnetic survey is a type of geophysical prospecting. It uses one or more magnetometers carried by an aircraft to measure the magnetic field from the subsurface across a large area. By careful processing of the data, it is possible to extract the contributions to the total field coming from local magnetic sources at or close to the surface. This provides a means of remote terrain modeling, as well as detection of magnetic anomalies (i.e. large concentrations of magnetized material). Some of the general considerations needed for an accurate analysis of Earth's magnetic field are explained below. Traditional surveying methods are also briefly mentioned and compared to a real-time approach. These considerations are later put into practice for the development of the *LiveSensing* real-time mapping tool, detailed in later sections.

1.1 Geomagnetic sources

Earth's magnetic field, also called geomagnetic field, is often approximated as that of a dipole. However, this is only valid for qualitative purposes, and an in-depth analysis leads to a slightly different and much more complex model. It is necessary to account for effects such as emanated charged particles from the sun, which interact with the field especially at high altitudes, or the heterogeneity of the planet's composition. This results in the total field being a superposition of several components, which should be considered when processing magnetic data. These are usually divided into two main groups, as explained in [1]:

- *Internal sources* denote those located below Earth's surface, namely core and crustal fields. The core component amounts to 95% or more of the total geomagnetic field at Earth's surface, and is caused by dynamo action of molten iron currents in the planet's core. With negligible time variation (often measured in secular terms) for the time spans here considered and spatial scales much larger than that of a survey area, these contributions can be considered constant both in time and space. As for the crustal field, its origin is in magnetized rocks and materials in Earth's lithosphere. While time variations are also negligible here, spatial scales are notably smaller, with some components having magnetic footprints of a few meters. These sometimes called super-crustal fields are created by local magnetic anomalies positioned very close to or at the surface, and they reflect the local geological variations that are the target for the considered aeromagnetic surveys.

- *External sources* refer to contributions from electric currents in the ionosphere (90 to 1000Km altitude) and magnetosphere (over 10,000Km altitude). In contrast to internal sources, these are dynamic and change according to geomagnetic conditions. These variations take place over much shorter time periods and must therefore be considered. They also have the added effect of inducing secondary currents in Earth's interior, which in turn generate a secondary, induced magnetic contribution to the external fields that is directly dependent on the primary external components' amplitude and time derivative.

These must all be taken into account during a survey, with the goal being that contributions from local crustal anomalies can be isolated and analyzed. This can be achieved by taking all of the unwanted field sources as a baseline which can be subtracted from the measured data, since their magnetic footprint is generally much larger than the survey area. The complexity lies in that some of them (namely, external sources) are time-dependent and therefore contribute to a time-varying baseline. Advanced models can be used to approximate these variations in order to filter them out.

1.2 Traditional surveying techniques

In order to isolate local surface variations in the magnetic field, it is important to account for the diurnal changes in the external field sources. One common practice is to have a static base station in

the survey area monitoring the total field. As discussed above, most contributions to the geomagnetic field have a spatial footprint many times larger than the general survey area. This means that they generally can be assumed uniform within a small region. As a result, local relative magnetic variations can be isolated by subtracting the base station's readings from the collected data. For optimal results, the station would be positioned as close as possible to the survey area itself (ideally in the middle, but it would affect the collected data), in order to make the uniformity assumption as accurate as possible.

There are other influences that should be accounted for: navigational and measurement errors, as well as parasitic contributions from the aircraft itself can have an impact. The latter is mitigated by positioning the magnetometer as far as possible from the aircraft's main body. As for the first two, data levelling is required to reduce their effect. Aeromagnetic surveys follow a particular flight pattern. The main magnetic data comes from parallel *lines* that cover the whole survey area. Perpendicular to those, *ties* are flown periodically for control. This fixed pattern, shown in Figure 1.1, is the base for levelling techniques during the post-processing: having crossover points at certain spots in the survey area provides multiple samples for one geographical location, which can be used to estimate measurement errors. These estimations can be used to develop directional filters to be applied to each individual flight line or tie, subtracting the error contribution (see [2] for more details). Line and tie separation are directly related to the desired and attainable resolution of the processed result.

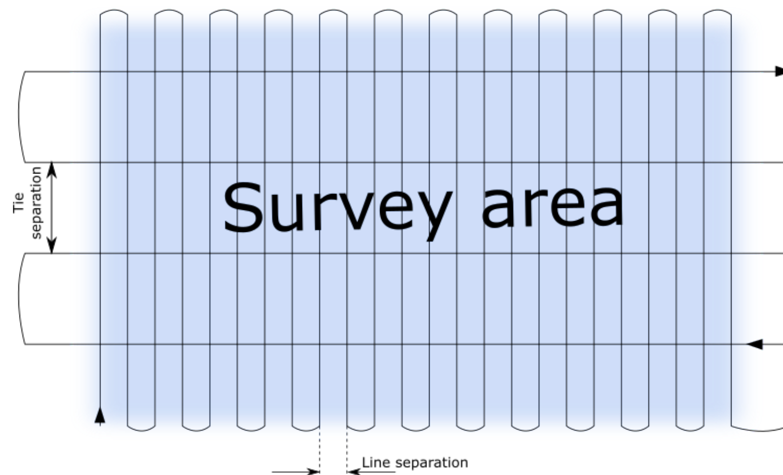


Figure 1.1: Survey flight pattern.

An alternative approach that aims for higher resolution and reduced external contributions relies on the use of multiple magnetometers. [3] describes how both the horizontal and/or vertical magnetic gradients can be calculated by carrying two or three sensors on the aircraft. This means that time-changing contributions as well as those with a large spatial footprint have the same effect on all sensors. As a result, it eliminates the need for a base station and provides a more accurate solution to dealing with transient effects. It also enhances small-scale anomalies located close or at the surface, which present a much smaller spatial wavelength and therefore a more noticeable shift in the measured gradient. The cost of this comes in the form of needing more sensors in the aircraft, which all have to be as isolated as possible from from parasitic fields. This method may also suffer in rugged topography, where frequent gradient changes are expected due to aircraft elevation shifts.

In general, traditional post-processed approaches yield accurate and satisfactory results, since complex levelling and filtering are available. They present however one main disadvantage: the time elapsed between a survey and the results themselves can often be of several days. While this is acceptable for some applications, there are cases where shorter times are required, even if some accuracy is lost in exchange. It is also common that, while the high-quality post-processed product is

still valuable, having an immediate preview offering an estimate result can be very beneficial. From a practical standpoint, it can also be argued that having the possibility of a real-time visualization of the logged data can provide important feedback to a trained eye. This could result in human or hardware errors being identified and solved on the spot, possibly preventing the need to re-run certain measurements, as opposed to if they were detected hours or even days later.

1.3 Real-time implications

Looking at the typical surveying procedure, it becomes apparent that some of the processing techniques cannot be applied on a real time-basis. This applies particularly to those that depend on the flight pattern, since ties are generally not available until the very end of a survey. As a result, traditional error removal is not available. A similar case applies to directional filters: high-pass filtering can prove to be difficult along a line that is dynamically growing. Furthermore, many modern aeromagnetic surveys are carried out by battery-powered UAVs. This means that several return flights are expected while scanning a large area in order to exchange depleted batteries. While the samples associated to said flights can be manually discarded to preserve the aforementioned grid pattern (Figure 1.1), this is not possible for real-time processing.

Given these limitations, a simple approach is to not rely on flight patterns and behaviors, since they may not be reliable or available. [4] introduces the concept of a moving-window filter as a method to accomplish it. While it is described in terms of a post-processing technique, where the full data set is available and used for filtering, the idea can be adapted for a real-time application. It is represented in Figure 1.2, where for every new data sample all the processed points within a certain distance of it are updated to reflect the new information. Points further away remain unaltered, since the field at those can be assumed independent of distant-enough behaviors. This requires a careful choice of the window radius, as it may introduce inaccuracies if too small. A compromise must be made between result fidelity and computation power, since it is unfeasible to re-process the whole area every time on a high-sample rate, real-time application. Having a fixed amount of data to process for every new sample is the key for generating results dynamically. In comparison, traditional methods take in the entirety of the existing data at once, making it computationally unfeasible to do at a high rate.

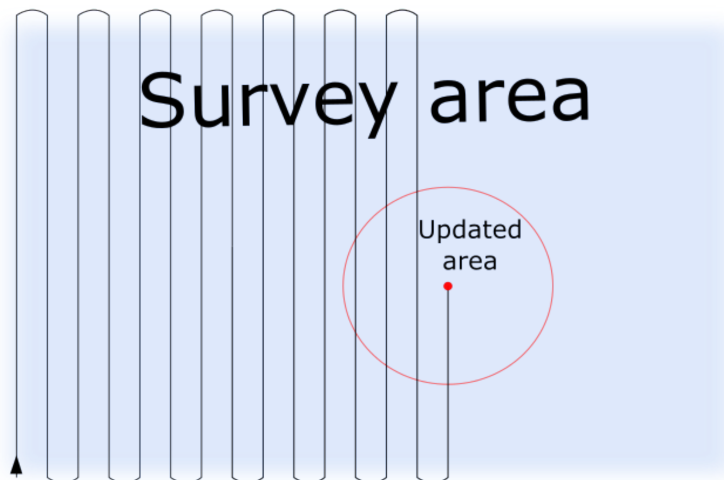


Figure 1.2: Moving window concept.

All in all, real-time approaches are limited in terms of available tools and computation power. However, simplified processes can be used to provide a lower-quality but immediate view of what would come out of more complex post-processing techniques. This can be immensely valuable in some scenarios, even despite the accuracy difference. Therefore, they are meant to be seen as a complementary tool to existing post-processing techniques, rather than as an alternative.

1.4 Project objectives

With the previous considerations, this projects takes on the real-time geomagnetic data processing problem. The goal is to build a tool that provides dynamic feedback of the gathered information during a survey. As such, it is intended to provide a preview of the measured field, which in turn helps detect any abnormal behavior in the data collection process. Most frequently, issues like these are not visible until post-processing begins and the logged data is manually inspected. This means that the possibility of real time detection and prevention of such problems can result in large time savings. For added utility, other useful control data other than the magnetic readings are also considered. With that in mind, the initial project objectives are as follows:

- Android-compatible Python 3 software package for real-time processing of magnetic data.
- Adaptability to a variable sampling rate communication.
- Ability to display corrected field and gradient information.
- GUI-based interactivity for switching between different screens/plots.
- Plan and carry out a test in a real-world environment.

2 Implementation

The real-time processing software *LiveSensing* provides an immediate preview of the magnetic readings over a survey area. As such, it is meant as a field tool for initial result estimations as well as on-the-spot error detection and prevention. Given the on-going nature of the associated UAV surveying project that it was built for, the two core ideas behind the chosen design are upgradability and modularity. For this reason, it is presented as a platform with several basic options which can easily be expanded to fit future necessities or adapt to changes in its requirements. To facilitate this and provide a simple cross-platform solution, *Python 3.7* was the language of choice (official documentation available at [5]). Following this, and heavily motivated by the initial requirement of android compatibility, Kivy was the selected GUI framework. This provides an interface solution fully built with both touchscreen and keyboard/mouse input systems in mind, as well as compatibility with all major platforms (for more information, see [6]). While the interest on android later switched to Raspbian OS, which stayed as the main target system, the tool was built so that little to no modifications would be needed in case android was needed again down the line. As for all other main operating systems (i.e. Windows 10, MacOS and most Linux distributions), compatibility is expected out of the box.

LiveSensing was built following a heavily multi-process, object-oriented approach. The general program architecture, depicted in Figure 2.1, consists on a main process and two sub-processes that take care of a self-contained task each. The goal is to allow each of the three main tasks to make use of a different CPU core, achieving high resource utilization. These tasks are:

- **Data Gathering**: Translating new incoming samples to the expected core-program format.
- **Application core**: GUI handling, data plotting and core logic.
- **Data Processing**: Processing of new samples and generation of interpolated grid.

The main problem associated with this division lies on Python's Global Interpreter Lock (GIL), which serializes access to a single Python interpreter for different threads. It essentially means that multi-core operation is not achievable with standard threads (see [7] for more details). This is overcome using the `multiprocessing` library, which spawns new full OS-level processes each with their own interpreter. While it offers the versatility of running a fully independent program in each of them, it also introduces

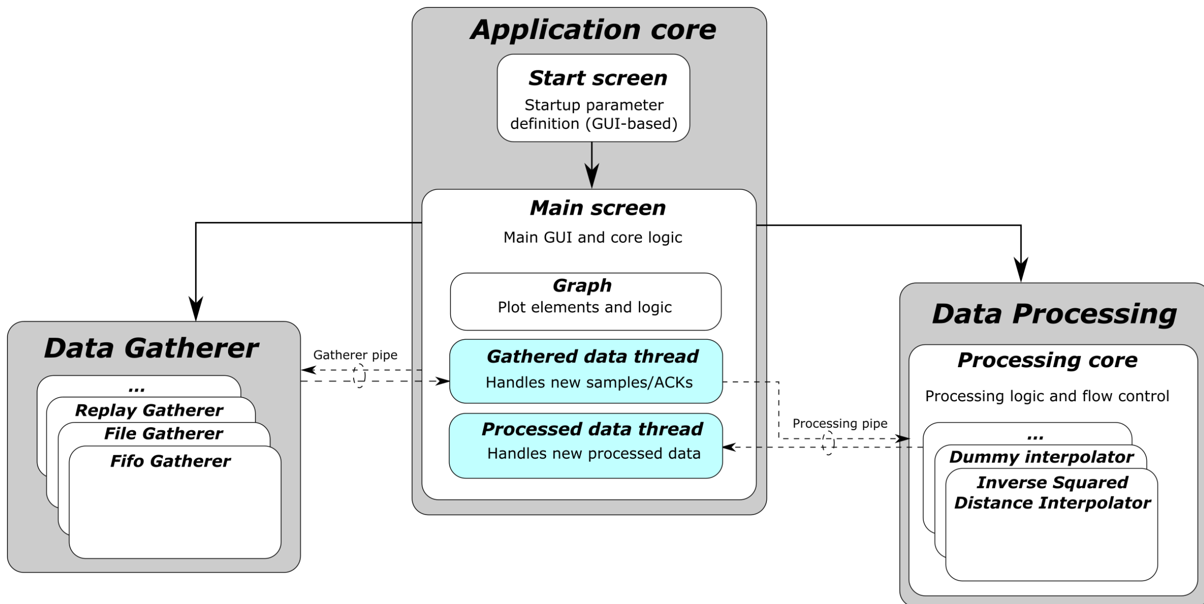


Figure 2.1: General architecture: Three sub-processes (grey) with internal threads (blue) for I/O.

overheads as well as limits communication between them to pipes since no variable sharing is possible. Despite that, the resource availability that it provides was considered enough to justify its limitations, as it would allow for much heavier processing implementations in the future without having a big impact on total program performance (since it would all be running on a separate core that still has ample headroom). Another advantage is that from an upgradability perspective, having one single communication pipe with a defined API makes adding modules to the two sub-processes a relatively easy task, since little knowledge of the application core code is needed. Such API consists on a fixed message or *packet* format between processes, based on a python dictionary containing several fields. This provides code readability and maintainability (since fields can be named appropriately) as well as flexibility to add, drop or modify them before forwarding the packet to another process. Following subsections give a more detailed view of the inner workings of each of these processes.

2.1 Application core

The application core refers to the main process in the tool. It controls all GUI-related features, and handles the interfaces with both sub-processes. It is also in charge of all standard plotting, since the graph must be embedded into the interface which forces it to be generated with other GUI elements. There are two stages, depicted in Figure 2.1:

1. An initial settings screen is shown where the user can select the survey and graphic configurations. These are passed on to the main program screen, which initializes the sub-processes and interface accordingly.

2. A main screen, containing all the user tools necessary during a survey. A main thread is used for handling GUI elements and responsiveness. Inter-process flow control is done through secondary internal threads that handle pipe I/O. Graphing logic and drawing is also executed here.

Note that this section focuses on the application logic, and GUI-related programming details are skipped. For this reason, aspects related to the start screen functionalities are not addressed here. If these are needed, Section 5 provides an overview as well as a guide for expanding it. For a more general guide on the available interface actions, see Section 4.

2.1.1 Gathered Data thread

This thread acts as a data flow manager between the Gatherer and the Processing sides. While communication between the two could be done through a direct pipe connecting them, having an intermediate node in the main process allows the implementation of control fields or even whole packets that can be intercepted in the main process. While these could also be routed through the processing side and forwarded back to the main process, having traffic control information within the processing module goes against the idea of self-containment and adds unnecessary complexity to it.

The behavior of this thread is depicted in Figure 2.2. One of two packet types is expected to come from the gatherer, and all others are ignored. The most common one contains information from a new magnetic sample, as well as a battery voltage reading from the magnetometer. Upon receiving these, internal sample counter and battery voltage variables are updated. After stripping the type and battery fields off the packet for reduced size, it is forwarded to the processing module. The second packet type corresponds to an Acknowledge-like message coming from the Gatherer, which represents that a command sent by the user was successfully received and processed. These cause a warning to the user to be generated with a generic "Command received" message. Note that Kivy forces both the on-screen control information and the warning to be drawn on the main thread, and not on this one. For this reason, the former is scheduled to be redrawn on a fixed 1 second period through Kivy's graphical framework, while for the latter a warning is scheduled to be drawn as soon as possible when the packet is received.

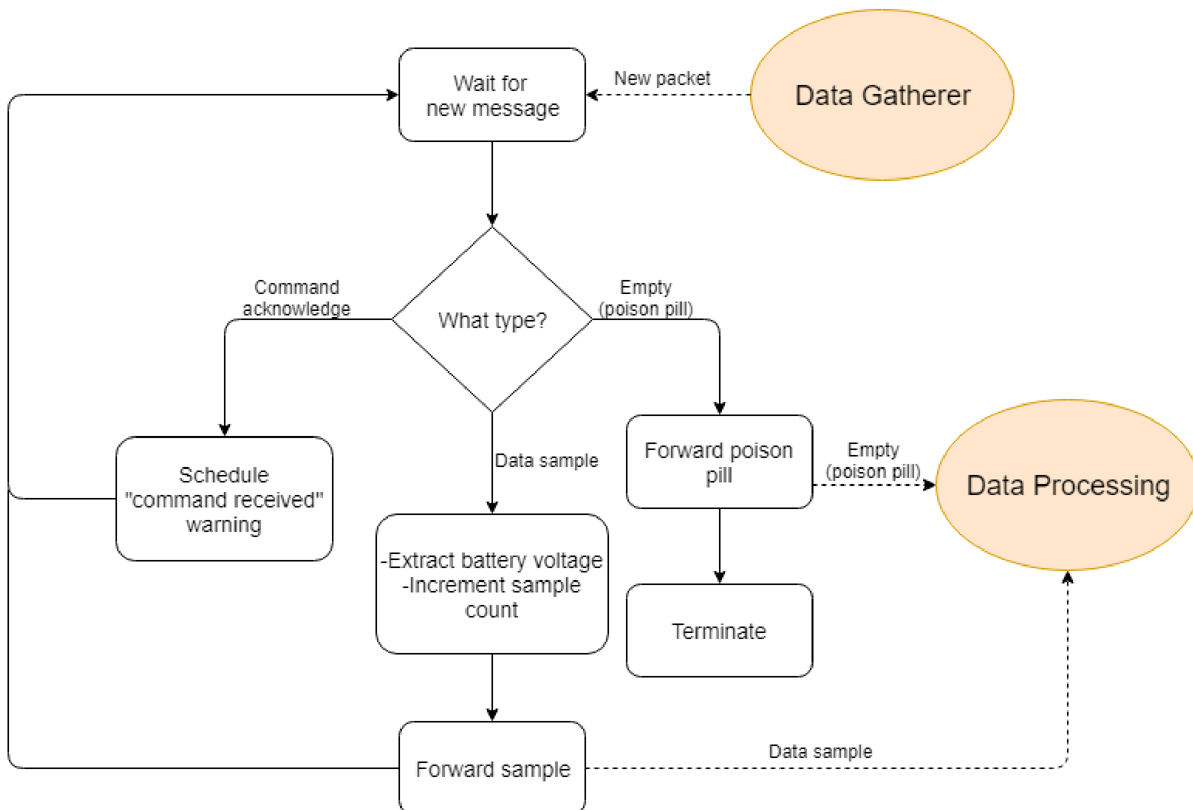


Figure 2.2: Gathered Data thread (flowchart).

Finally, the thread incorporates a termination mechanism based on poison pills[1]: when either an empty packet or *None* is received, the thread will forward it to the processing module and terminate itself. This technique is meant for resource liberation once the gatherer determines that the data flow has ended. It is meant to follow the *Acknowledge* packet for a "stop" command, as a confirmation

[1] "The poison pill is a way to tell the thread to stop what it is doing and die. [...] This needs to be the last item used." [8]

that the gatherer will no longer be operating, but it ultimately depends on the specific gatherer implementation how and when it is sent. Thanks to it, program fluidity slightly increases once the survey is finalized, so interacting with the final plot is easier.

2.1.2 Processed Data thread

This thread is meant as a listener for the processing results. In this case however, no packet forwarding is necessary since it gets the final processed data. Instead, the thread is in charge of calling the corresponding method from the `Graph` API corresponding to either new interpolated data (in matrix form), or updated vectors with the accumulated coordinate and field values of all samples. This procedure is shown in Figure 2.3.

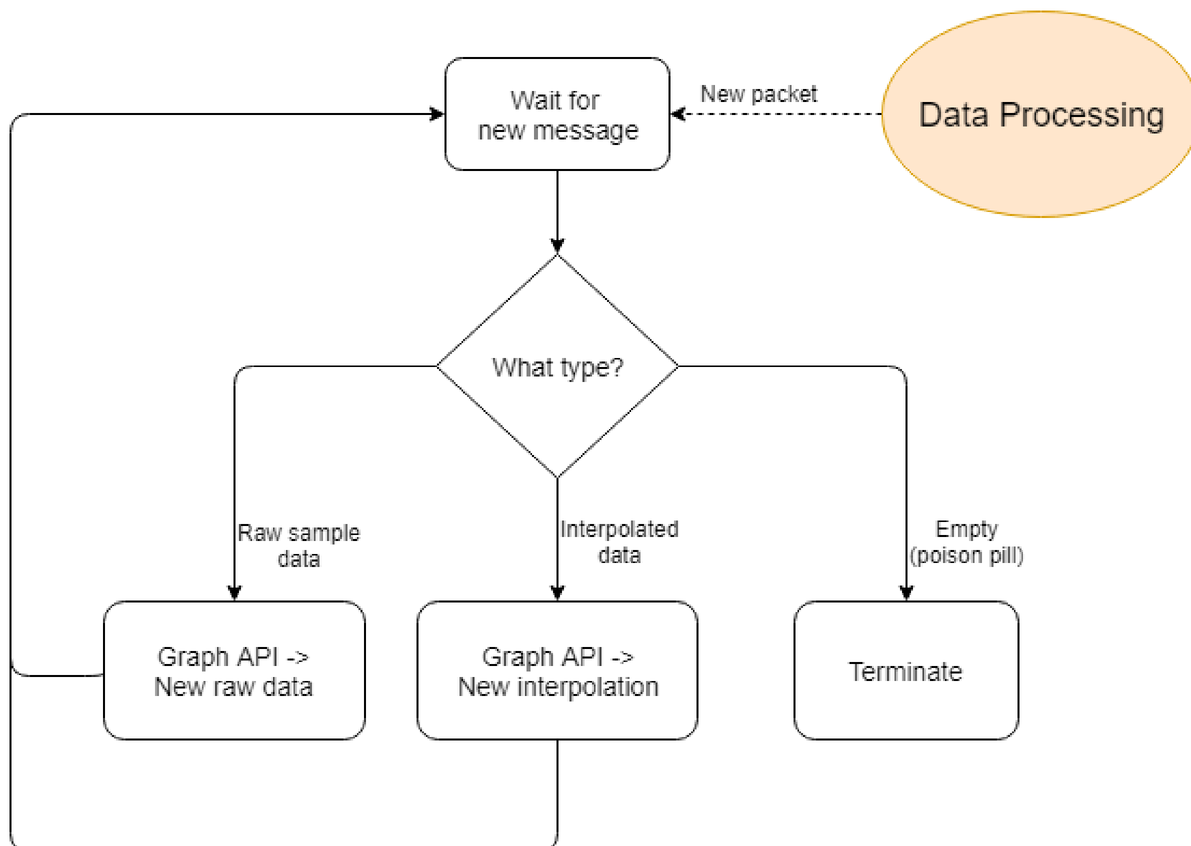


Figure 2.3: Processed Data thread (flowchart).

Like for the previous case, termination is considered. This means that when a poison pill is sent from the gatherer, it will make its way through the gathered data thread, processing module and back into the processed data thread, terminating all of them while doing so.

2.1.3 Graph logic

The entirety of the plotting logic is contained within the `Graph` class in the application core. It contains three main flags that dictate its behavior. A simple API is provided to access and modify these from outside the class, allowing dynamic user access to it during a survey. These flags control three main functionalities:

- **Flight trace**: The path along which samples are being collected is by default shown on the graph. This gives an idea of the accuracy in each survey section, expecting a better result where samples

are closer together. This can be toggled off and back on at any time, e.g. in case it covers some important survey area.

- **Plot updating**: While by default new results are plotted as soon as they are available, this can be turned off. As a result, data from a specific time can be carefully inspected. Note that new results are still being collected and stored, so as soon as updating is toggled back on the latest data will be shown in the graph.
- **Linear interpolation**: Normally the processed and interpolated matrix is desired. However, monitoring of the raw samples is also possible and can be toggled at any time. This is done by performing a linear interpolation on the raw data values received from the Processed Data thread. This requires no regular matrix data to be generated, so the same coordinate vectors used for plotting the flight path can be used. In order to provide this functionality, the raw field values associated to each coordinate pair must also be sent. This adds a slight overhead, since more data needs to be transferred between processes, but its contribution is negligible to the overall application performance.

The ability to independently modify each of these flags means that two copies for each processed result must be kept, one containing the most recent update while the other having the currently plotted data. When plot updating is activated, the plotted data will match the most recent update as soon as possible. However, when updating is deactivated a separate copy is kept of the state of all results at the time of deactivation, so features like hiding the flight path and re-plotting it again are available without using the updated data.

One difficulty lies in the need to receive updates on a secondary thread while being limited to plotting on the main one. This is handled through the use of Kivy's *triggers*. These allow scheduling of a certain function or method to be executed in the main thread in the next iteration of Kivy's internal clock. As such, whenever new data is received, thread-safe methods are available in the Graph API to store the new data. Afterwards, the corresponding trigger is activated so that this data is handled in the main thread as appropriate, or left in the temporary storage to be overwritten by the next result in case plot updating is deactivated. Another important feature is that multiple activations of a trigger before the scheduling is due do not stack, and only a single call to the binded method is generated. As a result, if the hardware is not powerful enough to have the redrawing rate keep up with the processing rate, the main thread will only produce as many plot updates as it can handle. This however is an undesirable situation, since the sending of the whole interpolated matrix and raw data is in itself time consuming and would be a waste if it was done more often than it should. For this reason, it is best to set the plot update interval to match the hardware's capabilities when initializing the application.

While these options provide a certain versatility as to what is drawn, one big limitation is being unable to interact with the plot itself. This is caused by the lack of support from the plotting library, `matplotlib`, for Kivy as a backend (see [9] for the official matplotlib documentation). In order to overcome this, a community add-on was used to provide the necessary infrastructure between the two libraries [10]. However, this only achieves a static image-like plot. In order to have standard functionality (e.g. zoom, dragging, etc.), an option is available to re-generate the current plot in a standalone window with the standard matplotlib backend. Note that this new plot is completely isolated from the main application, so the flag-based features mentioned above only dictate how such new plot is created, but do not affect it once already drawn.

2.2 Data Gatherers

The data gatherer is the part of the program that translates the incoming data from whichever format to the dictionary-based packets used throughout the application. As such, it can be seen as a bridge with the application body that is built for one specific data stream. The inclusion of this intermediary

block allows the *LiveSensing* tool to easily be adapted to new hardware or radio link implementations. The structure and role of the gatherer was kept very simple so as to make the implementation of a new one as easy as possible (specific requirements described in Section 5.1).

The one common thing for all gatherers is a single full-duplex pipe connecting to the application core. It implements a simple and non-restrictive communication protocol, where the user can send certain commands from the main interface to control the behavior of the data stream. The defined messages from and towards the gatherer are listed in Tables 2.1 and 2.2 respectively. While the packets have a strict format and the way they are processed is fixed in the application core, the same is not true the other way around: commands to the gatherer do not cause any effect in the application core, and they are simply sent whenever the user wants. As such, they have no imposition associated to them, and the listed expected behaviors are simply a suggestion. In fact, received acknowledge messages are not internally linked to any previously sent command, and can be sent freely from the gatherer or even not sent at all. This was done so that when doing new implementations, no extra time needs to be spent in features that may be unneeded for a certain use case. At the same time, any reaction from the gatherer's side can be implemented for each of the three listed commands, in case different responsive features to the currently offered ones are desired.

Gatherer → Application core	
Packet type	**Meaning**
Sample packet	Contains all the fields expected from a new sample.
Acknowledge packet	Generates "Command received" feedback message, meant to indicate a command was properly processed.
None / Empty packet	Gatherer is terminating. Causes Gathered Data thread to terminate in the application core.

Table 2.1: Data Gatherer packet types.

Application core → Gatherer	
Command	**Expected behavior**
`"log"`	Initialize the link, start the data stream.
`"pause"`	Temporarily stop the data stream, discard samples while paused.
`"stop"`	End the transmission and terminate the gatherer process.

Table 2.2: Data Gatherer commands.

Finally, the last task of a data gatherer is to log all incoming data in a standard format (see Section 4), so it can later be analyzed. This is not intended to replace any logging done in the aircraft itself, since the high sampling rate of modern magnetometers is likely to be sub-sampled before transmission for bandwidth and real-time computation limitations, on top of reliability of a radio link not being guaranteed. Nonetheless, being able to review the received data can prove useful in certain scenarios, e.g. to re-create the real-time survey using a file-based gatherer. In any case, it ultimately depends on the gatherer implementation whether such a log file is created, since for some applications it may not be needed.

The internal gatherer structure is not fixed in any way. The only restriction is the need to listen for commands from and write new packets to the gatherer pipe. This concurrent I/O suggests the model shown in Figure 2.4. Two parallel threads take the roles of command and data handlers. Notice that both the data logging and command acknowledgements are tagged as optional, since there is no imposition to implement those features. The ACK can also be sent from the data thread, since it may

be simpler in some cases. The poison pill and gatherer termination mechanisms were omitted, as they depend on each individual case. The shown architecture is used for the three available gatherers.

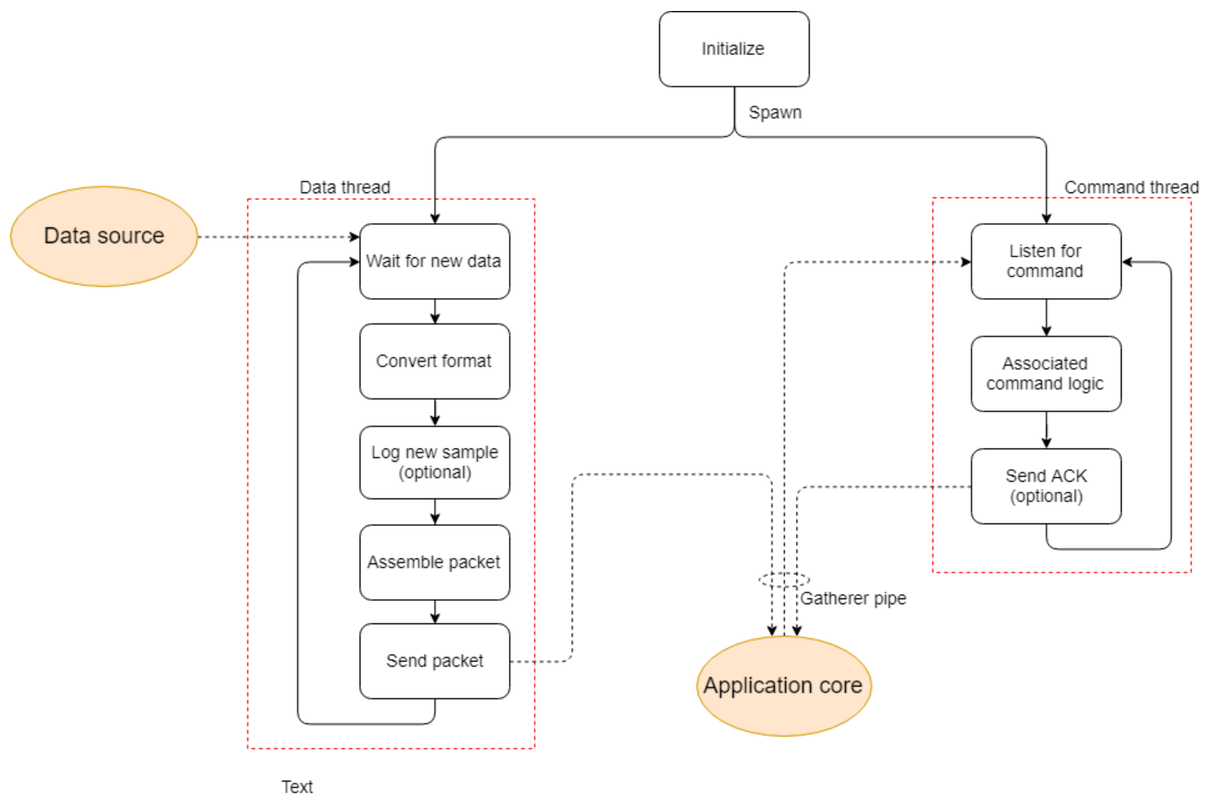


Figure 2.4: General gatherer architecture (flowchart).

2.2.1 File Data Gatherer

It is the main gatherer used during the development of the tool. Reads data from a text file, which contains sample information. It implements all the described features and commands and is meant as a reference for future gatherer additions. It can be configured to accept any number of magnetometers, making it flexible for all survey types. It is built such that the log file format matches exactly the expected input file, once stripped of its headers. This allows feeding such log into the gatherer to later replay an already finished survey, with the possibility of using a different processing configuration.

2.2.2 Fifo Data Gatherer

Represents an implementation of a real radio link. It communicates with the local transceiver of said link, the *MagPro* application included in the source code. To do so, two file-like FIFO (First In First Out) pipes are used, so that each program can write to one and read from the other. Since alternative, more robust communication schemes are possible, this is solely meant as a proof of concept to allow actual field testing. In this implementation, commands are sent through one FIFO pipe and transmitted to the aircraft end. This provides features such as stopping or re-initializing the data logging at the aircraft, which would normally have to be done by hand after flying it back. Acknowledge packets are generated only when an acknowledgement is received from the aircraft, so in this case they come from the data source and therefore are sent from the data thread. Due to the encoding of the received information, a large amount of parsing and formatting is necessary. This is an example of the benefits of using an intermediary data gatherer instead of feeding samples directly to the application core: all of these data conversions can be externalized from the main code, making data formats a non-issue past the gatherer.

2.2.3 Replay Data Gatherer

This gatherer is a modified version of the File Data Gatherer. It also reads data samples from a text file, following the same structure. It is meant for replaying surveys out of the log files generated at the aircraft side when using the *MagPro* tool and the Fifo Data Gatherer. Since these greatly differ in format from the much simpler locally generated files, more complex parsing is necessary. It is meant as an auxiliary tool in case the local logs are lost or incomplete, e.g. if the connection with the aircraft broke at some point. No local logging is implemented in this case, since it is meant for reading log files to begin with, and a simpler logging example is provided by the File Data Gatherer.

2.3 Data Processing

The Data Processing sub-process is responsible for doing all the required calculations and generating a regular matrix of processed data out of the incoming samples. It also efficiently stores valid raw data samples to allow flight trace plotting and linear interpolation. This module is expected to be able to handle heavy computation, which is the reason why a dedicated process was assigned to it. As it encompasses all processing-related tasks, its behavior can be greatly customized through the initial settings. The result is that different configurations capabilities can be utilized, accounting for survey needs or hardware limitations. Three main processing aspects can be modified:

- **Measurement mode**: It refers to the way the magnetic field is obtained. In the current implementation, vertical gradient as well as single-sensor surveys are considered. Parameters describing the used configuration can also be defined, i.e. the separation between both magnetometers in the vertical gradient case. Note that single-sensor mode can be selected even when data from more sensors is being received, in which case only the first one is considered.

- **Interpolation mode**: Some interpolation techniques may be more accurate than others for certain topologies or applications. The computation demands associated also depend greatly on the chosen method and configuration. Again, each interpolation tool has its own set of variables which can be customized.

- **Plot step**: One noticeable overhead is generated by the frequent transmissions between processes. This is especially important about the interpolated matrix, which can contain up to hundreds of thousands of elements in large survey areas. This parameter adjusts how many new input samples are expected before sending an updated output, allowing the user to manually adjust for high sample-rate surveys where not many graph updates are necessary every second.

This module runs on a main method that handles all incoming samples. Before each one is processed, it must be check for integrity. A sample's integrity lock is an output from the magnetometer that determines whether it is properly detecting Earth's magnetic field. Sudden changes like oscillations during the flight may cause the sensor to momentarily lose track of such field, and samples generated during that time must be discarded to avoid corrupting the result. For added security in this regard, a sample is only processed if the integrity lock is as expected not only for itself, but also for the preceding and following samples (when they exist, i.e. not for the first and last ones). This is done by buffering each sample until the next one is received. After this, the value to be interpolated is extracted from the sample. While for a single-sensor survey the reading itself is used, when using two sensors on a vertical configuration the gradient must be calculated as described in eq. 2.1:

$$\nabla_{\text{vert}} = \frac{B_{\text{up}} - B_{\text{down}}}{d} \tag{2.1}$$

where d is the sensor separation. This value and its associated position are inserted into the raw data arrays and also passed to the interpolator. Finally, the new raw data arrays and interpolated matrix are sent back to the application core if appropriate based on the chosen plot step. This whole procedure

is depicted in Fig. 2.5, where *process sample* stands for extraction of the value to interpolate from, insertion into the raw data array, feeding to the interpolator, and sending the result if appropriate. Processing of the first and last sample has been omitted from the chart since it must be done outside the main loop. Notice the inclusion of the poison pill mechanism: the whole process terminates once the last sample is received, and the pill is sent back to the Processed Data thread. It is worth noting that no special filtering is applied, since the data quality obtained from the vertical gradient approach is considered sufficient (see Section 3 for a result analysis). As for single-sensor operation, diurnal effects are not accounted for since no base station hardware was available, but ideally temporal field changes should be subtracted for a more accurate result.

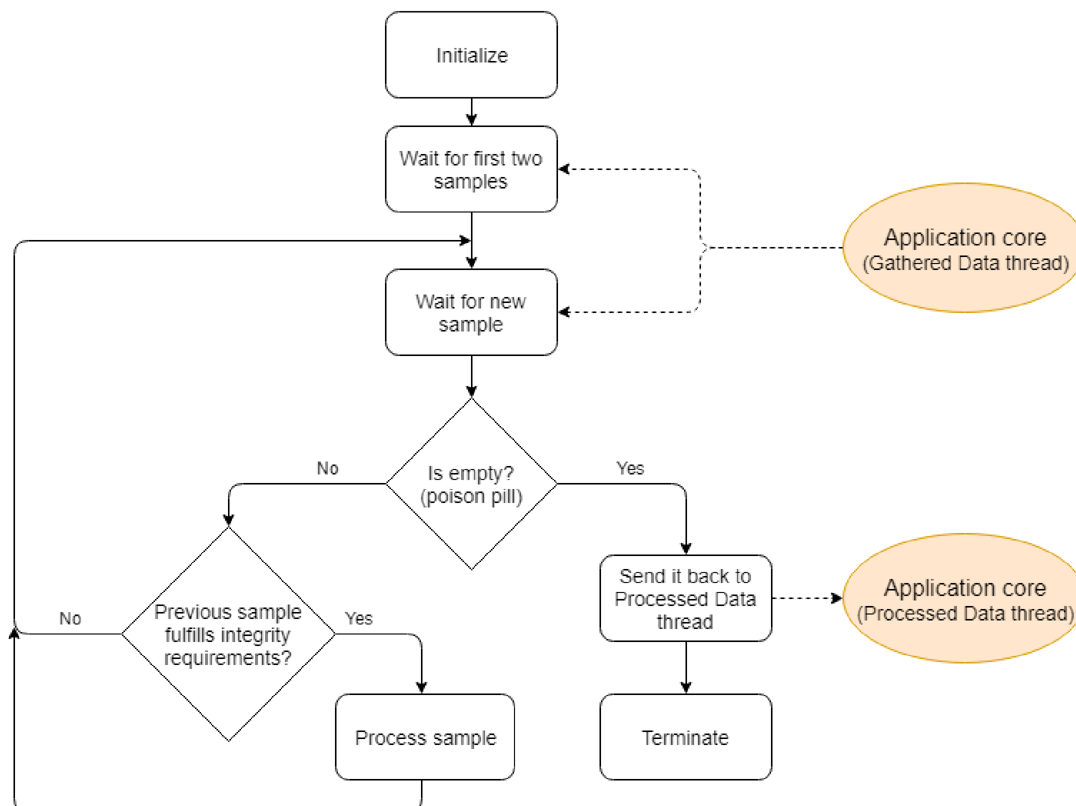


Figure 2.5: Processing general flowchart.

The most important part of the Data Processing sub-process are the interpolator modules. They are designed with a very simple API to simplify the task of implementing new ones in the future: they must be able to receive individual samples, and have an internal method that sends the result to the application core when the main Data Processing logic needs to (see Section 5.2 for more details). Currently one custom interpolation method is available with its corresponding interpolator (see Section 2.3.2). An additional interpolator, *Dummy Interpolator*, is available in case no custom interpolation is desired at all. This would be the case if the a linearly interpolated plot, which is generated directly from the raw data values, is considered enough. Having this additional interpolator allows to have an empty shell that complies with the mentioned API so the main processing logic can interact with it like with any other interpolator. The gain is that new samples are discarded without any computation, and more importantly no matrix is sent, saving on overhead from transmission times in both Application Core and Data Processing sub-processes.

2.3.1 Interpolation methods

The interpolation process consists on the generation of a regular grid out of the incoming samples. This grid can later be drawn for visualization of the data. There are many different techniques for this task, with varying levels of complexity and computational requirements. The latter is especially crucial, since it must be feasible to compute updates relatively often in order to offer a real-time evolution of the surveyed area. A description of four common interpolation methods as well as an estimation of their performance is given in [11]. Out of these, the viability of a real-time approach as well as the accuracy of the results reduce the list to two main possibilities: Inverse Square Distance and Krigin interpolation.

The Inverse Square Distance method assigns values to each point in the grid based on nearby samples, giving more weight to closer ones. This turns it into an approximate method, as the original sample values themselves are not part of the interpolation function that is generated. The analytical expression obtained is:

$$f(x,y) = \frac{\sum_{j=1...N} w(d_j)v_j}{\sum_{j=1...N} w(d_j)} \tag{2.2}$$

where v_j is sample j, d_j is the Euclidean distance from sample j to any point of the surface, and $w(d)$ is a weighting function. Such a function can be fine-tuned for optimal performance. In this case, the suggested formula from [11] is used:

$$w(d) = \begin{cases} \frac{1}{d_{min}^2}, & \text{if } d \le d_{min} \\ \frac{1}{d^2}, & \text{if } d_{min} < d < d_{max} \\ 0, & \text{if } d > d_{max} \end{cases} \tag{2.3}$$

where d_{min} prevents infinite weight for samples overlapping a data point (d=0), and d_{max} avoids using too distant points. The advantages of this approach are the simple implementation and, more importantly, the limited computation and its scalability. Since for every new sample only points within a radius or d_{max} are affected, only those need to be updated and therefore the computational cost does not depend on the total survey area. These parameters could be configured to reflect a balance between accuracy and cost, with larger radius meaning more points needing to be updated for each new sample. On top of this, extrapolation of field values is not a problem, since the method tends to the average of all samples on far-away points. While this is not a good prediction, it does not yield unreasonably high or low values that would need to be filtered out to avoid disturbing the final plot of the grid. The cost for this is the scarce predictive features of the method in general: behaviors such as local maxima/minima are not accurately detected since the algorithm relies on a sort of weighed average of all nearby samples for each point.

In contrast, the Kriging method provides an exact interpolation, where data samples are part of the final interpolating function. The surface obtained through this method is described by:

$$f(x,y) = a_1 + a_2 x + a_3 y + \sum_{j=1...N} b_j k(d_j) \tag{2.4}$$

where $k(d)$ is the generalized covariance. Under certain conditions, Kriging can be equivalent to a spline interpolation for the following covariance expression:

$$k(d) = \begin{cases} 1 + \frac{1}{c_1}\left(\frac{dc_0}{d_{max}}\right)^2 ln\left(\frac{dc_0}{d_{max}}\right), & \text{if } d \le d_{max} \\ 0, & \text{if } d > d_{max} \end{cases} \tag{2.5}$$

with c_0, c_1 and d_{max} being constants to be selected. While the covariance is distance-limited in a similar way as the weighting function from eq. 2.3, the a_i and b_j coefficients are not: they are

obtained by solving a linear system of N+3 equations, N being the total number of original data points (samples). This means that for every new sample, ideally the whole linear system would have to be solved again, introducing notably higher computational costs. Furthermore, the longer the survey the larger the linear system would be, effectively increasing the cost for each new sample that is processed. While it can be argued that those coefficients could be re-calculated periodically instead of for every new sample keeping a high accuracy, this does not solve the scalability problem. In terms of the interpolation quality obtained through this method, [11] show that for a variety of cases the Kriging algorithm manages both the most and the least accurate results compared to the other three, when run under different configurations (e.g. sample density, selection of constants, etc.). This makes it an unstable choice, which would need careful tuning and adaptation to each particular case to offer the best results. Finally, a spline interpolation presents problems at the edges of the survey area, where the extrapolated function may take values out of the expected range, disturbing the representation of this data if not properly filtered. For its complexity and its scalability problems, the Inverse Squared Distance method was chosen.

2.3.2 Inverse Squared Distance interpolator

An interpolator is a totally modular implementation of an interpolation method. It is responsible for handling new individual data samples and processing them to update the resulting interpolated matrix. For this reason, handling such matrix (e.g. generation, expansion and efficient operation) is a great part of designing one. The current available interpolator is based on the Inverse Squared Distance method introduced above. The scientific computing library *numpy* is used for most operations, taking advantage of its optimized, C-based matrix operations (official documentation at [12]). For this reason, most computations are done in matrix form instead of an element-by-element way.

In order to initialize this interpolator, a starting data point must be given, forcing its initialization to happen after the first sample has been received. The reason is that a central point is needed to create the matrix around, since it is linked to a geographical position. To do it, an initial size of 30x30m is chosen to provide room for the first several samples. It is then filled with a regular array of points separated from each other a certain distance S (chosen by the user at application start). As a result, the initial matrix will contain $30/S$ (rounded down) rows and columns. The value S must however be turned from meters into degrees for both latitude ϕ and longitude λ. A spherical approximation is used to determine these values. A certain angle in latitude always translates to a fixed distance. However, looking at Figure 2.6 one can see that the length related to a certain longitude shift depends on the latitude at which said shift occurred. Considering this, different latitude and longitude steps are defined (in degrees) taking the latitude of the initial point:

$$S_{lat} = \frac{\pi S}{180R} \tag{2.6}$$

$$S_{long} = \frac{\pi S}{180Rcos(\phi_0)} \tag{2.7}$$

In this approximation, a relatively small survey area is assumed where S_{long} can be considered constant. For the scales considered, where a single survey covers a few kilometers at most (and often not all at once), this approximation is more than enough and does not introduce noticeable distortion when drawing the area.

The other consideration related to the matrix is its expandability. Since the total area of a survey is not known, the interpolator must be able to dynamically expand the matrix when a new sample gets "too close" to an edge. Specifically, whenever a sample is within d_{max} range of the edge of the current matrix, it must be expanded. The reason is that, as shown by equations 2.2 and 2.3, all points at $d < d_{max}$ from a sample are affected by it, so if no points have been created yet for part of this area

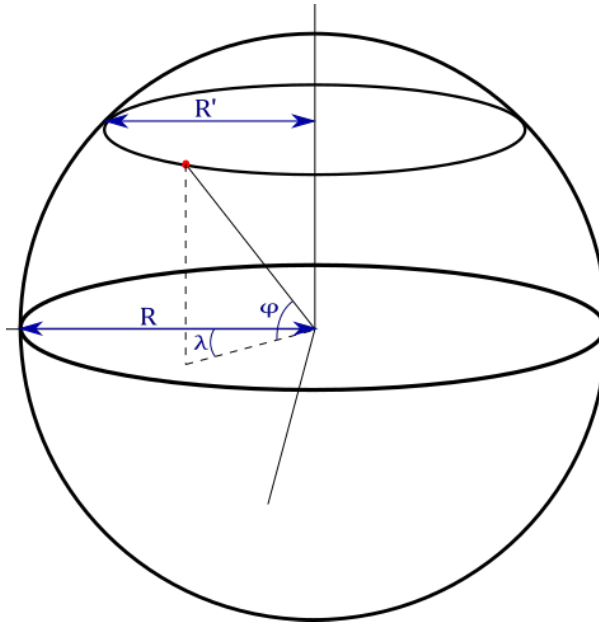


Figure 2.6: Latitude (ϕ) and longitude (λ) on ideal sphere.

(i.e. part of the circle with radius d_{max} goes outside the current matrix boundaries) they must be. When this happens, a sub-matrix full of zeros is generated and appended to the original one at the side where the boundaries were crossed, as shown in Figure 2.7. The total expansion achieved by this is of $(15 + d_{max})$m. 15m was chosen because if the matrix was expanded just enough to fit the new sample, it is very likely that every new sample when flying in a particular direction would cause an expansion. This means that memory for the whole updated matrix must be re-allocated each time, which is very slow. For this reason, a relatively large margin is given so that several more samples can fit in that direction. d_{max} is added on top to prevent still going out of bounds if an extremely large d_{max} value is set, which could make it so that the 15m expansion is not enough. This is however unlikely, since no noticeable benefit can be seen past $d_{max} = 6 - 10$m (being the default value at 6m). In order to ensure that matrix boundaries are never exceeded, they must be checked for every new sample before any interpolation is done to it.

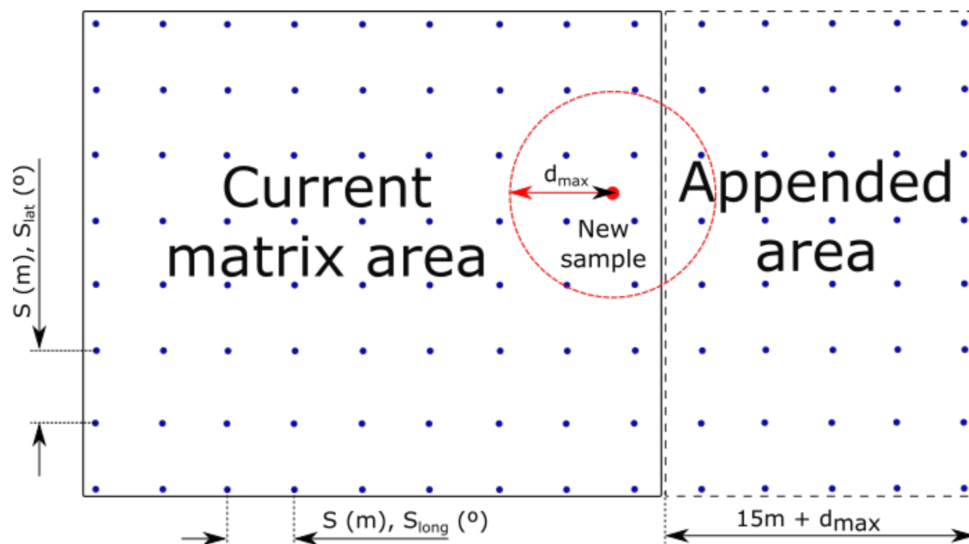


Figure 2.7: Matrix expansion.

Once the matrix is prepared, the Inverse Squared Distance algorithm is applied each time a new sample arrives. However, this would mean having to go through all previous samples if eq. 2.2 is applied directly. This can however be reformulated as:

$$f_{N+1}(x,y) = \frac{f_N(x,y) \cdot \sum_{j=1...N} w(d_j) + w(d_{N+1}) v_{N+1}}{\sum_{j=1...N} w(d_j) + w(d_{N+1})} \tag{2.8}$$

where f_N is the interpolation function for N points and f_{N+1} is the updated function once a new sample arrives. In other words, when new data is introduced, the new interpolated values can be obtained from the old ones if the old weights are known. When put in matrix notation and considering only scalar operations between equal-size matrices, it looks like:

$$M_{N+1} = \frac{M_N W_N + w(d_{N+1}) v_{N+1}}{W_N + w(d_{N+1})} \tag{2.9}$$

where M_{N+1} is the new interpolated matrix and $W_i = \sum_{j=1...N} w(d_j)$ an identically-sized matrix containing the accumulated weights for the first i samples. After every new sample, the accumulated weight matrix would also need to be updated:

$$W_{N+1} = W_N + w(d_{N+1}) \tag{2.10}$$

This provides a much more efficient solution than looping through the stored raw data array. On top of that, it achieves good scalability since the same matrix operation is done independently of the number of previous samples. The cost for it is the need to store the W matrix within the interpolator, applying to it the same updates and expansions when necessary as in the interpolated matrix. This can be further optimized by considering the behavior of the weighting function (eq. 2.3). Since it is 0 for every value outside of the d_{max} range, it means that W_{N+1} and M_{N+1} will only differ from W_N and M_N in the set of points where $w(d_{N+1}) > 0$, i.e. those within a circle of radius d_{max} around the new sample. This can be taken advantage of by defining a window around the coordinates of each new input that just fits such circle, and operating solely on the sub-matrix containing it. This is represented in Figure 2.8. Always operating in such a fixed-size sub-matrix M' further improves the scalability of this method, now being independent on both the duration of the survey and the area it covers (number of samples and total matrix size respectively). Note that the drawing and sending speeds will be affected by the total size of the matrix, even if the interpolation itself is not.

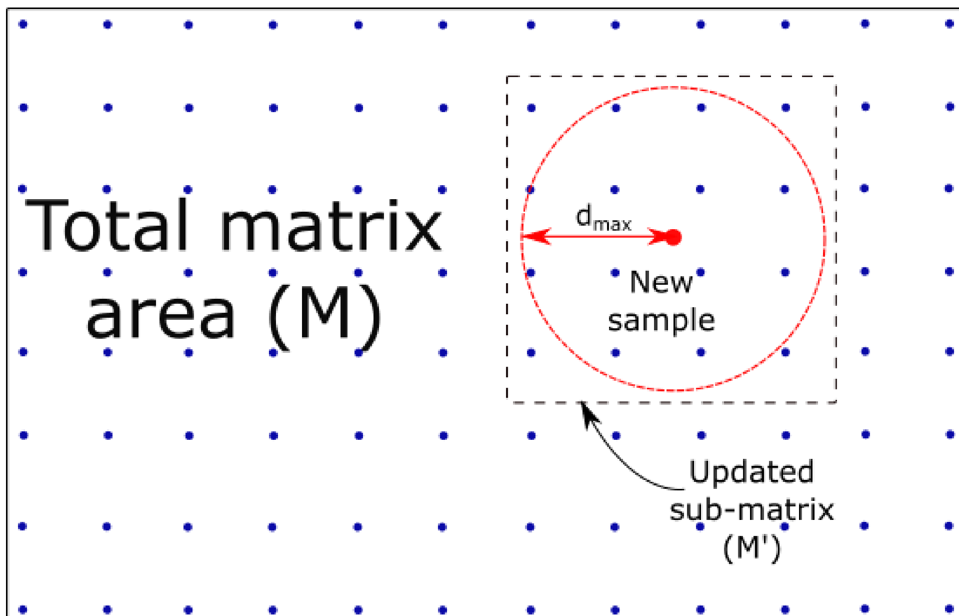


Figure 2.8: Sub-matrix window.

Regarding the distance calculations, they must be done using the coordinate system. The Haversine formula is used to achieve this, which provides the great-circle distance between any two points. Note that this still relies on a spherical approximation of Earth. However, the previous assumption where S_{long} was constant throughout the whole matrix does not affect distance results, since the actual coordinates of each point are used even if they do not represent an actual perfect rectangle when accounting from this. The formulation for this is as follows:

$$a = sin^2(\Delta\phi/2) + cos(\phi_1) \cdot cos(\phi_2) \cdot sin^2(\Delta\lambda/2) \tag{2.11}$$

$$d = 2R \cdot atan2(\sqrt{a}, \sqrt{1-a}) \tag{2.12}$$

where d is the distance between (ϕ_1, λ_1) and (ϕ_2, λ_2) (both in radians), and $\Delta\phi = \phi_2 - \phi_1$, $\Delta\lambda = \lambda_2 - \lambda_1$. This was implemented using the trigonometric tools included in the *numpy* library, which allow the function to be applied directly to the whole sub-matrix M' with respect to the new sample's coordinates.

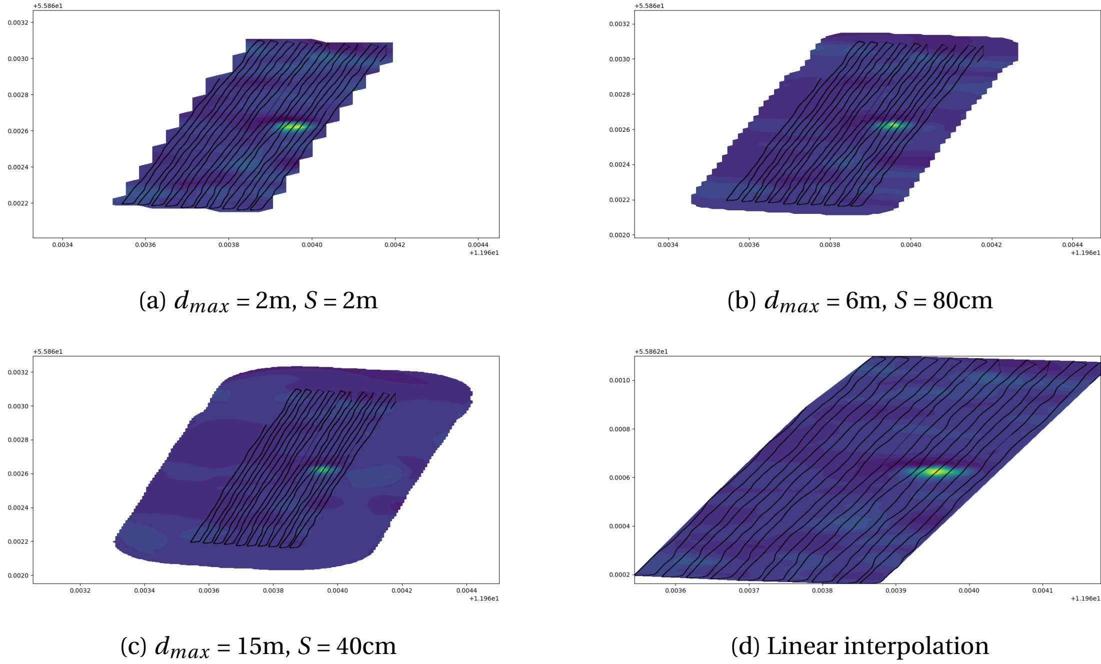
Lastly, there is the problem of values that have not yet been initialized, i.e. that have never been within d_{max} range of any sample. These must not be shown when drawing the interpolated matrix, and therefore must be kept track of. For this purpose, a *masked array* is generated from M and transmitted to the Application Core. It contains both the information from M itself as well as a parallel array of booleans identifying whether each point should be plotted or not. This does not add a large overhead, since a boolean matrix is nowhere near the size as a float matrix, especially considering that Python mostly uses full-precision (64-bit) floats.

3 Results and capabilities

The *LiveSensing* tool provides a preview of the magnetic field over a surveyed area. Its quality and usability rely on the accuracy of the results and the computational cost associated. These were tested using a vertical gradient configuration, were the magnetic data is extracted from a log file corresponding during a real survey. Single-sensor results are not extracted from this data set, because the available samples had the base geomagnetic field subtracted, effectively removing diurnal effects. While it does not affect gradient plots, since those effects would be filtered out by the gradient calculation itself, it would cause unrealistic results for unfiltered single-sensor data. Real test flight data is also included and discussed.

3.1 File-based analysis

First, the on-paper capabilities of the tool are tested using the File Data Gatherer. The pre-processed data comes from the log file of a 20Hz sampling rate survey. This allows replaying the same survey any number of times, varying parameters or even inter-sample delay. First, a general overview of four different configurations is shown in Figure 3.1. These range from the lowest quality available in 3.1a to the highest quality and most computation-heavy one in 3.1c. For comparison, 3.1d shows a linearly interpolated result. While they are too broad of a view to tell any detail, they offer a good visualization of the two main interpolator parameters described in Section 2.3.2 as well as what they represent. A larger update window, defined through d_{max}, gives a more accurate result since more samples affect the value of each interpolated point. Cell spacing however, represented by S, is related to the resulting resolution: the closer together the interpolated points are, the more are used to represent a certain area. A balance must be found between the two, since both affect the computational cost of the process. Also, for a fixed survey area having the points closer together generates a larger matrix, increasing the overhead due to transmission times between sub-processes as well as the drawing costs.



(a) d_{max} = 2m, S = 2m

(b) d_{max} = 6m, S = 80cm

(c) d_{max} = 15m, S = 40cm

(d) Linear interpolation

Figure 3.1: General survey area overview with different configurations.

In order to determine the accuracy obtained with each configuration, the previous results are analyzed again focusing around the detected anomaly. For the most lightweight approach, Figure 3.2 shows a detailed view of the results from 3.1a. While a magnetic anomaly is clearly detected, resolution is very lacking and smaller anomalies could be overseen. In comparison, Figure 3.3 shows the same area for the most taxing configuration. In this case, 25 times more samples are used to represent the same area, due to the difference in point spacing. This forced resolution increase gives a much finer grid where smaller anomalies can be detected. However, due to the weighted averaging behavior of the interpolation method, it also tends to generate artificial maxima and minima on grid points that fall very close to high-field sample. This explains why all the detected high field areas fall on top of flight lines. The result is a big tendency to highlight small anomalies even if they are part of a larger one. This may be beneficial if the target is known to be small, but can also result in artifacts where one large anomaly is split into several smaller ones. The limited prediction capabilities of this method also show that even small anomalies may be hidden if they fall between lines, since the interpolated surface is not accurate far from sample points. This limitation means that there is little benefit to be gained from even smaller S values, since at this point the resolution is essentially limited by the line spacing.

An alternative to the Inverse Squared Distance method is a simple linear interpolation between the samples. Figure 3.4 shows the same area as before processed in this way. The result is a higher resolution than what is offered by the lightest custom interpolation, with a more clear indication of where the center of the anomaly is. In this case, large anomalies can be accurately pin-pointed. This offers a notably higher quality than that of the lowest configuration for the custom interpolation. However, smaller effects may be hidden, making mid and high resolution custom interpolations interesting in certain situations.

One more consideration is that in a real scenario it is unlikely that stable a 20Hz transmission is possible, especially when long-range radio links are used. Instead, the usual approach is to use some sort of sub-sampling or sample discarding to make the communication more robust. This means that, while the entirety of measured samples are still getting logged at the aircraft side, only a portion of

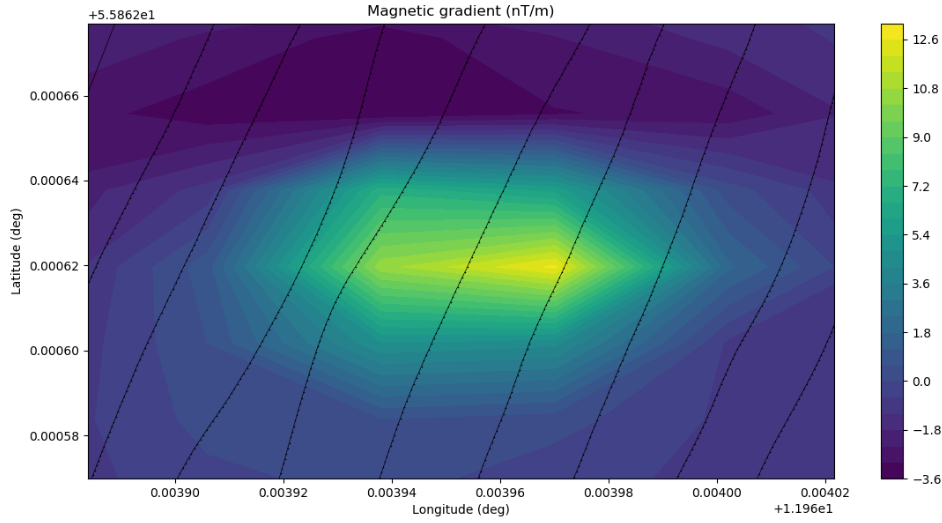


Figure 3.2: d_{max} = 2m, S = 2m, anomaly details.

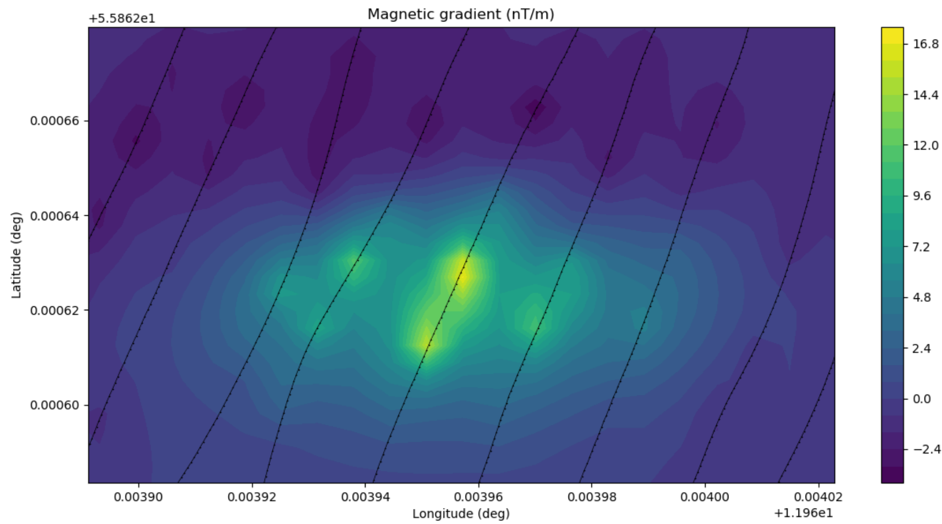


Figure 3.3: d_{max} = 15m, S = 40cm, anomaly details.

those are available for the real-time processing. In this case, the maximum resolution achievable is further decreased, due to the higher sample separation within one line. Figure 3.5 illustrates the quality difference when the original data is sub-sampled to 2Hz and the most demanding configuration is used. Compared to the full 20Hz sample set, small differences can be seen, but the result is for the most part the same. This effect is even less visible for lower-quality interpolation configurations, where the resolution is more limited by the point separation S than by the distance between samples. The reason for the very small impact when effectively having a 10x reduction in the total number of samples is related to the interpolation method itself. As mentioned above, one key limiting factor using the full 20Hz sample set is the separation between lines. The interpolator's inability to accurately predict the field between flight lines meant the effective resolution was limited even for very densely packed samples within each line. As a result, having such density severely reduced does not affect the results as much as it could be expected, since it did not offer a large benefit to begin with. This positions the Inverse Squared Distance method as a very viable interpolation technique when the

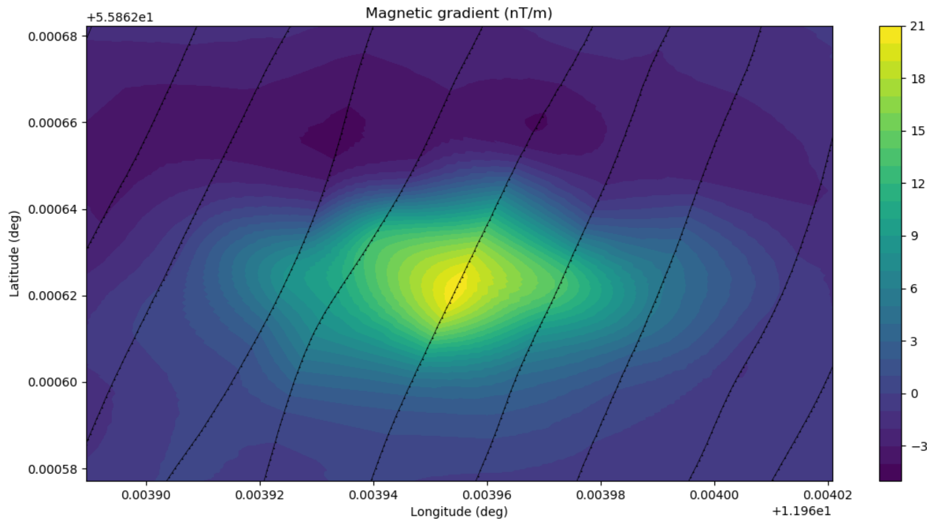


Figure 3.4: Linear interpolation, anomaly details.

combination of sample rate, line separation and flight speed cause a relatively even distribution of samples across the survey area. However, more complex alternatives may offer better quality results when this is not the case, e.g. when sample separation within a line is much smaller than the line separation.

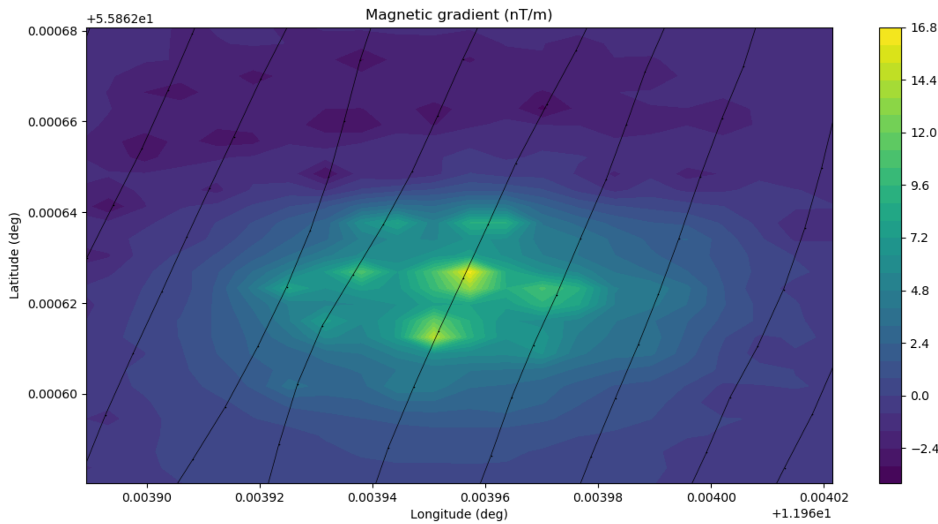


Figure 3.5: d_{max} = 15m, S = 40cm, anomaly details (2Hz).

The last consideration is the performance of each of the shown configurations. There are many contributions to the total computational cost. However, three types of operations stand out as the main causes for long execution times. These are:

- *Transmission overheads*: They are caused by two sub-processes being blocked while information flows between them. It has two components: access latency, i.e. the time that the sending sub-process has to wait for the other to start listening to the pipe, and transmission time. The

former can be assumed constant for every transmission, while the latter scales with the amount of information being sent. For this reason, transmission overheads are most noticeable between the Data Processing and the Application Core, due to the amount of exchanges and particularly the heavy interpolated matrix.

- *Drawing speed*: Drawing the flight trace needs relatively light computation. However, the heatmap associated with the interpolated data takes a lot of resources to draw. Even with the trigger-based implementation, which limits the plot update rate to the hardware's capabilities, this can have a very large impact on the rest of the program. The reason is that whenever an update to the plot is being generated, the Data Gatherer and Data Processing are likely to try to send more information before the new drawing is complete. As a result, both are forced to wait until the Application Core is free again to handle I/O operations.
- *Processing speed*: Since the processing implementation is totally linear, the Data Processing module does not accept a new sample until the current one is fully processed and interpolated. This can limit the maximum sample rate that can be handled.

It can be difficult to isolate the effect and delay caused by each of them in a single run. However, file-based simulations allow to repeat the exact same processing and operations under any conditions. In order to have an idea of the computational requirements of each case, the File Data Gatherer was modified so no delay was introduced between samples, effectively sending them to the Application Core as fast as it could handle them. A timing mechanism was also implemented, which measures the exact time the application takes from when the first sample is extracted to the moment the file is closed. This provides an accurate benchmarking tool to compare any parameter combinations to each other. Table 3.1 collects the execution times for a variety of scenarios, which can be used to measure their performance relative to each other. The same text file containing 14563 samples gathered at 20Hz was used, with a plot step of 2 for all tests. The environment and external conditions were kept identical, with no background processes that could influence the results. An average of three runs is shown for each case to account for small fluctuations. For reference, a Windows 10 installation running on an AMD Ryzen 5 2500U mobile CPU was used.

Configuration	**Execution time (s)**	
	NO PLOTTING	PLOTTING
Custom interpolation d_{max} = 2m, S = 2m	5.95	100.06
Custom interpolation d_{max} = 6m, S = 80cm	7.58	119.47
Custom interpolation d_{max} = 15m, S = 40cm	47.10	200.86
Linear interpolation	2.40	67.46

Table 3.1: Application benchmarking results.

Looking at the results, the most evident thing is the huge difference in execution times that deactivating the plotting of the results can make. Also, this difference is not a constant value across all four cases, but rather increases with the matrix size. This means that a light, low-density matrix takes less resources to draw than a heavy one. And when no matrix is needed at all, as is the case for the linear interpolation, the drawing times (difference between plotting and no-plotting results) are the smallest. In terms of the no-plotting times alone, the processing speed and transmission overheads are the main causes for these. The linear interpolation value should be taken as the baseline, since it involves no custom interpolator or matrix transmission/drawing. Compared to it, the low and medium-quality custom interpolations experience only a small time increase. Considering that these

two implement the full custom interpolation, and that one has to process a sub-matrix with over 50 times the number of points of the other (due to difference in window size and sample density), the very small delta between the two means that the processing speed is not a bottleneck. Therefore, for relatively small matrices the Data Processing sub-process is able to handle samples faster than the Application Core can provide them even at no inter-sample delay, with a slight time increase for larger transmission overheads. This changes for the highest quality configuration, where the no-plotting times jump noticeably higher than in the other two cases. While the total matrix size can be estimated around 4 times larger than that of the medium quality test, transmission times cannot account for the large difference between the two results. This shows the point at which the processing speed starts being a limiting factor, where the processing module does not process a sample faster than the application core can provide the next one. It must be kept in mind however that this configuration uses an update window much larger than necessary, which together with the low point separation generates a sub-matrix with over 5500 times the number of points of that of the lowest quality configuration. This sub-matrix has to be extracted and operated on for every new sample.

In summary, execution times depend mostly on plotting rates, with processing speed being a limiting factor on slow systems or very intensive configurations. To reduce the impact that plotting has, the plot step option is available. This offers the possibility to select the update interval of the graph, so that drawing takes less resources. During the carried tests, around 2-8 plot updates were handled every second, even though hundreds of updates were received in that time for the selected plot step. On a real use-case, more than two updates per second are unnecessary, and ideally the plot step should be set so that the Data Processing sub-process only sends updates at a rate that they can be drawn. That would greatly reduce the drawing load as well as the transmission overhead (especially for custom interpolation methods). For resource-limited systems, these tests show that limiting the drawing rate and the update window size are the most important optimizations. High-quality results can still be achieved for values of d_{max} around 10m and low spacing S, while reducing notably the computational costs compared to d_{max}=15m (400m^2 compared to 900m^2 for the update window area). In case maximum performance is required, linear interpolation proved to be faster while offering higher quality than the lowest-grade custom configuration.

3.2 Radio link results

The aforementioned capabilities are associated to a file-based data collection system. A real scenario would however require real-time connection to an aircraft, with all the associated difficulties that it can bring: range and bandwidth limitations, robustness, etc. To do this, the *MagPro*[13] tool was developed in a separate project. It establishes a point-to-point radio link between two Raspberry Pi systems through the use of E32-TTL-1W transceiver modules working at the 433MHz band. It defines a custom communication protocol for a semi-duplex link, and interfaces with the *LiveSensing* application through the dedicated Fifo Data Gatherer. It is designed for a 2Hz sampling rate operation, imposed by the maximum rate of the available GPS sensors. The tests here described use a GSMP-35U magnetometer [14] for data sensing. Unfortunately, the late availability of said radio link severely limited the testing possibilities. With that in mind, the following results are offered as a proof-of-concept of a real-time communication of aeromagnetic data, rather than a finalized product.

Due to hardware unavailability, all the tests presented here only use a single magnetic sensor. Ideally, a vertical gradient configuration should be tested as no diurnal effect removal is available for one magnetometer. This does not however affect the presented results, since no full survey was done. Instead, the few tests that could be performed in the available time focused on testing the radio link itself, particularly its range and robustness. Therefore, and because of the extreme time limitations, no scenario can be created where processing would be meaningful, since no proper grid of points can be extracted from the measured data. For similar reasons, sub-optimal WiFi antennae had to be used, which are expected to introduce noticeable loss at both ends. Appropriate 433MHz monopole

antennae are expected for any real application, which would achieve higher power efficiency and range.

The initial test session made use of the first working prototype of the radio link. As such, its resilience to changes in the radio conditions as well as general stability and synchronization between both ends were the main testing targets. To avoid the need for a suitable area where to fly the aircraft, movement of the sensor side was re-created by hand-carrying all of the necessary equipment. The results is a ground-level line-of-sight link, which was held up to 300m. After that, the connection broke and could not be re-established due to de-synchronization issues between both ends in the link. Other unstable behaviors were also identified and partially fixed for the next test. No graphical data is available, since no logging was used as it was a stability test. It must be noted that the achieved range should not be used as a reference, since in this scenario many more obstacles and reflections are present that negatively affect wave propagation. Furthermore, the unavailability of a proper portable power supply caused the processing side to be slightly undervolted, possibly influencing the maximum power that its transceiver could transmit and as a result the maximum communication range.

The late execution of the first session allowed only two days for fixing the encountered link instability problems before the actual flight test. This updated version of the *MagPro* radio link controller is the final one distributed with the *LiveSensing* application. In this occasion, a more realistic scenario was created where sensors were mounted on a UAV and flown in the same conditions as in a survey. Despite that, no proper coverage of an area could be done to test processing capabilities, since availability of the aircraft was heavily time-constrained. Two flights were performed to test general link behavior and range capabilities respectively. All data was logged, and the Replay Data Gatherer was later designed so these tests could be re-run and analyzed. Figure 3.6 shows the results for the first flight, where the top-left area containing the low field values represents the starting area. The low magnetic readings from this point are likely caused by the different measuring conditions, since the aircraft was static and at a lower altitude than during normal survey operation. The flight trace shows very high GPS instability, with large position jumps. Sample integrity was manually checked for all of these jumps, and no discarded or unsent samples were detected. This means the position inaccuracies are fully associated to the GPS module, which could not properly track the aircraft, and not to the *MagPro* or *LiveSensing* implementations.

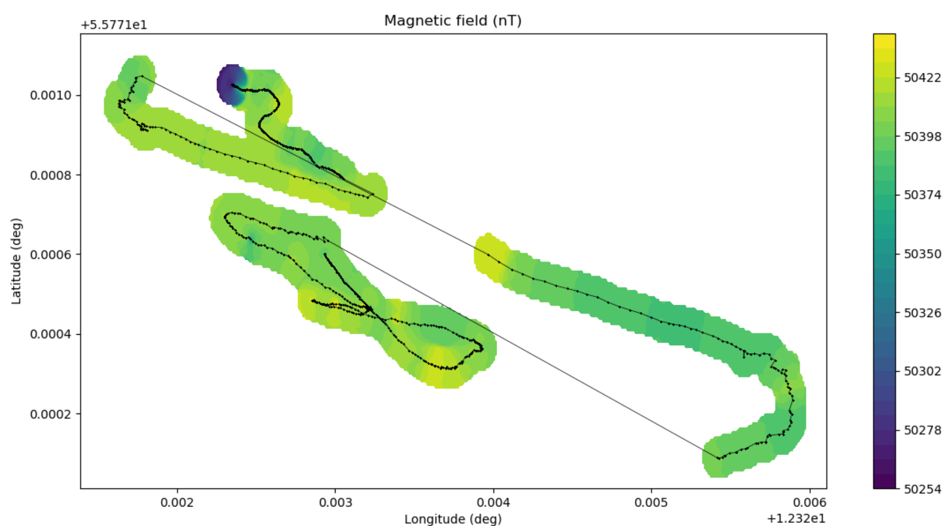


Figure 3.6: Test flight results (1). d_{max}=6m, S=80cm, 904 samples.

The second flight results are depicted in Figure 3.7. In this case, the main target was to find the maximum connection range, which was tested by flying away in a straight line. Compared to the previous plot, much higher GPS consistency can be seen. That hints towards long initialization times in the GPS module, which needs to properly synchronize with the available satellites. This idea is reinforced considering that the return path is totally stable. In terms of range, the achieved distance was 810m, measured through external UAV control tools to avoid GPS-induced errors. At this point the connection was still stable, and therefore the actual maximum range of this implementation is estimated around or over 1Km. This could not be tested due to legal limitations which prevented further flying, so the maximum distance is unknown.

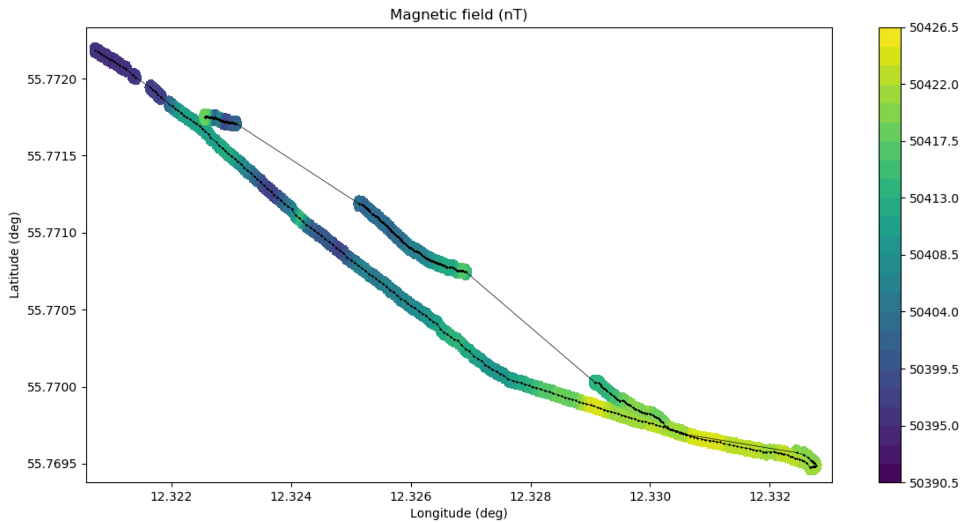


Figure 3.7: Test flight results (2). d_{max}=6m, S=80cm, 525 samples.

While more testing is necessary for definitive conclusions, these results offer some insight on the possibility of real-time magnetic processing. First, the most limiting factor can be identified as the GPS module, which introduced very large position errors on top of forcing a 2Hz sample rate, effectively discarding (not sending) 9 out of every 10 magnetic readings. Compared to the cost of the main surveying equipment, a slightly more advanced GPS tracking device seems very reasonable. That is expected to get rid of tracking errors, and it would allow higher sample rates. At the current selected configuration, the bitrate that *MagPro* could transmit corresponds to 4 or 5 single-sensor samples per second including GPS information. Up to a 10Hz rate could be achieved by using a more complex and optimized transmission protocol. As for range, 1Km or more is estimated to be viable with the used configuration, with over 800m being tested and guaranteed. This could be greatly enhanced with the introduction of proper 433MHz antennae, which again would add negligible costs to the project but unfortunately could not be obtained on time for the performed tests. An estimation of the improvement can be made by calculating propagation loss. Friis formula for free space propagation describes the received power as:

$$P_{rx} = G_t G_r P_{tx} \left(\frac{\lambda}{4\pi d} \right)^2 \tag{3.1}$$

for a constant transmitted power P_{tx} and antenna-dependent gains. Applying this, the attenuation is calculated and represented in Figure 3.8. For a 1Km estimated current range, the associated free-space attenuation at 433MHz is 85.2dB. While this cannot be improved, a better antenna selection can introduce additional power budget to be used for longer ranges. A conservative estimation of additional 5dB of gain from an appropriate antenna gives 10 more dB (5 from each link end) that can

be spent in propagation attenuation. Based on this assumption, a range of 3Km should be possible with an attenuation of 94.7dB. However, and despite choosing rather pessimistic values for current maximum range and antenna gain, this is an idealistic approach. First of all, proper line-of-sight radio links require a certain volume around the imaginary point-to-point line. A connection can be approximated as line-of-sight as long as the first Fresnel ellipsoid is free of any obstacles. The maximum radius of such ellipsoid is present at the middle point of the link, and can be calculated as:

$$R = \sqrt{\frac{\lambda d}{4}} \tag{3.2}$$

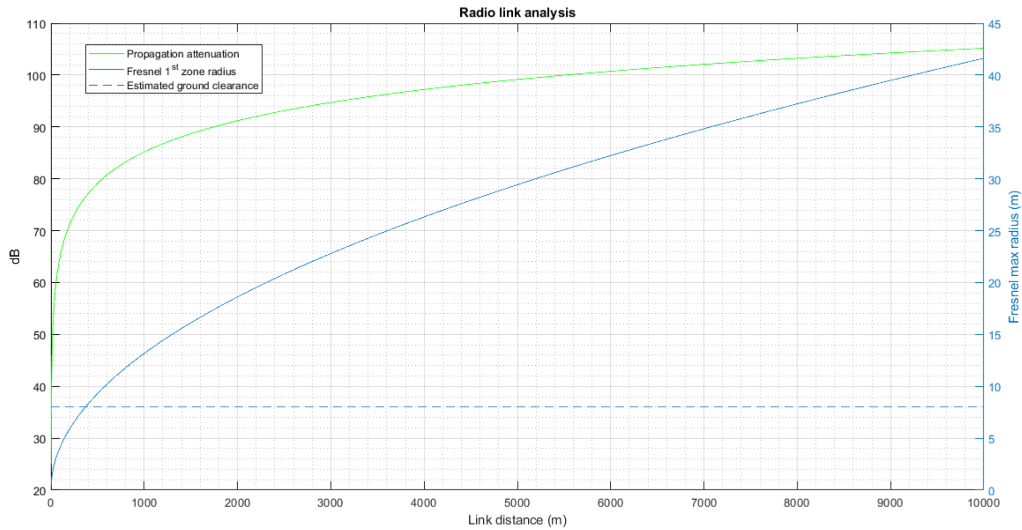


Figure 3.8: Free-space attenuation (green), Fresnel's first zone max. radius (blue).

Considering a flight altitude of around 15m and that the processing side is at ground level, clearance from the end-to-end line to the ground can be estimated at about 8m at the middle point of the link. The Fresnel zone maximum radius as well as this estimated level are also represented in Figure 3.8, showing that ideal unobstructed propagation cannot be assumed past the 400m range due to low ground clearance. On top of this, multipath propagation is expected from ground reflections, further affecting the result. All in all, the above estimation can only give an idea of the improvement that appropriate radio equipment could make, easily doubling the currently available range. More than that could be possible considering the conservative gain assumptions, so further testing is required. This extra range could allow remote survey possibilities, or it could be exchanged for faster and less robust transmission and coding schemes, bringing higher rates and therefore improved resulting quality.

4 Using the tool

This section focuses on the different features and concerns that directly affect the user. A quick guide of the installation process is offered, as well as a description of the different used dependencies. The different configurations and user-accessible features are described with a practical approach. Finally, certain non-trivial usability considerations are introduced. The goal is a self-contained overview of the available tools and techniques without the need to dive into the technicalities of it all.

4.1 Installation and dependencies

The *LiveSensing* application was built using Python 3. As long as all needed dependencies are available, full functionality is expected for all 3.X builds. Rigorous testing however was only conducted Python 3.5.X and above, and no working guarantees are offered for lower versions. On top of the standard library, some external packages are used. These are listed below, as well as the recommended version for each of them and a documentation reference when appropriate. All of them should be available through the *pip* tool for a simple installation. See [15] for details on setting up Python or *pip*.

- *numpy* (v1.16.11) [12]. Efficient multi-dimensional data computing.
- *matplotlib* (v3.0.2) [9]. Plotting utilities.
- *kivy* (v1.10.1) [6]. Graphical framework. Needs additional packages:
  - Direct dependencies. See the kivy installation guide for each platform [16].
  - *kivy-garden*. Handles kivy community addons. Needed for matplotlib backend [10]. This is included with the *LiveSensing* distribution, but the kivy-garden package is necessary.

In terms of compatibility, the application was built with the goal of no restrictions, and it is expected to run without issues on all major operating systems as long as all dependencies are available. Development tests however only covered Windows 10 and Raspbian release 8+. This said, the included *MagPro* tool relies on the use of the Raspberry Pi GPIO (General-Purpose Input/Output) pins for transceiver connection and is designed with Raspbian OS in mind, thus making it (and the associated Fifo Data Gatherer) available exclusively on that platform. See the *MagPro* documentation [13] for a physical connection and set-up guide.

4.2 User guide

The *LiveSensing* application consists on two main screens. They offer access to the full range of tools and configurations that a user may require for fluent operation during a field survey. Additionally, some other features and customizations are available outside the GUI that may come in useful for later analysis or development. Note that, while all necessary local services and dependencies are handled by the application, external ones must be launched separately. For the current implementation, this means that *MagPro* is started and interacted with automatically by *LiveSensing* locally, but it has to be initialized manually at the aircraft side.

4.2.1 Start screen

The start screen is a configuration panel with all the different available parameters. Figure 4.1 shows a capture of it, having three clearly defined sections. These set the operation mode and characteristics used for the sample processing. Once these are selected, the "START" button launches the main application screen. This locks the chosen configuration for the duration of the survey. Therefore, if some options have to be changed after entering the main screen, the application must be restarted.

The first section is labeled "Interpolation". It defines the technique used to transform the scattered sample points into a regular grid of points that can be drawn. Two methods are shown, which should be selected depending on the scenario and hardware capabilities:

- *Linear Only*: Linear interpolation is a simple and fast method that generates a plot directly out of the scattered data samples. When selecting a custom interpolation method, it is possible to switch to a linearly interpolated plot at any time. However, this means that the regular grid is being generated but not being plotted. If hardware capabilities are a concern, selecting this option enforces that only linear interpolation is available, which means that no processing time

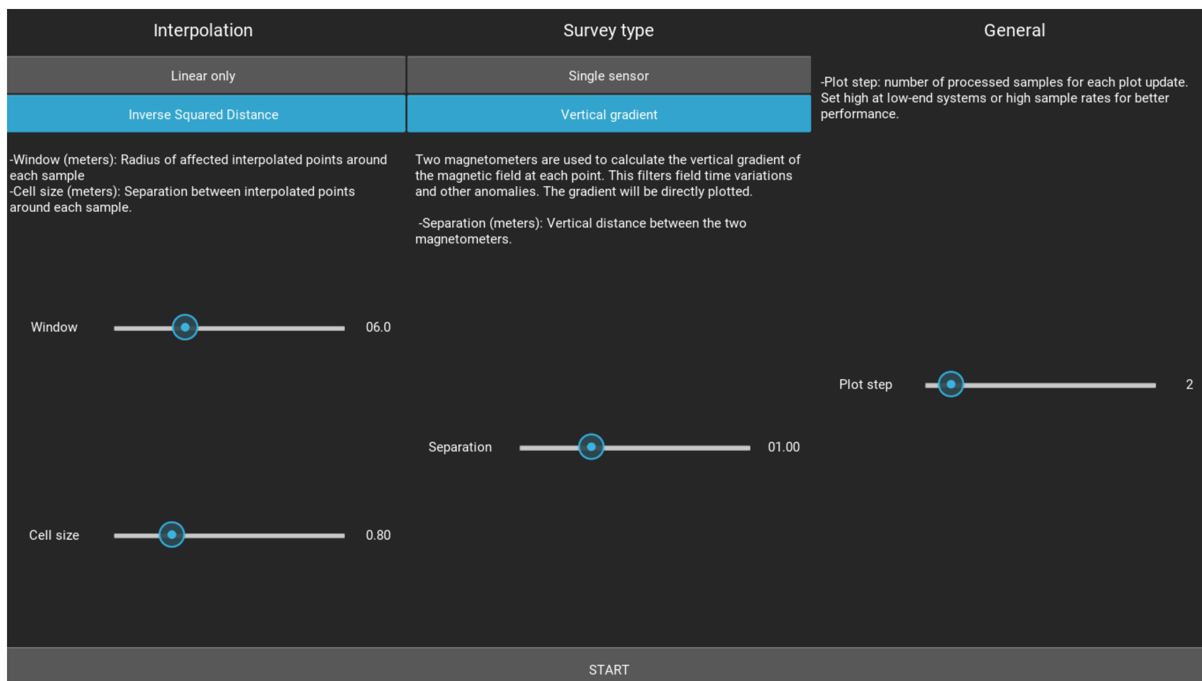


Figure 4.1: Start screen.

will be spent at all creating such grid. In exchange, the option to toggle between linear and custom interpolation is lost. No parameter selection is available for linear interpolation.

- *Inverse Squared Distance*: A more advanced alternative to linear interpolation. Each new sample updates all the interpolated grid points within a certain distance of it, customizable by the "Window" parameter. Higher values produce more accurate results, at the cost of increased computation. Window sizes larger than 10m are usually unnecessary and have small benefits. The separation between adjacent interpolated points can also be chosen through the "Cell size" slider. Smaller separation yields a higher-resolution matrix, but again at the cost of longer processing and drawing times. If performance is a concern, the "Linear Only" option is recommended over the lowest-quality custom configuration, since it often offers similar or better results while being slightly faster (see Section 3).

Next, "Survey type" describes the method used for evaluating the magnetic field. "Single sensor" takes in data directly from one magnetometer and performs the interpolation and plotting out of it, with no further correction or filtering. "Vertical gradient" relies on two sensors vertically aligned to calculate the gradient of the field at each spot. In this case, the separation between them must be entered since the results are inversely scaled by it. Whenever possible, "Vertical gradient" is the recommended configuration since it filters out time-varying effects that cannot currently be avoided in "Single sensor" mode. Note that independently of the number of magnetometers used, "Single sensor" can always be selected, in which case the data from only one of them is used (the first one from each sample, non-configurable).

Finally, the "General" section is meant for parameters that are common to all interpolation and field measurement methods. It includes the "Plot step" option, which defines every how many samples a redraw attempt is done on the general plot. Since updating the plot takes a large amount of resources, this option allows to adapt the update frequency to the available hardware and sample rate. It must be noted that even for a high selected update rate (low plot step while having a high sample rate), redraws will only be done as often as the hardware can handle, skipping some redraw attempts if they happen to often. Even then, an attempt by itself consumes some resources, so the plot step should

be set such that no more redraw attempts are scheduled than the hardware can manage to complete. Higher plot steps result in better performance due to less plot updates. In a real scenario, having the graph update more than 1 or 2 times per second is usually unnecessary.

These sections contain parameters for the currently implemented features and tools. However, it is likely that some future upgrades to the application will require additional configuration possibilities. For an overview on how to add some elements to the existing GUI layout, see Section 5.

4.2.2 Main screen

The main screen is the one that is present for the majority of the survey. Once it is loaded, the data processing parameters are fixed and cannot be changed. Its elements, shown in Figure 4.2, include a dynamic graph as well as a variety of control buttons. The plotting configurations for the graph, as well as the sample gathering flow, can be configured through these. The graph itself offers no direct interaction possibilities, and behaves as a changing image. The buttons can be divided into three groups, each for a certain set of tasks.

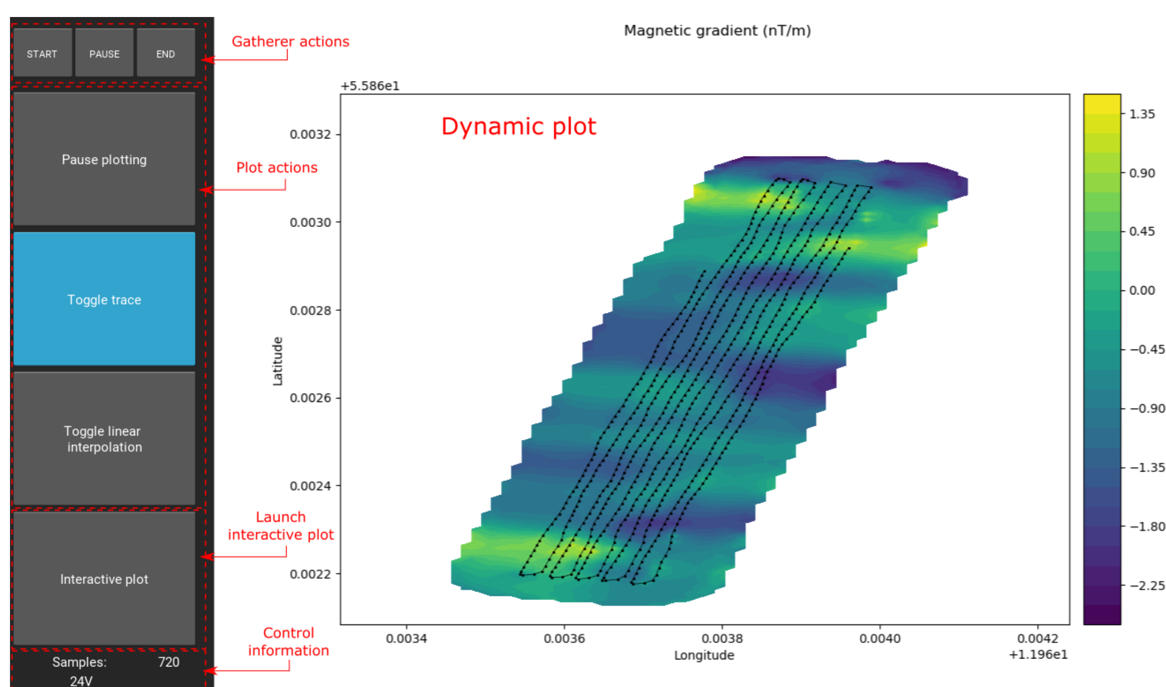


Figure 4.2: Main screen.

On the top left corner, three "gatherer actions" are available. Each of them is linked to a specific command, which is sent to the Data Gatherer when it is pressed. This gives three different actions or instructions to help control the incoming data flow. Their associated exact behavior depends on the Gatherer implementation (see Section 2.2). After each of them is processed, the Gatherer notifies so and a "Command received." temporary popup is shown. This is meant as feedback for the user, which can be useful under poor radio conditions. The suggested effects for each command, which are implemented in the currently available gatherers, are:

- "START": Starts the data transmission. On a radio link communication, it also starts logging data on the aircraft side. Also used for resuming the gathering after pausing.
- "PAUSE": Temporarily stops gathering data. Useful when some set of samples is not wanted, e.g. when in need of a battery change. Closes the current log file at the aircraft side and starts a new one, so as to simplify post-processing.

- "STOP": Stops the data transmission. Closes the log file and terminates the radio link and gatherer. To be used at the end of a survey.

Below them, three toggleable buttons define the plotting mode. Each of them works individually as a flag, and they do not interfere with each other. They affect the information that is displayed on the dynamic graph, either by hiding/showing it or by altering the way it is drawn. Their effects are:

- "Pause plotting": It stops representing new updates to the graph. Those are still received and stored, so as soon as plotting is resumed the latest information is shown. Useful for focusing on an area during a survey without a constantly updating graph, or to avoid spending resources on low-end machines. It also provides a way to accelerate file-based survey replays (see performance results at Section 3).
- "Toggle trace": Shows/hides the flight path plot. Meant for preventing the trace plot from hiding important information.
- "Toggle linear interpolation": Switches between drawing the regular grid from a custom interpolation and a linearly interpolated plot. Can be used to compare two processing techniques in case artifacts are present on the plot. This button is not available if "Linear only" is selected as the interpolation method, since no alternative to a linear solution is available.

The last button is labeled "Interactive plot". When pressed, it creates a separate window containing a fully interactive plot in the default matplotlib environment. This allows actions such as zooming and dragging, as well as getting exact coordinate values when hovering over a certain point. It also allows saving the figure for future reference. The graph contains the exact same elements as the one from the main screen at the moment the button was pressed, including the trace and interpolation configuration from the plot actions. It does not however update with subsequent samples, and is meant for detailed inspection or exporting the plot. For a smooth operation, it is recommended to pause plotting in the main screen when handling the interactive plot, since it frees up some resources. Note that while it runs in a separate window, it is still linked to the *LiveSensing* application, and will therefore be closed when the main application is terminated.

Finally, control information is available in the bottom left corner. This includes a sample counter and battery information. Having the total sample count can be very valuable, since it acts as a simple check for connection stability. This would be hard to determine by looking at the graph alone due to the plot step. Furthermore, it is common to have to discard a batch of samples due to invalid integrity on the magnetic data, but this does not represent a bad connection. In contrast, a steadily increasing counter is not affected by any of these and can be relied on. As for the voltage indicator, it is used to determine the state of the battery powering the magnetometer(s). External aircraft control tools can generally not provide this information, since the magnetometer power source is usually a separate one. This information relies on the magnetometer model sending the voltage reading, and was implemented with the GSMP-35U sensor model [14] as a reference.

4.2.3 Managing Gatherers

A gatherer represents one specific way of collecting data. By default, *LiveSensing* is configured to get real-time samples from a radio link controlled by the *MagPro* tool. This is the general expected behavior and use case, and it is not likely to be switched in the middle of any field testing session. For this reason, and to avoid cluttering the interface, no graphical configuration was implemented for gatherer configuration. Selection of a certain gatherer can be done by changing one specific line of code within the `src/main_screen.py` file, at the import headers:

```
from data_gatherers import XXXXXXX as DataGatherer
```

where XXXXXXX stands for the desired one. Three are three gatherers currently implemented in src/data_gatherers.py, two of which are mainly for testing purposes:

- FifoDataGatherer: Default one, handles interactions with the *MagPro* tool. Meant for real field testing.
- FileDataGatherer: Reads data from a text file, where the format matches that of the local log files (with their headers stripped). Meant for replaying surveys during post-processing for further analysis. Also useful for testing new application features, and as a reference for other gatherers. Can be used on files with any number of magnetic readings.
- ReplayDataGatherer: Same purpose as the FileDataGatherer, but meant for the log file format generated at the aircraft side. Useful for survey replay in case the local log is missing. The input file is selected in the same way. Does not generate a log file itself.

Both file-based gatherers read from a certain text file within the input_files folder. The file name can be selected through the file variable within the __init__ method in both cases. Additionally, n_magnetometers sets the number of magnetic readings in the file for the FileDataGatherer. Having both the gatherer and its parameters selected directly through the code allows it to follow a very simple design. As a result, implementation of new gatherers is very straightforward and requires no interface or core code modification.

In general, gatherers are expected to generate log files in the log/ directory (although it is not mandatory). The suggested format, implemented for the FifoDataGatherer and the FileDataGatherer, is:

```
LONGITUDE LATITUDE READING_1 READING_2 ... INT_LOCK BATTERY
```

where there are as many reading values as magnetometers in the survey. All values are stored as a float (coordinates in decimal degrees), except the integrity lock which is either 1 or 0. Battery information represents the power supply voltage reading from the magnetometer at the time of each sample.

5 Expanding the tool

Modularity and upgradability are two of the core ideas behind the design of *LiveSensing*. As such, certain components can easily be added without the need to go through most of the application code. These are meant to bring new features or compatibility with other tools. The two main modular aspects come in the form of Data Gatherers and Interpolators. On top of these, broader modifications are also simplified by the isolation between sub-processes shown in Figure 2.1: changes to one of the three sub-processes do not affect or interfere with the other two, since their only connection is through a single, well-defined pipe.

5.1 Creating a new gatherer

All gatherers are defined into src/data_gatherers.py. Their general design was kept as simple as possible, with the few associated parameters hardcoded into them. The reason is that these modules are identified as the most common upgrade point, possibly needing several different gatherers for a variety of data streams. The only hard requirements are the need to listen from and write to a pipe that connects it to the application core. Listened-to packets contain commands that may each be processed in any way (or not at all). Implemented commands (each a different string) as well as their suggested associated effect are listed in Table 2.2. As for packets written to the pipe, a more strict dictionary-based format is required. Each field of the packet is accessed through its key, which is a

string identifier. These fields and their keys are listed in Table 5.1. Notice that most fields apply only to the "sample" packet type. In contrast, an "ACK" packet only needs the "type" field. All unexpected extra values are ignored in the application core and sent transparently to the processing side, as suggested by Figure 2.2.

Dictionary key	Field type	Field description
"type"	string	Identifies ACK ("type": "ACK") or sample ("type": "sample") packets.
"longitude"	float	Longitude coordinate in degrees of the sample's location.
"latitude"	float	Latitude coordinate in degrees of the sample's location
"reading_n"	float	n-th magnetic reading for the sample. As many as magnetometers are used. Starts at "reading_1".
"battery"	float	Magnetometer power supply voltage. If multiple are used, the one corresponding to "reading_1".
"integrity_lock"	boolean	Validity of all magnetometer readings. If any is invalid, "integrity_lock":False.

Table 5.1: Gatherer packet fields.

Based on the gatherer architecture suggested in Figure 2.4, a bare-bones code can be provided for any given gatherer. The threading library is used for the thread implementation, but no additional imports are needed since it is also used for the already-implemented gatherers:

```
class NewGatherer:
  def __init__(self, pipe):
    self.pipe = pipe

    t1 = Thread(target=self._data_thread)
    t2 = Thread(target=self._command_thread)
    t1.start()
    t2.start()
    t1.join()
    t2.join()

  def _data_thread(self):
    while True:
      packet = {}  # Initialize packet
      # Receive new data
      # Format new data
      # Fill in packet fields
      self.pipe.send(packet)
    self.pipe.send(None)  # Poison pill at termination

  def _command_thread(self):
    while True:
      command = self.pipe.recv()
      # Process command
```

Notice that ACK packets are not used in this example. Whenever an ACK is sent, a "Command received" message is shown to the user, and so they can be useful in cases when good radio conditions are not guaranteed during a survey. However, their implementation depends heavily on the specific gatherer. It is important to remember that if the command thread is responsible for sending them, thread safety must be added around any pipe writing: if both threads write to the same pipe at the same time, data may be corrupted. See the available FileDataGatherer code for a simple implementation of this, as well as for a data logging example. A thread termination condition is not included either, so the poison pill is never sent. It is recommended to implement a way to break out of both loops when no more gathering is expected, e.g. when a "stop" command is received. This causes a chain reaction that also terminates the Data Processing sub-process as well as some threads in the application core, freeing up resources for a more smooth interaction with the final plot.

Once a new gatherer is implemented, it is selected by importing it directly at `src/main_screen.py` as described in Section 4.2.3. The lack of GUI implementation means that no further modifications are needed outside. The disadvantage of this is that gatherer parameters must be coded directly into its class definition.

5.2 Creating a new interpolator

An interpolator represents the technique used to generate the plotted grid. High-quality plots may require alternative, more sophisticated methods than the available Inverse Squared Distance. Implemented new interpolators are accessed within `src/processing.py`, where all the code for the Data Processing sub-process is located. They have a defined, simple API that the main processing loop uses to interact with them. This results in a bare-bones interpolator having the following methods:

```
class NewInterpolator:
  def __init__(self, pipe, parm_1, param_2, ..., param_n):
    self.pipe = pipe
    # Initialize other internal parameters

  def new_sample(self, new_longitude, new_latitude, new_field):
    # Handle new data sample

  def send(self):
    # Send updated interpolation graph to Application Core
```

where `param_i` are the different parameters associated with the interpolator operation. Having it handle the transmission of its results allows implementations like the `DummyInterpolator`, where all three required methods return immediately with no defined instructions. This effectively avoids all computation and sending overheads when the "Linear Only" interpolation mode is selected, discarding samples while complying with the mandatory API. It also gives flexibility for multi-threaded interpolators, allowing them to implement thread safety to all accesses to their stored matrix if necessary, as opposed to if such data was extracted and sent by the main processing loop. On a normal scenario with data actually being sent, the interpolation packet follows the dictionary-based structure listed in Table 5.2.

Dictionary key	Field type	Field description
"type"	string	Differentiates raw sample and interpolated data. Always `"type": "interp"`.
"long"	float array	1-D array of longitude values associated to interpolated matrix.
"lat"	float array	1-D array of latitude values associated to interpolated matrix.
"data"	float matrix	2-D array containing interpolated data.

Table 5.2: Interpolator packet fields.

Unlike new gatherers, new interpolators require a more complex process than defining the interpolator class itself. The reason is that they are integrated into the application GUI, making it easy to switch between them as well as to configure their parameters. This also means however that, when creating a new one, it must be implemented into the application core's start screen with its parameter selection menu. This is all done within the `src/start_screen.py` file, where an `InterpolationAccordion` class is defined that contains all the available interpolation methods. In order to add a new one, the following lines must be added within its `__init__` method:

```
new_interpolator = NewInterpolatorItem()
new_interpolator.bind(collapse=self.update_selected)
```

```
self.add_widget(new_interpolator)
```

where `NewInterpolator` is a class representing the parameter menu layout for the desired new interpolator. Each different `InterpolatorItem` must contain a dictionary named `self.params` with its current parameter selection as well as a `"type": "interpolator_name"` field. Whenever one of the interpolators within the accordion is un-collapsed (i.e. selected), the item's `bind` method shown above calls `self.update_selected` from the `InterpolationAccordion`, which keeps a reference to which of its items is the selected one. Once the user proceeds to the main screen, the parameter dictionary from the selected interpolation method is passed transparently to the Data Processing sub-process. All `InterpolatorItem` classes must inherit from kivy's `AccordionItem` class, which defines the core accordion functionality as well as GUI-related methods such as `bind` or `add_widget`. See the `InverseDistanceSquaredItem` class for an example of the specific syntax needed to design the parameter selection layout.

Once at the Data Processing module, the dictionary is passed to the main data collection method, `packet_collection`. This is done in order to wait for the first sample, since a starting point in space is often needed in order to create the initial matrix around it. For this reason, the interpolator is not initialized until the data transmission has started. Once the first sample is received a new field is appended to the dictionary: `"point_zero": (long_zero, lat_zero)`, where the tuple contains the longitude and latitude of the initial sample. The `"type"` field is then extracted and the corresponding interpolator is initialized based on its value. Notice that if the keys associated to all dictionary fields match the variable names in the interpolator's `__init__` method, dictionary unpacking can be used instead of manually extracting all parameters.

5.3 Future work

The results obtained with the current version provide invaluable feedback. Thanks to it, existing issues can be identified as well as possible solutions for them. Additionally, during the development of *LiveSensing* certain ideas had to be discarded due to time constraints. As such, some upgrade paths are proposed for the immediate future, either to improve existing behavior or to bring additional features.

First of all, the data gathering process can be improved on radio link scenarios. The `FifoDataGatherer` and link handling tool, *MagPro*, use a FIFO structure (also called named pipe) to communicate between them. This implementation can be optimized, so data does not require extensive string parsing and formatting for every sample. Additional features to the radio communication protocol could also greatly enhance link resiliency, such as adaptive sampling rates. That would require a more flexible gatherer, since sample discarding would be needed in slow systems. In the current implementation, where the sampling rate is fixed at 2Hz, hardware limitations are not an issue and therefore all data is processed. If necessary, a timeout would have to be implemented around the gatherer sending routine, such that if the application core is too busy to receive more data for a long time such data is discarded. This would prevent an accumulation of unprocessed samples.

When thinking about adaptive rates, having a fixed plot step is not a good solution either. Instead, a time-based plot update schedule is more fitting. This way, the graph would not be forced to be re-drawn too quickly when high rates are received, or cause large delays between visual updates at low rates. It would require an external timer within the Data Processing module that would be responsible for sending new processed results on a fixed period, independently of how many samples are received.

Another possible upgrade to the Data Processing module is the accuracy of its interpolator. The current implementation provides an efficient and scalable solution, but does not perform as good as other

alternatives in terms of high resolution results. Section 2.3.1 mentions that one option is the Krigin interpolation technique. To partially overcome the associated scalability problem, solving of the large linear system could be handled by a parallel thread that does the computation periodically, instead of every new sample. Additionally, long surveys would need some way of splitting the interpolated area, or else the computational cost would steadily increase with the number of samples. As for the extrapolation issues, careful filtering would be needed so no points outside the sample-covered area are considered. Another option would be to apply extrapolation-safe techniques at the edges, using other interpolations such as the Inverse Squared Distance method. Merging the areas interpolated by both methods would however present a problem.

On top of the interpolator, one notable weakness of the current implementation is the lack of single-sensor field filtering. While gradient computations take care of this when two magnetometers are available, the lack of a base station means that undesired time-varying elements cannot be filtered out. Although real-time fixed measurements would provide the best solution, the effect from time-varying contributions can be dimmed by modeling them through mathematical approximations. These could improve the results for single-sensor flights.

Finally, transmission overheads can be significantly reduced. At the current state, the results from Table 3.1 show a reasonably low impact in overall performance, so they were not considered a priority. However, they can grow significantly when accounting for vast survey areas. In these cases, the interpolation matrix can be big enough to present a problem if it has to be transmitted every time there is a graph update. This effect can be significantly reduced by sending only the updated window instead of the whole matrix. Indexes defining the window's location within the grid would also have to be sent, so the correct values can be updated within the `Graph` class. The whole matrix would still have to be transmitted sometimes, i.e. when it had to be expanded. The reason is that, while matrix expansion can also be handled within the application core, the overhead savings are probably not worth the added complexity and computational load in the graphing logic.

6 Discussion

Traditional magnetic surveying techniques suffer from long processing times. Real-time visualization of data can alleviate this problem. For it to be feasible, alternative methods must be developed. These should offer lighter computational costs as well as be applicable on a per-sample basis, rather than requiring the entire final grid. As a result, real-time techniques are faced with clear limitations not present for post-processing approaches, meaning that they are not expected to be matched in quality and precision. It is not strictly necessary however for the entirety of the processing and drawing routines to be executed with every single new reading: modern hardware often offers quite high sampling rates, not being practical to aim for a redraw at each sample. A balance must be chosen between result quality and update interval. Complex techniques are usually associated with higher details and accuracy at the cost of longer execution times, which lower the maximum plotting rate. Additionally, the fact that real-time techniques are presented as a tool for on-the-field assistance enforces the need to make the implementation easily portable and accessible. This can present further connectivity and/or computation complications.

LiveSensing is implemented as a lightweight application capable of simple real-time plotting routines. The Raspberry Pi was chosen as the main target system. The reason is that it provides a small platform that can easily be taken to the field with the rest of the measuring equipment. Its I/O possibilities make a large range of modules easily accessible to it, which may not be as simple for something like a standard laptop. The main feature in this regard are its GPIO pins, which are used for interfacing with the radio transceiver. This choice however also presents important drawbacks. First of all, computation is severely restricted for such a system, which narrows down processing possibilities.

For the implemented solution, Table 3.1 shows that a modern mid-range laptop can handle a high-rate data stream at an unnecessarily demanding configuration. In comparison, a Raspberry Pi would be close to its limit even when working with the sub-sampled version of such data. Additionally, having a portable processing solution means that a direct radio link must be established with the aircraft. This introduces range and bandwidth limitations, as well as the need to fully handle such connection.

An alternative would be to rely on the mobile network for radio connection. This could be achieved by introducing a 3G or 4G module in the aircraft. Range problems would be solved, since as long as mobile connectivity is available the data could be transmitted. Bandwidth would not be an issue either, as a mobile connection can provide more capacity than what is typically available through a long-range radio link. Finally, processing could be centralized in a server as opposed to having a very light system being responsible for it, mitigating the computation limitations. The processed results could then be made accessible through the internet (e.g. via a web interface) so they are visible by the surveying team. The big drawback is that this approach is strictly limited by mobile network coverage. As a result, real-time data visualization would not be available at all in most unpopulated or remote areas, as well as over large lakes or deep into the sea/ocean. Since it is not uncommon for surveys to take place on these locations, it represents a big limitation. In consequence, this option was discarded despite the clear advantages that it can offer when accessible. An always-available solution was therefore prioritized over a potentially higher-quality and less restrictive approach that may not be usable in some locations.

As for implementation, the current design spends most of its processing resources on the interpolation techniques, with gradient computations used as filtering when possible. As a result, single-sensor plots are expected to perform notably worse than double-sensor ones. This is unfortunately imposed by the lack of base station equipment, which gives no control over diurnal effects. In terms of result quality, gradient plots from Section 3 show that magnetic anomalies can be successfully located by real-time operation. This fulfills expectations and shows potential as a useful surveying tool, offering both a result preview and, more importantly, feedback on possible abnormal behaviors (e.g. due to human error). Single-sensor effectiveness is however expected to be lower, especially in long surveys where diurnals have a stronger effect. The lack of a raw data set means that the exact effect cannot be tested. Performance-wise, the multi-process architecture achieves high resource utilization that allows a light system like the Raspberry Pi to keep up with the gathering rate. The largest bottleneck in moderate configurations (or linear interpolation) lies in the drawing speed, with a plot update rate over 2Hz being completely unachievable and even 1Hz being hardly maintainable for medium-sized matrices. This points towards a scalability issue, where prolonged surveys may have too much data to redraw. One possible yet complex solution would be to split the plotting area into regions which are updated individually if necessary, although several architecture changes would be needed for this.

Real world results show a big difference to file-based ones, mainly due to the experienced tracking instability. The result is an unusable plot due to sudden position jumps. These problems were associated to the GPS module, and a more stable one should eliminate most of them. This would not bring perfect tracking accuracy though, since GPS measurements can have up to several meters of error even under ideal conditions. Such deviations usually change slowly, and so they are expected to introduce a semi-constant position shift for the whole survey. While that can be problematic when computing the final, post-processed mapping, it is not here. The reason is that general survey monitoring and data integrity are far more important than the precise locations of mapped points. The exact impact that these errors can have in a full survey is unknown and testing is required, since rapid changes in their value could greatly affect validity of the plotted data. In such case, additional GPS sensors would be necessary to apply error correcting measures.

Aside from tracking problems, radio link results offer a much more optimistic view in other aspects.

Range is estimated at 2Km or more for adequate hardware, which should provide enough reach for the vast majority of cases. Accurate testing is highly recommended, since this is a conservative estimation and better values may be achievable. Available capacity is also considered enough for real-time needs, managing up to 4-5Hz of single-sensor sample rate or possibly close to 10Hz for more efficient transmission protocols. This could be further increased by using more bitrate-oriented coding schemes when range is not a concern. Although the results from Section 3 show small benefits from high rates for the current implementation, more advanced methods could make use of them more effectively. The unused capacity could also be used for additional control data, giving many more monitoring possibilities.

7 Conclusion

Traditional geomagnetic surveys suffer from a lack of feedback during data gathering. A real-time mapping software is implemented for data visualization and monitoring purposes. It shows that even budget modern hardware can be used for simple real-time processing routines. Simplified techniques are used to achieve plots that offer a good approximation of the magnetic field over the surveyed area. Customization of processing parameters allows for performance adjustments to fit hardware with different computational capabilities. Unavailability of adequate hardware presented important challenges, and is the main attributed cause to the poor real-world performance observed during testing. Equipment acquisition is therefore deemed necessary in order to achieve a proper communication link as well as accurate data. The importance of a better position tracking system is especially remarked, as it is crucial for valid plot generation. An updated radio link protocol is also recommended to make full utilization of the available resources. Finally, upgrading of the described tool is encouraged. Expansion paths are identified for increased performance, plot quality and/or monitoring capabilities in different sections of the program architecture.

References

[1] Nils Olsen and Claudia Stolle. *Satellite Geomagnetism.* Article. DTU Space, 2012.

[2] A.P.J. Luyendyk. 'Processing of airborne magnetic data'. In: *AGSO Journal of Australian Geology & Geophysics* (1997).

[3] D.R. Cowan, M. Balgent and S. Cowan. 'Aeromagnetic Gradiometers - a perspective'. In: *Exploration Geophysics* (1995).

[4] Eirik Mauring and Ola Kihle. 'Leveling aerogeophysical data using a moving differential median filter'. In: *Geophysics* (2006).

[5] Python Software Foundation. *Python 3.7.3 documentation.* URL: `https://docs.python.org/3/` (visited on 02/06/2019).

[6] *Kivy documentation.* URL: `https://kivy.org/doc/stable/` (visited on 02/06/2019).

[7] Python Software Foundation. *Initialization, Finalization, and Threads.* URL: `https://docs.python.org/3/c-api/init.html#thread-state-and-the-global-interpreter-lock` (visited on 03/06/2019).

[8] Atul S. Khot. *Scala Functional Programming Patterns.* Ed. by Packt Publishing Ltd. 2015. Chap. 10, p. 235.

[9] Matplotlib development team. *Matplotlib 3.0.2 documentation.* URL: `https://matplotlib.org/3.0.2/contents.html` (visited on 03/06/2019).

[10] Community add-on. *Matplotlib backends using Kivy.* URL: `https://github.com/kivy-garden/garden.matplotlib` (visited on 04/06/2019).

[11] C. Caruso and F. Quarta. 'Interpolation Methods Comparison'. In: *Computers Math. Applic.* 35.12 (1998), pp. 109–126.

[12] SciPy developers. *Numpy and Scipy Documentation.* URL: `https://docs.scipy.org/doc/` (visited on 07/06/2019).

[13] Jakub Srna. *Wireless data transmission.* Technical University of Denmark (DTU), May 2019.

[14] Inc. GEM Systems. *GSMP-35U Instruction Manual.* 2016.

[15] Python Software Foundation. *Python Packaging User Guide. Installing Packages.* URL: `https://packaging.python.org/tutorials/installing-packages/` (visited on 11/06/2019).

[16] *Kivy Installation Guide.* URL: `https://kivy.org/doc/stable/gettingstarted/installation.html` (visited on 11/06/2019).

Appendix: Source code

The *LiveSensing* application is organized in the following directory structure:

```
LiveSensing/
├── libs/
├── input_files/
├── log/
├── media/
├── src/
│   ├── MagPro/
│   ├── data_gatherers.py
│   ├── interactive_plot.py
│   ├── main_screen.py
│   ├── processing.py
│   └── start_screen.py
├── main.py
└── readme.txt
```

All custom code components listed here are appended below. Code within the `libs` directory contains Kivy garden community add-ons, namely a kivy backend for matplotlib, and is therefore not included. It is distributed with the source code to avoid common installation errors as well as future unavailability. *MagPro* source code and utilities, located under the `MagPro` directory, are also not included and the documentation provided by its author should be consulted if necessary [13].

main.py

```
# -*- coding: utf-8 -*-

if __name__ == '__main__':
    # All kivy-related imports done inside the if __name__ to prevent parallel
                                                processes from creating new windows
    import sys
    sys.path.insert(0, './src')

    from kivy.config import Config
    from kivy.app import App
    from kivy.uix.screenmanager import ScreenManager

    from start_screen import StartScreen


    class MyApp(App):

        def build(self):
            self.icon = "media/icon.ico"
            self.title = "LiveSensing"
            sm = ScreenManager()
            sm.add_widget(StartScreen(name="start"))
            return sm


    # Three lines to ensure non-fullscreen, maximized window on launch
    Config.set('graphics', 'fullscreen', 0)
    Config.set('graphics', 'borderless', 0)
    Config.set('graphics', 'window_state', 'maximized')
```

```
    Config.write()

    app = MyApp()
    app.run()
```

start_screen.py

```
from kivy.core.window import Window
from kivy.uix.textinput import TextInput
from kivy.uix.boxlayout import BoxLayout
from kivy.uix.accordion import Accordion, AccordionItem
from kivy.uix.slider import Slider
from kivy.uix.filechooser import FileChooserIconView
from kivy.uix.label import Label
from kivy.uix.gridlayout import GridLayout
from kivy.uix.screenmanager import Screen
from kivy.uix.button import Button
from kivy.uix.widget import Widget

from main_screen import MainScreen


class StartScreen(Screen):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        Window.clearcolor = (.15, .15, .15, 1)  # Dark gray background

        layout = BoxLayout(orientation="vertical")

        inner_layout = GridLayout(cols=3)
        inner_layout.add_widget(Label(text="Interpolation", size_hint=(0.3, 0.
                                        09), font_size='20dp'))
        inner_layout.add_widget(Label(text="Survey type", size_hint=(0.3, 0.09)
                                        , font_size='20dp'))
        inner_layout.add_widget(Label(text="General", size_hint=(0.3, 0.09),
                                        font_size='20dp'))

        self.interp_accordion = InterpolationAccordion()
        self.mode_accordion = MeasurementModeAccordion()
        self.general_settings = GeneralSettingsLayout()

        inner_layout.add_widget(self.interp_accordion)
        inner_layout.add_widget(self.mode_accordion)
        inner_layout.add_widget(self.general_settings)

        continue_btn = Button(text="START", size_hint=(1, 0.07))
        continue_btn.bind(on_press=self.proceed_to_main)

        layout.add_widget(inner_layout)
        layout.add_widget(continue_btn)

        self.add_widget(layout)

    def proceed_to_main(self, *args):
        interp_params = self.interp_accordion.selected_params
        meas_params = self.mode_accordion.selected_params
        general_params = self.general_settings.params

        self.manager.add_widget(MainScreen(interp_params, meas_params,
                                        general_params, name="main"))
        self.manager.current = "main"


# -------------------- Interpolation -----------------
class InterpolationAccordion(Accordion):
    """
    Contains all the interpolation methods in an accordion fashion.
```

```
    To add a new one, create a new AccordionItem with the corresponding fields,
                                                  and a method to update.

    CAREFUL - Parameters to the interpolator (in processing.py) are passed with
                                                  the params dictionary key
    as a variable name. Dictionary keys must be chosen accordingly.
    """
    def __init__(self, **kwargs):
        super().__init__(orientation="vertical", **kwargs)
        self._selected = None  # AccordionItem representing selected
                                                  interpolation method

        lin = LinearOnlyItem()
        lin.bind(collapse=self.update_selected)
        self.add_widget(lin)

        id2 = InverseDistanceSquaredItem()
        id2.bind(collapse=self.update_selected)
        self.add_widget(id2)

    @property
    def selected_params(self):
        """
        Returns the dictionary containing the start parameters corresponding to
                                                  the selected interpolation
                                                  method.
        """
        return self._selected.params

    def update_selected(self, instance, collapse):
        """
        Keeps track of which on the available Accordion items is selected, so
                                                  the parameters for the
                                                  appropriate
        mode can be extracted when initializing MainScreen.
        :param instance: Accordion Item that called this method upon being
                                                  updated
        :param collapse: Boolean. Represents updated state of the AccordionItem
                                                  (False = selected)
        """
        if not collapse:  # To only have the newly uncollapsed (selected)
                                                  accordion item
            self._selected = instance


class InverseDistanceSquaredItem(AccordionItem):
    """
    Accordion Item to be place inside the InterpolationAccordion. Includes all
                                                  necessary fields,
    parameters and handling functions for an Inverse Distance Squared
                                                  interpolator.
    """

    def __init__(self, **kwargs):
        super().__init__(title="Inverse Squared Distance", **kwargs)

        min_window_size = 2
        max_window_size = 15
        init_window_size = 6
        min_cell_size = 0.4
        max_cell_size = 2
        init_cell_size = 0.8
        self.params = {"type": "inv_dist_squared"}
```

```
        layout = BoxLayout(orientation="vertical", padding=(10, 10))

        desc = "-Window (meters): Radius of affected interpolated points around
                                                    each sample\n\
-Cell size (meters): Separation between interpolated points around each sample.
                                            "
        desc_label = Label(text=desc, size_hint=(1, None), padding=(0, 10))
        # Over-complicated workaround to avoid using .kv file:
        desc_label.bind(
            width=lambda *x: desc_label.setter('text_size')(desc_label, (
                                                        desc_label.width, None)),
            texture_size=lambda *x: desc_label.setter('height')(desc_label,
                                                        desc_label.texture_size[1])
                                                        )

        layout_inner = GridLayout(cols=3)

        # Window size slider and labels
        layout_inner.add_widget(Label(text="Window", size_hint=(0.7, 1)))
        window_slider = Slider(min=min_window_size, max=max_window_size,
                                                    size_hint=(2, 1))
        window_slider.bind(value=self.on_change_window)
        layout_inner.add_widget(window_slider)
        self.window_value = Label(size_hint=(0.3, 1))  # Need to keep reference
                                                    for updating value label
        layout_inner.add_widget(self.window_value)

        window_slider.value = init_window_size

        # Cell size slider and labels
        layout_inner.add_widget(Label(text="Cell size", size_hint=(0.7, 1)))
        cell_slider = Slider(min=min_cell_size, max=max_cell_size, size_hint=(2
                                                    , 1))
        cell_slider.bind(value=self.on_change_cell)
        layout_inner.add_widget(cell_slider)
        self.cell_value = Label(size_hint=(0.3, 1))  # Need to keep reference
                                                    for updating value label
        layout_inner.add_widget(self.cell_value)

        cell_slider.value = init_cell_size

        layout.add_widget(desc_label)
        layout.add_widget(layout_inner)
        self.add_widget(layout)

    def on_change_window(self, instance, value):
        self.window_value.text = "{:04.1f}".format(value)
        self.params["dmax"] = value

    def on_change_cell(self, instance, value):
        self.cell_value.text = "{:04.2f}".format(value)
        self.params["step_meters"] = value


class LinearOnlyItem(AccordionItem):

    def __init__(self, **kwargs):
        super().__init__(title="Linear only", **kwargs)

        self.params = {"type": "linear_only"}

        layout = BoxLayout(orientation="vertical", padding=(10, 10))
```

```
        desc = "Linear Interpolation can be toggled in all other interpolation
                                                        modes, \
but the main selected mode is kept running in the background.\n\n\
Selecting this mode avoids additional computation in the background for
                                                increased performance."
        desc_label = Label(text=desc, padding=(0, 10))
        # Over-complicated workaround to avoid using .kv file:
        desc_label.bind(
            width=lambda *x: desc_label.setter('text_size')(desc_label, (
                                                        desc_label.width, None)),
            texture_size=lambda *x: desc_label.setter('height')(desc_label,
                                                        desc_label.texture_size[1])
                                                        )

        layout.add_widget(desc_label)
        self.add_widget(layout)


# ----------------------- Measurement mode -------------------
class MeasurementModeAccordion(Accordion):
    """
    -Contains all the measurement modes in an accordion fashion.
    -To add a new one, create a new AccordionItem with the corresponding fields
                                                , and a method to update each into
    the parameter dictionary. Bind that AccordionItem to the update_selected
                                                method like the existing ones.
    """
    def __init__(self, **kwargs):
        super().__init__(orientation="vertical", **kwargs)

        self._selected = None  # AccordionItem representing selected
                                                measurement mode

        single_sensor = SingleSensorItem()
        single_sensor.bind(collapse=self.update_selected)

        v_gradient = VerticalGradientItem()
        v_gradient.bind(collapse=self.update_selected)

        self.add_widget(single_sensor)
        self.add_widget(v_gradient)

    @property
    def selected_params(self):
        """
        Returns the dictionary containing the start parameters corresponding to
                                                the selected measurement mode.
        """
        return self._selected.params

    def update_selected(self, instance, collapse):
        """
        Keeps track of which on the available Accordion items is selected, so
                                                the parameters for the
                                                appropriate
        mode can be extracted when initializing MainScreen.
        :param instance: Accordion Item that called this method upon being
                                                updated
        :param collapse: Boolean. Represents updated state of the AccordionItem
                                                (False = selected)
        """
        if not collapse:
```

```
            self._selected = instance


class SingleSensorItem(AccordionItem):
    def __init__(self, **kwargs):
        super().__init__(title="Single sensor", **kwargs)
        self.params = {"type": "single_sensor"}

        layout = BoxLayout(orientation="vertical", padding=(10, 10))

        desc = "A single magnetometer is used, and the field values are drawn
                                        directly. This does not correct
                                        \
for time or spacial variations on the field, and those must be filtered."
        desc_label = Label(text=desc, size_hint=(1, None), padding=(0, 10))
        # Over-complicated workaround to avoid using .kv file:
        desc_label.bind(
            width=lambda *x: desc_label.setter('text_size')(desc_label, (
                                        desc_label.width, None)),
            texture_size=lambda *x: desc_label.setter('height')(desc_label,
                                        desc_label.texture_size[1])
                                        )

        separator_widget = Widget()

        layout.add_widget(desc_label)
        layout.add_widget(separator_widget)
        self.add_widget(layout)


class VerticalGradientItem(AccordionItem):
    def __init__(self, **kwargs):
        super().__init__(title="Vertical gradient", **kwargs)
        self.params = {"type": "vertical_gradient"}
        min_separation = 0.10
        max_separation = 3
        default_separation = 1

        layout = BoxLayout(orientation="vertical", padding=(10, 10))

        desc = "Two magnetometers are used to calculate the vertical gradient
                                        of the magnetic field at each
                                        point. \
This filters field time variations and other anomalies. The gradient will be
                                directly plotted.\n\n \
-Separation (meters): Vertical distance between the two magnetometers."
        desc_label = Label(text=desc, size_hint=(1, None), padding=(0, 10))
        # Over-complicated workaround to avoid using .kv file:
        desc_label.bind(
            width=lambda *x: desc_label.setter('text_size')(desc_label, (
                                        desc_label.width, None)),
            texture_size=lambda *x: desc_label.setter('height')(desc_label,
                                        desc_label.texture_size[1])
                                        )

        layout_inner = GridLayout(cols=3)

        layout_inner.add_widget(Label(text="Separation", size_hint=(0.7, 1)))
        separation_slider = Slider(min=min_separation, max=max_separation,
                                        size_hint=(2, 1))
        separation_slider.bind(value=self.on_change_separation)
        layout_inner.add_widget(separation_slider)
```

```
        self.separation_value = Label(size_hint=(0.3, 1))  # Need to keep
                                                            reference for updating
        layout_inner.add_widget(self.separation_value)

        separation_slider.value = default_separation
        layout.add_widget(desc_label)
        layout.add_widget(layout_inner)
        self.add_widget(layout)

    def on_change_separation(self, instance, value):
        self.separation_value.text = "{:05.2f}".format(value)
        self.params["spacing"] = value


# ----------------------- Initial position information ---------
class PositionAccordion(Accordion):
    """
    -Contains all the methods for entering a-priori location information.
    -To add a new one, create a new AccordionItem with the corresponding fields
                                            , and a method to update each into
    the parameter dictionary. Bind that AccordionItem to the update_selected
                                            method like the existing ones.
    """
    def __init__(self, **kwargs):
        super().__init__(orientation="vertical", **kwargs)

        self._selected = None  # AccordionItem representing selected a-priori
                                                            location info

        # from_file = FromFileItem()
        # from_file.bind(collapse=self.update_selected)
        # self.add_widget(from_file)

        manual = ManualItem()
        manual.bind(collapse=self.update_selected)
        self.add_widget(manual)

        no_info = NoInfoItem()
        no_info.bind(collapse=self.update_selected)
        self.add_widget(no_info)

    @property
    def selected_params(self):
        """
        Returns the dictionary containing the start parameters corresponding to
                                                            the selected a-priori position
                                                            info.
        """
        return self._selected.params

    def update_selected(self, instance, collapse):
        """
        Keeps track of which on the available Accordion items is selected, so
                                                            the parameters for the
                                                            appropriate
        mode can be extracted when initializing MainScreen.
        :param instance: Accordion Item that called this method upon being
                                                            updated
        :param collapse: Boolean. Represents updated state of the AccordionItem
                                                            (False = selected)
        """
        if not collapse:
            self._selected = instance
```

```
class NoInfoItem(AccordionItem):
    def __init__(self, **kwargs):
        super().__init__(title="No a-priori info", **kwargs)

        self.params = {"type": "no_info"}

        self.add_widget(Label(text="First sample will be\nused as starting
                                        point"))


class ManualItem(AccordionItem):
    def __init__(self, **kwargs):
        super().__init__(title="Manual selection", **kwargs)

        self.params = {"type": "manual"}

        self.add_widget(TextInput(text="Latitude", multiline=False, size_hint=(
                                        1, 0.2)))
        self.add_widget(TextInput(text="Longitude", multiline=False, size_hint=
                                        (1, 0.2)))


class GeneralSettingsLayout(BoxLayout):
    def __init__(self, **kwargs):
        super().__init__(orientation="vertical", padding=(10, 10), **kwargs)
        self.params = {}
        desc = "-Plot step: number of processed samples for each plot update.
                                        Set high at low-end systems or
                                        high \
sample rates for better performance."
        desc_label = Label(text=desc, size_hint=(1, None), padding=(0, 10))
        # Over-complicated workaround to avoid using .kv file:
        desc_label.bind(
            width=lambda *x: desc_label.setter('text_size')(desc_label, (
                                            desc_label.width, None)),
            texture_size=lambda *x: desc_label.setter('height')(desc_label,
                                            desc_label.texture_size[1])
                                            )
        self.add_widget(desc_label)

        layout_inner = GridLayout(cols=3)

        # Plot step slider and labels
        layout_inner.add_widget(Label(text="Plot step", size_hint=(0.7, 1)))
        step_slider = Slider(min=1, max=10, size_hint=(2, 1), step=1)
        step_slider.bind(value=self.on_change_step)
        layout_inner.add_widget(step_slider)
        self.step_value = Label(size_hint=(0.3, 1))  # Need to keep reference
                                        for updating
        layout_inner.add_widget(self.step_value)
        step_slider.value = 2

        self.add_widget(layout_inner)

    def on_change_step(self, instance, value):
        self.step_value.text = str(value)
        self.params["step"] = value
```

main_screen.py

```
from matplotlib.figure import Figure
from mpl_toolkits.axes_grid1.axes_divider import make_axes_locatable

from kivy.garden.matplotlib.backend_kivyagg import FigureCanvasKivyAgg
from kivy.clock import Clock
from kivy.uix.boxlayout import BoxLayout
from kivy.uix.gridlayout import GridLayout
from kivy.uix.label import Label
from kivy.uix.screenmanager import Screen
from kivy.uix.button import Button
from kivy.uix.togglebutton import ToggleButton
from kivy.uix.widget import Widget
from kivy.graphics import Rectangle, Color

from threading import Thread, Lock
from multiprocessing import Process, Pipe

from data_gatherers import FifoDataGatherer as DataGatherer  # FileDataGatherer
                                        , FifoDataGatherer, TestDataGatherer
from processing import DataProcessing
from interactive_plot import InteractivePlotter

import logging


class MainScreen(Screen):
    def __init__(self, interp_params, meas_params, general_params, **kwargs):
        super().__init__(**kwargs)

        meas_mode = meas_params["type"]  # The graph title must show either
                                            magnetic gradient or magnetic
                                            field
        if meas_mode == "vertical_gradient":
            self.graph = Graph(graph_title="Magnetic gradient (nT/m)")
        elif meas_mode == "single_sensor":
            self.graph = Graph(graph_title="Magnetic field (nT)")
        else:  # Should be unreachable if properly designed
            self.graph = Graph(graph_title="Unspecified")

        self.gatherer_pipe, ext_gatherer_pipe = Pipe()  # Local and external
                                            pipe ends are created, local is
                                             stored
        self.processing_pipe, ext_processing_pipe = Pipe()

        # Control UI components that will require updating ----------
        self.battery = 0
        self.battery_label = Label(text=str(self.battery)+"V", size_hint_x=0.3)
        self.packet_counter = 0
        self.counter_label = Label(text=str(self.packet_counter))
        # -------------------------------
        self.ack_popup = CustomPopup(msg="Command received.", pos_hint={"
                                            center_x": 0.5, "center_y": 0.5
                                            })
        self.error_popup = CustomPopup(msg="Error.", pos_hint={"center_x": 0.5,
                                             "center_y": 0.5})
        self.showing_popup = False  # Prevents stacking popups when multiple
                                            ACKs are received in quick
                                            succession
        self.showing_error = False
        self.ack_trigger = Clock.create_trigger(self.show_ack_popup)  # Used by
                                             any thread to show popup in
```

```
                                        the main one
        self.error_trigger = Clock.create_trigger(self.show_error_popup)

        # Data Gatherer and Processing processes launched, with threads in the
                                        main process as listeners ---
        gatherer_process = Process(target=DataGatherer,
                                   args=(ext_gatherer_pipe,),
                                   daemon=True)
        processing_process = Process(target=DataProcessing,
                                     args=(ext_processing_pipe, interp_params,
                                                                           meas_params
                                                                           ,
                                                                           general_params
                                                                           .
                                                                           pop
                                                                           ("
                                                                           step
                                                                           ")
                                                                           ),
                                     daemon=True)
        gatherer_process.start()
        processing_process.start()

        # Threads for handling concurrent new samples and processed data flows
                                        respectively
        Thread(target=self._parallel_gatherer_interface, daemon=True).start()
        Thread(target=self._parallel_processing_interface, daemon=True).start()
        # --------------------------------

        linear_only = (interp_params["type"] == "linear_only")  # Describes if
                                        custom interpolation is
                                        selected

        self.generate_screen_elements(linear_only)

    def generate_screen_elements(self, linear_only):
        """
        Defines how the graphical layout is built and structured
        ;param linear_only; Boolean. If true, only linear interpolation is
                                        available and no toggle button
                                        is needed
        """
        layout = BoxLayout(orientation="horizontal", spacing=10)
        inner_layout = BoxLayout(orientation="vertical", size_hint=(.2, 1),
                                        padding=[10, 10], spacing=6)

        # Gatherer controls ---------------
        gatherer_layout = BoxLayout(orientation="horizontal", padding=[0, 10],
                                        spacing=1, size_hint_y=0.5)

        gather_start_btn = Button(text="[size=11]START[/size]", markup=True)
        gather_start_btn.cmd = "log"
        gather_start_btn.bind(on_press=self._send_gatherer_cmd)
        gatherer_layout.add_widget(gather_start_btn)

        gather_pause_btn = Button(text="[size=11]PAUSE[/size]", markup=True)
        gather_pause_btn.cmd = "pause"
        gather_pause_btn.bind(on_press=self._send_gatherer_cmd)
        gatherer_layout.add_widget(gather_pause_btn)

        gather_end_btn = Button(text="[size=11]END[/size]", markup=True)
        gather_end_btn.cmd = "stop"
        gather_end_btn.bind(on_press=self._send_gatherer_cmd)
```

```
        gatherer_layout.add_widget(gather_end_btn)
        # End of gatherer controls ------

        inner_layout.add_widget(gatherer_layout)

        plot_pause_btn = ToggleButton(text="Pause plotting")
        plot_pause_btn.bind(on_press=self.graph.toggle_updating_figure)
        inner_layout.add_widget(plot_pause_btn)

        trace_toggle_btn = ToggleButton(text="Toggle trace", state="down")
        trace_toggle_btn.bind(on_press=self.graph.toggle_showing_trace)
        inner_layout.add_widget(trace_toggle_btn)

        if linear_only:  # No need for tricontour toggling, by default only
                                        plotting tricontours
            self.graph._tricontour_mode = True
        else:  # Tricontour toggle button is created
            tricontour_toggle_btn = ToggleButton(text="Toggle linear \n
                                            interpolation")
            tricontour_toggle_btn.bind(on_press=self.graph.
                                            toggle_tricontour_use)
            inner_layout.add_widget(tricontour_toggle_btn)

        external_plot_btn = Button(text="Interactive plot")
        external_plot_btn.bind(on_press=self.graph.interactive_plot)
        inner_layout.add_widget(external_plot_btn)

        # ----- Bottom control information ------
        control_layout = GridLayout(cols=2, padding=(10, 0), size_hint_y=None,
                                        spacing=[10, 0])

        counter_desc = Label(text="Samples: ")
        control_layout.add_widget(counter_desc)
        self.counter_label.size_hint_x = 0.3
        control_layout.add_widget(self.counter_label)

        control_layout.add_widget(self.battery_label)

        control_layout.height = len(control_layout.children)*15  # Forced
                                        height for proper display of
                                        all info

        inner_layout.add_widget(control_layout)
        # ----- End of bottom control -----

        layout.add_widget(inner_layout)
        layout.add_widget(self.graph.canvas)

        Clock.schedule_interval(self._update_control_info, 1)

        self.add_widget(layout)

    def _send_gatherer_cmd(self, instance, *args):
        """
        To be called through the gatherer control buttons.
        Sends the command to the gatherer associated with said button.
        :param instance: button that called this method
        """
        logging.info("[Main] Sending command - " + instance.cmd)
        try:
            self.gatherer_pipe.send(instance.cmd)
        except BrokenPipeError:
            self.error_trigger()
```

```
    def _parallel_gatherer_interface(self):
        """
        To be called on a different Thread.
        Handles the connection from the DataGatherer and towards the Processing
                                                        .
        Checks each data packet, pops information not relevant to the
                                                        processing side, and forwards
                                                        the rest
        """

        packet = self.gatherer_pipe.recv()  # Initial non-empty packet to enter
                                                         the loop

        while packet:  # Will exit if None or {} is received
            packet_type = packet.pop("type")
            if packet_type == "sample":
                self.battery = packet.pop("battery")
                self.processing_pipe.send(packet)
                self.packet_counter += 1

            elif packet_type == "ACK":
                self.ack_trigger()  # ACK Popup is shown

            packet = self.gatherer_pipe.recv()

        self.processing_pipe.send(None)  # None as a poison pill

    def _parallel_processing_interface(self):
        """
        To be called on a different Thread
        Handles the connection from the Processing.
        Calls the corresponding update method each time new data is received.
        """
        packet = self.processing_pipe.recv()

        while packet:
            packet_type = packet.pop("type")
            if packet_type == "raw":  # Length 3 for new trace plots (x_coord,
                                                             y_coord, field_vect)
                self.graph.new_sample(packet["long"], packet["lat"], packet["
                                                                 data"])
            elif packet_type == "interp":
                self.graph.new_interpolated(packet["long"], packet["lat"],
                                                                 packet["data"])

            packet = self.processing_pipe.recv()

    def _update_control_info(self, *args):
        """
        Redraws the control information (sample counter and battery voltage).
        To be called in main thread.
        """
        self.counter_label.text = str(self.packet_counter)
        self.battery_label.text = str(self.battery) + "V"

    def show_ack_popup(self, *args):
        """
        Draws the ACK popup and schedules it to be deleted in 1 second.
        To prevent stacking of multiple popups when called in quick succession,
                                                         it only draws it if not
                                                        already drawn.
        To be called from the main thread or through ack_trigger.
```

```
        """
        if not self.showing_popup:
            self.showing_popup = True
            self.add_widget(self.ack_popup)
            Clock.schedule_once(self.remove_ack_popup, 1)

    def remove_ack_popup(self, *args):
        """
        Removes the ACK popup from the screen and sets the corresponding check
                                        variable to False.
        """
        self.remove_widget(self.ack_popup)
        self.showing_popup = False

    def show_error_popup(self, *args):
        """
        Draws the error popup and schedules it to be deleted in 1 second.
        To prevent stacking of multiple popups when called in quick succession,
                                         it only draws it if not
                                        already drawn.
        To be called from the main thread or through error_trigger.
        """
        if not self.showing_error:
            self.showing_error = True
            self.add_widget(self.error_popup)
            Clock.schedule_once(self.remove_error_popup, 1)

    def remove_error_popup(self, *args):
        """
        Removes the error popup from the screen and sets the corresponding
                                        check variable to False.
        """
        self.remove_widget(self.error_popup)
        self.showing_error = False


class CustomPopup(Widget):
    def __init__(self, msg, **kwargs):
        super().__init__(**kwargs)
        self.size_hint = (None, None)
        self.size = (200, 75)

        # Background is set as a rectangle
        with self.canvas:
            Color(.7, .15, .15, 1)
            self.background = Rectangle(pos=self.center, size=(self.width, self
                                            .height))

        self.label = Label(text=msg, pos=self.center, size=(self.width, self.
                                        height))
        self.add_widget(self.label)

        # Dynamic binding necessary to update rectangle and label size/pos when
                                         parent size/pos changes
        self.bind(pos=self.redraw, size=self.redraw)

    def redraw(self, *args):
        """
        Updates children position and size whenever the parent's position or
                                        size changes.
        Alternative to .kv language.
        """
        self.background.size = self.size
```

```
        self.background.pos = self.pos
        self.label.size = self.size
        self.label.pos = self.pos


class Graph:
    """
    Represents a contour (heatmap) + trace plot.
    Keeps track of both plots themselves as well as direct references to the
                                        information they show.
    Implements all the necessary logic to toggle between the available plotting
                                        modes, and limits the number
    of plot attempts through triggers.
    """

    def __init__(self, graph_title):
        """
        Sets the axes and internal variables necessary for operation.
        :param graph_title: Graph title may vary depending on interpolation
                                        mode, so is left as a variable
        """
        self.figure = Figure()
        self.canvas = FigureCanvasKivyAgg(self.figure)
        self.ax = self.figure.add_subplot(111)
        self.color_ax = make_axes_locatable(self.ax).append_axes("right", size=
                                        "5%", pad="2%")  # For colorbar

        self.figure.suptitle(graph_title)

        # Triggers used to notify main thread of new sample/interpolated values
        self.update_xyz_trigger = Clock.create_trigger(self._update_xyz_plot)
        self.update_contour_trigger = Clock.create_trigger(self.
                                        _update_contour_plot)

        self.ax.set_xlabel("Longitude")
        self.ax.set_ylabel("Latitude")

        # xyz_coord variables store the currently plotted values. May differ
                                        from the new_ ones if plotting
                                        is paused
        self.xx = []
        self.yy = []
        self.zz = []
        self.x_coord = []  # xyz_coord store raw data values: GPS coordinates
                                        and field values
        self.y_coord = []
        self.z_coord = []

        # new_XXX variables store the most updated vector of values. Can be
                                        updated from any thread (thread
                                        -safe)
        self.new_x_coord = []
        self.new_y_coord = []
        self.new_z_coord = []
        self.new_xx = []  # xx, yy, zz: interpolated vectors (xx/yy) and matrix
                                         (zz) of values
        self.new_yy = []
        self.new_zz = []

        # Locks to handle multithreaded access to new_XX and new_X_coord
                                        variables
        self.coord_lock = Lock()
        self.contour_lock = Lock()
```

```
        self.flight_plot, = self.ax.plot(self.x_coord, self.y_coord, 'kx-',
                                        markersize=1.5, linewidth=0.5)
        self.contour_plot = None

        # Operating configuration
        self._showing_trace = True  # Whether or not the flight trace is
                                        plotted at all
        self._showing_contour = True  # Whether or not the heatmap is plotted
                                        at all
        self._tricontour_mode = False  # (In case heatmap is plotted) whether
                                        it's a tricontour or normal
                                        contour
        self._updating_figure = True  # For pausing graph updates

        self.canvas.draw()

    def new_interpolated(self, xx, yy, zz):
        """
        THREAD SAFE
        Updates the latest stored interpolation data.
        Calls the plot redraw trigger to handle the new data as appropriate in
                                        main thread.
        """
        with self.contour_lock:
            self.new_xx = xx
            self.new_yy = yy
            self.new_zz = zz
        self.update_contour_trigger()

    def new_sample(self, new_x, new_y, new_z):
        """
        THREAD SAFE
        Takes three (1D) arrays and updates the latest stored coordinate data.
        Calls the plot redraw trigger to update the graph.
        """
        with self.coord_lock:
            self.new_x_coord = new_x
            self.new_y_coord = new_y
            self.new_z_coord = new_z

        self.update_xyz_trigger()  # The trigger is called to execute
                                        _handle_new_sample() in main
                                        thread

    def _update_xyz_plot(self, *args):
        """
        ONLY CALLED INTERNALLY THROUGH update_xyz_trigger.
        Executes on main thread.
        Checks whether or not to update existing plotted values.
        Executed only once in the next Kivy cycle, even if called multiple
                                        times (if called through
                                        trigger)

        :param args: This method is called by an Event trigger, which sends
                                        additional parameters.
        """
        if self._updating_figure:
            self._update_stored_coord()
            self._plot_redraw()

    def _update_contour_plot(self, *args):
        """
```

```
        ONLY CALLED INTERNALLY THROUGH update_contour_trigger.
        Executes on main thread.
        Checks whether or not to update existing plotted contour values.
        Executed only once in the next Kivy cycle, even if called multiple
                                        times (if called through
                                        trigger)

        :param args: This method is called by an Event trigger, which sends
                                        additional parameters.
        """
        if self._updating_figure:
            self._update_stored_contour()
            self._contour_redraw()

    def _update_stored_coord(self):
        """
        Updates stored (old) values for raw sample data values with the latest
                                        ones
        """
        with self.coord_lock:
            self.x_coord = self.new_x_coord
            self.y_coord = self.new_y_coord
            self.z_coord = self.new_z_coord

    def _update_stored_contour(self):
        """
        Updates stored (old) values for interpolated vectors and matrix with
                                        the latest ones
        """
        with self.contour_lock:
            self.xx = self.new_xx
            self.yy = self.new_yy
            self.zz = self.new_zz

    def toggle_tricontour_use(self, *args):
        """
        Used to toggle between tricontour and non-tricontour plotting.

        :param args: This method is called by a button, which sends additional
                                        parameters.
        """
        logging.info("Toggling tricontour")
        if self._tricontour_mode:
            # When tricontour plotting turned off, the contour is erased and
                                        contour_plot goes back to
                                        None
            self._tricontour_mode = False
            self._remove_contour()
            if self._showing_contour:  # If tricontour is turned off and
                                        heatmap is being shown,
                                        plot normal contour
                self._contour_redraw()
        else:
            self._tricontour_mode = True

        self._plot_redraw()  # Graph redrawn after toggling, no need to wait
                                        for next sample for an update

    def toggle_updating_figure(self, *args):
        """
        Toggles internal parameter _updating_figure, and updates the figure if
                                        turned on
```

```
        :param args: This method is called by a button, which sends additional
                                                parameters.
        """
        if self._updating_figure:
            self._updating_figure = False
        else:
            self._updating_figure = True
            self.update_xyz_trigger()
            self.update_contour_trigger()

    def toggle_showing_trace(self, *args):
        """
        Toggles internal parameter _showing_trace, and updates the figure
                                                accordingly.
        If it is turned off, the flight plot must be manually modified to not
                                                represent anything.

        :param args: This method is called by a button, which sends additional
                                                parameters.
        """
        if self._showing_trace:
            self.flight_plot.set_data([], [])
            self._showing_trace = False
        else:
            self._showing_trace = True

        self._plot_redraw()

    def _remove_contour(self):
        """
        Removes all contour components from the graph, since the whole contour
                                                cannot be removed in one single
            action, like a normal plot would.
        When a contour is removed, the stored value for it goes back to None.
                                                This allows the prevention of
                                                removing
            a not-existing contour (previously removed)
        """
        if self.contour_plot is not None:
            for col in self.contour_plot.collections:
                col.remove()
            self.color_ax.clear()
            self.contour_plot = None

    def _plot_redraw(self):
        """
        Updates all (selected) plots that depend on the stored raw data vectors
                                                 (x_coord, y_coord, z_coord).
        Namely, updates the flight trace and tricontour plots (if possible).
        """
        if self._showing_trace:
            self.flight_plot.set_data(self.x_coord, self.y_coord)

        if self._showing_contour and self._tricontour_mode:
            if len(self.z_coord) > 4:
                self._remove_contour()
                self.contour_plot = self.ax.tricontourf(self.x_coord, self.
                                                        y_coord, self.z_coord,
                                                        levels=30)
                self.figure.colorbar(self.contour_plot, cax=self.color_ax)

        self.canvas.draw()
```

```
    def _contour_redraw(self):
        """
        Checks and redraws the heatmap if appropriate, removing the previous
                                        one.
        Only applicable for standard interpolation plots (not linear tricontour
                                        , which is done in _plot_redraw
                                        )
        Updates the drawn figure.
        """
        if self._showing_contour and not self._tricontour_mode:
            self._remove_contour()
            if len(self.zz) > 4:
                self.contour_plot = self.ax.contourf(self.xx, self.yy, self.zz,
                                                levels=100)
                self.figure.colorbar(self.contour_plot, cax=self.color_ax)

        self.canvas.draw()

    def interactive_plot(self, *args):
        """
        Gives current plot conditions and parameters to an external process
                                        which launches the default
                                        matplotlib
        interactive plot. Ensures no multithreading inconsistencies on given
                                        parameters through locks.
        """
        with self.coord_lock, self.contour_lock:
            if len(self.x_coord) < 4:
                return  # Prevents plotting attempts when very few samples are
                                                still available
            trace_data = (self.x_coord, self.y_coord)

            if self._tricontour_mode:
                contour_data = self.z_coord
            elif len(self.zz) < 4:
                return  # Prevents plotting attempts when interpolated matrix
                                                is not ready
            else:
                contour_data = (self.xx, self.yy, self.zz)

            p = Process(target=InteractivePlotter,
                        args=(self._showing_trace, trace_data, self.
                                                        _tricontour_mode
                                                        , contour_data)
                                                        ,
                        daemon=True)
            p.start()
```

interactive_plot.py

```
import matplotlib.pyplot as plt


class InteractivePlotter:
    """
    Generates an interactive plot using the default matplotlib backend.
    Since the plot blocks execution, this must be launched in a parallel thread
                                                    or (preferably) process
    """
    def __init__(self, showing_trace, trace_data, linear_mode, contour_data):
        if showing_trace:
            plt.plot(trace_data[0], trace_data[1], 'kx-', markersize=0.5,
                                                    linewidth=0.5)
        if linear_mode:
            plt.tricontourf(trace_data[0], trace_data[1], contour_data, levels=
                                                    30)
        else:
            plt.contourf(contour_data[0], contour_data[1], contour_data[2],
                                                    levels=30)
        plt.colorbar()
        plt.xlabel('Longitude (deg)')
        plt.ylabel('Latitude (deg)')
        # plt.title('Magnetic gradient (nT/m)')
        plt.show()
```

processing.py

```
# -*- coding: utf-8 -*-

import numpy as np
from threading import Lock
import time  # For efficiency testing


class DataProcessing:
    """
    Processing core. Takes in new data samples, outputs computed data in the
                                        form of ready-to-plot 2-D matrices
    (for the heatmap) and vectors containing the updated flight trace.
    """
    def __init__(self, pipe, interp_params, meas_params, plot_step):
        """
        :param pipe: Connects with the application core. Used for receiving
                                            samples and sending computed
                                            data
        :param interp_params: Parameters strictly for interpolator
                                            initialization
        :param meas_params: Form of measuring the field (e.g. v_gradient,
                                            h_gradient, direct...)
        :param plot_step: Every how many samples the processed data is sent
        """
        self._SEND_PACKET_INTERVAL = plot_step  # # of processed packets
                                            between each plot update
        self.unsent_packets = 0  # Keeps track of the interval counter - send 1
                                             out of X valid packets
        self.processing_pipe = pipe
        self.pipe_lock = Lock()
        self.raw_data = RawDataManager()  # Where sample field/coord values are
                                             stored/handled

        self.interp = None

        meas_type = meas_params["type"]

        if meas_type == "vertical_gradient":  # field_calc refers to the
                                            appropriate field calculation
                                            method
            self.V_SPACING = meas_params["spacing"]
            self.field_calc = self._v_grad_calc
        elif meas_type == "single_sensor":
            self.field_calc = self._single_magn_calc
        else:
            raise UnsupportedMeasModeException

        self.packet_collection(interp_params)

    def packet_collection(self, interp_params):
        """
        Main Data Processing loop.
        Collects incoming samples from the app_core.
        Checks integrity bit, and passes the previous one to the interpolator
                                            and vector builder.
        Packets are processed if they and the ones immediately before/after
                                            didn't have integrity problems.
        """

        # First packet is received for initial point, in case it's necessary
                                            for initializing interpolator
```

```
        prev_packet = self.processing_pipe.recv()
        interp_params["point_zero"] = (prev_packet["longitude"], prev_packet["
                                        latitude"])

        interp_type = interp_params.pop("type")
        if interp_type == "inv_dist_squared":
            self.interp = InverseDistanceInterpolator(pipe=self.processing_pipe
                                        , **interp_params)
        elif interp_type == "linear_only":
            self.interp = DummyInterpolator(pipe=self.processing_pipe, **
                                        interp_params)
        else:
            raise UnsupportedInterpolationException

        # There should be many packets in a survey - "None" in the first two
                                        not considered

        current_packet = self.processing_pipe.recv()

        prev_integrity = prev_packet.pop("integrity_lock")
        current_integrity = current_packet.pop("integrity_lock")

        if prev_integrity and current_integrity:  # First packet needs special
                                        treatment - not surrounded by
                                        other 2
            self.process_packet(prev_packet)

        next_packet = self.processing_pipe.recv()

        while next_packet:  # Breaks when None is received
            next_integrity = next_packet.pop("integrity_lock")
            # Packet is processed if both it and its immediate neighbours have
                                        proper integrity lock
            if prev_integrity and current_integrity and next_integrity:
                self.process_packet(current_packet)
            else:
                # If one of the three has problems, integrity variables must
                                        cycle (not all True)
                prev_integrity, current_integrity = current_integrity,
                                        next_integrity
            # No need to keep track of prev_packet (already sent/discarded),
                                        only of its integrity state
            current_packet = next_packet
            next_packet = self.processing_pipe.recv()

        # Last packet before the None also needs special treatment
        self.unsent_packets = self._SEND_PACKET_INTERVAL  # Hack to force
                                        sending the last samples
        if prev_integrity and current_integrity:
            self.process_packet(current_packet)

        # Once a None packet is received, it is sent back as poison pill for
                                        the
                                        _parallel_processing_interface
                                        () thread
        self.send(None)

    def process_packet(self, packet):
        """
        Performs all necessary operations to a valid packet:
        "Field" value calculation, appending to raw data vectors, feeding to
                                        interpolator and sending if
                                        appropriate.
```

```
        :param packet: Input, pre-processed packet (dictionary)
        """
        self.field_calc(packet)
        self.raw_data.new_sample(packet)
        self.interp.new_sample(packet["longitude"], packet["latitude"], packet[
                                                        "field"])
        self.unsent_packets += 1
        if self.unsent_packets == self._SEND_PACKET_INTERVAL:
            self.unsent_packets = 0
            self.send(self.raw_data.get_stored_data())
            self.interp.send()

    def send(self, data_vect):
        """
        Sends stored sample data vectors
        A lock is needed to account for interpolators that do multithreaded use
                                                        of this pipe.
        :param data_vect: Packet containing accumulated sample values: {x_vect,
                                                        y_vect, z_vect}
        """
        with self.pipe_lock:
            self.processing_pipe.send(data_vect)

    def _v_grad_calc(self, packet):
        """
        To be used through the field_calc method if necessary.
        Modifies the packet to take two different magnetic field readings and
                                                        translate them to a single "
                                                        field" value
        that represents the vertical gradient of the field.
        :param packet: Packet containing lower and upper magnetometer readings
        """
        packet["field"] = -(packet.pop("reading_2") - packet.pop("reading_1"))
                                                        / self.V_SPACING

    def _single_magn_calc(self, packet):
        """
        To be used through the field_calc method if necessary.
        Modifies the packet to change the "reading_1" value for a "field" value
                                                        .
        Trivial method, used to have a shared API with more complex, multi-
                                                        sensor field calculations.
        :param packet: Packet containing lower and upper magnetometer readings
        """
        packet["field"] = packet.pop("reading_1")


class RawDataManager:
    """
    Receives new single samples and efficiently puts them into an array.
                                                        Provides a simple API for accessing
    such an array where normally only a part of it contains valuable
                                                        information. This way there is no
                                                        need
    to take indexing into consideration when the stored useful data is to be
                                                        accessed.
    """
    def __init__(self):
        self.capacity = 200
        self.elements = 0
        self.longitude_vect = np.zeros(self.capacity)
        self.latitude_vect = np.zeros(self.capacity)
        self.field_vect = np.zeros(self.capacity)
```

```
    def new_sample(self, packet):
        """
        Takes in packets and efficiently adds their value to a dynamic array
        :param packet: pre-processed packet where magnetometer readings have
                                        been translated to a field
                                        value
        """
        if self.elements == self.capacity:  # Buffer resizing in case limit is
                                        reached
            self.capacity *= 2

            new_longitude_vect = np.zeros(self.capacity)
            new_latitude_vect = np.zeros(self.capacity)
            new_field_vect = np.zeros(self.capacity)

            new_longitude_vect[:self.elements] = self.longitude_vect
            new_latitude_vect[:self.elements] = self.latitude_vect
            new_field_vect[:self.elements] = self.field_vect

            self.longitude_vect = new_longitude_vect
            self.latitude_vect = new_latitude_vect
            self.field_vect = new_field_vect

        self.longitude_vect[self.elements] = packet["longitude"]
        self.latitude_vect[self.elements] = packet["latitude"]
        self.field_vect[self.elements] = packet["field"]
        self.elements += 1

    def get_stored_data(self):
        """
        Returns a tuple containing all samples so far
        :return: (longitude, latitude, field) where each is a vector with an
                                        element per sample
        """
        return {"type": "raw",
                "long": self.longitude_vect[:self.elements],
                "lat": self.latitude_vect[:self.elements],
                "data": self.field_vect[:self.elements]}


class InverseDistanceInterpolator:
    def __init__(self, pipe, point_zero, step_meters=0.1, dmin=0.01, dmax=8):
        """
        :param pipe: Pipe through which the interpolated result is sent
        :param step_meters: Grid cell spacing
        :param point_zero: Initial point for the interpolation: (longitude,
                                        latitude), due to similarity
                                        with (x, y)
        notation, in decimal degrees
        """

        self.pipe_out = pipe

        # Constants for distance calculation ---------
        earth_radius = 6371000
        expand_meters = 15  # Distance (m) to be added to one side when the
                                        matrix is expanded

        self._D_MIN = dmin  # Max/min distance for weight function (meters)
        self._D_MAX = dmax
        self._WINDOW_SIZE = int(2*self._D_MAX/step_meters + 2)
```

```
        self._LAT_STEP = 180/np.pi*step_meters/earth_radius  # Separation in latitude between contiguous grid points
        self._LONG_STEP = self._LAT_STEP/np.cos(point_zero[1]*np.pi/180)
        self._EXPAND_SIZE = int(expand_meters/step_meters)  # number of rows/columns to be added each time

        # Auxiliary for boundary check: Prevent new samples being within current range but not whole window around them
        self._D_MAX_LAT = (self._D_MAX/step_meters + 2)*self._LAT_STEP  # Note: +2 to have 2 extra _LAT_STEP of margin
        self._D_MAX_LONG = (self._D_MAX/step_meters + 2)*self._LONG_STEP

        self.interp_matrix = np.zeros((2*self._EXPAND_SIZE + 1, 2*self._EXPAND_SIZE + 1))  # Zero-filled
        self.accumulated_D_matrix = np.zeros((2*self._EXPAND_SIZE + 1, 2*self._EXPAND_SIZE + 1))

        # interp_long/lat, mesh_long/lat: stored for efficiency
        # np.arange does not include the last (stop) element. size+0.5 used to ensure it does include all
        self.interp_long = np.arange(start=(point_zero[0] - self._EXPAND_SIZE*self._LONG_STEP),
                                     stop=point_zero[0] + (self._EXPAND_SIZE+0.5)*self._LONG_STEP,
                                     step=self._LONG_STEP)
        self.interp_lat = np.arange(start=(point_zero[1] - self._EXPAND_SIZE*self._LAT_STEP),
                                    stop=point_zero[1] + (self._EXPAND_SIZE+0.5)*self._LAT_STEP,
                                    step=self._LAT_STEP)
        # Meshgrid from references to originals (no copying for speed)
        self.mesh_long, self.mesh_lat = np.meshgrid(self.interp_long, self.interp_lat, copy=False)

        (self.lat_min, self.lat_max) = (self.interp_lat[0], self.interp_lat[-1])
        (self.long_min, self.long_max) = (self.interp_long[0], self.interp_long[-1])
        """
        # Debugging -------------------
        print("Expand size (elements): "+str(self._EXPAND_SIZE)+" - Window size (elements): "+str(self._WINDOW_SIZE))
        print("LONG_STEP: "+str(self._LONG_STEP)+" - LAT_STEP: "+str(self._LAT_STEP))
        print("Initial interp_long: "+str(self.interp_long.shape)+" - Initial interp_lat: "+str(self.interp_lat.shape))
        print("Point_zero: "+str(point_zero))
```

```
        print("Long limits: "+str((self.long_min, self.long_max))+" - Lat
                                        limits: "+str((self.lat_min,
                                        self.lat_max)))
        """

    def new_sample(self, new_long, new_lat, new_field):
        """
        Takes a new sample with its longitude, latitude and (post-processed)
                                        field values, and performs an
        interpolation using the previous samples together with the current one.
                                         When it is done, it sends the
        result through the associated output pipe.
        :param new_long:
        :param new_lat:
        :param new_field:
        :return:
        """
        """
        # Debugging -------------------
        print("------------NEW SAMPLE------------------")
        print("New sample: "+str((new_long, new_lat)))
        # -------------------------------------------
        """
        self._check_boundaries(new_long, new_lat)

        # Indexes of moving window of points to update (up to, not including
                                        the stop)
        long_center_flindex = (new_long - self.long_min)/self._LONG_STEP  #
                                        Float approximation of
                                        longitude index
        long_start_index = int(long_center_flindex - self._WINDOW_SIZE/2)
        long_stop_index = long_start_index + self._WINDOW_SIZE
        lat_center_flindex = (new_lat - self.lat_min)/self._LAT_STEP
        lat_start_index = int(lat_center_flindex - self._WINDOW_SIZE/2)
        lat_stop_index = lat_start_index + self._WINDOW_SIZE

        # First latitude (row), then longitude (column). Counter-intuitive:
        windowed_interp = self.interp_matrix[lat_start_index: lat_stop_index,
                                        long_start_index:
                                        long_stop_index]
        windowed_D = self.accumulated_D_matrix[lat_start_index: lat_stop_index,
                                         long_start_index:
                                        long_stop_index]
        windowed_lat = self.mesh_lat[lat_start_index: lat_stop_index,
                                        long_start_index:
                                        long_stop_index]
        windowed_long = self.mesh_long[lat_start_index: lat_stop_index,
                                        long_start_index:
                                        long_stop_index]

        distances = haversine_distance(windowed_long, windowed_lat, new_long,
                                        new_lat)

        weighed_distances = self._weighting_function(distances)

        # Using [:, :] updates the values in the original interp_matrix instead
                                         of overwriting windowed_interp
                                         variable
        windowed_interp[:, :] *= windowed_D
        windowed_interp[:, :] += weighed_distances*new_field
        windowed_D[:, :] += weighed_distances
        aux_windowed_D = windowed_D > 0
        windowed_interp[aux_windowed_D] /= windowed_D[aux_windowed_D]
```

```
        """
        # For visualizing window sizes only
        self.interp_matrix *= 0
        windowed_interp[:, :] = distances
        """

    def _check_boundaries(self, long, lat):
        """
        Checks if the indicated longitude and latitude fall withing the
                                                existing grid, and enlarges the
                                                 grid
        if they are outside.
        Grid is enlarged by a fixed amount (_EXPAND_SIZE rows/columns), and the
                                                 new sample is assumed to fall
                                                inside.
        :param long: New sample's longitude (decimal degrees)
        :param lat: New sample's latitude (decimal degrees)
        """
        """
        # Debugging ------------------------------
        print("-----------CHECK BOUNDARIES--------------")
        print("Longitude limits: "+str((self.long_min, self.long_max))+\
            " - Latitude limits: "+str((self.lat_min, self.lat_max)))
        print("Size of interp_matrix before expansion: " + str(self.
                                                interp_matrix.shape))
        print("size of accumulated_D_matrix before expansion: " + str(self.
                                                accumulated_D_matrix.shape))
        # --------------------------------------------------------
        """
        if (long+self._D_MAX_LONG) > self.long_max:
            # A zero-filled matrix is appended "on the right" of interp_matrix.
                                                     inter_long array
                                                    regenerated
            # Number of appended columns: _EXPAND_SIZE + however many are
                                                    necessary between long_max
                                                    and the new long
            total_expand = self._EXPAND_SIZE + int((long+self._D_MAX_LONG -
                                                    self.long_max)/self.
                                                    _LONG_STEP)
            self.long_max += total_expand*self._LONG_STEP
            appended_array1 = np.zeros((self.interp_matrix.shape[0],
                                                    total_expand))
            appended_array2 = np.zeros((self.interp_matrix.shape[0],
                                                    total_expand))

            self.interp_matrix = np.hstack((self.interp_matrix, appended_array1
                                                    ))
            self.accumulated_D_matrix = np.hstack((self.accumulated_D_matrix,
                                                    appended_array2))
            self.interp_long = np.arange(start=self.long_min,
                                         stop=self.long_max + 0.5*self.
                                                                        _LONG_STEP
                                                                        ,

                                                                        #
                                                                        +
                                                                        0
                                                                        .
                                                                        5
```

```
                                                                                to
                                                                                avoid
                                                                                rounding
                                                                                errors
                                  step=self._LONG_STEP)
            # Meshgrids need to be re-created
            self.mesh_long, self.mesh_lat = np.meshgrid(self.interp_long, self.
                                        interp_lat, copy=False)

        elif (long-self._D_MAX_LONG) < self.long_min:
            # A zero-filled matrix is appended "on the left" of interp_matrix.
                                        interp_long array
                                        regenerated
            # Number of appended columns: _EXPAND_SIZE + however many are
                                        necessary between the new
                                        long and long_min
            total_expand = self._EXPAND_SIZE + int((self.long_min - (long-self.
                                        _D_MAX_LONG))/self.
                                        _LONG_STEP)
            self.long_min -= total_expand*self._LONG_STEP
            appended_array1 = np.zeros((self.interp_matrix.shape[0],
                                        total_expand))
            appended_array2 = np.zeros((self.interp_matrix.shape[0],
                                        total_expand))

            self.interp_matrix = np.hstack((appended_array1, self.interp_matrix
                                        ))
            self.accumulated_D_matrix = np.hstack((appended_array2, self.
                                        accumulated_D_matrix))
            self.interp_long = np.arange(start=self.long_min,
                                  stop=self.long_max + 0.5*self.
                                                                                _LONG_STEP
                                                                                ,
                                  step=self._LONG_STEP)
            self.mesh_long, self.mesh_lat = np.meshgrid(self.interp_long, self.
                                        interp_lat, copy=False)

        if (lat+self._D_MAX_LAT) > self.lat_max:
            # A zero-filled matrix is appended at the end of interp_matrix.
                                        interp_lat array
                                        regenerated
            # Number of appended rows: _EXPAND_SIZE + however many are
                                        necessary between the new
                                        lat and lat_max
            total_expand = self._EXPAND_SIZE + int((lat+self._D_MAX_LAT - self.
                                        lat_max)/self._LAT_STEP)
            self.lat_max += total_expand*self._LAT_STEP
            appended_array1 = np.zeros((total_expand, self.interp_matrix.shape[
                                        1]))
            appended_array2 = np.zeros((total_expand, self.interp_matrix.shape[
                                        1]))

            self.interp_matrix = np.vstack((self.interp_matrix, appended_array1
                                        ))
            self.accumulated_D_matrix = np.vstack((self.accumulated_D_matrix,
                                        appended_array2))

            self.interp_lat = np.arange(start=self.lat_min,
```

```
                                    stop=self.lat_max + 0.5*self._LAT_STEP,
                                    step=self._LAT_STEP)
            self.mesh_long, self.mesh_lat = np.meshgrid(self.interp_long, self.
                                            interp_lat, copy=False)

        elif (lat-self._D_MAX_LAT) < self.lat_min:
            # A zero-filled matrix is appended at the beginning of
                                            interp_matrix. interp_lat
                                            array regenerated
            # Number of appended rows: _EXPAND_SIZE + however many are
                                            necessary between lat_min
                                            and the new lat
            total_expand = self._EXPAND_SIZE + int((self.lat_min - (lat - self.
                                            _D_MAX_LAT))/self._LAT_STEP
                                            )
            self.lat_min -= total_expand*self._LAT_STEP
            appended_array1 = np.zeros((total_expand, self.interp_matrix.shape[
                                            1]))
            appended_array2 = np.zeros((total_expand, self.interp_matrix.shape[
                                            1]))

            self.interp_matrix = np.vstack((appended_array1, self.interp_matrix
                                            ))
            self.accumulated_D_matrix = np.vstack((appended_array2, self.
                                            accumulated_D_matrix))

            self.interp_lat = np.arange(start=self.lat_min,
                                    stop=self.lat_max + 0.5*self._LAT_STEP,
                                    step=self._LAT_STEP)
            self.mesh_long, self.mesh_lat = np.meshgrid(self.interp_long, self.
                                            interp_lat, copy=False)
        """
        # - Debugging ----------------------------------------------
        print("Size of interp_matrix after expansion: " + str(self.
                                        interp_matrix.shape))
        print("size of accumulated_D_matrix after expansion: " + str(self.
                                        accumulated_D_matrix.shape))
        print("New? Longitude limits: "+str((self.long_min, self.long_max))+\
            " - Latitude limits: "+str((self.lat_min, self.lat_max)))
        ("--------------------END OF CHECK BOUNDARIES-----------------------")
        # print
                                        ---------------------------------------------

        """

    def _weighting_function(self, dist_matrix):
        """
        Generates the result of the interpolator weighting function. Works on
                                        the whole distance matrix at
                                        once
        :param dist_matrix: matrix containing the distance to the newest sample
                                         from all points within the
                                        window
        :return: Associated weighting function result to the input distance (
                                        matrix)
        """
        dist_matrix[dist_matrix < self._D_MIN] = self._D_MIN
        dist_matrix[dist_matrix > self._D_MAX] = np.inf  # Numpy handles
                                        division by infinite, could be
                                        improved
        return 1/(dist_matrix*dist_matrix)

    def send(self):
```

```
        # Masking the interpolated matrix to hide non-updated values.
        masked_matrix = np.ma.array(self.interp_matrix, copy=False, mask=(self.
                                        interp_matrix == 0))
        dict_out = {"type": "interp", "long": self.interp_long, "lat": self.
                                        interp_lat, "data":
                                        masked_matrix}
        self.pipe_out.send(dict_out)


class DummyInterpolator:
    """
    Interpolator used for linear-only interpolation, which requires no
                                        interpolated matrix generation.
    To avoid unnecessary overhead, no processing is done and no interpolated
                                        data is sent back to the main
                                        program.
    This class only exists to provide a consistent interpolator-independent
                                        flow for the packet handling.
    """
    def __init__(self, pipe, **kwargs):
        pass

    def new_sample(self, *args, **kwargs):
        pass

    def send(self, *args, **kwargs):
        pass


def haversine_distance(long_a, lat_a, long_b, lat_b):
    """
    Calculates great-circle distance (in meters) between points A and B using
                                        the haversine formula (
                                        approximation).
    Errors below 0.6%.
    All parameters in degrees.
    Works for array inputs as long as there is the same number of points in A
                                        and B, or one of both is a single
                                        point.
    :param long_a: Longitude of point A.
    :param lat_a: Latitude of point A.
    :param long_b: Longitude of point B.
    :param lat_b: Latitude of point B.
    :return: Distance in meters between two points. If one (or both) inputs are
                                         an array, returns array of
                                        distances
    """
    earth_radius = 6371000
    to_radians = np.pi/180

    lat_a_rad = lat_a*to_radians
    lat_b_rad = lat_b*to_radians
    delta_long_rad = (long_b - long_a)*to_radians
    delta_lat_rad = lat_b_rad - lat_a_rad

    aux_lat = np.sin(delta_lat_rad/2)
    aux_long = np.sin(delta_long_rad/2)

    a = aux_lat*aux_lat + np.cos(lat_a_rad)*np.cos(lat_b_rad)*aux_long*aux_long
    c = 2*np.arctan2(np.sqrt(a), np.sqrt(1-a))
    return earth_radius*c
```

```
class UnsupportedMeasModeException(Exception):
    pass


class UnsupportedInterpolationException(Exception):
    pass
```

data_gatherers.py

```
# -*- coding: utf-8 -*-


import time
from threading import Thread, Lock, Event
from datetime import datetime
import os
from pathlib import Path

# Paths defined for OS-independent operation
_MAIN_PATH = Path("src/")
_LOG_PATH = Path("log/")
_FILES_PATH = Path("input_files/")


class FileDataGatherer:
    """
    Gathers data from an existing text file, used for testing and as proof of
                                        concept.
    Expected input file format the same as the output log file, but with no
                                        headers line and fixed at 2
                                        magnetometers.
    It uses the same interface with the main program as other possible
                                        DataGatherer modules.
    """
    def __init__(self, pipe):
        """
        :param pipe: Duplex pipe connecting to application core. Used for
                                                sending samples/ACKs and
                                                receiving commands.
        """
        # file = "survey_example_2Hz.txt"  # Input file
        file = "survey_example_20Hz.txt"
        n_magnetometers = 2  # Number of magnetometers

        self.filein = str(_FILES_PATH / file)
        time_now = datetime.now().strftime("%Y_%m_%d__%H_%M_%S")
        self.fileout = str(_LOG_PATH / "{}.txt".format(time_now))
        self.n_magnetometers = n_magnetometers

        # Adaptive header to number of magnetometers
        with open(self.fileout, "w") as f:
            headers = "## Columns: LONGITUDE LATITUDE "
            for m in range(self.n_magnetometers):
                headers += "READING_{} ".format(m+1)
            headers += "INT_LOCK BATTERY\n"
            f.write(headers)

        self.pipe = pipe
        self.pipe_lock = Lock()

        self.reading = Event()  # Enhanced boolean, allows blocking until set
        self.unpaused = Event()
        self.unpaused.set()

        t1 = Thread(target=self._command_thread)
        t2 = Thread(target=self._data_gather)
        t1.start()
        t2.start()
        t1.join()
        t2.join()
```

```
def _command_thread(self):
    """
    Handles incoming commands to the Gatherer and processes them as
                                    appropriate.
    "log" (start), "pause" and "stop" are considered, any other commands
                                    are ignored.
    """

    cmd = ""

    while cmd != "stop":
        cmd = self.pipe.recv()

        if cmd == "log":
            self.start = time.time()
            self.reading.set()
            self.unpaused.set()
            self.send({"type": "ACK"})
        elif cmd == "pause":
            self.unpaused.clear()
            self.send({"type": "ACK"})
        elif cmd == "stop":
            self.reading.clear()
            self.unpaused.set()  # To prevent _data_gatherer to infinitely
                                            wait for unpause
            self.send({"type": "ACK"})

def send(self, d):
    """
    Ensures thread-safe use of the pipe, so ACKs and data packets are not
                                    sent at the same time.

    :param d: formatted packet, meant for ACK or data.
    """
    with self.pipe_lock:
        self.pipe.send(d)

def _data_gather(self):
    """
    Sample packet creation loop.
    To be re-coded for each Data Gatherer type
    FileDataGatherer:
        Waits for start command. Then reads lines from input file one at a
                                        time while checking for
                                        pause or stop
    commands in each iteration. Formats the dictionary sample packet and
                                    sends it for each input line.
                                    If stopped or
    EOF is reached, None is sent as a poison pill and the thread is
                                    terminated.
    """
    self.reading.wait()  # Waits until first "start" command is sent

    with open(self.filein, "r") as filein, open(self.fileout, "a") as
                                    fileout:

        self.unpaused.wait()  # Blocks until unpaused if self.paused is set
        while self.reading.is_set():
            dict_out = {"type": "sample"}

            line = filein.readline()
            line_elements = line.split(" ")
```

```
                if len(line_elements) == 1:  # Blank line may come before EOF
                    break

                # Packet components
                dict_out["longitude"] = float(line_elements[0])
                dict_out["latitude"] = float(line_elements[1])

                log_line = "{} {} ".format(dict_out["longitude"], dict_out["
                                                        latitude"])

                for m in range(self.n_magnetometers):
                    reading = line_elements[2+m]
                    dict_out["reading_{}".format(m+1)] = float(reading)
                    log_line += reading+" "

                dict_out["battery"] = float(line_elements[2+self.
                                                        n_magnetometers])
                dict_out["integrity_lock"] = bool(int(line_elements[3+self.
                                                        n_magnetometers]))

                log_line += "{} {}\n".format(dict_out["battery"], int(dict_out[
                                                        "integrity_lock"]))

                fileout.write(log_line)  # Output file is updated with a new
                                                        line

                self.send(dict_out)
                time.sleep(0)  # Recreates delay between samples
                self.unpaused.wait()
        self.end = time.time()
        print("Time: {}".format(self.end-self.start))
        self.send(None)  # Poison pill to signal Gatherer termination


class ReplayDataGatherer:
    """
    Test Gatherer to re-play a survey plot from the drone log file.
    Fundamentally the same as FileDataGatherer, but it is set up for a
                                        different input file format.
    Expected input file: GPS and single magnetometer, with lots of control data
                                        that need to be filtered out.

    NOTE: VERY sensitive to input file format, meant exclusively for the log
                                        file generated at the drone side.
    """
    def __init__(self, pipe):
        """
        :param pipe: Duplex pipe connecting to application core. Used for
                                        sending samples/ACKs and
                                        receiving commands.
        """
        file = "10-05-2019-10_51_42_test_flight.txt"  # Input file

        self.filein = str(_FILES_PATH / file)
        self.index_error_counter = 0  # Sanity checks, due to possible
                                        inconsistencies in input file
        self.value_error_counter = 0

        self.pipe = pipe
        self.pipe_lock = Lock()

        self.reading = Event()
```

```
        self.unpaused = Event()
        self.unpaused.set()

        Thread(target=self._command_thread).start()
        self._data_gather()

    def _command_thread(self):
        """
        Handles incoming commands to the Gatherer and processes them as
                                               appropriate.
        "log" (start), "pause" and "stop" are considered, any other commands
                                               are ignored.
        """

        cmd = ""

        while cmd != "stop":
            cmd = self.pipe.recv()

            if cmd == "log":
                self.start = time.time()
                self.reading.set()
                self.unpaused.set()
                self.send({"type": "ACK"})
            elif cmd == "pause":
                self.unpaused.clear()
                self.send({"type": "ACK"})
            elif cmd == "stop":
                self.reading.clear()
                self.unpaused.set()  # To prevent _data_gatherer to infinitely
                                                       wait for unpause
                self.send({"type": "ACK"})

    def send(self, d):
        """
        Ensures thread-safe use of the pipe, so ACKs and data packets are not
                                               sent at the same time.

        :param d: formatted packet, meant for ACK or data.
        """
        with self.pipe_lock:
            self.pipe.send(d)

    def _data_gather(self):
        """
        MAIN LOOP for the Data Gatherer
        To be re-coded for each Data Gatherer type
        FileDataGatherer:
            Reads input file, formats packet dictionary
        """
        self.reading.wait()

        with open(self.filein, "r") as filein:
            last_timestamp = 0
            self.unpaused.wait()  # Blocks until unpaused

            while self.reading.is_set():  # One line represents one reading in
                                                   the test file
                line = filein.readline()

                if not (115 < len(line) < 150):
                    continue  # Line is skipped if it does not have the
                                                           expected length
```

```
            try:
                gps, magnetometer = line.split("&")

                gps_elements = gps.split(",")
                magnet_elements = magnetometer.split(" ")
                new_timestamp = float(gps_elements[1])

                if new_timestamp == last_timestamp:
                    continue  # Sample is skipped if same GPS is found (GPS
                                                                    at 2Hz,
                                                                magnetometer at
                                                                    20Hz)
                last_timestamp = new_timestamp

                dict_out = {"type": "sample"}

                latitude_coded = float(gps_elements[2])
                if gps_elements[3] == "S":  # "North/South" to +/-
                    latitude_coded *= -1
                longitude_coded = float(gps_elements[4])  # "East/West" to
                                                        +/-
                if gps_elements[5] == "W":
                    longitude_coded *= -1

                latitude_aux = round(latitude_coded, -2)  # 100xDegrees
                latitude = latitude_aux / 100 + (latitude_coded -
                                                        latitude_aux) / 60
                longitude_aux = round(longitude_coded, -2)
                longitude = longitude_aux / 100 + (longitude_coded -
                                                        longitude_aux) / 60

                field = float(magnet_elements[1])
                i_lock = bool(int(magnet_elements[2]))

                battery = float(magnet_elements[9])

                # Dictionary (inter-process "packet") components
                dict_out["longitude"] = longitude
                dict_out["latitude"] = latitude
                dict_out["reading_1"] = field
                dict_out["battery"] = battery
                dict_out["integrity_lock"] = i_lock

                self.send(dict_out)
                self.unpaused.wait()

            except IndexError:
                self.index_error_counter += 1
            except ValueError:
                self.value_error_counter += 1
        # Sanity checks to look for unaccounted file inconsistencies
        print("Index Error number: " + str(self.index_error_counter))
        print("Value Error number: " + str(self.value_error_counter))
        self.send(None)  # Poison pill to signal Gatherer termination


class FifoDataGatherer:
    """
    Communicates with receiver script through FIFO files and receives data in
                                        real time.
    """
    def __init__(self, pipe):
        """
```

```
        :param pipe: Duplex pipe connecting to application core. Used for
                                                sending samples/ACKs and
                                                receiving commands.
        """
        self.pipe = pipe
        self.pipe_lock = Lock()
        time_now = datetime.now().strftime("%Y_%m_%d__%H_%M_%S")
        self.fileout = str(_LOG_PATH / "{}.txt".format(time_now))
        self.FIFO_CMD = "fifo_cmd"
        self.FIFO_DATA = "fifo_dat"

        try:
            os.mkfifo(self.FIFO_CMD)
        except OSError:
            os.remove(self.FIFO_CMD)
            os.mkfifo(self.FIFO_CMD)

        try:
            os.mkfifo(self.FIFO_DATA)
        except OSError:
            os.remove(self.FIFO_DATA)
            os.mkfifo(self.FIFO_DATA)

        with open(self.fileout, "w") as f:
            headers = "## Columns: LONGITUDE LATITUDE [MAGNETOMETER_1,
                                                    MAGNETOMETER_2, ...]
                                                    INT_LOCK BATTERY\n"
            f.write(headers)

        # Thread(target=self._data_thread, daemon=False).start()
        t1 = Thread(target=self._init_receiver, daemon=True).start()  #
                                                Separate thread for the
                                                receiver C++ script
        t2 = Thread(target=self._command_thread, daemon=True).start()
        # self._command_thread()
        self._data_thread()
        print("Starting joining")
        t1.join()
        print("Init receiver joined")
        t2.join()
        print("Command thread joined")

    def _command_thread(self):
        """
        Handles incoming commands to the Gatherer and processes them as
                                                appropriate.
        "log" (start), "pause" and "stop" are considered, any other commands
                                                are ignored.
        """

        print("[Data Gatherer] Starting sending thread")
        cmd = ""
        while cmd != "stop":
            cmd = self.pipe.recv()
            print("[Data Gatherer] Command will be sent - " + cmd)
            with open(self.FIFO_CMD, "w") as cmd_file:
                cmd_file.write(cmd)
            print("[Data Gatherer] Command has been sent - "+cmd)

        print("[Data Gatherer] Command thread was terminated")
        os.remove(self.FIFO_CMD)
        print("command thread ded")
```

```
    def _data_thread(self):
        print("[Data Gatherer] Data thread started")
        with open(self.fileout, "w") as fileout:
            reading = True
            while reading:
                with open(self.FIFO_DATA, "r") as data_file:
                    line = data_file.readline()
                print("[Data Gatherer] Line has been read")
                try:
                    line = line.strip()
                    if "None" in line:
                        reading = False
                        print("[Data Gatherer] None received - terminating")
                        break
                    elif "acknowledge" in line:
                        dict_out = {"type": "ACK"}
                        print("[Data Gatherer] ACK received")
                    else:
                        dict_out = {"type": "sample"}

                        print("[DataGatherer] Data sample received" + line)

                        line_elements = line.split(",")
                        gps_data = line_elements[0:3]  # GPS[0] is timestamp,
                                                        unnecessary for
                                                         now
                        latitude_coded = float(gps_data[1])  # float version of
                                                         DDMM.MMMM
                        longitude_coded = float(gps_data[2])  # float version
                                                        of DDDMM.MMMM

                        latitude_aux = round(latitude_coded, -2)  # 100xDegrees
                        latitude = latitude_aux/100 + (latitude_coded-
                                                        latitude_aux)/
                                                        60
                        longitude_aux = round(longitude_coded, -2)
                        longitude = longitude_aux/100 + (longitude_coded-
                                                        longitude_aux)/
                                                        60

                        dict_out["latitude"] = latitude
                        dict_out["longitude"] = longitude

                        dict_out["battery"] = abs(float(line_elements[4]))

                        n_magnetometers = (len(line_elements)-3) // 2  # 3
                                                        fields for gps.
                                                         +2 fields per
                                                        magnetometer

                        i_lock = True
                        logged_sample = "{} {}".format(dict_out["longitude"],
                                                        dict_out["
                                                        latitude"])  #
                                                        log entry

                        for m in range(n_magnetometers):
                            dict_out["reading_{}".format(m+1)] = float(
                                                                line_elements
                                                                [3+2*m])
                            i_lock = (i_lock and (float(line_elements[4 + 2*m])
                                                                 > 0))  #
                                                                i_lock
```

```
                                                            covers all
                                                            magnetometers
                    logged_sample += " {}".format(dict_out["reading_{}"
                                                            .format(m+1
                                                            )])
                dict_out["integrity_lock"] = i_lock

                logged_sample += " {} {}\n".format(dict_out["battery"],
                                                          int(dict_out["
                                                         integrity_lock"
                                                         ]))
                fileout.write(logged_sample)

            print(dict_out)
            self.send(dict_out)
        except IndexError:
            print("BAD SAMPLE!")

    self.send(None)

def delete_fifos(self):
    """
    Unimplemented.
    Deletes FIFO files. To be called always at program shutdown.
    """
    try:
        os.remove(self.FIFO_DATA)
        os.remove(self.FIFO_CMD)
    except OSError:
        pass

def send(self, d):
    """
    Ensures thread-safe use of the pipe, so ACKs and data packets are not
                                        sent at the same time.

    :param d: formatted packet, meant for ACK or data.
    """
    with self.pipe_lock:
        self.pipe.send(d)

def _init_receiver(self):
    """
    To be called on a separate thread/process.
    Launches the MagPro utility to handle the radio communications and
                                        gives it control of the thread.
    MagPro must be launched manually at the aircraft side.
    """
    os.system(str(_MAIN_PATH / "MagPro" / "MagPro"))
```

Technical University of Denmark

DTU Space
Elektrovej Building 328
DK 2800 Kgs. Lyngby

www.space.dtu.dk