



**Magnetic Anomaly Mapping for Navigation**

**THESIS**

Luke T Bergeron, B.S.E.E., Captain, USAF

AFIT-ENG-MS-23-M-010

**DEPARTMENT OF THE AIR FORCE  
AIR UNIVERSITY**

**AIR FORCE INSTITUTE OF TECHNOLOGY**

**Wright-Patterson Air Force Base, Ohio**

DISTRIBUTION STATEMENT A  
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views expressed in this document are those of the author and do not reflect the official policy or position of the United States Air Force, the United States Department of Defense or the United States Government. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

AFIT-ENG-MS-23-M-010

Magnetic Anomaly Mapping for Navigation

THESIS

Presented to the Faculty  
Department of Electrical and Computer Engineering  
Graduate School of Engineering and Management  
Air Force Institute of Technology  
Air University  
Air Education and Training Command  
in Partial Fulfillment of the Requirements for the  
Degree of Master of Science in Electrical Engineering

Luke T Bergeron, B.S.E.E., B.S.E.E.  
Captain, USAF

March 23, 2023

DISTRIBUTION STATEMENT A  
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AFIT-ENG-MS-23-M-010

Magnetic Anomaly Mapping for Navigation

THESIS

Luke T Bergeron, B.S.E.E., B.S.E.E.  
Captain, USAF

Committee Membership:

Aaron P Nielsen, Ph.D  
Chair

Clark N Taylor, Ph.D  
Member

David A Woodburn, Ph.D  
Member

## Abstract

Magnetic navigation (MagNav) has the potential to provide a global form of navigation that does not rely on external signals. MagNav uses magnetic measurements of the Earth's anomaly magnetic field and compares those measurements to a magnetic anomaly map in order to determine the user's position [1].

Widespread use of MagNav will require a database of fully-sampled, low-altitude magnetic anomaly maps. Existing magnetic anomaly map databases usually come from under- or poorly-sampled surveys. In this work, we provide an easy to follow MagNav anomaly map generation framework and set of survey collection metrics/requirements in an effort to help facilitate and standardize the creation of such a database.

We explore the necessary equipment, sensors, and software algorithms required to conduct a magnetic anomaly survey and generate a map for use with MagNav. We show that this can be accomplished using one or more low-cost fixed wing unmanned aerial vehicles (UAVs) to conduct aeromagnetic surveys with minimal commercial off-the-shelf (COTS) avionics/sensors and, in certain cases, no local ground magnetic reference station or flown tie lines.

Existing magnetic anomaly maps are focused on geological interpretation. This work lays out requirements for anomaly maps focused on navigation potential. It further describes and implements a process to create navigable maps from airborne data. Lastly, we present a novel method of estimating diurnal and space-weather related magnetic noise encountered during an aeromagnetic survey without a dedicated local reference magnetometer.

We conducted simulations modeling different aeromagnetic survey processes and

algorithms to validate the MagNav mapping framework and provide insight on when to use certain processes and algorithms versus others. We then apply the validated MagNav mapping framework to create a magnetic anomaly map of a flight test area as an example use-case.

Lastly, a novel method of estimating temporal magnetic noise encountered during an aeromagnetic survey without a dedicated local reference magnetometer is presented.

# Contents

	Page
Abstract . . . . .	iv
List of Figures . . . . .	viii
List of Tables . . . . .	xiv
I. Introduction . . . . .	1
1.1 Problem Background . . . . .	1
1.2 Research Objectives . . . . .	2
1.3 Document Overview . . . . .	3
II. Background and Literature Review . . . . .	5
2.1 Magnetic Field Components of the Earth . . . . .	5
2.1.1 Core Field . . . . .	5
2.1.2 Anomaly Field . . . . .	6
2.2 External Magnetic Fields . . . . .	8
2.2.1 Temporal Effects . . . . .	9
2.2.2 Body Effects . . . . .	10
2.3 Survey Platform Types . . . . .	11
2.4 Survey Magnetometers . . . . .	15
2.4.1 Fluxgate Magnetometers . . . . .	15
2.4.2 Magnetoresistive Magnetometers . . . . .	17
2.4.3 Optically Pumped Magnetometers . . . . .	18
2.5 Survey Platform Calibration . . . . .	19
2.5.1 Spin Test . . . . .	20
2.5.2 Tolles-Lawson Calibration . . . . .	24
2.6 Ground Reference Stations . . . . .	27
2.7 Survey Path Planning . . . . .	28
2.8 Anomaly Map Leveling . . . . .	29
III. Methodology . . . . .	34
3.1 Preamble . . . . .	34
3.2 Extended Reference Station Analysis . . . . .	34
3.3 Simulated Aeromagnetic Surveys . . . . .	37
3.3.1 Simulated Magnetic Anomaly Field . . . . .	38
3.3.2 Simulated Survey Measurements . . . . .	39
3.3.3 Simulated Noise and Distortion . . . . .	39
3.3.4 Map Creation from Simulated Measurements . . . . .	43
3.4 Real-World Aeromagnetic Surveys . . . . .	43
3.4.1 UAV Design . . . . .	44

	Page
3.4.2 UAV Survey .....	47
3.4.3 MFAM Sensor Head Meshing .....	48
3.4.4 UAV Calibration .....	48
3.4.5 Survey Data Collection .....	50
3.4.6 Survey Data Preparation .....	53
3.4.7 Noise and Distortion Removal .....	55
3.4.8 Map Interpolation .....	57
3.4.9 Map Comparison .....	58
<b>IV. Results and Analysis.....</b>	<b>59</b>
4.1 Preamble .....	59
4.2 Extended Reference Station Analysis Results.....	59
4.3 Simulated Aeromagnetic Survey Results .....	64
4.4 Real-World Aeromagnetic Survey Results .....	72
4.4.1 UAV Survey Results .....	72
4.4.2 UAV Calibration Results .....	72
4.4.3 MFAM Sensor Head Validity Results .....	77
4.4.4 Map Leveling Results .....	80
4.4.5 Map Comparisons .....	82
<b>V. Conclusions .....</b>	<b>87</b>
5.1 Future Work.....	90
<b>Appendix A. Final Maps.....</b>	<b>93</b>
<b>Appendix B. Map Correlations .....</b>	<b>97</b>
<b>Appendix C. Proposed MagNav Survey Process .....</b>	<b>98</b>
<b>Appendix D. Proposed MagNav GeoTIFF File Standard .....</b>	<b>99</b>
<b>Appendix E. Python Code .....</b>	<b>108</b>
<b>Appendix F. MAMMAL Python Library .....</b>	<b>343</b>
<b>Appendix G. GeoScrapers Python Library .....</b>	<b>540</b>
<b>Bibliography .....</b>	<b>555</b>
<b>Acronyms .....</b>	<b>558</b>

## List of Figures

Figure	Page
1 Example map of the Earth's core field according the IGRF 1990 model [2].....	6
2 Example map of the Earth's anomaly field according the WDMAM [3]. .....	7
3 Projection of the Earth's magnetic anomaly field onto the core field with exaggerated proportions [4] .....	8
4 Magnetic reference station data from the Fredericksburg USGS observatory showing diurnal effects once per day. Data for this plot was obtained from INTERMAGNET: <a href="https://www.intermagnet.org/data-donnee/download-eng.php">https://www.intermagnet.org/data-donnee/download-eng.php</a> .....	9
5 Magnetic reference station data from several different magnetic observatories [5] .....	10
6 Typical intensities of permanent magnetic moments (body effect) at different locations on an aeromagnetic survey UAV. The body effects at the starboard wing (a), fuselage (b), and the port wing (c) are depicted [6].....	11
7 Trade-space between the different magnetic survey vehicle options available [6].....	12
8 Comparison of various magnetometer type sensitivities. $E$ is the Earth's magnetic field strength and GMN is the geomagnetic noise strength [7].....	16
9 Depiction of fluxgate magnetometer working principles. The field lines on the left hand side represents the external field during the different modes of operation. The plots on the right represents the hysteresis curve of the core material [7]. .....	17
10 The various components and construction of an optically pumped magnetometer [7].....	18
11 Depiction of conical dead-zones for a Geometrics 823A optically pumped magnetometer [8].....	19

Figure	Page
12 Depiction of non-orthogonality for vector magnetometers. $(x, y, z)$ is the perfectly orthogonal coordinate system of the magnetic field and $(x', y', z')$ is the non-orthogonal coordinate system of the misaligned vector sensors [9]. . . . .	21
13 Example results of applying a spin test calibration to a fluxgate magnetometer and the calculated sensitivities, offsets, and non-orthogonality angles [9]. . . . .	23
14 Example of scalar magnetometer readings before and after calibration using the Tolles-Lawson method [5]. . . . .	26
15 Example cloverleaf flight path used to determine calibration FOM [5]. . . . .	27
16 World map of INTERMAGNET observatories as of July 2022. Sites marked in red are active and those marked in grey are inactive. Image clipped from <a href="https://intermagnet.github.io/metadata/#/map">https://intermagnet.github.io/metadata/#/map</a> . . . . .	28
17 Example aeromagnetic survey flight path with dense, parallel flight lines and sparse orthogonal tie lines [1]. . . . .	29
18 Magnetic anomaly truth map for simulated aeromagnetic surveys. . . . .	38
19 Simulated survey flight path and magnetometer measurement locations. . . . .	40
20 Simulated survey magnetic anomaly measurements without noise or distortion. . . . .	40
21 Temporal variation from the INTERMAGNET observatory in Fredericksburg, Virginia. This dataset was used as the true temporal variation when simulating survey data. . . . .	42
22 SIG Rascal UAV modified for aeromagnetic surveyance. . . . .	44
23 View of SIG Rascal UAV magnetometers from within the vehicle. Camera was placed in the fuselage directly under the main wings and is facing towards the tail of the UAV. . . . .	45

Figure		Page
24	Tail-mounted SIG Rascal UAV magnetometers. . . . .	46
25	MFAM and VMR magnetometers in their combined, 3D printed mount. . . . .	47
26	Approximate survey, UAV ground control station, and magnetic ground reference station locations for the Camp Atterbury UAV range. . . . .	51
27	The three flight profiles used for the aeromagnetic anomaly surveys at the Camp Atterbury UAV range. . . . .	51
28	Magnetic ground reference station enclosure interior. . . . .	52
29	Magnetic ground reference station enclosure, generator, and solar panels placed next to the Camp Atterbury UAV runway . . . . .	52
30	Comparison of raw scalar magnitude values from Boulder and Fredericksburg INTERMAGNET reference stations over a 2 day period. . . . .	60
31	Comparison of zero-starting scalar magnitude values from Boulder and Fredericksburg INTERMAGNET reference stations over a 2 day period. . . . .	60
32	Comparison between the true and estimated temporal variation at the Fredericksburg INTERMAGNET reference station using the ERSM method. . . . .	62
33	Comparison of raw scalar magnitude values from Boulder and Fredericksburg INTERMAGNET reference stations over the 24 hour period after calibration. . . . .	62
34	Comparison between the true and estimated temporal variation at the Fredericksburg INTERMAGNET reference station using the ERSM method. The optimal scale and offset parameters that were used for this estimation were calculated using the previous 48 hours worth of data from the same sites. . . . .	63
35	Simulation truth magnetic anomaly map. . . . .	64
36	Generated magnetic anomaly map with all survey data and no added noise. . . . .	65

Figure	Page
37 Difference between the truth anomaly map (Figure 35) and the map created with all survey data and no added noise (Figure 36).....	65
38 Generated magnetic anomaly map without accounting for temporal variation .....	66
39 Difference between the truth anomaly map (Figure 35) and the map created without temporal variation corrections (Figure 38).....	67
40 Generated magnetic anomaly map using estimated temporal variation with the ERSM method.....	68
41 Difference between the truth anomaly map (Figure 35) and the map using estimated temporal variation with the ERSM method(Figure 40). .....	68
42 Generated maps where simulated magnetic measurements were biased by 0.1 (a), 1 (b), and 10 nT (c) .....	69
43 Difference between the truth anomaly map (Figure 35) and the maps where the simulated magnetic measurements were biased by 0.1 (a), 1 (b), and 10 nT (c) .....	69
44 Generated maps where random Gaussian noise with standard deviations of 1 (a), 10 (b), and 100 nT (c) were applied to the simulated scalar magnetometer measurements. ....	71
45 Difference between the truth anomaly map (Figure 35) and maps where random Gaussian noise with standard deviations of 1 (a), 10 (b), and 100 nT (c) were applied to the simulated scalar magnetometer measurements. ....	71

Figure	Page
46 Raw MFAM scalar, MFAM compass (vector), and VMR magnetometer data collected during the static UAV calibration maneuver. The orientation oscillations and heading changes of the aircraft can be seen in the vector magnetometer plots (bottom 6 plots, 3 for the MFAM compass and 3 for the VMR). Noise artifacts that correlate to these movements can be observed in the MFAM scalar magnetometer plot (top plot). These artifacts are due to body effects and heading error encountered during the calibration maneuver. ....	73
47 Calibrated MFAM scalar and VMR magnetometer data from the static UAV calibration maneuver. The Tolles-Lawson coefficients used are given in Table 9. The VMR vector magnetometer was used to calculate the direction cosines. ....	76
48 Results of applying the Tolles-Lawson coefficients in Table 9 to the (a) first 1 km Atterbury survey, (b) second 1 km Atterbury survey, (c) 2 km Atterbury survey, and (d) 4 km Atterbury survey. The blue traces represent the raw measurements and the orange traces represent the calibrated measurements. Note that in all cases except for (a), the calibrated measurements had less heading error and ringing compared to the raw measurements. ....	78
49 Spatial plots showing MFAM sensor head validity for each sample in the (a) first 1 km Atterbury survey, (b) second 1 km Atterbury survey, (c) 2 km Atterbury survey, and (d) 4 km Atterbury survey. Green dots represent a sample where both MFAM sensor head measurements were valid, red when only MFAM sensor head 1 was valid, and blue when only MFAM sensor head 2 was valid. ....	79
50 Comparison between different methods of map leveling techniques for all aeromagnetic surveys based on calculated corrugation FOM. ....	81

Figure	Page	
51	Maps of the first 1 km Atterbury survey after leveling with (a) no technique, (b) pseudo tie line leveling using PCA, (c) leveling each flight line individually against all intersecting tie lines, and (d) tie line leveling all flight lines simultaneously using a plane of best fit. ....	93
52	Maps of the second 1 km Atterbury survey after leveling with (a) no technique, (b) pseudo tie line leveling using PCA, (c) leveling each flight line individually against all intersecting tie lines, and (d) tie line leveling all flight lines simultaneously using a plane of best fit. ....	94
53	Maps of the 2 km Atterbury survey after leveling with (a) no technique, (b) pseudo tie line leveling using PCA, (c) leveling each flight line individually against all intersecting tie lines, and (d) tie line leveling all flight lines simultaneously using a plane of best fit. ....	95
54	Maps of the 4 km Atterbury survey after leveling with (a) no technique, (b) pseudo tie line leveling using PCA, (c) leveling each flight line individually against all intersecting tie lines, and (d) tie line leveling all flight lines simultaneously using a plane of best fit. ....	96
55	2D correlation plots between the first 1 km PCA and second 1 km plane of best fit leveled Atterbury survey maps for the (a) scalar, (b) easterly gradient, and (c) northerly gradient rasters. ....	97
56	2D correlation plots between the 2 km and 4 km per flight line leveled Atterbury survey maps for the (a) scalar, (b) easterly gradient, and (c) northerly gradient rasters. ....	97
57	2D correlation plots between the second 1 km plane of best fit and 4 km per flight line leveled Atterbury survey maps for the (a) scalar, (b) easterly gradient, and (c) northerly gradient rasters. ....	97

## List of Tables

Table		Page
1	Comparison of flight characteristics for different UAV types averaged across 55 aeromagnetic surveying studies [6].....	14
2	Extended reference station analysis site location data. These values were obtained from the headers in the data files from each site.....	35
3	Simulated survey parameters. ....	39
4	Optimal ERSM parameters for Boulder to Fredericksburg.....	61
5	Simulated map errors due to bias. ....	69
6	Simulated map errors due to random noise. ....	70
7	UAV survey results.....	72
8	UAV calibration results. Y denotes the given Tolles-Lawson coefficients were calculated and used where N signifies such coefficients were not used.....	74
9	Tolles-Lawson calibration coefficients used for survey data processing. ....	75
10	Optimal Map Leveling techniques and Corrugation FOM for Each Atterbury Survey. ....	82
11	Maximum Pearson correlation coefficients between the optimally leveled Maps of the Atterbury aeromagnetic surveys. The lags at which the maximum correlation coefficients were found are also given in parenthesis under each correlation coefficient. Plots of the corresponding 2D correlations can be found in Appendix B. ....	83
12	Zero-lag Pearson correlation coefficients between the optimally leveled Maps of the Atterbury aeromagnetic surveys. Plots of the corresponding 2D correlations can be found in Appendix B. ....	84

# Magnetic Anomaly Mapping for Navigation

## I. Introduction

### 1.1 Problem Background

Virtually all modern aircraft, both military and civilian, heavily rely on Global Navigation Satellite System (GNSS) solutions to navigate. While GNSS-based navigation allows the user to quickly and accurately determine their absolute position with low cost hardware, a backup navigation solution is desired in the event GNSS signals become unavailable during flight. Currently, magnetic navigation (MagNav) is the most promising alternate navigation method for flying in an environment lacking GNSS availability. This is because the magnetic anomaly field used by MagNav solutions is available during both day and night, unaffected by weather or terrain, detectable with passive sensors, and is virtually impossible to jam or spoof, but requires the use of anomaly maps [1][10]. One of the most pressing issues with widespread adoption of MagNav is the severe lack of low altitude, adequately sampled magnetic anomaly maps spanning the entire United States (US). Nearly all publicly available magnetic anomaly maps were generated for geological prospecting (i.e., locating metal ore deposits), are not fully sampled, and older maps suffer from pixel geolocation errors [1][10]. This means that of the few, limited magnetic map databases available, none of them are suitable for use with MagNav in the US and a new database must be created. Since processes and software algorithms used to create magnetic anomaly maps are largely proprietary, filling the navigation map availability gap will require developing a low cost, open source, and detailed framework for end-to-end magnetic

anomaly map creation.

## 1.2 Research Objectives

The main research objective of this effort is to develop a detailed framework for end-to-end magnetic anomaly map creation. This framework will be highly flexible and serve as a comprehensive guide to a wide range of users wishing to create magnetic anomaly maps for MagNav. An individual or group of researchers who have no prior experience with aeromagnetic surveyance should be able to use this framework to develop their own high quality magnetic anomaly maps relatively easily and cheaply. This framework will cover all major aspects of the data collection and processing including sensor/vehicle choice, sensor/vehicle calibration, survey path planning, removal of non-anomalous magnetic field components from survey data, map leveling, map interpolation, and map file formatting.

Another research objective includes the design of a new algorithm that allows users to leverage data from magnetic reference stations located several thousand kilometers from the survey area for removal of the diurnal and space weather effects from survey data. This would provide a substantial improvement over the 100 km maximum distance generally accepted in the current literature for aeromagnetic surveys [11]. Such an algorithm may render the need for a local magnetic reference station unnecessary and reduce the overall cost of aeromagnetic surveys.

Additionally, we propose an updated and more accurate guidance on maximum sampling distance requirements for fully sampled magnetic anomaly maps. Currently, it is generally accepted that survey samples need only be  $h$  distance apart where  $h$  is the height of the survey AGL (above ground level) to be considered fully sampled [1]. However, MagNav anomaly maps must be sampled with no greater than  $h/2$  distance between adjacent samples to capture the entire magnetic anomaly spacial frequency

content at a given altitude.

In order to make the survey data processing accessible and extensible, the bulk of the software developed for processing survey data for this research effort was compiled into a Python library named Magnetic Anomaly Map Making for Air and Land (MAMMAL). The MAMMAL library can parse a wide variety of magnetic and other sensor data files, magnetic anomaly maps, and survey aircraft autopilot flight logs. Once parsed, the data can be further processed as appropriate to generate magnetic anomaly maps. This includes functions for data filtering, removal of known sources of noise, sensor/vehicle calibration, map/flight line leveling, map interpolation, and map export into a proposed standard data format for MagNav (see Appendix D). The proposed standard MagNav map format is a GeoTIFF with extensive contextual information about the used survey equipment and processing along with several data bands.

This standardized MagNav map format will greatly simplify the process of fusing and databasing large numbers of aeromagnetic anomaly maps. Adoption of this standard by the navigation community will ensure seamless interoperability of aeromagnetic anomaly maps with future MagNav algorithms.

Lastly, a series of magnetic anomaly maps for MagNav research purposes were created from survey data collected over the unmanned aerial vehicle (UAV) flight test range at Camp Atterbury (Edinburgh, Indiana). Both the developed aeromagnetic framework and MAMMAL library were used to create and evaluate maps from these surveys to establish the utility of both the framework and library.

### 1.3 Document Overview

The remainder of this thesis is organized as follows. Background information on the Earth's magnetic field components, external sources of magnetism and magnetic

noise, and literature review of current magnetic surveying techniques are discussed in Chapter II. In Chapter III, the methodology of the anomaly map making software package is discussed. This chapter also discusses how the aeromagnetic maps of the UAV flight test range at Camp Atterbury were created and analyzed from start to finish. Chapter IV details the results of running the anomaly map making software package under various constraints; including both with simulated and real aeromagnetic survey data. In Chapter V, the results of Chapter IV are analyzed and a standardized framework on how aeromagnetic anomaly maps for MagNav should be generated and saved is proposed. Suggestions for future work are also discussed in Chapter V.

## **II. Background and Literature Review**

This chapter covers the background information required for aeromagnetic anomaly surveying and map creation for magnetic navigation (MagNav). Section 2.1 discusses Earth’s magnetic field components. Section 2.2 covers external magnetic fields generated by sources other than Earth. Section 2.3 provides comparisons between different platforms available for magnetic anomaly mapping. Section 2.5 discusses the spin and Tolles-Lawson calibration methods for aeromagnetic surveying platforms. Section 2.6 covers the use of different types of ground magnetic reference stations including International Real-time Magnetic Observatory Network (INTERMAGNET) reference stations. Section 2.7 discusses common magnetic anomaly survey path profiles and design considerations. Finally, Section 2.8 covers different methods used to level magnetic anomaly survey data.

### **2.1 Magnetic Field Components of the Earth**

The Earth’s magnetic field can be broken down into two main components: Earth’s core field and anomaly field. The Earth’s core field must be identified and subtracted from survey magnetometer measurements to uncover the anomaly field signature of the surveyed area.

#### **2.1.1 Core Field**

The largest component of magnetic measurements during surveys is the Earth’s core field. This field is generated by the circular flow of molten ferrous material in Earth’s outer core. Comprising over 98% of Earth’s generated magnetic field, typical values of the core field include 70 000 nT at the North/South poles, 50 000 nT at the mid-latitudes, and 20 000 nT at the Equator [3]. While the Earth’s core field varies

over time, this variation occurs slowly. This allows models to be created of the Earth's core field. Currently, there are two main models used: the World Magnetic Model (WMM) and International Geomagnetic Reference Field (IGRF) [3][2][12]. While both models are accurate spherical harmonic models updated every 5 years, this research will specifically use the IGRF model for core field estimation [12].

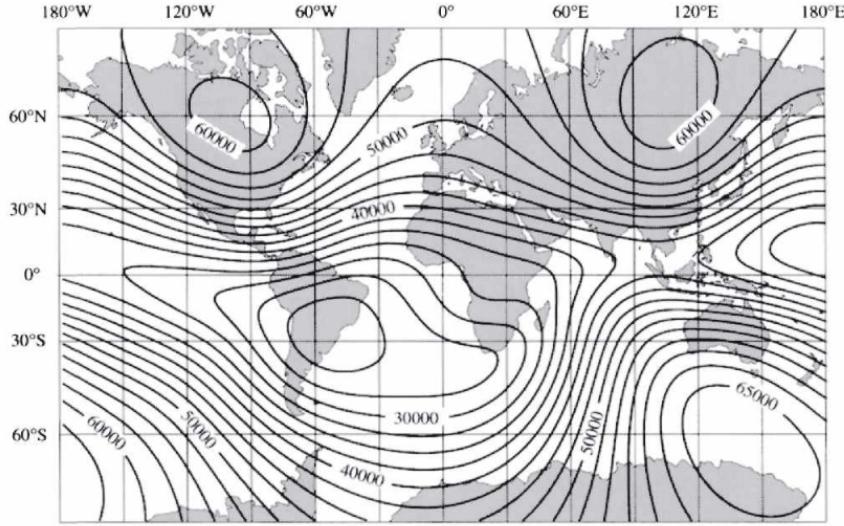


Figure 1: Example map of the Earth's core field according to the IGRF 1990 model [2].

### 2.1.2 Anomaly Field

The Earth's anomaly field is comprised of both permanent and induced magnetic fields in the Earth's crust. Typically in the  $\pm 500$  nT range, it accounts for approximately 2% of the Earth's total magnetic field at any given location [1][3].

The permanent magnetic anomaly moments stem from the cooling and solidifying of molten material on or near the Earth's surface. Material loses its magnetization when heated above its respective Curie point. As the material cools below the Curie point, it retains a permanent magnetic moment proportional to and in the same direction as the external magnetic field applied during the cooling process (e.g.

the Earth's core magnetic field). As different materials heat and cool in different locations along the Earth's surface across geological timescales, the permanent magnetic anomaly field varies across the Earth's surface [1][3].

The induced magnetic anomaly field is generated by the passing of external magnetic fields through magnetically susceptible material in the Earth's crust. Since the vast majority of the magnetic field strength at any given point on the Earth's crust is due to the Earth's core field, the induced magnetic anomaly field is largely aligned and proportional to the Earth's core field at a given location [3].

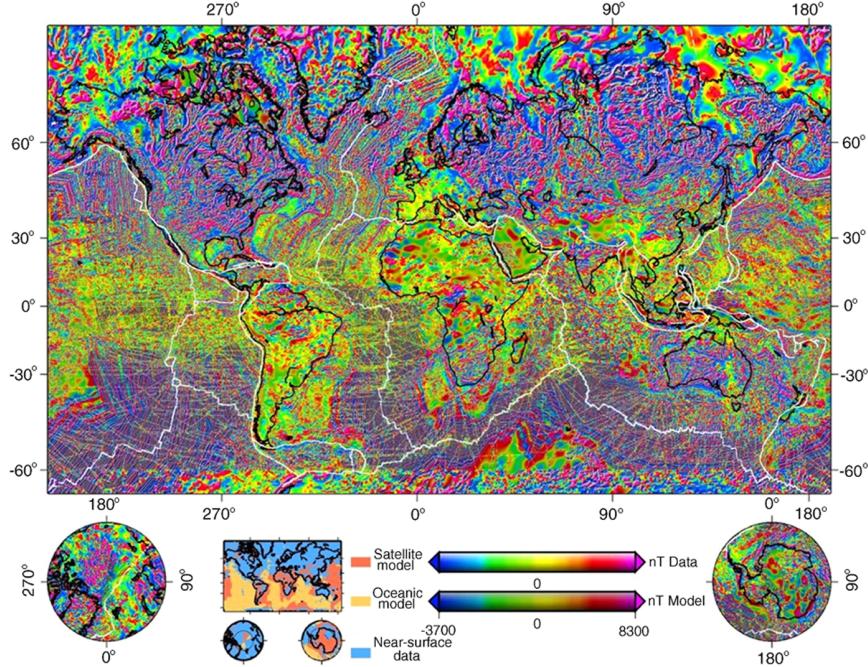


Figure 2: Example map of the Earth's anomaly field according to the the World Digital Magnetic Anomaly Map (WDMAM) [3].

The Earth's total field ( $\vec{B}_{total}$ ) is therefore given as the vector sum of the core ( $\vec{B}_{core}$ ) and anomaly ( $\vec{B}_{anomaly}$ ) fields as given in Equation (1) [4].

$$\vec{B}_{total} = \vec{B}_{core} + \vec{B}_{anomaly} \quad (1)$$

As mentioned earlier, both the permanent and induced magnetic anomaly field

components are closely aligned with the Earth's core field and are much smaller in magnitude ( $\|\vec{B}_{core}\| \gg \|\vec{B}_{anomaly}\|$ ) [3]. This allows the following approximation to be made [4]:

$$\|\vec{B}_{total}\| \approx \|\vec{B}_{core}\| + \|\vec{B}_{anomaly}\| \quad (2)$$

In this case, the anomaly field component is projected onto the core field and added together to approximate the total field. As long as  $\|\vec{B}_{core}\| \gg \|\vec{B}_{anomaly}\|$  remains valid,  $\|\vec{B}_{anomaly}\|$  can be calculated as given in Equation (3). This approximation and projection is depicted in Figure 3 with exaggerated proportions for visibility [1][4].

$$\|\vec{B}_{anomaly}\| \approx \|\vec{B}_{total}\| - \|\vec{B}_{core}\| \quad (3)$$

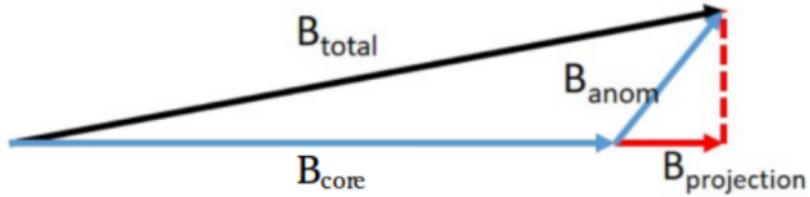


Figure 3: Projection of the Earth's magnetic anomaly field onto the core field with exaggerated proportions [4].

## 2.2 External Magnetic Fields

When conducting magnetic anomaly surveys, several sources of magnetic noise external to the Earth's fields must be accounted for to accurately estimate the anomaly field. These mainly include diurnal, space weather, and body effects [13].

### 2.2.1 Temporal Effects

Temporal effects are the sum of all temporal variations in the total magnetic field at a given location. More specifically, temporal effects include diurnal and space weather effects [13].

Diurnal effects are regularly occurring temporal variations caused by the heating and cooling cycles of the ionosphere by the Sun. As the temperature of the ionosphere changes, “tidal winds...drive ionospheric plasma against the geomagnetic field, inducing electric fields and currents” [14]. These induced currents cause gradual 10 to 50 nT shifts in the total magnetic field approximately once a day as seen in Figure 4. These variations remain largely consistent day-to-day in any given location. The diurnal effect, however, differs slightly in magnitude and shape between different locations within the same time frame as seen in Figure 5 [1][5]. Diurnal effects are also called the solar-quiet (Sq) magnetic field variation [13][14].

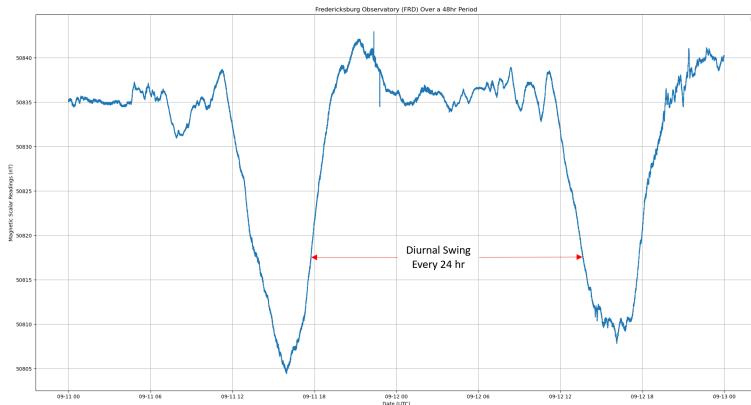


Figure 4: Magnetic reference station data from the Fredericksburg United States Geological Survey (USGS) observatory showing diurnal effects once per day. Data for this plot was obtained from INTERMAGNET: <https://www.intermagnet.org/data-donnee/download-eng.php>

Space weather effects are temporal variations in the total magnetic field that are generated by charged particles impacting Earth’s atmosphere. More specifically,

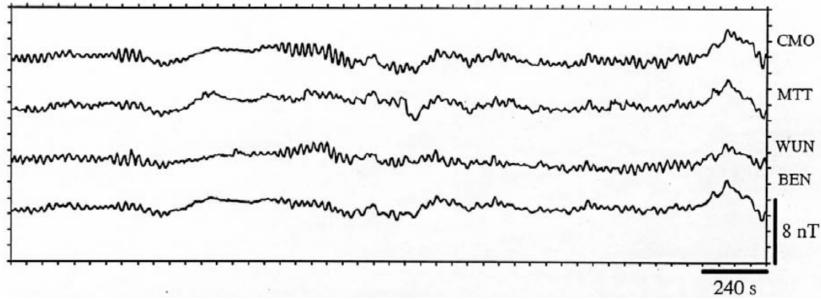


Figure 5: Magnetic reference station data from several different magnetic observatories [5]

ionized particles radiating from the Sun impact Earth’s magnetosphere causing eddy currents. These eddy currents in the magnetosphere induce magnetic fields that, when detected near the Earth’s surface, typically vary by  $\pm 10$  nT. Larger variations can occur during periods of magnetic storms and magnetic anomaly mapping should be avoided during these storms [4][13][14]. Forecasts of magnetic storms and other space weather can be accessed from the Space Weather Prediction Center: <https://www.swpc.noaa.gov/>.

To remove all temporal effects during a magnetic anomaly survey, ground magnetic reference stations near the survey site are typically used [13]. Ground reference stations are covered in more detail in Section 2.6.

### 2.2.2 Body Effects

Body effects are magnetic moments generated by the survey vehicle during the survey process that distort the magnetometer readings. Typically such moments are generated by permanent magnets, electromagnets, ferrous metal, or large masses of conductive material near the vehicle’s magnetometer. The sources of these body effects are sorted into three main categories: permanent, induced, and eddy current moments [4][15].

While there are calibration methods to cancel body effects (as described further

in Section 2.5), calibration methods are not perfect and survey vehicles should be designed to mitigate body effects to the greatest extent reasonable. This includes maximizing the distance between the magnetometer and power supplies, motors, engines, Radio Frequency (RF) devices, high current components, and ferrous metal such as steel. It is helpful to survey the vehicle similar to that in Figure 6 before selecting a magnetometer mount location so that the selected location minimizes body effects [6].

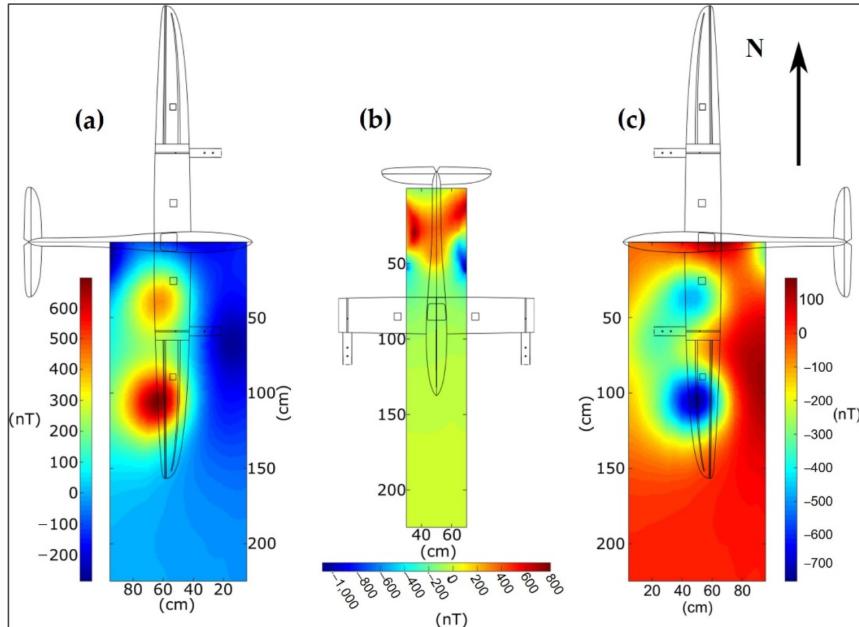


Figure 6: Typical intensities of permanent magnetic moments (body effect) at different locations on an aeromagnetic survey UAV. The body effects at the starboard wing (a), fuselage (b), and the port wing (c) are depicted [6].

### 2.3 Survey Platform Types

There are three main platform categories that can be used for magnetic anomaly surveys: terrestrial, unmanned aerial vehicle (UAV)-borne, and crewed airborne. Selecting the appropriate platform type category depends on the requirements of the magnetic map in terms of operation difficulty/cost, coverage efficiency, and resolu-

tion.

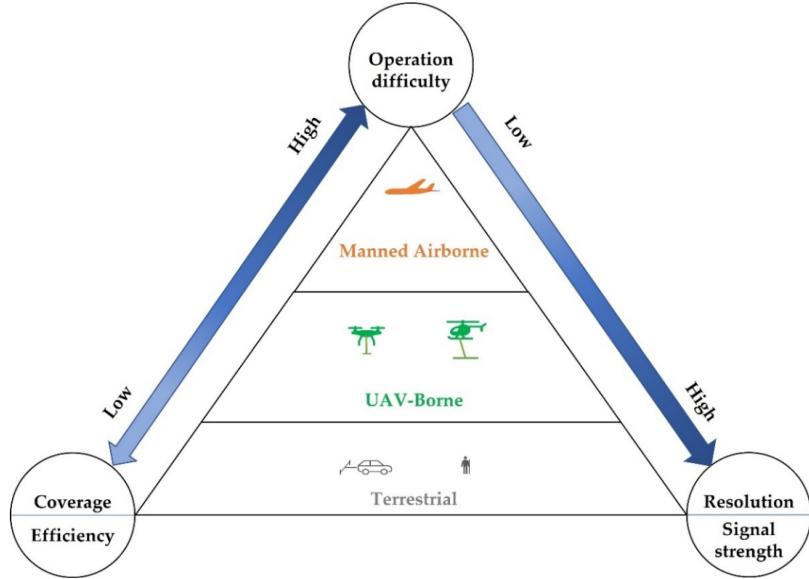


Figure 7: Trade-space between the different magnetic survey vehicle options available [6].

For MagNav, magnetic anomaly maps must have total coverage of the surveyed area and have high resolution (fully sampled). Additionally, it is preferred to minimize survey operation difficulty and cost. Utilizing the platform type trade-space visual in Figure 7, our requirements suggest the usage of a UAV-borne type. Using a UAV-borne type platform, the entire survey area can be covered regardless of hazardous terrain where terrestrial platforms might fail. For instance, a magnetic surveying truck can not completely survey an area if that area includes a large lake, but a UAV would not have such problems. Additionally, aeromagnetic navigation maps do not require the high spatial resolution provided by a ground-based survey method. Due to this higher resolution and lower survey altitude, man-made noise is more likely to be greater when using terrestrial survey platforms compared to UAV survey platforms. Also, utilizing a UAV is much cheaper and easier to procure, maintain, and operate compared to a manned airborne platform, thus minimizing cost and complexity [1][6].

Within the UAV-borne category, several different types of vehicles are available: rotary wing, fixed wing, and airship UAVs. Table 1 compares the flight characteristics of each of these UAV types averaged across a large number of recent aeromagnetic survey studies. Airship UAVs are fairly limited in both average endurance (2.00 h) and airspeed (46.80 km/h), making large MagNav aeromagnetic surveys highly inefficient. Rotary wing UAVs, including both multi-rotors (i.e., quadcopters) and unmanned helicopters also suffer from limited average endurances (0.43-1.70 h) and airspeeds (54.75-61.67 km/h). On the other hand, fixed wing UAVs, on average, have a substantially longer average endurance (6.94 h) and vastly higher average cruise speed (106.75 km/h), allowing for faster surveys of larger areas without sacrificing survey spatial resolution [6]. For these reasons, this research effort will focus exclusively on aeromagnetic surveying with fixed wing UAVs.

Table 1: Comparison of flight characteristics for different UAV types averaged across 55 aeromagnetic surveying studies [6].

UAV Type	Average Maximum Payload (kg)	Average Endurance (h)	Average Cruise Speed (km/h)	Common Uses
Multi-Rotor	4.02	0.43	54.75	Mineral Exploration, UXO Detection
Airship	8.67	2.00	46.80	Geophysical Exploration
Helicopter	23.88	1.70	61.67	Aeromagnetic Surveying, Geophysical Exploration
Fixed Wing	40.40	6.94	106.75	Aeromagnetic Surveying, Geophysical Exploration

## 2.4 Survey Magnetometers

There are two main types of magnetic sensors: vector and scalar magnetometers. Vector magnetometers measure the magnetic field strength in each of the three spacial directions of the sensor's body frame simultaneously. Scalar magnetometers measure only the magnitude of the magnetic field strength. For two ideal, co-located vector and scalar magnetometers taking measurements at the same instant, the magnitude of the vector measurement ( $\|\vec{B}_{vec}\|$ ) is equal to the scalar measurement ( $B_{scalar}$ ) as given in Equation (4).

$$\|\vec{B}_{vec}\| = \sqrt{B_x^2 + B_y^2 + B_z^2} = B_{scalar} \quad (4)$$

There are many different physical implementations of both vector and scalar magnetometers that vary in sensitivity, range, reliability, and cost (see Figure 8) [7]. For this research, only a subset of magnetometers are discussed. These include fluxgate, magnetoresistive, and optically pumped magnetometers. Each of which are sensitive enough to accurately measure the magnetic anomaly field [1][6][7].

### 2.4.1 Fluxgate Magnetometers

Fluxgate magnetometers are vector sensors that use series of coupled coils with ferromagnetic cores to detect external magnetic fields. In the case of a 3-axis fluxgate magnetometer, 3 sets of individual fluxgate sensors are fastened together so each sensor is measuring the external magnetic field in mutually orthogonal directions ( $x$ ,  $y$ , and  $z$ ) [7].

Each one of these sensors are comprised of a drive coil and sense coil both wrapped around a common ferromagnetic core. The drive coil is driven with a sinusoidal current where, when the current is maximum, the induced magnetic field causes

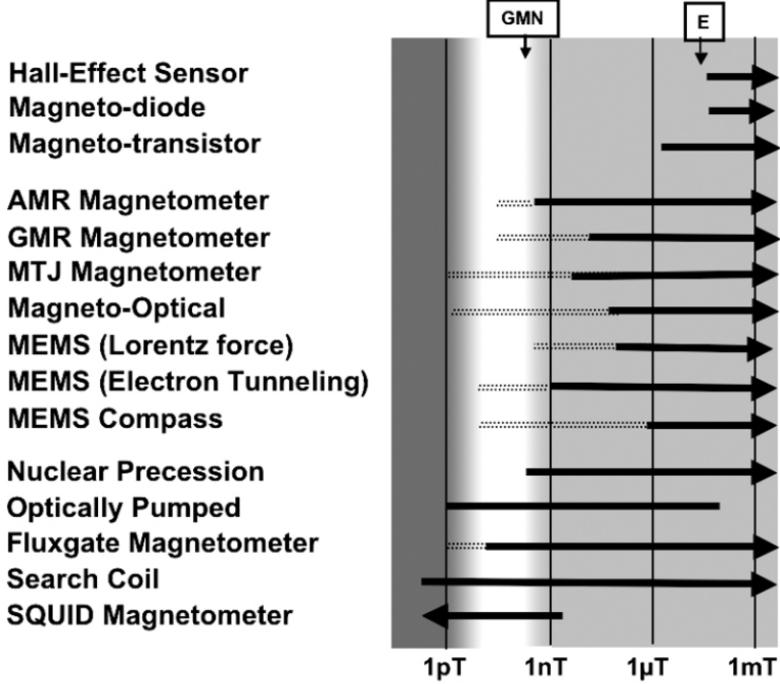


Figure 8: Comparison of various magnetometer type sensitivities.  $E$  is the Earth’s magnetic field strength and GMN is the geomagnetic noise strength [7].

the ferromagnetic core to saturate. When saturated, the ferromagnetic core does not “attract” the external magnetic field and the magnetic flux through the sense coil due to the external field is decreased. As the current through the drive coil is decreased, the ferromagnetic core becomes unsaturated and “attracts” the external magnetic field. This causes the magnetic flux through the sense coil due to the external field to increase. As a result of this process, the sense coil’s voltage at the second harmonic of the drive current frequency is proportional to the external magnetic field in the direction of the ferromagnetic core [7].

With a sensitivity range of  $10^{-2}$  to  $10^7\text{nT}$ , sample rate of up to  $10\text{ kHz}$ , and unit prices 20 to 50 times lower than industry standard scalar sensors, fluxgate magnetometers are a cost-effective choice for aeromagnetic survey sensors [7][16]. However, fluxgate outputs can have large temperature dependence if not designed well or compensated properly [17].

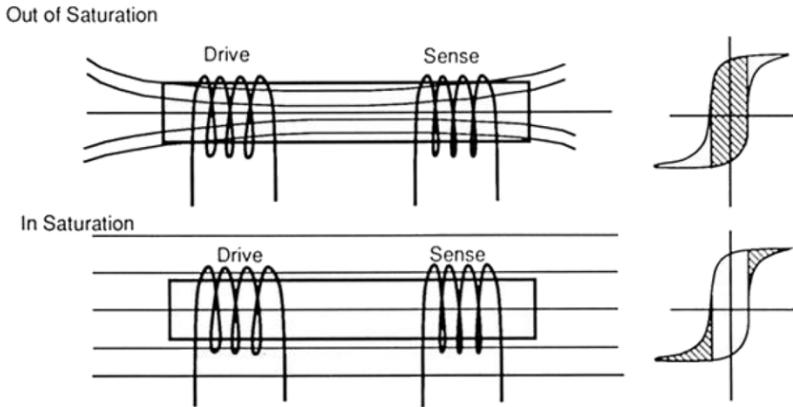


Figure 9: Depiction of fluxgate magnetometer working principles. The field lines on the left hand side represents the external field during the different modes of operation. The plots on the right represents the hysteresis curve of the core material [7].

#### 2.4.2 Magnetoresistive Magnetometers

Magnetoresistive magnetometers sense magnetic field strength by determining changes in a known material's electrical resistance. The construction of these types of magnetometers is relatively simple and usually consists of a constant current source, a strip of conductive material with known resistance in the absence of an external magnetic field, and a circuit to measure the voltage drop across the material strip. Since these devices are fairly simple, they are some of the cheapest and most widely used magnetometers in areas outside of aeromagnetic surveying. While being cheap and simple, it is entirely feasible to use vector magnetoresistive magnetometers to aid in the aeromagnetic survey platform calibration process (as described in Section 2.5). It is, however, currently under investigation as to whether or not magnetoresistive magnetometers have high enough accuracy and precision to replace the more expensive magnetometers currently used to take survey measurements (i.e. fluxgate or optically pumped magnetometers) [7].

### 2.4.3 Optically Pumped Magnetometers

Optically pumped magnetometers are scalar magnetometers that measure magnetic field strength by observing the interactions between circularly polarized light and alkali vapor. This can be achieved in two ways.

One way is to optically pump the alkali vapor with circularly polarized light until each alkali atom's single valence electron are put into a low energy state. Then an RF signal at the correct frequency is radiated parallel to the circularly polarized light and flips all of the valence electrons from their low to high energy state. The rate at which the electrons flip states after the RF field is applied and released is proportional to the total external magnetic field strength [7].

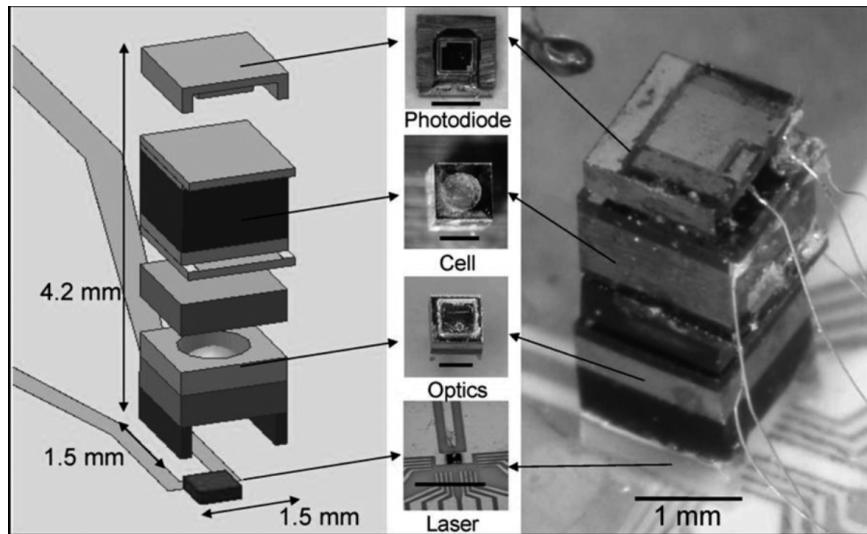


Figure 10: The various components and construction of an optically pumped magnetometer [7].

Another method includes nuclear procession. In this case, the vapor atoms are optically pumped as previously described, but a coil around the vapor is used to detect the procession of the vapor atoms. This procession frequency is proportional to the total external magnetic field strength [7].

Unlike other sensors, optically pumped magnetometers suffer from dead-zones.

Dead-zones are regions where, if the total magnetic field points through this zone, the sensor's output is invalid (see Figure 11). These dead-zones are usually expanding conical zones located along the sensor's equator/long axis. The angle of the dead-zones are minimized during sensor design and may vary between different sensor models. When the magnetic field vector is in the sensor's dead-zone, the magnetometer is not able to observe the magnetic field and outputs either 0 nT or an error signal. For aeromagnetic surveys using optically pumped magnetometers, it is advisable to use two such sensors mounted orthogonal to each other so that if one sensor is in a dead zone, the other can still produce valid measurements [1][8].

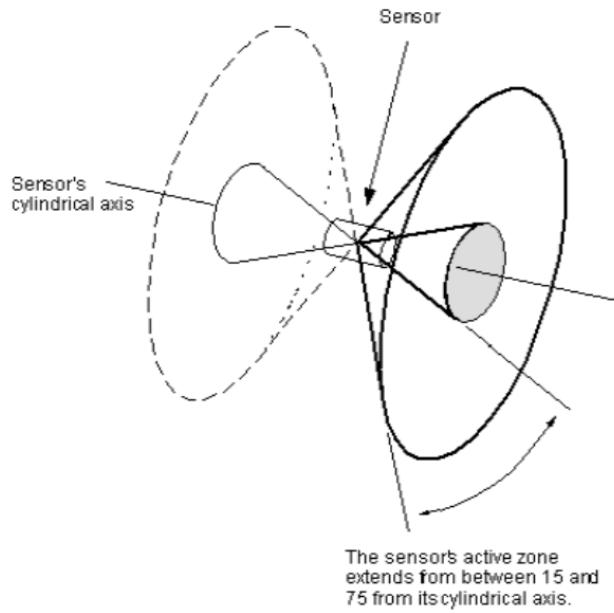


Figure 11: Depiction of conical dead-zones for a Geometrics 823A optically pumped magnetometer [8].

## 2.5 Survey Platform Calibration

Magnetometer measurements taken during aeromagnetic surveys can be distorted from body effects as mentioned in Section 2.2.2. The majority of these body effects

can be removed through calibration of the vector and scalar magnetometers. Vector magnetometers can be calibrated by a spin test and scalar magnetometers are typically calibrated by the Tolles-Lawson calibration method [13][15].

### 2.5.1 Spin Test

Typically, vector magnetometers are comprised of 3 individual sensors that each measure the magnetic field at approximately the same location, but along mutually orthogonal directions. There are three types of errors associated with vector magnetometers: scaling, non-orthogonality, and bias errors [9][16].

Scaling error is a linear noise term that is proportional to the magnetic field aligned with a given sensor. Since vector magnetometers have 3 sensors, there are 3 scaling terms ( $s_1, s_2, s_3$ ), called sensitivities, that represent the scaling error for each internal sensor. These sensitivities are used to construct a  $3 \times 3$  diagonal matrix ( $S$ ) that scales the magnetic field into vector magnetometer readings as given in Equation (5). The inverse of this matrix ( $S^{-1}$ ) can be applied to the observed vector magnetometer readings to remove the scale errors [9][16].

$$S = \begin{bmatrix} s_1 & 0 & 0 \\ 0 & s_2 & 0 \\ 0 & 0 & s_3 \end{bmatrix} \quad (5)$$

$$S^{-1} = \begin{bmatrix} \frac{1}{s_1} & 0 & 0 \\ 0 & \frac{1}{s_2} & 0 \\ 0 & 0 & \frac{1}{s_3} \end{bmatrix}$$

Non-orthogonality error is caused by the sensors not being mounted perfectly orthogonal to each other. If any 2 of these sensors have an angle ( $u$ ) of less than

90 deg between their sense directions, there will be some amount of magnetic field strength that is detected by both (see Figure 12). This causes an error in both magnitude and direction of the vector magnetometer's output. The greater the error in physical sensor mounting/alignment, the greater the non-orthogonality errors in the magnetometer output [16][9].

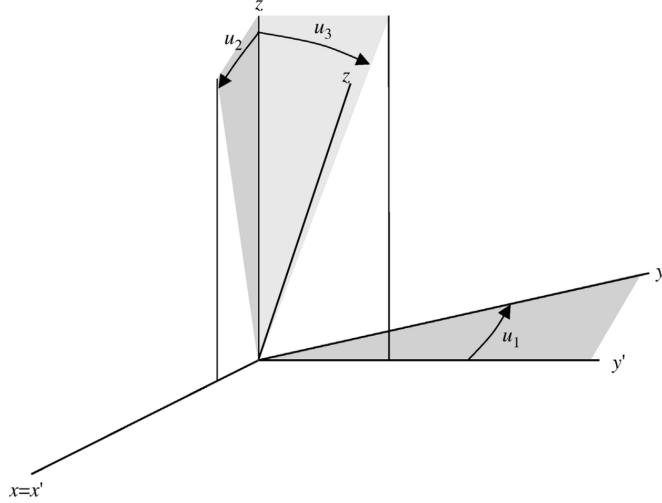


Figure 12: Depiction of non-orthogonality for vector magnetometers.  $(x, y, z)$  is the perfectly orthogonal coordinate system of the magnetic field and  $(x', y', z')$  is the non-orthogonal coordinate system of the misaligned vector sensors [9].

Since non-orthogonality causes a skewing of the vector magnetometer readings, a  $3 \times 3$  matrix ( $P$ ) is used to map the magnetic field coordinate system into the vector magnetometer coordinate system and is given by Equation (6). The inverse of this matrix ( $P^{-1}$ ) can be applied to the observed vector magnetometer readings to remove the non-orthogonality errors [9].

$$P = \begin{bmatrix} 1 & 0 & 0 \\ -\sin(u_1) & \cos(u_1) & 0 \\ \sin(u_2) & \sin(u_3) & \sqrt{1 - \sin^2(u_2) - \sin^2(u_3)} \end{bmatrix}$$

$$P^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ \frac{\sin(u_1)}{\cos(u_1)} & \frac{1}{\cos(u_1)} & 0 \\ -\frac{\sin(u_1)\sin(u_3)+\cos(u_1)\sin(u_2)}{\cos(u_1)\sqrt{1-\sin(2u_2)\sin(2u_3)}} & -\frac{\sin(u_3)}{\cos(u_1)\sqrt{1-\sin(2u_2)\sin(2u_3)}} & \frac{1}{\sqrt{1-\sin(2u_2)\sin(2u_3)}} \end{bmatrix} \quad (6)$$

Where  $u_1$ ,  $u_2$ , and  $u_3$  are the angles between the perfectly orthogonal coordinate system of the magnetic field and the coordinate system of the misaligned vector sensors as depicted in Figure 12 [9].

The bias error is a simple offset for each of the three internal sensors of the vector magnetometer. Since there are three such sensors in the vector magnetometer, three bias terms are used to construct a column vector ( $O$ ) that biases the magnetic field components into vector magnetometer readings and is given in Equation (7). This column vector can be subtracted from the vector magnetometer readings to correct the bias errors [9].

$$O = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \quad (7)$$

Together, these corrections can be applied to the vector measurements, as given in Equation (8), to transform the distorted raw magnetometer data ( $F$ ) into accurate vector magnetic field measurements ( $B$ ) [9].

$$B = P^{-1} \cdot S^{-1} \cdot (F - O) \quad (8)$$

In order to find the sensitivities, non-orthogonality, and bias correction terms that minimize the error of the calibrated measurements ( $B$ ), we fix the magnetometer in space and in a known constant magnetic field and vary the magnetometer's pitch and roll angles. Expected vector magnetometer measurements ( $B$ ) can be calculated by rotating the known constant magnetic field moment by the negative pitch and roll angles (i.e., if the magnetometer "pitches up" in the world frame, the magnetic readings should "pitch down" by the same angle in the sensor's frame). A least-squares optimization is then used to find the values of  $P^{-1}$ ,  $S^{-1}$ , and  $O$  that minimizes the error between the expected and calibrated observed measurements [9].

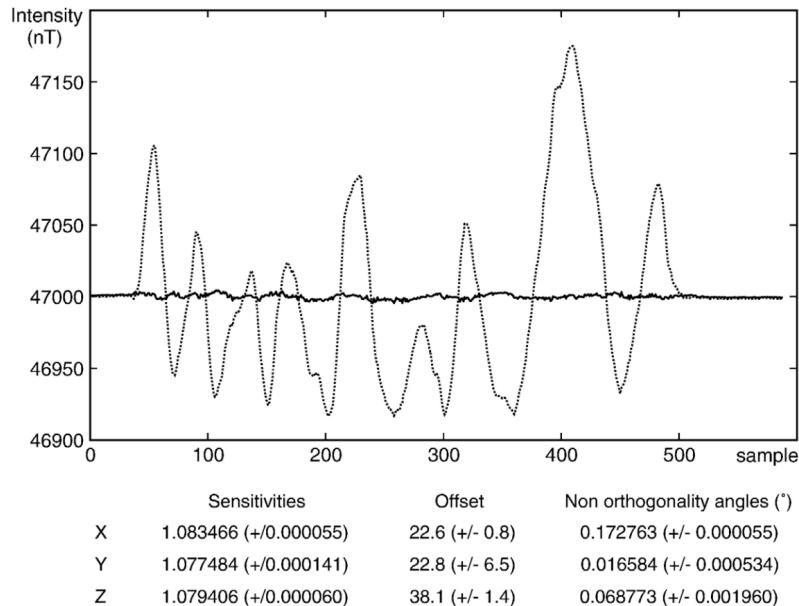


Figure 13: Example results of applying a spin test calibration to a fluxgate magnetometer and the calculated sensitivities, offsets, and non-orthogonality angles [9].

### 2.5.2 Tolles-Lawson Calibration

In order to reduce the body effects (as described in Section 2.2.2) seen by the survey scalar magnetometer, a Tolles-Lawson calibration can be conducted [1][15].

This typically consists of a box flight pattern at high altitude while pitching, rolling, and yawing the survey airplane. Each side of the box is split into three sections where a different angle is oscillated at a constant frequency and amplitude while the other angles remain constant. It is important that the box flight be conducted at a sufficiently high altitude so that the magnetic field remains constant across the perimeter of the flight path. This will ensure that all deviations in the magnetometer readings will be due to body effects induced by the angle oscillations and not changes in the external magnetic field. Alternatively, the flight can be conducted at a much lower altitude if in an area encompassed by a fully sampled aeromagnetic anomaly map. In this case, the changes in measurements due to sources of magnetic anomaly can be directly removed by sampling the map along the known flight path [1][15][5][18].

An example Tolles-Lawson calibration flight could be designed where the survey aircraft flies 90 km/h at 500 m altitude; conducts a box flight pattern where each side is 5.3 km long; and each side consists of 2 Hz,  $\pm 5$  degrees oscillations of first pitch, then roll, then yaw angles [1][15].

After a calibration flight is conducted, 18 Tolles-Lawson calibration coefficients ( $a_1, \dots, a_{18}$ ) are calculated. Of the 18, there are 3 permanent, 6 induced, and 9 eddy current magnetization coefficients [18]. The disturbance field generated by body effects are given by [15] as

$$B_{dist} = B_{perm} + B_{ind} + B_{eddy} \quad (9)$$

where the individual body effect terms are estimated by [15] as

$$B_{perm} = a_1 \cos(X) + a_2 \cos(Y) + a_3 \cos(Z) \quad (10)$$

$$\begin{aligned} B_{ind} = B_t & (a_4 \cos^2(Z) + a_5 \cos(X) \cos(Y) + a_6 \cos(X) \cos(Z) \\ & + a_7 \cos^2(Y) + a_8 \cos(Y) \cos(Z) + a_9 \cos^2(X)) \end{aligned} \quad (11)$$

$$\begin{aligned} B_{eddy} = B_t & (a_{10} \cos(X) \cos(\dot{X}) + a_{11} \cos(X) \cos(\dot{Y}) + a_{12} \cos(X) \cos(\dot{Z}) \\ & + a_{13} \cos(Y) \cos(\dot{X}) + a_{14} \cos(Y) \cos(\dot{Y}) + a_{15} \cos(Y) \cos(\dot{Z}) \\ & + a_{16} \cos(Z) \cos(\dot{X}) + a_{17} \cos(Z) \cos(\dot{Y}) + a_{18} \cos(Z) \cos(\dot{Z})) \end{aligned} \quad (12)$$

where  $B_t$  is the magnitude of the magnetic field and the direction cosines are calculated using the vector magnetometer readings as

$$\begin{aligned} \cos(X) &= \frac{T}{B_t} \\ \cos(Y) &= \frac{L}{B_t} \\ \cos(Z) &= \frac{V}{B_t} \end{aligned} \quad (13)$$

where T, L, and V are the transverse, longitudinal, and vertical components of the magnetic field in the vector magnetometer's coordinate frame [1][15].

In order to calculate the 18 coefficients ( $a_1, \dots, a_{18}$ ) for a given survey aircraft, the magnetic data from a calibration flight is band-pass filtered such that only effects of the maneuvers remain. A least-squares optimization is then used to find the coefficient values that minimize these effects.

After calculation of the Tolles-Lawson coefficients, it is customary to calculate a Figure of Merit (FOM) that signifies the quality of the calibration. This is done by flying a stationary cloverleaf flight pattern (see Figure 15) where the pitch, roll, and yaw angles are oscillated during each pass in the same exact manner as during the

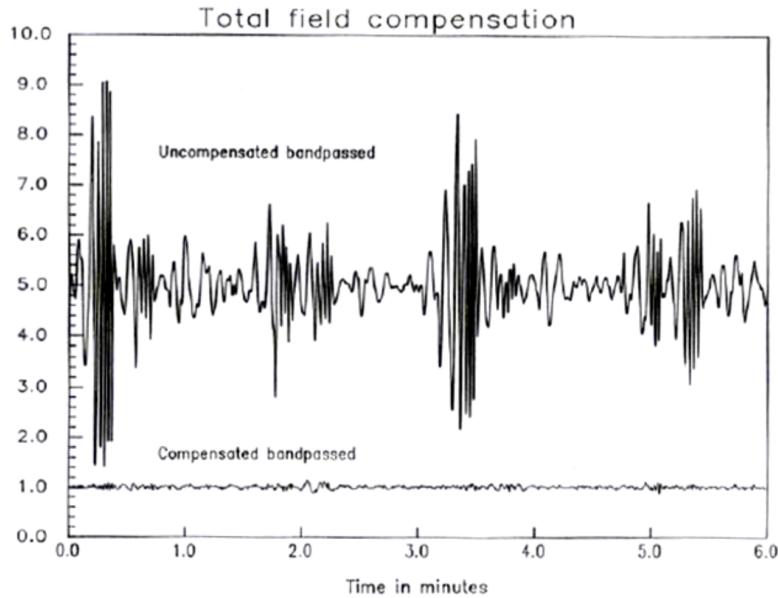


Figure 14: Example of scalar magnetometer readings before and after calibration using the Tolles-Lawson method [5].

box calibration flight. After compensating for temporal variations using a ground reference station and body effects using the optimized Tolles-Lawson coefficients, the total error is translated to a FOM [5]. According to [1], typical FOM values for geological prospecting companies are less than 1 nT.

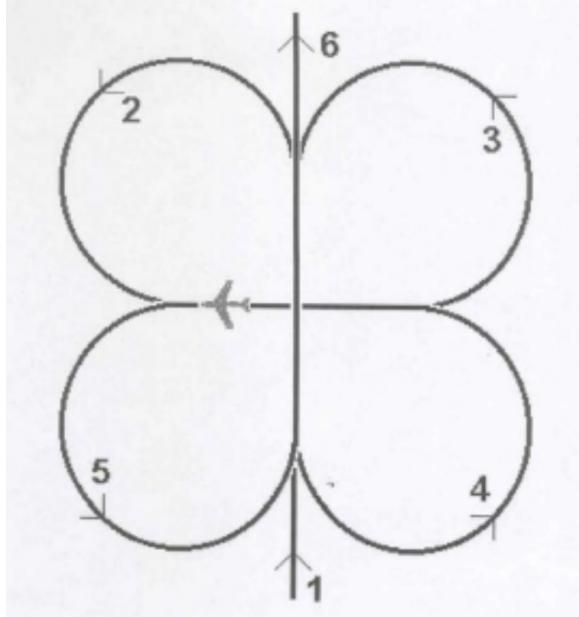


Figure 15: Example cloverleaf flight path used to determine calibration FOM [5].

## 2.6 Ground Reference Stations

In order to remove temporal effects, as described in Section 2.2.1, a ground reference station is used to collect scalar magnetic data concurrently with the survey flight near the survey area. The generally accepted maximum distance between the survey area and ground reference station is 100 km, but this distance should be minimized [11]. The difference between the Earth's core field magnitude at the location of the ground reference station and the station's readings are calculated over the duration. This difference signal is then subtracted from the survey flight readings to remove diurnal and space weather effects. To estimate Earth's core field magnitude at the ground reference station's location, the WMM or IGRF model may be used [13].

In addition to survey specific ground reference stations, there exists a global network of approximately 100 static magnetic observatories called INTERMAGNET as seen in Figure 16. High quality, 1 Hz timestamped vector and scalar magnetometer data from these observatories are available to the public daily via the organization's

home web-page (<https://intermagnet.github.io/>) [19]. INTERMAGNET data is potentially useful for removing temporal effects from aeromagnetic surveys if the distance between the survey site and the nearest INTERMAGNET observatory is reasonable.



Figure 16: World map of INTERMAGNET observatories as of July 2022. Sites marked in red are active and those marked in grey are inactive. Image clipped from <https://intermagnet.github.io/metadata/#/map>.

## 2.7 Survey Path Planning

Aeromagnetic surveys are typically flown in a grid-like search pattern of parallel flight lines and orthogonal tie lines as seen in Figure 17. The flight lines are used to capture the overall magnetic anomaly signature over the survey area [20]. In order for an aeromagnetic anomaly map to be fully sampled, the distance between flight

lines must be no more than “one-half the depth to the shallowest significant sources” [20], which, for MagNav, are surface sources [1]. Therefore, flight lines for MagNav surveys should be spaced no farther than one-half the minimum survey altitude above ground level (AGL).

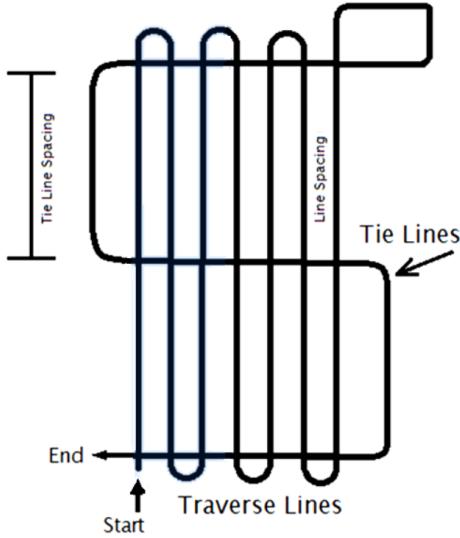


Figure 17: Example aeromagnetic survey flight path with dense, parallel flight lines and sparse orthogonal tie lines [1].

Tie lines, on the other hand, are used to compensate for corrugation and tilt in the flight line magnetometer data through a process called map leveling [1][5][13][20]. Since tie lines are used for map leveling and not for capturing the anomaly signature, tie line spacing is generally much larger than the flight line spacing. It is typical to have 8 or 10 times more tie line than flight line spacing [20].

## 2.8 Anomaly Map Leveling

As discussed in Section 2.7, traditional aeromagnetic anomaly map leveling uses a sparse set of tie lines flown orthogonal to the survey’s flight lines. It is assumed that the magnetic anomaly field remains constant for the duration of the survey flight at a

given location regardless of the heading of the aircraft. Therefore, where the flight and tie lines intersect (after subtracting the Earth's core field, temporal, and body effects), the anomaly value at that location should be equal for both passes. A polynomial fit using a least-squares approach is then applied to the difference between the flight and tie lines at the intersections. This error polynomial is then subtracted from the flight line data [1][13][21]. It is also common for micro-levelling techniques to be applied to the flight line data to ensure the map will be created with the lowest possible amount of distortion or tilt. Micro-levelling techniques are manually intensive and usually proprietary [1][13]. They are not discussed in detail here.

There are several issues with traditional tie-line based map leveling. One is that the requirement to re-fly the survey area increases the total time and cost of the survey. Another issue is that differences between the flight and tie lines at the intersections are generally larger than the estimated accuracy of the survey itself. This means there are larger, systematic issues with the survey design other than leveling errors. This could be due to imperfect calibration of the magnetometers, altitude differences between the flight- and tie-lines, or possibly errors in temporal effect corrections [21][22][23].

Modern map leveling techniques without the use of tie-lines are being developed with promising results [21][23][22]. For this research effort, two types of tie-line leveling techniques were explored along with one tie-line absent approach as given by Zhang et al. (2018) [22].

White and Beamish (2015) [21] leveled several aeromagnetic datasets using the long wavelength features of the map. First, the original flight line data was interpolated to the desired grid spacing and applied a two dimensional low-pass filter. The original and filtered flight line data were differenced. Next, this difference data was fitted by a first-degree polynomial using a least-squares approach. The polynomial was then used to subtract the short wavelength errors from the flight line data. A

similar leveling technique was then used to level the adjusted flight line data to the IGRF values across the survey area. The results of this map leveling technique was found to outperform traditional tie line map leveling [21].

Ishihara (2015) [23] developed a map leveling technique that takes advantage of neighboring flight line data across time. The correction term for each flight line sample is a weighted sum of all other samples. For this leveling technique, there are two cost functions that determine the sample weights: one cost function for the amount of time between samples and the distance between samples. This approach was found to reduce map leveling errors by up to a factor of 5 and accurately estimate temporal effects without a ground reference station. However, the performance is tightly linked to choosing appropriate cost functions, which may not be straightforward or easily implemented [23].

Zhang et al. (2018) [22] designed a new map leveling technique using the principal component analysis (PCA) of pseudo tie lines. In this method, it is assumed that the majority of the difference between any two adjacent flight lines is due to map leveling errors. This assumption holds true if the magnetic anomaly gradient is sufficiently low in the tie line direction [22].

To implement this approach, all flight lines ( $\mathbf{D} = [\mathbf{d}^0, \mathbf{d}^1, \dots, \mathbf{d}^L]$ ) are leveled to a single flight line. This reference line is chosen arbitrarily, although usually the first or last flight line in the survey is chosen for simplicity. The difference between the each adjacent flight lines are then computed ( $\Delta\mathbf{D}$ ). A set of pseudo tie lines are chosen that samples the difference data perpendicular to the flight line direction ( $\Delta\mathbf{D}_{tie}$ ) [22].

The location of these pseudo tie lines are chosen arbitrarily. However, due to the assumption that the line to line differences is primarily due to leveling errors, pseudo tie line locations should be chosen where the magnetic anomaly gradient in the tie

line direction is minimal. Also, in order to properly interpolate map leveling errors near the edges of the survey data, a pseudo tie line is typically placed at each end of the survey data set [22].

Next, the covariance matrix of the pseudo tie line dataset is calculated ( $\mathbf{C}_x$ ). The eigenvalue and rotation matrix ( $\Lambda$  and  $\mathbf{R}$ , respectively) of  $\Delta\mathbf{D}_{tie}$  are then determined by taking the singular value decomposition of  $\mathbf{C}_x$ ,

$$\mathbf{C}_x = \mathbf{R}\Lambda\mathbf{R}^T \quad (14)$$

The principal components ( $\Psi$ ) of the pseudo-tie-lines are then calculated,

$$\Psi = \mathbf{R}^T \cdot \Delta\mathbf{D}_{tie} = \begin{bmatrix} \Psi_1 \\ \Psi_2 \\ \vdots \\ \Psi_n \end{bmatrix} \quad (15)$$

where the individual  $n$  principal components are  $\Psi_1, \Psi_2, \dots, \Psi_n$  where the cumulative contribution rate of the  $m$ th principal component is given by

$$\delta_m = \frac{\sum_{j=1}^m \Lambda_{jj}}{\sum_{i=1}^n \Lambda_{ii}} \quad (16)$$

The set of principal components that have a cumulative contribution rate ( $\delta_m$ ) of 85% are then used to model the leveling errors at the flight and pseudo tie line intersection points ( $\widehat{\Delta\mathbf{D}_{tie}}$ ). It is important to note that Zhang et al. (2018) did not describe this cumulative contribution rate as a user defined parameter, therefore, the 85% value is assumed to be constant and used in all cases.  $\widehat{\Delta\mathbf{D}_{tie}}$  is then reconstructed using the smaller set of principal components,

$$\widehat{\Delta D}_{tie} = \mathbf{R} \cdot \Psi = \mathbf{R} \begin{bmatrix} \Psi_1 \\ \Psi_2 \\ \vdots \\ \Psi_m \end{bmatrix} \quad (17)$$

The leveling errors across all survey data points ( $\widehat{\Delta D} = [\widehat{\Delta D}^0, \widehat{\Delta D}^1, \dots, \widehat{\Delta D}^L]$ ) are then calculated by linearly interpolating  $\widehat{\Delta D}_{tie}$  in the flight line direction at all intermediate flight line sample locations. The leveling error values in  $\widehat{\Delta D}$  are then subtracted from the original survey data to produce leveled flight lines [22].

This PCA method was found to outperform polynomial map leveling with proprietary micro-leveling when applied to the same survey data set [22]. The method's superior performance and minimal user input makes it a prime candidate for MagNav aeromagnetic anomaly map leveling.

### III. Methodology

#### 3.1 Preamble

This chapter covers the methods used to collect, analyze, and process data for this research effort. Section 3.2 covers the process used to develop and analyze a novel approach to using magnetic reference station data. Section 3.3 discusses the methodology used to generate and process simulated aeromagnetic data. Finally, Section 3.4 details the equipment and procedures used to collect, process, and analyze real-world aeromagnetic survey data with a fixed wing unmanned aerial vehicle (UAV).

#### 3.2 Extended Reference Station Analysis

In an effort to eliminate the need for such a local magnetic ground reference station, analysis was done on concurrent data from several geologically separated International Real-time Magnetic Observatory Network (INTERMAGNET) magnetic observatories [19]. The aim of this study was to determine if temporal magnetic variations at a given survey area could be inferred from an INTERMAGNET observatory located over 100 km away with reasonable accuracy.

The two observatories in this study were the United States Geological Survey (USGS) INTERMAGNET sites in Boulder, Colorado and Fredericksburg, Virginia (approximately 2400 km distance apart). 1 Hz data from both of these sites collected during the 2019-09-12T00:00:00Z to 2019-09-12T23:59:59Z time frame were analyzed.

The premise of the study was that a hypothetical aeromagnetic survey was conducted over a 48 h period directly over the USGS observatory in Fredericksburg, Virginia. The data from that observatory serves as “truth” data for the temporal magnetic variation during the hypothetical survey. Furthermore, the USGS observatory in Boulder, Colorado would serve as the *extended reference station* who’s data

Table 2: Extended reference station analysis site location data. These values were obtained from the headers in the data files from each site.

<b>Site City and State</b>	<b>Geodetic Latitude (°)</b>	<b>Geodetic Longitude (°)</b>	<b>Altitude Above MSL (m)</b>
Boulder, Colorado	40.137	-105.237	1682
Fredericksburg, Virginia	38.205	-77.373	69

will be processed to infer the temporal magnetic variations at the *local reference station* in Fredericksburg, Virginia.

After an initial inspection of the observatory data, correlations between the data sets were identified and an approach to calibrate the extended reference station data to match the local reference station was developed. To do this, the data sets from each observatory were parsed and denoised. The noise removal process involved detecting and removing outliers in the exponential weighted forward and backward moving average of the observatory scalar measurements. Replacement values were then linearly interpolated where the outliers were removed, thus providing a smoothed set of scalar values.

Next, the International Geomagnetic Reference Field (IGRF) core field values for each observatory for the day of the hypothetical survey was calculated and subtracted from total field intensity values for their respective observatories. This provides the local temporal variations for both the extended and local reference stations over the course of the hypothetical survey.

The temporal variations of both observatories were compared. Analysis showed that the lower frequency content caused by the diurnal effect were similar but shifted in time proportional to the difference in observatory longitude. The high frequency content caused by space weather effects matched in both amplitude and time. Additionally, it was noted that the Boulder temporal variations could match more closely

the Fredericksburg variations if the Boulder data was optimally scaled and biased. It is important to note that there is less than 2 degrees of latitude difference between the Boulder and Fredericksburg observatories. Larger differences in latitude between stations may cause varying results, but are not investigated in this research effort.

As a result of the analysis, we applied various operations on the data from the Boulder temporal variations to match that of the Fredericksburg observatory.

First, the difference in longitude ( $\Delta\lambda$ ) was calculated as the difference between the extended reference station longitude ( $\lambda_e$ ) and local reference station longitude ( $\lambda_l$ ) as given in eq. (18).

$$\Delta\lambda = \lambda_e - \lambda_l \quad (18)$$

The difference in longitude was then divided by the Earth's angular rotation rate  $\omega_{\text{Earth}} = 0.004^\circ/\text{sec}$  to find the longitudinal time offset ( $t_l$ ) in seconds as given in eq. (19) as given in eq. (19).

$$t_l = \frac{\Delta\lambda}{\omega_{\text{Earth}}} . \quad (19)$$

The extended reference station temporal magnitude variation readings were then both high- and low-pass filtered with second order Butterworth filters and the resulting data sets saved separately. We found a common corner frequency of 0.33 cycles/h isolated the Earth rotation effects from the other common effects. The low-pass filtered temporal variations were then time shifted by the longitudinal time offset found from (19) and added to the high-pass filtered temporal variation. The output of this operation is the extended reference station data time-aligned to the local reference station.

Next, an optimization was run on the time-aligned extended reference station

data to find the best scale ( $a$ ) and offset ( $b$ ) that minimized the amount of error ( $e_{le}$ ) between the processed extended reference station and local reference station data. The error function minimized during this optimization is given in eq. (20)

$$e_{le} = (B_l - ((a \cdot B_{eta}) + b))^2 \quad (20)$$

where  $e_{le}$  is the minimized error,  $B_l$  is the local reference station temporal variation readings,  $a$  is the tuned scale parameter,  $b$  is the tuned offset parameter, and  $B_{eta}$  is the time-aligned extended reference station temporal variation readings. Once the optimal scale and offset parameters were found, they were applied to the time-aligned extended reference station temporal magnitude variation readings to find the inferred local temporal variations. The results of this process was then compared against the true local temporal variations by calculating a Root Mean Square Error (RMSE) Figure of Merit (FOM). This process was then repeated using the same optimal scale and offset parameters on the data from the same INTERMAGNET sites on the following day to test repeatability. The results of these tests can be found in Section 4.2.

### 3.3 Simulated Aeromagnetic Surveys

Before taking real-world measurements, software simulations of aeromagnetic surveys over a predefined magnetic anomaly field were created using Python. These simulated surveys were then processed into maps using several different procedures. Applying various map making techniques on the same survey data with an associated “truth map” provided insight on how each choice in the map making process affected the accuracy of the final map.

### 3.3.1 Simulated Magnetic Anomaly Field

The generated magnetic anomaly field used to conduct the simulated aeromagnetic surveys was modeled as a bivariate Gaussian distribution with predefined covariance and scaling. A modified Gaussian distribution was chosen for the magnetic anomaly field model since it is both twice continuously differentiable such as moments found in real-world anomaly maps and because such distributions are easy to generate. The final magnetic anomaly model ( $B_{sa}$ ) used is given in eq. (21) and a plot of the model is shown in Figure 18.

$$B_{sa} = 20 \cdot \mathcal{N} \left( \begin{bmatrix} -77.373 \\ 38.205 \end{bmatrix}, \begin{bmatrix} 1e^{-6} & 0 \\ 0 & 2e^{-6} \end{bmatrix} \right) \quad (21)$$

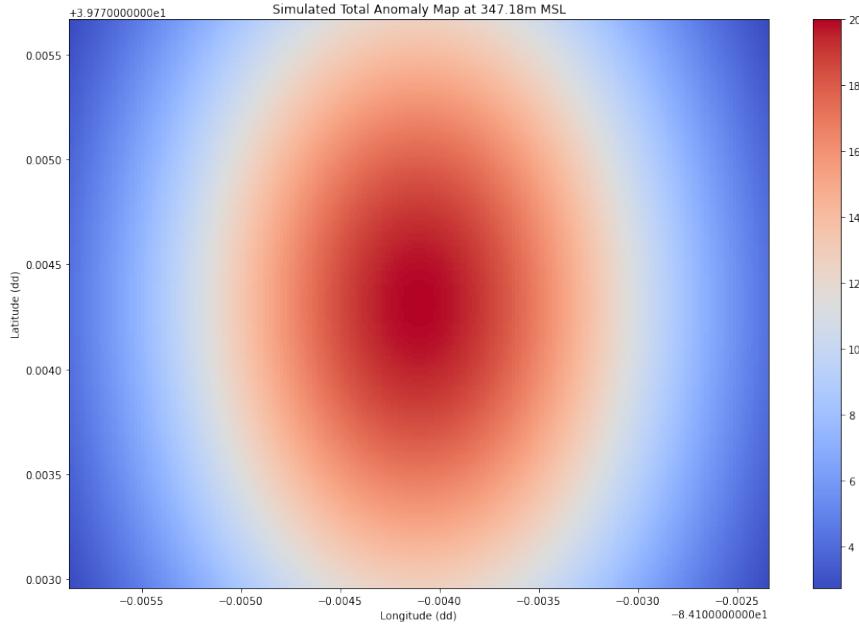


Figure 18: Magnetic anomaly truth map for simulated aeromagnetic surveys.

The simulated magnetic anomaly field model was then sampled at finely spaced grid intervals with each sample associated with a hypothetical geodetic latitude, longitude, and altitude location. The samples and associated location data were then

compiled into a magnetic anomaly truth map. This map was used to evaluate the accuracy of the magnetic anomaly reconstruction software used for the real-world surveys conducted for this research effort.

### 3.3.2 Simulated Survey Measurements

A simulated flight path was then constructed based on a number of flight and sensor performance parameters similar to those expected in the real-world surveys. These parameters and their associated values are given in Table 3. Based on these parameters, a simulated survey flight path and magnetometer measurement locations was generated as shown in Figure 19 and the resulting timestamped magnetic anomaly measurements were recorded as shown in Figure 20.

Table 3: Simulated survey parameters.

Parameter	Value	Unit
Flight start date/time	2019-09-12T04:40:00Z	UTC
Flight speed	20	m/s
Survey altitude	347.18	m above MSL
Flight line spacing	15	m
Tie line spacing	75	m

### 3.3.3 Simulated Noise and Distortion

In order to transform the simulated magnetic anomaly measurements into more realistic, raw magnetometer measurements one would expect to see from a real-world survey, simulated noise and distortions were added. This included temporal variation, a constant bias due to the Earth's core field, and varying levels of sensor noise due to bias and random Gaussian noise.

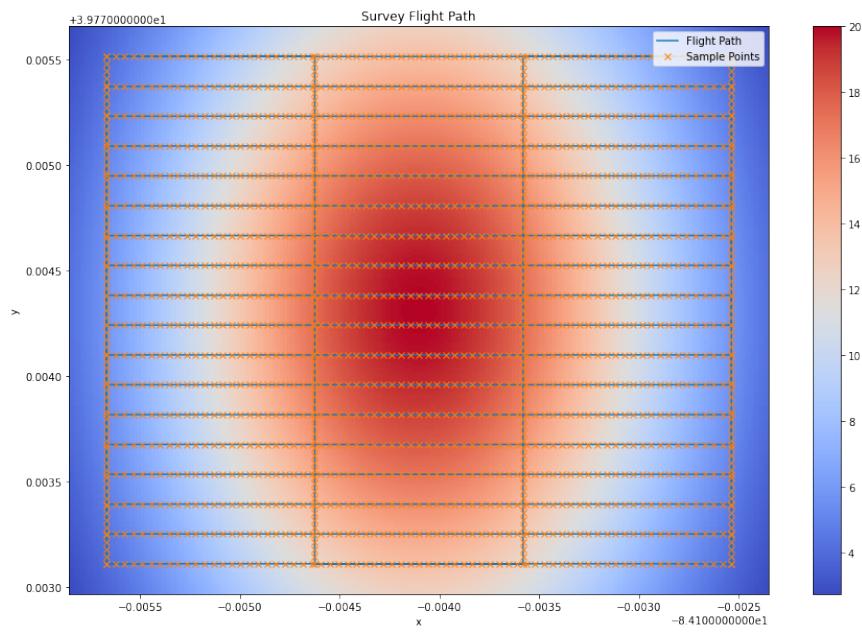


Figure 19: Simulated survey flight path and magnetometer measurement locations.

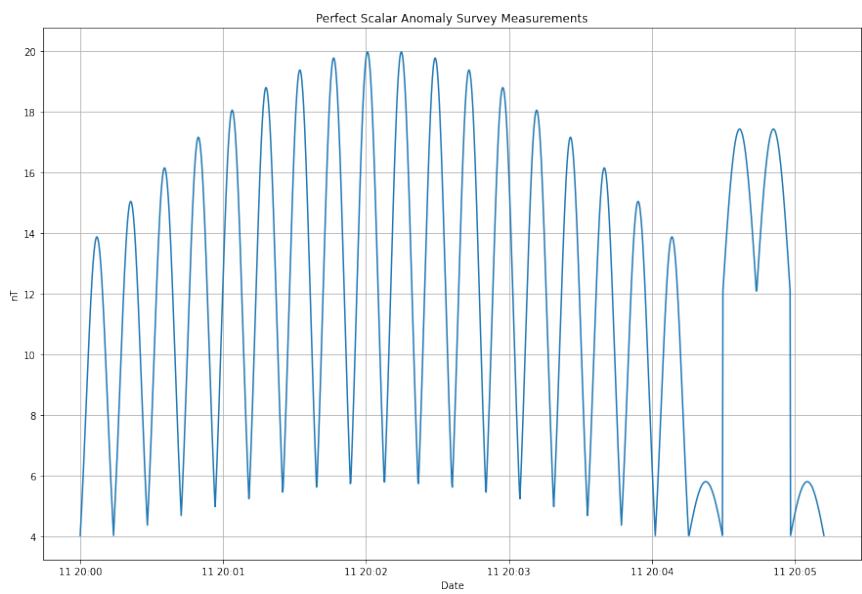


Figure 20: Simulated survey magnetic anomaly measurements without noise or distortion.

For temporal variation, the magnetic reference data from the Fredericksburg observatory was used. Data from the observatory was collected for the same date and time as the simulated flight.

Next, the Earth's core field component at the observatory was estimated using the IGRF model and removed from the raw measurements, resulting in the true temporal variation as seen in Figure 21. These temporal variation were then directly added to the simulated magnetometer measurements. For the bias due to Earth's core field, the core field magnitude was estimated using the IGRF model at the center location of the survey area. This value was then added to all simulated magnetometer measurements. Aircraft body effects and leveling errors were assumed to be negligible and, as such, were not added to the simulated measurements. While ignored for simplicity, the simulation code does allow users to add these effects if desired.

Lastly, different levels of magnetometer noise was added to the measurements. The first simulated map was generated using data with no sensor noise. The second map was generated with no sensor noise, but temporal variation was not accounted for. The third map also used perfect sensor measurements, but the temporal variation was accounted for using an extended reference station as described in Section 4.2. The next three simulated maps accounted for temporal variations using the local reference station data, but the flight magnetometer scalar data included various levels of bias. This included maps with 0.1, 1, and 10 nT of bias in the sensor measurements. Similarly, the last three simulated maps were made with random Gaussian noise added to the flight magnetometer scalar data with standard deviations of 1, 10, and 100 nT. The resulting maps can be found in Section 4.3.

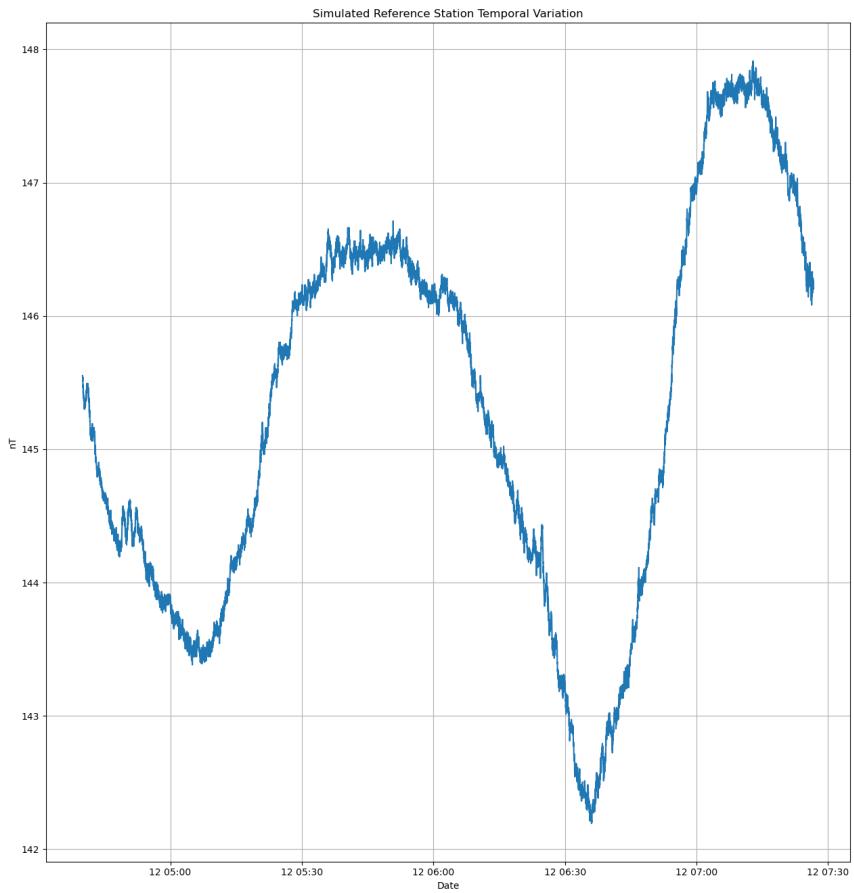


Figure 21: Temporal variation from the INTERMAGNET observatory in Fredericksburg, Virginia. This dataset was used as the true temporal variation when simulating survey data.

### 3.3.4 Map Creation from Simulated Measurements

To create a map from the simulated measurements, the noise and distortions (as detailed in Section 3.3.3) were removed and the resulting magnetic anomaly values were interpolated between survey sample locations.

In order to evaluate how different noise source affects the overall accuracy of the interpolated map, multiple additional maps were created with certain noise sources included in the estimated magnetic anomaly data. More specifically, one map was created with no effort to remove temporal effects. Another map was created using the Extended Reference Station Modeling (ERSM) approach (as discussed in Section 3.2) to remove the temporal effects using calibrated reference station data from the INTERMAGNET observatory in Boulder, Colorado.

Another set of maps were created with additional magnetic noise. A set of three maps were created with different biases applied to the simulated magnetometer measurements representing a possible mixture of both sensor and diurnal biases. These biases included 0.1 nT, 1.0 nT, and 10 nT. Similarly, a set of three maps were created with differing levels of additive white Gaussian noise with standard deviations of 1.0 nT, 10 nT, and 100 nT representing sensor and space weather effect noise.

In all cases, each map was generated using a 2D radial basis function interpolation function and compared to the original truth map of the simulated magnetic anomaly field. A RMSE value was calculated as a comparison between each interpolated and truth map for further analysis.

## 3.4 Real-World Aeromagnetic Surveys

Several real-world aeromagnetic surveys were conducted to test the effectiveness of our custom Python library for creating magnetic navigation (MagNav) maps which was developed as part of this research effort. These surveys also allowed us improve

the high-level aeromagnetic anomaly map making process as put forth in Appendix C. In all, 4 overlapping surveys were conducted at the Camp Atterbury UAV flight test range near Edinburgh, Indiana. To accomplish these surveys, high accuracy scalar and vector magnetometers were used with a small UAV that was modified and calibrated for magnetic surveyance. A local magnetic ground reference station with a high accuracy scalar magnetometer was also used to account for temporal magnetic variations. Survey data post processing was done in Python using the MAMMAL library.

### 3.4.1 UAV Design

A modified SIG Rascal model airplane flew the real-world aeromagnetic surveys (as seen in Figure 22) with magnetometers mounted in the tail for survey data collection. The vehicle's construction consists primarily of balsa wood with a MonoKote covering and a twin piston, gas-fueled engine.



Figure 22: SIG Rascal UAV modified for aeromagnetic surveyance.

In order to mitigate body effects on magnetic measurements as much as possible, ferrous metal parts originally used on the UAV were replaced with either non-metallic or aluminum parts. Additionally, the elevator and rudder servos were moved from their original mounts (directly beneath the horizontal stabilizer) to the center of the aircraft. Servo connections to their respective control surfaces were used with Kevlar rope in a “pull-pull” configuration. This modification increased the distance between the servos and the magnetometers mounted inside the tail, mitigating body effects and electromagnetic noise on the magnetometer measurements.



Figure 23: View of SIG Rascal UAV magnetometers from within the vehicle. Camera was placed in the fuselage directly under the main wings and is facing towards the tail of the UAV.

Several sensors and processors were used on the UAV. We employed a Hex Pix-hawk 2.1 autopilot for control during the survey flights. In addition to the autopilot, an NVIDIA Jetson Xavier NX Developer Kit performed logging for all sensor data, creating a combined log file for post flight analysis. We used a Geometrics MFAM magnetometer and a TwinLeaf VMR magnetometer for magnetic field sensing and compensation. Both magnetometers were mounted in the tail of the aircraft to minimize magnetic noise during the survey. Other electronics on-board include a wing



Figure 24: Tail-mounted SIG Rascal UAV magnetometers.

mounted differential pressure airspeed sensor and two Global Navigation Satellite System (GNSS) receivers (one connected to the autopilot for navigation and the other connected to the datalogger for data timestamping).

The MFAM is a dual sensor with both an optically pumped caesium scalar magnetometer and a magnetoresistive vector magnetometer. The MFAM was chosen for its high accuracy, low noise level, and small package [8]. The scalar data from this magnetometer was processed to create the magnetic anomaly map values and the vector data served as a backup sensor for calibrating the scalar data. The VMR is a 3 axis magnetoresistive vector magnetometer that was chosen for its relatively low price for performance plus its small package size [24]. The data from the VMR vector magnetometer were to calibrate the MFAM scalar data using the Tolles-Lawson method. Both magnetometers were mounted in a combined, 3D printed mount similar to that in Figure 25.

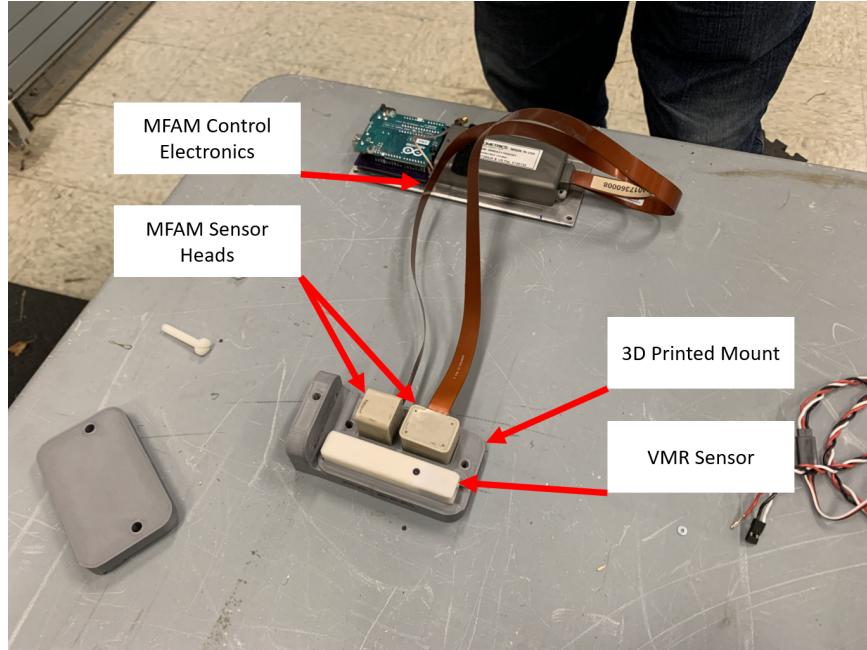


Figure 25: MFAM and VMR magnetometers in their combined, 3D printed mount.

### 3.4.2 UAV Survey

Before choosing a final mount location on the UAV for the magnetometers, a magnetic survey of the aircraft was conducted and analyzed using a spare MFAM magnetometer. This survey was conducted by taking the UAV outdoors to a magnetically quiet area between the Air Force Institute of Technology (AFIT) buildings and east parking lot. First, a baseline scalar reading of the area is recorded without the presence of the UAV. Next, the UAV was placed on the ground where the baseline measurement was recorded. Then, more scalar readings were collected at several places around the UAV including at the nose, main landing gear, center of the left main wing, trailing edge of the left main wing root, tail boom, and trailing edge of the elevator. Scalar readings were also recorded at one servo both at idle and under load. All readings were then compared to the baseline magnetic field value and the location with the least perturbed reading was chosen for the location of the magnetometer mount. The XYZ distances in the UAV's body frame between the GNSS receiver

antenna and magnetometer mount location was then measured and recorded. This lever arm was found to be  $-0.800\text{ m}$  in the x-direction,  $-0.051\text{ m}$  in the y-direction, and  $0.064\text{ m}$  in the z-direction. The x-axis points through the nose, the y-axis points through the right wingtip, and the z-axis points through the bottom of the UAV.

### 3.4.3 MFAM Sensor Head Meshing

Since the MFAM scalar magnetometer has two sensor heads mounted mutually orthogonal in the UAV, the valid readings of each sensor head (readings within the IGRF estimated core field  $\pm 500\text{ nT}$ ) were averaged together. When a sample from one of the sensor heads was invalid, an estimated sample was generated by linearly interpolating between valid samples of the same sensor head. This was done to remove large spikes and dead-zone artifacts from the dataset before further processing. The output of this operation was a single *meshed* magnetic field estimate for each set of MFAM sensor head measurements throughout the collects. This set of meshed scalar values were then used for all subsequent processing.

### 3.4.4 UAV Calibration

In order to estimate the UAV body effects on the scalar magnetometer readings, a stationary Tolles-Lawson calibration was performed. While calibration for aeromagnetic vehicles are usually conducted during specialized high altitude flights, the UAV used in this research was small and light enough for a stationary calibration. This was chosen over a flown calibration because it was assumed to generate better results having theoretically zero change in external magnetic field for the duration of the maneuver.

The stationary Tolles-Lawson calibration was performed with the UAV in a magnetically quiet area on the Camp Atterbury UAV airstrip. Although the vehicle was

not fueled, all other aspects of the UAV’s configuration mirrored that of when flying a survey and all avionic electronics were powered on during the calibration. After the UAV was ready for calibration, everyone assisting in the calibration removed all metal and electronics from their persons. Next, the UAV was picked up by one researcher near the main wing and another researcher held the UAV steady around the tail where both the vector and scalar magnetometers were mounted. The researcher holding the UAV near the main wing performed the Tolles-Lawson pitch, roll, and yaw oscillations in 8 directions separated by  $45^\circ$ . Each oscillation had a frequency of approximately 1 Hz and amplitude of approximately  $20^\circ$ . The researcher holding the UAV at the tail ensured that the location of the sensors changed minimally throughout the calibration.

The logged data from all sensors were then time-aligned by interpolating all samples against a common set of timestamps. Additionally, the scalar magnetometer values were obtained by meshing the MFAM sensor head readings (as described in Section 3.4.3). The time-aligned and meshed sensor data was then processed to find the optimal Tolles-Lawson calibration parameters and then applied to the survey datasets for removal of the UAV’s body effects. TwinLeaf and MFAM vector magnetometers were used independently to find the direction cosines of the UAV during the calibration. The meshed sensor head readings and direction cosines were then used to find the optimal Tolles-Lawson coefficient values.

All possible combinations of vector magnetometer and Tolles-Lawson coefficient subsets were tested to see which combination produced the most accurate results. Calibration accuracy was calculated by first applying the optimal Tolles-Lawson coefficient values to the original averaged scalar values. Next, a bias was added to the calibrated scalar values such that both the original and calibrated datasets had identical mean values. Lastly, a RMSE value was calculated between the original and biased

calibrated datasets. The results of this calibration can be found in Section 4.4.2.

### 3.4.5 Survey Data Collection

Real-world data was collected in an attempt to accurately map the magnetic anomalies over the Camp Atterbury UAV range. This was done over the course of 4 survey flights of various sizes over the same general area. Each survey was flown at an altitude of 400 m above MSL with flight lines in the North-South direction, flight line spacing of 200 m, and 3-5 evenly spaced tie lines in the East-West direction. The first survey flight was conducted directly over the Camp Atterbury UAV range airstrip on 3 November 2022 and was approximately 1 km by 1.5 km in size. The second survey flight was a repeat of the first survey. The third survey extended the original survey area by about 1 km to the south and the fourth survey likewise extended the original by about 3 km to the south. The second through fourth survey flights were all flown on 8 November 2022. In between the second and third survey flights, a ground calibration was performed (as specified in 3.4.4) on the Camp Atterbury UAV runway.

During each survey, the UAV ground control station remained at approximately 39.3432672° Latitude and -86.0094733° Longitude and the magnetic ground reference station at 39.342137° Latitude and -86.009226° Longitude. The approximate locations of both can be seen in Figure 26. The ground control station consisted of a truck, full sized trailer, external generator, and all electronics necessary to fly the survey UAV. The ground reference station was used to collect against temporal magnetic effects during each of the survey and consisted of one MFAM scalar magnetometer, a solar powered generator, and data logging electronics.

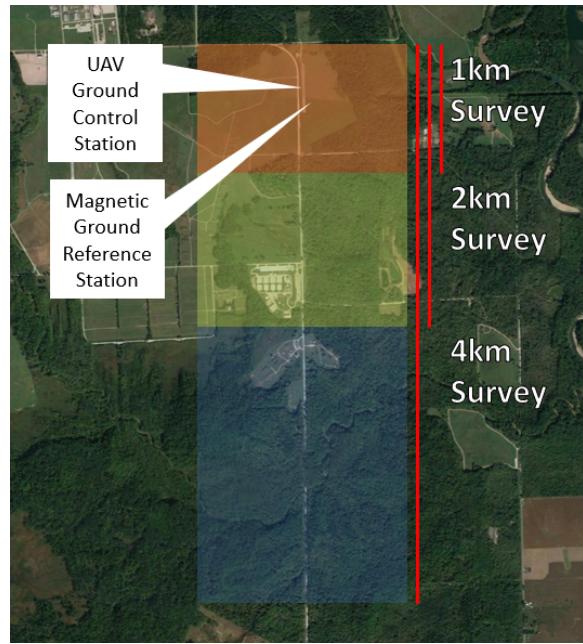


Figure 26: Approximate survey, UAV ground control station, and magnetic ground reference station locations for the Camp Atterbury UAV range.

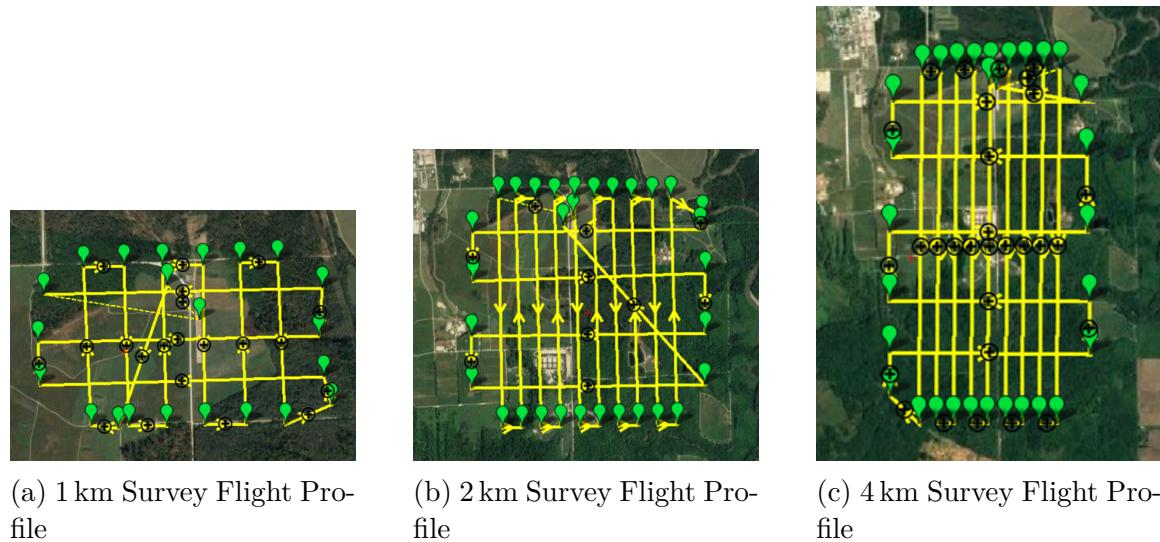


Figure 27: The three flight profiles used for the aeromagnetic anomaly surveys at the Camp Atterbury UAV range.

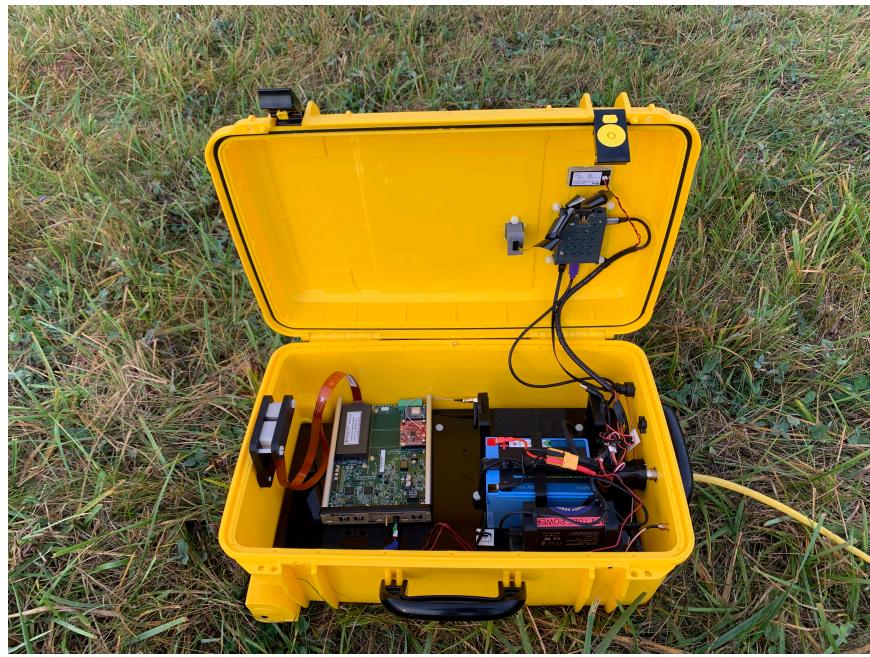


Figure 28: Magnetic ground reference station enclosure interior.



Figure 29: Magnetic ground reference station enclosure, generator, and solar panels placed next to the Camp Atterbury UAV runway

### 3.4.6 Survey Data Preparation

Three logs of sensor data were simultaneously recorded for each survey. The on-board Jetson Xavier generated a Lightweight Communications and Marshalling (LCM) log containing data from both the TwinLeaf and MFAM that were mounted on the UAV. Also, the UAV Pixhawk autopilot generated a comma-separated values (CSV) file containing vehicle geolocation, orientation, and other flight sensor data. Lastly, the magnetic ground reference station generated a CSV file containing all MFAM development kit magnetometer data.

The magnetic ground reference station CSV files were parsed and reformatted into an easier to use, tabular format with customized column names. Additional columns were added to note the ground reference station's location, sensor head valid flags, IGRF estimated core field values, and a final column for the *meshed* sensor head readings. The process of meshing the MFAM sensor head readings is described in Section 3.4.3. Then, a low pass filter with a cutoff frequency of 0.5 Hz was applied to the *meshed* sensor head readings. Lastly, the entire dataset was down-sampled to 1 Hz and saved as a separate CSV file.

Similarly to the magnetic ground reference station CSV files, the UAV Pixhawk CSV files were parsed and tabulated for ease of use. During this process, the date, time, pitch, roll, azimuth, geodetic latitude, longitude, and altitude values were extracted from the log and saved in a separate, tabular CSV file.

Unlike the CSV logs, the UAV LCM logs are streams of individual LCM binary packets where each packet contains different sensor values at various times in the flight. Magnetometer, accelerometer, barometer, and temperature data from the on-board MFAM and TwinLeaf sensors were included in each LCM log. After parsing a given LCM log, each sensor's data was individually saved as a tabular CSV file where the file name was the respective sensor name.

A flight/tie line CSV was then manually generated for each survey. This CSV denotes the start and stop geodetic latitude and longitude coordinates, line type, and line number of all flight and tie lines. The UAV Pixhawk mission waypoint files served as useful references during this process.

A combined log of all flight data for each survey was created to prevent the need to re-parse the raw data. This was done by first parsing the converted UAV CSV files containing the Pixhawk geolocation, MFAM scalar magnetometer, MFAM vector magnetometer, and TwinLeaf vector magnetometer data. The IGRF estimated core field magnitude was then calculated at each sample location and time in Coordinated Universal Time (UTC). Next, valid flags were generated for each MFAM sensor head sample. As described in Section 3.4.3, a sensor head sample is considered valid if it is within  $\pm 500$  nT of the IGRF estimated core field magnitude. Then, all valid samples for each sensor head were linearly interpolated at the MFAM sample rate and low-pass filtered with a cutoff frequency of 5 Hz. All parsed, calculated, and derived data fields were then linearly interpolated to 10 Hz. This 10 Hz dataset served as the fundamental basis of the combined log.

In order to calculate the exact geodetic latitude, longitude, and altitude locations of the magnetometers, the XYZ offset between the GNSS receiver antenna and magnetometer mount location (as described in Section 3.4.2) was rotated according to the UAV's orientation and added as an offset to the geodetic location of the UAV's GNSS antenna location. These magnetometer locations were then added to the combined flight log.

The geodetic location of the magnetometers was then compared to the given survey flight/tie line CSV. The sample closest to each waypoint was then marked as a flight/tie line start/stop point. All samples between a line's start and stop samples were marked with the corresponding line number and line type. Any samples not

associated with a flight or tie line were given a default line type value which was different from that of a flight or tie line. The line type and number for each sample were then added to the combined flight log. The log was then saved as a tabular CSV for further processing.

### 3.4.7 Noise and Distortion Removal

Several steps were taken to remove unwanted noise and distortion. The first step was to minimize the heading error of the MFAM scalar magnetometer. This was accomplished by averaging the low pass filtered valid measurements from each sensor head as described in Section 3.4.6. The averaged sensor head measurements were then calibrated using the optimal Tolles-Lawson coefficients and vector magnetometer data as determined in Section 3.4.4. The calibrated scalar data was then forced to retain the same mean as the uncalibrated scalar data.

Next, the magnetic ground reference data was processed to find the temporal variation during the given survey. This variation was then subtracted from the calibrated scalar data. To do this, all large spikes in the estimated temporal variation datasets were removed. This was done by finding averaging the forward and backward exponential weighted moving average (EWMA) of the reference magnetometer samples with a span of 10 samples each. Then, all reference magnetometer samples greater than 0.5 nT from the averaged EWMA was rejected and replaced by linearly interpolating the valid samples. Next, the IGRF core field value was subtracted from the de-spiked ground reference station scalar magnetometer data to obtain the temporal variation. The temporal variation data was then interpolated at the timestamps of the flight data and subtracted from the flight scalar magnetometer readings.

Additionally, the core field component was removed by subtracting the IGRF magnitude values calculated at the location of each sample. Then, a low pass filter

was applied to the processed scalar data to find the estimated magnetic anomaly values. The low pass filter's cutoff frequency was calculated with Equation (22), where  $f_{max}$  is the maximum expected magnetic frequency content (Hz) due to the anomaly field,  $v_{max}$  is the maximum ground speed (m/s), and  $a_{min}$  is the minimum altitude (m) above ground level (AGL) seen during the entire survey flight.

$$f_{max} = \frac{v_{max}}{a_{min}} \quad (22)$$

The final step before pixel interpolation was flight line leveling. The dataset of each survey was independently leveled using several techniques including:

- Leveling using pseudo tie lines with principal component analysis (PCA)
- Leveling each flight line individually against all intersecting tie lines
- Leveling all flight lines against all tie lines simultaneously

The PCA based pseudo tie line leveling approach is explained in detail in Section 2.8. Each PCA leveled map was generated using 2 pseudo tie lines where one was placed a quarter of the north-south distance from the north end of the survey area and the other three quarters south of the north end. The reference flight line was always the first flight line flown in the survey.

The individual flight line leveling technique calculates leveling corrections for each flight line independent of the others. In this process, the difference between the flight line to be leveled and all intersecting tie lines are calculated at the intersection points. Then, the coefficients of a first order polynomial are optimized using a least-squares approach. This first order polynomial maps the location of each flight line sample to a correction value such that the total error between the corrected flight line samples and intersecting tie lines are minimized. This process is then repeated for all remaining flight lines.

The simultaneous flight line leveling approach is similar to the individual flight line approach. However, instead of generating a single first order correction polynomial for each individual flight line, a single “plane of best fit” matrix is generated. This matrix is calculated by finding the least-squares matrix that maps the line intersection locations to the difference in magnetic intensity at those intersection points. This matrix is then used to map all flight line sample locations to a plane of correction values that minimizes the difference between the corrected flight line and tie lines values.

### 3.4.8 Map Interpolation

Once all sources of noise and distortion were removed and flight lines were leveled, all scalar magnetic anomaly datasets were spatially interpolated to a finely-spaced rectangular grid. Additionally, the non-leveled dataset for each survey was similarly interpolated for reference. In every case, a radial basis function interpolator was used to evaluate all map pixel values and a final 2D low pass filter was applied with a cutoff wavelength slightly greater than the survey altitude AGL in both the North and East directions. Lastly, each map was exported as a GeoTIFF with a format as given in Appendix D. In all, there were 3 total maps per survey, each with anomaly data leveled differently for comparison.

After interpolation, analysis was done to generate a corrugation FOM (in units of  $dB^{-1}$ ) for each generated map for each survey. It is important to note that higher values of corrugation FOM means less corrugation is present in the map. Corrugation is striping in a magnetic anomaly map in the tie line direction and is generally caused by sensor heading errors [5][13].

This FOM was calculated by first applying a high pass filter to the given map in the tie line direction where the cutoff wavelength was 380 m (slightly less than the

survey altitude AGL). The filtered map was then averaged in the flight line direction to produce a single waveform. This waveform served to estimate the overall corrugation in the map due to heading error and potentially other factors. Next, a fast Fourier transform (FFT) of the waveform was generated and the largest magnitude within the band given by Equation (23) was selected. In Equation (23),  $B_c$  is the band of corrugation wave-numbers and  $a_{agl}$  is the survey altitude above AGL in meters. The inverse of the largest magnitude within  $B_c$  was then set as the map's corrugation FOM ( $dB^{-1}$ ) where larger values correspond to less average corrugation.

$$\frac{1}{a_{agl}(1 + 0.05)} \leq B_c \leq \frac{1}{a_{agl}(1 - 0.05)} \quad (23)$$

### 3.4.9 Map Comparison

The corrugation FOM for each generated map were compared. The map with the highest corrugation FOM (least corrugation present) for each survey was selected for further analysis and comparison. Of these maps, the Pearson correlation coefficients of the scalar and gradient rasters were calculated between the overlapping portions of the two selected 1 km survey maps and also between the selected 2 and 4 km survey maps. The Pearson correlation coefficients of the scalar and gradient rasters were also calculated between the map with the highest corrugation FOM among both 1 km surveys and the map with the highest corrugation FOM among the 2 and 4 km surveys as given in Chapter IV.

## IV. Results and Analysis

### 4.1 Preamble

This chapter details the results and analysis of the proposed Extended Reference Station Modeling (ERSM) methodology and overall results for simulated and real aeromagnetic surveys. Section 4.2 covers the results of the ERSM methodology. Section 4.3 discusses and analyzes the results of all simulated surveys. Finally, Section 4.4 details the results and analysis of the real-world aeromagnetic surveys.

### 4.2 Extended Reference Station Analysis Results

When testing the novel ERSM, the two raw scalar magnitude datasets from the International Real-time Magnetic Observatory Network (INTERMAGNET) sites at Boulder and Fredericksburg were compared as seen in Figure 30. With no processing, the Root Mean Square Error (RMSE) between the two datasets was 267.32 nT. Most of this error was due to a bias between the datasets. To quickly remove the bias and more easily analyze the datasets, both datasets were biased such that the first samples started at zero. The resulting plot can be seen in Figure 31.

With both initial samples starting at zero, the new RMSE between the dataset was greatly improved to 10.87 nT despite two large, low frequency periodic swings of approximately 25 to 30 nT. These swings occurred once per day in each dataset and are due to the diurnal effect as described in Section 2.2.1. It is easy to see from Figure 31 that much of the diurnal swings overlap, but the Fredericksburg dataset depicts the diurnal swing occurring about 2 to 3 h before the overlapping swings in the Boulder dataset. Adding a time shift on the Boulder dataset by accounting for an approximate difference in longitude of about  $27.86^\circ$  and Earth's rate of rotation (approximately  $15^\circ/\text{h}$ ), the overlap between the two datasets increased.

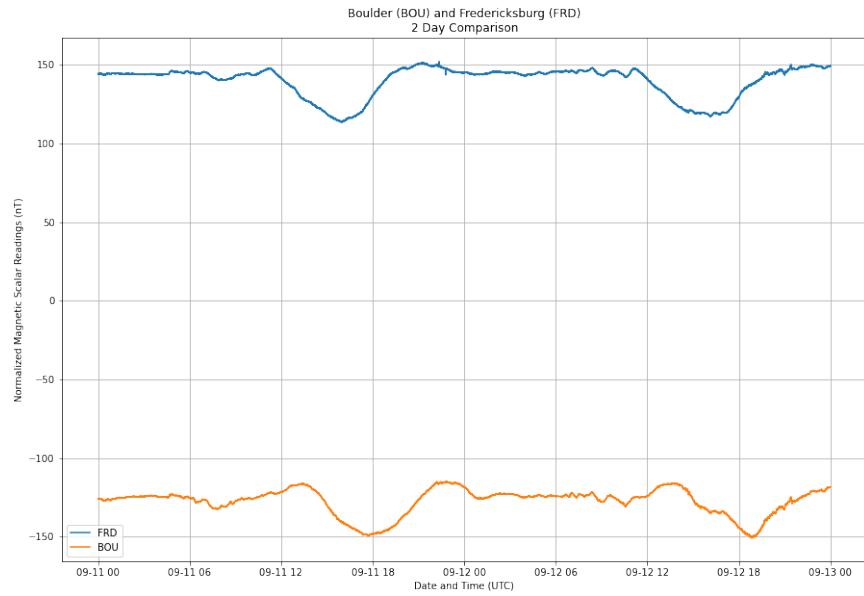


Figure 30: Comparison of raw scalar magnitude values from Boulder and Fredericksburg INTERMAGNET reference stations over a 2 day period.

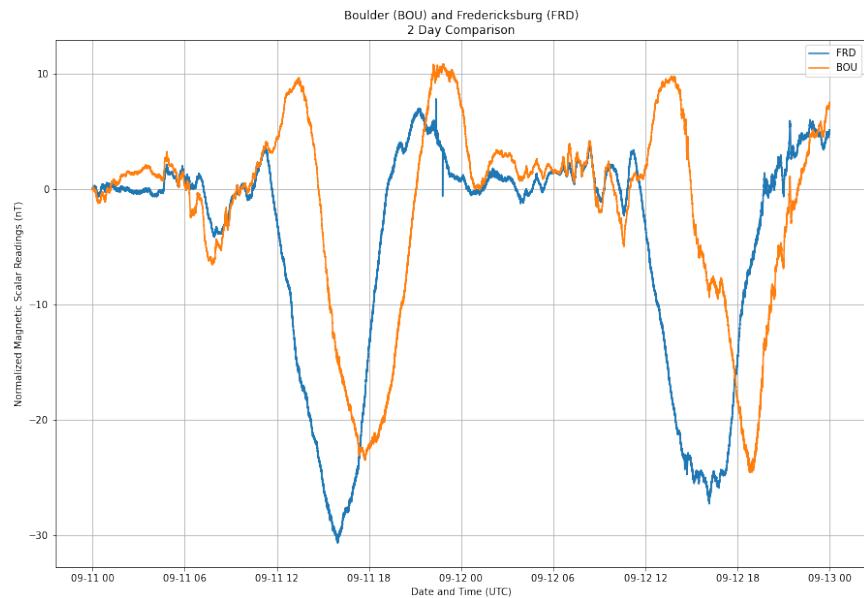


Figure 31: Comparison of zero-starting scalar magnitude values from Boulder and Fredericksburg INTERMAGNET reference stations over a 2 day period.

It was also realized that while the larger, lower frequency content was misaligned between the raw datasets, much of the high frequency content matched up to a bias factor. These variations are highly likely to be caused by space weather effects as described in Section 2.2.1.

After shifting the low frequency content due to the diurnal effect and preserving the alignment of the high frequency space weather noise, a least-squares optimization was used to find optimal scale and offset parameters as described in Section 3.2. The parameters and their calculated values are found in Table 4. After these parameters were applied to the frequency-based time-shifted Boulder dataset, the RMSE difference between the two datasets dropped from 266.88 nT to 4.63 nT as seen in Figure 32.

Table 4: Optimal ERSM parameters for Boulder to Fredericksburg.

Parameter Type	Parameter Value
Scale	1.09
Offset	277.69 nT

This ERSM method was then applied to the data from the same INTERMAGNET sites over a 24 h period following the originally analyzed datasets. This process also used the same scale and offset parameters as found in Table 4. The raw scalar data comparison can be seen in Figure 33 and the result of the ERSM process can be seen in Figure 34. The difference between the estimated and true temporal variation on the following day’s worth of data remained small and had a maximum instantaneous error of about 10 nT, an overall RMSE of about 3.64 nT and, more importantly, had an error standard deviation of 3.24 nT, despite the two sites being approximately 2400 km apart.

It is important to note, however, that the accuracy of this approach varies with the

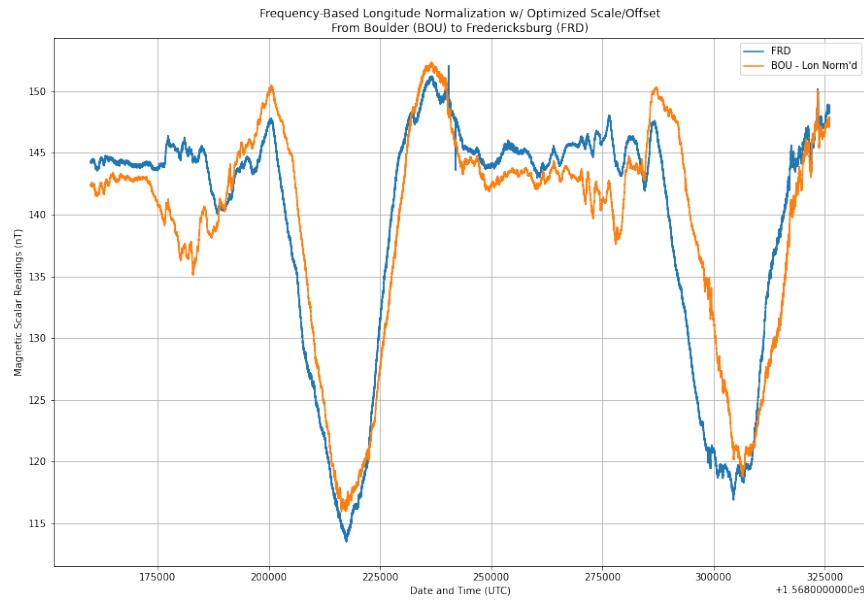


Figure 32: Comparison between the true and estimated temporal variation at the Fredericksburg INTERMAGNET reference station using the ERSM method.

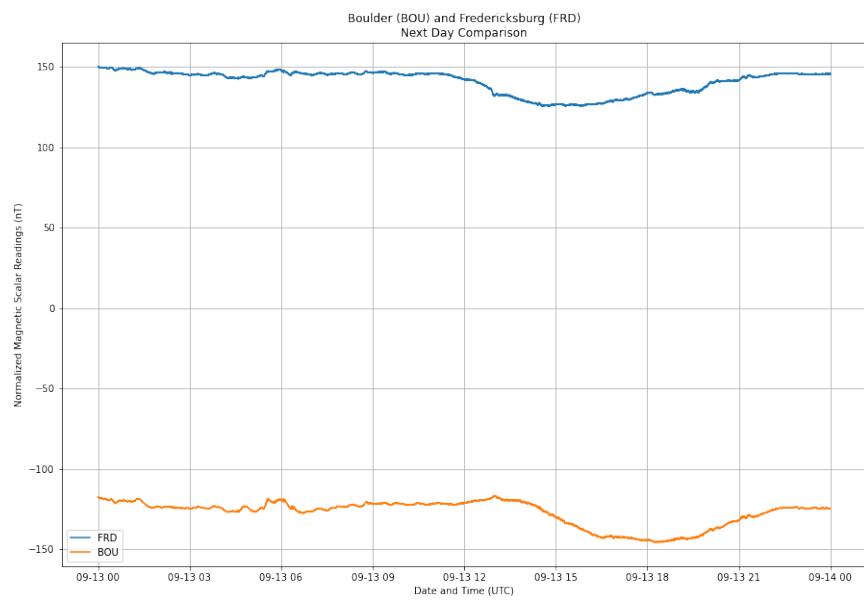


Figure 33: Comparison of raw scalar magnitude values from Boulder and Fredericksburg INTERMAGNET reference stations over the 24 hour period after calibration.

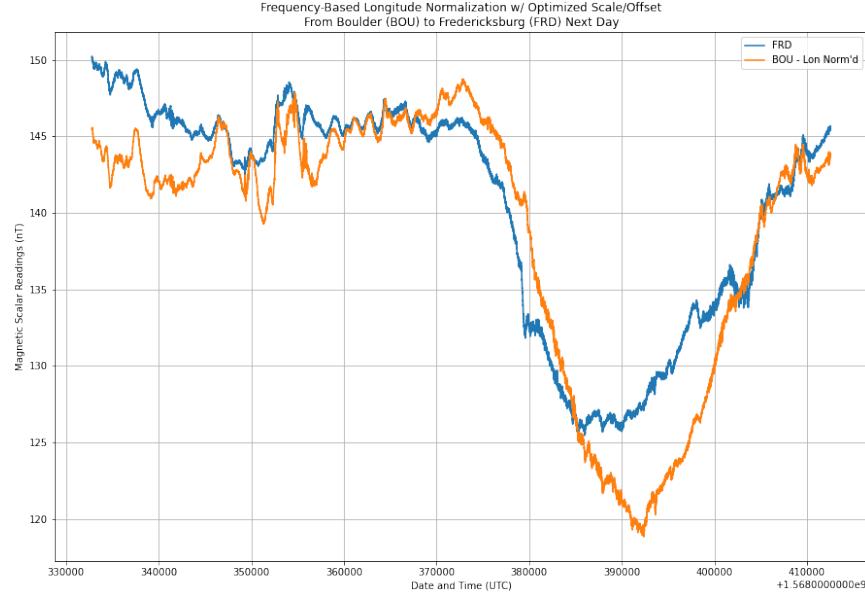


Figure 34: Comparison between the true and estimated temporal variation at the Fredericksburg INTERMAGNET reference station using the ERSM method. The optimal scale and offset parameters that were used for this estimation were calculated using the previous 48 hours worth of data from the same sites.

time of day. During times when the diurnal effect is minimal, the error in estimated temporal variation is fairly small (usually not larger than 2 nT) compared to when the effect is in full swing (up to approximately 20 nT). However, when the diurnal effect is large, the error can usually be estimated as a bias. If conducting a survey with the ERSM method during a time period fully contained in either the falling or rising edge of the day's diurnal swing, the estimated temporal variation will usually add no more than a bias to all estimated magnetic anomaly values. Since we are creating navigation maps, the gradient of the map is much more useful than the absolute anomaly values. This means that a bias in estimated temporal variation will likely have minimal impact on the usefulness of the resulting magnetic navigation (MagNav) map.

### 4.3 Simulated Aeromagnetic Survey Results

A simulated magnetic anomaly, survey data, and set of reconstructed maps were generated to test the functionality and accuracy of the MagNav map Python library developed for this research effort. The true magnetic anomaly map from which the simulated survey data was derived is described in Section 3.3.1 and is shown in Figure 35. Following the data processing steps as laid out in Section 3.3.4, several maps were generated and compared with the truth map.

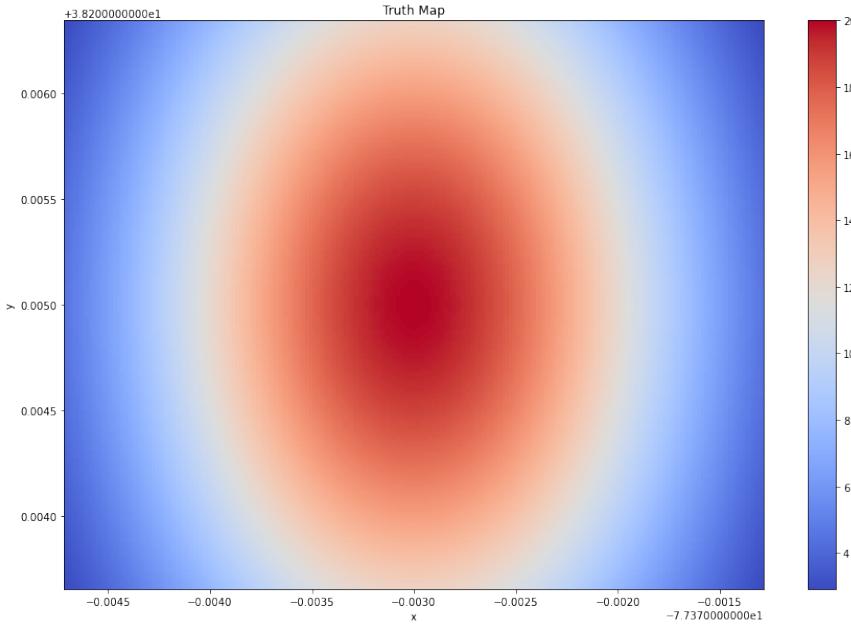


Figure 35: Simulation truth magnetic anomaly map.

The first generated map (Figure 36) used simulated survey data with no added noise or distortion. Performing quite well, the overall RMSE between the generated and truth maps was approximately 0.07 nT. As can be seen in Figure 37, almost all of the error was on the edges of the generated map, especially in the corners. This is almost certainly due to extrapolating pixel values beyond the far edges of the simulated flight lines (edge effects). This source of error is present in all generated maps in this simulation.

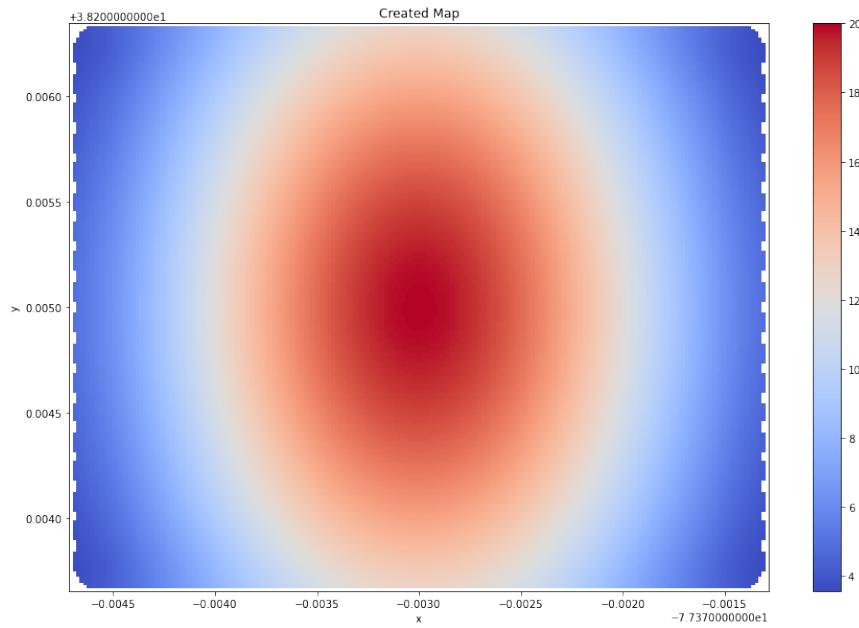


Figure 36: Generated magnetic anomaly map with all survey data and no added noise.

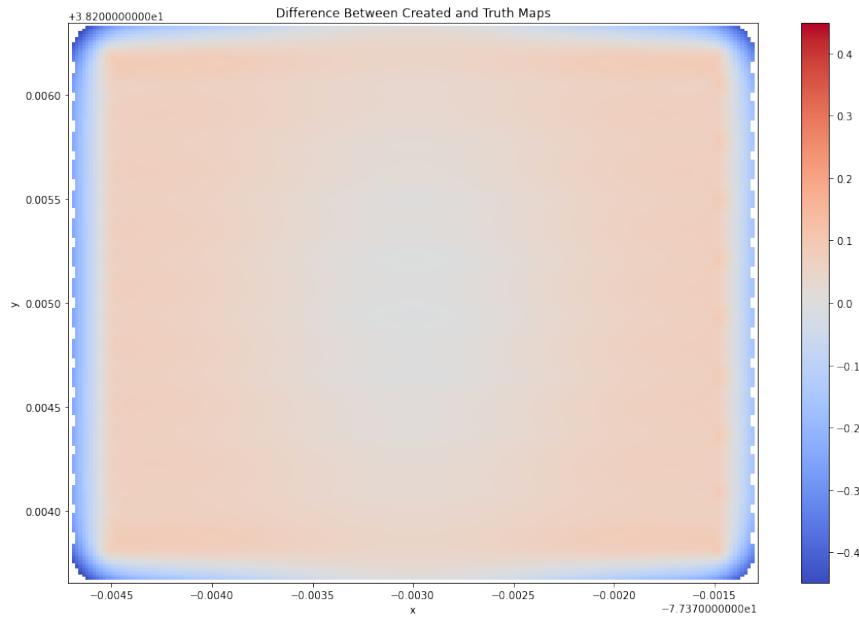


Figure 37: Difference between the truth anomaly map (Figure 35) and the map created with all survey data and no added noise (Figure 36).

Another map was generated using an identical process, except without temporal variation corrections. The resulting map (Figure 38) was found to have an RMSE value of about 145.31 nT. In addition to the corner extrapolation effects, the generated map has an overall bias of approximately -147 nT (as seen in Figure 39). Also, there is a ramp-like error in the North/South (tie line) direction with an amplitude of about 2.5 nT. This bias and ramp error are due to the overall bias and diurnal swing of the temporal variations that were omitted during the generation of this particular map. Lastly, there is a small amount of random noise due to space weather effects that haven't been accounted for.

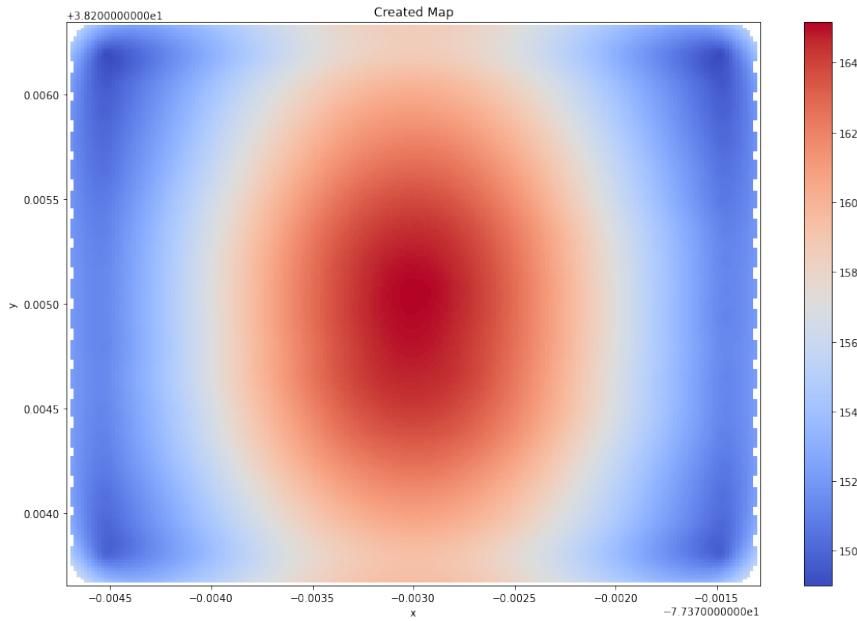


Figure 38: Generated magnetic anomaly map without accounting for temporal variation.

A map was also generated with an estimate of the temporal variation using the ERSM method as described in Section 3.2. The true temporal variation used data from the Fredericksburg INTERMAGNET site and data from the Boulder site was used as the extended reference station data. The ERSM parameters were optimally calculated and applied to the Boulder site data similarly to how described in Sec-

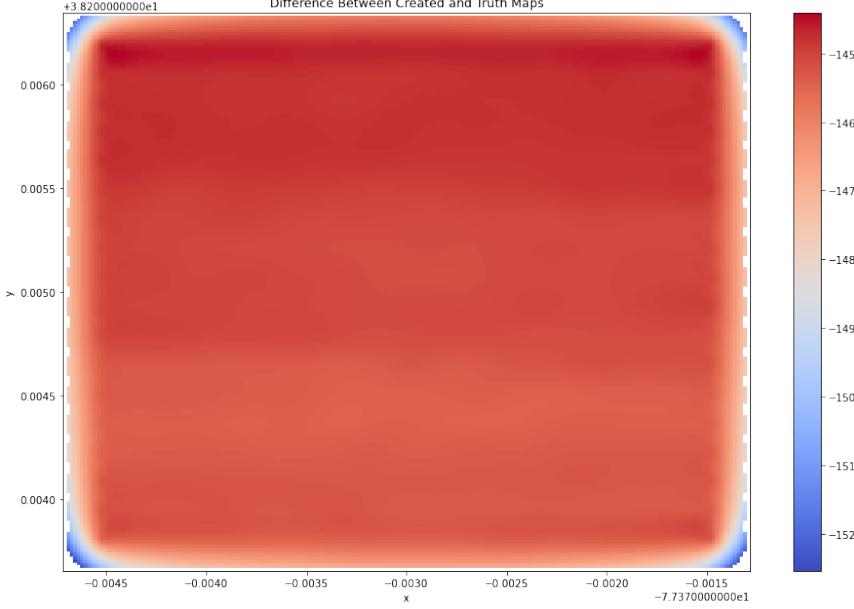


Figure 39: Difference between the truth anomaly map (Figure 35) and the map created without temporal variation corrections (Figure 38).

tion 4.2. However, instead of using all magnetic reference data the day prior to the simulated survey, only data within the same time frame (but previous day) of the simulated survey was used to find the ERSM parameters. The resulting map (Figure 40) was fairly accurate with an RMSE of about 2.29 nT as seen in Figure 41.

#### 4.3.0.1 Constant Bias Analysis

In order to test the effect of bias on map accuracy, a set of 3 maps were created with biases of 0.1 nT, 1 nT, and 10 nT. An example of this can be seen in Figure 42c where a 10 nT bias was added to the simulated survey magnetometer measurements. From Figure 43c it is plain to see that while the random noise is quite low, there is a constant error of  $-10$  nT throughout the map due to the added measurement bias. Similar results were obtained for the 0.1 nT and 1 nT bias maps. The RMSE values for each map can be found in Table 5.

From these results we see that any constant bias to the magnetic measurements

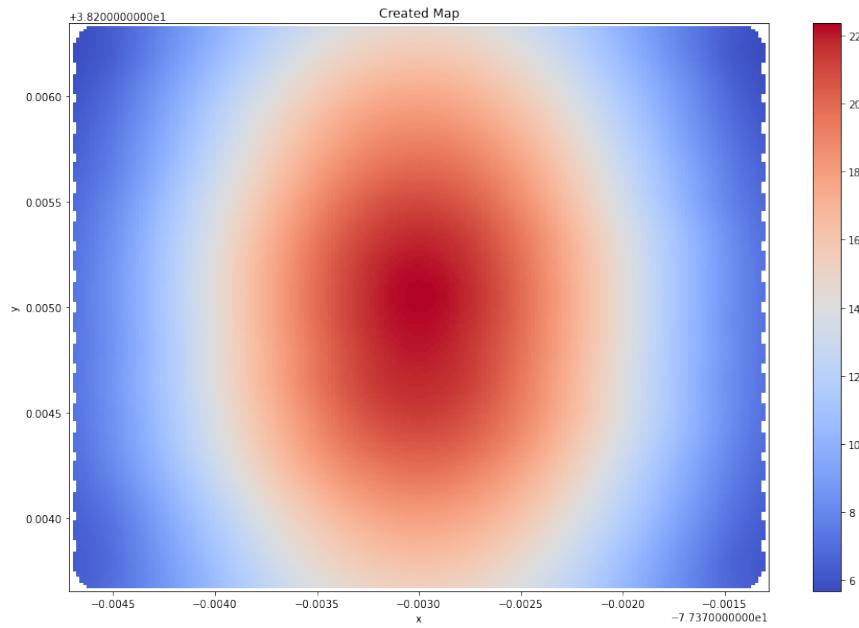


Figure 40: Generated magnetic anomaly map using estimated temporal variation with the ERSM method.

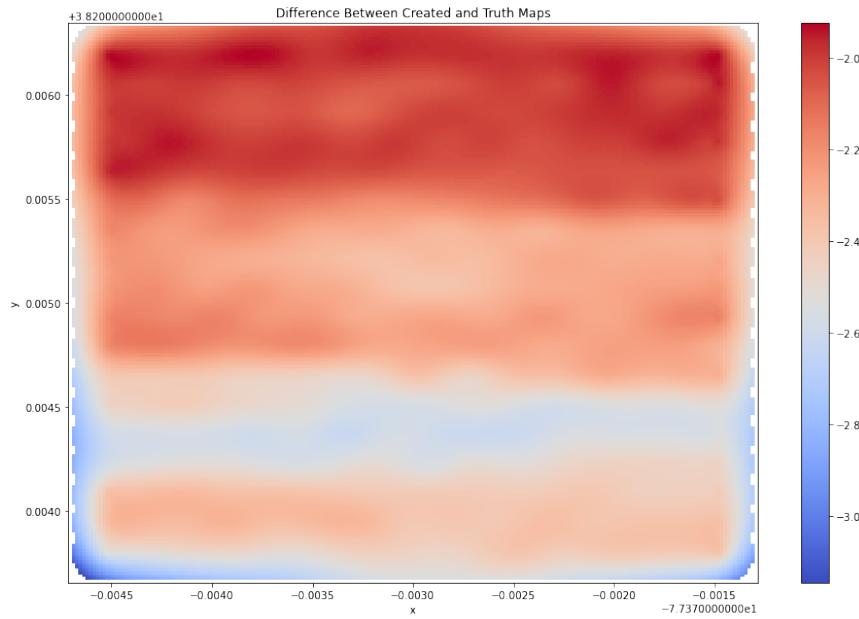


Figure 41: Difference between the truth anomaly map (Figure 35) and the map using estimated temporal variation with the ERSM method (Figure 40).

Table 5: Simulated map errors due to bias.

Measurement Bias (nT)	Generated Map RMSE (nT)
0.1	0.09
1.0	0.97
10.0	9.98

will only add a bias to the map pixels and not significantly alter the map's gradient. This assumes that the bias is present and constant across all magnetic measurements during the given survey.

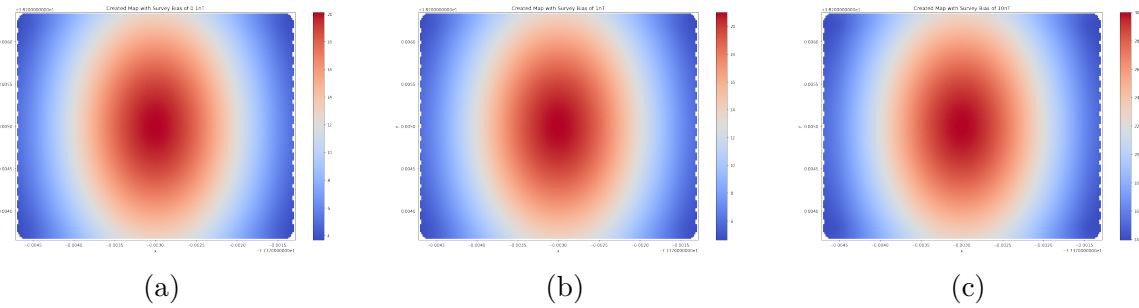


Figure 42: Generated maps where simulated magnetic measurements were biased by 0.1 (a), 1 (b), and 10 nT (c).

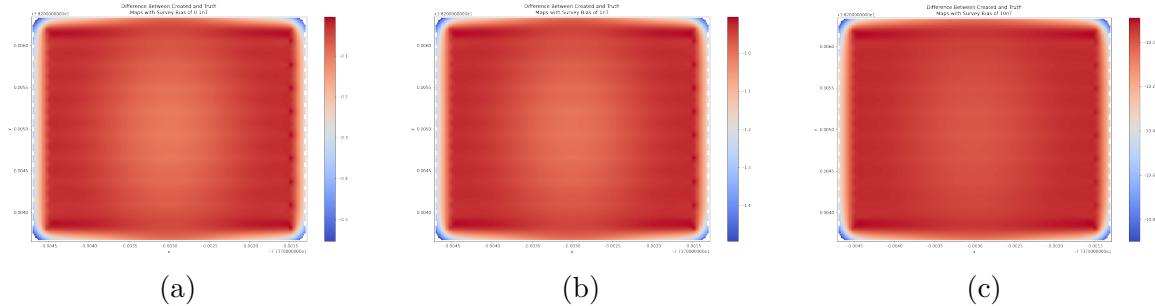


Figure 43: Difference between the truth anomaly map (Figure 35) and the maps where the simulated magnetic measurements were biased by 0.1 (a), 1 (b), and 10 nT (c).

#### 4.3.0.2 Random Noise Analysis

In order to test the effect of Gaussian random noise on map accuracy, a set of 3 maps were created with noise standard deviations of 1 nT, 10 nT, and 100 nT. An example of this can be seen in Figure 44a where random noise with a standard deviation of 1 nT was added to the simulated survey magnetometer measurements. From Figure 45a we see that while the average error is quite low, the added random noise caused significant errors in localized areas. The amount of area that each noisy measurement has an effect on is proportional to both the sample distance along the flight line and the distance between flight lines due to spatial interpolation with a radial basis function. Similar results were obtained for the 1 and 10 nT random noise maps. The RMSE values for each map can be found in Table 6.

Table 6: Simulated map errors due to random noise.

Measurement Noise Standard Deviation (nT)	Generated Map RMSE (nT)
1	0.52
10	4.66
100	48.39

These results show that unless either the magnetometer measurements, generated map, or both are smoothed via filtering, significant distortions in the final map gradient can occur. The data also show that the overall RMSE map error due to random noise is approximately half the noise standard deviation. This means the map error is proportional to the random Gaussian noise level assuming all other sources of error are ignored. From the noised maps shown in Figure 44, it is obvious that even small amounts of random Gaussian noise can severely degrade or even completely mask-out the original source field. Such maps would be useless in a MagNav application.

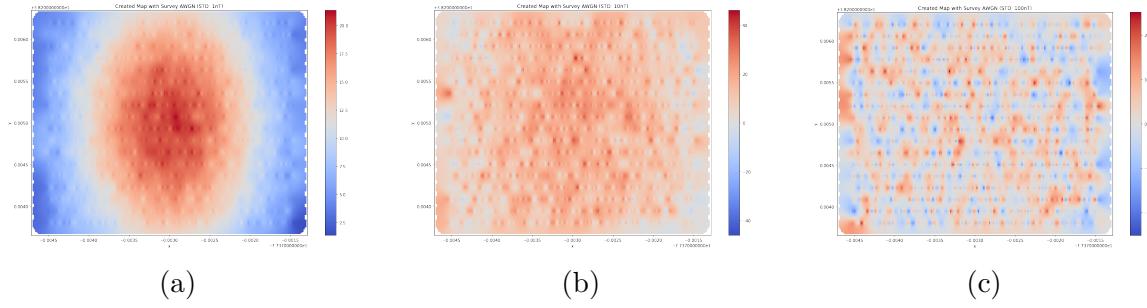


Figure 44: Generated maps where random Gaussian noise with standard deviations of 1 (a), 10 (b), and 100 nT (c) were applied to the simulated scalar magnetometer measurements.

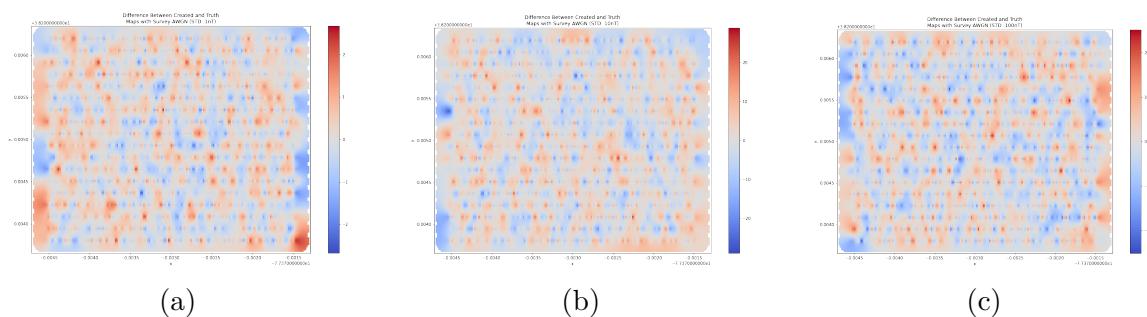


Figure 45: Difference between the truth anomaly map (Figure 35) and maps where random Gaussian noise with standard deviations of 1 (a), 10 (b), and 100 nT (c) were applied to the simulated scalar magnetometer measurements.

## 4.4 Real-World Aeromagnetic Survey Results

### 4.4.1 UAV Survey Results

The survey unmanned aerial vehicle (UAV) was itself surveyed to determine the optimal location for the on-board magnetometers using an extra available MFAM scalar magnetometer. The absolute difference between the measured background magnetic intensity (51 039 nT) and the measured magnetic intensity at various points of the UAV are shown in Table 7. According to the survey results, the tail boom of the UAV had the least amount of magnetic distortion and the on-board magnetometers were then mounted in that location. The low amount of distortion observed in the tail boom is almost certainly due to its maximal distance from the main engine in the UAV nose, electronics/power supplies/servos in the UAV cabin, metal of the main landing gear, and the small amount of metal in the tail landing gear.

Table 7: UAV survey results.

Location On UAV	Magnetic Distortion (nT)
Nose Cone	89
Main Landing Gear	71
Elevator Servo	261
Main Wing Root (Trailing Edge)	59
Tail Boom	1
Tail (Trailing Edge)	68

### 4.4.2 UAV Calibration Results

After mounting the magnetometers in the UAV tail boom, the scalar magnetometers were then calibrated using a static Tolles-Lawson calibration as described in Section 3.4.4. The raw scalar and vector magnetometer plots from both the MFAM

and VMR sensors can be seen in Figure 46. Multiple sets of Tolles-Lawson calibration parameters were generated with various vector magnetometers and combinations of coefficient types.

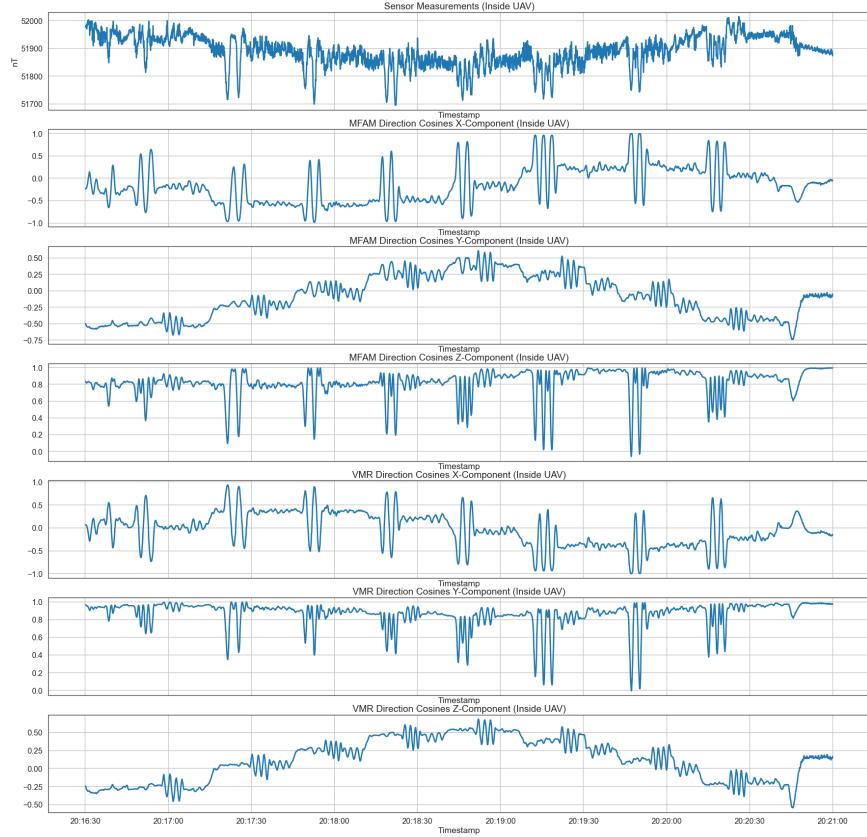


Figure 46: Raw MFAM scalar, MFAM compass (vector), and VMR magnetometer data collected during the static UAV calibration maneuver. The orientation oscillations and heading changes of the aircraft can be seen in the vector magnetometer plots (bottom 6 plots, 3 for the MFAM compass and 3 for the VMR). Noise artifacts that correlate to these movements can be observed in the MFAM scalar magnetometer plot (top plot). These artifacts are due to body effects and heading error encountered during the calibration maneuver.

As can be seen in Table 8, the smallest calibration error (RMSE of 20.42 nT) was achieved when using the VMR vector magnetometer and all Tolles-Lawson coefficient types (permanent, induced, and eddy). However, there was no more than 3 nT difference in RMSE when using one vector sensor versus using the other while using the same Tolles-Lawson coefficient types. There was also no more than 2 nT difference

in RMSE between various subsets of Tolles-Lawson coefficient types when using the same vector sensor except when only using eddy terms. This likely means that there are no eddy components to the body effects of the data. It is also important to note that a large portion of the RMSE values are likely due to the large amount of random noise and persisting MFAM heading errors.

The best performing Tolles-Lawson coefficients are given in Table 9 and were used to calibrate the real-world survey data. The post-calibration scalar and vector plots using these coefficients can be seen in Figure 47.

Table 8: UAV calibration results. Y denotes the given Tolles-Lawson coefficients were calculated and used where N signifies such coefficients were not used.

Vector Sensor	Permanent Coefficients	Induced Coefficients	Eddy Coefficients	RMSE (nT)
MFAM Compass	Y	Y	Y	22.53
MFAM Compass	Y	Y	N	22.54
MFAM Compass	Y	N	N	23.43
MFAM Compass	N	Y	N	24.12
MFAM Compass	N	N	Y	54.84
VMR	Y	Y	Y	20.42
VMR	Y	Y	N	20.51
VMR	Y	N	N	22.77
VMR	N	Y	N	21.82
VMR	N	N	Y	54.81

The Tolles-Lawson coefficients in Table 9 were applied to all four survey datasets. The raw and calibrated scalar measurements for each survey are shown in Figure 48.

Table 9: Tolles-Lawson calibration coefficients used for survey data processing.

<b>Coefficient</b>	<b>Coefficient Value</b>
First Permanent ( $a_1$ )	-1.86687725e+01
Second Permanent ( $a_2$ )	1.33975396e+02
Third Permanent ( $a_3$ )	-1.80762945e+02
First Induced ( $a_4$ )	1.69023832e-01
Second Induced ( $a_5$ )	-3.92262356e-03
Third Induced ( $a_6$ )	-1.84382741e-03
Fourth Induced ( $a_7$ )	1.71830230e-01
Fifth Induced ( $a_8$ )	-1.61173781e-04
Sixth Induced ( $a_9$ )	1.72575427e-01
First Eddy ( $a_{10}$ )	-4.31927864e-04
Second Eddy ( $a_{11}$ )	-8.21512835e-05
Third Eddy ( $a_{12}$ )	-4.37609432e-05
Fourth Eddy ( $a_{13}$ )	-1.06838978e-04
Fifth Eddy ( $a_{14}$ )	-1.22444017e-04
Sixth Eddy ( $a_{15}$ )	-2.76294434e-04
Seventh Eddy ( $a_{16}$ )	-8.51727772e-05
Eighth Eddy ( $a_{17}$ )	3.16374022e-05
Ninth Eddy ( $a_{18}$ )	-2.77441572e-05

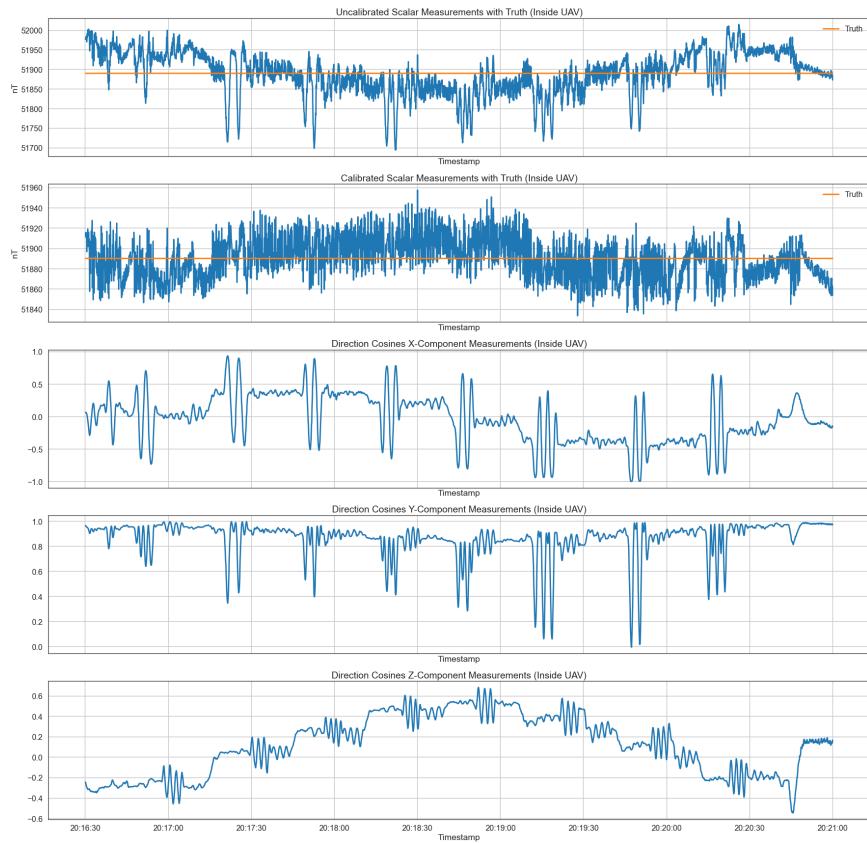


Figure 47: Calibrated MFAM scalar and VMR magnetometer data from the static UAV calibration maneuver. The Tolles-Lawson coefficients used are given in Table 9. The VMR vector magnetometer was used to calculate the direction cosines.

As can be seen in these plots, applying the Tolles-Lawson coefficients greatly reduced heading error and body effects in all datasets except for the 1 km Atterbury Survey (First Attempt). Calibrating this particular dataset caused a noticeable increase of approximately 90 nT in heading error plus ringing of up to 400 nT in certain places. The large ringing effects occur during turns between each flight/tie line and may be caused by higher order terms in the magnetic body effects not being addressed by the Tolles-Lawson calibration.

These findings suggest that the Tolles-Lawson coefficients in Table 9 are valid for all survey datasets except for the 1 km Atterbury Survey (First Attempt) dataset. For this reason, Tolles-Lawson calibration was skipped when processing the dataset of the first 1 km Atterbury survey and simply used the raw scalar measurements.

It is important to note that the first 1 km Atterbury survey was conducted on the week previous to both the calibration maneuver and other Atterbury surveys. During the time between the first 1 km Atterbury survey and the other three surveys and calibration maneuver, there were small inadvertent changes in the mounting locations of both the magnetometers and other avionics and differences in vehicle power supply wiring. It is likely these changes affected the true optimal Tolles-Lawson coefficients of the 1 km Atterbury Survey (First Attempt) dataset.

#### 4.4.3 MFAM Sensor Head Validity Results

Each dataset was analyzed to determine when each MFAM sensor head produced valid and invalid measurements as described in Section 3.4.3. Both MFAM sensor heads were valid for the vast majority of the time as can be seen in Figure 49. However, the MFAM sensor head 2 was occasionally producing invalid measurements likely due to the magnetic field being in the sensor head's dead-zone. These invalid measurements happened most often when flying directly into the West.

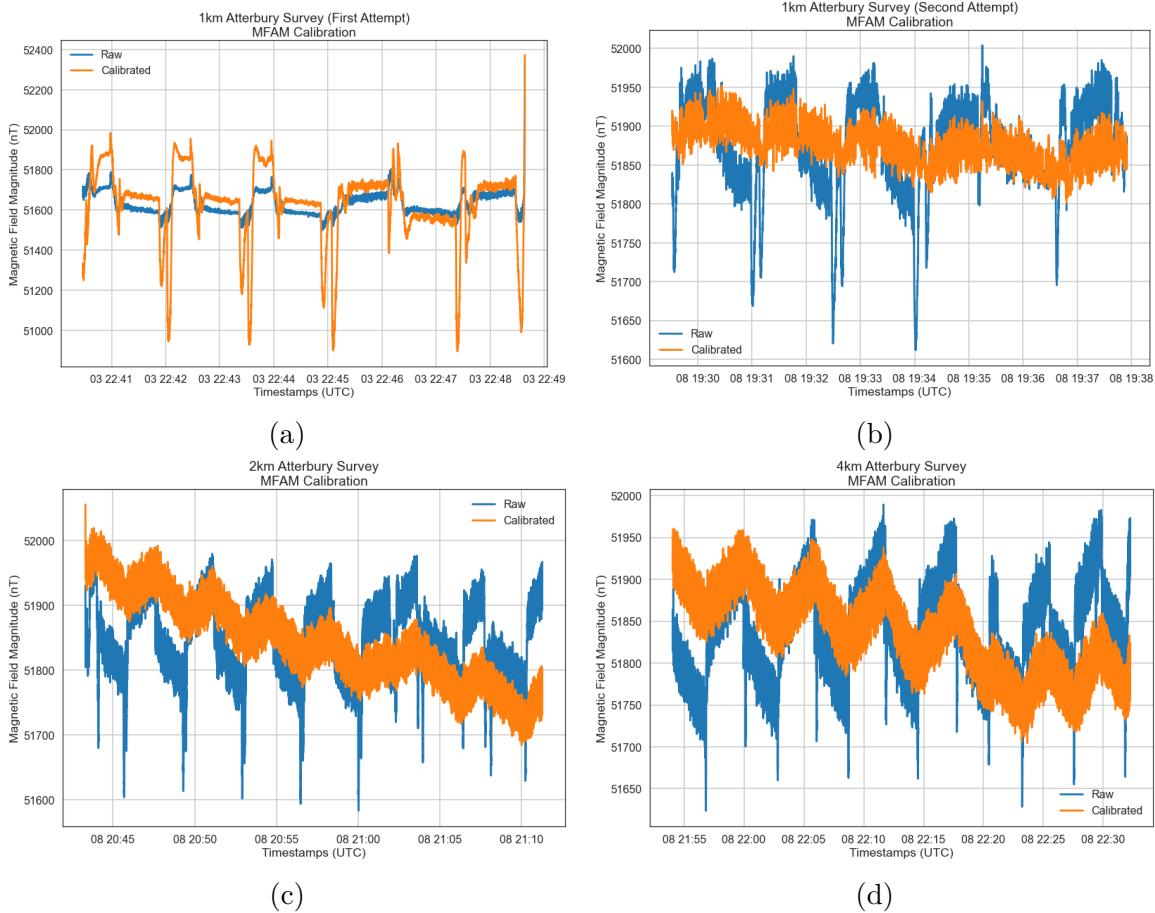


Figure 48: Results of applying the Tolles-Lawson coefficients in Table 9 to the (a) first 1 km Atterbury survey, (b) second 1 km Atterbury survey, (c) 2 km Atterbury survey, and (d) 4 km Atterbury survey. The blue traces represent the raw measurements and the orange traces represent the calibrated measurements. Note that in all cases except for (a), the calibrated measurements had less heading error and ringing compared to the raw measurements.

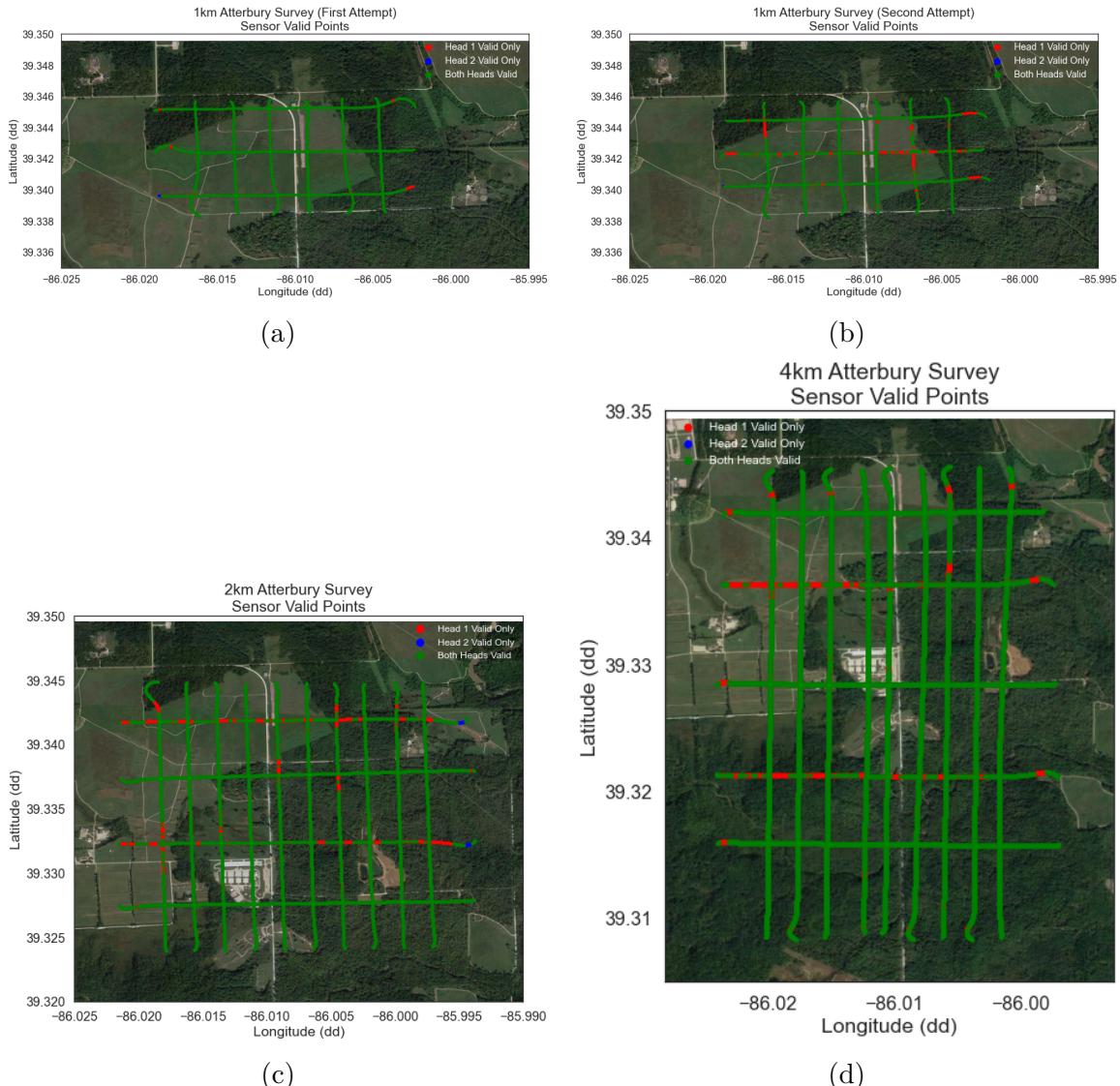


Figure 49: Spatial plots showing MFAM sensor head validity for each sample in the (a) first 1 km Atterbury survey, (b) second 1 km Atterbury survey, (c) 2 km Atterbury survey, and (d) 4 km Atterbury survey. Green dots represent a sample where both MFAM sensor head measurements were valid, red when only MFAM sensor head 1 was valid, and blue when only MFAM sensor head 2 was valid.

Since most samples from both sensor heads were valid, the sensor head meshing technique described in Section 3.4.3 was considered to remain valid for these datasets. It is important to note, however, that the few large and continuous sensor head drop-outs present in the flight lines of the second 1 km and 2 km survey datasets may have introduced noise artifacts to an unknown degree. The large, continuous drop-outs in the tie lines had a much lower impact, because these drop-outs occurred almost exclusively between the flight and tie line intersections.

#### 4.4.4 Map Leveling Results

After calibration and removal of all known noise sources, various map leveling techniques were applied to each survey dataset. This included applying no leveling technique; pseudo tie line leveling using principal component analysis (PCA); leveling each flight line individually against all intersecting tie lines; and tie line leveling all flight lines simultaneously using a plane of best fit. The resulting maps of each survey for each leveling technique can be found in Appendix A.

Inspecting the maps from the various leveling techniques for each survey, it is apparent that maps with pseudo tie line leveling using PCA greatly diverged from those using conventional tie leveling techniques. More specifically, the pseudo tie line leveled maps tended to have around two stripes of alternating high and low estimated magnetic anomaly values in the flight line direction near the “reference” flight line. These stripe artifacts and overall mismatching with the conventional tie line leveled maps suggest that precise tuning of the input parameters is needed for quality output maps. The tunable input parameters include:

- Minimum cumulative contribution rate threshold
- Number of pseudo tie lines

- Placement of each pseudo tie line

After map leveling, a corrugation Figure of Merit (FOM) was generated for each interpolated map. Figure 50 shows a comparison between each map leveling technique for each flown aeromagnetic survey. This figure shows that each of the different leveling techniques performed better at removing corrugation for different surveys. More specifically, the PCA approach was the best at rejecting corrugation for the first 1 km survey, but consistently performed worse than all other leveling techniques (including no leveling) for all other surveys. The plane of best fit leveling approach provided the best map results for the second 1 km survey and the per flight line leveling technique was best for both the 2 and 4 km surveys. The optimal map leveling technique for each survey and corresponding corrugation FOM are given in Table 10.

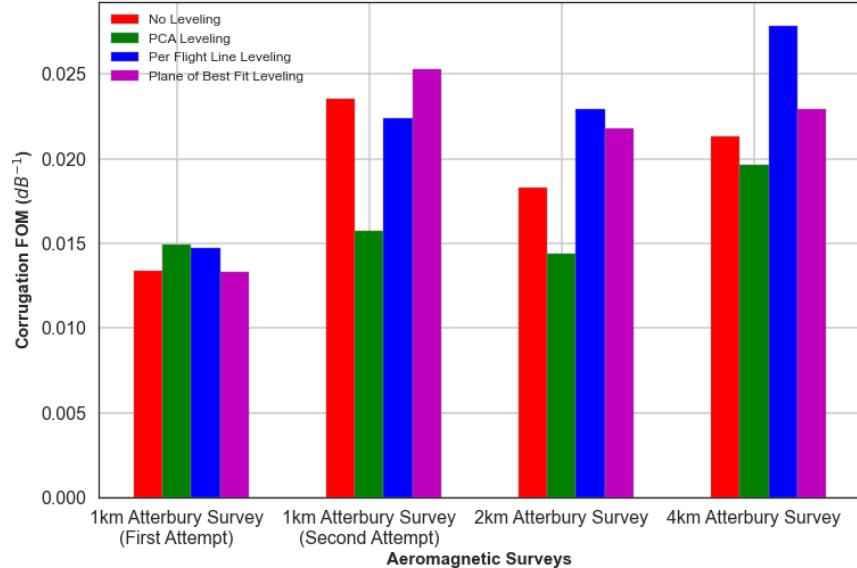


Figure 50: Comparison between different methods of map leveling techniques for all aeromagnetic surveys based on calculated corrugation FOM.

Table 10: Optimal Map Leveling techniques and Corrugation FOM for Each Atterbury Survey.

Aeromagnetic Survey	Optimal Map Leveling	Corrugation FOM ( $dB^{-1}$ )
1 km Atterbury Survey (First Attempt)	PCA Leveling	0.015
1 km Atterbury Survey (Second Attempt)	Plane of Best Fit Leveling	0.025
2 km Atterbury Survey	Per Flight Line Leveling	0.023
4 km Atterbury Survey	Per Flight Line Leveling	0.028

#### 4.4.5 Map Comparisons

The correlation between the best leveled 1 km maps as given in Table 10 were found and the Pearson correlation coefficients were calculated. A resulting coefficient of 1 means the maps are fully correlated and a coefficient of 0 means the maps have no correlation. These maps included the PCA leveled first and plane of best fit second 1 km Atterbury surveys. Similarly, the correlation and Pearson correlation coefficients were found between the per flight line leveled 2 km and 4 km Atterbury surveys. Also, the correlation and Pearson correlation coefficients were calculated between the plane of best fit leveled second 1 km and per flight line leveled 4 km surveys. The results of these correlations are shown in Appendix B, the maximum Pearson correlation coefficients (and their corresponding lags) can be found in Table 11, while the zero-lag Pearson correlation coefficients (autocorrelation coefficients) are given in Table 12.

According to Table 12, the zero-lag correlation between the scalar rasters of the two optimally leveled 1 km maps are highly anticorrelated (-0.88) with almost no correlation in the easterly gradient (-0.05), but high correlation in the northerly gradient (0.73). It is important to note that the overall upwards shift of the eastern half of

Table 11: Maximum Pearson correlation coefficients between the optimally leveled Maps of the Atterbury aeromagnetic surveys. The lags at which the maximum correlation coefficients were found are also given in parenthesis under each correlation coefficient. Plots of the corresponding 2D correlations can be found in Appendix B.

<b>Compared Survey Maps</b>		<b>Maximum Scalar Correlation (Lags at Max)</b>	<b>Maximum Easterly Gradient Correlation (Lags at Max)</b>	<b>Maximum Northerly Gradient Correlation (Lags at Max)</b>
1 km	1 km	0.39 (102, -6)	0.60 (80, 0)	0.76 (0, 0)
1 km (First Attempt) (PCA Leveled)	(Second Attempt) (Plane of Best Fit Leveled)			
2 km (Per Flight Line Leveled)	4 km (Per Flight Line Leveled)	0.64 (0, -1)	0.27 (-85, -142)	0.30 (151, -28)
1 km (Second Attempt) (Plane of Best Fit Leveled)	4 km (Per Flight Line Leveled)	0.83 (0, -1)	0.38 (44, -1)	0.40 (-103, -1)

Table 12: Zero-lag Pearson correlation coefficients between the optimally leveled Maps of the Atterbury aeromagnetic surveys. Plots of the corresponding 2D correlations can be found in Appendix B.

<b>Compared Survey Maps</b>		<b>Zero-Lag Scalar Correlation</b>	<b>Zero-Lag Easterly Gradient Correlation</b>	<b>Zero-Lag Northerly Gradient Correlation</b>
1 km				
1 km	(Second			
(First Attempt)	Attempt)	-0.88	-0.05	0.73
(PCA Leveled)	(Plane of Best			
	Fit Leveled)			
2 km	4 km			
(Per Flight	(Per Flight	0.64	0.04	0.16
Line Leveled)	Line Leveled)			
1 km				
(Second	4 km			
Attempt)	(Per Flight	0.81	-0.46	0.22
(Plane of Best	Line Leveled)			
Fit Leveled)				

the PCA leveled first 1 km Atterbury survey map was due to map leveling. Although the differently leveled maps associated with this survey had higher corrugation, they did not exhibit such a radical shift at the midpoint between the east and west halves of the map. Additionally, the plane of best fit leveled second 1 km Atterbury survey map has the opposite overall trend, where the western half of the map included higher scalar values than the eastern half. This caused the high scalar anticorrelation and northerly gradient correlation.

Conversely, there is a somewhat strong zero-lag correlation (0.64) when comparing the 2 and 4 km per flight line leveled maps. However, both the easterly and northerly gradient rasters have low correlations (0.04 and 0.16, respectively). By inspection, the two maps have a similar overall structure. They both have a maximum in their northwest corner with values gradually decreasing to the southeast corner. However, the 4 km map has more high frequency structure and has a more westerly overall gradient while the 2 km map is smoother with a largely northwesterly gradient.

The highest zero-lag correlation (0.81) was found between the plane of best fit second 1 km and per flight line leveled 4 km Atterbury surveys. However, the northerly gradient correlation was fairly weak (0.22) and the easterly gradient had a somewhat strong anticorrelation (-0.46). Similarly to the 2 and 4 km map comparison, the second 1 km and 4 km maps have similar overall trends with higher values in the west with lower values in the east making the overall correlation high. That being said, the highest values in the second 1 km map is in the southwest corner with values slightly decreasing to the north of that maximum. Meanwhile, the 4 km map has a maximum point at the northwest corner and decreases to the south. This caused the high anticorrelation in the easterly gradients.

The inconsistent correlation coefficients found in Table 11 and Table 12 show that there were issues in the underlying data used to generate at least one of the maps

and may or may not be feasible for navigation. While it is uncertain exactly what source of error/noise caused the underlying data to be compromised, it is likely that persistent heading error, calibration imperfections, and MFAM sensor head drop-outs played a role.

## V. Conclusions

The proposed magnetic navigation (MagNav) survey process and Magnetic Anomaly Map Making for Air and Land (MAMMAL) survey processing Python library successfully created a set of reproducible magnetic anomaly maps showing strong correlation provided that good air frame calibration and other requirements are followed. Additionally, a standardized MagNav map file format was developed and proposed for seamless interoperability between MagNav systems. Extensive contextual metadata was also included as a part of the file standard for easier debugging and analysis of the given map data. The final survey process can be found in Appendix C and a proposed MagNav map file standard can be found in Appendix D.

Analysis of the maps generated using simulated survey data showed that a bias in survey data does not affect the map gradient. Conversely, high-frequency random Gaussian noise with a standard deviations as small as 1 nT greatly affected the map gradient compared to the truth map requiring low-pass filtering to match the spatial frequency extent of the data. A similar low-pass filtering of all magnetic data (both from a ground reference station and the survey vehicle) must be accomplished produce accurate magnetic maps.

In addition to simulated surveys, several overlapping, real-world surveys were conducted over Camp Atterbury (Edinburgh, Indiana) according to the proposed MagNav survey process. After the surveys were completed using an autonomous fixed-wing UAV, the resulting data was compiled and processed using the MAMMAL library. Testing several map leveling techniques, the final map for each survey was chosen based on which map leveling technique produced the least amount of corrugation as indicated by the corrugation FOM. The correlation coefficients of the overlapping portions of these final maps were then calculated. Correlation coefficients of up to 0.81 were obtained, indicating at least two of the generated maps were fairly

accurate and repeatable. However, one survey had a large negative correlation (-0.88) with at least one other overlapping map indicating an underlying data issue. This was almost certainly caused by inaccurate Tolles-Lawson coefficients due to aircraft configuration changes between when that particular survey was flown and when the calibration maneuver was conducted (see Figure 48a).

We introduced a novel method of estimating local temporal magnetic variations using ground magnetic reference stations more than 100 km up to at least 2400 km away from the given survey area. This method, Extended Reference Station Modeling (ERSM), works by first temporarily monitoring the temporal variation in both the local and extended areas. The lower frequency components of the dataset from the extended reference station are then time shifted based on the longitudinal difference between the local and extended areas. Next, optimal scale and offset parameters are found using a least squares approach to minimize the difference between the local and longitude normalized extended datasets. These parameters are then used to process proceeding data from the extended reference station to estimate local temporal variation. It was shown that the local temporal magnetic variations can be estimated using the ERSM method with both an error standard deviation and overall RMSE of less than 5 nT for a period of at least 24 h after the calibration period. These results show the ERSM method may be accurate enough to allow MagNav aeromagnetic surveys to be conducted without the need for a dedicated local ground reference station.

It is important to note that the highest error produced (approximately 10 nT) when using ERSM occurs during daily diurnal swings. Therefore, when using this method, it is best to avoid conducting surveys during these swings. If this cannot be avoided, surveys should be limited to only to one half of the diurnal swing. For instance, if the entire survey is conducted during a single downward or upward shift

in the diurnal swing, the error in temporal variation will likely be a bias of about 10 nT or less. Such a bias is of limited consequence to the overall map gradient (see Section 4.3.0.1). Alternatively, if the survey extends beyond that limited portion of the diurnal swing, the error is no longer a simple bias and can cause error shifts of up to about 15 nT. For example, if one survey is conducted while using the ERSM where the survey starts before the diurnal swing and finishes during the downward diurnal shift, there will likely be very little temporal effect error in the survey points before the swing starts and up to 10 nT error during the downward shift. This would cause there to be two sections in the survey data with a 10 nT offset between them. This in turn would cause large errors in map gradient, especially where the transition between samples taken before and during the diurnal swing occurred.

Finally, the requirements for a MagNav survey and map are given below:

- Small scale ( $<1 \text{ km}^2$  area) surveys should be conducted with rotary wing UAVs to ensure all survey points are properly spaced, especially if the survey aircraft cannot leave the survey boundaries or if a significant obstacle strike risk (*e.g.* treetops, buildings, etc.) exists
- Large scale ( $>1 \text{ km}^2$  area) surveys should be conducted with either fixed-wing UAVs or crewed aircraft due to the longer average endurance of such platforms compared to rotary wing UAVs [6]
- Survey unmanned aerial vehicle (UAV) should be appropriately sanitized/de-gauussed
- Survey magnetometer(s) should be mounted in the least magnetically noisy location on the UAV
- Survey UAV must have at least a vector magnetometer for Tolles-Lawson calibration purposes

- Survey UAV must be calibrated for each day a survey is collected
- Surveys should be conducted at or below the lowest altitude required for navigation
- Surveys must have flight line distances no greater than one half the survey minimum altitude AGL
- Surveys should include at least 1 tie line for every 4-8 flight lines, but no less than 3 tie lines overall [20]
- Tie lines should be evenly spaced and cover as much of the survey area as possible
- Maps must be exported and saved with the proper formatting and metadata according the MagNav map standard (see Appendix D)

## 5.1 Future Work

Further flight testing should be conducted to improve the accuracy and reliability of the proposed aeromagnetic anomaly survey process. The Camp Atterbury UAV flight test area should be re-surveyed a number of times with various UAV platform and magnetometer types. More specifically, at least one rotary wing UAV with a suspended magnetometer payload should be developed and used for aeromagnetic surveys. Both fixed and rotary wing platforms should be flown with optically pumped and high quality fluxgate magnetometers on separate survey flights. Analysis of these additional flight tests will provide insight on which UAV platform and magnetometer pair provides the most reliable and repeatable data for generating magnetic anomaly maps.

Additionally, future work should address the current inability to validate generated magnetic anomaly maps without spending resources re-surveying the same areas by geological prospecting companies. Inspired by personal discussions with faculty at the Autonomy and Navigation Technology (ANT) Center at Air Force Institute of Technology (AFIT), this could be rectified by placing a powerful, oscillating electromagnetic solenoid at a known location within the survey area during collection. The driving frequency of the solenoid should be well above the expected magnetic anomaly frequency as seen by the survey aircraft's magnetometer. After applying a band-pass filter to the estimated magnetic anomaly data, the average strength of the magnetic field produced by the solenoid can be estimated at each point along the survey flight path. Comparison between the measured and theoretical average magnetic field strength from the solenoid for each survey sample may provide insight on the accuracy of the underlying estimated magnetic anomaly data. Interestingly, the underlying magnetic anomaly data should not be affected by the solenoid assuming the driving signal of the solenoid is high enough. Applying a low-pass filter should remove the high frequency solenoid content while outputting unperturbed estimated anomaly samples that can be used to generate a map as if the solenoid was absent. Additionally, the difference between the expected and measured solenoid magnetic field strength values might be useful for generating map leveling corrections. Further surveys should be conducted to test the proposed usefulness of such a solenoid during collection.

Lastly, the navigation worthiness of all generated anomaly maps should be further tested. According to personal discussions with faculty at the ANT Center at AFIT, this can be done in simulation by having a MagNav filter process inertial measurement unit (IMU) and magnetometer data from a random flight path through the survey area along with the generated anomaly map. The MagNav filter output can then

be compared to the true location of the UAV throughout the given flight. This comparison will provide useful insights on how accurate the generated anomaly map (and the underlying survey process as a whole) is for navigation purposes.

## Appendix A. Final Maps

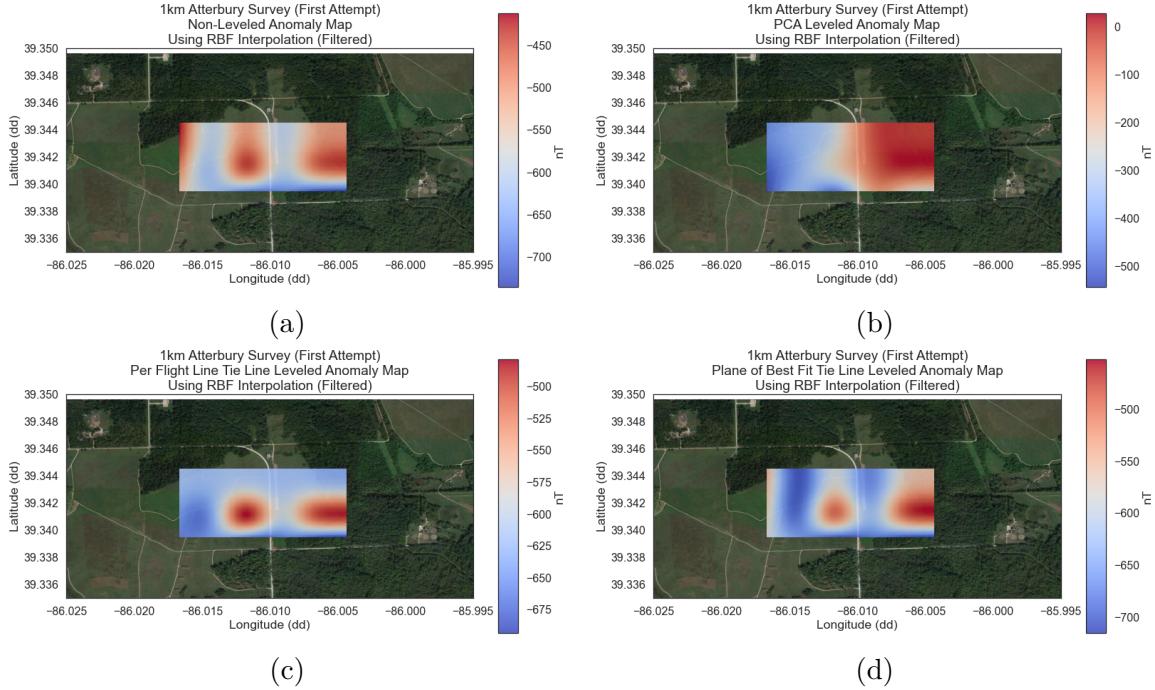


Figure 51: Maps of the first 1 km Atterbury survey after leveling with (a) no technique, (b) pseudo tie line leveling using principal component analysis (PCA), (c) leveling each flight line individually against all intersecting tie lines, and (d) tie line leveling all flight lines simultaneously using a plane of best fit.

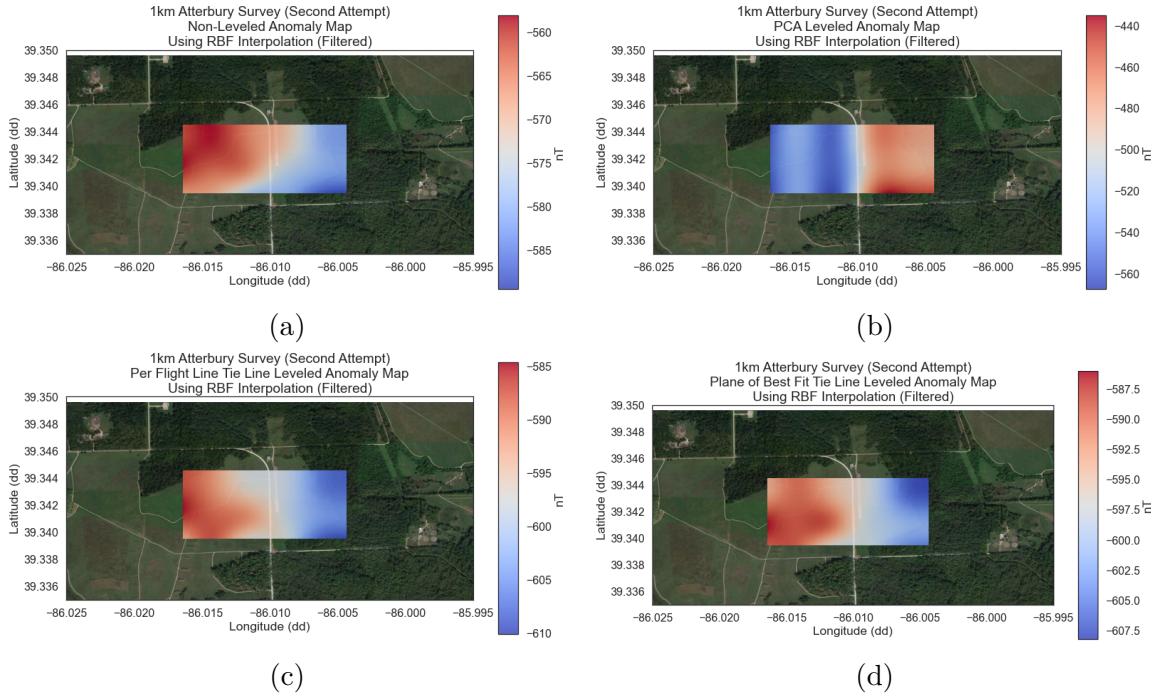


Figure 52: Maps of the second 1 km Atterbury survey after leveling with (a) no technique, (b) pseudo tie line leveling using PCA, (c) leveling each flight line individually against all intersecting tie lines, and (d) tie line leveling all flight lines simultaneously using a plane of best fit.

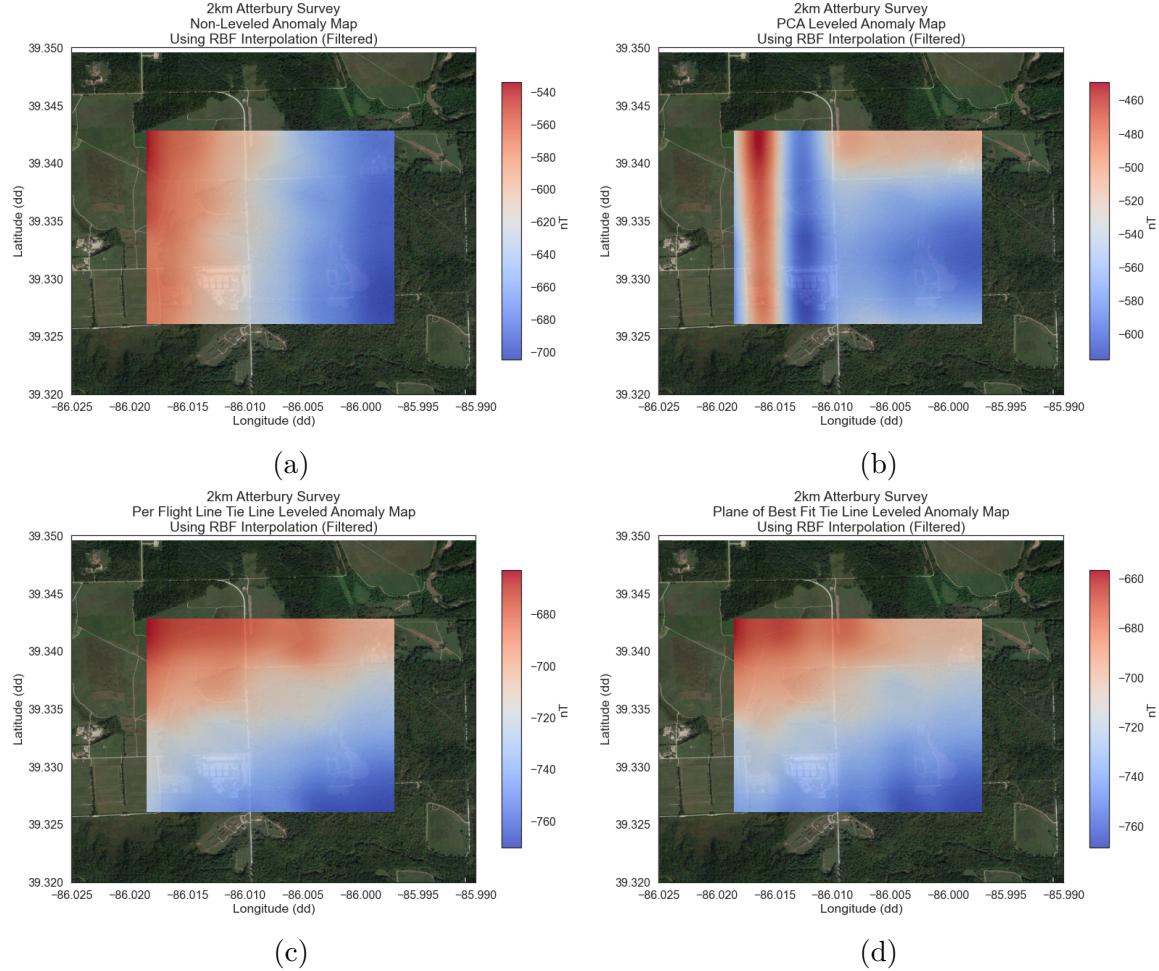


Figure 53: Maps of the 2 km Atterbury survey after leveling with (a) no technique, (b) pseudo tie line leveling using PCA, (c) leveling each flight line individually against all intersecting tie lines, and (d) tie line leveling all flight lines simultaneously using a plane of best fit.

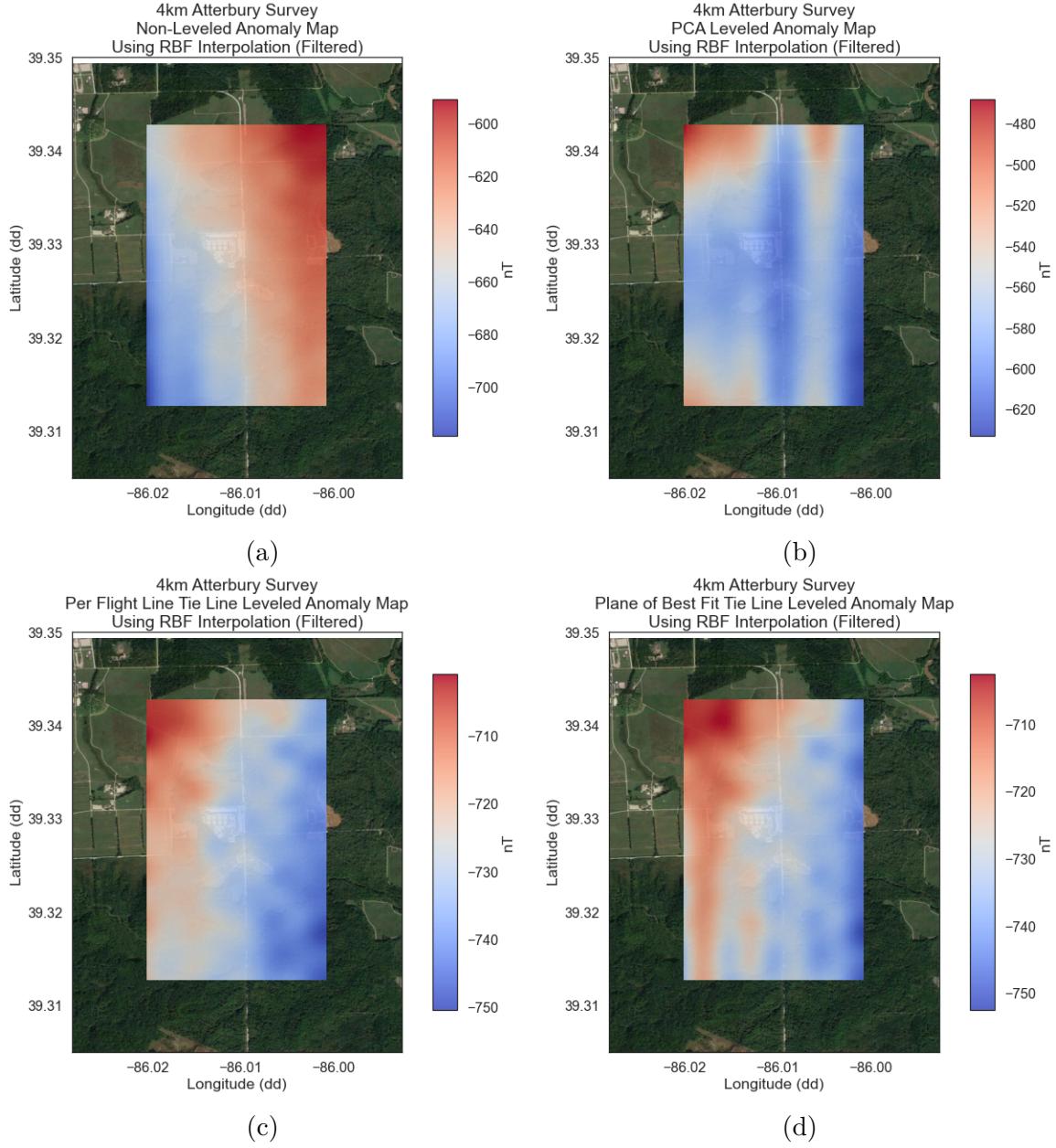


Figure 54: Maps of the 4 km Atterbury survey after leveling with (a) no technique, (b) pseudo tie line leveling using PCA, (c) leveling each flight line individually against all intersecting tie lines, and (d) tie line leveling all flight lines simultaneously using a plane of best fit.

## Appendix B. Map Correlations

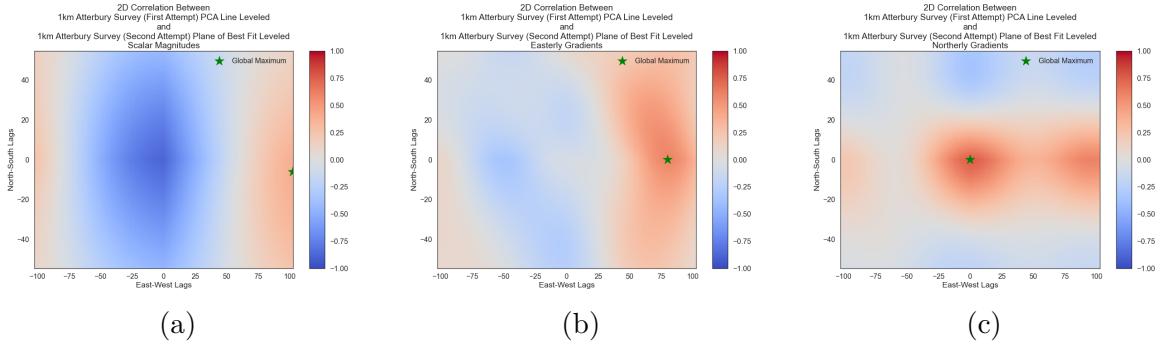


Figure 55: 2D correlation plots between the first 1 km PCA and second 1 km plane of best fit leveled Atterbury survey maps for the (a) scalar, (b) easterly gradient, and (c) northerly gradient rasters.

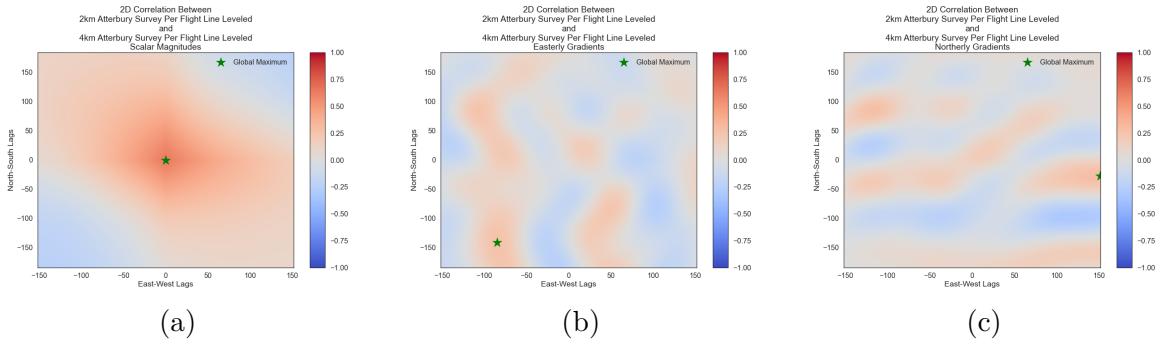


Figure 56: 2D correlation plots between the 2 km and 4 km per flight line leveled Atterbury survey maps for the (a) scalar, (b) easterly gradient, and (c) northerly gradient rasters.

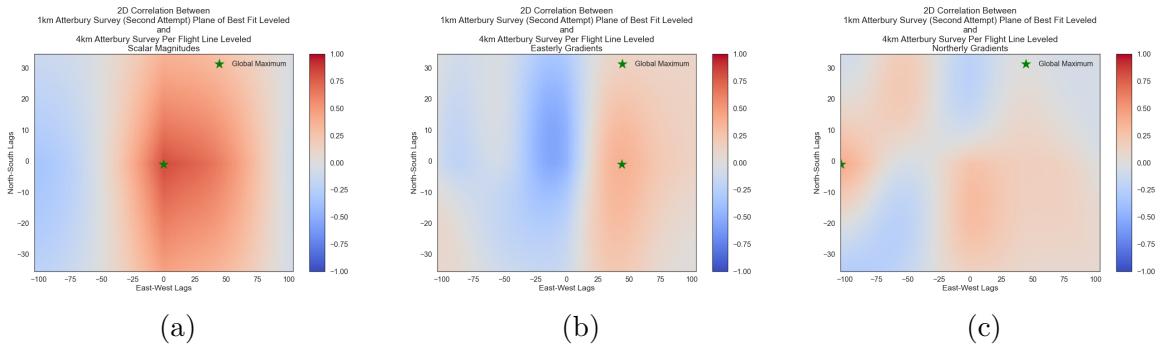
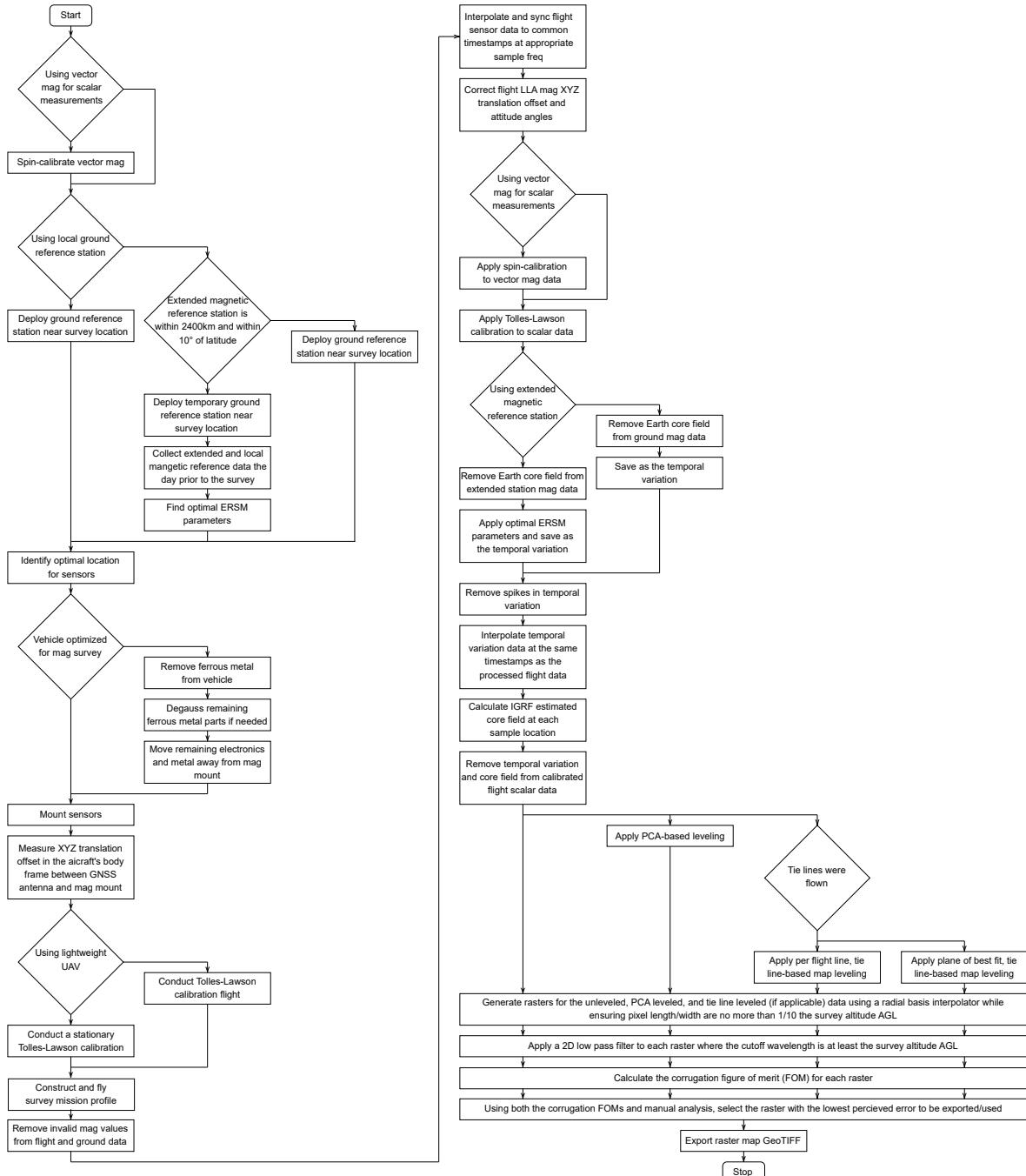


Figure 57: 2D correlation plots between the second 1 km plane of best fit and 4 km per flight line leveled Atterbury survey maps for the (a) scalar, (b) easterly gradient, and (c) northerly gradient rasters.

## Appendix C. Proposed MagNav Survey Process



## Appendix D. Proposed MagNav GeoTIFF File Standard

Magnetic anomaly maps for MagNav must be standardized in a easy to use, common file format with consistent use of units. This will ensure plug-and-play interoperability between all future MagNav filters and maps generated by various sources.

The GeoTIFF format is a highly versatile extension designed to represent various geospatial data and is ubiquitous in the geospatial data processing discipline with many mapping tools and software already supporting the file format. For this reason, all MagNav survey maps should be published as GeoTIFF files with the following metadata and fields:

- Coordinate reference system:
  - WGS84
- Orientation of raster bands:
  - North up
- Invalid pixel value:
  - NaN
- Top level metadata:
  - Metadata field name: “Description”
    - \* Standardized value: “MagNav Aeromagnetic Anomaly Map”
  - Metadata field name: “ProcessingApp”
    - \* Description of the application name and version used to generate the map file
  - Metadata field name: “SurveyDateUTC”

- \* Approximate UTC data of the survey in an ISO 8601 formatted string
- Metadata field name: “SampleDistM”
  - \* Approximate distance between each magnetic reading along a given flight line in meters
- Metadata field name: “xResolutionM”
  - \* Pixel width in meters
- Metadata field name: “yResolutionM”
  - \* Pixel height in meters
- Metadata field name: “ExtentDD”
  - \* Extent of map in degrees decimal
  - \* Example: “[−84.0958, 39.7617, −84.0484, 39.7823]”
- Metadata field name: “ScalarType”
  - \* Description of the make/model/type of scalar magnetometer used
- Metadata field name: “VectorType”
  - \* Description of the make/model/type of vector magnetometer used
- Metadata field name: “ScalarSensorVar”
  - \* Survey scalar magnetometer variance in nT
- Metadata field name: “VectorSensorVar”
  - \* Survey vector magnetometer variance in nT
- Metadata field name: “POC”
  - \* Point of contact information about the organization who conducted the survey and produced the map (no standard format for the information in this metadata field)

- Metadata field name: “KML”
  - \* Keyhole Markup Language (KML) document text that specifies the timestamped survey sample locations; flight/tie line average directions, distances, and altitudes for each sub-survey area; and location of roads, power lines, and substations
  - \* The timestamped survey sample locations must be represented by a top-level GxTimeSpan named “FlightPath” with UTC timestamps; WGS84 coordinates; and the altitude mode set to “absolute”.
  - \* The sub-survey areas must be represented by a top-level folder of polygons named “SubSurveyAreas”. Each sub-survey area polygon must have the following description: “FL Dir: (fldir) $^{\circ}$ , FL Dist: (fldist)m, TL Dir: (tldir) $^{\circ}$ , TL Dist: (tldist)m, Alt: (alt)m above MSL” where:
    - “(fldir)” is replaced with the average flight line direction in degrees off North
    - “(fldist)” is replaced with the average flight line distance in meters
    - “(tldir)” is replaced with the average tie line direction in degrees off North (if tie lines not present, set to -90)
    - “(tldist)” is replaced with the average tie line distance in meters (if tie lines not present, set to 0)
    - “(alt)” is replaced with the average altitude in meters above mean sea level (MSL)
    - Directions must within the range [0 $^{\circ}$ , 180 $^{\circ}$ ] except for the tie line direction if tie lines are not present (set value to -90)
  - \* The road locations must be represented by a top-level multigeometry of line strings named “Roads” with WGS84 coordinates and the altitude mode to “clampToGround”

- \* The power line locations must be represented by a top-level multi-geometry of line strings named “PowerLines” with WGS84 coordinates and the altitude mode set to “clampToGround”
- \* The substation locations must be represented by a top-level multi-geometry of polygons named “Substations” with WGS84 coordinates and the altitude mode set to “clampToGround”
- Metadata field name: “LevelType”
  - \* Description of the algorithm used for map leveling
- Metadata field name: “TLCoeffTypes”
  - \* Ordered list of Tolles-Lawson coefficient types used
  - \* Example: “[Permanent, Induced, Eddy]”
- Metadata field name: “TLCoeffs”
  - \* Ordered list of Tolles-Lawson coefficients used
  - \* Example: “[0.62, 0.70, 0.55, 0.24, 0.49, 0.28, 0.43, 0.57, 0.90, 0.80, 0.84, 0.14, 0.42, 0.58, 0.85, 0.86, 0.80, 0.73]”
- Metadata field name: “CSV”
  - \* Comma-separated values (CSV) document text that includes all pertinent survey data points and data processing steps.
  - \* Minimum required columns include:
    - TIMESTAMP: Coordinated Universal Time (UTC) timestamps (s)
    - LAT: Latitudes (dd)
    - LONG: Longitudes (dd)
    - ALT: Altitudes above MSL (m)

- DC\_X: Direction cosine X-Components (dimensionless)
- DC\_Y: Direction cosine Y-Components (dimensionless)
- DC\_Z: Direction cosine Z-Components (dimensionless)
- F: Raw scalar measurements (nT)

\* Suggested columns include (may vary depending on exact steps used to produce the original map values):

- F\_CAL: Calibrated scalar measurements (nT)
- F\_CAL\_IGRF: Calibrated scalar measurements without core field (nT)
- F\_CAL\_IGRF\_TEMPORAL: Calibrated scalar measurements without core field or temporal corrections (nT)
- F\_CAL\_IGRF\_TEMPORAL\_FILT: Calibrated scalar measurements without core field or temporal corrections after low pass filtering (nT)
- F\_CAL\_IGRF\_TEMPORAL\_FILT\_LEVEL: Calibrated scalar measurements without core field or temporal corrections after low pass filtering and map leveling (nT)

– Metadata field name: “InterpType”

\* Description of the algorithm used for map pixel interpolation

– Metadata field name: “FinalFiltCut”

\* Cutoff wavelength of the 2D low pass filter applied to the interpolated scalar pixel values

– Metadata field name: “FinalFiltOrder”

\* Order number of the 2D low pass filter applied to the interpolated scalar pixel values

- Band 1:
  - NxM raster array of scalar magnetic anomaly values in nT
  - Band metadata:
    - \* Metadata field name: “Type”
      - Standardized value: “F”
    - \* Metadata field name: “Units”
      - Standardized value: “nT”
    - \* Metadata field name: “Direction”
      - Standardized value: “n/a”
- Band 2:
  - NxM raster array of North magnetic anomaly component values in nT  
(optional - if no data provided, fill band with NaNs)
  - Band metadata:
    - \* Metadata field name: “Type”
      - Standardized value: “X”
    - \* Metadata field name: “Units”
      - Standardized value: “nT”
    - \* Metadata field name: “Direction”
      - Standardized value: “North”
- Band 3:
  - NxM raster array of East magnetic anomaly component values in nT (optional - if no data provided, fill band with NaNs)
  - Band metadata:

- \* Metadata field name: “Type”
  - Standardized value: “Y”
- \* Metadata field name: “Units”
  - Standardized value: “nT”
- \* Metadata field name: “Direction”
  - Standardized value: “East”
- Band 4
  - NxM raster array of downward magnetic anomaly component values in nT  
(optional - if no data provided, fill band with NaNs)
  - Band metadata:
    - \* Metadata field name: “Type”
    - Standardized value: “Z”
    - \* Metadata field name: “Units”
    - Standardized value: “nT”
    - \* Metadata field name: “Direction”
    - Standardized value: “Down”
- Band 5
  - NxM raster array of pixel altitudes in meters above MSL
  - Band metadata:
    - \* Metadata field name: “Type”
    - Standardized value: “ALT”
    - \* Metadata field name: “Units”
    - Standardized value: “m MSL”

- \* Metadata field name: “Direction”
  - Standardized value: “n/a”
- Band 6
  - NxM raster array of pixel interpolation standard deviation values in nT  
(optional - if no data provided, fill band with NaNs)
  - Band metadata:
    - \* Metadata field name: “Type”
      - Standardized value: “STD”
    - \* Metadata field name: “Units”
      - Standardized value: “nT”
    - \* Metadata field name: “Direction”
      - Standardized value: “n/a”
- Band 7
  - NxM raster array of pixel easterly gradients
  - Band metadata:
    - \* Metadata field name: “Type”
      - Standardized value: “dX”
    - \* Metadata field name: “Units”
      - Standardized value: “nT”
    - \* Metadata field name: “Direction”
      - Standardized value: “East”
- Band 8
  - NxM raster array of pixel northerly gradients

- Band metadata:
  - \* Metadata field name: “Type”
    - Standardized value: “dY”
  - \* Metadata field name: “Units”
    - Standardized value: “nT”
  - \* Metadata field name: “Direction”
    - Standardized value: “North”

## **Appendix E. Python Code**

# intermagnet\_temporal\_analysis

January 19, 2023

```
[ ]: from os import getcwd
      from os.path import dirname, join

      import matplotlib.pyplot as plt
      import numpy as np
      import pandas as pd
      from scipy import interpolate

      import MAMMAL.Diurnal as Diurnal
      from MAMMAL.Parse import parseIM as pi
      from MAMMAL.Utils import ProcessingUtils as pu

      %matplotlib inline
      plt.rcParams["figure.figsize"] = (15, 10) # (w, h)

      SRC_DIR      = getcwd()
      ANALYSIS_DIR = join(SRC_DIR, 'analysis_data')
      TEST_DIR     = join(SRC_DIR, 'test_data')
```

## 1 Load INTERMAGNET Data

```
[ ]: df_dict = pi.loadInterMagData(ANALYSIS_DIR)

      from_name = 'BOU'
      to_name   = 'FRD'

      from_df = df_dict[from_name]
      to_df   = df_dict[to_name]

      to_df = pu.reject_outliers(to_df,
                                 window_size=250,
                                 std_lim=3)

      to_IGRF_f   = np.array(to_df.IGRF_F)[0]
      from_IGRF_f = np.array(from_df.IGRF_F)[0]
```

```

to_f    = np.array(to_df.F)
from_f  = np.array(from_df.F)

to_f_no_core    = np.array(to_df.F) - to_IGRF_f
from_f_no_core = np.array(from_df.F) - from_IGRF_f

to_t    = np.array(to_df.epoch_sec)
from_t = np.array(from_df.epoch_sec)

plt.figure()
plt.title('Boulder (BOU) and Fredericksburg (FRD)\n2 Day Comparison')
plt.xlabel('Date and Time (UTC)')
plt.ylabel('Normalized Magnetic Scalar Readings (nT)')
plt.plot(to_df.datetime, to_f_no_core, label=to_name)
plt.plot(from_df.datetime, from_f_no_core, label=from_name)
plt.legend()
plt.grid()

plt.figure()
plt.title('Boulder (BOU) and Fredericksburg (FRD)\n2 Day Comparison')
plt.xlabel('Date and Time (UTC)')
plt.ylabel('Normalized Magnetic Scalar Readings (nT)')
plt.plot(to_df.datetime, to_f_no_core - to_f_no_core[0], label=to_name)
plt.plot(from_df.datetime, from_f_no_core - from_f_no_core[0], label=from_name)
plt.legend()
plt.grid()

print('RMSE:', pu.rmse(to_f_no_core, from_f_no_core[:len(to_f)])) # Extra slicing needed because dimensions don't match as a result of rejecting outliers
print('RMSE:', pu.rmse(to_f_no_core - to_f_no_core[0], from_f_no_core[:len(to_f)] - from_f_no_core[0])) # Extra slicing needed because dimensions don't match as a result of rejecting outliers

```

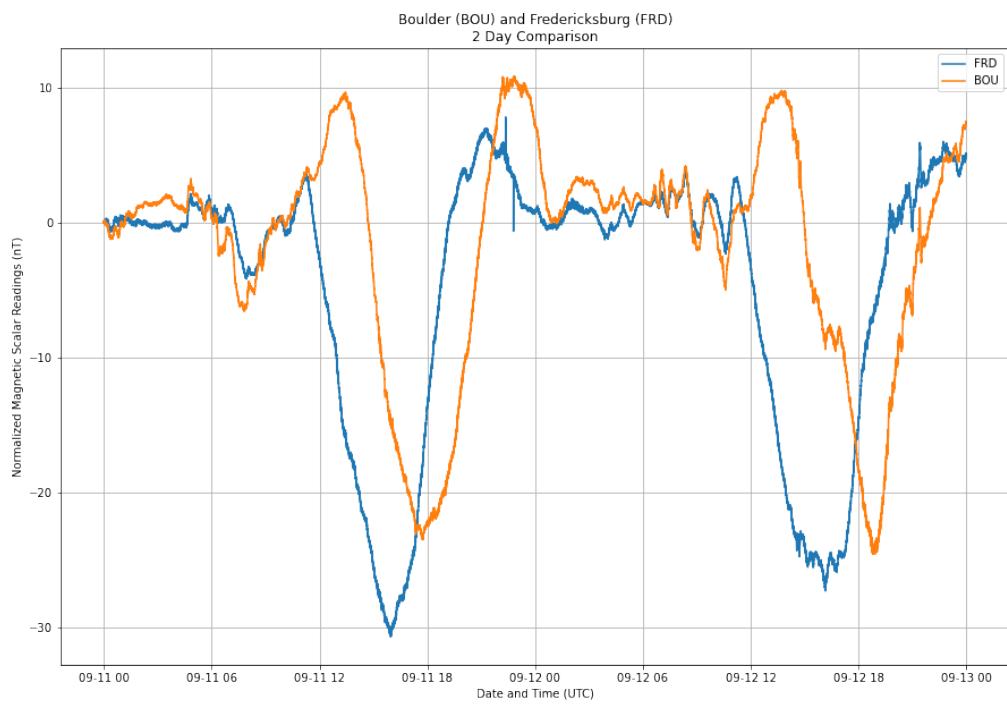
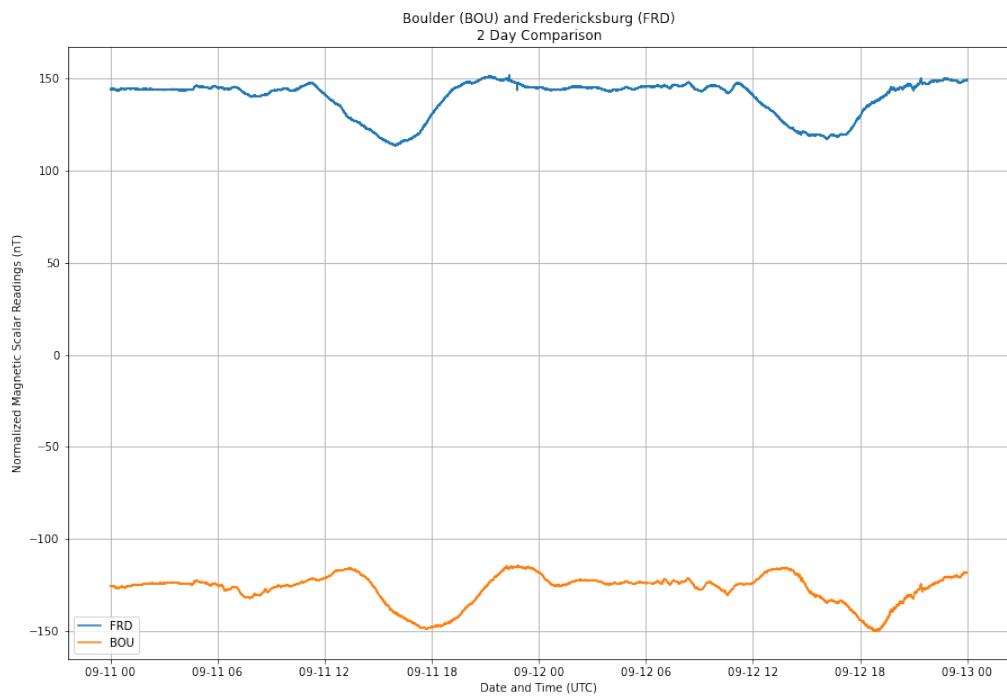
```

Loaded bou20190911psec.sec
Loaded bou20190912psec.sec
Loaded frd20190911psec.sec
Loaded frd20190912psec.sec

0%|          | 0/692 [00:00<?, ?it/s]C:\Users\ltber\Anaconda3\Lib\site-packages\pandas\core\dtypes\cast.py:2221: RuntimeWarning: invalid value encountered in cast
      casted = element.astype(dtype)
100%|       | 692/692 [00:01<00:00, 584.82it/s]

RMSE: 267.3244711396219
RMSE: 10.869259973212195

```



## 2 Simple Longitude Normalization

```
[ ]: lon_diff      = from_df.LONG.mean() - to_df.LONG.mean()
lon_t_offset = pd.Timedelta(seconds=lon_diff / Diurnal.E_ROT_DEG_S)
bou_shift_t  = from_df.epoch_sec + lon_t_offset.total_seconds()

plt.figure()
plt.title('Simplified Longitude Normalization\nFrom Boulder (BOU) to
    ↪Fredericksburg (FRD)')
plt.xlabel('Date and Time (UTC)')
plt.ylabel('Normalized Magnetic Scalar Readings (nT)')
plt.plot(to_df.datetime, to_f_no_core, label=to_name)
plt.plot(from_df.datetime + lon_t_offset, from_f_no_core, label='{} - Lon
    ↪Norm\\'d'.format(from_name))
plt.legend()
plt.grid()

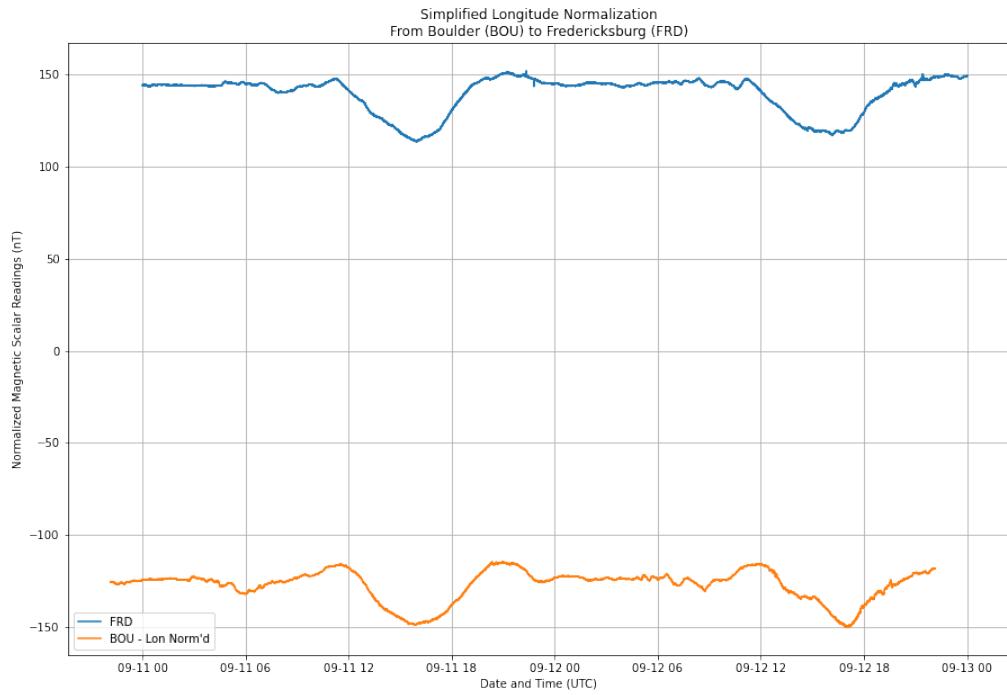
interp_lpf = interpolate.interp1d(bou_shift_t, from_f_no_core, 'cubic')

interp_mask = np.logical_and(to_t >= bou_shift_t.min(), to_t <= bou_shift_t.
    ↪max())
interp_t     = to_t(interp_mask) # Clip interpolation times

bou_shift_interp = interp_lpf(interp_t)

print('RMSE:', pu.rmse(to_f_no_core[interp_mask],
    bou_shift_interp))
```

RMSE: 266.84286256867483



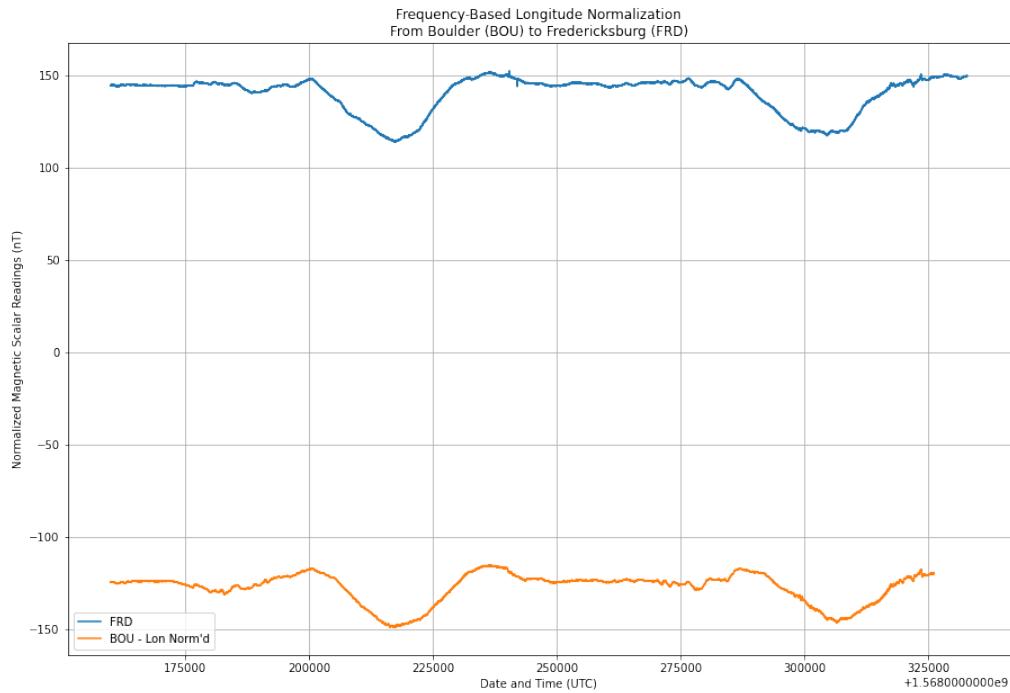
### 3 Frequency-Based Longitude Normalization

```
[ ]: from_combined_t, from_combined_f, _, _, _ = Diurnal.longitude_norm(from_df, to_df.LONG.mean())
from_combined_f_no_core = from_combined_f - from_IGRF_f

plt.figure()
plt.title('Frequency-Based Longitude Normalization\nFrom Boulder (BOU) to\nFredericksburg (FRD)')
plt.xlabel('Date and Time (UTC)')
plt.ylabel('Normalized Magnetic Scalar Readings (nT)')
plt.plot(to_df.epoch_sec, to_f_no_core, label=to_name)
plt.plot(from_combined_t, from_combined_f_no_core, label='{} - Lon Norm\''d''.format(from_name))
plt.legend()
plt.grid()

print('RMSE:', pu.rmse(to_f_no_core[interp_mask],
                      from_combined_f_no_core[:len(to_f_no_core[interp_mask])]))
```

RMSE: 266.87837777008167



## 4 Find Optimal Scale and Offset Parameters

```
[ ]: interp_combined = interpolate.interp1d(from_combined_t,
                                          from_combined_f_no_core, 'cubic')

interp_mask = np.logical_and(to_t >= from_combined_t.min(), to_t <=
                           from_combined_t.max())
interp_t      = to_t(interp_mask) # Clip interpolation times

from_combined_interp = interp_combined(interp_t)

offset, scale      = Diurnal.calibrate([0, 1], from_combined_interp,
                                         to_f_no_core(interp_mask))
from_combined_opt_f = Diurnal.apply_cal([offset, scale], from_combined_f_no_core)

print('Optimal scale:', scale)
print('Optimal offset:', offset)

plt.figure()
```

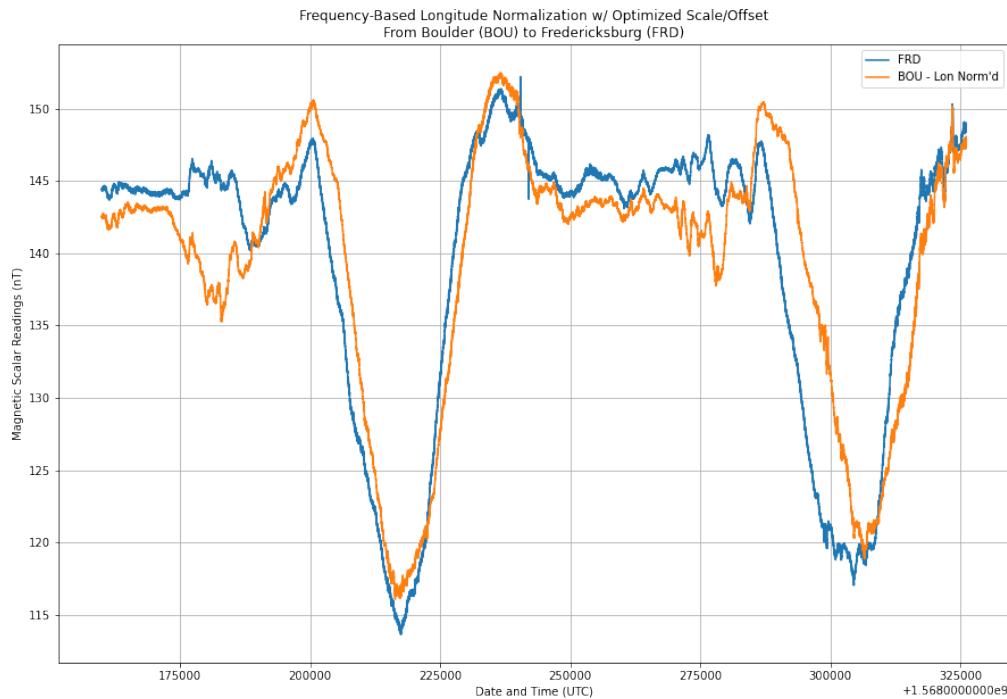
```
plt.title('Frequency-Based Longitude Normalization w/ Optimized Scale/  
-Offset\nFrom Boulder (BOU) to Fredericksburg (FRD)')  
plt.xlabel('Date and Time (UTC)')  
plt.ylabel('Magnetic Scalar Readings (nT)')  
plt.plot(to_df.epoch_sec[interp_mask], to_f_no_core[interp_mask], label=to_name)  
plt.plot(from_combined_t, from_combined_opt_f, label='{} - Lon Norm\''d'.  
format(from_name))  
plt.legend()  
plt.grid()  
  
print('RMSE:', pu.rmse(to_f_no_core[interp_mask],  
                      from_combined_opt_f[:len(to_f_no_core[interp_mask])]))  
print('STD:', (to_f_no_core[interp_mask] -  
              from_combined_opt_f[:len(to_f_no_core[interp_mask])]).std())
```

Optimal scale: 1.085134182785885

Optimal offset: 277.6792162949522

RMSE: 4.630734469499371

STD: 4.630728729444269



## 5 Test Optimal Parameters for Next Day's Data

```
[ ]: df_dict = pi.loadInterMagData(TEST_DIR)

from_name = 'BOU'
to_name   = 'FRD'

from_df = df_dict[from_name]
to_df   = df_dict[to_name]

to_df = pu.reject_outliers(to_df,
                           window_size=250,
                           std_lim=3)

to_IGRF_f    = np.array(to_df.IGRF_F)[0]
from_IGRF_f = np.array(from_df.IGRF_F)[0]

to_f    = np.array(to_df.F)
from_f = np.array(from_df.F)

to_f_no_core    = np.array(to_df.F) - to_IGRF_f
from_f_no_core = np.array(from_df.F) - from_IGRF_f

to_t    = np.array(to_df.epoch_sec)
from_t = np.array(from_df.epoch_sec)

plt.figure()
plt.title('Boulder (BOU) and Fredericksburg (FRD)\nNext Day Comparison')
plt.xlabel('Date and Time (UTC)')
plt.ylabel('Normalized Magnetic Scalar Readings (nT)')
plt.plot(to_df.datetime, to_f_no_core, label=to_name)
plt.plot(from_df.datetime, from_f_no_core, label=from_name)
plt.legend()
plt.grid()

from_combined_t, from_combined_f, _, _, _ = Diurnal.longitude_norm(from_df, to_df.LONG.mean())
from_combined_f_no_core = from_combined_f - from_IGRF_f

interp_combined = interpolate.interp1d(from_combined_t, from_combined_f_no_core, 'cubic')

interp_mask = np.logical_and(to_t >= from_combined_t.min(), to_t <= from_combined_t.max())
interp_t      = to_t[interp_mask] # Clip interpolation times
```

```

from_combined_interp = interp_combined(interp_t)

from_combined_opt_f = Diurnal.apply_cal([offset, scale], ↴
    ↪from_combined_f_no_core)

plt.figure()
plt.title('Frequency-Based Longitude Normalization w/ Optimized Scale/\
    ↪Offset\nFrom Boulder (BOU) to Fredericksburg (FRD) Next Day')
plt.xlabel('Date and Time (UTC)')
plt.ylabel('Magnetic Scalar Readings (nT)')
plt.plot(to_df.epoch_sec[interp_mask], to_f_no_core[interp_mask], label=to_name)
plt.plot(from_combined_t, from_combined_opt_f, label='{} - Lon Norm\''d'\
    ↪format(from_name)) )
plt.legend()
plt.grid()

print('RMSE:', pu.rmse(to_f_no_core, from_f_no_core[:len(to_f)])) # Extra\
    ↪slicing needed because dimensions don't match as a result of rejecting\
    ↪outliers
print('RMSE:', pu.rmse(to_f_no_core[interp_mask],\
    from_combined_opt_f[:len(to_f_no_core[interp_mask])]))
print('STD:', (to_f_no_core[interp_mask] -\
    from_combined_opt_f[:len(to_f_no_core[interp_mask])]).std())

```

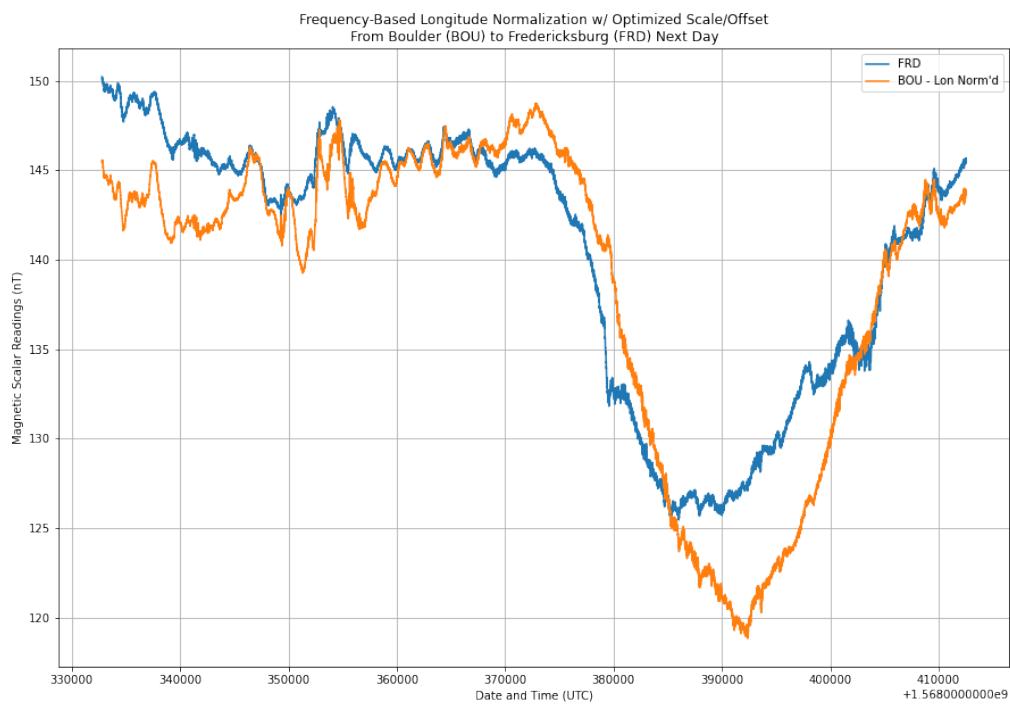
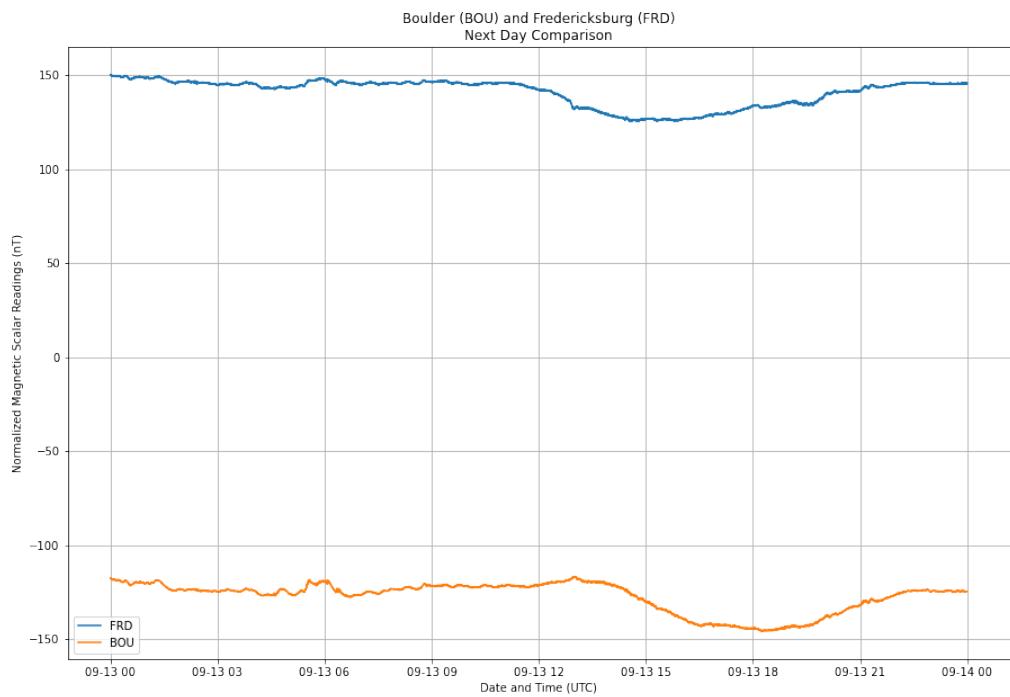
```

Loaded bou20190913psec.sec
Loaded frd20190913psec.sec
Loaded frn20190913psec.sec

0% | 0/346 [00:00<?, ?it/s]C:\Users\ltber\Anaconda3\Lib\site-
packages\pandas\core\dtypes\cast.py:2221: RuntimeWarning: invalid value
encountered in cast
    casted = element.astype(dtype)
100% | 346/346 [00:00<00:00, 584.44it/s]

RMSE: 268.2980777028462
RMSE: 3.635806103790872
STD: 3.2382384905629134

```





## generate\_data\_example

January 3, 2023

```
[ ]: import datetime as dt
import sys
from os import getcwd
from os.path import dirname, join

import matplotlib.pyplot as plt
import numpy as np
import pytz

import MAMMAL.Simulator as sim
from MAMMAL.Parse import parseIM as pim
from MAMMAL.Utils import coordinateUtils as cu
from MAMMAL.Utils import mapUtils as mu
from MAMMAL.VehicleCal import TL as tl

%matplotlib inline
plt.rcParams["figure.figsize"] = (25, 20) # (w, h)

debug = True # Set to True to enable debug printouts plus plots
```

## 1 Simulated Spin Test Parameters

```
[ ]: spin_out_dir      = getcwd()
spin_lat            = 38.205 # Fredericksburg (dd)
spin_lon            = -77.373 # Fredericksburg (dd)
spin_height_m       = 69      # Height of Fredericksburg above MSL
spin_start_dt       = dt.datetime(2019, 9, 12, 8, 40, 0, tzinfo=pytz.utc)
spin_start_dt       = dt.datetime.fromtimestamp(spin_start_dt.timestamp())
spin_headings      = np.linspace(0, 720, 1000)
spin_elevations    = np.linspace(0, 7200, 1000)
spin_a              = np.array([[0.1, 0.0, 0.0],
                               [0.0, 0.2, 0.0],
                               [0.0, 0.0, 1.0]])
spin_b              = np.array([1, 10, 20])
```

## 2 Simulated Tolles-Lawson Box Flight Parameters

```
[ ]: tl_out_dir      = getcwd()
tl_center_lat    = 38.205 # Fredericksburg (dd)
tl_center_lon    = -77.373 # Fredericksburg (dd)
tl_height_m      = 2000
tl_start_dt      = dt.datetime(2019, 9, 12, 8, 40, 0, tzinfo=pytz.utc)
tl_start_dt      = dt.datetime.fromtimestamp(tl_start_dt.timestamp())
tl_box_xlen_m    = 500
tl_box_ylen_m    = 1000
tl_c              = np.array([0, # Perm x
                                0, # Perm y
                                0, # Perm z
                                0, # Ind xx
                                0, # Ind yy
                                0, # Ind zz
                                0, # Ind xy
                                0, # Ind xz
                                0, # Ind yz
                                0, # Eddy xx'
                                0, # Eddy yy'
                                0, # Eddy yx'
                                0, # Eddy yx'
                                0, # Eddy zx'
                                0, # Eddy xy'
                                0, # Eddy zy'
                                0, # Eddy xz'
                                0]) # Eddy yz'

tl_vel_mps       = 20
tl_sample_hz     = 50
tl_dither_hz     = 1
tl_dither_amp    = 10
tl_terms         = tl.ALL_TERMS
```

## 3 Simulated Reference Station Data Parameters

```
[ ]: ref_out_dir    = getcwd()
ref_lat          = 38.205 # Fredericksburg (dd)
ref_lon          = -77.373 # Fredericksburg (dd)
ref_height_m     = 69      # Height of Fredericksburg above MSL
ref_start_dt     = dt.datetime(2019, 9, 12, 8, 40, 0, tzinfo=pytz.utc)
ref_start_dt     = dt.datetime.fromtimestamp(ref_start_dt.timestamp())
ref_dur_s        = 10000
ref_scale        = 1
ref_offset        = 0
ref_awgn_std     = 0
```

```

ref_sample_hz = 1
ref_id        = 'FRD'
ref_in_dir    = getcwd()
if ref_id is not None and ref_in_dir is not None:
    ref_file_df = pim.loadInterMagData(ref_in_dir)[ref_id]
else:
    ref_file_df = None

```

```

Loaded bou20190911psec.sec
Loaded bou20190912psec.sec
Loaded frd20190911psec.sec
Loaded frd20190912psec.sec
Loaded frn20190911psec.sec
Loaded frn20190912psec.sec

```

## 4 Simulated Anomaly Map Parameters

```

[ ]: map_out_dir      = getcwd()
map_loc_name       = 'test'
map_center_lat     = 38.205 # Fredericksburg (dd)
map_center_lon     = -77.373 # Fredericksburg (dd)
map_height_agl_m   = 30.48 # 100ft AGL
map_height_m        = 69 + map_height_agl_m # Height of Fredericksburg above MSL
#+ survey height AGL
map_upcontinue     = False
map_x_dist_m       = 300
map_y_dist_m       = 300
map_dx_m           = map_height_agl_m / 20
map_dy_m           = map_height_agl_m / 20
map_start_dt       = dt.datetime(2019, 9, 12, 8, 40, 0, tzinfo=pytz.utc)
map_start_dt       = dt.datetime.fromtimestamp(map_start_dt.timestamp())
map_anomaly_locs   = np.array([[map_center_lat], # dd
                                [map_center_lon]]) # dd
map_anomaly_scales = np.array([20]) # nT
map_anomaly_covs   = np.zeros((1, 2, 2))
map_anomaly_covs[0, :, :] = np.diag([0.000001, 0.000002])

```

## 5 Simulated Survey Flight Parameters

```

[ ]: survey_out_dir      = getcwd()
survey_height_m         = map_height_m
survey_start_dt         = dt.datetime(2019, 9, 12, 8, 40, 0, tzinfo=pytz.utc)
survey_start_dt         = dt.datetime.fromtimestamp(survey_start_dt.timestamp())
survey_vel_mps          = 19
survey_e_buff_m          = 15.24 # 50ft
survey_w_buff_m          = 15.24 # 50ft

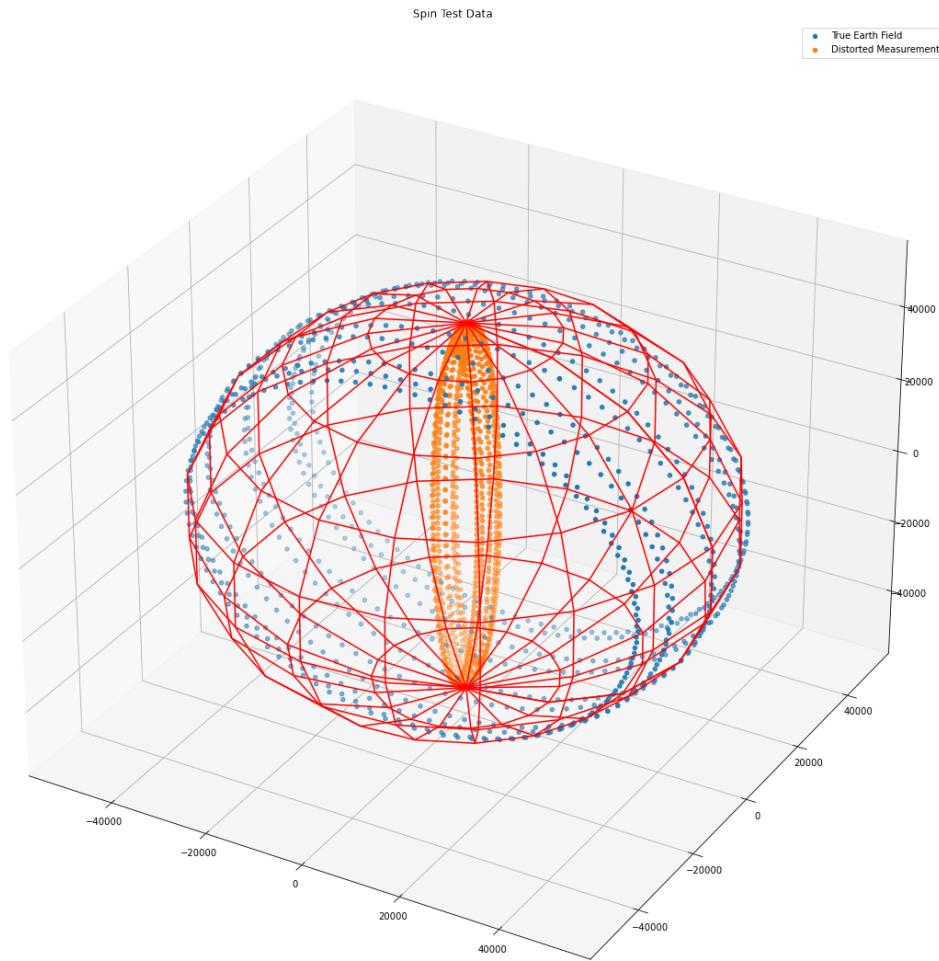
```

```
survey_n_buff_m      = 15.24 # 50ft
survey_s_buff_m      = 15.24 # 50ft
survey_sample_hz     = 5
survey_ft_line_dist_m = map_height_agl_m / 2
survey_ft_line_dir   = sim.HORIZ
survey_scalar_awgn_std = 0
survey_use_tie_lines = True
survey_tie_dist_m    = survey_ft_line_dist_m * 5
```

## 6 Generate Simulated Spin Test Data

```
[ ]: spin_df = sim.gen_spin_data(out_dir      = spin_out_dir,
                                lat          = spin_lat,
                                lon          = spin_lon,
                                height       = spin_height_m,
                                date         = spin_start_dt,
                                headings     = spin_headings,
                                elevations   = spin_elevations,
                                a            = spin_a,
                                b            = spin_b,
                                debug        = debug)
```

```
Generating simulated spin test data
Generating perfect simulated spin test measurements
Applying vector distortion to simulated spin test data
Calculating IGRF values for simulated spin test
Exporting simulated spin test data as a CSV
Saved data to c:\Users\ltber\Downloads\mammal-Beta\mammal-
Beta\data\test\spin_2019_9_12_0.csv
```



## 7 Generate Simulated Tolles-Lawson Box Flight Data

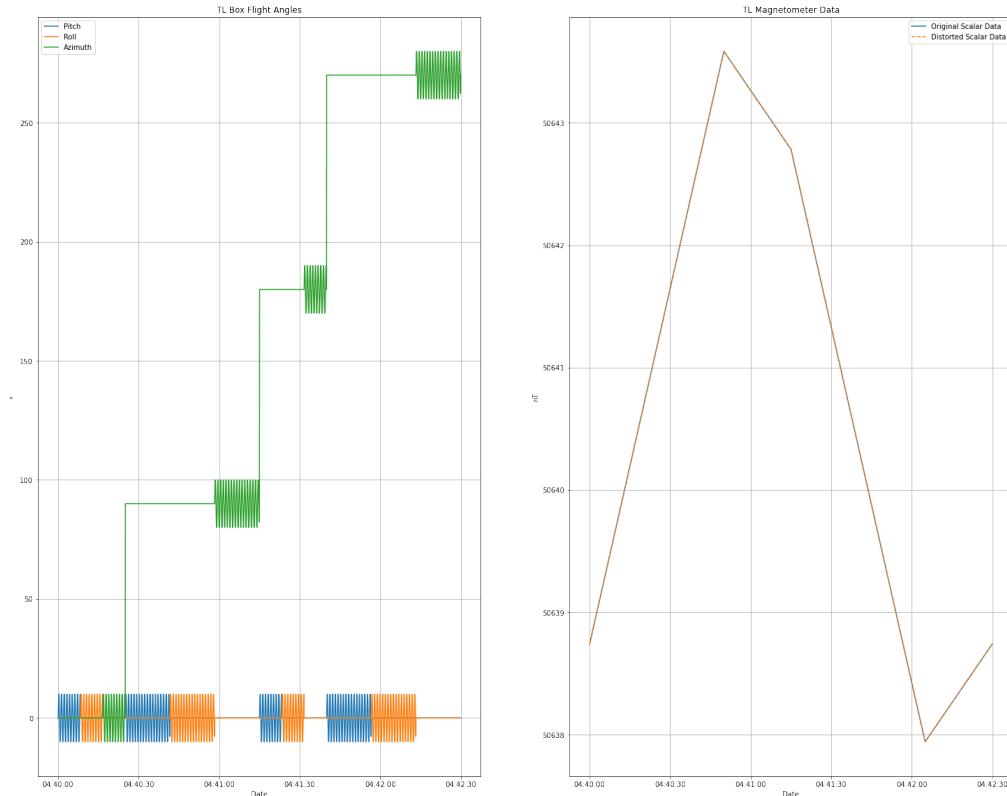
```
[ ]: tl_df = sim.gen_TL_data(out_dir      = tl_out_dir,
                           center_lat = tl_center_lat,
                           center_lon = tl_center_lon,
                           height     = tl_height_m,
                           start_dt   = tl_start_dt,
                           box_xlen_m = tl_box_xlen_m,
                           box_ylen_m = tl_box_ylen_m,
                           c          = tl_c,
                           vel_mps    = tl_vel_mps,
```

```

sample_hz = tl_sample_hz,
dither_hz = tl_dither_hz,
dither_amp = tl_dither_amp,
terms      = tl_terms,
a          = spin_a,
b          = spin_b,
debug      = debug)

```

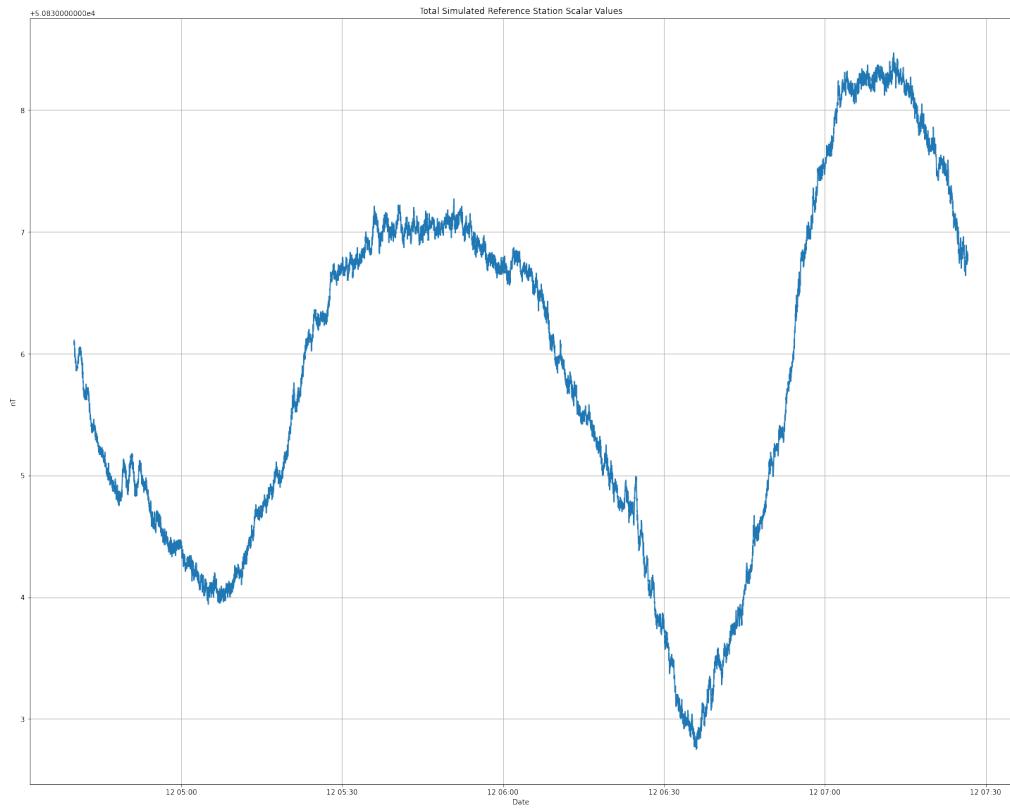
Generating simulated TL calibration flight data  
 Calculating IGRF values for simulated TL calibration flight  
 Dithering orientation angles (1 Hz,  $\pm 10^\circ$ )  
 Dithering pitch angles  
 Dithering roll angles  
 Dithering azimuth angles  
 Generating true TL readings (assuming no anomaly - only IGRF is used)  
 Applying TL distortion to simulated calibration flight data  
 Applying spin test distortion to simulated simulated TL readings  
 Exporting simulated TL flight data as a CSV  
 Saved data to c:\Users\ltber\Downloads\mammal-Beta\mammal-Beta\data\test\tl\_2019\_9\_12\_0.csv



## 8 Generate Simulated Reference Station Data

```
[ ]: ref_df = sim.gen_ref_station_data(out_dir      = ref_out_dir,
                                         lat        = ref_lat,
                                         lon        = ref_lon,
                                         height     = ref_height_m,
                                         start_dt   = ref_start_dt,
                                         dur_s      = ref_dur_s,
                                         scale      = ref_scale,
                                         offset     = ref_offset,
                                         awgn_std   = ref_awgn_std,
                                         sample_hz  = ref_sample_hz,
                                         file_df    = ref_file_df,
                                         debug      = debug)
```

```
Generating simulated reference station data
Incorporating file data into simulated reference data
Incorporating scale and offset into simulated reference data
Incorporating AWGN into simulated reference data
Calculating IGRF values at simulated reference station
Adding IGRF core field to simulated reference station scalar values
Projecting simulated reference station scalar values into vector values using
IGRF direction cosines
Exporting simulated reference station data as a CSV
Saved data to c:\Users\ltber\Downloads\mammal-Beta\mammal-
Beta\data\test\ref_2019_9_12_0.csv
```

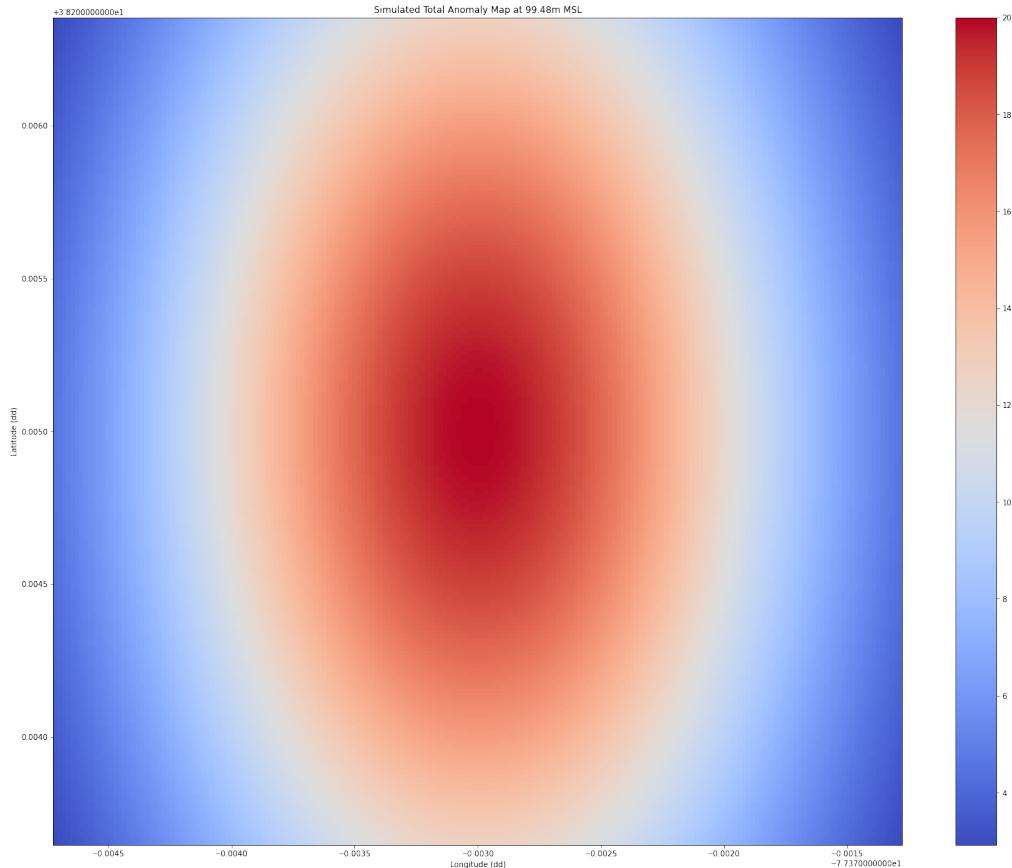


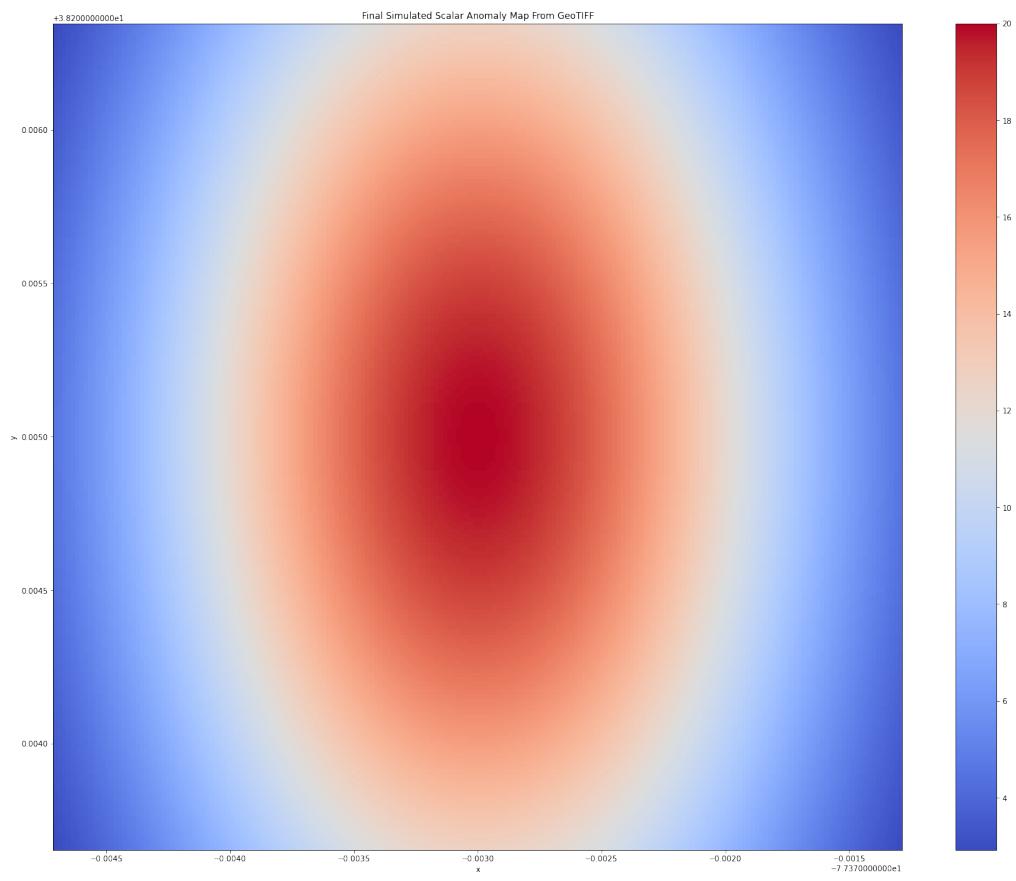
## 9 Generate Simulated Anomaly Map

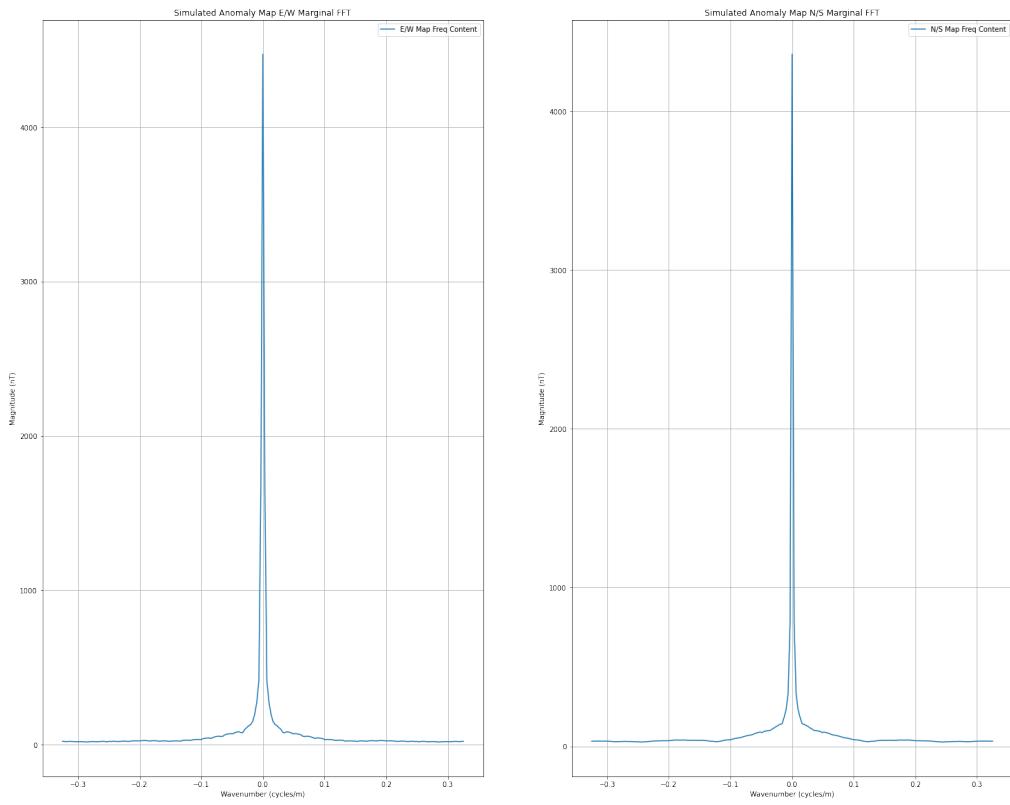
```
[ ]: sim_map = sim.gen_sim_map(out_dir      = map_out_dir,
                             location       = map_loc_name,
                             center_lat     = map_center_lat,
                             center_lon     = map_center_lon,
                             dx_m           = map_dx_m,
                             dy_m           = map_dy_m,
                             x_dist_m       = map_x_dist_m,
                             y_dist_m       = map_y_dist_m,
                             height         = map_height_m,
                             date           = map_start_dt,
                             anomaly_locs   = map_anomaly_locs,
                             anomaly_scales = map_anomaly_scales,
                             anomaly_covs   = map_anomaly_covs,
                             upcontinue     = map_upcontinue,
                             debug          = debug)
```

```
if debug:  
    mu.plt_freqs(sim_map[0], 'Simulated Anomaly')
```

Generating simulated anomaly map  
Processing simulated anomaly structures at MSL  
Calculating IGRF values for simulated map  
Projecting simulated map scalar measurements into vector measurements using IGRF  
direction cosines  
Exporting simulated map as a GeoTIFF







## 10 Generate Simulated Survey Flight Data

```
[ ]: survey_df = sim.gen_survey_data(out_dir           = map_out_dir,
                                    map               = sim_map,
                                    survey_height_m = survey_height_m,
                                    survey_start_dt = survey_start_dt,
                                    survey_vel_mps  = survey_vel_mps,
                                    survey_e_buff_m = survey_e_buff_m,
                                    survey_w_buff_m = survey_w_buff_m,
                                    survey_n_buff_m = survey_n_buff_m,
                                    survey_s_buff_m = survey_s_buff_m,
                                    sample_hz        = survey_sample_hz,
                                    ft_line_dist_m  = survey_ft_line_dist_m,
                                    ft_line_dir     = survey_ft_line_dir,
                                    a                = spin_a,
                                    b                = spin_b,
                                    c                = tl_c,
                                    terms            = tl_terms,
```

```

        scalar_awgn_std = survey_scalar_awgn_std,
        diurnal_df      = ref_df,
        diurnal_dist    = np.array([0, 1]), # [offset]
        ↵(nT), scale]

        use_tie_lines   = survey_use_tie_lines,
        tie_dist_m     = survey_tie_dist_m,
        debug          = debug)

```

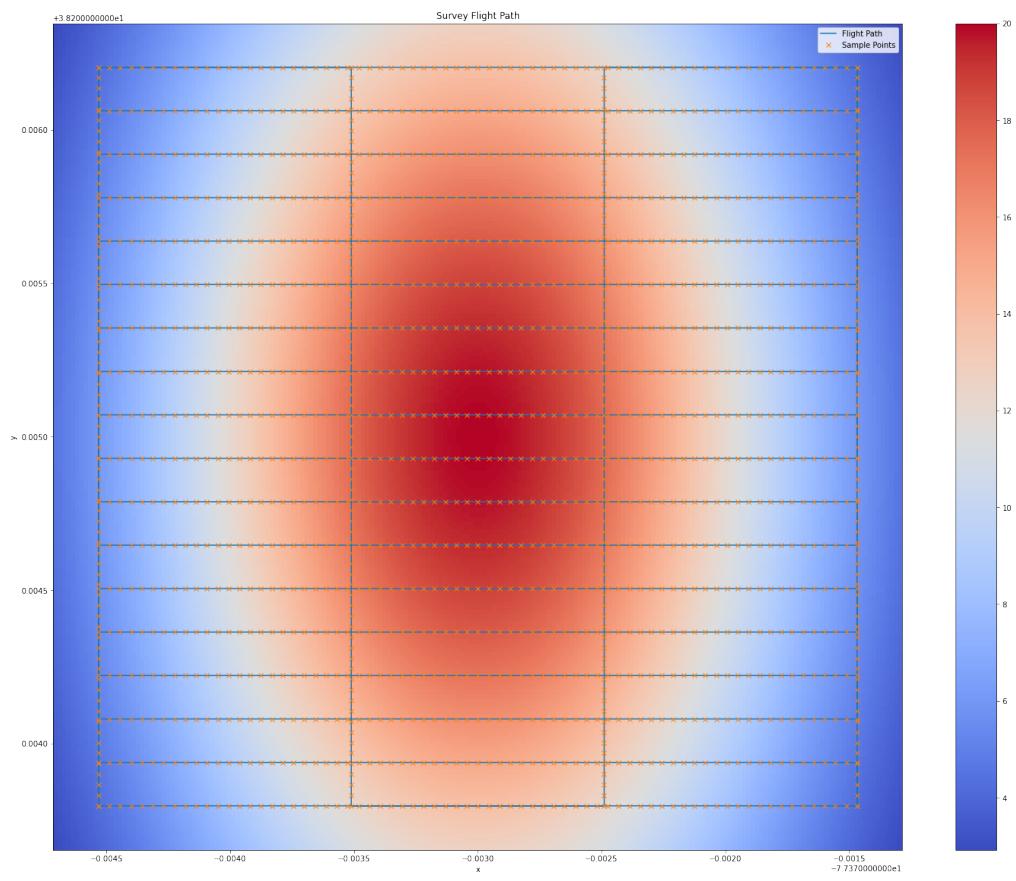
```

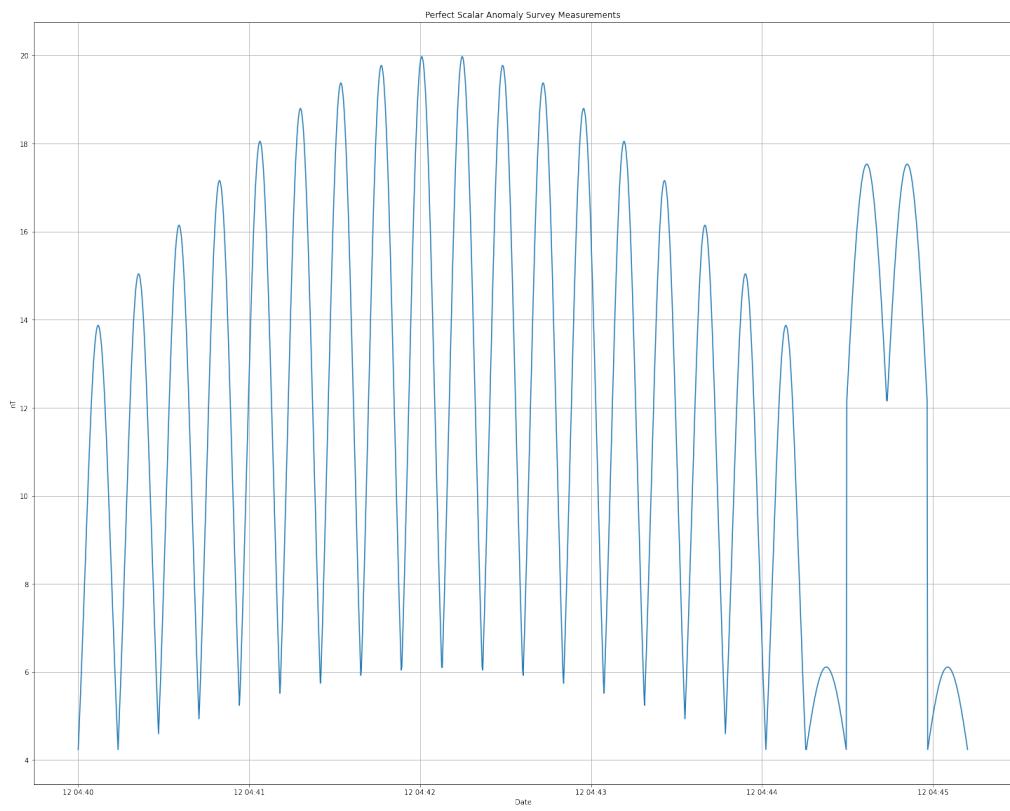
Generating simulated survey data
Generating simulated survey flight path
Calculating simulated survey scalar anomaly, azimuth, and IGRF values
100%|      | 1562/1562 [00:14<00:00, 110.19it/s]

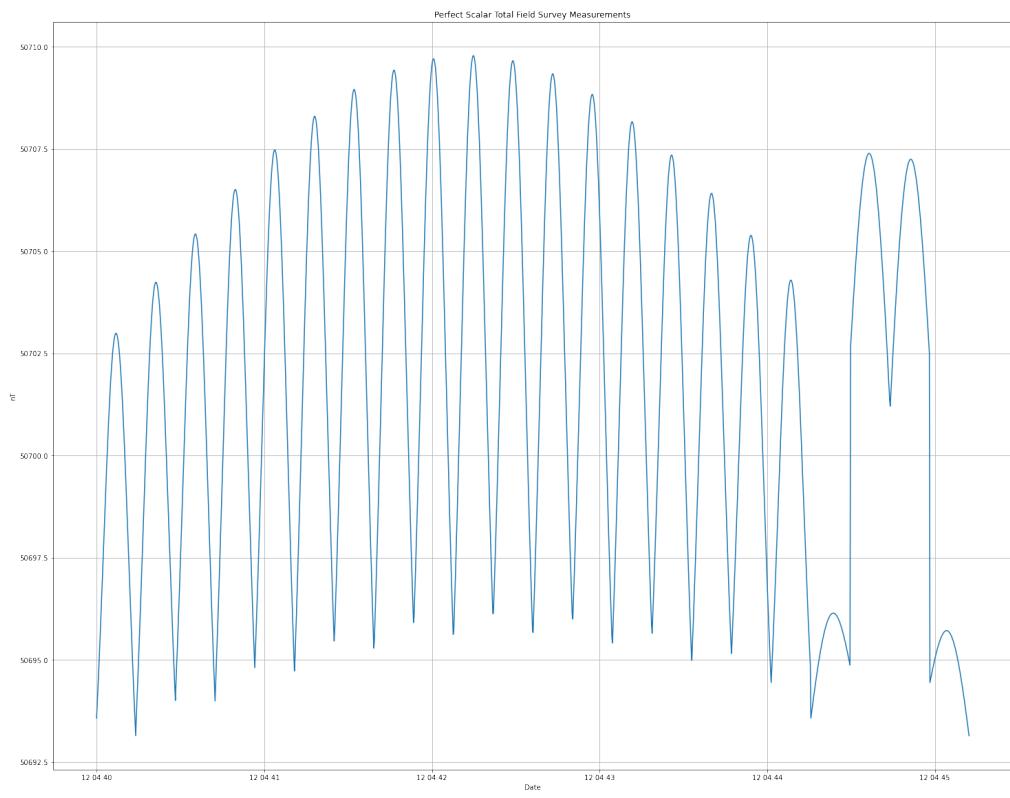
Adding core field to simulated survey scalar measurements
100%|      | 1561/1561 [00:00<00:00, 520164.34it/s]

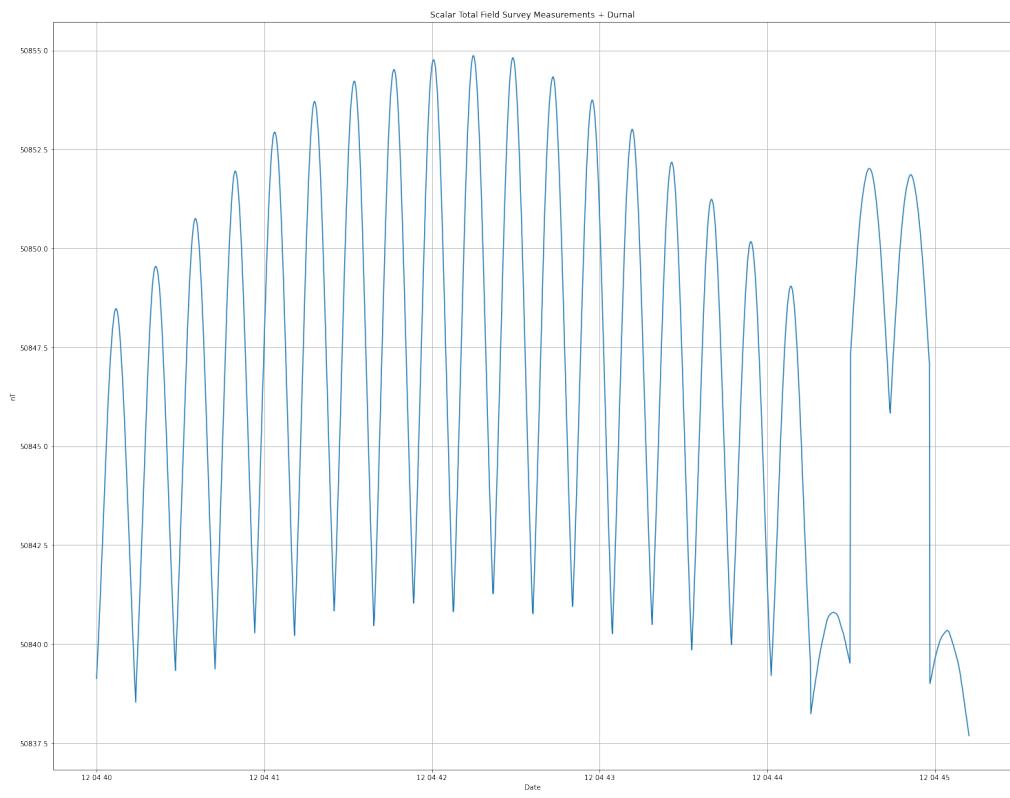
Adding diurnal to simulated survey scalar measurements
Applying TL distortion to simulated survey scalar measurements
Projecting simulated survey scalar measurements into NED vector measurements
using IGRF direction cosines
Rotating NED vector measurements into sensor's body frame
Applying spin test distortion to simulated survey vector measurements
Exporting simulated survey data as a CSV
Survey start datetime/timestamp: 2019-09-12 04:40:00/1568277600.0s
Survey end datetime/timestamp: 2019-09-12 04:45:12.200000/1568277952.6181314s
Flight line samples end at timestamp: 1568277600.402999s
Survey Duration: 352.61813139915466s
Saved data to c:\Users\ltber\Downloads\mammal-Beta\mammal-
Beta\data\test\survey_2019_9_12_0.csv

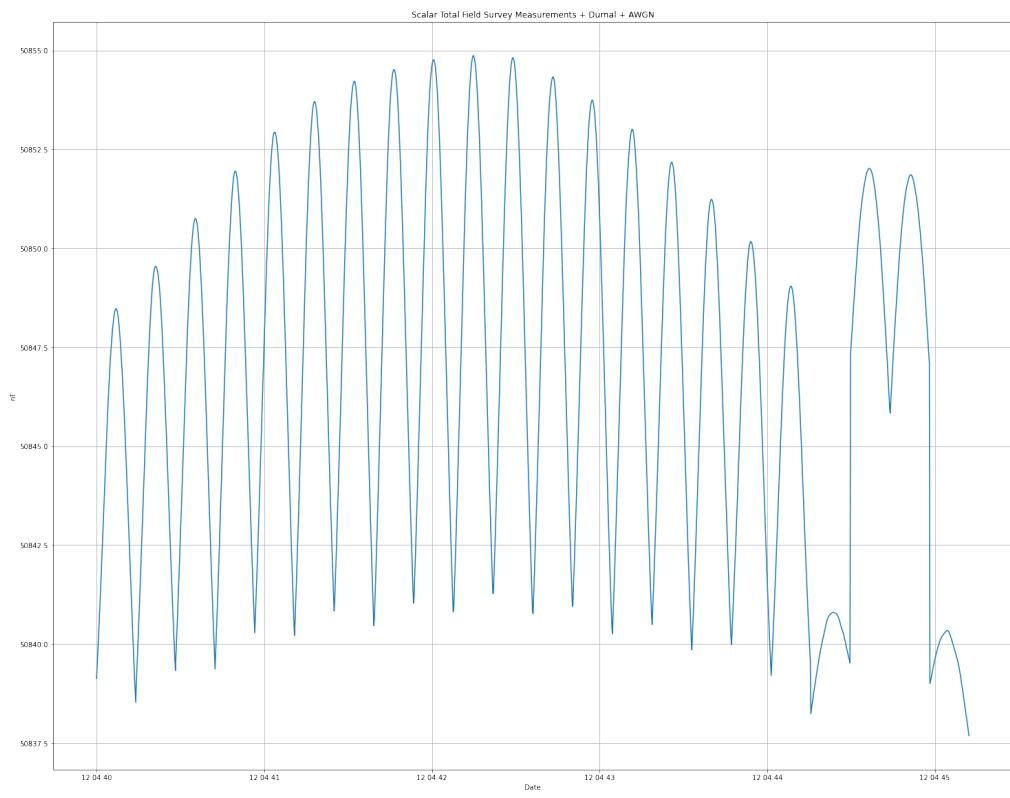
```

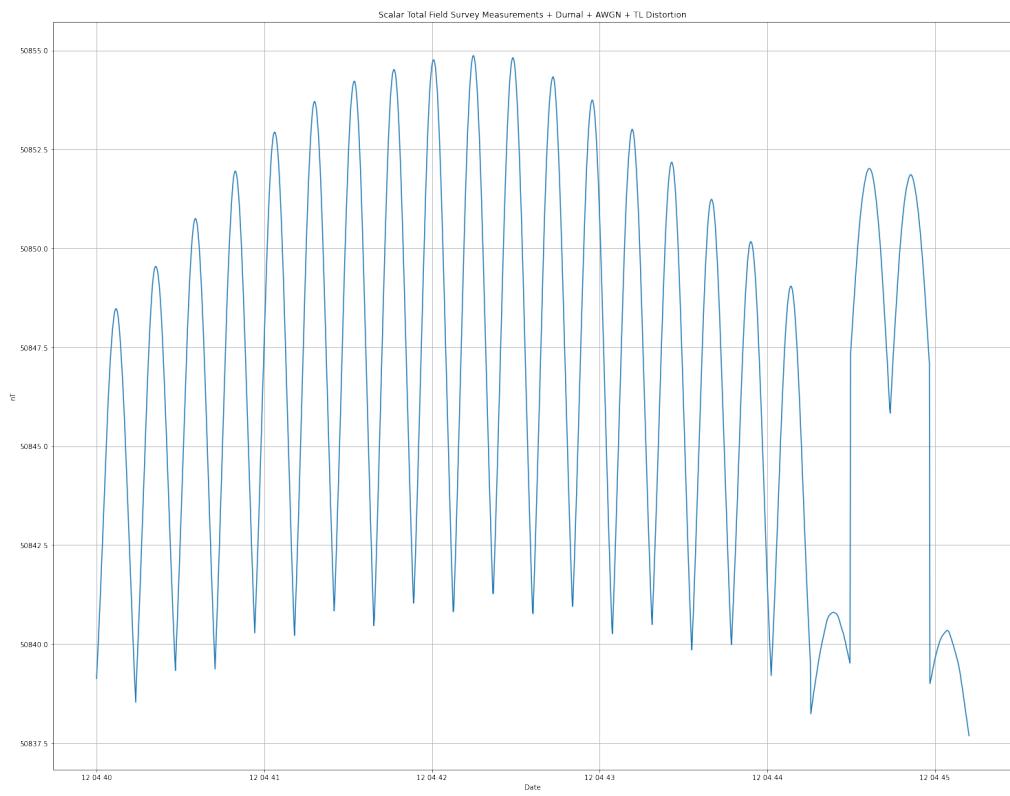












# generate\_map\_example

January 3, 2023

```
[ ]: import sys
import datetime as dt
from copy import deepcopy
from os import getcwd
from os.path import dirname, join

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from matplotlib import cm
from scipy import interpolate
import pytz

import MAMMAL
import MAMMAL.Diurnal as Diurnal
from MAMMAL.Parse import parseIM as pim
from MAMMAL.Parse import parseRaster as pr
from MAMMAL.Utils import mapUtils as mu
from MAMMAL.Utils import ProcessingUtils as pu

%matplotlib inline
plt.rcParams["figure.figsize"] = (15, 10) # (w, h)

debug = False # Set to True to enable debug printouts plus plots
```

## 1 Map-Maker Initialization

```
[ ]: loc_alt_msl_m      = 69 # Height of Fredericksburg above MSL
survey_alt_agl_m = 30 # Height of survey AGL
survey_alt_msl_m = loc_alt_msl_m + survey_alt_agl_m

survey = MAMMAL.MapMaker(map_loc_name = 'reconstructed_example',
                        alt_m_msl      = survey_alt_msl_m,
                        alt_m_agl      = survey_alt_agl_m,
                        data_dir       = getcwd(),
```

```

        debug           = debug)

survey.interp_type = 'rbf'

survey.spin_fname  = join(getcwd(), r'spin_2019_9_12_0.csv')
survey.tl_fname    = join(getcwd(), r'tl_2019_9_12_0.csv')
survey.ref_fname   = join(getcwd(), r'ref_2019_9_12_0.csv')
survey.survey_fname = join(getcwd(), r'survey_2019_9_12_0.csv')

```

## 2 Load Truth Map

```
[ ]: TRUTH_MAP_FNAME = join(getcwd(), r'example_99m_2019_9_12_0.tif')

truth_map = pr.parse_raster(TRUTH_MAP_FNAME)
```

## 3 Find Spin Test Calibration Parameters

```
[ ]: a, b = survey.spin_params()

print(a)
print(b)

[[ 9.9999999e-02  5.57695083e-10 -2.91527922e-11]
 [-2.27964052e-10  2.00000000e-01  1.30701651e-09]
 [ 2.65361785e-11  1.86757457e-08  1.00000000e+00]]
[ 0.99998747  9.99999273  20.00002557]
```

## 4 Find Tolles-Lawson Calibration Parameters

```
[ ]: c = survey.tl_params()

print(c)

[-2.53712549e-03  7.48175161e-03 -3.11947526e-02 -3.41711931e-07
 -3.18443778e-07  6.60171150e-07  8.37594903e-08  5.57889498e-08
 -1.99531115e-09 -2.05085081e-07 -3.78049611e-07 -4.81279418e-07
 -5.74745693e-08  1.83084903e-08 -8.09687986e-08  1.49256175e-07
 -1.30927883e-07  8.88101495e-08]
```

## 5 Generate Map Using “Truth” Reference Station Data (Fredricksburg - FRD)

```
[ ]: map = survey.gen_map()

interp_map = map[mu.SCALAR].interp(x=truth_map.x, y=truth_map.y)

plt.figure()
interp_map.plot(cmap=cm.coolwarm)
plt.title('Created Map')

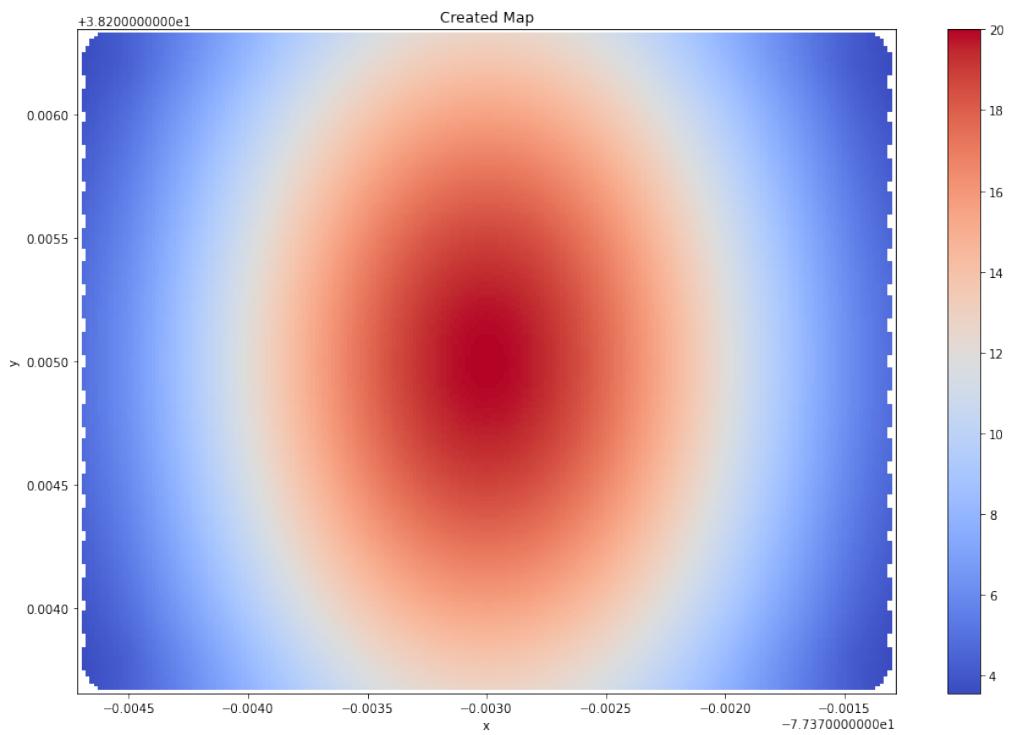
plt.figure()
truth_map[0].plot(cmap=cm.coolwarm)
plt.title('Truth Map')

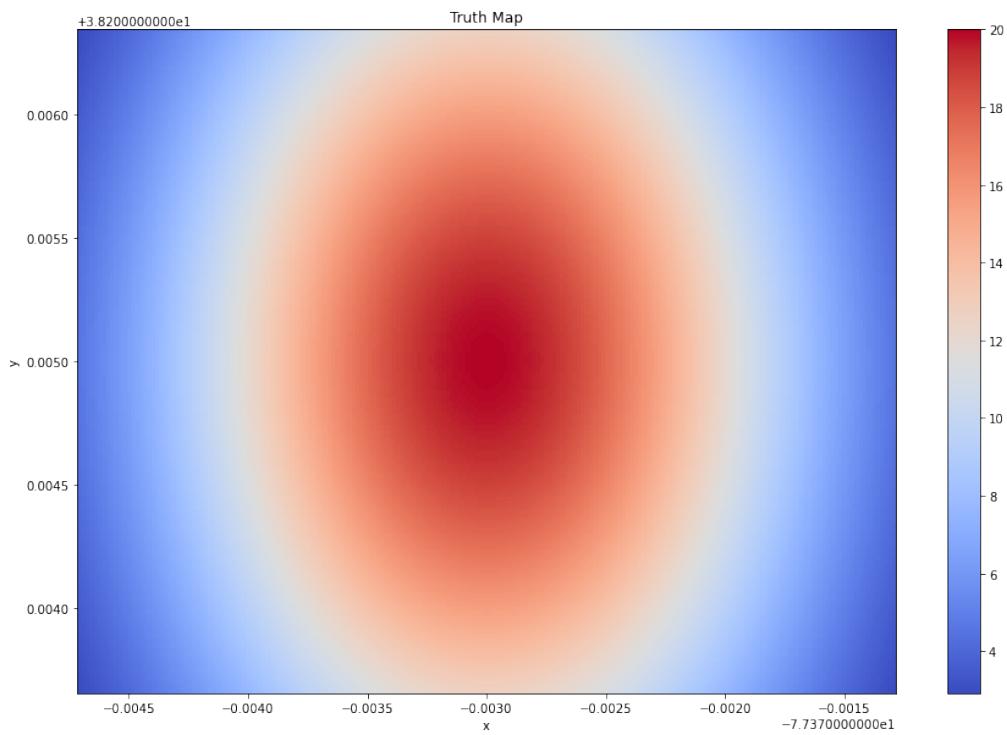
plt.figure()
error_map = truth_map[mu.SCALAR] - interp_map
error_map.plot(cmap=cm.coolwarm)
plt.title('Difference Between Created and Truth Maps')

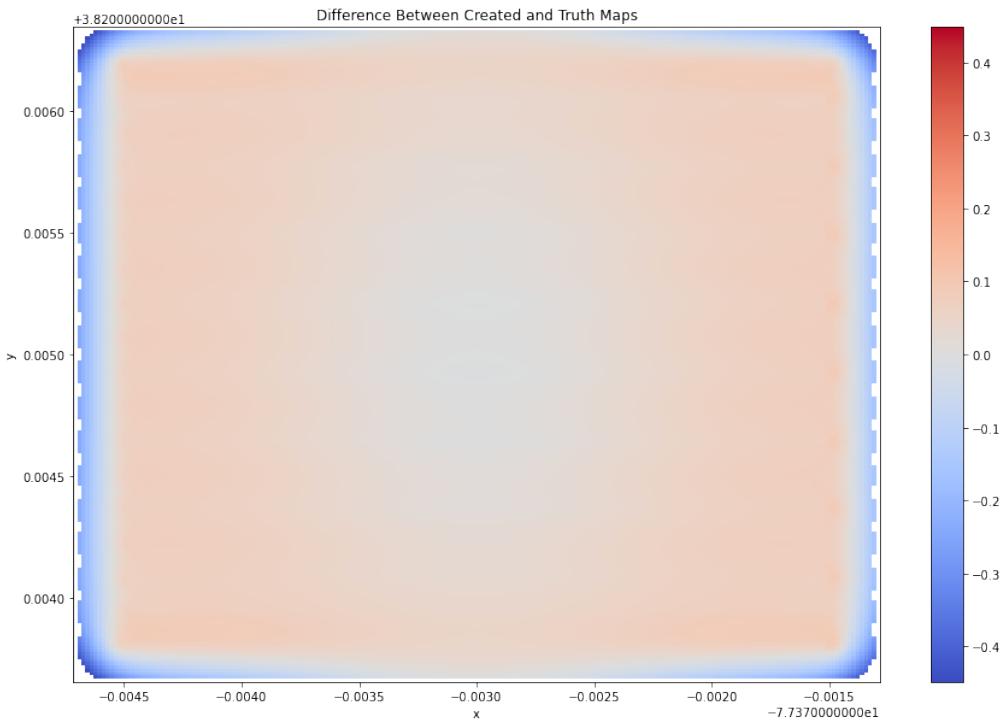
map_rmse = pu.rmse(interp_map.data, truth_map[mu.SCALAR].data)
print('Generated scalar map RMSE: {}nT'.format(map_rmse))
```

100% | 39204/39204 [00:02<00:00, 16934.57it/s]

Generated scalar map RMSE: 0.06657043985851316nT







## 6 Generate Map Without Reference Station Data

```
[ ]: survey.map_loc_name = 'reconstructed_no_ref'
survey.ref_df          = None

map = survey.gen_map(ref_use_internal=True)

interp_map = map[mu.SCALAR].interp(x=truth_map.x, y=truth_map.y)

plt.figure()
interp_map.plot(cmap=cm.coolwarm)
plt.title('Created Map')

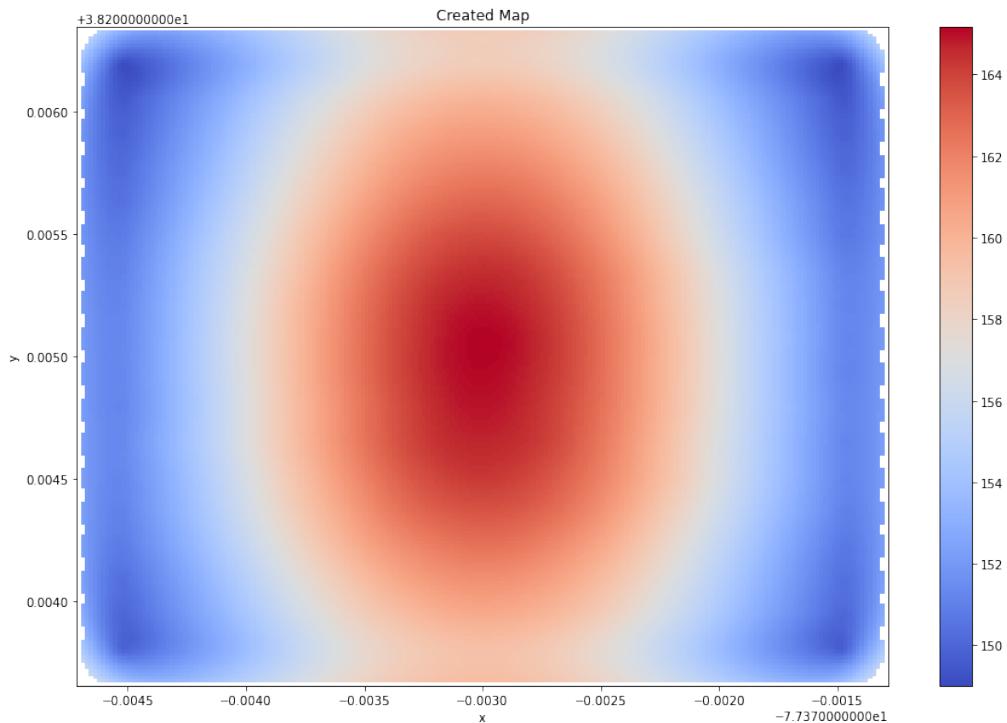
plt.figure()
truth_map[0].plot(cmap=cm.coolwarm)
plt.title('Truth Map')

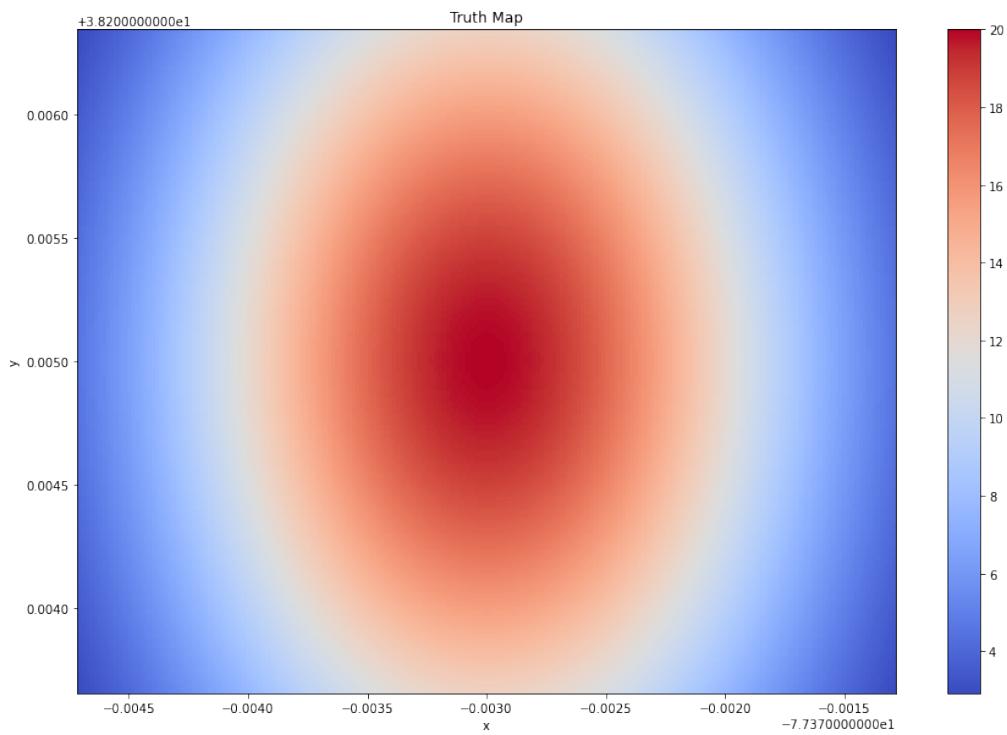
plt.figure()
error_map = truth_map[mu.SCALAR] - interp_map
error_map.plot(cmap=cm.coolwarm)
plt.title('Difference Between Created and Truth Maps')
```

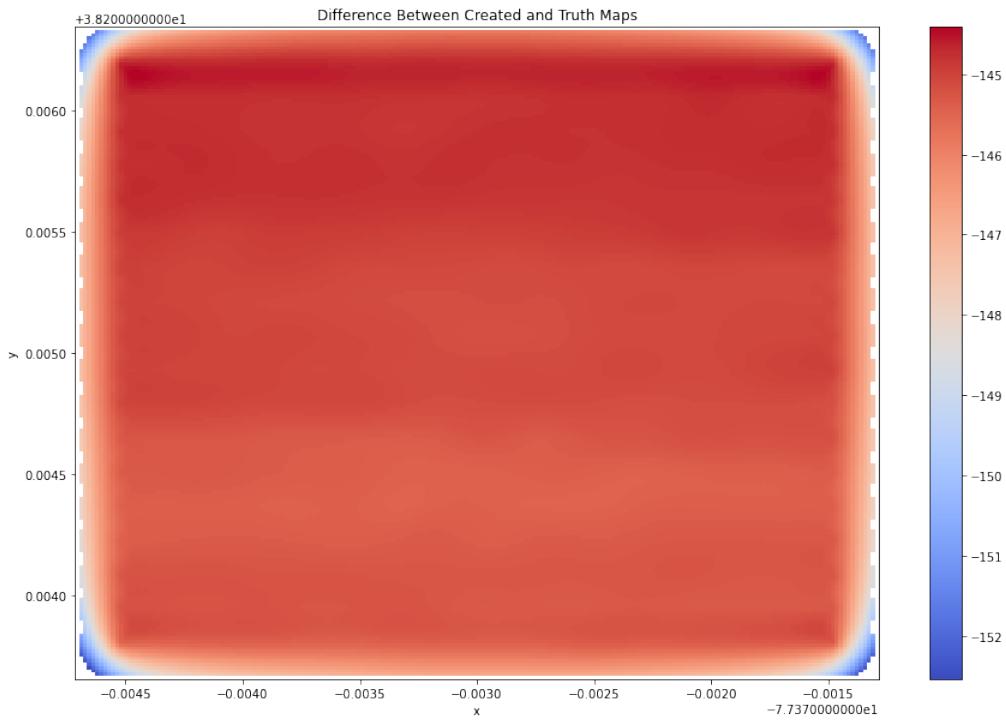
```
map_rmse = pu.rmse(interp_map.data, truth_map[mu.SCALAR].data)
print('Generated scalar map RMSE: {}nT'.format(map_rmse))
```

100% | 39204/39204 [00:02<00:00, 17162.57it/s]

Generated scalar map RMSE: 145.30984423308965nT







## 7 Generate Map Using Calibrated Reference Station Data from Boulder (BOU) - 2400km from FRD

```
[ ]: ref_dict = pim.loadInterMagData(TEST_DIR)

truth_ref_df = ref_dict['FRD']
bou_ref_df   = ref_dict['BOU']
frn_ref_df   = ref_dict['FRN']

truth_t       = np.array(truth_ref_df.epoch_sec)
truth_f       = np.array(truth_ref_df.F)
truth_IGRF_f  = np.array(truth_ref_df.IGRF_F)[0]
truth_f_no_core = truth_f - truth_IGRF_f

bou_t, bou_f, _, _, _ = Diurnal.longitude_norm(bou_ref_df,
                                                truth_ref_df.LONG.mean())
bou_f_no_core = bou_f - np.array(bou_ref_df.IGRF_F)[0]

interp_combined = interpolate.interp1d(bou_t, bou_f_no_core, 'cubic')
```

```

survey_t      = np.array(survey.survey_df.epoch_sec)
interp_mask = np.logical_and(np.logical_and(truth_t >= bou_t.min(),
                                             truth_t <= bou_t.max()),
                            np.logical_and(truth_t >= (survey_t[0] - 86400 - u
↳2*3600),
                                           truth_t <= (survey_t[-1] - 86400 + u
↳2*3600))) # Interpolate based on data from around the time of the survey on↳
↳the previous day
interp_t      = truth_t[interp_mask] # Clip interpolation times

far_interp = interp_combined(interp_t)

offset, scale = Diurnal.calibrate([0, 1], far_interp,↳
↳truth_f_no_core[interp_mask])
far_opt_f     = Diurnal.apply_cal([offset, scale], bou_f_no_core)

print('Optimal scale:', scale)
print('Optimal offset:', offset)

survey.map_loc_name = 'reconstructed_bou_ref'
survey.ref_scale    = scale
survey.ref_offset   = offset

map = survey.gen_map(ref_df=bou_ref_df)

interp_map = map[mu.SCALAR].interp(x=truth_map.x, y=truth_map.y)

plt.figure()
interp_map.plot(cmap=cm.coolwarm)
plt.title('Created Map')

plt.figure()
truth_map[0].plot(cmap=cm.coolwarm)
plt.title('Truth Map')

plt.figure()
error_map = truth_map[mu.SCALAR] - interp_map
error_map.plot(cmap=cm.coolwarm)
plt.title('Difference Between Created and Truth Maps')

map_rmse = pu.rmse(interp_map.data, truth_map[mu.SCALAR].data)
print('Generated scalar map RMSE: {}nT'.format(map_rmse))

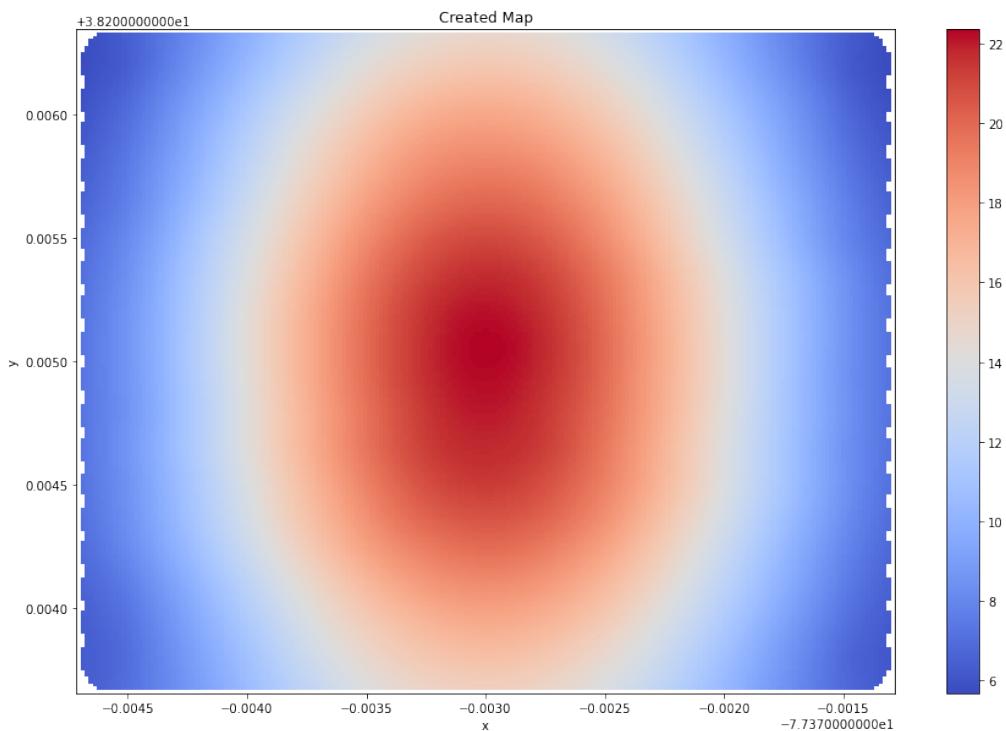
# Clean up
survey.ref_scale = 1
survey.ref_offset = 0

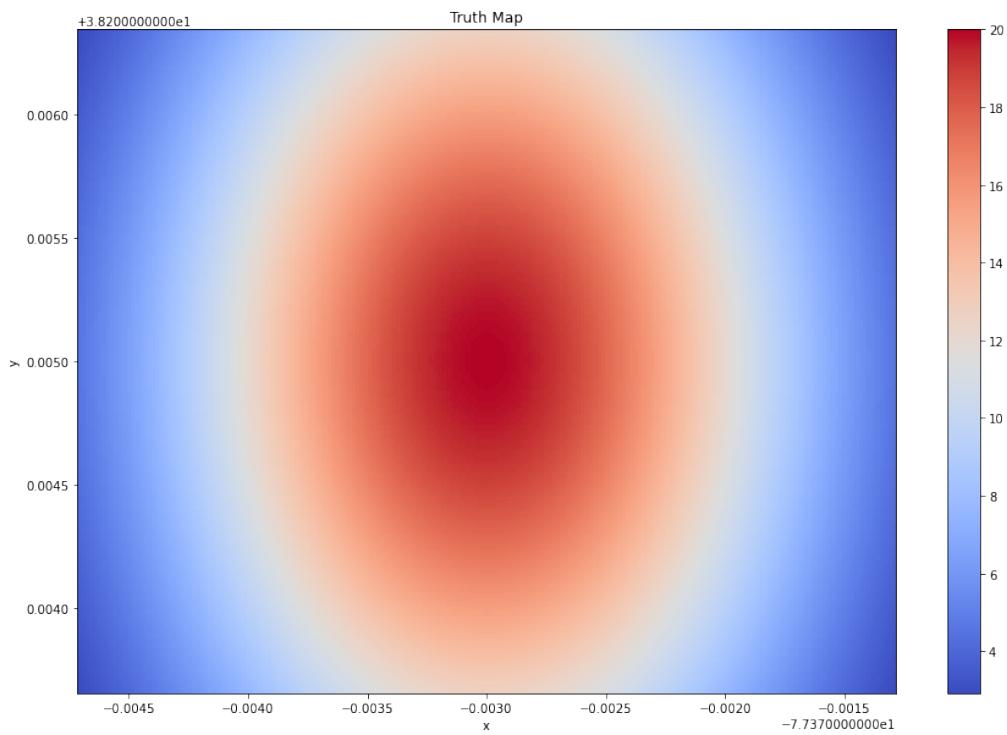
```

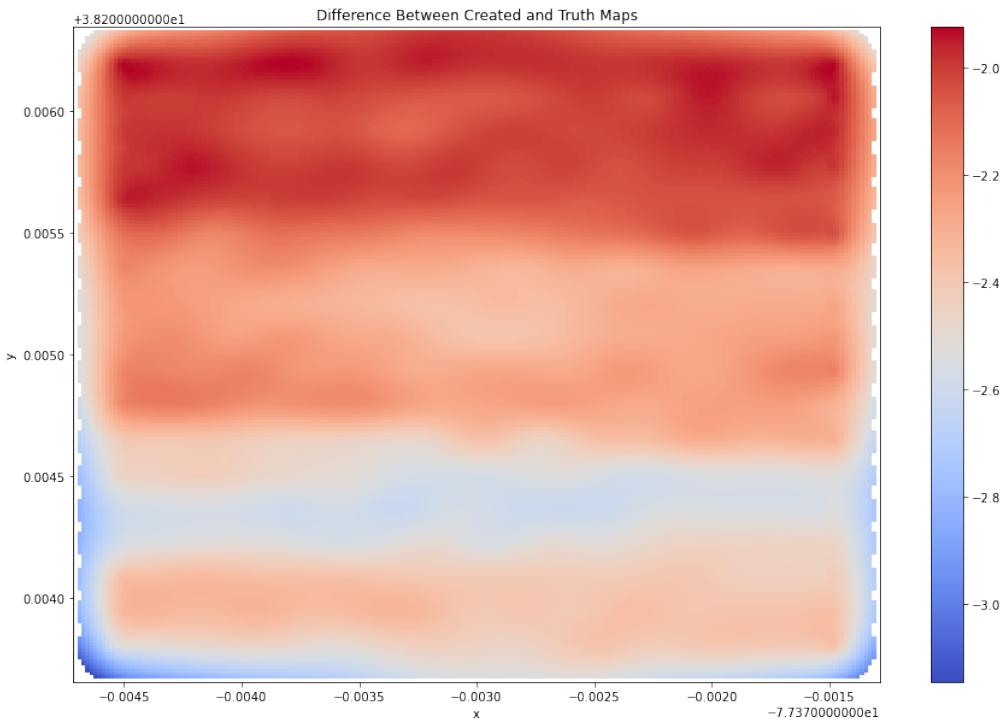
```

Loaded bou20190911psec.sec
Loaded bou20190912psec.sec
Loaded frd20190911psec.sec
Loaded frd20190912psec.sec
Loaded frn20190911psec.sec
Loaded frn20190912psec.sec
Optimal scale: 0.1354037206490241
Optimal offset: 159.832254129058
100%|      | 39204/39204 [00:02<00:00, 16890.77it/s]
Generated scalar map RMSE: 2.286562816950017nT

```







## 8 Generate Map Using Calibrated Reference Station Data from Fresno (FRN) - 4400km from FRD

```
[ ]: frn_t, frn_f, _, _, _ = Diurnal.longitude_norm(frn_ref_df,
                                                    truth_ref_df.LONG.mean())
frn_f_no_core = frn_f - np.array(frn_ref_df.IGRF_F)[0]

interp_combined = interpolate.interp1d(frn_t, frn_f_no_core, 'cubic')

survey_t      = np.array(survey_df.epoch_sec)
interp_mask = np.logical_and(np.logical_and(truth_t >= bou_t.min(),
                                            truth_t <= bou_t.max()),
                            np.logical_and(truth_t >= (survey_t[0] - 86400 - 2*3600),
                                          truth_t <= (survey_t[-1] - 86400 + 2*3600))) # Interpolate based on data from around the time of the survey on the previous day
interp_t      = truth_t(interp_mask) # Clip interpolation times

far_interp = interp_combined(interp_t)
```

```

offset, scale = Diurnal.calibrate([0, 1], far_interp, ↴
    ↪truth_f_no_core[interp_mask])
far_opt_f      = Diurnal.apply_cal([offset, scale], frn_f_no_core)

print('Optimal scale:', scale)
print('Optimal offset:', offset)

survey.map_loc_name = 'reconstructed_frn_ref'
survey.ref_scale    = scale
survey.ref_offset   = offset

map = survey.gen_map(ref_df=frn_ref_df)

interp_map = map[mu.SCALAR].interp(x=truth_map.x, y=truth_map.y)

plt.figure()
interp_map.plot(cmap=cm.coolwarm)
plt.title('Created Map')

plt.figure()
truth_map[0].plot(cmap=cm.coolwarm)
plt.title('Truth Map')

plt.figure()
error_map = truth_map[mu.SCALAR] - interp_map
error_map.plot(cmap=cm.coolwarm)
plt.title('Difference Between Created and Truth Maps')

map_rmse = pu.rmse(interp_map.data, truth_map[mu.SCALAR].data)
print('Generated scalar map RMSE: {}nT'.format(map_rmse))

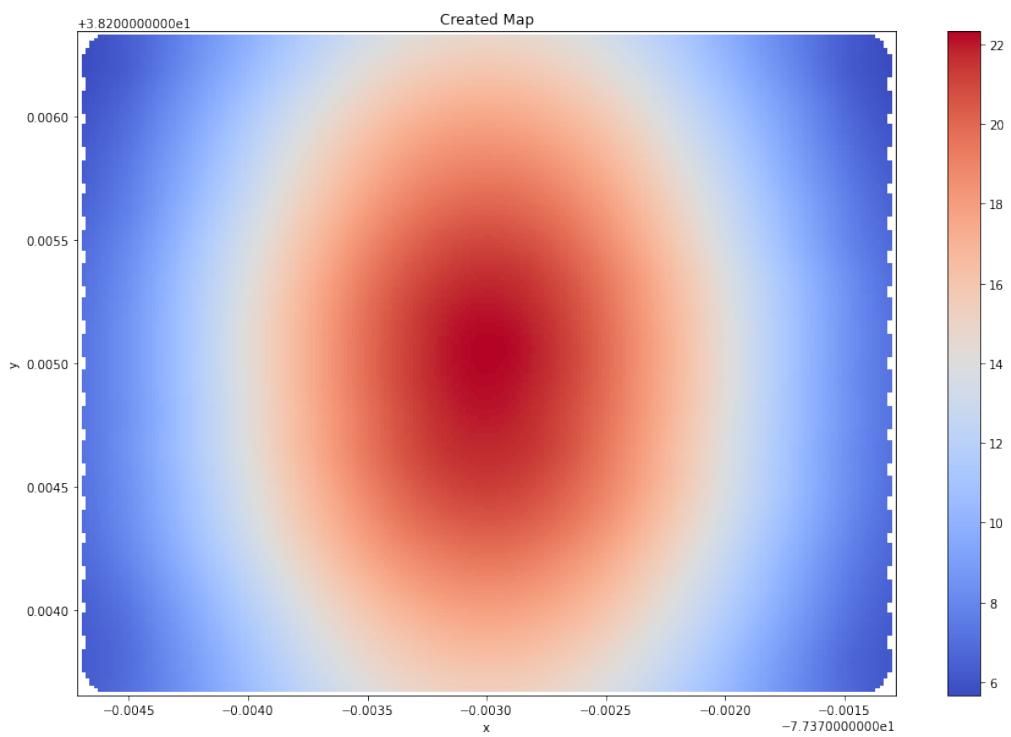
# Clean up
survey.ref_scale = 1
survey.ref_offset = 0

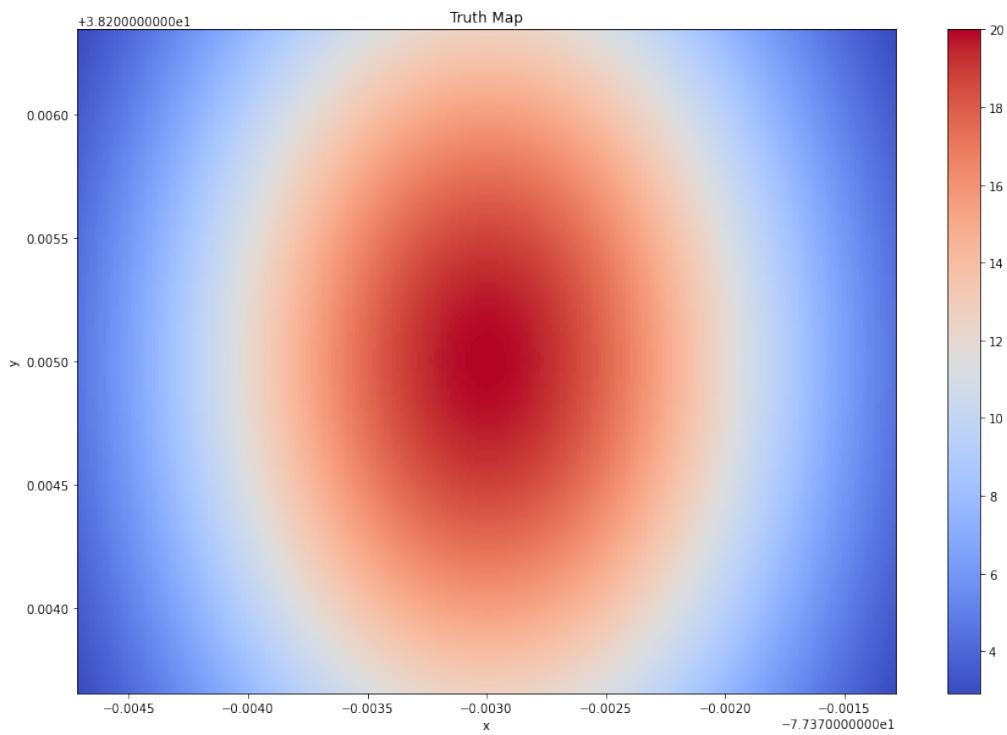
```

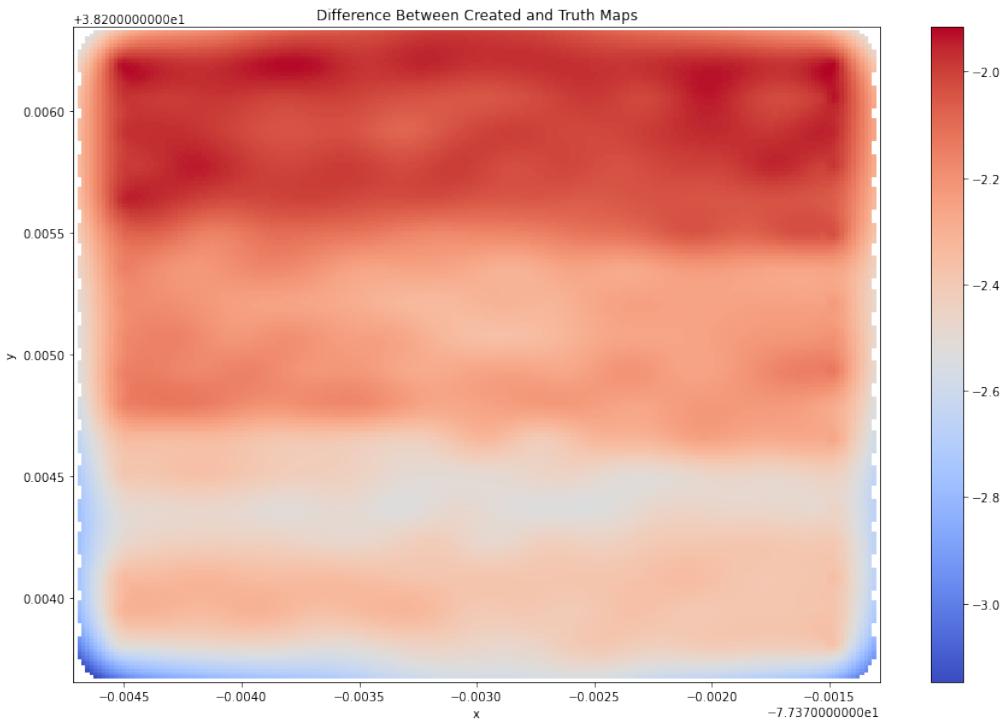
```

Optimal scale: 0.14690255812962025
Optimal offset: 173.34683437172754
100%| 39204/39204 [00:02<00:00, 16971.40it/s]
Generated scalar map RMSE: 2.2613877196742886nT

```







## 9 Generate Map Using Biased Scalar Measurements

```
[ ]: biases      = [0.1, 1, 10]
survey_df = deepcopy(survey.survey_df)

for bias in biases:
    biased_survey_df = deepcopy(survey_df)
    biased_survey_df.F += bias

    survey.map_loc_name = 'reconstructed_{}nT_bias'.format(bias)

    map = survey.gen_map(survey_df=biased_survey_df)

    interp_map = map[mu.SCALAR].interp(x=truth_map.x, y=truth_map.y)

    plt.figure()
    interp_map.plot(cmap=cm.coolwarm)
    plt.title('Created Map with Survey Bias of {}nT'.format(bias))

    plt.figure()
    truth_map[0].plot(cmap=cm.coolwarm)
```

```

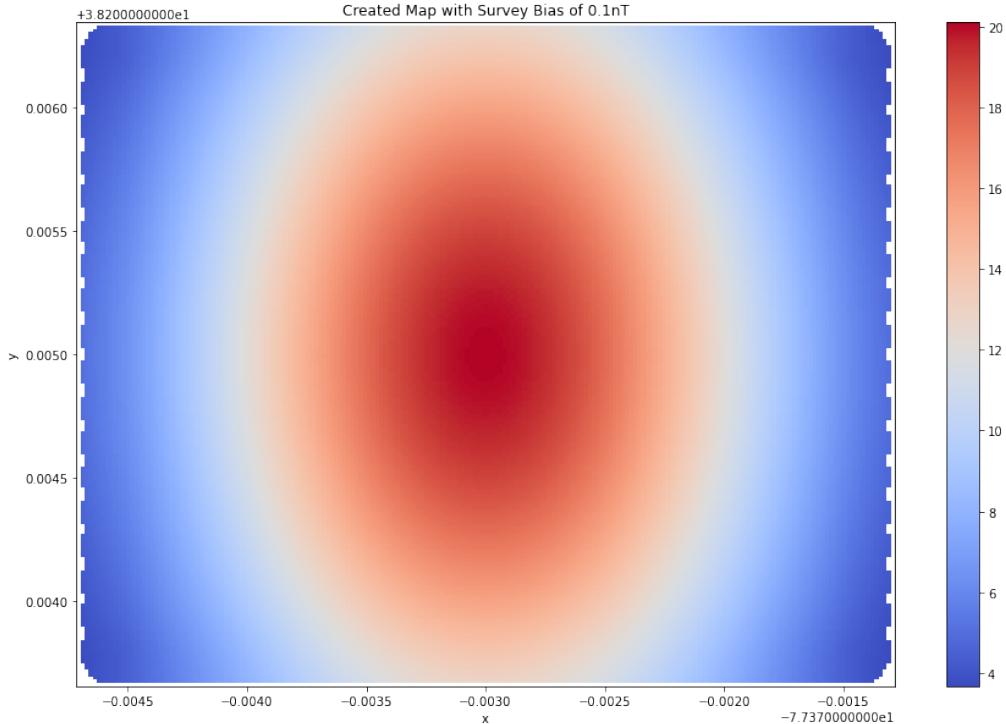
plt.title('Truth Map')

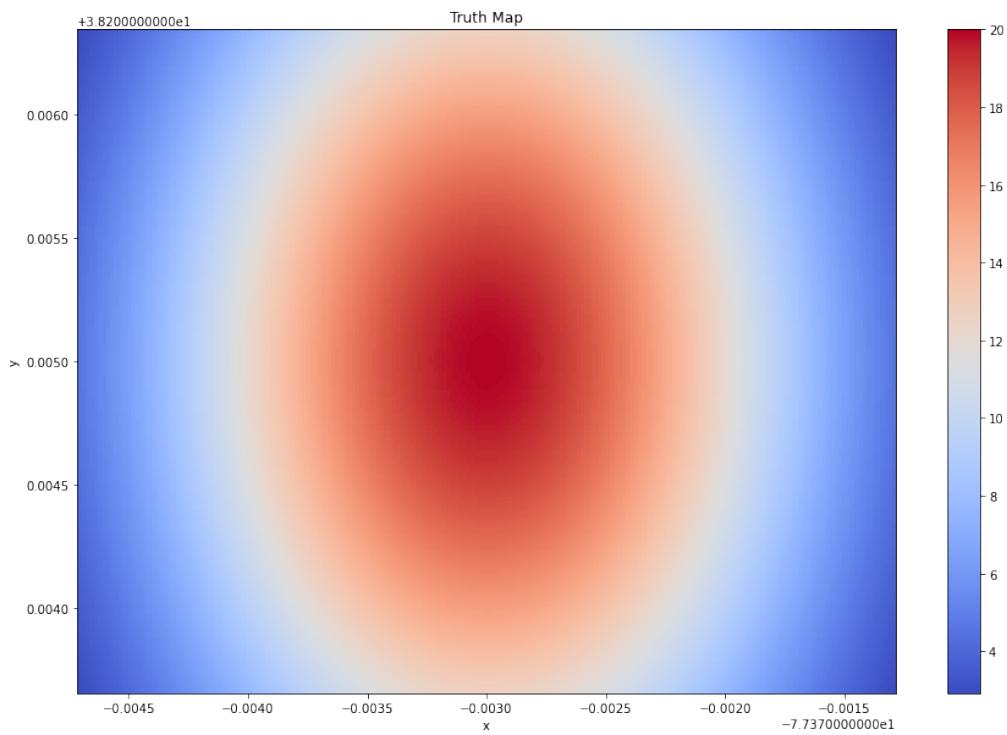
plt.figure()
error_map = truth_map[mu.SCALAR] - interp_map
error_map.plot(cmap=cm.coolwarm)
plt.title('Difference Between Created and Truth\nMaps with Survey Bias of' + 
        '\u2122{}nT'.format(bias))

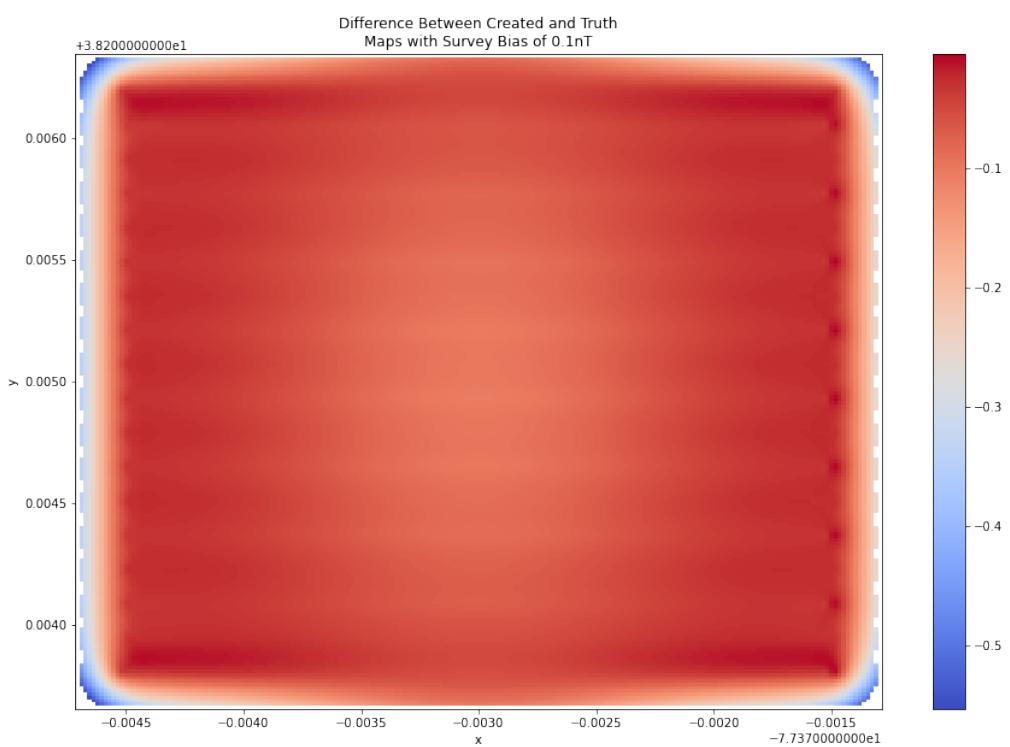
map_rmse = pu.rmse(interp_map.data, truth_map[mu.SCALAR].data)
print('Generated scalar map RMSE with survey bias of {}nT: {}nT'.
      format(bias, map_rmse))

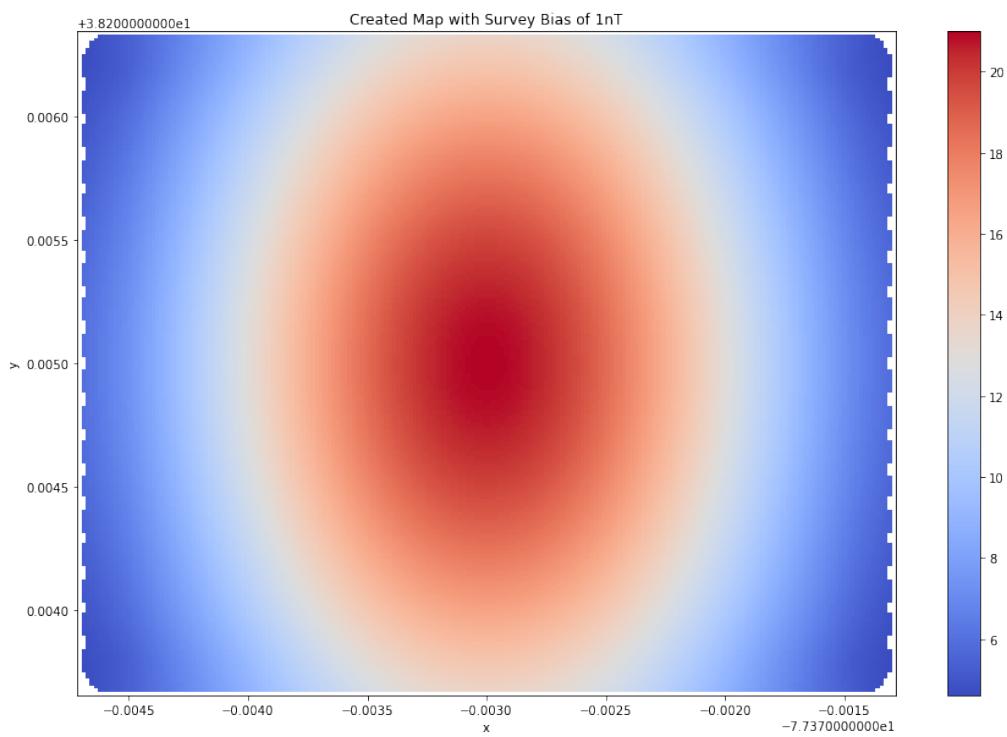
```

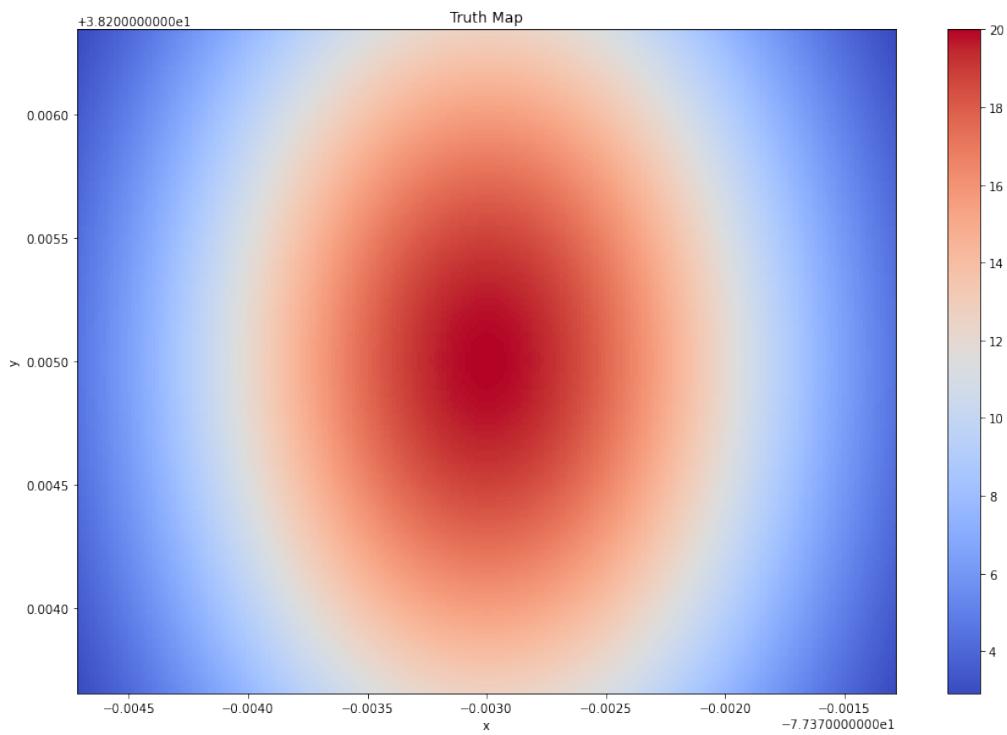
100% | 39204/39204 [00:02<00:00, 16561.72it/s]  
Generated scalar map RMSE with survey bias of 0.1nT: 0.09149344671020931nT  
100% | 39204/39204 [00:02<00:00, 16775.36it/s]  
Generated scalar map RMSE with survey bias of 1nT: 0.9727882906087355nT  
100% | 39204/39204 [00:02<00:00, 15236.69it/s]  
Generated scalar map RMSE with survey bias of 10nT: 9.983606669552199nT

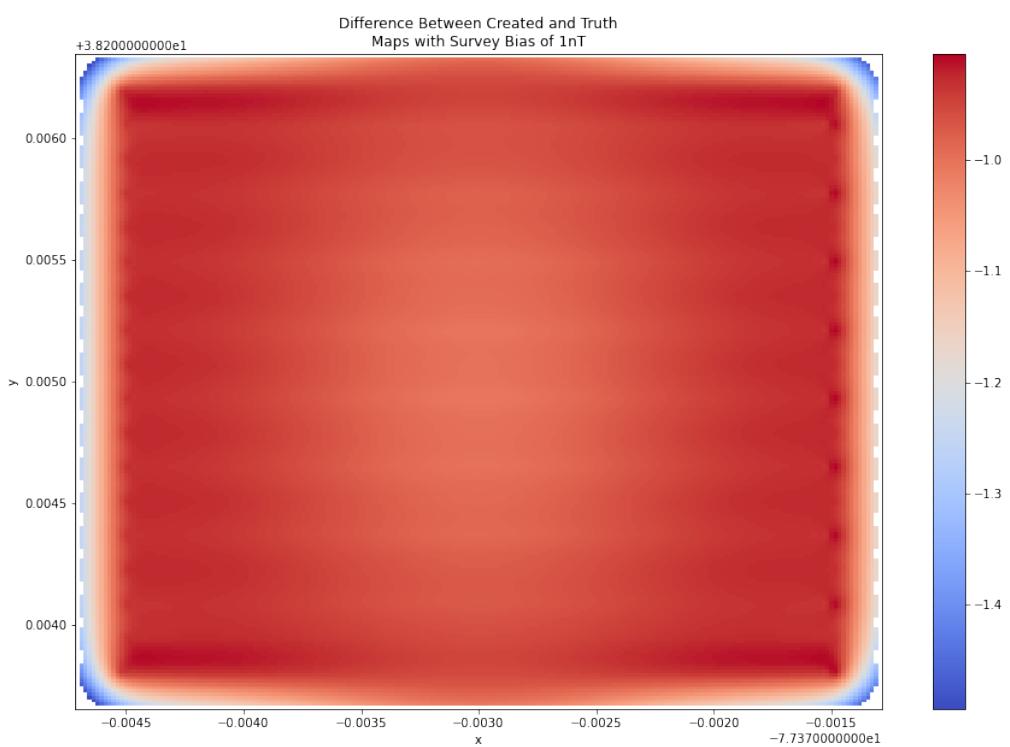


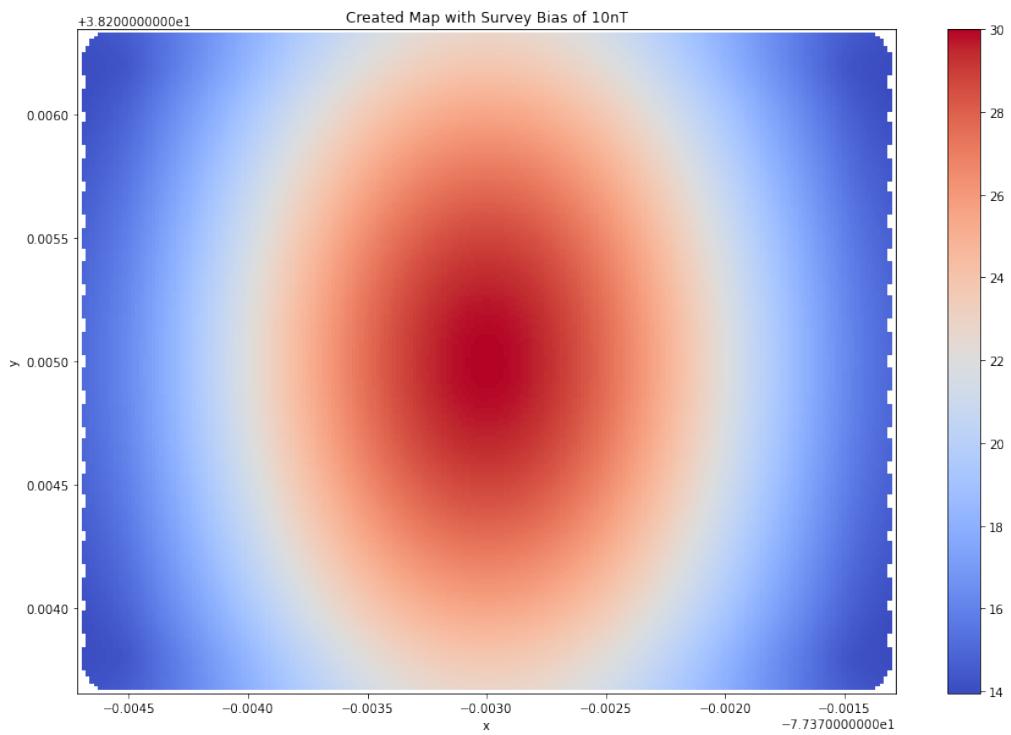


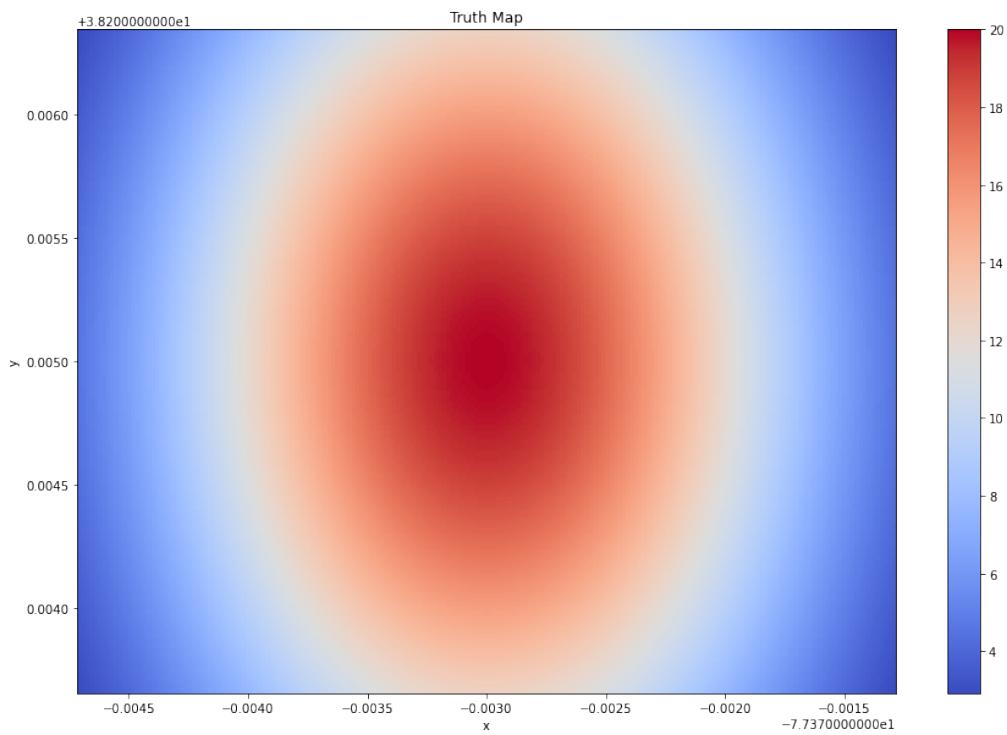


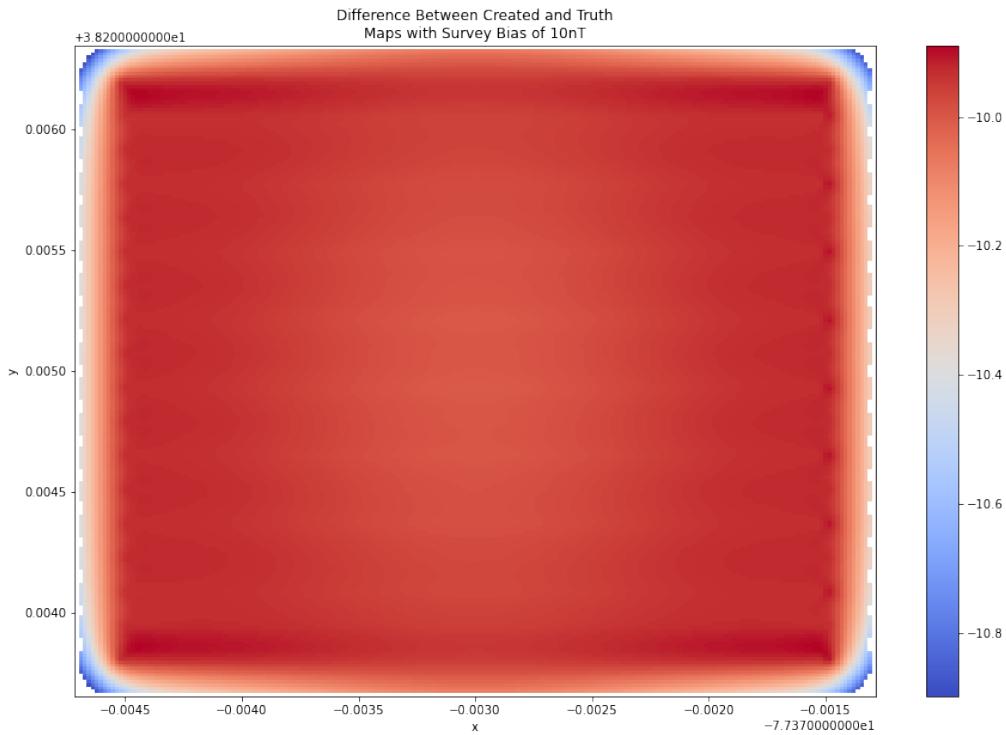












## 10 Generate Map Using Scalar Measurements with Additive White Gaussian Noise (AWGN)

```
[ ]: awgn_stds = [1, 10, 100]

for awgn_std in awgn_stds:
    noised_survey_df = deepcopy(survey_df)
    noised_survey_df.F += np.random.randn(len(noised_survey_df.F)) * awgn_std

    map = survey.gen_map(survey_df=noised_survey_df)

    interp_map = map[mu.SCALAR].interp(x=truth_map.x, y=truth_map.y)

    plt.figure()
    interp_map.plot(cmap=cm.coolwarm)
    plt.title('Created Map with Survey AWGN (STD: {}nT)'.format(awgn_std))

    plt.figure()
    truth_map[0].plot(cmap=cm.coolwarm)
    plt.title('Truth Map')
```

```

plt.figure()
error_map = truth_map[mu.SCALAR] - interp_map
error_map.plot(cmap=cm.coolwarm)
plt.title('Difference Between Created and Truth\nMaps with Survey AWGN (STD: {}nT)'.format(awgn_std))

map_rmse = pu.rmse(interp_map.data, truth_map[mu.SCALAR].data)
print('Generated scalar map RMSE with survey AWGN (STD: {}nT): {}nT'.format(awgn_std, map_rmse))

```

100% | 39204/39204 [00:02<00:00, 16647.15it/s]

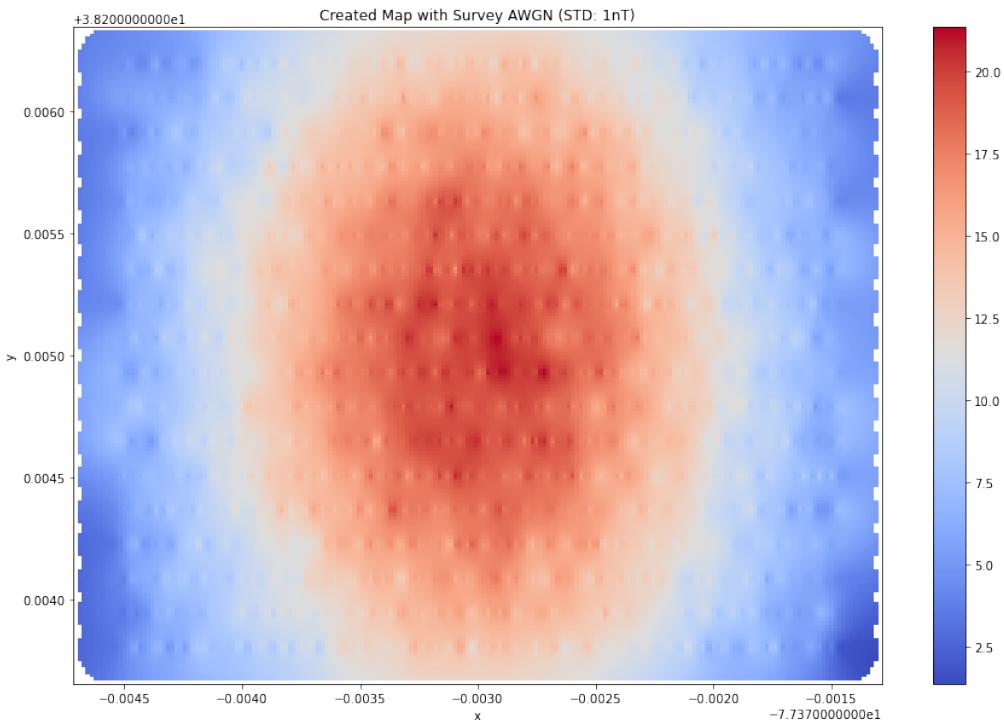
Generated scalar map RMSE with survey AWGN (STD: 1nT): 0.5214322025204605nT

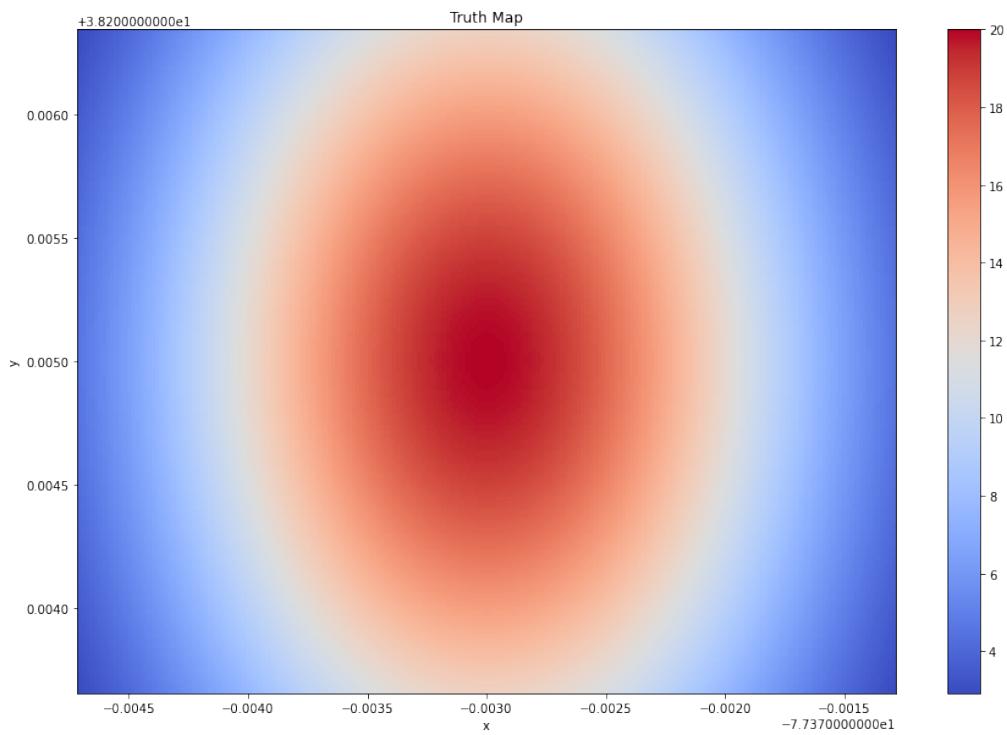
100% | 39204/39204 [00:02<00:00, 16625.77it/s]

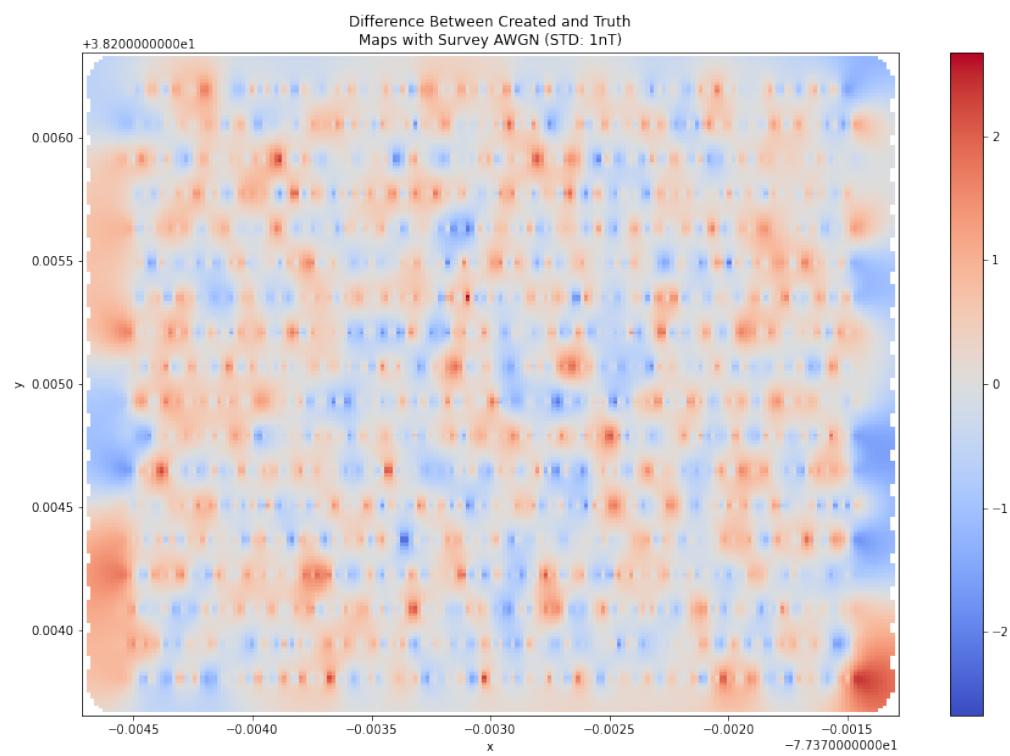
Generated scalar map RMSE with survey AWGN (STD: 10nT): 4.661728818571068nT

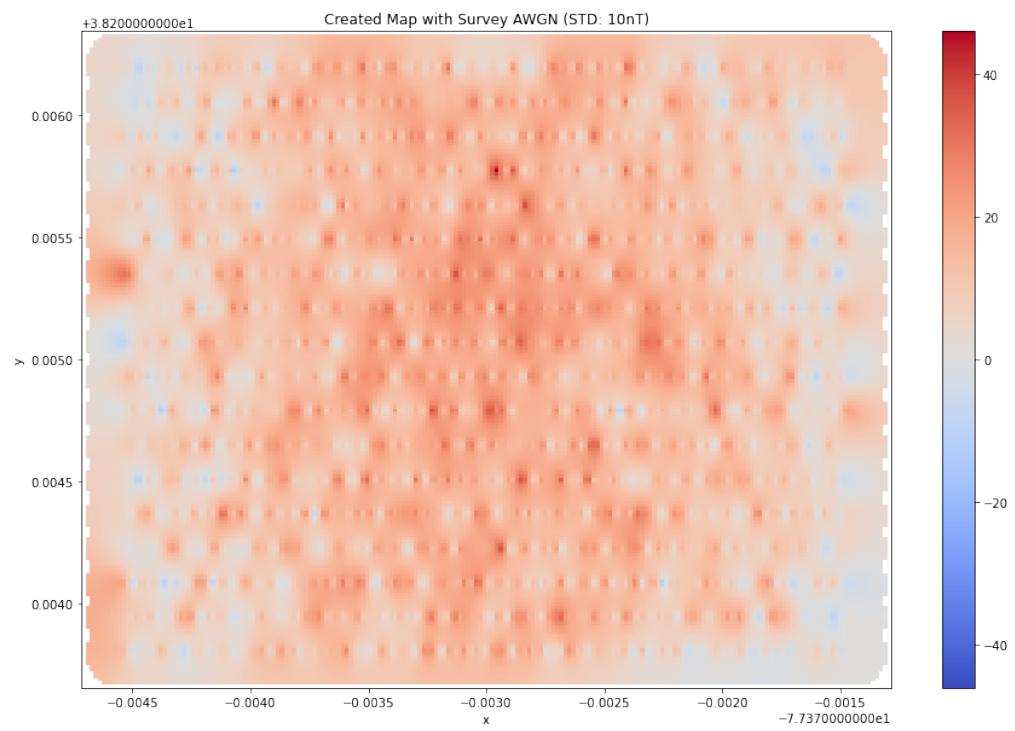
100% | 39204/39204 [00:02<00:00, 16668.05it/s]

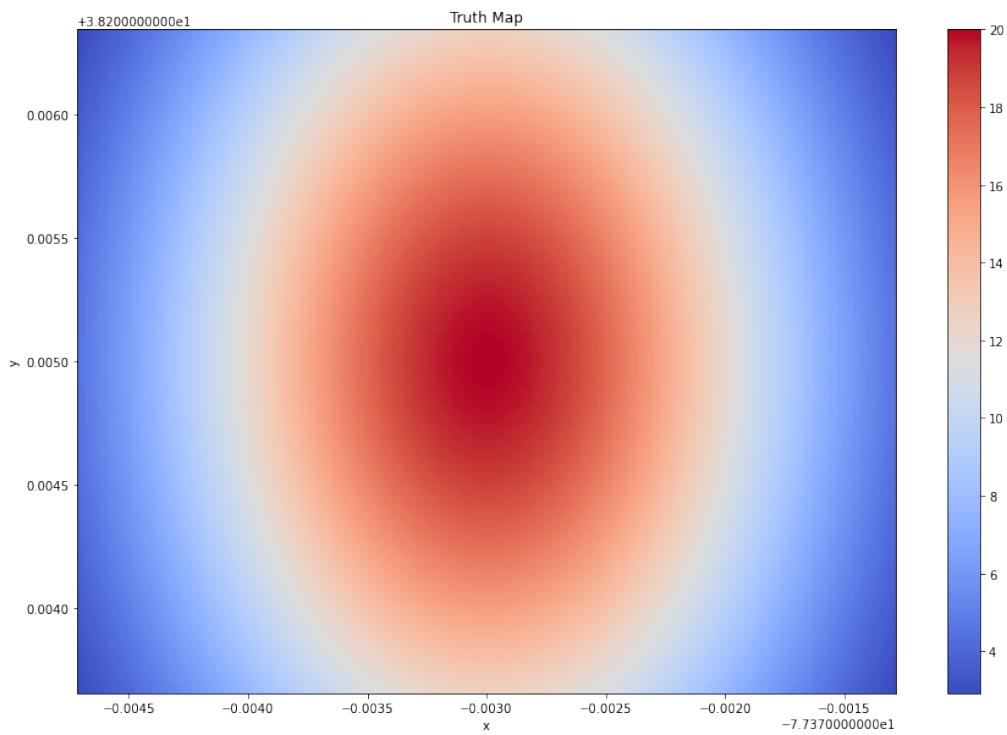
Generated scalar map RMSE with survey AWGN (STD: 100nT): 48.39170483060359nT

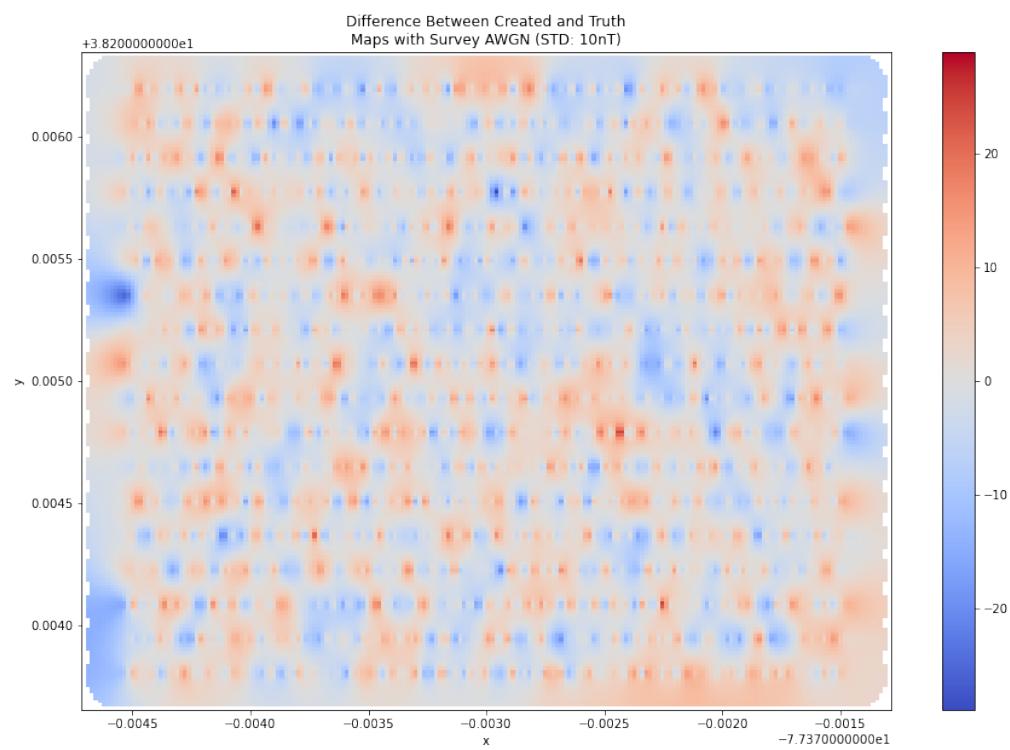


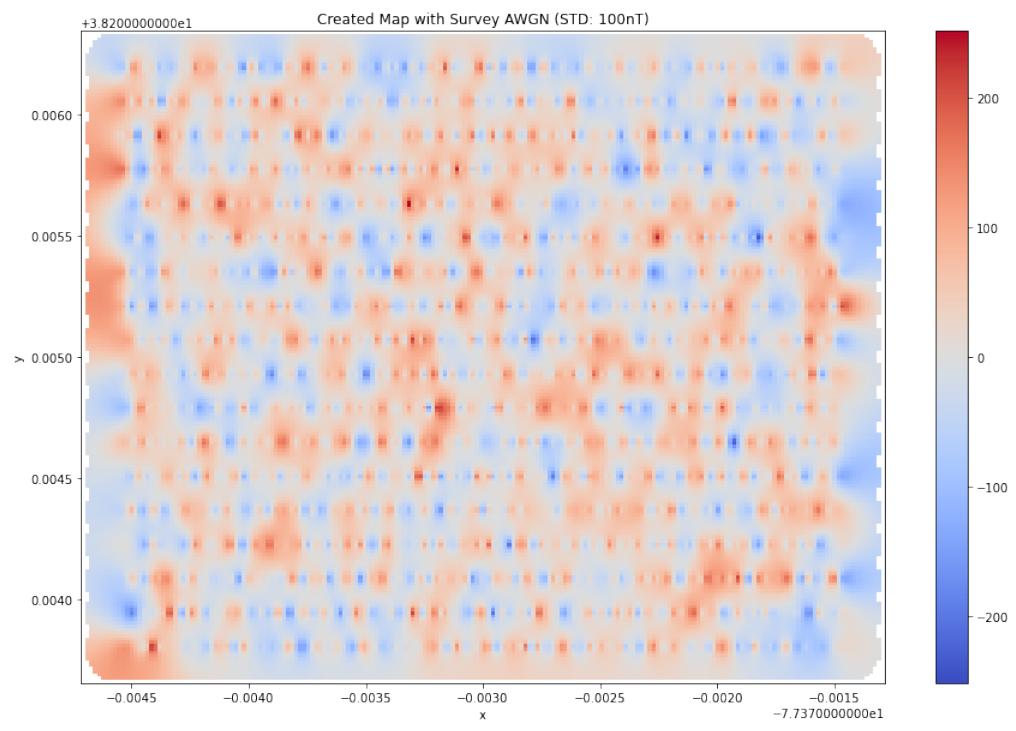


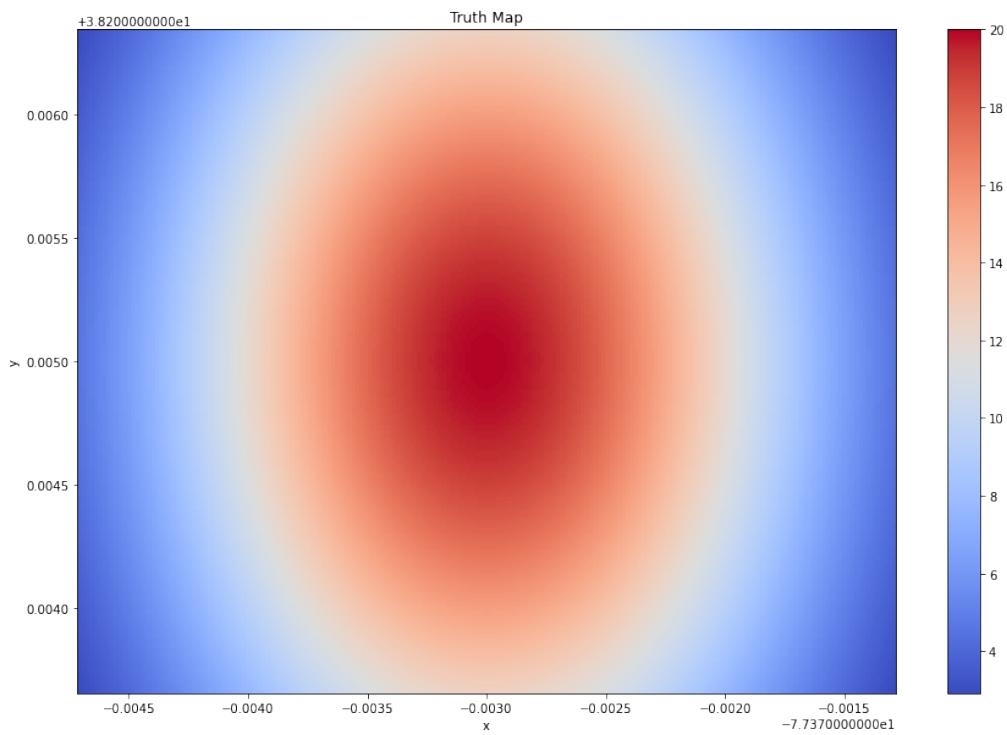


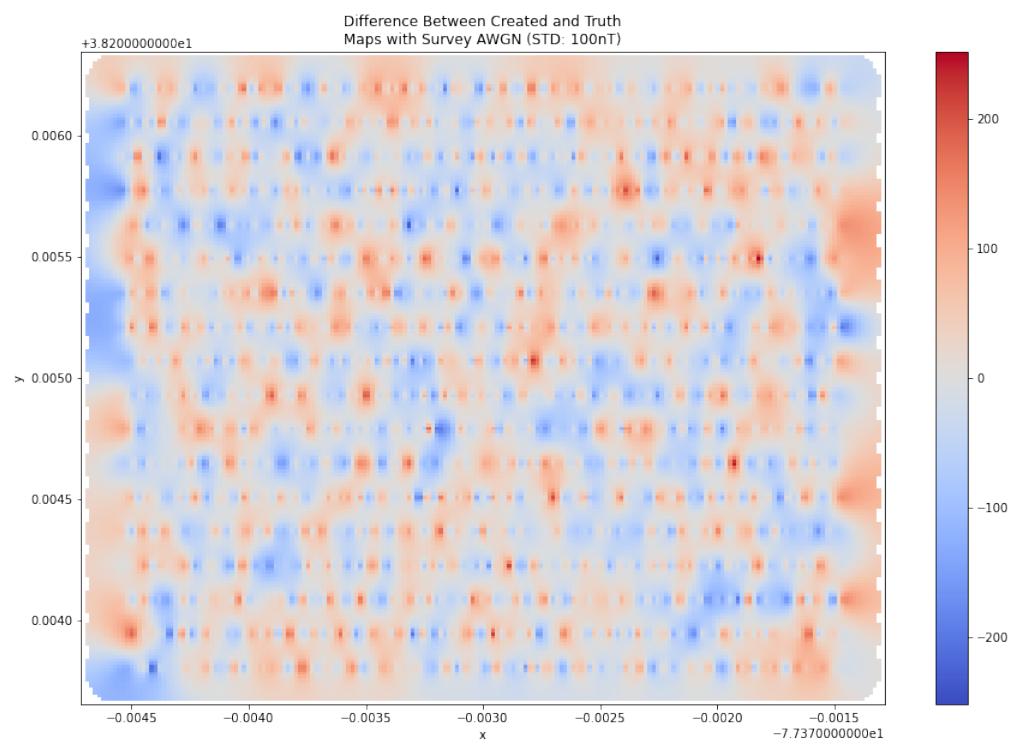












## \_\_\_\_\_process\_ground\_cal\_\_\_\_\_

December 20, 2022

```
[ ]: import sys
import datetime as dt
from os.path import join

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import scipy.linalg as la

PROJ_DIR = r'C:\Users\ltber\Desktop\files\AFIT\Research\WPAFB_Survey\mammal'
sys.path.append(PROJ_DIR)

from MAMMAL.Utils import ProcessingUtils as pu
from MAMMAL.Utils import Filters as filt
from MAMMAL.VehicleCal import magUtilsTL as magtl

%matplotlib inline
plt.rcParams["figure.figsize"] = (30, 30) # (w, h)
plt.style.use(['seaborn-poster', 'seaborn-white'])
pd.set_option('mode.chained_assignment', None)

SURVEY_DIR = r'D:\__atterbury_data__\ground_cal_'

LOG_CSV_FNAME = '__ground_cal__.csv'
CAL_NAME      = 'MFAM Ground Tolles Lawson'

START_DT = dt.datetime(2022, 11, 8, 20, 16, 30)
END_DT   = dt.datetime(2022, 11, 8, 20, 21, 0)

def precal_plots(timestamps, b_scalar, mfam_vec_dcs, vmr_dcs):
    _, (ax0, ax1, ax2, ax3, ax4, ax5, ax6) = plt.subplots(7, 1, sharex=True)

    ax0.plot(timestamps, b_scalar)
    ax0.set_title('Sensor Measurements (Inside UAV)')
```

```

ax0.set_ylabel('nT')
ax0.set_xlabel('Timestamp')
ax0.grid()

ax1.plot(timestamps, mfam_vec_dcs[:, 0])
ax1.set_title('MFAM Direction Cosines X-Component (Inside UAV)')
ax1.set_xlabel('Timestamp')
ax1.grid()

ax2.plot(timestamps, mfam_vec_dcs[:, 1])
ax2.set_title('MFAM Direction Cosines Y-Component (Inside UAV)')
ax2.set_xlabel('Timestamp')
ax2.grid()

ax3.plot(timestamps, mfam_vec_dcs[:, 2])
ax3.set_title('MFAM Direction Cosines Z-Component (Inside UAV)')
ax3.set_xlabel('Timestamp')
ax3.grid()

ax4.plot(timestamps, vmr_dcs[:, 0])
ax4.set_title('VMR Direction Cosines X-Component (Inside UAV)')
ax4.set_xlabel('Timestamp')
ax4.grid()

ax5.plot(timestamps, vmr_dcs[:, 1])
ax5.set_title('VMR Direction Cosines Y-Component (Inside UAV)')
ax5.set_xlabel('Timestamp')
ax5.grid()

ax6.plot(timestamps, vmr_dcs[:, 2])
ax6.set_title('VMR Direction Cosines Z-Component (Inside UAV)')
ax6.set_xlabel('Timestamp')
ax6.grid()

def postcal_plots(scalar_ts, b_scalar, b_scalar_truth, cal_scalar, dcs):
    _, (ax0, ax1, ax2, ax3, ax4) = plt.subplots(5, 1, sharex=True)

    ax0.plot(scalar_ts, b_scalar)
    ax0.plot(scalar_ts, b_scalar_truth, label='Truth')
    ax0.set_title('Uncalibrated Scalar Measurements with Truth (Inside UAV)')
    ax0.set_ylabel('nT')
    ax0.set_xlabel('Timestamp')
    ax0.legend()
    ax0.grid()

    ax1.plot(scalar_ts, cal_scalar)
    ax1.plot(scalar_ts, b_scalar_truth, label='Truth')

```

```

ax1.set_title('Calibrated Scalar Measurements with Truth (Inside UAV)')
ax1.set_ylabel('nT')
ax1.set_xlabel('Timestamp')
ax1.legend()
ax1.grid()

ax2.plot(scalar_ts, dcs[:, 0])
ax2.set_title('Direction Cosines X-Component Measurements (Inside UAV)')
ax2.set_xlabel('Timestamp')
ax2.grid()

ax3.plot(scalar_ts, dcs[:, 1])
ax3.set_title('Direction Cosines Y-Component Measurements (Inside UAV)')
ax3.set_xlabel('Timestamp')
ax3.grid()

ax4.plot(scalar_ts, dcs[:, 2])
ax4.set_title('Direction Cosines Z-Component Measurements (Inside UAV)')
ax4.set_xlabel('Timestamp')
ax4.grid()

```

## 1 Load Calibration Data

```
[ ]: log_df = pd.read_csv(join(SURVEY_DIR, LOG_CSV_FNAME), parse_dates=['datetime'])
log_df = log_df[(log_df.datetime > START_DT) & (log_df.datetime < END_DT)]

datetimes = log_df.datetime
timestamps = np.array(log_df.epoch_sec)
```

## 2 Compare Each Head Vs Averaged Values

```
[ ]: plt.rcParams["figure.figsize"] = (20, 10) # (w, h)

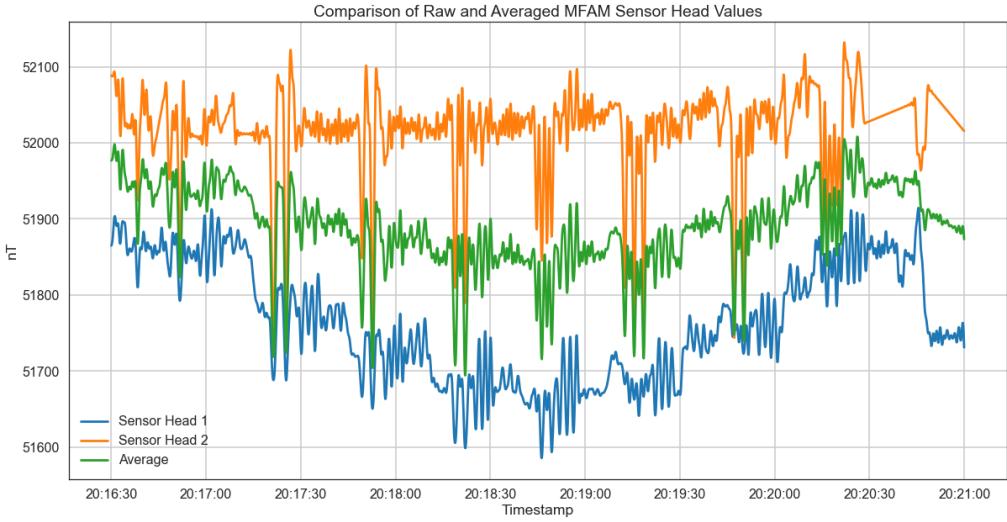
plt.figure()
plt.title('Comparison of Raw and Averaged MFAM Sensor Head Values')
plt.plot(log_df.datetime, filt.lpf(log_df.SCALAR_1_LPF, 1, 10, 10), label='Sensor Head 1')
plt.plot(log_df.datetime, filt.lpf(log_df.SCALAR_2_LPF, 1, 10, 10), label='Sensor Head 2')
plt.plot(log_df.datetime, (filt.lpf(log_df.SCALAR_1_LPF, 1, 10, 10) + filt.lpf(log_df.SCALAR_2_LPF, 1, 10, 10)) / 2.0, label='Average')
plt.xlabel('Timestamp')
plt.ylabel('nT')
plt.grid()
plt.legend()
```

```

plt.show()

plt.rcParams["figure.figsize"] = (30, 30) # (w, h)

```



### 3 Calibrate MFAM Sensor in UAV

#### 4 Prune/Preprocess Data

```

[ ]: b_scalar = (log_df.SCALAR_1_LPF + log_df.SCALAR_2_LPF) / 2.0

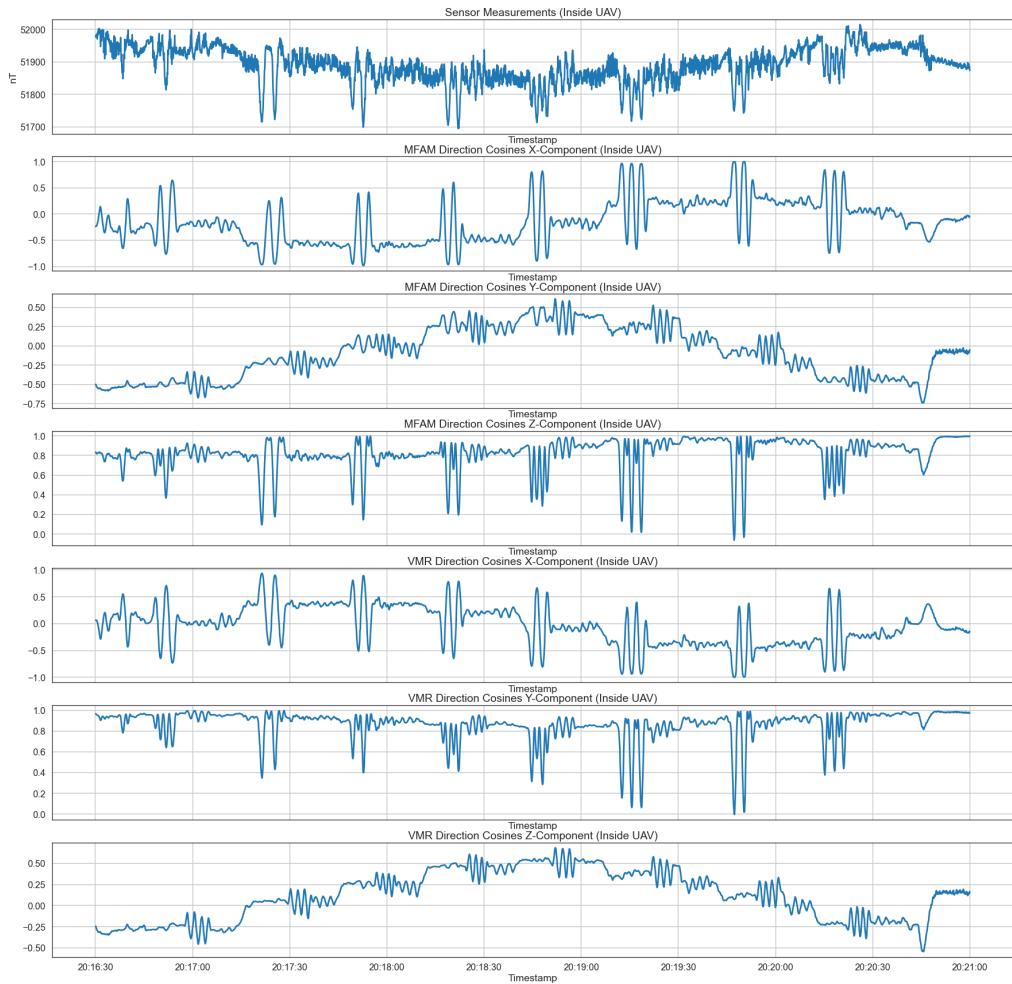
vmr = np.hstack([np.array(log_df.X)[:, np.newaxis],
                 np.array(log_df.Y)[:, np.newaxis],
                 np.array(log_df.Z)[:, np.newaxis]])
vmr_magnitude = la.norm(vmr, axis=1)
vmr_dcs = vmr / vmr_magnitude[:, np.newaxis]

mfam_vec = np.hstack([np.array(log_df.X_MFAM)[:, np.newaxis],
                      np.array(log_df.Y_MFAM)[:, np.newaxis],
                      np.array(log_df.Z_MFAM)[:, np.newaxis]])
mfam_vec_magnitude = la.norm(mfam_vec, axis=1)
mfam_vec_dcs = mfam_vec / mfam_vec_magnitude[:, np.newaxis]

```

## 5 Pre-Calibration Plots

```
[ ]: precal_plots(datetimes,
                  b_scalar,
                  mfam_vec_dcs,
                  vmr_dcs)
```



## 6 Find/Apply Tolles Lawson (TL) Calibration Coefficients (All TL Terms + MFAM Vector Sensor)

```
[ ]: tl_terms = magtl.DEFAULT_TL_TERMS

b_vector = mfam_vec

scalar_ts = timestamps
delta_t    = np.diff(scalar_ts).mean()

filt = magtl.Filter(order = 10,
                     fcut   = [0.1, 2.0],
                     btype  = 'band',
                     ftype  = 'butter',
                     fs     = (1.0 / delta_t))

scalar_terms = magtl.tolles_lawson_coefficients(vector      = b_vector,
                                                 y_value     = b_scalar,
                                                 time_delta = delta_t,
                                                 apply_filter = True,
                                                 mag_filter  = filt,
                                                 terms       = tl_terms)

print('TL Terms:', scalar_terms)

body_effects = magtl.tlc_compensation(vector = b_vector,
                                         tlc     = scalar_terms,
                                         terms   = tl_terms)

cal_scalar = b_scalar - body_effects
cal_scalar -= (cal_scalar.mean() - b_scalar.mean()) # Bad fix for issue where TL adds a huge offset

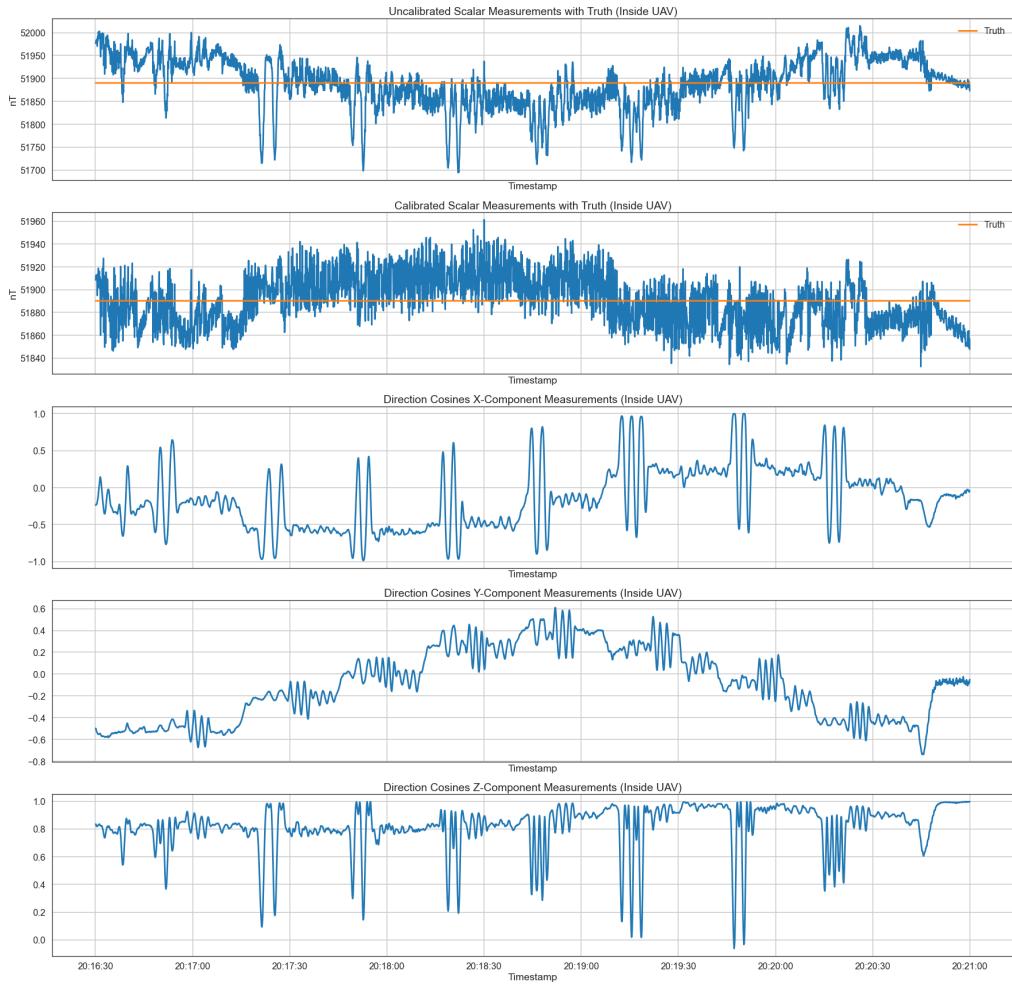
b_scalar_truth = np.ones(b_scalar.shape) * b_scalar.mean()
b_scalar_error = np.abs(cal_scalar - b_scalar_truth)

print('RMSE:', pu.rmse(cal_scalar, b_scalar_truth))
```

TL Terms: [ 5.70657355e+01 -1.12763248e+02 2.21579936e+02 -1.29875018e+04  
-4.99187899e+02 -7.12973512e+04 1.13955716e+05 -9.57637214e+04  
-4.50833457e+04 1.83142845e+04 -8.99594518e+03 2.92805550e+03  
 3.61167791e+03 5.45811696e+04 -3.29564152e+03 4.12762648e+03  
-8.64986919e+03 2.67089442e+04]  
RMSE: 22.525991430215043

## 7 Post-Calibration Plots (All TL Terms + MFAM Vector Sensor)

```
[ ]: postcal_plots(datetimes,
                  b_scalar,
                  b_scalar_truth,
                  cal_scalar,
                  mfam_vec_dcs)
```



## 8 Find/Apply TL Calibration Coefficients (Permanent and Induced TL Terms + MFAM Vector Sensor)

```
[ ]: tl_terms = [magtl.TollesLawsonTerms.PERMANENT,
                magtl.TollesLawsonTerms.INDUCED]

scalar_terms = magtl.tolles_lawson_coefficients(vector      = b_vector,
                                                y_value     = b_scalar,
                                                time_delta = delta_t,
                                                apply_filter = True,
                                                mag_filter   = filt,
                                                terms        = tl_terms)

print('TL Terms:', scalar_terms)

body_effects = magtl.tlc_compensation(vector = b_vector,
                                         tlc    = scalar_terms,
                                         terms  = tl_terms)

cal_scalar = b_scalar - body_effects
cal_scalar -= (cal_scalar.mean() - b_scalar.mean()) # Bad fix for issue where TL adds a huge offset

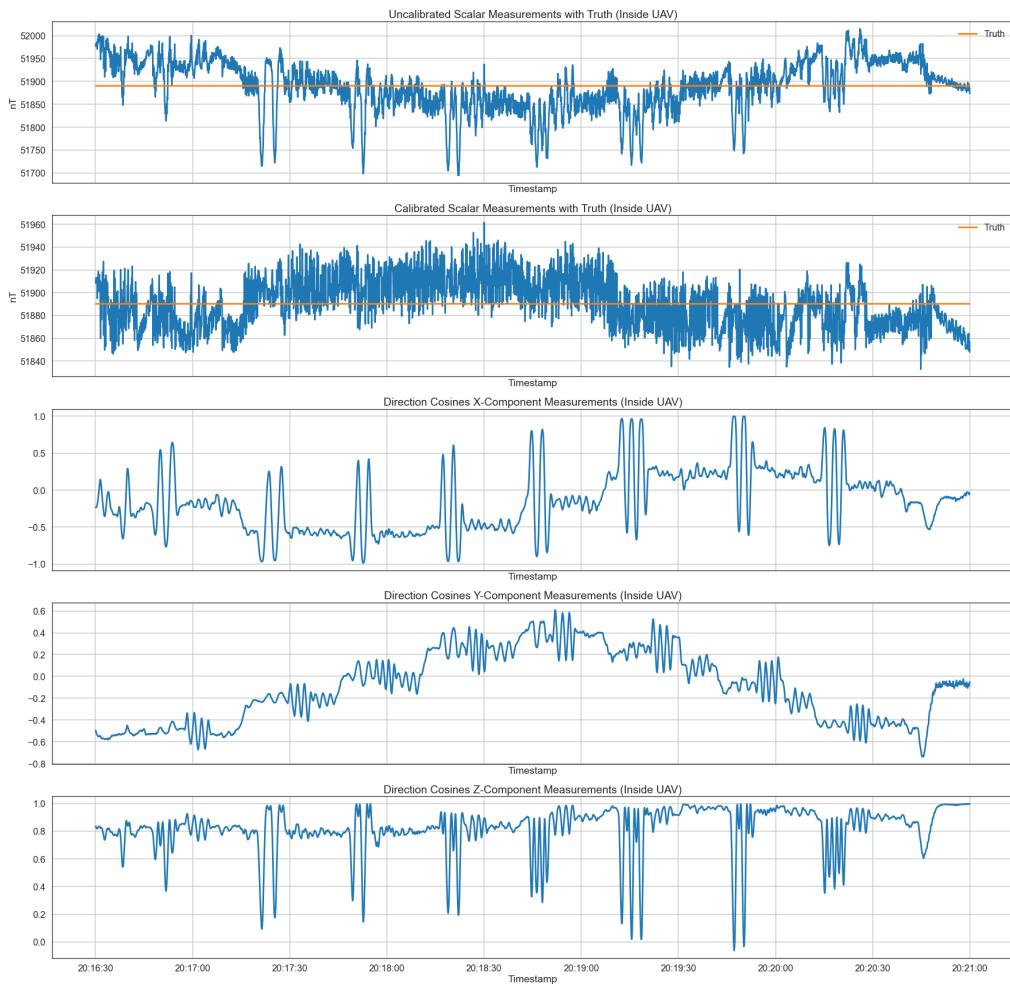
b_scalar_truth = np.ones(b_scalar.shape) * b_scalar.mean()
b_scalar_error = np.abs(cal_scalar - b_scalar_truth)

print('RMSE:', pu.rmse(cal_scalar, b_scalar_truth))
```

TL Terms: [ 5.26214900e+01 -1.21125228e+02 2.15786105e+02 -9.95712052e+04  
 1.61563819e+03 -6.94848124e+04 2.98398549e+04 -9.33704452e+04  
 -1.31525435e+05]  
RMSE: 22.539035708860144

## 9 Post-Calibration Plots (Permanent and Induced TL Terms + MFAM Vector Sensor)

```
[ ]: postcal_plots(datetimes,
                  b_scalar,
                  b_scalar_truth,
                  cal_scalar,
                  mfam_vec_dcs)
```



## 10 Find/Apply TL Calibration Coefficients (Permanent TL Terms + MFAM Vector Sensor)

```
[ ]: tl_terms = [magtl.TollesLawsonTerms.PERMANENT]

scalar_terms = magtl.tolles_lawson_coefficients(vector      = b_vector,
                                                y_value     = b_scalar,
                                                time_delta = delta_t,
                                                apply_filter = True,
                                                mag_filter  = filt,
                                                terms       = tl_terms)
```

```

print('TL Terms:', scalar_terms)

body_effects = magtl.tlc_compensation(vector = b_vector,
                                       tlc      = scalar_terms,
                                       terms   = tl_terms)

cal_scalar = b_scalar - body_effects
cal_scalar -= (cal_scalar.mean() - b_scalar.mean()) # Bad fix for issue where TL adds a huge offset

b_scalar_truth = np.ones(b_scalar.shape) * b_scalar.mean()
b_scalar_error = np.abs(cal_scalar - b_scalar_truth)

print('RMSE:', pu.rmse(cal_scalar, b_scalar_truth))

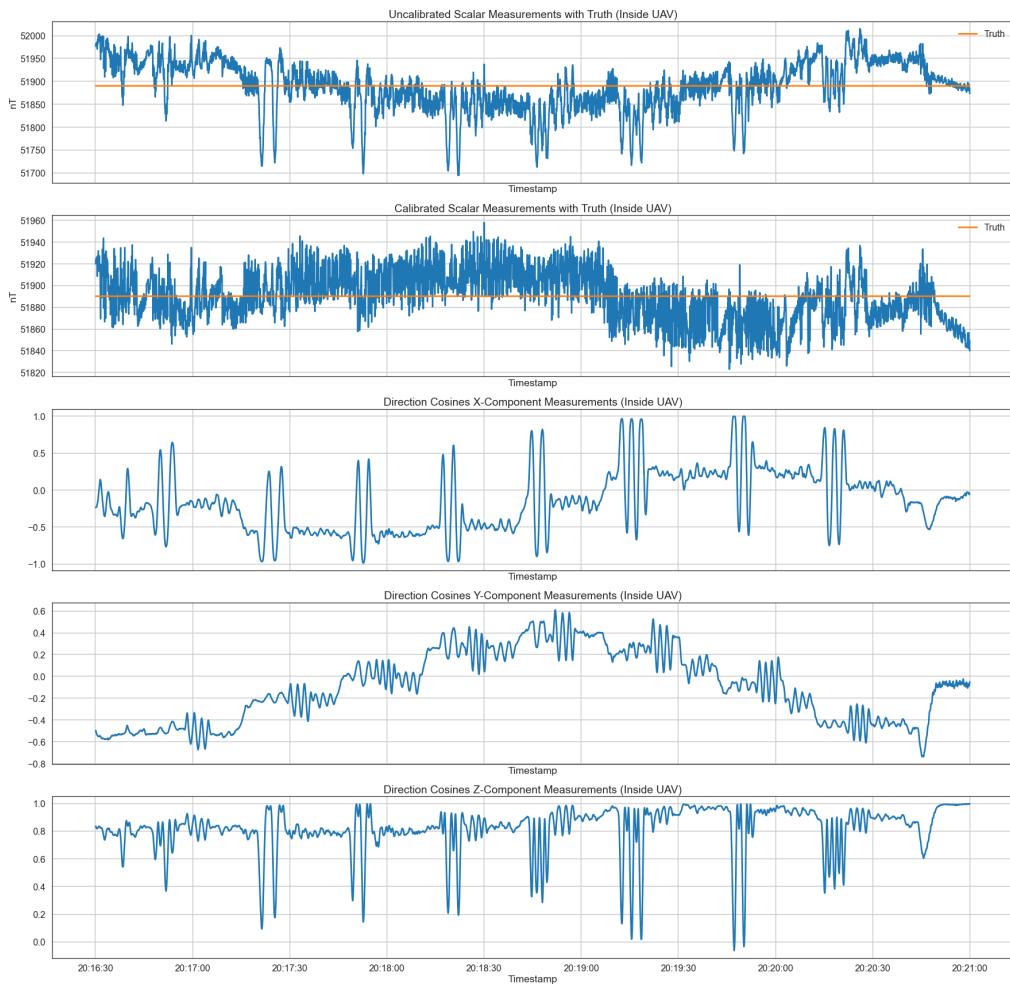
```

TL Terms: [ 43.26481885 -143.49388132 200.73471691]

RMSE: 23.436190685700087

## 11 Post-Calibration Plots (Permanent TL Terms + MFAM Vector Sensor)

```
[ ]: postcal_plots(datetimes,
                  b_scalar,
                  b_scalar_truth,
                  cal_scalar,
                  mfam_vec_dcs)
```



## 12 Find/Apply TL Calibration Coefficients (Induced TL Terms + MFAM Vector Sensor)

```
[ ]: tl_terms = [magtl.TollesLawsonTerms.INDUCED]

scalar_terms = magtl.tolles_lawson_coefficients(vector      = b_vector,
                                                y_value     = b_scalar,
                                                time_delta = delta_t,
                                                apply_filter = True,
                                                mag_filter  = filt,
                                                terms       = tl_terms)
```

```

print('TL Terms:', scalar_terms)

body_effects = magtl.tlc_compensation(vector = b_vector,
                                       tlc      = scalar_terms,
                                       terms   = tl_terms)

cal_scalar = b_scalar - body_effects
cal_scalar -= (cal_scalar.mean() - b_scalar.mean()) # Bad fix for issue where TL adds a huge offset

b_scalar_truth = np.ones(b_scalar.shape) * b_scalar.mean()
b_scalar_error = np.abs(cal_scalar - b_scalar_truth)

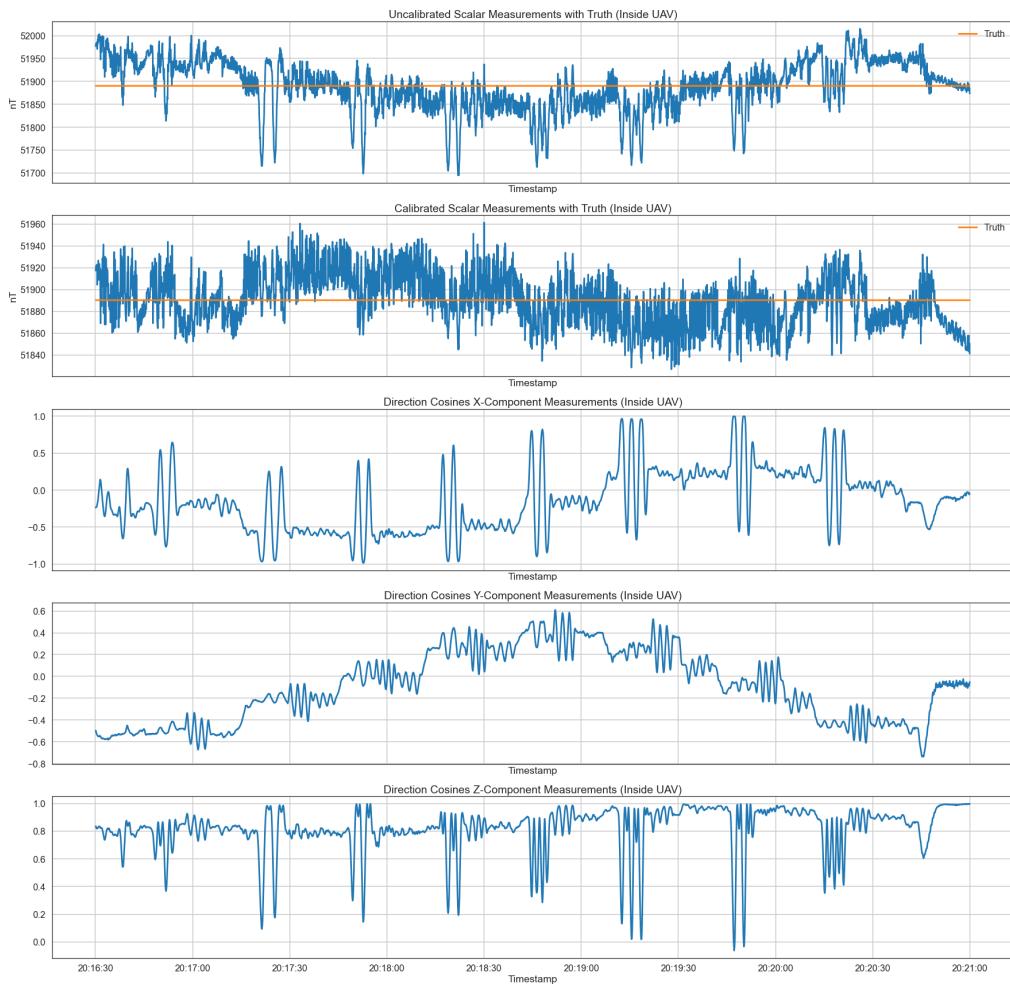
print('RMSE:', pu.rmse(cal_scalar, b_scalar_truth))

```

TL Terms: [-359251.03614993 -8172.52611237 72344.64328544 -126547.01296027  
-323613.64715722 1263.96680369]  
RMSE: 24.10670188328794

## 13 Post-Calibration Plots (Induced TL Terms + MFAM Vector Sensor)

```
[ ]: postcal_plots(datetimes,
                  b_scalar,
                  b_scalar_truth,
                  cal_scalar,
                  mfam_vec_dcs)
```



## 14 Find/Apply TL Calibration Coefficients (Eddy TL Terms + MFAM Vector Sensor)

```
[ ]: tl_terms = [magtl.TollesLawsonTerms.EDDY]

scalar_terms = magtl.tolles_lawson_coefficients(vector      = b_vector,
                                                y_value      = b_scalar,
                                                time_delta   = delta_t,
                                                apply_filter = True,
                                                mag_filter   = filt,
                                                terms        = tl_terms)
```

```

print('TL Terms:', scalar_terms)

body_effects = magtl.tlc_compensation(vector = b_vector,
                                       tlc      = scalar_terms,
                                       terms   = tl_terms)

cal_scalar = b_scalar - body_effects
cal_scalar -= (cal_scalar.mean() - b_scalar.mean()) # Bad fix for issue where TL adds a huge offset

b_scalar_truth = np.ones(b_scalar.shape) * b_scalar.mean()
b_scalar_error = np.abs(cal_scalar - b_scalar_truth)

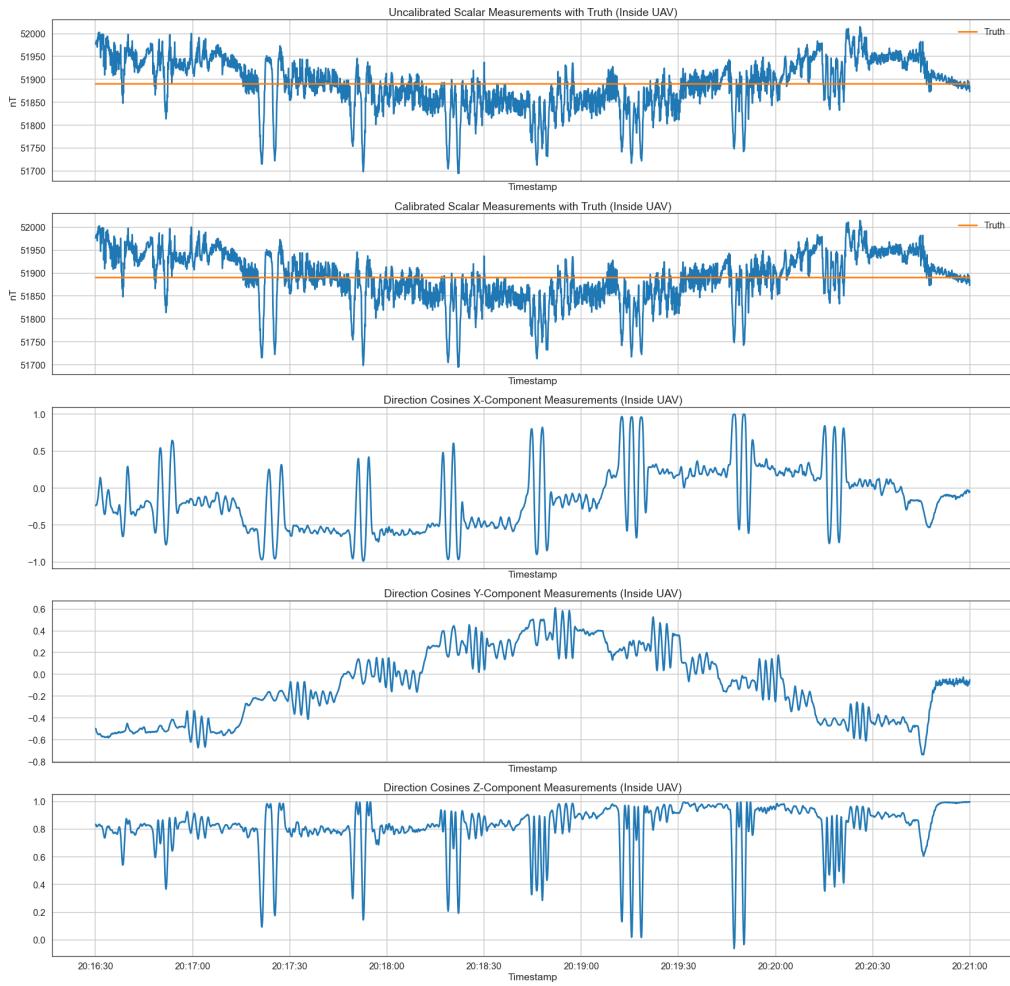
print('RMSE:', pu.rmse(cal_scalar, b_scalar_truth))

```

TL Terms: [ 7.76481406e+05 -1.09619892e+03 8.68036381e+03 1.20339628e+02  
  8.24690673e+05 -2.77691813e+03 6.40046539e+03 -6.37947861e+03  
  8.03093653e+05]  
RMSE: 54.83502706739999

## 15 Post-Calibration Plots (Eddy TL Terms + MFAM Vector Sensor)

```
[ ]: postcal_plots(datetimes,
                  b_scalar,
                  b_scalar_truth,
                  cal_scalar,
                  mfam_vec_dcs)
```



## 16 Find/Apply Tolles Lawson (TL) Calibration Coefficients (All TL Terms + VMR Vector Sensor)

```
[ ]: tl_terms = magtl.DEFAULT_TL_TERMS

b_vector = vmr

scalar_terms = magtl.tolles_lawson_coefficients(vector      = b_vector,
                                                y_value      = b_scalar,
                                                time_delta   = delta_t,
                                                apply_filter = True,
                                                mag_filter   = filt,
```

```

        terms      = tl_terms)

print('TL Terms:', scalar_terms)

body_effects = magtl.tlc_compensation(vector = b_vector,
                                         tlc      = scalar_terms,
                                         terms    = tl_terms)

cal_scalar = b_scalar - body_effects
cal_scalar -= (cal_scalar.mean() - b_scalar.mean()) # Bad fix for issue where
# TL adds a huge offset

b_scalar_truth = np.ones(b_scalar.shape) * b_scalar.mean()
b_scalar_error = np.abs(cal_scalar - b_scalar_truth)

print('RMSE:', pu.rmse(cal_scalar, b_scalar_truth))

```

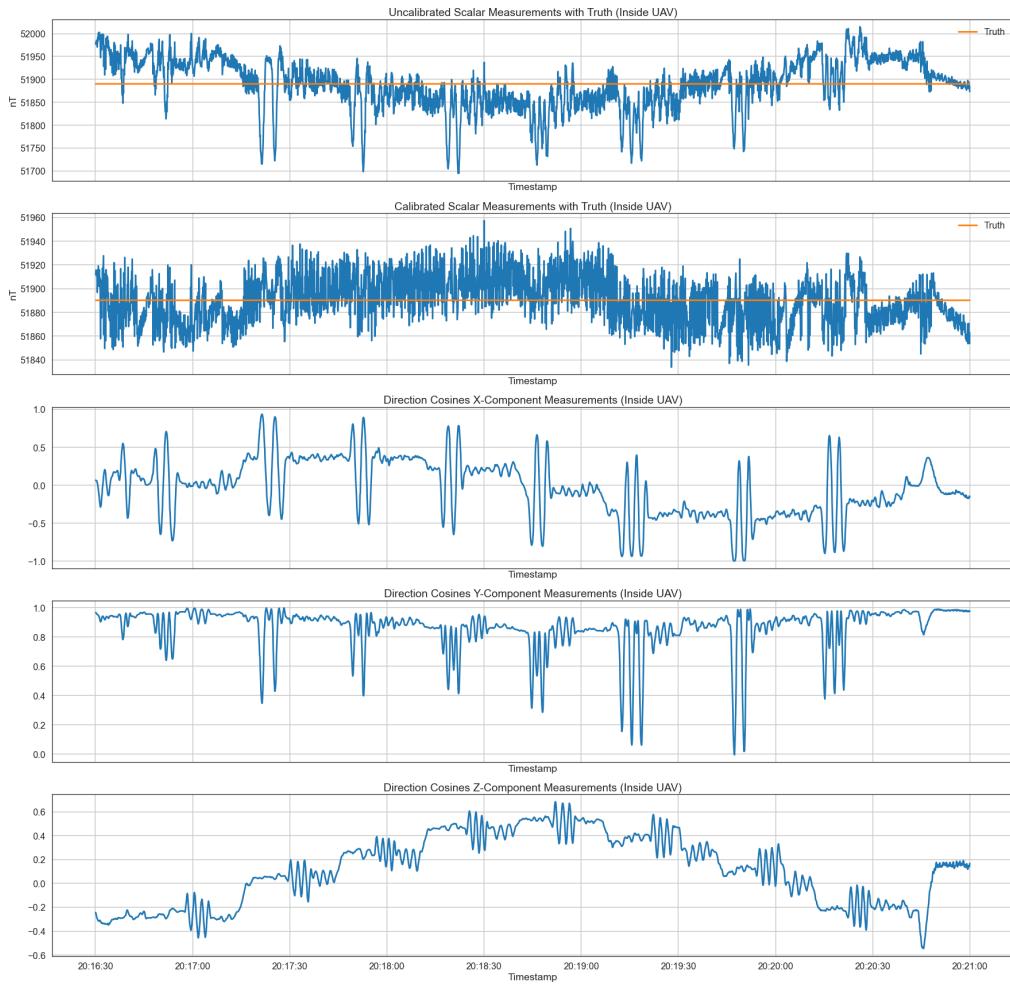
```

TL Terms: [-1.86687725e+01  1.33975396e+02 -1.80762945e+02  1.69023832e-01
-3.92262356e-03 -1.84382741e-03  1.71830230e-01 -1.61173781e-04
 1.72575427e-01 -4.31927864e-04 -8.21512835e-05 -4.37609432e-05
-1.06838978e-04 -1.22444017e-04 -2.76294434e-04 -8.51727772e-05
 3.16374022e-05 -2.77441572e-05]
RMSE: 20.421044137016235

```

## 17 Post-Calibration Plots (All TL Terms + VMR Vector Sensor)

```
[ ]: postcal_plots(datetimes,
                  b_scalar,
                  b_scalar_truth,
                  cal_scalar,
                  vmr_dcs)
```



## 18 Find/Apply TL Calibration Coefficients (Permanent and Induced TL Terms + VMR Vector Sensor)

```
[ ]: tl_terms = [magtl.TollesLawsonTerms.PERMANENT,
               magtl.TollesLawsonTerms.INDUCED]

scalar_terms = magtl.tolles_lawson_coefficients(vector      = b_vector,
                                                y_value      = b_scalar,
                                                time_delta   = delta_t,
                                                apply_filter = True,
                                                mag_filter   = filt,
                                                terms        = tl_terms)
```

```

print('TL Terms:', scalar_terms)

body_effects = magtl.tlc_compensation(vector = b_vector,
                                         tlc      = scalar_terms,
                                         terms   = tl_terms)

cal_scalar = b_scalar - body_effects
cal_scalar -= (cal_scalar.mean() - b_scalar.mean()) # Bad fix for issue where TL adds a huge offset

b_scalar_truth = np.ones(b_scalar.shape) * b_scalar.mean()
b_scalar_error = np.abs(cal_scalar - b_scalar_truth)

print('RMSE:', pu.rmse(cal_scalar, b_scalar_truth))

```

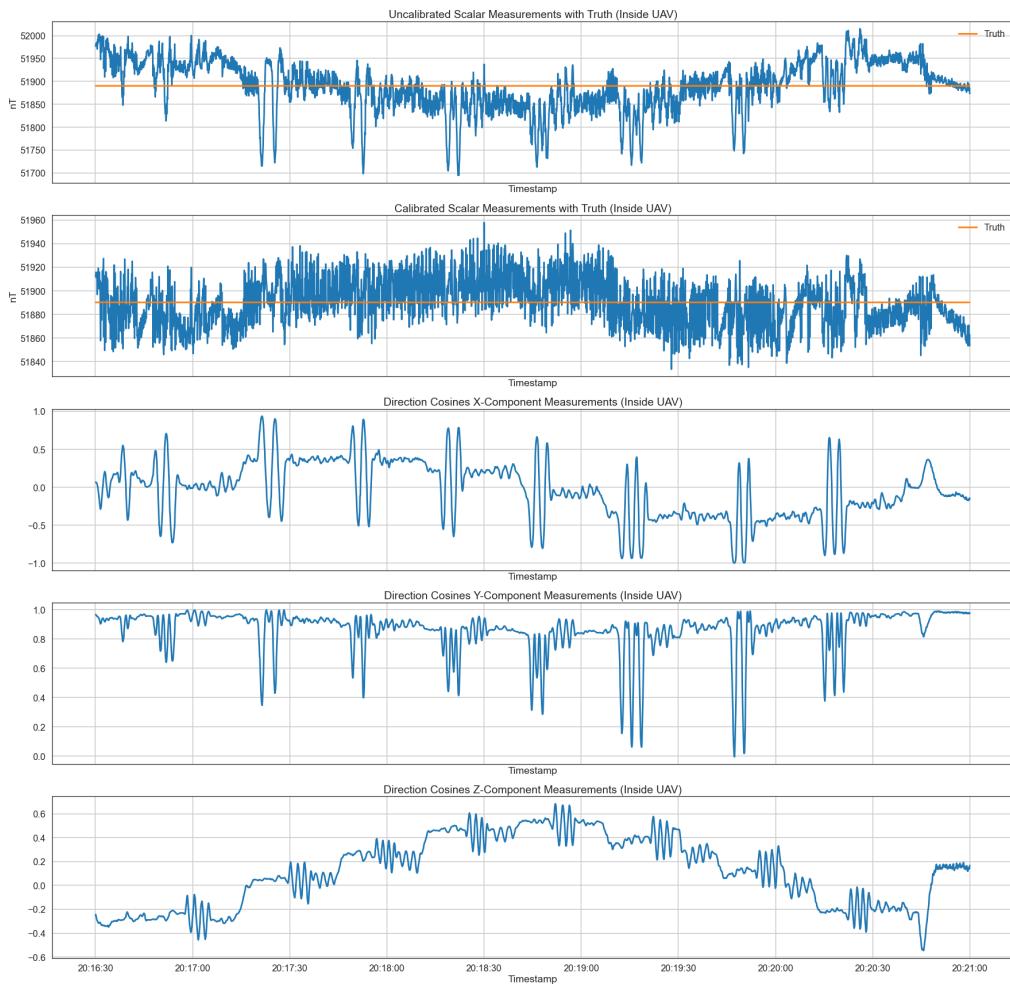
```

TL Terms: [-1.92711562e+01  1.34573533e+02 -1.79722461e+02  1.69142914e-01
-3.91198310e-03 -1.83581250e-03  1.71946543e-01 -1.88367158e-04
 1.72701774e-01]
RMSE: 20.504957380001493

```

## 19 Post-Calibration Plots (Permanent and Induced TL Terms + VMR Vector Sensor)

```
[ ]: postcal_plots(datetimes,
                  b_scalar,
                  b_scalar_truth,
                  cal_scalar,
                  vmr_dcs)
```



## 20 Find/Apply TL Calibration Coefficients (Permanent TL Terms + VMR Vector Sensor)

```
[ ]: tl_terms = [magtl.TollesLawsonTerms.PERMANENT]

scalar_terms = magtl.tolles_lawson_coefficients(vector      = b_vector,
                                                y_value     = b_scalar,
                                                time_delta = delta_t,
                                                apply_filter = True,
                                                mag_filter  = filt,
                                                terms       = tl_terms)
```

```

print('TL Terms:', scalar_terms)

body_effects = magtl.tlc_compensation(vector = b_vector,
                                       tlc      = scalar_terms,
                                       terms   = tl_terms)

cal_scalar = b_scalar - body_effects
cal_scalar -= (cal_scalar.mean() - b_scalar.mean()) # Bad fix for issue where TL adds a huge offset

b_scalar_truth = np.ones(b_scalar.shape) * b_scalar.mean()
b_scalar_error = np.abs(cal_scalar - b_scalar_truth)

print('RMSE:', pu.rmse(cal_scalar, b_scalar_truth))

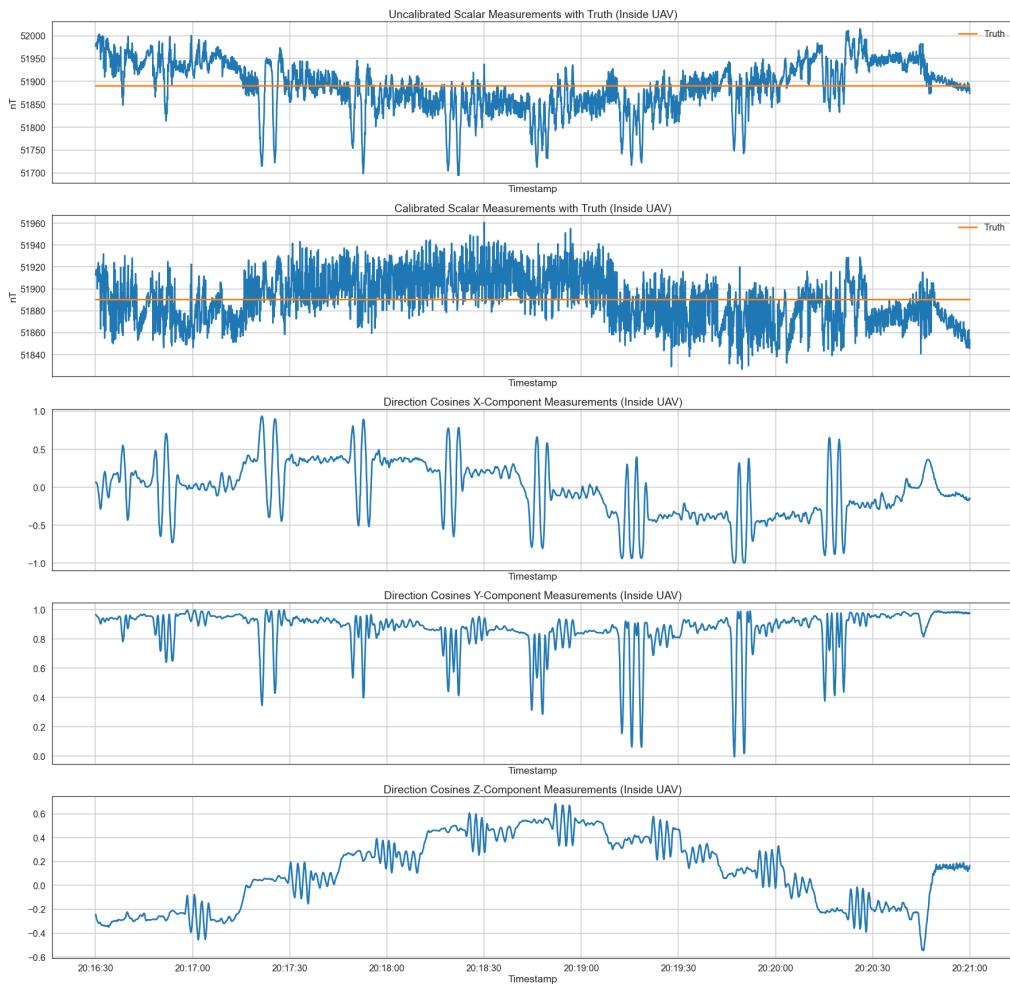
```

TL Terms: [-72.37506349 239.17181255 -131.38415659]

RMSE: 22.774860653433652

## 21 Post-Calibration Plots (Permanent TL Terms + VMR Vector Sensor)

```
[ ]: postcal_plots(datetimes,
                  b_scalar,
                  b_scalar_truth,
                  cal_scalar,
                  vmr_dcs)
```



## 22 Find/Apply TL Calibration Coefficients (Induced TL Terms + VMR Vector Sensor)

```
[ ]: tl_terms = [magtl.TollesLawsonTerms.INDUCED]

scalar_terms = magtl.tolles_lawson_coefficients(vector      = b_vector,
                                                y_value     = b_scalar,
                                                time_delta = delta_t,
                                                apply_filter = True,
                                                mag_filter  = filt,
                                                terms       = tl_terms)
```

```

print('TL Terms:', scalar_terms)

body_effects = magtl.tlc_compensation(vector = b_vector,
                                       tlc      = scalar_terms,
                                       terms   = tl_terms)

cal_scalar = b_scalar - body_effects
cal_scalar -= (cal_scalar.mean() - b_scalar.mean()) # Bad fix for issue where TL adds a huge offset

b_scalar_truth = np.ones(b_scalar.shape) * b_scalar.mean()
b_scalar_error = np.abs(cal_scalar - b_scalar_truth)

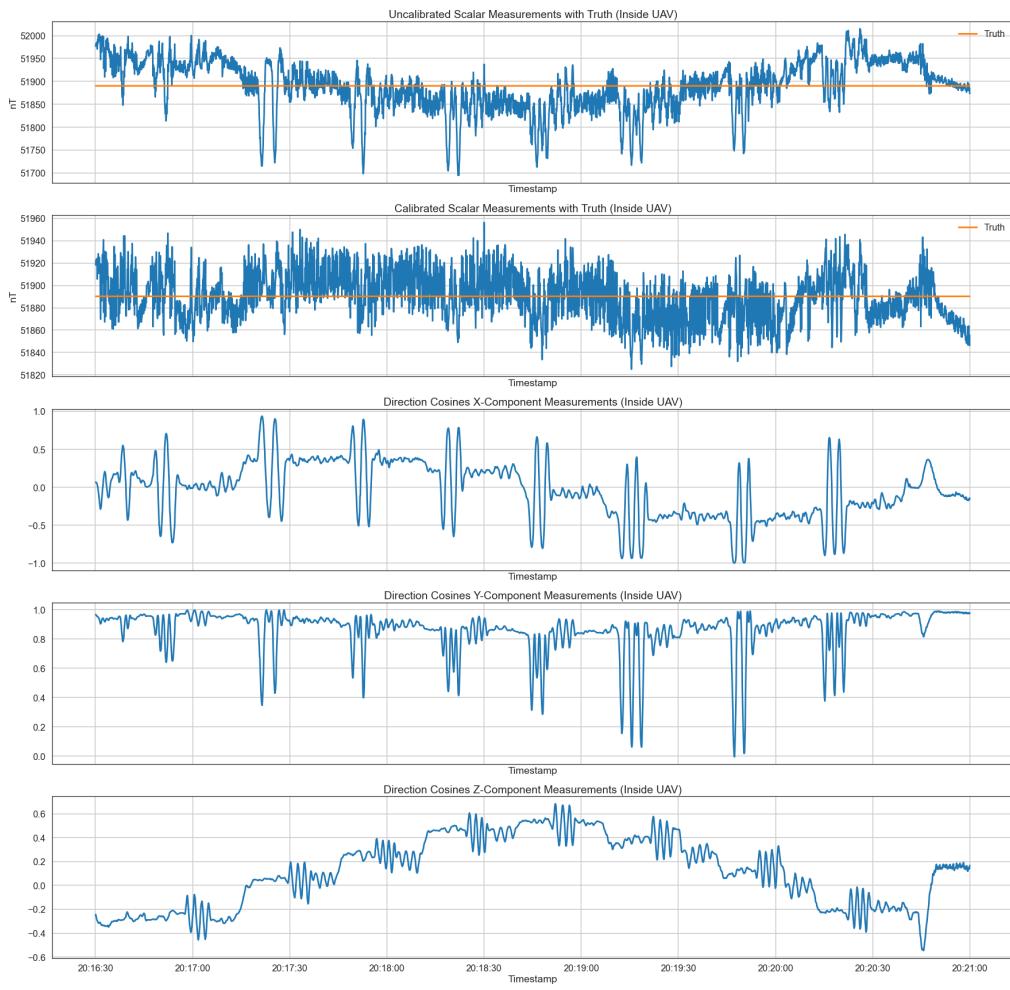
print('RMSE:', pu.rmse(cal_scalar, b_scalar_truth))

```

TL Terms: [ 0.12050356 -0.00380869 -0.0011042  0.12517448 -0.00311316  
0.12333525]  
RMSE: 21.822278973517687

## 23 Post-Calibration Plots (Induced TL Terms + VMR Vector Sensor)

```
[ ]: postcal_plots(datetimes,
                  b_scalar,
                  b_scalar_truth,
                  cal_scalar,
                  vmr_dcs)
```



## 24 Find/Apply TL Calibration Coefficients (Eddy TL Terms + VMR Vector Sensor)

```
[ ]: tl_terms = [magtl.TollesLawsonTerms.EDDY]

scalar_terms = magtl.tolles_lawson_coefficients(vector      = b_vector,
                                                y_value      = b_scalar,
                                                time_delta   = delta_t,
                                                apply_filter = True,
                                                mag_filter   = filt,
                                                terms        = tl_terms)
```

```

print('TL Terms:', scalar_terms)

body_effects = magtl.tlc_compensation(vector = b_vector,
                                       tlc      = scalar_terms,
                                       terms   = tl_terms)

cal_scalar = b_scalar - body_effects
cal_scalar -= (cal_scalar.mean() - b_scalar.mean()) # Bad fix for issue where TL adds a huge offset

b_scalar_truth = np.ones(b_scalar.shape) * b_scalar.mean()
b_scalar_error = np.abs(cal_scalar - b_scalar_truth)

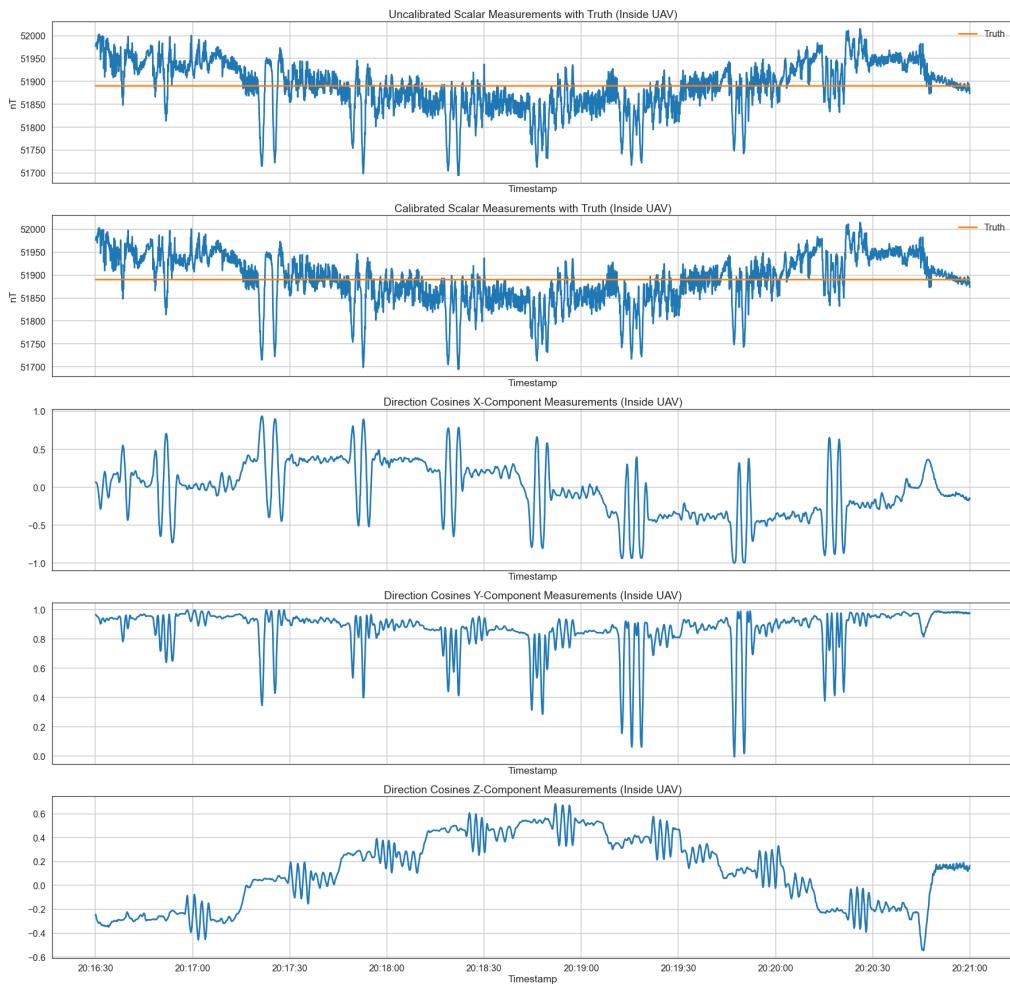
print('RMSE:', pu.rmse(cal_scalar, b_scalar_truth))

```

TL Terms: [ 1.77256213e-03 -1.07703822e-04 -1.41034304e-04 -1.24440384e-04  
 2.12831789e-03 -2.86511041e-04 -5.66089448e-05 3.04021500e-05  
 2.23973567e-03]  
RMSE: 54.8111443206224

## 25 Post-Calibration Plots (Eddy TL Terms + VMR Vector Sensor)

```
[ ]: postcal_plots(datetimes,
                   b_scalar,
                   b_scalar_truth,
                   cal_scalar,
                   vmr_dcs)
```



## \_\_\_\_\_process\_mini\_atterbury\_survey\_\_\_\_\_

December 20, 2022

```
[ ]: import sys
from os.path import join

import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
import pandas as pd
import scipy.linalg as la
from matplotlib import cm

PROJ_DIR = r'C:\Users\ltber\Desktop\files\AFIT\Research\WPAFB_Survey\mammal'
sys.path.append(PROJ_DIR)

from MAMMAL import Diurnal
from MAMMAL.MapLvl import pcaLvl
from MAMMAL.MapLvl import tieLvl
from MAMMAL.Parse import parseGSMP as pgsm
from MAMMAL.Parse import parseIM as pim
from MAMMAL.Parse import parseLCM as plcm
from MAMMAL.Utils import coordinateUtils as cu
from MAMMAL.Utils import Filters as filt
from MAMMAL.Utils import mapUtils as mu
from MAMMAL.Utils import ProcessingUtils as pu
from MAMMAL.VehicleCal import magUtilsTL as magtl

%matplotlib inline
plt.rcParams["figure.figsize"] = (30, 15) # (w, h)
plt.style.use(['seaborn-poster', 'seaborn-white'])
pd.set_option('mode.chained_assignment', None)

shrink = 0.8
aspect = 20 * 0.7
xlims = [-86.025, -85.995]
ylims = [39.335, 39.35]
cmap = cm.coolwarm
s = 10
```

```

alpha = 0.85

GMAP_EXTENT = [-86.054121, -85.981633, 39.319868, 39.349562]

SURVEY_DIR = r'D:\__atterbury_data__\mini_atterbury_survey_'
GMAP_FNAME = r'D:\__atterbury_data__\atterbury.jpg'
REF_FNAME = r'D:\__atterbury_data__\mini_atterbury_survey_\Ground_Reference.
    ↪CSV'
LOG_FNAME = r'D:
    ↪\__atterbury_data__\mini_atterbury_survey_\__mini_atterbury_survey__.csv'
KML_FNAME = r'Mini_Atterbury_Survey.kml'

SURVEY_NAME = '1km Atterbury Survey (First Attempt)'

MAX_TERRAIN_MSL = 230 # (m)
MAX_SURVEY_AGL = 400 # (m)
MAX_SURVEY_CRUISE = 30 # (m/s)

MAX_EXPECTED_FREQ = MAX_SURVEY_CRUISE / MAX_SURVEY_AGL # (hz)

TL_C = np.array([-1.86687725e+01, 1.33975396e+02, -1.80762945e+02, 1.
    ↪69023832e-01,
                    -3.92262356e-03, -1.84382741e-03, 1.71830230e-01, -1.
    ↪61173781e-04,
                    1.72575427e-01, -4.31927864e-04, -8.21512835e-05, -4.
    ↪37609432e-05,
                    -1.06838978e-04, -1.22444017e-04, -2.76294434e-04, -8.
    ↪51727772e-05,
                    3.16374022e-05, -2.77441572e-05])

TL_TERMS = magtl.DEFAULT_TL_TERMS
TL_COEFF_TYPES = ['Permanent', 'Induced', 'Eddy']

SCALAR_TYPE = 'Geometrics MFAM optically pumped caesium scalar magnetometer
    ↪with 2 sensor heads'
VECTOR_TYPE = 'TwinLeaf VMR anisotropic magnetoresistive vector magnetometer'

FINAL_filt_CUT = MAX_SURVEY_AGL
FINAL_filt_ORDER = 6

DX = 5 # Map pixel width (m)
DY = 5 # Map pixel height (m)

POC = '''Autonomy and Navigation Technology Center
        Air Force Institute of Technology
        Graduate School of Engineering and Management

```

```

2950 Hobson Way, Bldg 646, Rm 205
Wright Patterson AFB, Ohio 45433

Telephone: (937) 255-3636 Ext. 4671'''

PROCESS_APP = 'MAMMAL 0.0.1'

```

## 1 Load Survey Data

```

[ ]: img = mpimg.imread(GMAP_FNAME)

ref_df = pd.read_csv(REF_FNAME, parse_dates=['datetime'])

log_df      = pd.read_csv(LOG_FNAME, parse_dates=['datetime'])
datetimes   = log_df.datetime
timestamps  = np.array(log_df.epoch_sec)
sample_rate = 1.0 / np.diff(timestamps).mean()

```

## 2 Interpolate Reference Data

```

[ ]: # Despike reference data
span  = 10
delta = 0.5

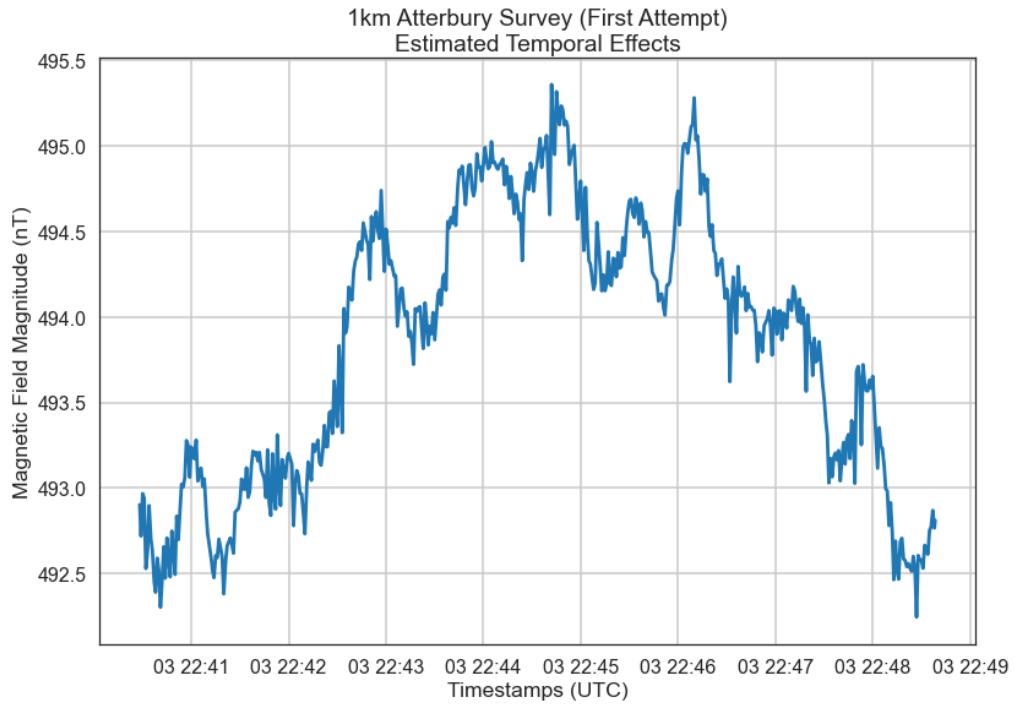
ref_despiked = pu.remove_outliers(ref_df.F, pu.ewma_fb(ref_df.F, span), delta)

ref_df.F = ref_despiked
ref_df.F = ref_df.F.interpolate()

# Interpolate reference data
_, ref_mag = Diurnal.interp_reference_df(df           = ref_df,
                                           timestamps = timestamps,
                                           survey_lon = log_df.LONG.mean(),
                                           subtract_core = True)

plt.figure()
plt.title('{}\\nEstimated Temporal Effects'.format(SURVEY_NAME))
plt.xlabel('Timestamps (UTC)')
plt.ylabel('Magnetic Field Magnitude (nT)')
plt.plot(log_df.datetime, ref_mag)
plt.grid()

```



### 3 Determine When Each Sensor Head was Valid During the Survey

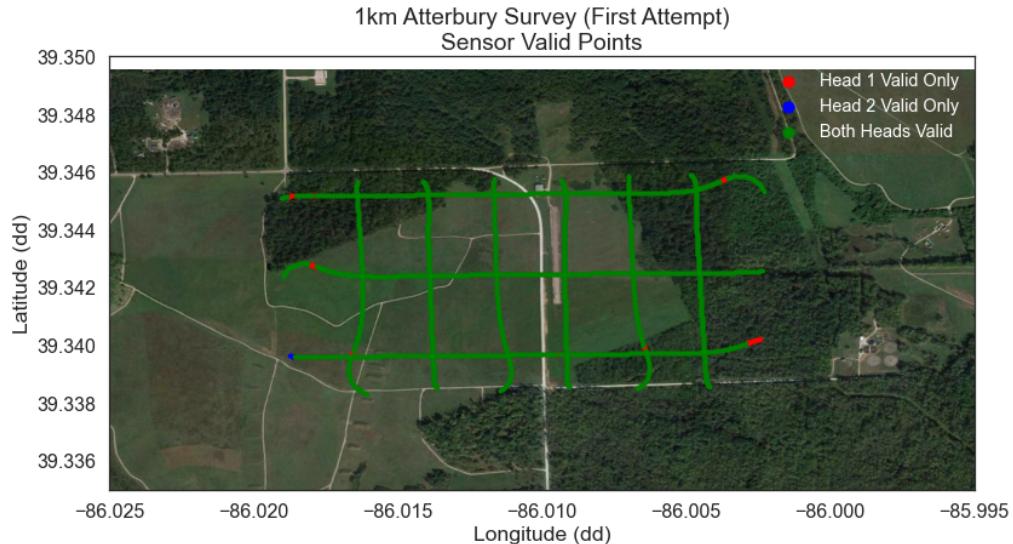
```
[ ]: plt.figure()
plt.title('{}\\nSensor Valid Points'.format(SURVEY_NAME))
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
plt.scatter(log_df.LONG[(log_df.SCALAR_1_VALID == 1) & (log_df.SCALAR_2_VALID == 0) & (log_df.LINE_TYPE != 0)],
            log_df.LAT[(log_df.SCALAR_1_VALID == 1) & (log_df.SCALAR_2_VALID == 0) & (log_df.LINE_TYPE != 0)],
            s=s,
            c='r',
            label='Head 1 Valid Only')
plt.scatter(log_df.LONG[(log_df.SCALAR_1_VALID == 0) & (log_df.SCALAR_2_VALID == 1) & (log_df.LINE_TYPE != 0)],
            log_df.LAT[(log_df.SCALAR_1_VALID == 0) & (log_df.SCALAR_2_VALID == 1) & (log_df.LINE_TYPE != 0)],
            s=s,
            c='b',
```

```

        label='Head 2 Valid Only')
plt.scatter(log_df.LONG[(log_df.SCALAR_1_VALID == 1) & (log_df.SCALAR_2_VALID == 1) & (log_df.LINE_TYPE != 0)],
            log_df.LAT[(log_df.SCALAR_1_VALID == 1) & (log_df.SCALAR_2_VALID == 1) & (log_df.LINE_TYPE != 0)],
            s=s,
            c='g',
            label='Both Heads Valid')
plt.legend(labelcolor='white', fontsize='x-large', markerscale=3)
plt.xlim(xlims)
plt.ylim(ylims)

```

[ ]: (39.335, 39.35)



## 4 Apply Tolles Lawson Calibration

```

[ ]: # Compile vector data
b_vector = np.hstack([np.array(log_df.X)[:, np.newaxis],
                      np.array(log_df.Y)[:, np.newaxis],
                      np.array(log_df.Z)[:, np.newaxis]])

dcs  = b_vector / la.norm(b_vector, axis=1)[:, np.newaxis]
dc_x = dcs[:, 0]
dc_y = dcs[:, 1]
dc_z = dcs[:, 2]

```

```

# Calibrate sensor head
f = (log_df.SCALAR_1_LPF + log_df.SCALAR_2_LPF) / 2.0

body_effects_scalar = magtl.tlc_compensation(vector = b_vector,
                                              tlc      = TL_C,
                                              terms   = TL_TERMS)

f_cal = f - body_effects_scalar

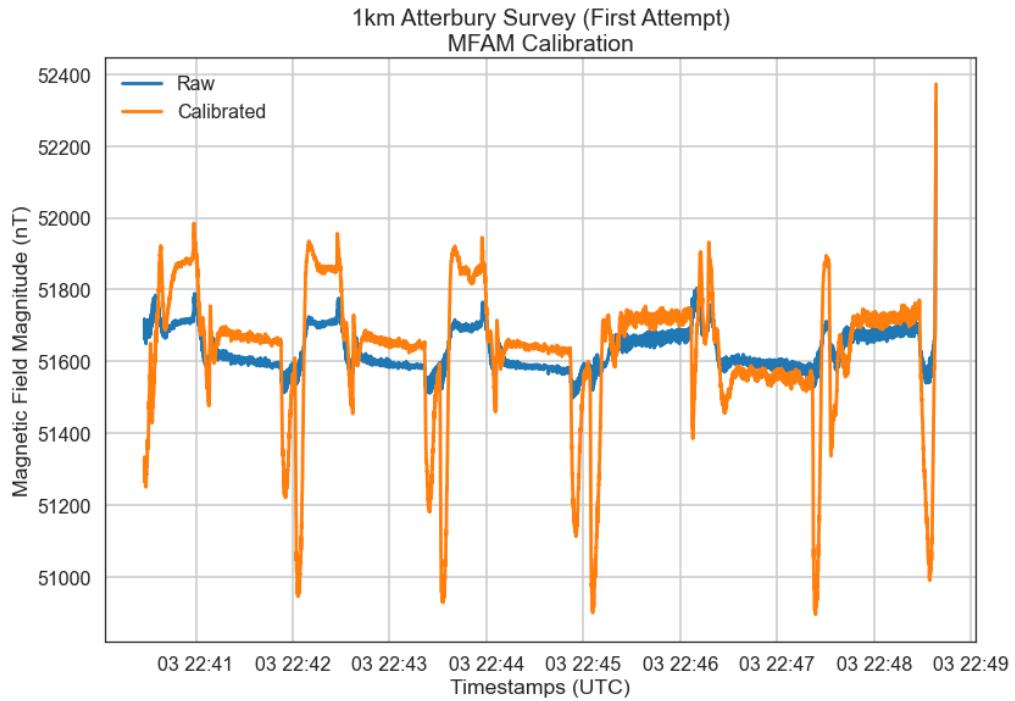
f_cal += (f.mean() - f_cal.mean())

# Update dataframe with calibrated data
log_df['SCALAR_CAL'] = f_cal
log_df['F']           = log_df.SCALAR_CAL

plt.figure()
plt.title('{}\\nMFAM Calibration'.format(SURVEY_NAME))
plt.xlabel('Timestamps (UTC)')
plt.ylabel('Magnetic Field Magnitude (nT)')
plt.plot(log_df.datetime, f,      label='Raw')
plt.plot(log_df.datetime, f_cal, label='Calibrated')
plt.legend()
plt.grid()

# Calibration was worse than original - skip cal step
log_df['F'] = f

```



## 5 Find Estimated Magnetic Anomaly Values

```
[ ]: f_cal_igrf           = f_cal - log_df.IGRF_F
f_cal_igrf_temporal      = f_cal_igrf - ref_mag
f_cal_igrf_temporal_filt = filt.lpf(f_cal_igrf_temporal, MAX_EXPECTED_FREQ, sample_rate)

process_dict = {'TIMESTAMP': timestamps,
               'LAT': log_df.LAT,
               'LONG': log_df.LONG,
               'ALT': log_df.ALT,
               'DC_X': dc_x,
               'DC_Y': dc_y,
               'DC_Z': dc_z,
               'F': f,
               'F_CAL': f_cal,
               'F_CAL_IGRF': f_cal_igrf,
               'F_CAL_IGRF_TEMPORAL': f_cal_igrf_temporal,
               'F_CAL_IGRF_TEMPORAL_FILTER': f_cal_igrf_temporal_filt}

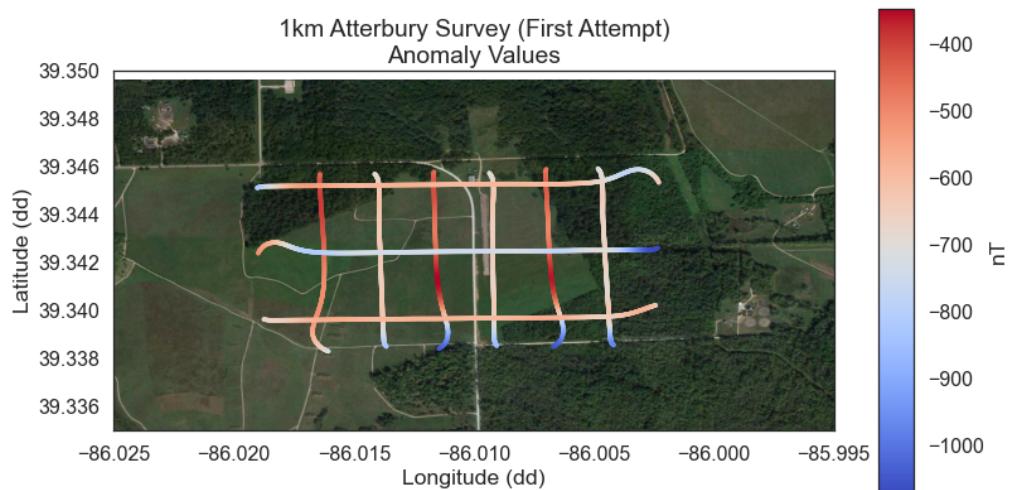
log_df.F = f_cal_igrf_temporal_filt
```

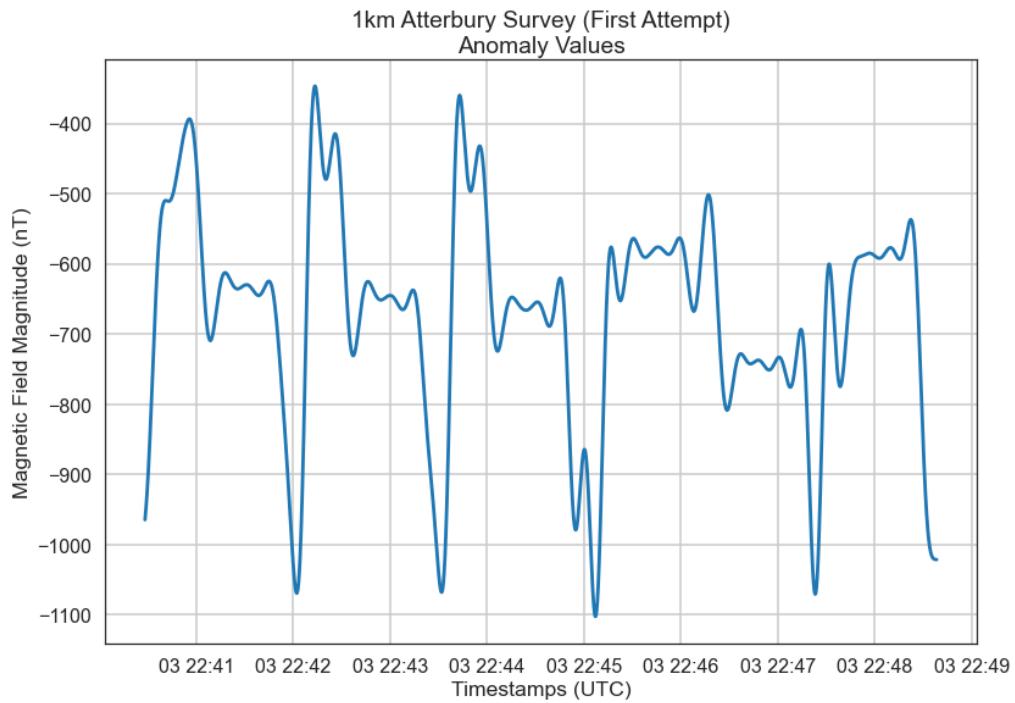
```

plt.figure()
plt.title('{}\\nAnomaly Values'.format(SURVEY_NAME))
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.scatter(log_df.LOGN[log_df.LINE_TYPE != 0],
                  log_df.LAT[log_df.LINE_TYPE != 0],
                  s=s,
                  c=log_df.F[log_df.LINE_TYPE != 0],
                  cmap=cmap)
plt.xlim(xlims)
plt.ylim(ylims)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)

plt.figure()
plt.title('{}\\nAnomaly Values'.format(SURVEY_NAME))
plt.xlabel('Timestamps (UTC)')
plt.ylabel('Magnetic Field Magnitude (nT)')
plt.plot(log_df.datetime, log_df.F)
plt.grid()

```





## 6 Crop Map Extent

```
[ ]: MIN_MAP_LAT = 39.3395
      MAX_MAP_LAT = 39.3445
```

## 7 Create Maps Based on Non-Leveled Flight Lines

```
[ ]: interp_type = 'RBF'
interp_df = pu.interp_flight_lines(anomaly_df      = log_df[(log_df.LAT >= MIN_MAP_LAT) & (log_df.LAT <= MAX_MAP_LAT)],
                                    dx          = DX,
                                    dy          = DY,
                                    max_terrain_msl = MAX_TERRAIN_MSL,
                                    buffer      = 0,
                                    interp_type = interp_type,
                                    neighbors   = None,
                                    skip_na_mask = True)

interp_lats    = interp_df['LAT']
interp_lons    = interp_df['LONG']
```

```

interp_scalar = interp_df['F']
interp_heights = interp_df['ALT']
interp_std     = interp_df['STD']

interp_scalar_LPF = filt.lpf2(interp_scalar,
                               MAX_SURVEY_AGL,
                               DX,
                               DY)

map_title = '{}\nNon-Leveled Anomaly Map\nUsing {} Interpolation'.
            .format(SURVEY_NAME, interp_type)

plt.figure()
plt.title(map_title)
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.pcolormesh(interp_lons,
                     interp_lats,
                     interp_scalar,
                     cmap=cmap,
                     alpha=alpha)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)

map_title = '{}\nNon-Leveled Anomaly Map\nUsing {} Interpolation (Filtered)'.
            .format(SURVEY_NAME, interp_type)

plt.figure()
plt.title(map_title)
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.pcolormesh(interp_lons,
                     interp_lats,
                     interp_scalar_LPF,
                     cmap=cmap,
                     alpha=alpha)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)

extent = [interp_lons.min(), interp_lats.min(), interp_lons.max(), interp_lats.
          .max()]
xmin, ymin, xmax, ymax = extent

```

```

flight_path = np.hstack([np.array(log_df.LAT)[:, np.newaxis],
                        np.array(log_df.LONG)[:, np.newaxis],
                        np.array(log_df.ALT)[:, np.newaxis],
                        np.array(log_df.epoch_sec)[:, np.newaxis]]))

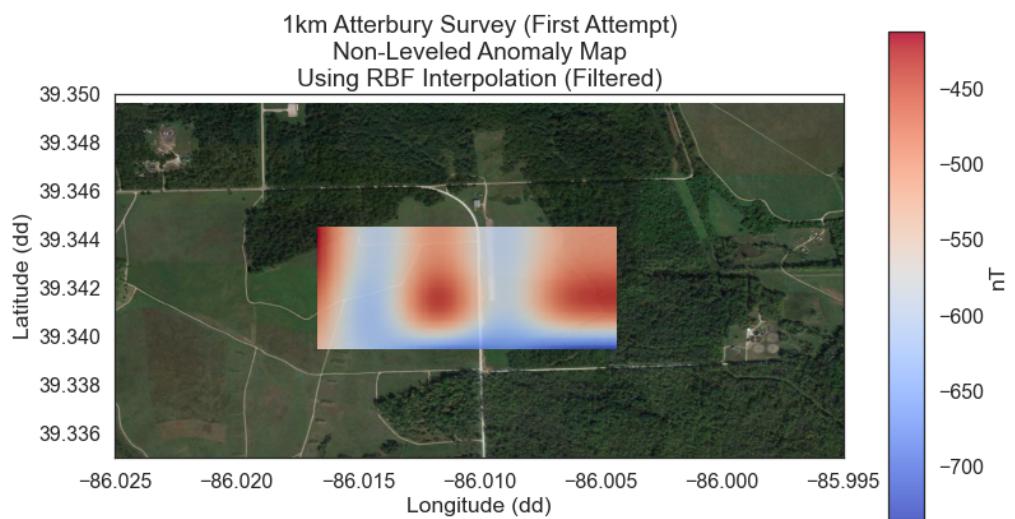
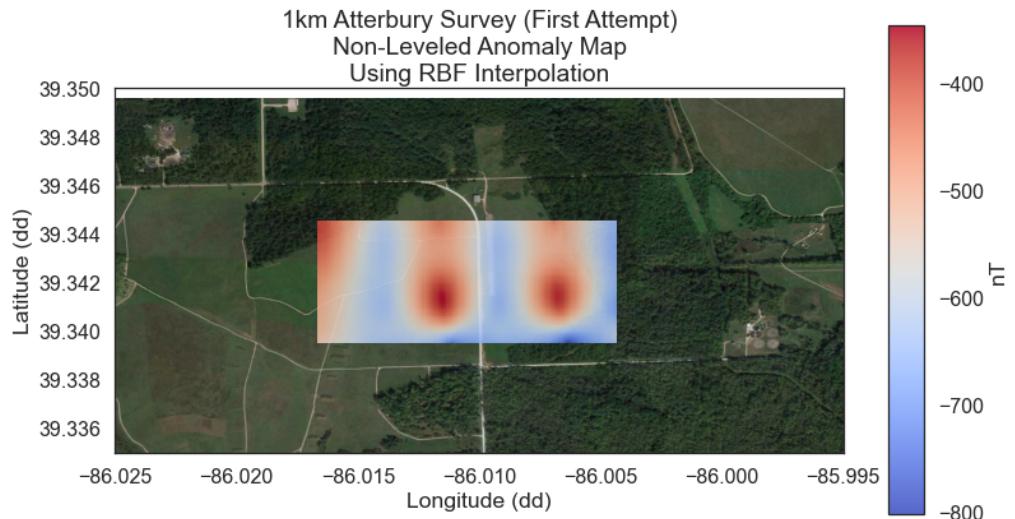
area_polys = [{'NAME': 'Survey Area',
               'FL_DIRFL_DISTTL_DIRTL_DISTLATLONGALT# Export map as a GeoTIFF
map = mu.export_map(out_dir
                     location = SURVEY_DIR,
                     date = ' '.join(map_title.split()),
                     lats = interp_lats,
                     lons = interp_lons,
                     scalar = interp_scalar_LPF,
                     heights = interp_heights,
                     process_df = pd.DataFrame(process_dict),
                     process_app = PROCESS_APP,
                     stds = interp_std,
                     vector = None,
                     scalar_type = SCALAR_TYPE,
                     vector_type = VECTOR_TYPE,
                     scalar_var = np.nan,
                     vector_var = np.nan,
                     poc = POC,
                     flight_path = flight_path,
                     area_polys = area_polys,
                     osm_path = None,
                     level_type = 'No leveling',
                     tl_coeff_types = TL_COEFF_TYPES,
                     tl_coeffs = TL_C,
                     interp_type = interp_type,
                     final_filt_cut = FINAL_FILT_CUT,
                     final_filt_order = FINAL_FILT_ORDER)

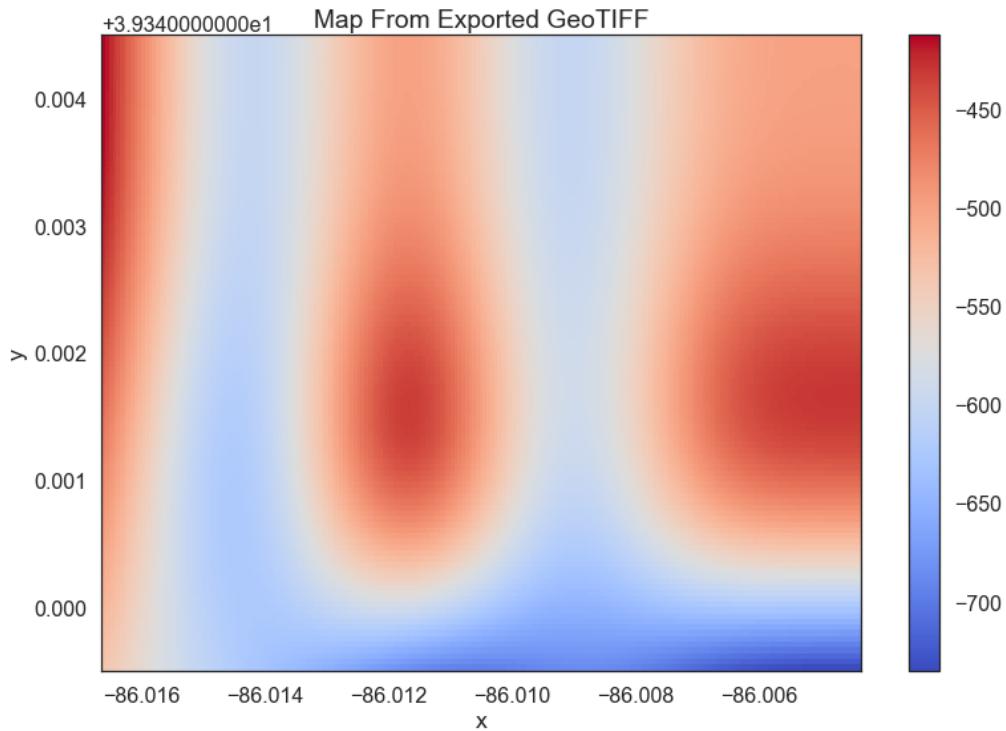
plt.figure()
map[0].plot(cmap=cm.coolwarm)
plt.title('Map From Exported GeoTIFF')

```

Interpolating survey anomaly data to map coordinates  
Running radial basis function (RBF) interpolation for all map pixels

[ ]: Text(0.5, 1.0, 'Map From Exported GeoTIFF')



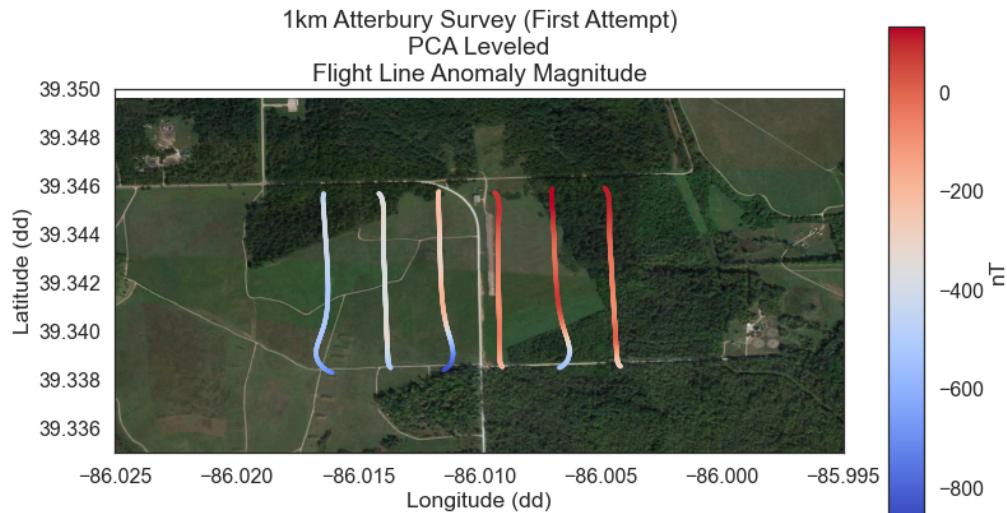


## 8 Apply PCA-Based Flight Line Leveling

```
[ ]: lvld_survey_df = pcaLvl.pca_lvl(survey_df = log_df,
                                      num_ptls = 2,
                                      ptl_locs = np.array([0.0, 1.0]))

plt.figure()
plt.title('{}\\nPCA Leveled\\nFlight Line Anomaly Magnitude'.format(SURVEY_NAME))
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.scatter(lvld_survey_df.LONG[lvld_survey_df.LINE_TYPE == 1],
                 lvld_survey_df.LAT[lvld_survey_df.LINE_TYPE == 1],
                 s=s,
                 c=lvld_survey_df.F[lvld_survey_df.LINE_TYPE == 1],
                 cmap=cmap)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)
```

[ ]: (39.335, 39.35)



## 9 Create Maps Based on PCA-Leveled Flight Lines

```
[ ]: interp_type = 'RBF'
interp_df = pu.interp_flight_lines(anomaly_df
    ↪lvld_survey_df[(lvld_survey_df.LAT >= MIN_MAP_LAT) & (lvld_survey_df.LAT <=
    ↪MAX_MAP_LAT)],
    dx          = DX,
    dy          = DY,
    max_terrain_msl = MAX_TERRAIN_MSL,
    buffer      = 0,
    interp_type = interp_type,
    neighbors   = None,
    skip_na_mask = True)

interp_lats  = interp_df['LAT']
interp_lons  = interp_df['LONG']
interp_scalar = interp_df['F']
interp_heights = interp_df['ALT']
interp_std    = interp_df['STD']

interp_scalar_LPF = filt.lpf2(interp_scalar,
    MAX_SURVEY_AGL,
    DX,
    DY)
```

```

map_title = '{}\nPCA Leveled Anomaly Map\nUsing {} Interpolation'.
    .format(SURVEY_NAME, interp_type)

plt.figure()
plt.title(map_title)
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.pcolormesh(interp_lons,
                     interp_lats,
                     interp_scalar,
                     cmap=cmap,
                     alpha=alpha)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)

map_title = '{}\nPCA Leveled Anomaly Map\nUsing {} Interpolation (Filtered)'.
    .format(SURVEY_NAME, interp_type)

plt.figure()
plt.title(map_title)
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.pcolormesh(interp_lons,
                     interp_lats,
                     interp_scalar_LPF,
                     cmap=cmap,
                     alpha=alpha)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)

process_dict['F_CAL_IGRF_TEMPORAL_FILT_LEVEL'] = lvld_survey_df.F

# Export map as a GeoTIFF
map = mu.export_map(out_dir
                    location = SURVEY_DIR,
                    date = ' '.join(map_title.split()),
                    lats = interp_lats,
                    lons = interp_lons,
                    scalar = interp_scalar_LPF,
                    heights = interp_heights,
                    process_df = pd.DataFrame(process_dict),
                    process_app = PROCESS_APP,
                    stds = interp_std,

```

```

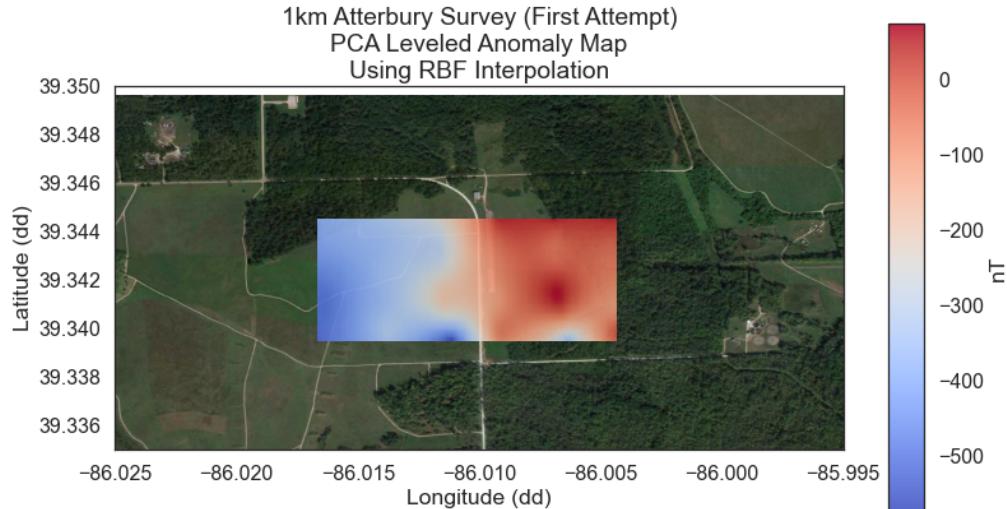
vector           = None,
scalar_type     = SCALAR_TYPE,
vector_type     = VECTOR_TYPE,
scalar_var      = np.nan,
vector_var      = np.nan,
poc              = POC,
flight_path     = flight_path,
area_polys      = area_polys,
osm_path         = None,
level_type       = 'PCA pseudo tie line leveling',
tl_coeff_types  = TL_COEFF_TYPES,
tl_coeffs        = TL_C,
interp_type      = interp_type,
final_filt_cut   = FINAL_FILT_CUT,
final_filt_order = FINAL_FILT_ORDER)

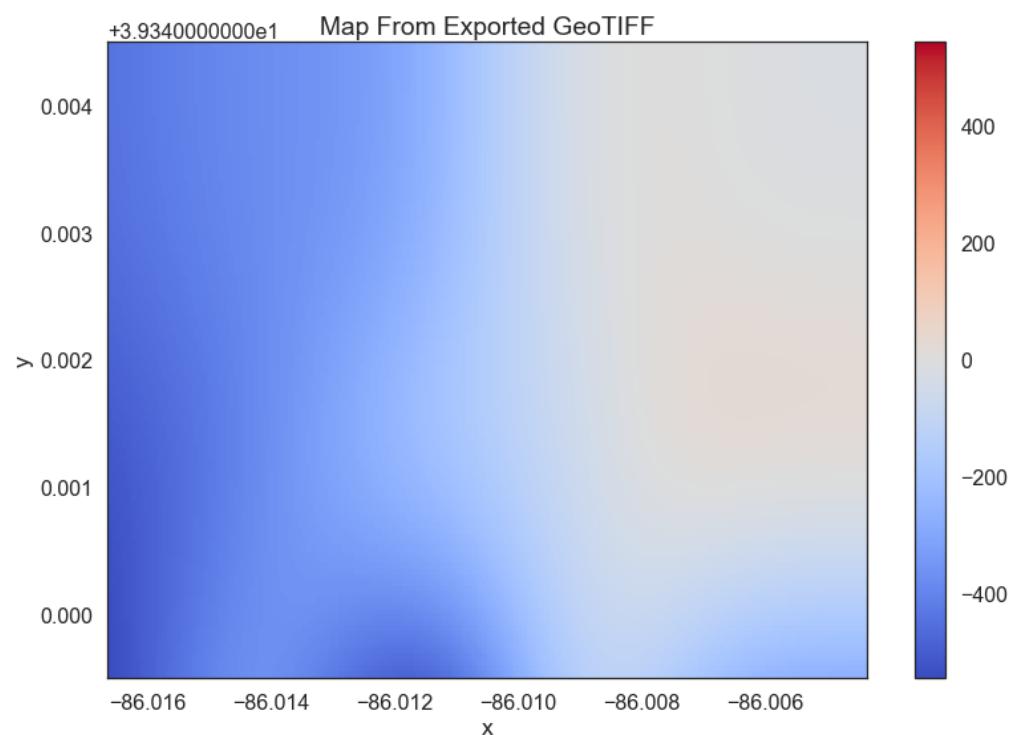
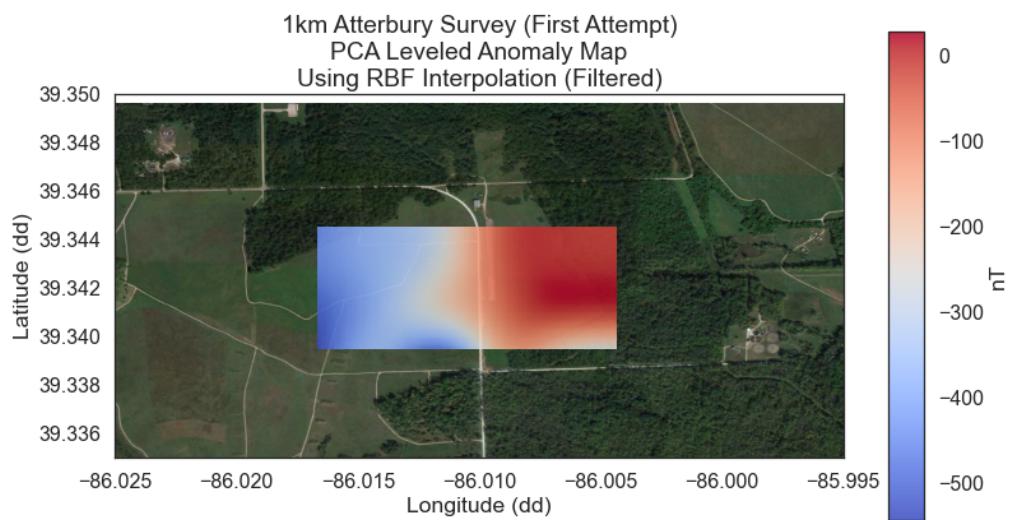
plt.figure()
map[0].plot(cmap=cm.coolwarm)
plt.title('Map From Exported GeoTIFF')

```

Interpolating survey anomaly data to map coordinates  
Running radial basis function (RBF) interpolation for all map pixels

[ ]: Text(0.5, 1.0, 'Map From Exported GeoTIFF')



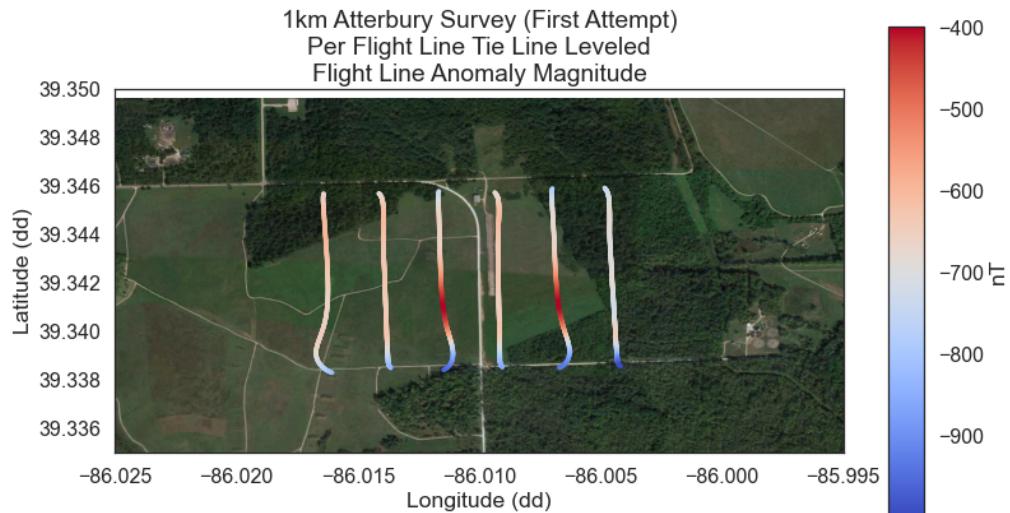


## 10 Apply Per Flight Line Tie Line-Based Flight Line Leveling

```
[ ]: lvld_survey_df = tieLvl.tie_lvl(survey_df = log_df,
                                     approach = 'lobf')

plt.figure()
plt.title('{}\\nPer Flight Line Tie Line Leveled\\nFlight Line Anomaly Magnitude'.
          format(SURVEY_NAME))
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.scatter(lvld_survey_df.LONG[lvld_survey_df.LINE_TYPE == 1],
                 lvld_survey_df.LAT[lvld_survey_df.LINE_TYPE == 1],
                 s=s,
                 c=lvld_survey_df.F[lvld_survey_df.LINE_TYPE == 1],
                 cmap=cmap)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)
```

```
[ ]: (39.335, 39.35)
```



## 11 Create Map Based on Per Flight Line Tie Line-Leveled Flight Lines

```
[ ]: interp_type = 'RBF'
interp_df = pu.interp_flight_lines(anomaly_df
                                   = lvld_survey_df[(lvld_survey_df.LAT >= MIN_MAP_LAT) & (lvld_survey_df.LAT <= MAX_MAP_LAT)],
                                   dx = DX,
                                   dy = DY,
                                   max_terrain_msl = MAX_TERRAIN_MSL,
                                   buffer = 0,
                                   interp_type = interp_type,
                                   neighbors = None,
                                   skip_na_mask = True)

interp_lats = interp_df['LAT']
interp_lons = interp_df['LONG']
interp_scalar = interp_df['F']
interp_heights = interp_df['ALT']
interp_std = interp_df['STD']

interp_scalar_LPF = filt.lpf2(interp_scalar,
                               MAX_SURVEY_AGL,
                               DX,
                               DY)

map_title = '{}\nPer Flight Line Tie Line Leveled Anomaly Map\nUsing {}'
            .format(SURVEY_NAME, interp_type)

plt.figure()
plt.title(map_title)
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.pcolormesh(interp_lons,
                     interp_lats,
                     interp_scalar,
                     cmap=cmap,
                     alpha=alpha)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)

map_title = '{}\nPer Flight Line Tie Line Leveled Anomaly Map\nUsing {}'
            .format(SURVEY_NAME, interp_type)

plt.figure()
```

```

plt.title(map_title)
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.pcolormesh(interp_lons,
                     interp_lats,
                     interp_scalar_LPF,
                     cmap=cmap,
                     alpha=alpha)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)

process_dict['F_CAL_IGRF_TEMPORAL_filt_LEVEL'] = lvld_survey_df.F

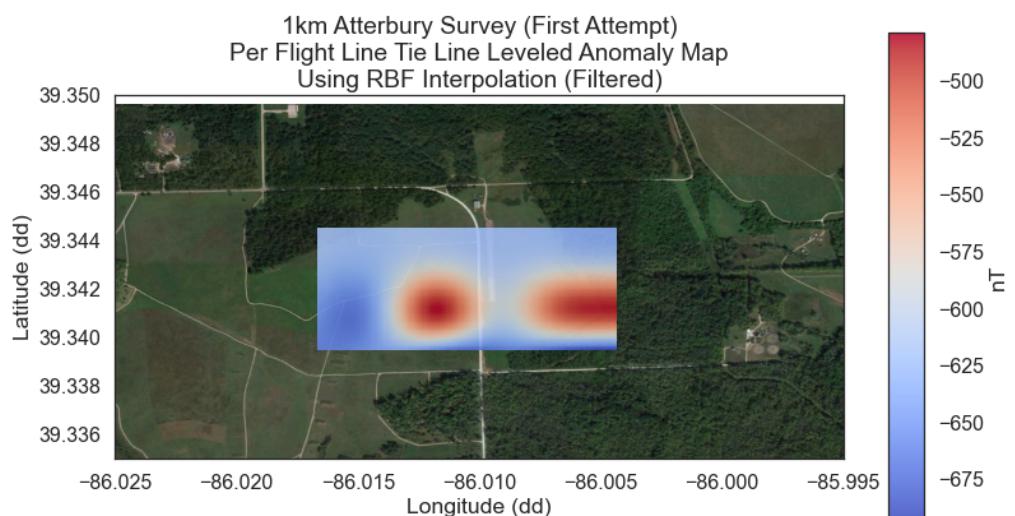
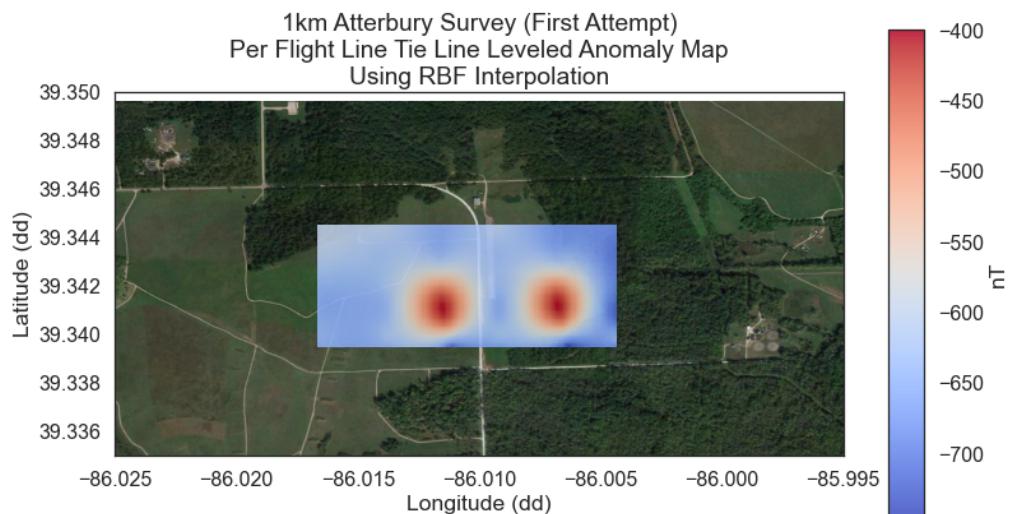
# Export map as a GeoTIFF
map = mu.export_map(out_dir
                     location = SURVEY_DIR,
                     date = ' '.join(map_title.split()),
                     lats = interp_lats,
                     lons = interp_lons,
                     scalar = interp_scalar_LPF,
                     heights = interp_heights,
                     process_df = pd.DataFrame(process_dict),
                     process_app = PROCESS_APP,
                     stds = interp_std,
                     vector = None,
                     scalar_type = SCALAR_TYPE,
                     vector_type = VECTOR_TYPE,
                     scalar_var = np.nan,
                     vector_var = np.nan,
                     poc = POC,
                     flight_path = flight_path,
                     area_polys = area_polys,
                     osm_path = None,
                     level_type = 'Per flight line tie line leveling',
                     tl_coeff_types = TL_COEFF_TYPES,
                     tl_coeffs = TL_C,
                     interp_type = interp_type,
                     final_filt_cut = FINAL_FILT_CUT,
                     final_filt_order = FINAL_FILT_ORDER)

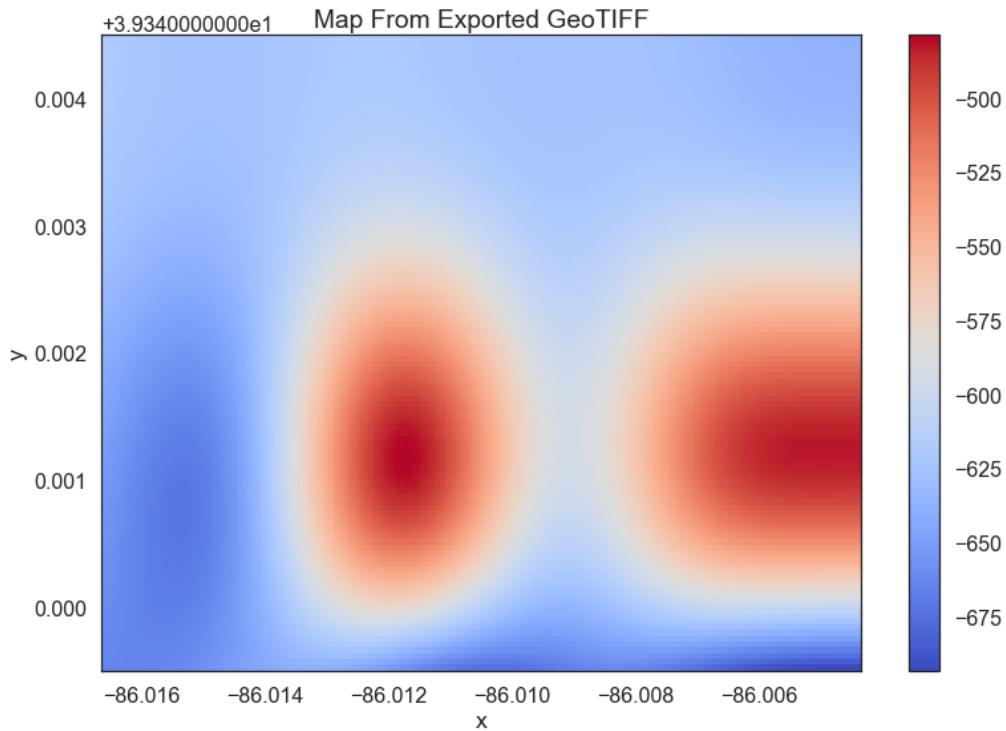
plt.figure()
map[0].plot(cmap=cm.coolwarm)
plt.title('Map From Exported GeoTIFF')

```

Interpolating survey anomaly data to map coordinates  
Running radial basis function (RBF) interpolation for all map pixels

[ ]: Text(0.5, 1.0, 'Map From Exported GeoTIFF')



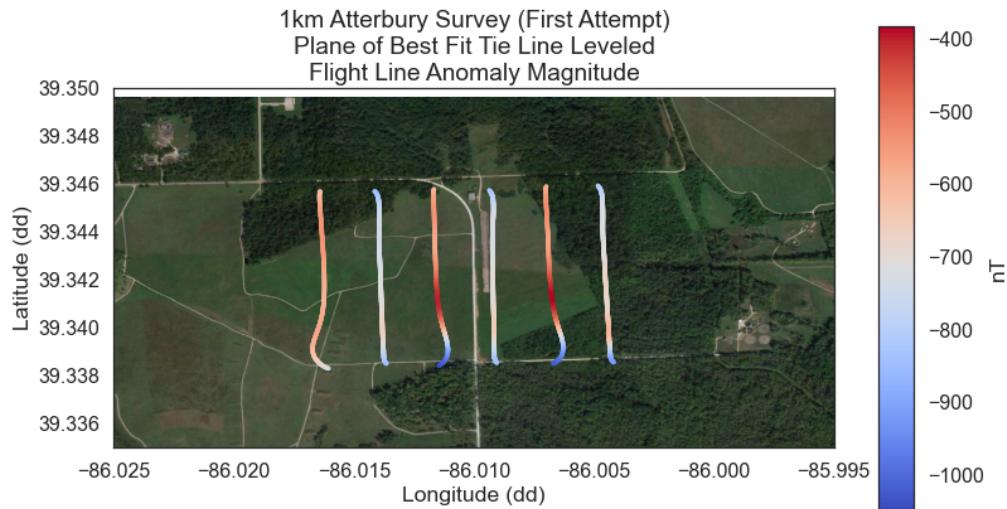


## 12 Apply Plane of Best Fit Tie Line-Based Flight Line Leveling

```
[ ]: lvld_survey_df = tieLvl.tie_lvl(survey_df = log_df,
                                     approach = 'lsq')

plt.figure()
plt.title('{} \nPlane of Best Fit Tie Line Leveled\nFlight Line Anomaly\nMagnitude'.format(SURVEY_NAME))
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.scatter(lvld_survey_df.LONG[lvld_survey_df.LINE_TYPE == 1],
                 lvld_survey_df.LAT[lvld_survey_df.LINE_TYPE == 1],
                 s=s,
                 c=lvld_survey_df.F[lvld_survey_df.LINE_TYPE == 1],
                 cmap=cmap)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)
```

```
[ ]: (39.335, 39.35)
```



## 13 Create Map Based on Plane of Best Fit Tie Line-Leveled Flight Lines

```
[ ]: interp_type = 'RBF'  
interp_df = pu.interp_flight_lines(anomaly_df  
    ↪lvld_survey_df[(lvld_survey_df.LAT >= MIN_MAP_LAT) & (lvld_survey_df.LAT <=  
    ↪MAX_MAP_LAT)],  
    dx          = DX,  
    dy          = DY,  
    max_terrain_msl = MAX_TERRAIN_MSL,  
    buffer      = 0,  
    interp_type = interp_type,  
    neighbors   = None,  
    skip_na_mask = True)  
  
interp_lats    = interp_df['LAT']  
interp_lons    = interp_df['LONG']  
interp_scalar  = interp_df['F']  
interp_heights = interp_df['ALT']  
interp_std     = interp_df['STD']  
  
interp_scalar_LPF = filt.lpf2(interp_scalar,  
                               MAX_SURVEY_AGL,  
                               DX,  
                               DY)
```

```

map_title = '{}\nPlane of Best Fit Tie Line Leveled Anomaly Map\nUsing {}'\
    .format(SURVEY_NAME, interp_type)

plt.figure()
plt.title(map_title)
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.pcolormesh(interp_lons,
                     interp_lats,
                     interp_scalar,
                     cmap=cmap,
                     alpha=alpha)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)

map_title = '{}\nPlane of Best Fit Tie Line Leveled Anomaly Map\nUsing {}'\
    .format(SURVEY_NAME, interp_type)

plt.figure()
plt.title(map_title)
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.pcolormesh(interp_lons,
                     interp_lats,
                     interp_scalar_LPF,
                     cmap=cmap,
                     alpha=alpha)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)

process_dict['F_CAL_IGRF_TEMPORAL_filt_LEVEL'] = lvld_survey_df.F

# Export map as a GeoTIFF
map = mu.export_map(out_dir
                    location = SURVEY_DIR,
                    date     = ' '.join(map_title.split()),
                    lats     = interp_lats,
                    lons     = interp_lons,
                    scalar   = interp_scalar_LPF,
                    heights  = interp_heights,
                    process_df = pd.DataFrame(process_dict),
                    process_app = PROCESS_APP,

```

```

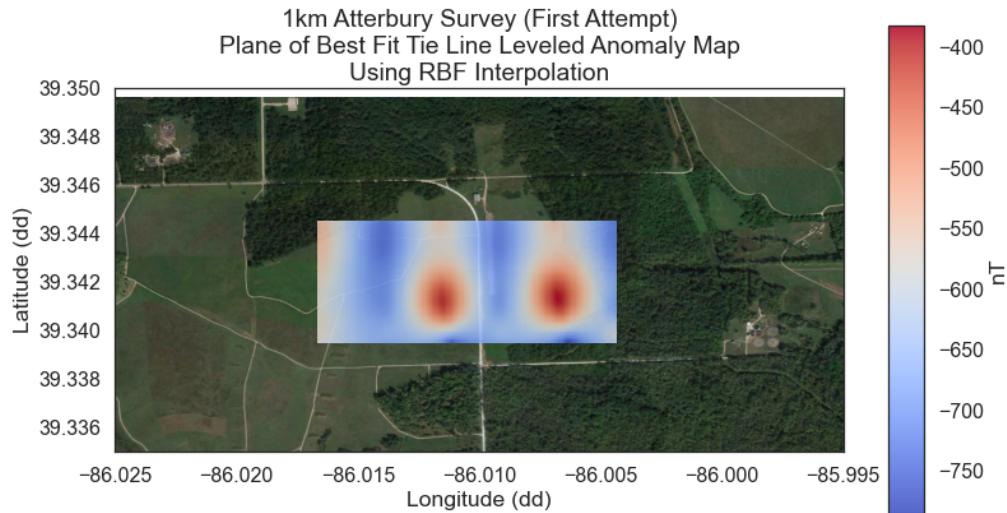
        stds          = interp_std,
        vector        = None,
        scalar_type   = SCALAR_TYPE,
        vector_type   = VECTOR_TYPE,
        scalar_var    = np.nan,
        vector_var    = np.nan,
        poc           = POC,
        flight_path   = flight_path,
        area_polys    = area_polys,
        osm_path      = None,
        level_type    = 'Plane of best fit tie line leveling',
        tl_coeff_types= TL_COEFF_TYPES,
        tl_coeffs     = TL_C,
        interp_type   = interp_type,
        final_filt_cut= FINAL_FILT_CUT,
        final_filt_order= FINAL_FILT_ORDER)

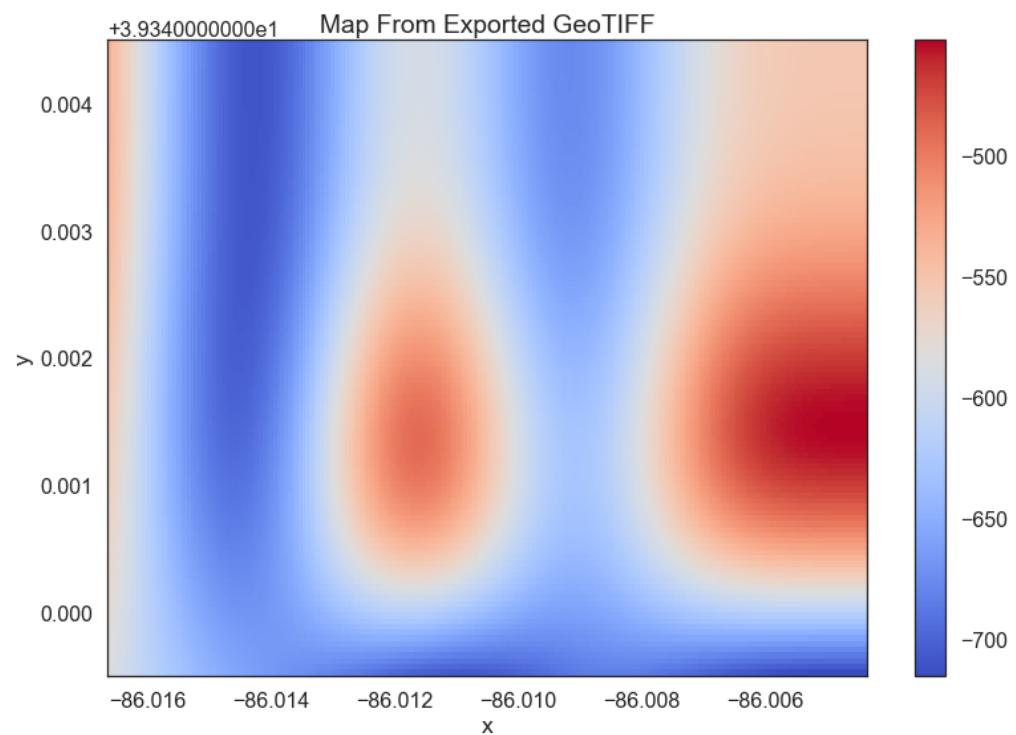
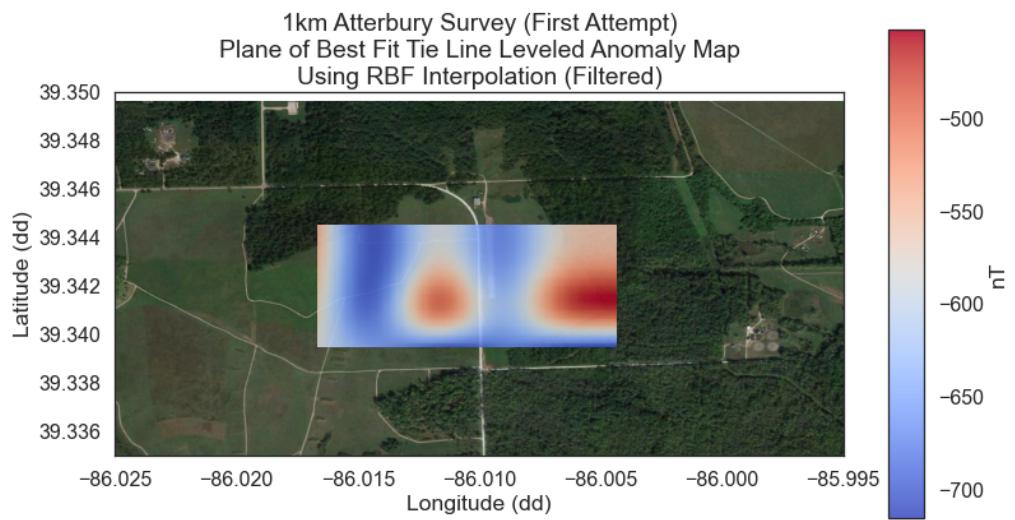
plt.figure()
map[0].plot(cmap=cm.coolwarm)
plt.title('Map From Exported GeoTIFF')

```

Interpolating survey anomaly data to map coordinates  
Running radial basis function (RBF) interpolation for all map pixels

[ ]: Text(0.5, 1.0, 'Map From Exported GeoTIFF')





## \_\_\_\_\_process\_mini\_atterbury\_survey\_2\_\_\_\_\_

December 20, 2022

```
[ ]: import sys
from os.path import join

import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
import pandas as pd
import scipy.linalg as la
from matplotlib import cm

PROJ_DIR = r'C:\Users\ltber\Desktop\files\AFIT\Research\WPAFB_Survey\mammal'
sys.path.append(PROJ_DIR)

from MAMMAL import Diurnal
from MAMMAL.MapLvl import pcaLvl
from MAMMAL.MapLvl import tieLvl
from MAMMAL.Parse import parseGSMP as pgsm
from MAMMAL.Parse import parseIM as pim
from MAMMAL.Parse import parseLCM as plcm
from MAMMAL.Utils import coordinateUtils as cu
from MAMMAL.Utils import Filters as filt
from MAMMAL.Utils import mapUtils as mu
from MAMMAL.Utils import ProcessingUtils as pu
from MAMMAL.VehicleCal import magUtilsTL as magtl

%matplotlib inline
plt.rcParams["figure.figsize"] = (30, 15) # (w, h)
plt.style.use(['seaborn-poster', 'seaborn-white'])
pd.set_option('mode.chained_assignment', None)

shrink = 0.8
aspect = 20 * 0.7
xlims = [-86.025, -85.995]
ylims = [39.335, 39.35]
cmap = cm.coolwarm
s = 10
```

```

alpha = 0.85

GMAP_EXTENT = [-86.054121, -85.981633, 39.319868, 39.349562]

SURVEY_DIR = r'D:\__atterbury_data__\mini_atterbury_survey_2_'
GMAP_FNAME = r'D:\__atterbury_data__\atterbury.jpg'
REF_FNAME = r'D:\__atterbury_data__\mini_atterbury_survey_2_\Ground_Reference.
    ↪CSV'
LOG_FNAME = r'D:
    ↪\__atterbury_data__\mini_atterbury_survey_2_\__mini_atterbury_survey_2__.
    ↪CSV'
KML_FNAME = r'Mini_Atterbury_Survey_2.kml'

SURVEY_NAME = '1km Atterbury Survey (Second Attempt)'

MAX_TERRAIN_MSL = 230 # (m)
MAX_SURVEY_AGL = 400 # (m)
MAX_SURVEY_CRUISE = 30 # (m/s)

MAX_EXPECTED_FREQ = MAX_SURVEY_CRUISE / MAX_SURVEY_AGL # (hz)

TL_C = np.array([-1.86687725e+01, 1.33975396e+02, -1.80762945e+02, 1.
    ↪69023832e-01,
                    -3.92262356e-03, -1.84382741e-03, 1.71830230e-01, -1.
    ↪61173781e-04,
                    1.72575427e-01, -4.31927864e-04, -8.21512835e-05, -4.
    ↪37609432e-05,
                    -1.06838978e-04, -1.22444017e-04, -2.76294434e-04, -8.
    ↪51727772e-05,
                    3.16374022e-05, -2.77441572e-05])

TL_TERMS = magtl.DEFAULT_TL_TERMS
TL_COEFF_TYPES = ['Permanent', 'Induced', 'Eddy']

SCALAR_TYPE = 'Geometrics MFAM optically pumped caesium scalar magnetometer
    ↪with 2 sensor heads'
VECTOR_TYPE = 'TwinLeaf VMR anisotropic magnetoresistive vector magnetometer'

FINAL_filt_CUT = MAX_SURVEY_AGL
FINAL_filt_ORDER = 6

DX = 5 # Map pixel width (m)
DY = 5 # Map pixel height (m)

POC = '''Autonomy and Navigation Technology Center
        Air Force Institute of Technology

```

```

Graduate School of Engineering and Management
2950 Hobson Way, Bldg 646, Rm 205
Wright Patterson AFB, Ohio 45433

Telephone: (937) 255-3636 Ext. 4671'''

PROCESS_APP = 'MAMMAL 0.0.1'

```

## 1 Load Survey Data

```
[ ]: img = mpimg.imread(GMAP_FNAME)

ref_df = pd.read_csv(REF_FNAME, parse_dates=['datetime'])

log_df      = pd.read_csv(LOG_FNAME, parse_dates=['datetime'])
datetimes   = log_df.datetime
timestamps  = np.array(log_df.epoch_sec)
sample_rate = 1.0 / np.diff(timestamps).mean()
```

## 2 Interpolate Reference Data

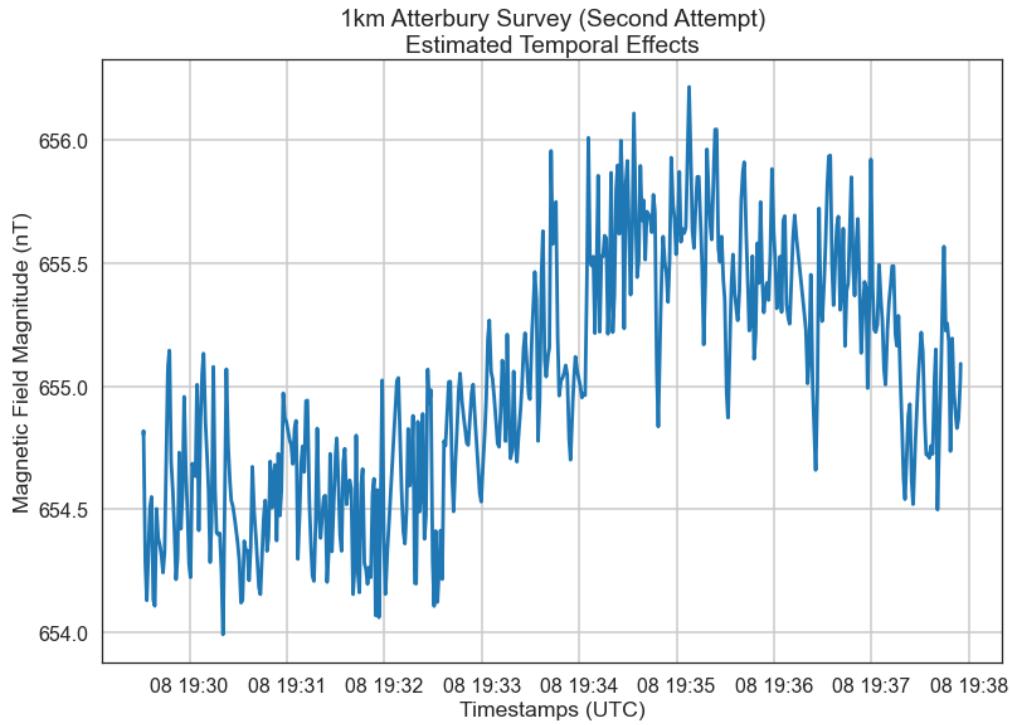
```
[ ]: # Despike reference data
span  = 10
delta = 0.5

ref_despiked = pu.remove_outliers(ref_df.F, pu.ewma_fb(ref_df.F, span), delta)

ref_df.F = ref_despiked
ref_df.F = ref_df.F.interpolate()

# Interpolate reference data
_, ref_mag = Diurnal.interp_reference_df(df           = ref_df,
                                           timestamps = timestamps,
                                           survey_lon = log_df.LONG.mean(),
                                           subtract_core = True)

plt.figure()
plt.title('{}\\nEstimated Temporal Effects'.format(SURVEY_NAME))
plt.xlabel('Timestamps (UTC)')
plt.ylabel('Magnetic Field Magnitude (nT)')
plt.plot(log_df.datetime, ref_mag)
plt.grid()
```



### 3 Determine When Each Sensor Head was Valid During the Survey

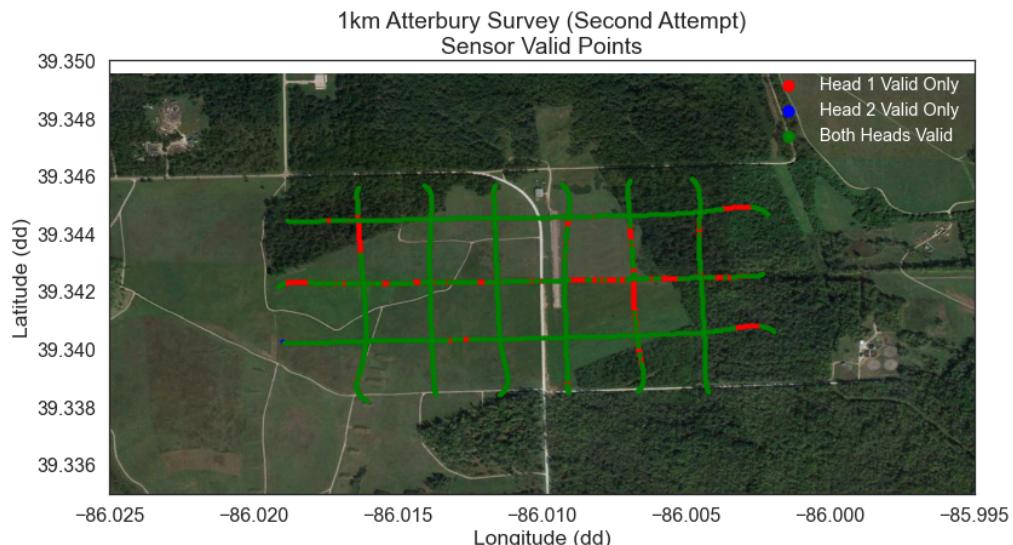
```
[ ]: plt.figure()
plt.title('{}\\nSensor Valid Points'.format(SURVEY_NAME))
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
plt.scatter(log_df.LONG[(log_df.SCALAR_1_VALID == 1) & (log_df.SCALAR_2_VALID == 1) & (log_df.LINE_TYPE != 0)],
            log_df.LAT[(log_df.SCALAR_1_VALID == 1) & (log_df.SCALAR_2_VALID == 1) & (log_df.LINE_TYPE != 0)],
            s=s,
            c='r',
            label='Head 1 Valid Only')
plt.scatter(log_df.LONG[(log_df.SCALAR_1_VALID == 0) & (log_df.SCALAR_2_VALID == 1) & (log_df.LINE_TYPE != 0)],
            log_df.LAT[(log_df.SCALAR_1_VALID == 0) & (log_df.SCALAR_2_VALID == 1) & (log_df.LINE_TYPE != 0)],
            s=s,
```

```

        c='b',
        label='Head 2 Valid Only')
plt.scatter(log_df.LONG[(log_df.SCALAR_1_VALID == 1) & (log_df.SCALAR_2_VALID == 1) & (log_df.LINE_TYPE != 0)],
            log_df.LAT[(log_df.SCALAR_1_VALID == 1) & (log_df.SCALAR_2_VALID == 1) & (log_df.LINE_TYPE != 0)],
            s=s,
            c='g',
            label='Both Heads Valid')
plt.legend(labelcolor='white', fontsize='x-large', markerscale=3)
plt.xlim(xlims)
plt.ylim(ylims)

```

[ ]: (39.335, 39.35)



## 4 Apply Tolles Lawson Calibration

```

[ ]: # Compile vector data
b_vector = np.hstack([np.array(log_df.X)[:, np.newaxis],
                      np.array(log_df.Y)[:, np.newaxis],
                      np.array(log_df.Z)[:, np.newaxis]])
dcs = b_vector / la.norm(b_vector, axis=1)[:, np.newaxis]
dc_x = dcs[:, 0]
dc_y = dcs[:, 1]
dc_z = dcs[:, 2]

```

```

# Calibrate sensor head
f = (log_df.SCALAR_1_LPF + log_df.SCALAR_2_LPF) / 2.0

body_effects_scalar = magtl.tlc_compensation(vector = b_vector,
                                              tlc      = TL_C,
                                              terms   = TL_TERMS)

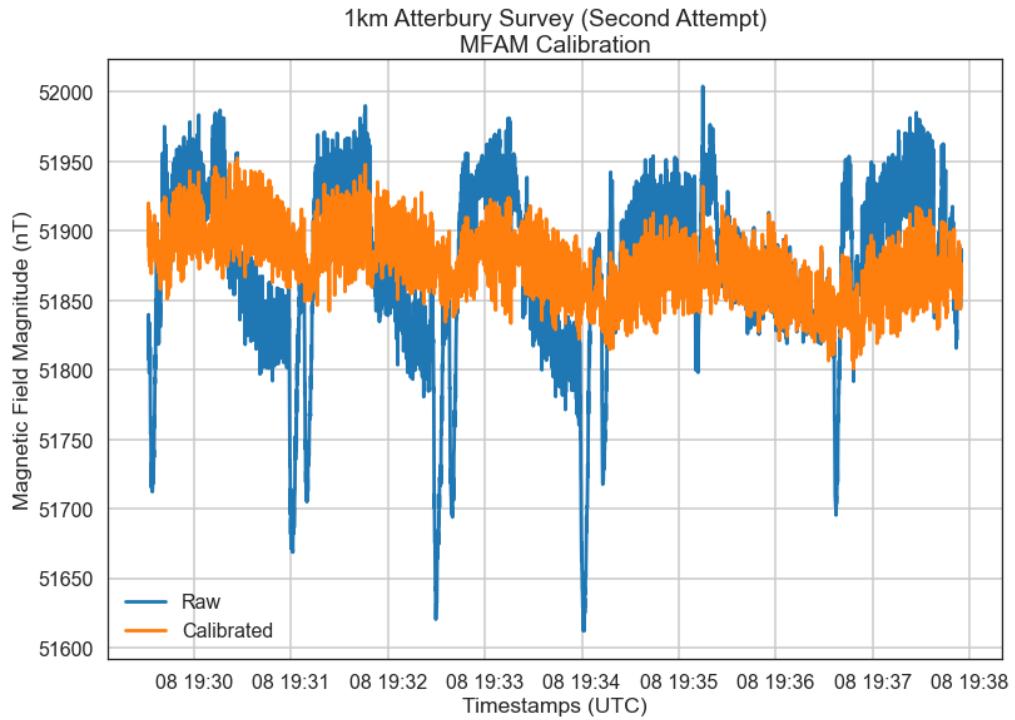
f_cal = f - body_effects_scalar

f_cal += (f.mean() - f_cal.mean())

# Update dataframe with calibrated data
log_df['SCALAR_CAL'] = f_cal
log_df['F']           = log_df.SCALAR_CAL

plt.figure()
plt.title('{}\\nMFAM Calibration'.format(SURVEY_NAME))
plt.xlabel('Timestamps (UTC)')
plt.ylabel('Magnetic Field Magnitude (nT)')
plt.plot(log_df.datetime, f,      label='Raw')
plt.plot(log_df.datetime, f_cal, label='Calibrated')
plt.legend()
plt.grid()

```



## 5 Find Estimated Magnetic Anomaly Values

```
[ ]: f_cal_igrf = f_cal - log_df.IGRF_F
f_cal_igrf_temporal = f_cal_igrf - ref_mag
f_cal_igrf_temporal_filt = filt.lpf(f_cal_igrf_temporal, MAX_EXPECTED_FREQ,
                                     sample_rate)

process_dict = {'TIMESTAMP': timestamps,
                'LAT': log_df.LAT,
                'LONG': log_df.LONG,
                'ALT': log_df.ALT,
                'DC_X': dc_x,
                'DC_Y': dc_y,
                'DC_Z': dc_z,
                'F': f,
                'F_CAL': f_cal,
                'F_CAL_IGRF': f_cal_igrf,
                'F_CAL_IGRF_TEMPORAL': f_cal_igrf_temporal,
                'F_CAL_IGRF_TEMPORAL_FILTER': f_cal_igrf_temporal_filt}

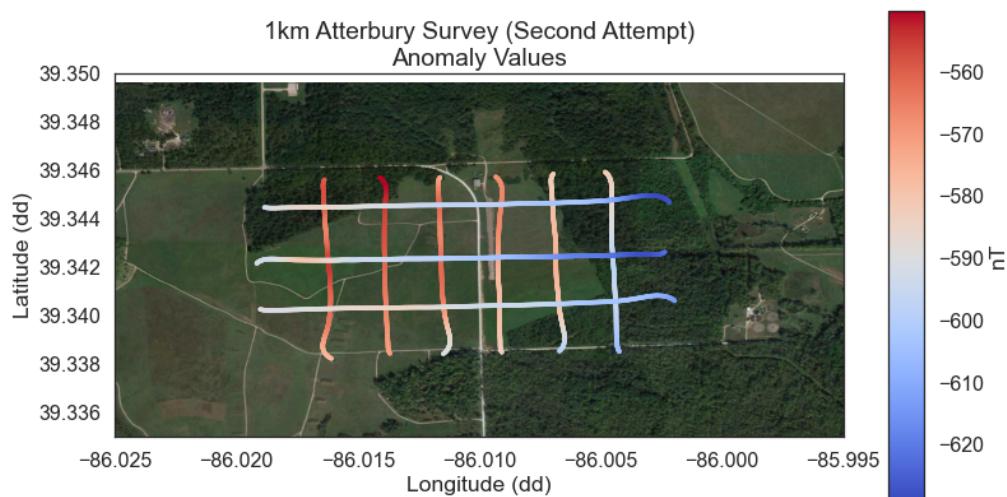
log_df.F = f_cal_igrf_temporal_filt
```

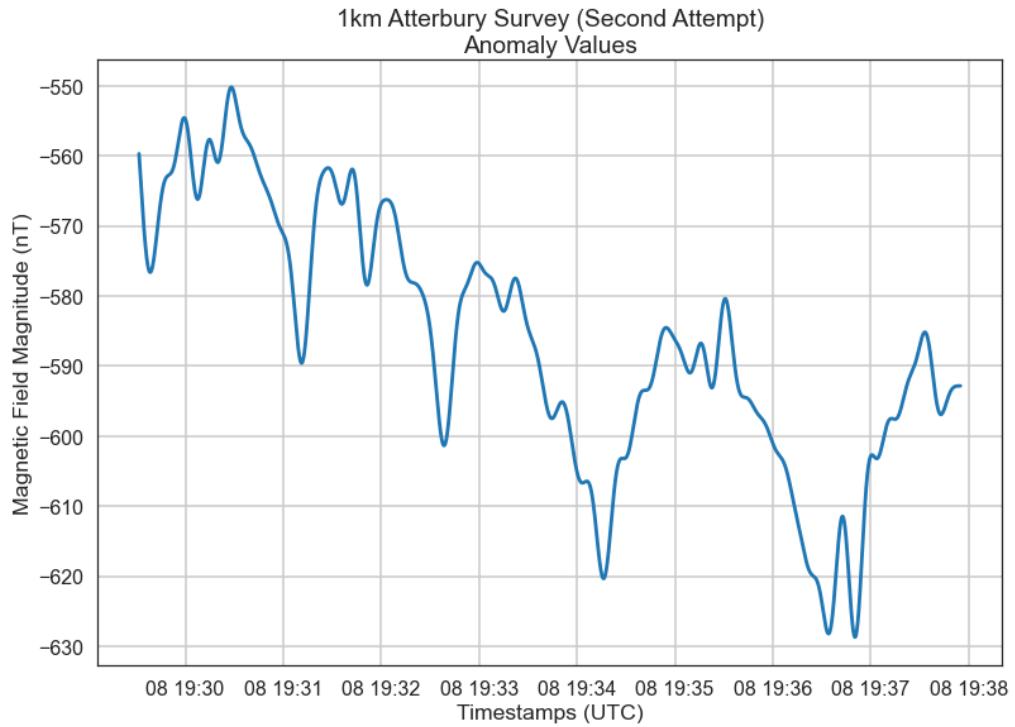
```

plt.figure()
plt.title('{}\\nAnomaly Values'.format(SURVEY_NAME))
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.scatter(log_df.LOGN[log_df.LINE_TYPE != 0],
                  log_df.LAT[log_df.LINE_TYPE != 0],
                  s=s,
                  c=log_df.F[log_df.LINE_TYPE != 0],
                  cmap=cmap)
plt.xlim(xlims)
plt.ylim(ylims)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)

plt.figure()
plt.title('{}\\nAnomaly Values'.format(SURVEY_NAME))
plt.xlabel('Timestamps (UTC)')
plt.ylabel('Magnetic Field Magnitude (nT)')
plt.plot(log_df.datetime, log_df.F)
plt.grid()

```





## 6 Crop Map Extent

```
[ ]: MIN_MAP_LAT = 39.3395
      MAX_MAP_LAT = 39.3445
```

## 7 Create Maps Based on Non-Leveled Flight Lines

```
[ ]: interp_type = 'RBF'
interp_df = pu.interp_flight_lines(anomaly_df
    ↪MIN_MAP_LAT) & (log_df.LAT <= MAX_MAP_LAT),
          dx           = DX,
          dy           = DY,
          max_terrain_msl = MAX_TERRAIN_MSL,
          buffer        = 0,
          interp_type   = interp_type,
          neighbors     = None,
          skip_na_mask  = True)

interp_lats   = interp_df['LAT']
interp_lons   = interp_df['LONG']
```

```

interp_scalar = interp_df['F']
interp_heights = interp_df['ALT']
interp_std     = interp_df['STD']

interp_scalar_LPF = filt.lpf2(interp_scalar,
                               MAX_SURVEY_AGL,
                               DX,
                               DY)

map_title = '{}\nNon-Leveled Anomaly Map\nUsing {} Interpolation'.
            format(SURVEY_NAME, interp_type)

plt.figure()
plt.title(map_title)
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.pcolormesh(interp_lons,
                     interp_lats,
                     interp_scalar,
                     cmap=cmap,
                     alpha=alpha)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)

map_title = '{}\nNon-Leveled Anomaly Map\nUsing {} Interpolation (Filtered)'.
            format(SURVEY_NAME, interp_type)

plt.figure()
plt.title(map_title)
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.pcolormesh(interp_lons,
                     interp_lats,
                     interp_scalar_LPF,
                     cmap=cmap,
                     alpha=alpha)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)

extent = [interp_lons.min(), interp_lats.min(), interp_lons.max(), interp_lats.
          max()]
xmin, ymin, xmax, ymax = extent

```

```

flight_path = np.hstack([np.array(log_df.LAT)[:, np.newaxis],
                        np.array(log_df.LONG)[:, np.newaxis],
                        np.array(log_df.ALT)[:, np.newaxis],
                        np.array(log_df.epoch_sec)[:, np.newaxis]]))

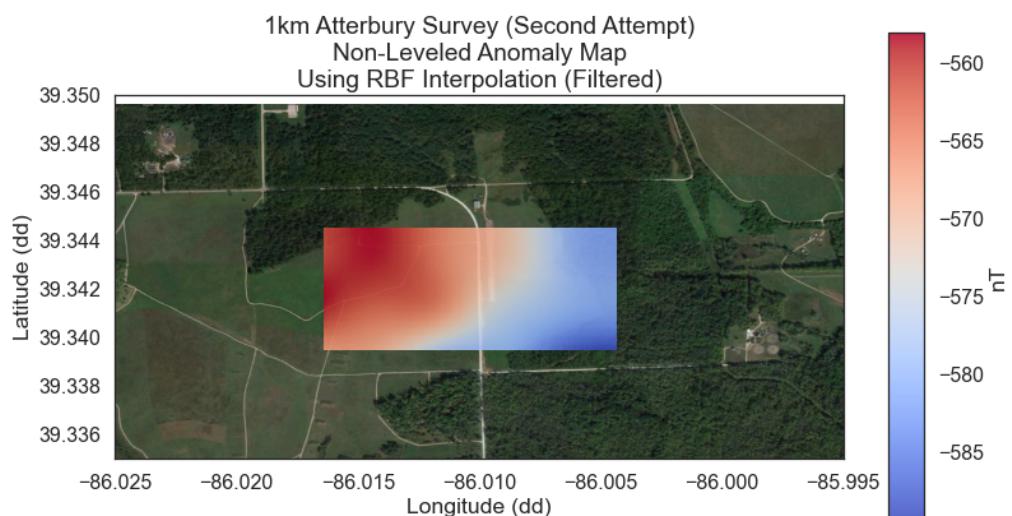
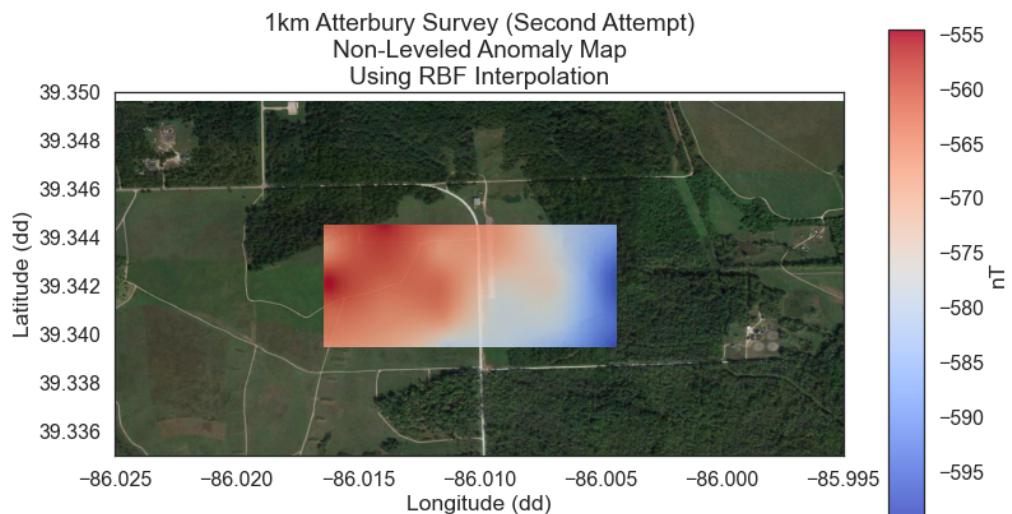
area_polys = [{'NAME': 'Survey Area',
               'FL_DIRFL_DISTTL_DIRTL_DISTLATLONGALT# Export map as a GeoTIFF
map = mu.export_map(out_dir
                     location = SURVEY_DIR,
                     date = ' '.join(map_title.split()),
                     lats = interp_lats,
                     lons = interp_lons,
                     scalar = interp_scalar_LPF,
                     heights = interp_heights,
                     process_df = pd.DataFrame(process_dict),
                     process_app = PROCESS_APP,
                     stds = interp_std,
                     vector = None,
                     scalar_type = SCALAR_TYPE,
                     vector_type = VECTOR_TYPE,
                     scalar_var = np.nan,
                     vector_var = np.nan,
                     poc = POC,
                     flight_path = flight_path,
                     area_polys = area_polys,
                     osm_path = None,
                     level_type = 'No leveling',
                     tl_coeff_types = TL_COEFF_TYPES,
                     tl_coeffs = TL_C,
                     interp_type = interp_type,
                     final_filt_cut = FINAL_FILT_CUT,
                     final_filt_order = FINAL_FILT_ORDER)

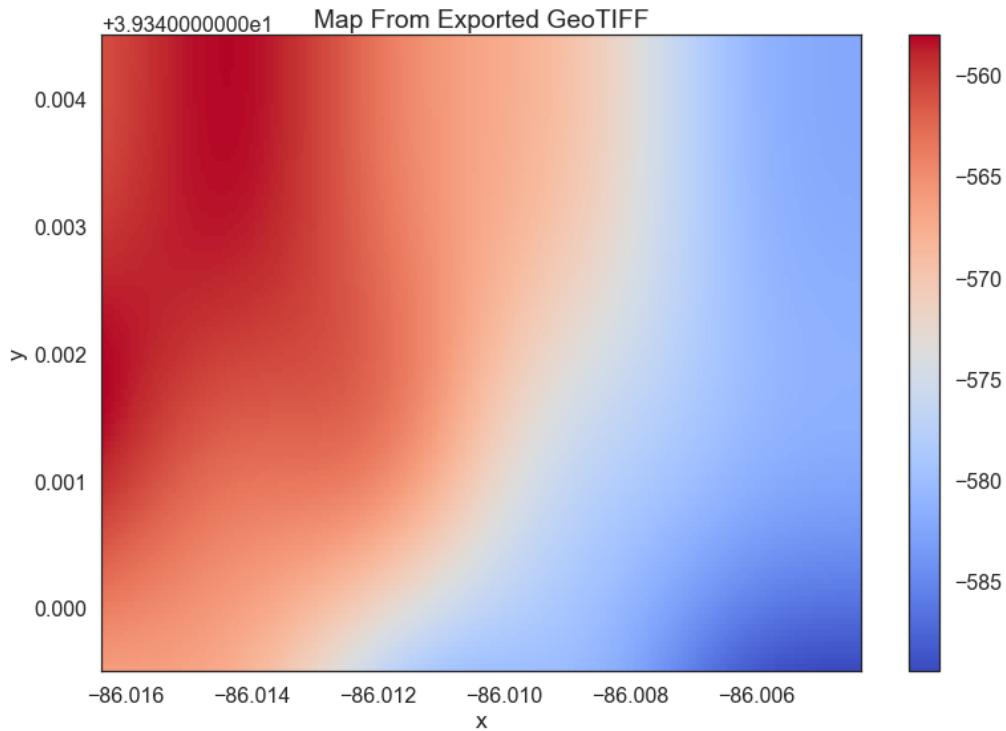
plt.figure()
map[0].plot(cmap=cm.coolwarm)
plt.title('Map From Exported GeoTIFF')

```

Interpolating survey anomaly data to map coordinates  
Running radial basis function (RBF) interpolation for all map pixels

[ ]: Text(0.5, 1.0, 'Map From Exported GeoTIFF')



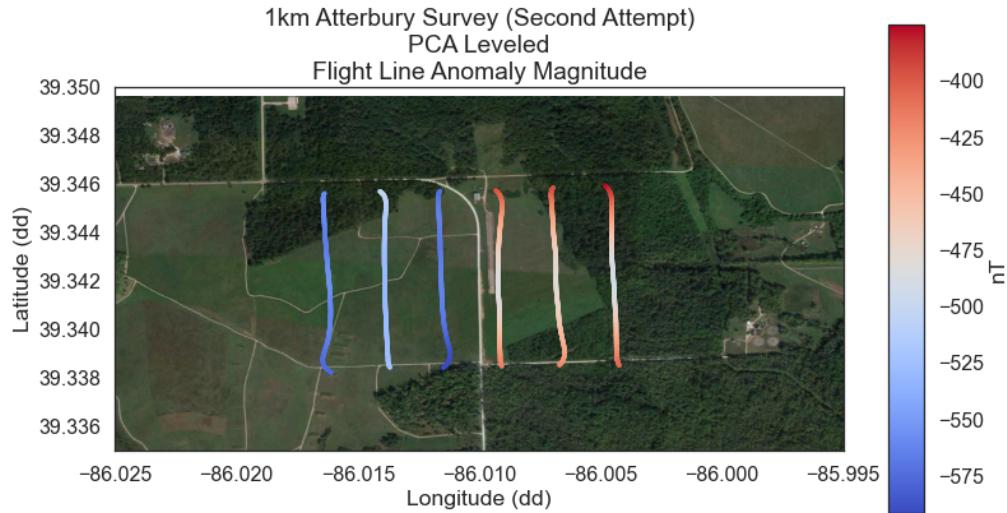


## 8 Apply PCA-Based Flight Line Leveling

```
[ ]: lvld_survey_df = pcaLvl.pca_lvl(survey_df = log_df,
                                      num_ptls = 2,
                                      ptl_locs = np.array([0.25, 0.75]))

plt.figure()
plt.title('{}\\nPCA Leveled\\nFlight Line Anomaly Magnitude'.format(SURVEY_NAME))
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.scatter(lvld_survey_df.LONG[lvld_survey_df.LINE_TYPE == 1],
                 lvld_survey_df.LAT[lvld_survey_df.LINE_TYPE == 1],
                 s=s,
                 c=lvld_survey_df.F[lvld_survey_df.LINE_TYPE == 1],
                 cmap=cmap)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)
```

```
[ ]: (39.335, 39.35)
```



## 9 Create Maps Based on PCA-Leveled Flight Lines

```
[ ]: interp_type = 'RBF'  
interp_df = pu.interp_flight_lines(anomaly_df  
    ↪ lvld_survey_df[(lvld_survey_df.LAT >= MIN_MAP_LAT) & (lvld_survey_df.LAT <=  
    ↪ MAX_MAP_LAT)],  
    dx = DX,  
    dy = DY,  
    max_terrain_msl = MAX_TERRAIN_MSL,  
    buffer = 0,  
    interp_type = interp_type,  
    neighbors = None,  
    skip_na_mask = True)  
  
interp_lats = interp_df['LAT']  
interp_lons = interp_df['LONG']  
interp_scalar = interp_df['F']  
interp_heights = interp_df['ALT']  
interp_std = interp_df['STD']  
  
interp_scalar_LPF = filt.lpf2(interp_scalar,  
    MAX_SURVEY_AGL,  
    DX,  
    DY)
```

```

map_title = '{}\nPCA Leveled Anomaly Map\nUsing {} Interpolation'.
    .format(SURVEY_NAME, interp_type)

plt.figure()
plt.title(map_title)
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.pcolormesh(interp_lons,
                     interp_lats,
                     interp_scalar,
                     cmap=cmap,
                     alpha=alpha)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)

map_title = '{}\nPCA Leveled Anomaly Map\nUsing {} Interpolation (Filtered)'.
    .format(SURVEY_NAME, interp_type)

plt.figure()
plt.title(map_title)
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.pcolormesh(interp_lons,
                     interp_lats,
                     interp_scalar_LPF,
                     cmap=cmap,
                     alpha=alpha)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)

process_dict['F_CAL_IGRF_TEMPORAL_FILT_LEVEL'] = lvld_survey_df.F

# Export map as a GeoTIFF
map = mu.export_map(out_dir
                    location = SURVEY_DIR,
                    date = ' '.join(map_title.split()),
                    lats = interp_lats,
                    lons = interp_lons,
                    scalar = interp_scalar_LPF,
                    heights = interp_heights,
                    process_df = pd.DataFrame(process_dict),
                    process_app = PROCESS_APP,
                    stds = interp_std,

```

```

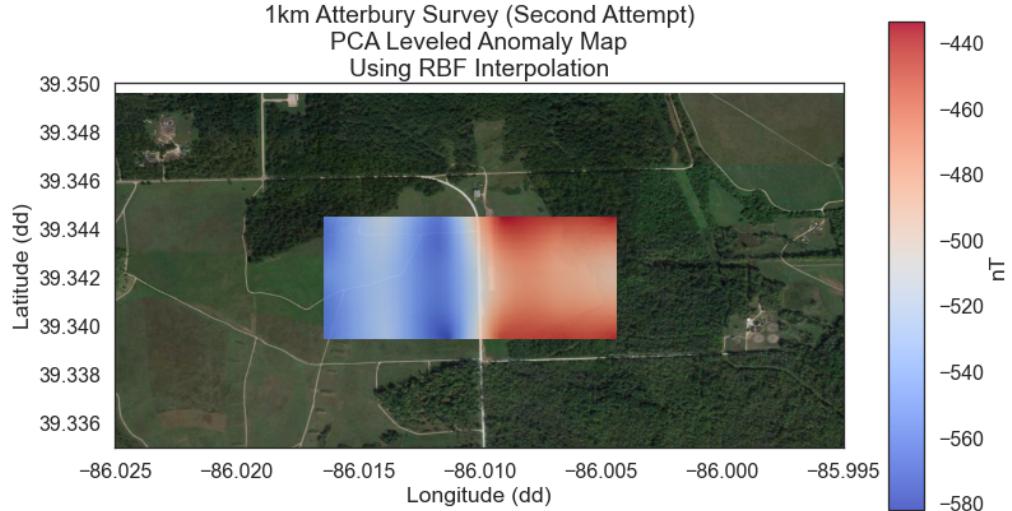
vector           = None,
scalar_type     = SCALAR_TYPE,
vector_type     = VECTOR_TYPE,
scalar_var      = np.nan,
vector_var      = np.nan,
poc              = POC,
flight_path     = flight_path,
area_polys      = area_polys,
osm_path         = None,
level_type       = 'PCA pseudo tie line leveling',
tl_coeff_types  = TL_COEFF_TYPES,
tl_coeffs        = TL_C,
interp_type      = interp_type,
final_filt_cut   = FINAL_FILT_CUT,
final_filt_order = FINAL_FILT_ORDER)

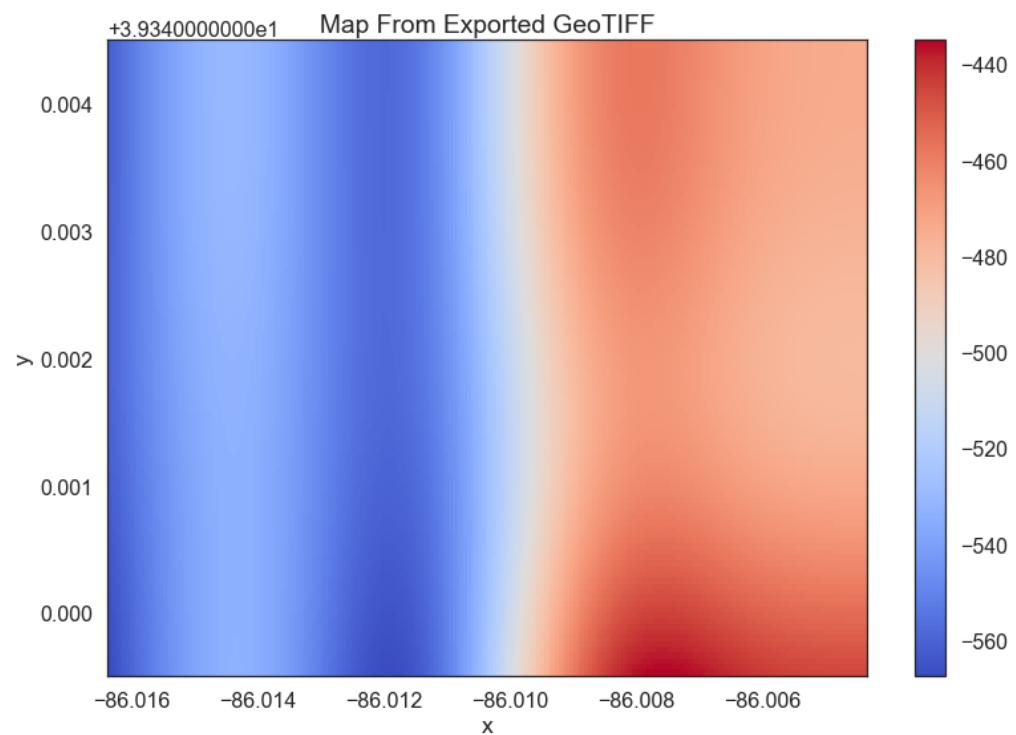
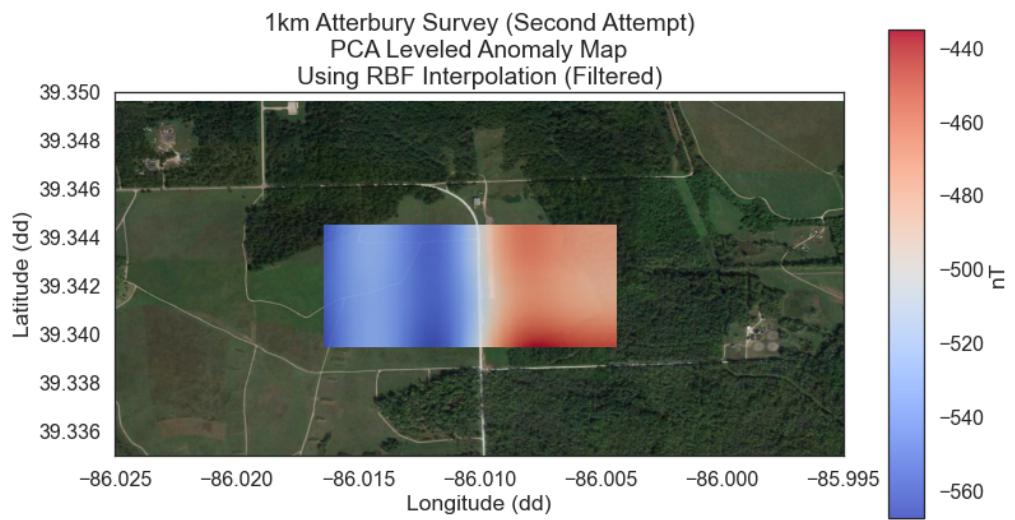
plt.figure()
map[0].plot(cmap=cm.coolwarm)
plt.title('Map From Exported GeoTIFF')

```

Interpolating survey anomaly data to map coordinates  
Running radial basis function (RBF) interpolation for all map pixels

[ ]: Text(0.5, 1.0, 'Map From Exported GeoTIFF')



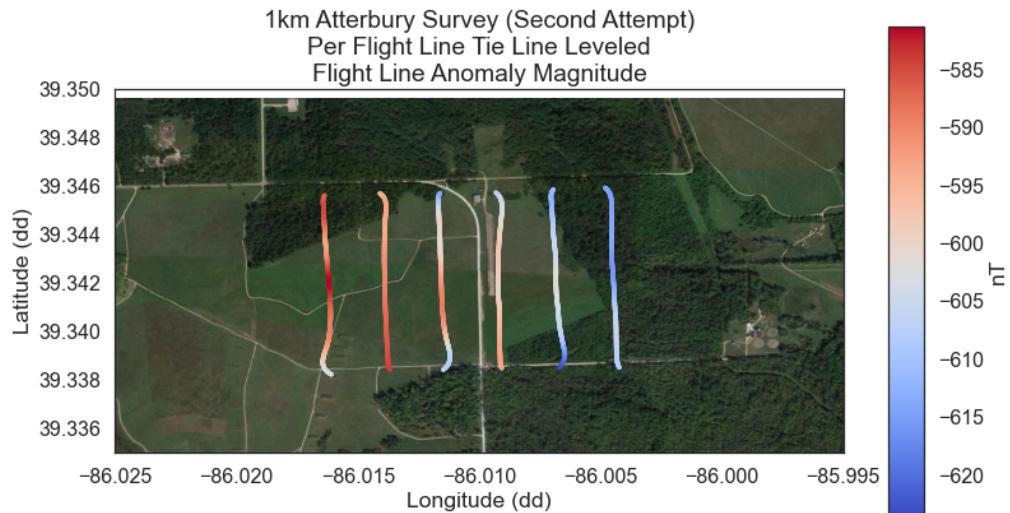


## 10 Apply Per Flight Line Tie Line-Based Flight Line Leveling

```
[ ]: lvld_survey_df = tieLvl.tie_lvl(survey_df = log_df,
                                     approach = 'lobf')

plt.figure()
plt.title('{}\\nPer Flight Line Tie Line Leveled\\nFlight Line Anomaly Magnitude'.
          format(SURVEY_NAME))
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.scatter(lvld_survey_df.LONG[lvld_survey_df.LINE_TYPE == 1],
                 lvld_survey_df.LAT[lvld_survey_df.LINE_TYPE == 1],
                 s=s,
                 c=lvld_survey_df.F[lvld_survey_df.LINE_TYPE == 1],
                 cmap=cmap)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)
```

```
[ ]: (39.335, 39.35)
```



## 11 Create Map Based on Per Flight Line Tie Line-Leveled Flight Lines

```
[ ]: interp_type = 'RBF'
interp_df = pu.interp_flight_lines(anomaly_df
                                   = lvld_survey_df[(lvld_survey_df.LAT >= MIN_MAP_LAT) & (lvld_survey_df.LAT <= MAX_MAP_LAT)],
                                   dx = DX,
                                   dy = DY,
                                   max_terrain_msl = MAX_TERRAIN_MSL,
                                   buffer = 0,
                                   interp_type = interp_type,
                                   neighbors = None,
                                   skip_na_mask = True)

interp_lats = interp_df['LAT']
interp_lons = interp_df['LONG']
interp_scalar = interp_df['F']
interp_heights = interp_df['ALT']
interp_std = interp_df['STD']

interp_scalar_LPF = filt.lpf2(interp_scalar,
                               MAX_SURVEY_AGL,
                               DX,
                               DY)

map_title = '{}\nPer Flight Line Tie Line Leveled Anomaly Map\nUsing {}'
            .format(SURVEY_NAME, interp_type)

plt.figure()
plt.title(map_title)
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.pcolormesh(interp_lons,
                     interp_lats,
                     interp_scalar,
                     cmap=cmap,
                     alpha=alpha)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)

map_title = '{}\nPer Flight Line Tie Line Leveled Anomaly Map\nUsing {}'
            .format(SURVEY_NAME, interp_type)

plt.figure()
```

```

plt.title(map_title)
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.pcolormesh(interp_lons,
                     interp_lats,
                     interp_scalar_LPF,
                     cmap=cmap,
                     alpha=alpha)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)

process_dict['F_CAL_IGRF_TEMPORAL_filt_LEVEL'] = lvld_survey_df.F

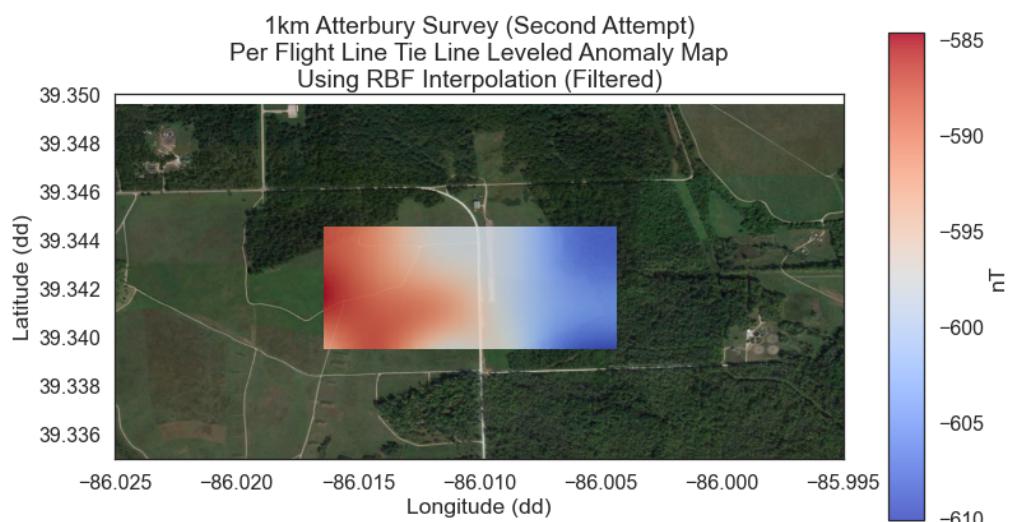
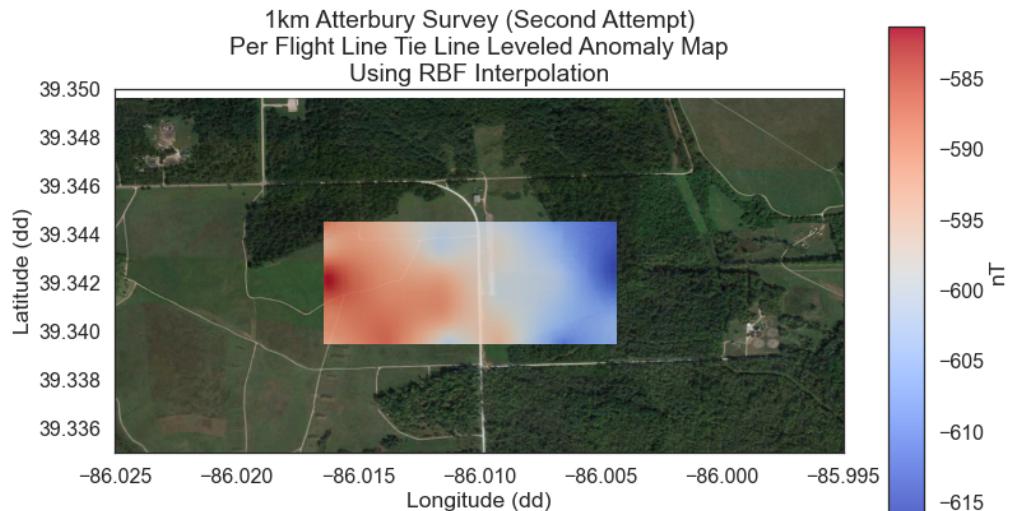
# Export map as a GeoTIFF
map = mu.export_map(out_dir
                     location = SURVEY_DIR,
                     date = ' '.join(map_title.split()),
                     lats = interp_lats,
                     lons = interp_lons,
                     scalar = interp_scalar_LPF,
                     heights = interp_heights,
                     process_df = pd.DataFrame(process_dict),
                     process_app = PROCESS_APP,
                     stds = interp_std,
                     vector = None,
                     scalar_type = SCALAR_TYPE,
                     vector_type = VECTOR_TYPE,
                     scalar_var = np.nan,
                     vector_var = np.nan,
                     poc = POC,
                     flight_path = flight_path,
                     area_polys = area_polys,
                     osm_path = None,
                     level_type = 'Per flight line tie line leveling',
                     tl_coeff_types = TL_COEFF_TYPES,
                     tl_coeffs = TL_C,
                     interp_type = interp_type,
                     final_filt_cut = FINAL_FILT_CUT,
                     final_filt_order = FINAL_FILT_ORDER)

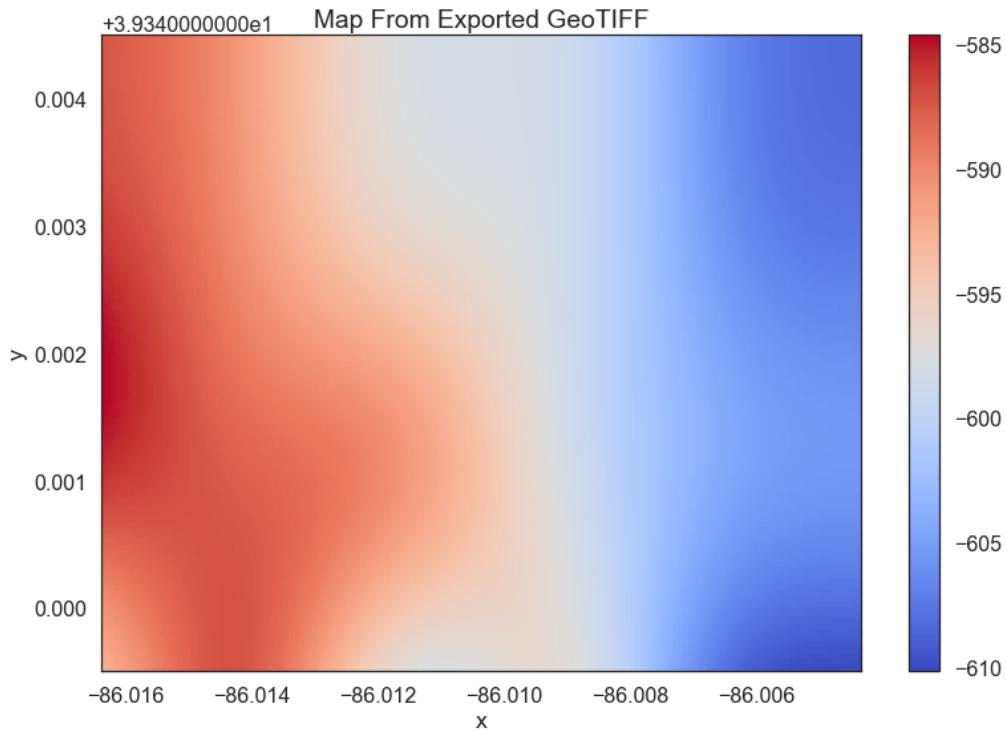
plt.figure()
map[0].plot(cmap=cm.coolwarm)
plt.title('Map From Exported GeoTIFF')

```

Interpolating survey anomaly data to map coordinates  
Running radial basis function (RBF) interpolation for all map pixels

[ ]: Text(0.5, 1.0, 'Map From Exported GeoTIFF')



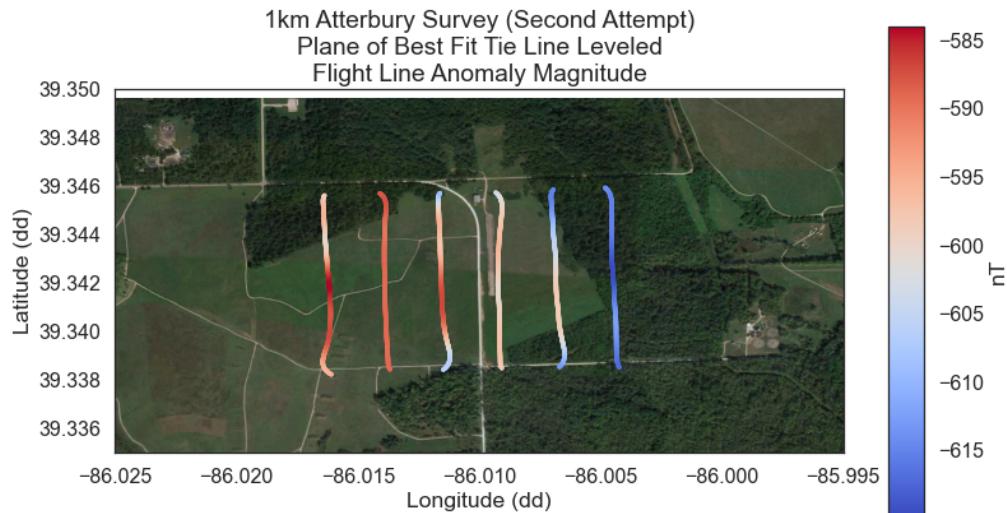


## 12 Apply Plane of Best Fit Tie Line-Based Flight Line Leveling

```
[ ]: lvld_survey_df = tieLvl.tie_lvl(survey_df = log_df,
                                     approach = 'lsq')

plt.figure()
plt.title('{} \nPlane of Best Fit Tie Line Leveled\nFlight Line Anomaly\nMagnitude'.format(SURVEY_NAME))
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.scatter(lvld_survey_df.LONG[lvld_survey_df.LINE_TYPE == 1],
                 lvld_survey_df.LAT[lvld_survey_df.LINE_TYPE == 1],
                 s=s,
                 c=lvld_survey_df.F[lvld_survey_df.LINE_TYPE == 1],
                 cmap=cmap)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)
```

```
[ ]: (39.335, 39.35)
```



## 13 Create Map Based on Plane of Best Fit Tie Line-Leveled Flight Lines

```
[ ]: interp_type = 'RBF'
interp_df = pu.interp_flight_lines(anomaly_df
    ↪ lvld_survey_df[(lvld_survey_df.LAT >= MIN_MAP_LAT) & (lvld_survey_df.LAT <=
    ↪ MAX_MAP_LAT)],
    dx          = DX,
    dy          = DY,
    max_terrain_msl = MAX_TERRAIN_MSL,
    buffer      = 0,
    interp_type = interp_type,
    neighbors   = None,
    skip_na_mask = True)

interp_lats   = interp_df['LAT']
interp_lons   = interp_df['LONG']
interp_scalar = interp_df['F']
interp_heights = interp_df['ALT']
interp_std    = interp_df['STD']

interp_scalar_LPF = filt.lpf2(interp_scalar,
                               MAX_SURVEY_AGL,
                               DX,
                               DY)
```

```

map_title = '{}\nPlane of Best Fit Tie Line Leveled Anomaly Map\nUsing {}'\
    .format(SURVEY_NAME, interp_type)

plt.figure()
plt.title(map_title)
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.pcolormesh(interp_lons,
                     interp_lats,
                     interp_scalar,
                     cmap=cmap,
                     alpha=alpha)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)

map_title = '{}\nPlane of Best Fit Tie Line Leveled Anomaly Map\nUsing {}'\
    .format(SURVEY_NAME, interp_type)

plt.figure()
plt.title(map_title)
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.pcolormesh(interp_lons,
                     interp_lats,
                     interp_scalar_LPF,
                     cmap=cmap,
                     alpha=alpha)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)

process_dict['F_CAL_IGRF_TEMPORAL_filt_LEVEL'] = lvld_survey_df.F

# Export map as a GeoTIFF
map = mu.export_map(out_dir
                    location = SURVEY_DIR,
                    date     = ' '.join(map_title.split()),
                    lats     = interp_lats,
                    lons     = interp_lons,
                    scalar   = interp_scalar_LPF,
                    heights  = interp_heights,
                    process_df = pd.DataFrame(process_dict),
                    process_app = PROCESS_APP,

```

```

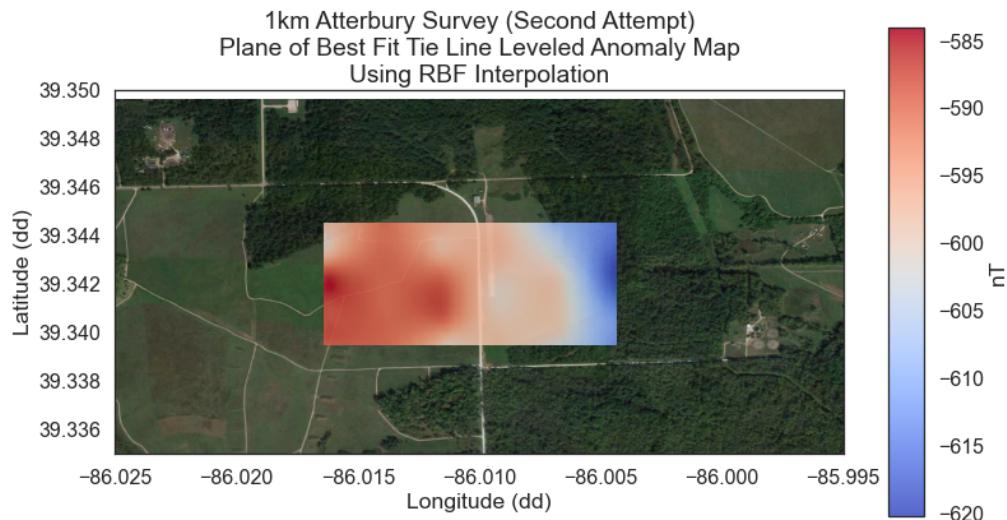
        stds          = interp_std,
        vector        = None,
        scalar_type   = SCALAR_TYPE,
        vector_type   = VECTOR_TYPE,
        scalar_var    = np.nan,
        vector_var    = np.nan,
        poc           = POC,
        flight_path   = flight_path,
        area_polys    = area_polys,
        osm_path      = None,
        level_type    = 'Plane of best fit tie line leveling',
        tl_coeff_types= TL_COEFF_TYPES,
        tl_coeffs     = TL_C,
        interp_type   = interp_type,
        final_filt_cut= FINAL_FILT_CUT,
        final_filt_order= FINAL_FILT_ORDER)

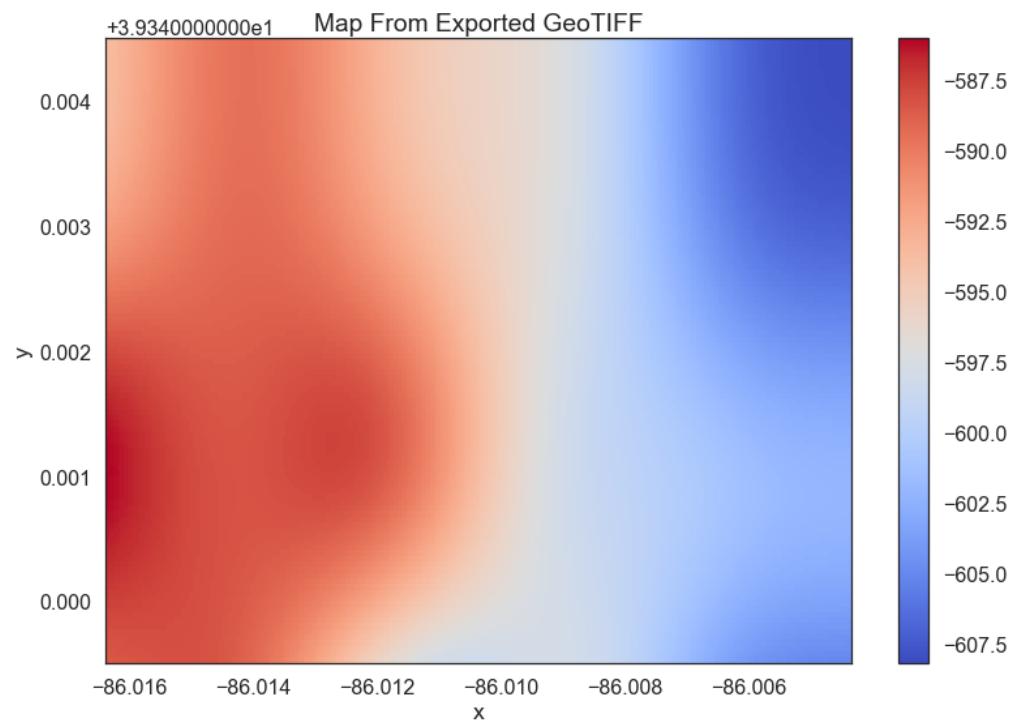
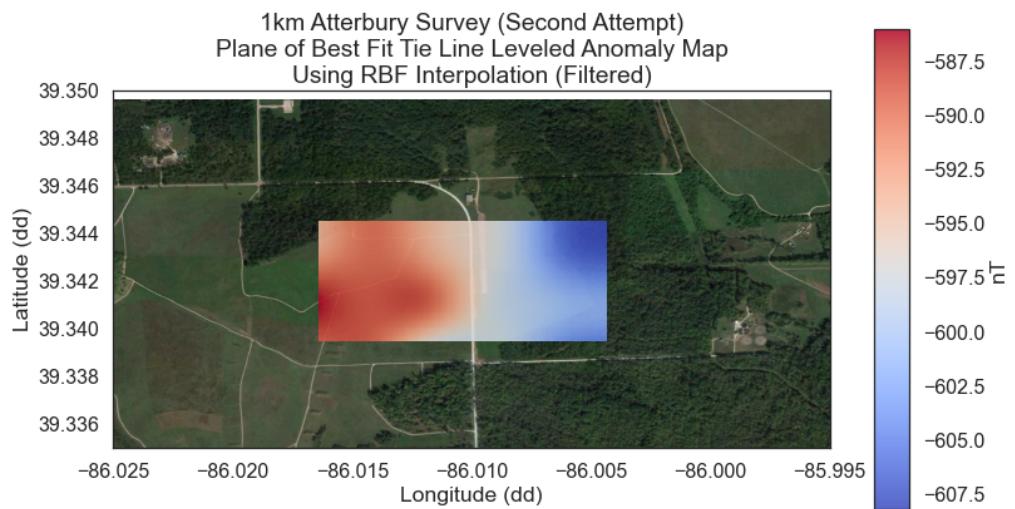
plt.figure()
map[0].plot(cmap=cm.coolwarm)
plt.title('Map From Exported GeoTIFF')

```

Interpolating survey anomaly data to map coordinates  
Running radial basis function (RBF) interpolation for all map pixels

[ ]: Text(0.5, 1.0, 'Map From Exported GeoTIFF')





## \_\_\_\_\_process\_2k\_atterbury\_survey\_\_\_\_\_

December 20, 2022

```
[ ]: import sys
from os.path import join

import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
import pandas as pd
import scipy.linalg as la
from matplotlib import cm

PROJ_DIR = r'C:\Users\ltber\Desktop\files\AFIT\Research\WPAFB_Survey\mammal'
sys.path.append(PROJ_DIR)

from MAMMAL import Diurnal
from MAMMAL.MapLvl import pcaLvl
from MAMMAL.MapLvl import tieLvl
from MAMMAL.Parse import parseGSMP as pgsm
from MAMMAL.Parse import parseIM as pim
from MAMMAL.Parse import parseLCM as plcm
from MAMMAL.Utils import coordinateUtils as cu
from MAMMAL.Utils import Filters as filt
from MAMMAL.Utils import mapUtils as mu
from MAMMAL.Utils import ProcessingUtils as pu
from MAMMAL.VehicleCal import magUtilsTL as magtl

%matplotlib inline
plt.rcParams["figure.figsize"] = (30, 15) # (w, h)
plt.style.use(['seaborn-poster', 'seaborn-white'])
pd.set_option('mode.chained_assignment', None)

shrink = 0.8
aspect = 20 * 0.7
xlims = [-86.025, -85.99]
ylims = [39.32, 39.35]
cmap = cm.coolwarm
s = 10
```

```

alpha = 0.85

GMAP_EXTENT = [-86.054121, -85.981633, 39.319868, 39.349562]

SURVEY_DIR = r'D:\__atterbury_data__\2k_atterbury_survey_'
GMAP_FNAME = r'D:\__atterbury_data__\atterbury.jpg'
REF_FNAME = r'D:\__atterbury_data__\2k_atterbury_survey_\Ground_Reference.csv'
LOG_FNAME = r'D:
    \__atterbury_data__\2k_atterbury_survey_\__2k_atterbury_survey__.csv'
KML_FNAME = r'2k_Atterbury_Survey.kml'

SURVEY_NAME = '2km Atterbury Survey'

MAX_TERRAIN_MSL = 230 # (m)
MAX_SURVEY_AGL = 400 # (m)
MAX_SURVEY_CRUISE = 30 # (m/s)

MAX_EXPECTED_FREQ = MAX_SURVEY_CRUISE / MAX_SURVEY_AGL # (hz)

TL_C = np.array([-1.86687725e+01, 1.33975396e+02, -1.80762945e+02, 1.
    -69023832e-01,
    -3.92262356e-03, -1.84382741e-03, 1.71830230e-01, -1.
    -61173781e-04,
    1.72575427e-01, -4.31927864e-04, -8.21512835e-05, -4.
    -37609432e-05,
    -1.06838978e-04, -1.22444017e-04, -2.76294434e-04, -8.
    -51727772e-05,
    3.16374022e-05, -2.77441572e-05])

TL_TERMS = magtl.DEFAULT_TL_TERMS
TL_COEFF_TYPES = ['Permanent', 'Induced', 'Eddy']

SCALAR_TYPE = 'Geometrics MFAM optically pumped caesium scalar magnetometer
    with 2 sensor heads'
VECTOR_TYPE = 'TwinLeaf VMR anisotropic magnetoresistive vector magnetometer'

FINAL_filt_CUT = MAX_SURVEY_AGL
FINAL_filt_ORDER = 6

DX = 5 # Map pixel width (m)
DY = 5 # Map pixel height (m)

POC = '''Autonomy and Navigation Technology Center
        Air Force Institute of Technology
        Graduate School of Engineering and Management
        2950 Hobson Way, Bldg 646, Rm 205

```

```

Wright Patterson AFB, Ohio 45433

Telephone: (937) 255-3636 Ext. 4671'''

PROCESS_APP = 'MAMMAL 0.0.1'

```

## 1 Load Survey Data

```

[ ]: img = mpimg.imread(GMAP_FNAME)

ref_df = pd.read_csv(REF_FNAME, parse_dates=['datetime'])

log_df      = pd.read_csv(LOG_FNAME, parse_dates=['datetime'])
datetimes   = log_df.datetime
timestamps  = np.array(log_df.epoch_sec)
sample_rate = 1.0 / np.diff(timestamps).mean()

```

## 2 Interpolate Reference Data

```

[ ]: # Despike reference data
span  = 10
delta = 0.5

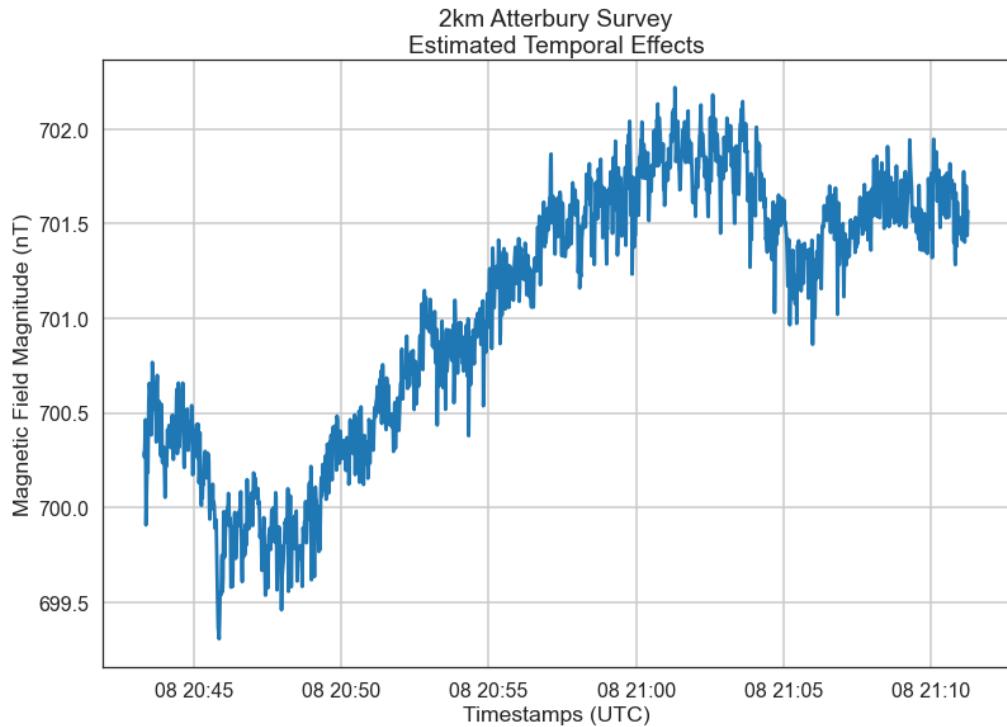
ref_despiked = pu.remove_outliers(ref_df.F, pu.ewma_fb(ref_df.F, span), delta)

ref_df.F = ref_despiked
ref_df.F = ref_df.F.interpolate()

# Interpolate reference data
_, ref_mag = Diurnal.interp_reference_df(df          = ref_df,
                                           timestamps = timestamps,
                                           survey_lon = log_df.LONG.mean(),
                                           subtract_core = True)

plt.figure()
plt.title('{}\\nEstimated Temporal Effects'.format(SURVEY_NAME))
plt.xlabel('Timestamps (UTC)')
plt.ylabel('Magnetic Field Magnitude (nT)')
plt.plot(log_df.datetime, ref_mag)
plt.grid()

```



### 3 Determine When Each Sensor Head was Valid During the Survey

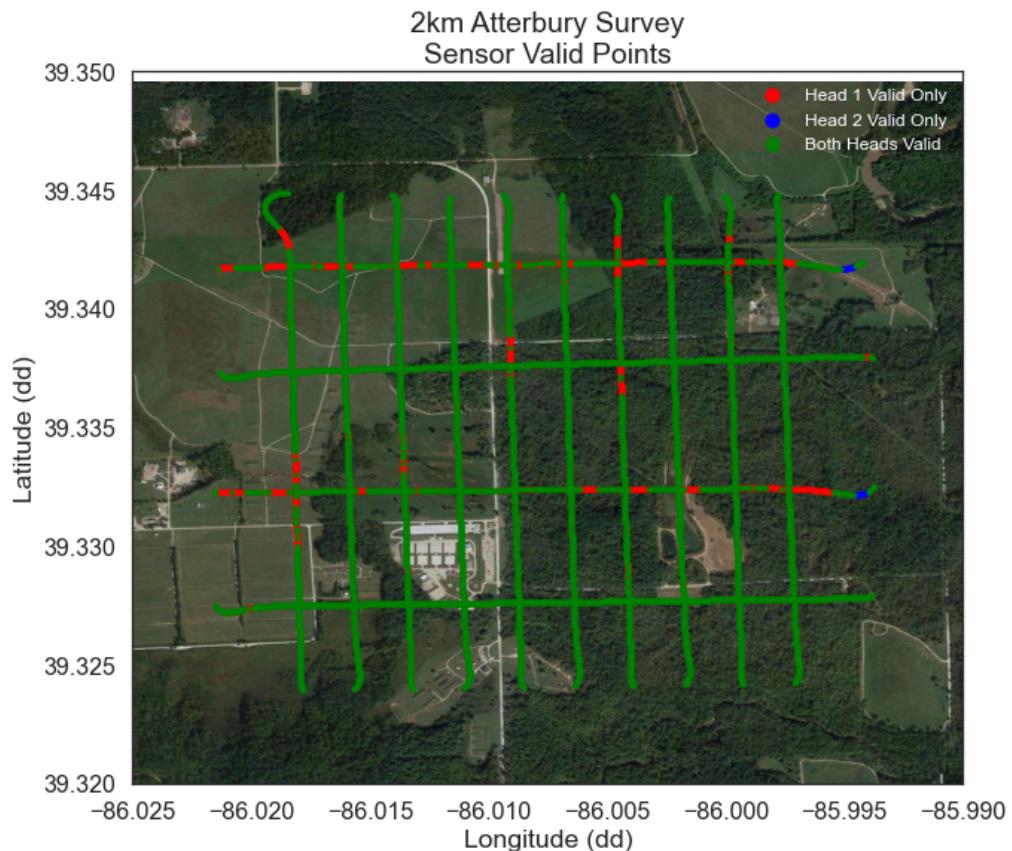
```
[ ]: plt.figure()
plt.title('{}\nSensor Valid Points'.format(SURVEY_NAME))
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
plt.scatter(log_df.LONG[(log_df.SCALAR_1_VALID == 1) & (log_df.SCALAR_2_VALID == 0) & (log_df.LINE_TYPE != 0)],
            log_df.LAT[(log_df.SCALAR_1_VALID == 1) & (log_df.SCALAR_2_VALID == 0) & (log_df.LINE_TYPE != 0)],
            s=s,
            c='r',
            label='Head 1 Valid Only')
plt.scatter(log_df.LONG[(log_df.SCALAR_1_VALID == 0) & (log_df.SCALAR_2_VALID == 1) & (log_df.LINE_TYPE != 0)],
            log_df.LAT[(log_df.SCALAR_1_VALID == 0) & (log_df.SCALAR_2_VALID == 1) & (log_df.LINE_TYPE != 0)],
            s=s,
```

```

        c='b',
        label='Head 2 Valid Only')
plt.scatter(log_df.LONG[(log_df.SCALAR_1_VALID == 1) & (log_df.SCALAR_2_VALID == 1) & (log_df.LINE_TYPE != 0)],
            log_df.LAT[(log_df.SCALAR_1_VALID == 1) & (log_df.SCALAR_2_VALID == 1) & (log_df.LINE_TYPE != 0)],
            s=s,
            c='g',
            label='Both Heads Valid')
plt.legend(labelcolor='white', fontsize='large', markerscale=3)
plt.xlim(xlims)
plt.ylim(ylims)

```

[ ]: (39.32, 39.35)



## 4 Apply Tolles Lawson Calibration

```
[ ]: # Compile vector data
b_vector = np.hstack([np.array(log_df.X)[:, np.newaxis],
                      np.array(log_df.Y)[:, np.newaxis],
                      np.array(log_df.Z)[:, np.newaxis]])

dcs  = b_vector / la.norm(b_vector, axis=1)[:, np.newaxis]
dc_x = dcs[:, 0]
dc_y = dcs[:, 1]
dc_z = dcs[:, 2]

# Calibrate sensor head
f = (log_df.SCALAR_1_LPF + log_df.SCALAR_2_LPF) / 2.0

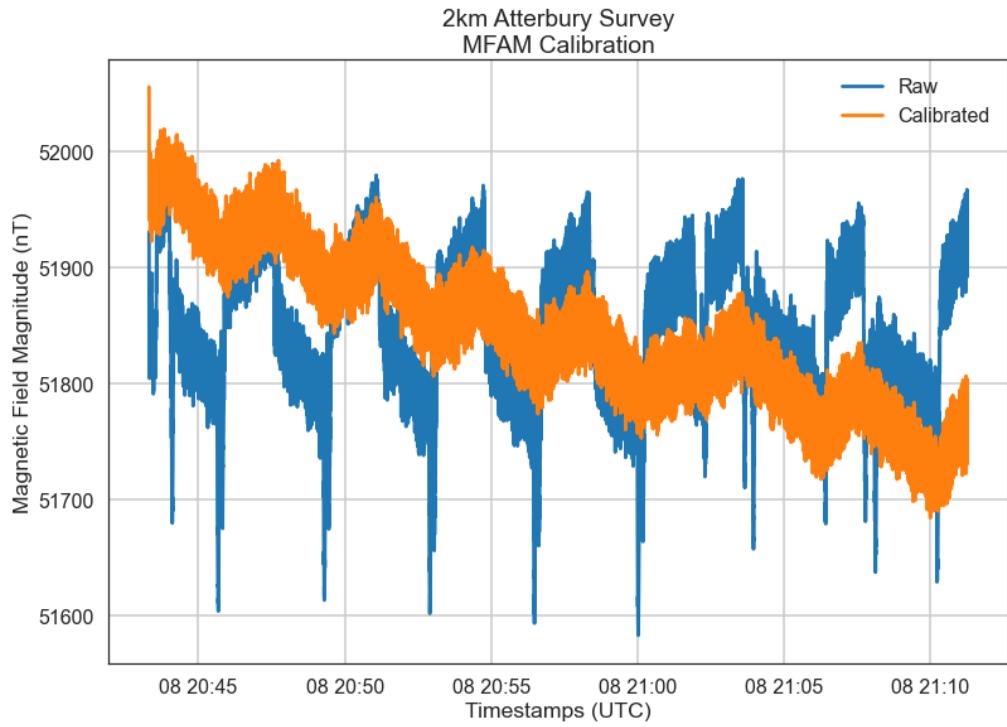
body_effects_scalar = magtl.tlc_compensation(vector = b_vector,
                                              tlc      = TL_C,
                                              terms    = TL_TERMS)

f_cal = f - body_effects_scalar

f_cal += (f.mean() - f_cal.mean())

# Update dataframe with calibrated data
log_df['SCALAR_CAL'] = f_cal
log_df['F']           = log_df.SCALAR_CAL

plt.figure()
plt.title('{}\\nMFAM Calibration'.format(SURVEY_NAME))
plt.xlabel('Timestamps (UTC)')
plt.ylabel('Magnetic Field Magnitude (nT)')
plt.plot(log_df.datetime, f,      label='Raw')
plt.plot(log_df.datetime, f_cal, label='Calibrated')
plt.legend()
plt.grid()
```



## 5 Find Estimated Magnetic Anomaly Values

```
[ ]: f_cal_igrf = f_cal - log_df.IGRF_F
f_cal_igrf_temporal = f_cal_igrf - ref_mag
f_cal_igrf_temporal_filt = filt.lpf(f_cal_igrf_temporal, MAX_EXPECTED_FREQ,
                                     sample_rate)

process_dict = {'TIMESTAMP': timestamps,
                'LAT': log_df.LAT,
                'LONG': log_df.LONG,
                'ALT': log_df.ALT,
                'DC_X': dc_x,
                'DC_Y': dc_y,
                'DC_Z': dc_z,
                'F': f,
                'F_CAL': f_cal,
                'F_CAL_IGRF': f_cal_igrf,
                'F_CAL_IGRF_TEMPORAL': f_cal_igrf_temporal,
                'F_CAL_IGRF_TEMPORAL_FILTER': f_cal_igrf_temporal_filt}
```

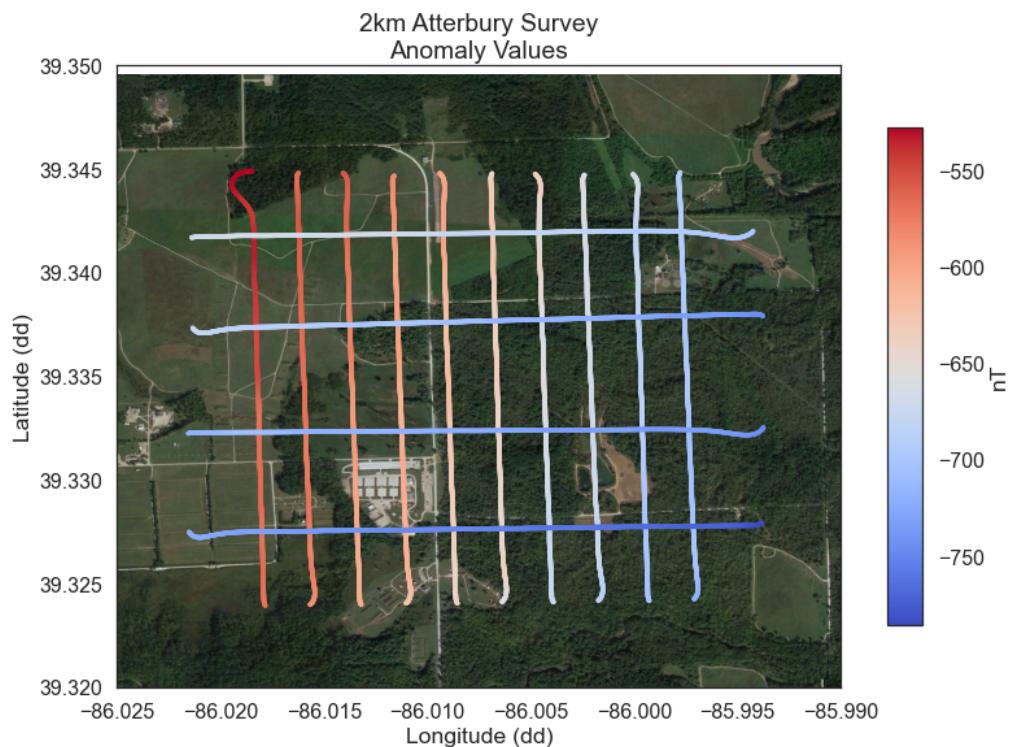
```

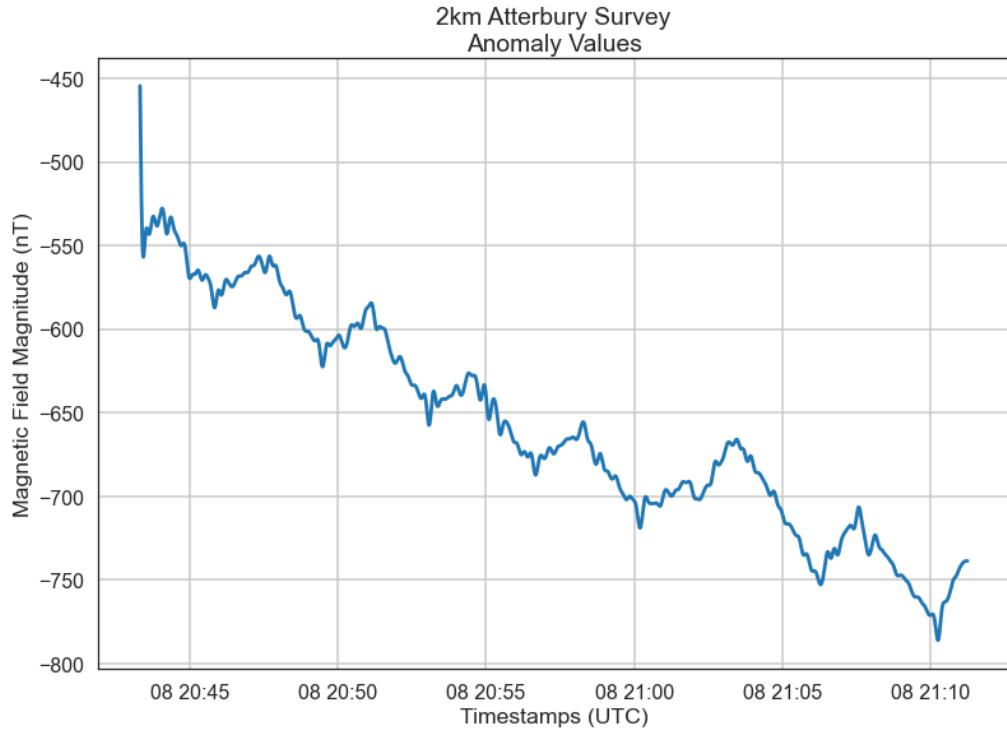
log_df.F = f_cal_igrf_temporal_filt

plt.figure()
plt.title('{}\\nAnomaly Values'.format(SURVEY_NAME))
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.scatter(log_df.LONG[log_df.LINE_TYPE != 0],
                 log_df.LAT[log_df.LINE_TYPE != 0],
                 s=s,
                 c=log_df.F[log_df.LINE_TYPE != 0],
                 cmap=cmap)
plt.xlim(xlims)
plt.ylim(ylims)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)

plt.figure()
plt.title('{}\\nAnomaly Values'.format(SURVEY_NAME))
plt.xlabel('Timestamps (UTC)')
plt.ylabel('Magnetic Field Magnitude (nT)')
plt.plot(log_df.datetime, log_df.F)
plt.grid()

```





## 6 Crop Map Extent

```
[ ]: MIN_MAP_LAT = 39.3261
      MAX_MAP_LAT = 39.3428
```

## 7 Create Maps Based on Non-Leveled Flight Lines

```
[ ]: interp_type = 'RBF'
interp_df = pu.interp_flight_lines(anomaly_df      = log_df[(log_df.LAT >= MIN_MAP_LAT) & (log_df.LAT <= MAX_MAP_LAT)],
                                    dx              = DX,
                                    dy              = DY,
                                    max_terrain_msl = MAX_TERRAIN_MSL,
                                    buffer          = 0,
                                    interp_type     = interp_type,
                                    neighbors       = None,
                                    skip_na_mask    = True)
```

```

interp_lats    = interp_df['LAT']
interp_lons    = interp_df['LONG']
interp_scalar  = interp_df['F']
interp_heights = interp_df['ALT']
interp_std     = interp_df['STD']

interp_scalar_LPF = filt.lpf2(interp_scalar,
                               MAX_SURVEY_AGL,
                               DX,
                               DY)

map_title = '{}\nNon-Leveled Anomaly Map\nUsing {} Interpolation'.
            format(SURVEY_NAME, interp_type)

plt.figure()
plt.title(map_title)
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.pcolormesh(interp_lons,
                     interp_lats,
                     interp_scalar,
                     cmap=cmap,
                     alpha=alpha)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)

map_title = '{}\nNon-Leveled Anomaly Map\nUsing {} Interpolation (Filtered)'.
            format(SURVEY_NAME, interp_type)

plt.figure()
plt.title(map_title)
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.pcolormesh(interp_lons,
                     interp_lats,
                     interp_scalar_LPF,
                     cmap=cmap,
                     alpha=alpha)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)

extent = [interp_lons.min(), interp_lats.min(), interp_lons.max(), interp_lats.
          max()]

```

```

xmin, ymin, xmax, ymax = extent

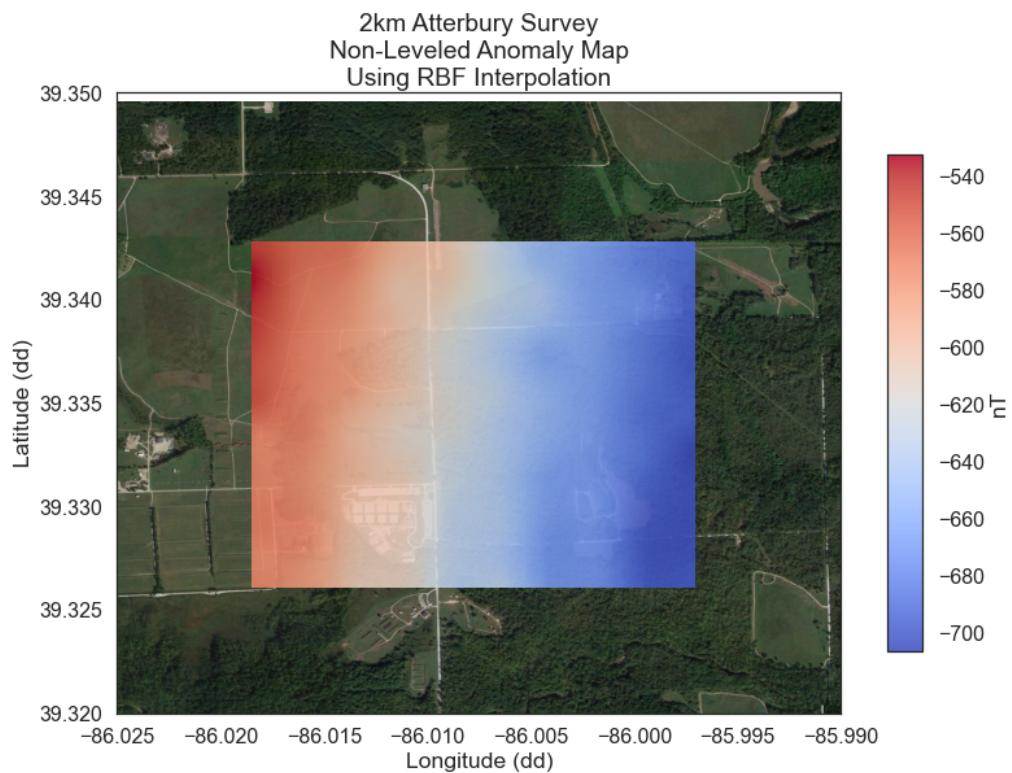
flight_path = np.hstack([np.array(log_df.LAT)[:, np.newaxis],
                        np.array(log_df.LONG)[:, np.newaxis],
                        np.array(log_df.ALT)[:, np.newaxis],
                        np.array(log_df.epoch_sec)[:, np.newaxis]])
area_polys = [{'NAME': 'Survey Area',
               'FL_DIRFL_DISTTL_DIRTL_DISTLATLONGALT# Export map as a GeoTIFF
map = mu.export_map(out_dir
                     = SURVEY_DIR,
                     location = ' '.join(map_title.split()),
                     date = log_df.datetime[0],
                     lats = interp_lats,
                     lons = interp_lons,
                     scalar = interp_scalar_LPF,
                     heights = interp_heights,
                     process_df = pd.DataFrame(process_dict),
                     process_app = PROCESS_APP,
                     stds = interp_std,
                     vector = None,
                     scalar_type = SCALAR_TYPE,
                     vector_type = VECTOR_TYPE,
                     scalar_var = np.nan,
                     vector_var = np.nan,
                     poc = POC,
                     flight_path = flight_path,
                     area_polys = area_polys,
                     osm_path = None,
                     level_type = 'No leveling',
                     tl_coeff_types = TL_COEFF_TYPES,
                     tl_coeffs = TL_C,
                     interp_type = interp_type,
                     final_filt_cut = FINAL_FILT_CUT,
                     final_filt_order = FINAL_FILT_ORDER)

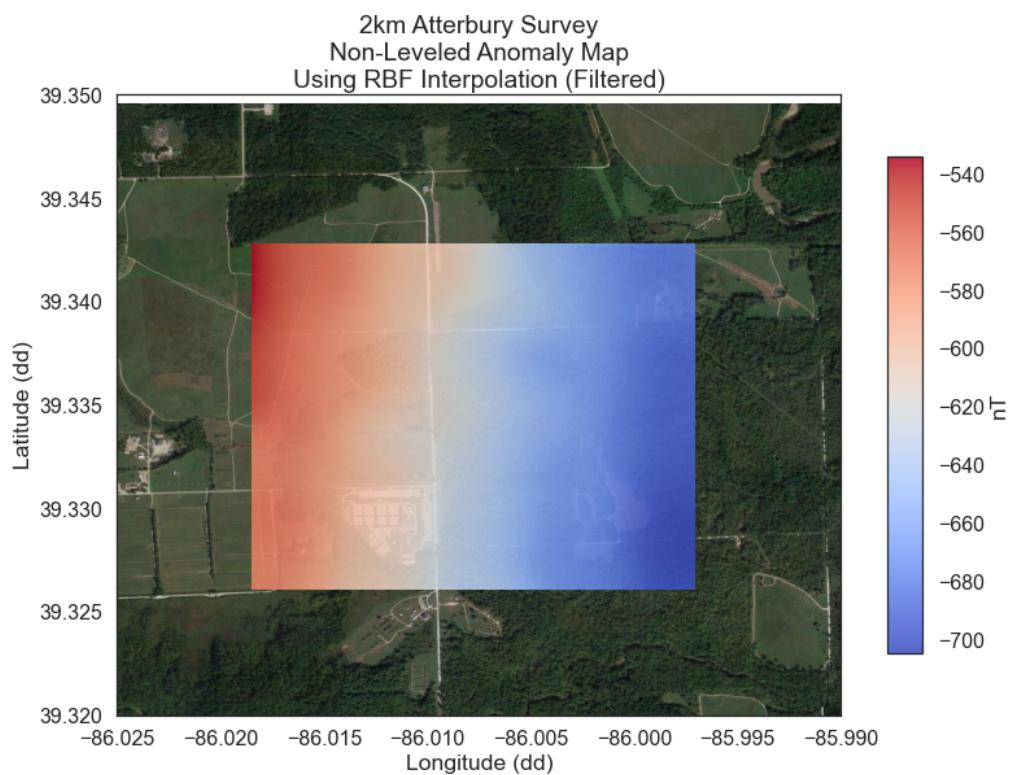
plt.figure()
map[0].plot(cmap=cm.coolwarm)
plt.title('Map From Exported GeoTIFF')

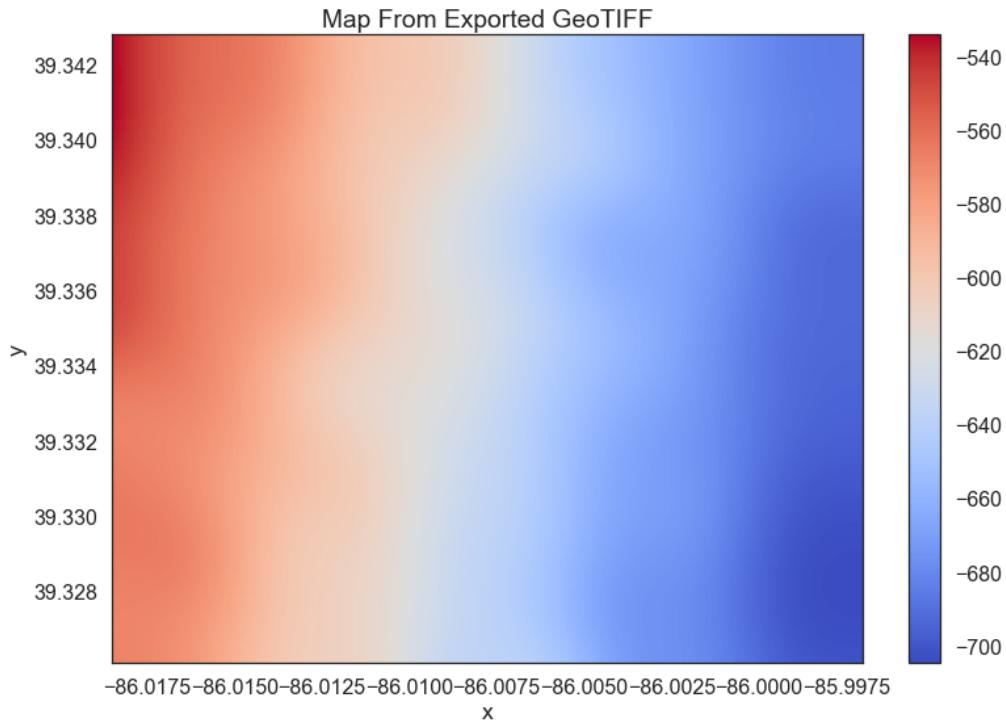
```

Interpolating survey anomaly data to map coordinates  
Running radial basis function (RBF) interpolation for all map pixels

[ ]: Text(0.5, 1.0, 'Map From Exported GeoTIFF')





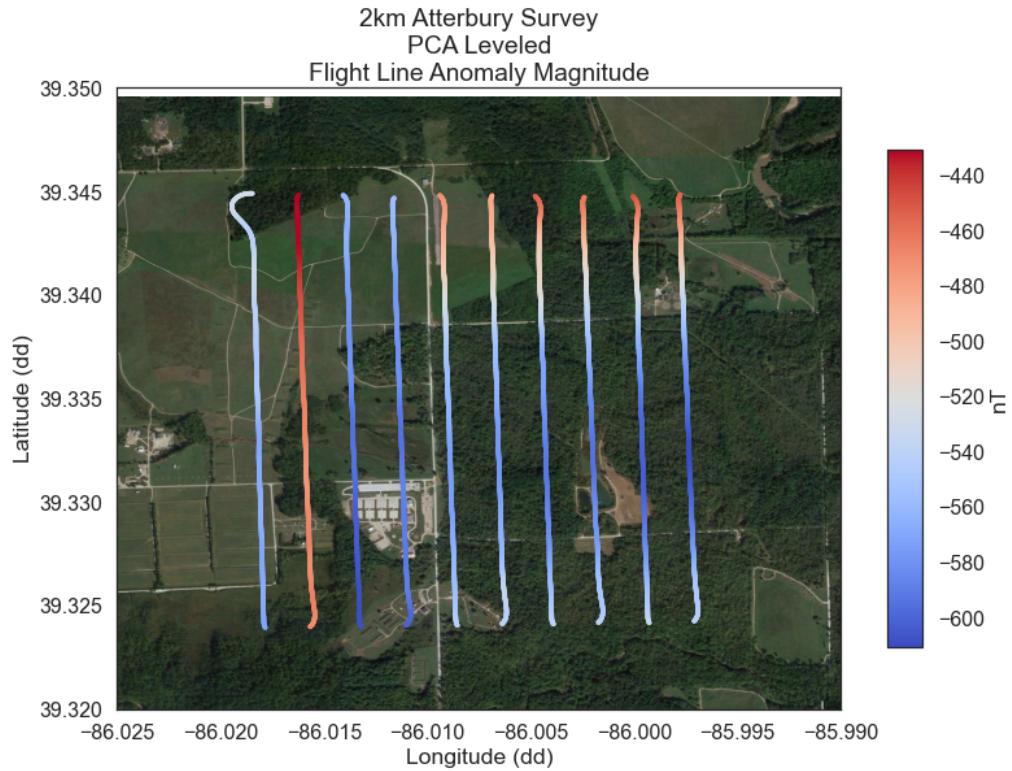


## 8 Apply PCA-Based Flight Line Leveling

```
[ ]: lvld_survey_df = pcaLvl.pca_lvl(survey_df = log_df,
                                      num_ptls = 2,
                                      pts_locs = np.array([0.25, 0.75]))

plt.figure()
plt.title('PCA Leveled\nFlight Line Anomaly Magnitude'.format(SURVEY_NAME))
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.scatter(lvld_survey_df.LONG[lvld_survey_df.LINE_TYPE == 1],
                 lvld_survey_df.LAT[lvld_survey_df.LINE_TYPE == 1],
                 s=s,
                 c=lvld_survey_df.F[lvld_survey_df.LINE_TYPE == 1],
                 cmap=cmap)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)
```

```
[ ]: (39.32, 39.35)
```



## 9 Create Maps Based on PCA-Leveled Flight Lines

```
[ ]: interp_type = 'RBF'  
interp_df = pu.interp_flight_lines(anomaly_df  
    ↪lvld_survey_df[(lvld_survey_df.LAT >= MIN_MAP_LAT) & (lvld_survey_df.LAT <=  
    ↪MAX_MAP_LAT)],  
    dx          = DX,  
    dy          = DY,  
    max_terrain_msl = MAX_TERRAIN_MSL,  
    buffer      = 0,  
    interp_type = interp_type,  
    neighbors   = None,  
    skip_na_mask = True)  
  
interp_lats    = interp_df['LAT']  
interp_lons    = interp_df['LONG']  
interp_scalar  = interp_df['F']
```

```

interp_heights = interp_df['ALT']
interp_std     = interp_df['STD']

interp_scalar_LPF = filt.lpf2(interp_scalar,
                               MAX_SURVEY_AGL,
                               DX,
                               DY)

map_title = '{}\nPCA Leveled Anomaly Map\nUsing {} Interpolation'.
            .format(SURVEY_NAME, interp_type)

plt.figure()
plt.title(map_title)
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.pcolormesh(interp_lons,
                     interp_lats,
                     interp_scalar,
                     cmap=cmap,
                     alpha=alpha)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)

map_title = '{}\nPCA Leveled Anomaly Map\nUsing {} Interpolation (Filtered)'.
            .format(SURVEY_NAME, interp_type)

plt.figure()
plt.title(map_title)
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.pcolormesh(interp_lons,
                     interp_lats,
                     interp_scalar_LPF,
                     cmap=cmap,
                     alpha=alpha)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)

process_dict['F_CAL_IGRF_TEMPORAL_FILT_LEVEL'] = lvld_survey_df.F

# Export map as a GeoTIFF
map = mu.export_map(out_dir           = SURVEY_DIR,
                    location        = ' '.join(map_title.split()),
```

```

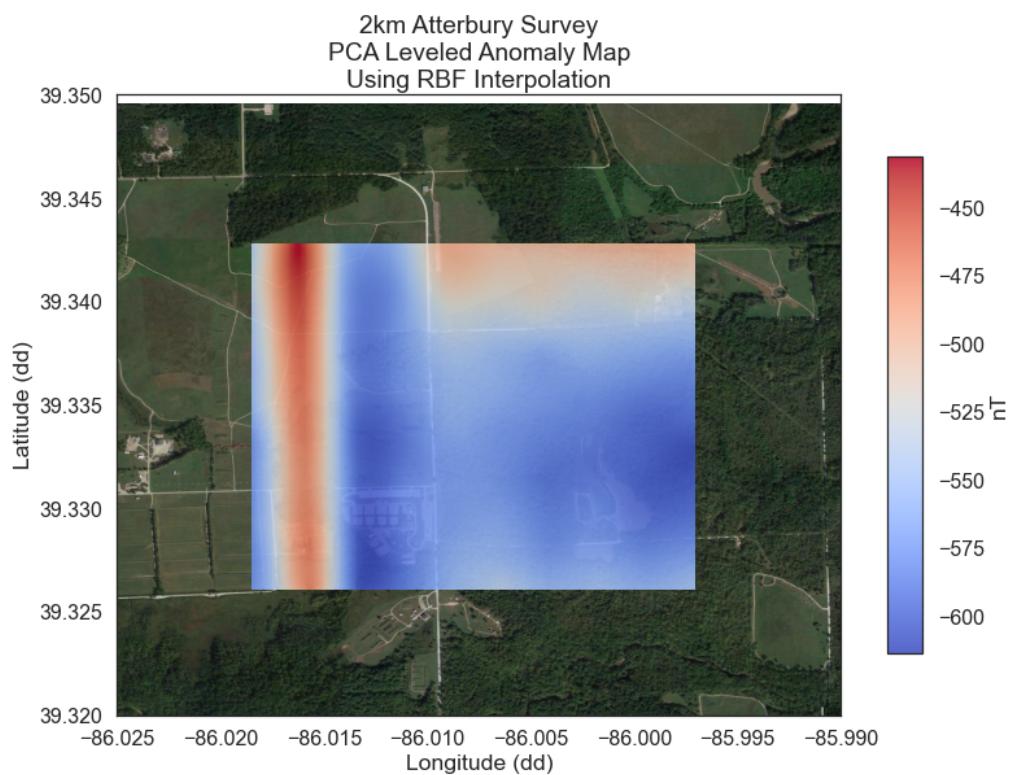
date          = log_df.datetime[0],
lats          = interp_lats,
lons          = interp_lons,
scalar        = interp_scalar_LPF,
heights       = interp_heights,
process_df    = pd.DataFrame(process_dict),
process_app   = PROCESS_APP,
stds          = interp_std,
vector         = None,
scalar_type   = SCALAR_TYPE,
vector_type   = VECTOR_TYPE,
scalar_var    = np.nan,
vector_var    = np.nan,
poc            = POC,
flight_path   = flight_path,
area_polys    = area_polys,
osm_path       = None,
level_type    = 'PCA pseudo tie line leveling',
tl_coeff_types = TL_COEFF_TYPES,
tl_coeffs     = TL_C,
interp_type   = interp_type,
final_filt_cut = FINAL_FILT_CUT,
final_filt_order = FINAL_FILT_ORDER)

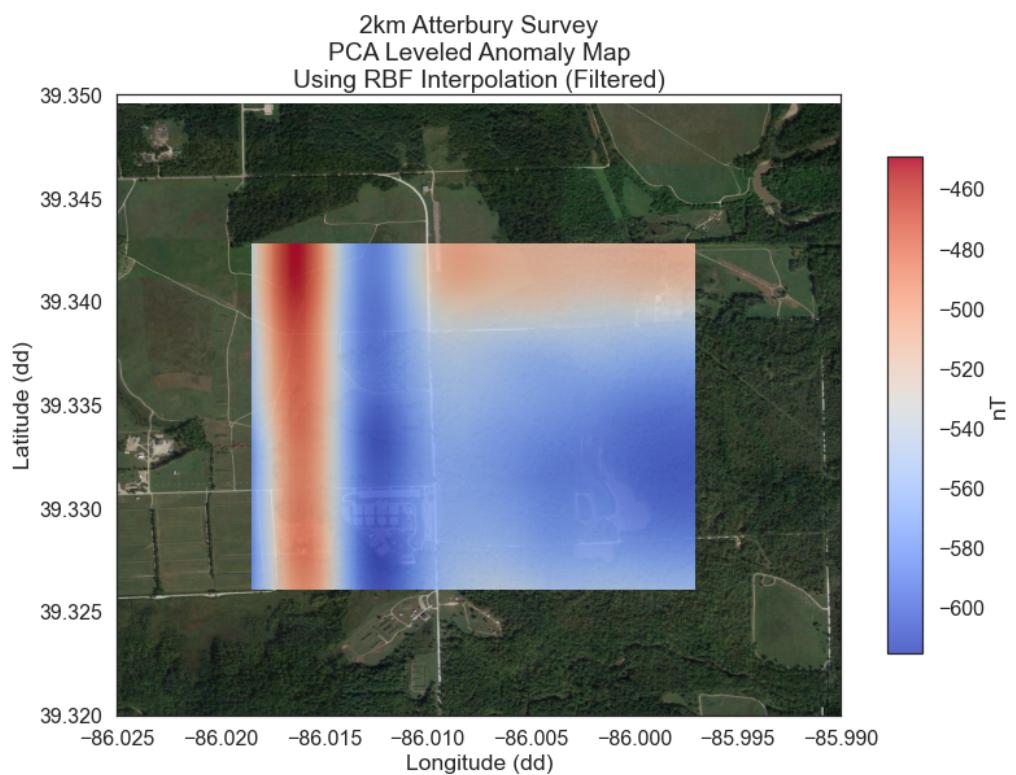
plt.figure()
map[0].plot(cmap=cm.coolwarm)
plt.title('Map From Exported GeoTIFF')

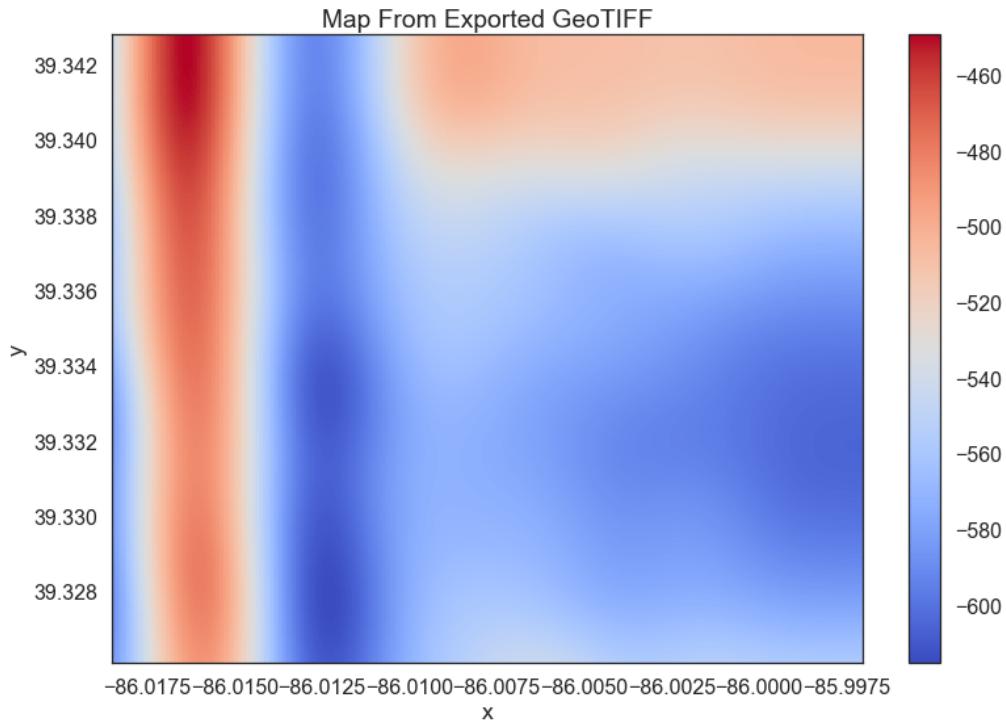
```

Interpolating survey anomaly data to map coordinates  
Running radial basis function (RBF) interpolation for all map pixels

[ ]: Text(0.5, 1.0, 'Map From Exported GeoTIFF')





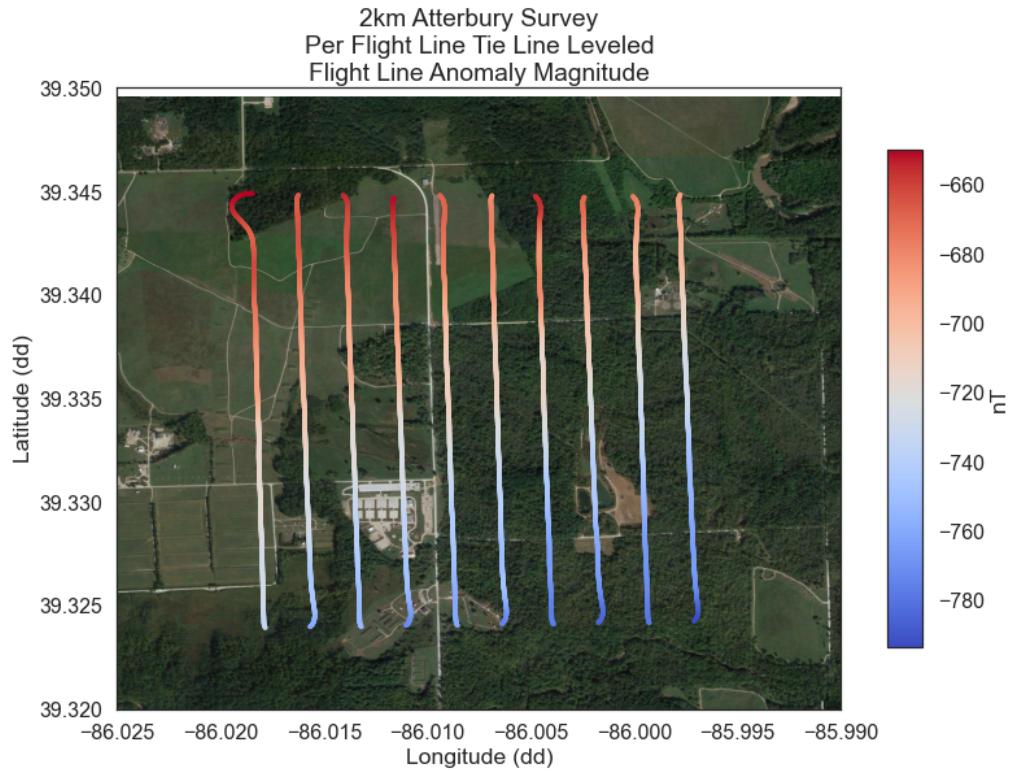


## 10 Apply Per Flight Line Tie Line-Based Flight Line Leveling

```
[ ]: lvld_survey_df = tieLvl.tie_lvl(survey_df = log_df,
                                     approach = 'lobf')

plt.figure()
plt.title('{}\\nPer Flight Line Tie Line Leveled\\nFlight Line Anomaly Magnitude'.
          format(SURVEY_NAME))
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.scatter(lvld_survey_df.LONG[lvld_survey_df.LINE_TYPE == 1],
                 lvld_survey_df.LAT[lvld_survey_df.LINE_TYPE == 1],
                 s=s,
                 c=lvld_survey_df.F[lvld_survey_df.LINE_TYPE == 1],
                 cmap=cmap)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)
```

```
[ ]: (39.32, 39.35)
```



## 11 Create Map Based on Per Flight Line Tie Line-Leveled Flight Lines

```
[ ]: interp_type = 'RBF'  
interp_df = pu.interp_flight_lines(anomaly_df  
    ↪lvld_survey_df[(lvld_survey_df.LAT >= MIN_MAP_LAT) & (lvld_survey_df.LAT <=  
    ↪MAX_MAP_LAT)],  
    dx          = DX,  
    dy          = DY,  
    max_terrain_msl = MAX_TERRAIN_MSL,  
    buffer      = 0,  
    interp_type = interp_type,  
    neighbors   = None,  
    skip_na_mask = True)  
  
interp_lats   = interp_df['LAT']  
interp_lons   = interp_df['LONG']
```

```

interp_scalar = interp_df['F']
interp_heights = interp_df['ALT']
interp_std = interp_df['STD']

interp_scalar_LPF = filt.lpf2(interp_scalar,
                               MAX_SURVEY_AGL,
                               DX,
                               DY)

map_title = '{}\nPer Flight Line Tie Line Leveled Anomaly Map\nUsing {}  
↳Interpolation'.format(SURVEY_NAME, interp_type)

plt.figure()
plt.title(map_title)
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.pcolormesh(interp_lons,
                     interp_lats,
                     interp_scalar,
                     cmap=cmap,
                     alpha=alpha)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)

map_title = '{}\nPer Flight Line Tie Line Leveled Anomaly Map\nUsing {}  
↳Interpolation (Filtered)'.format(SURVEY_NAME, interp_type)

plt.figure()
plt.title(map_title)
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.pcolormesh(interp_lons,
                     interp_lats,
                     interp_scalar_LPF,
                     cmap=cmap,
                     alpha=alpha)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)

process_dict['F_CAL_IGRF_TEMPORAL_FILTER_LEVEL'] = lvld_survey_df.F

# Export map as a GeoTIFF
map = mu.export_map(out_dir = SURVEY_DIR,

```

```

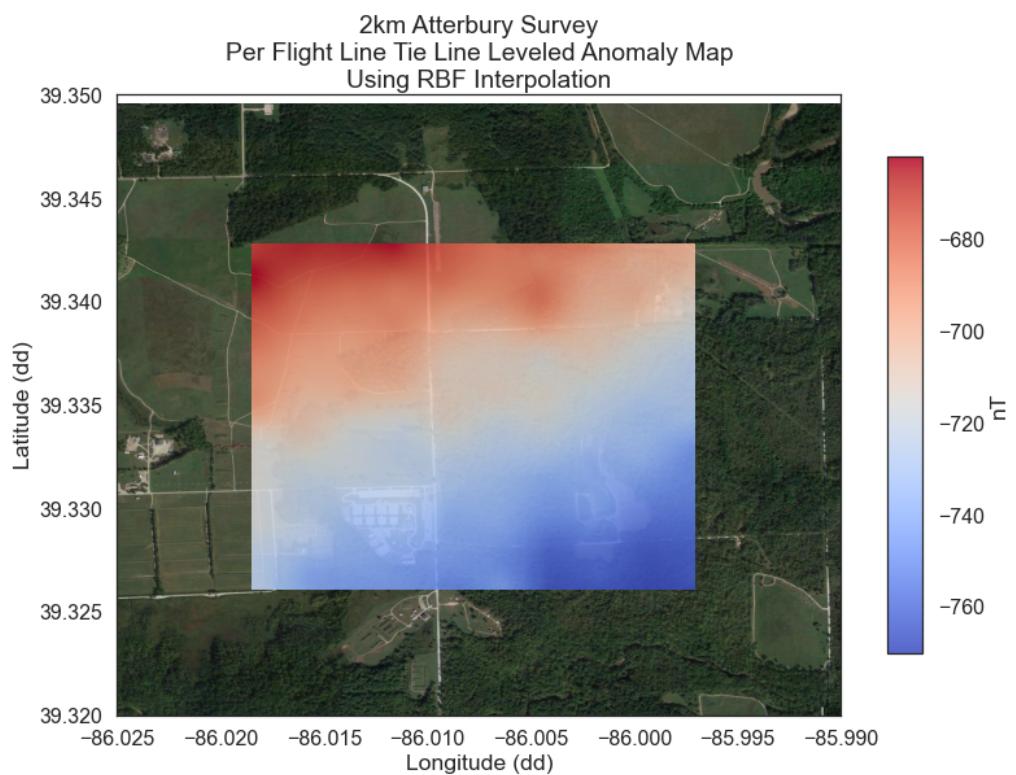
location      = ' '.join(map_title.split()),
date         = log_df.datetime[0],
lats          = interp_lats,
lons          = interp_lons,
scalar        = interp_scalar_LPF,
heights       = interp_heights,
process_df    = pd.DataFrame(process_dict),
process_app   = PROCESS_APP,
stds          = interp_std,
vector         = None,
scalar_type   = SCALAR_TYPE,
vector_type   = VECTOR_TYPE,
scalar_var    = np.nan,
vector_var    = np.nan,
poc            = POC,
flight_path   = flight_path,
area_polys    = area_polys,
osm_path       = None,
level_type    = 'Per flight line tie line leveling',
tl_coeff_types = TL_COEFF_TYPES,
tl_coeffs     = TL_C,
interp_type   = interp_type,
final_filt_cut = FINAL_FILT_CUT,
final_filt_order = FINAL_FILT_ORDER)

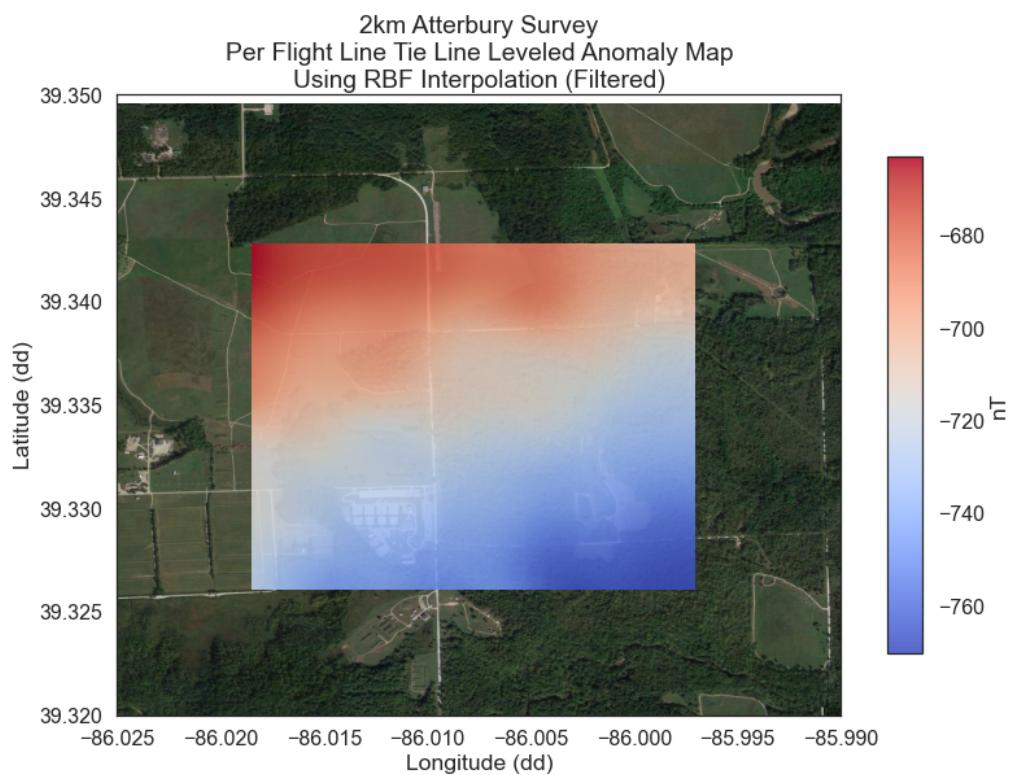
plt.figure()
map[0].plot(cmap=cm.coolwarm)
plt.title('Map From Exported GeoTIFF')

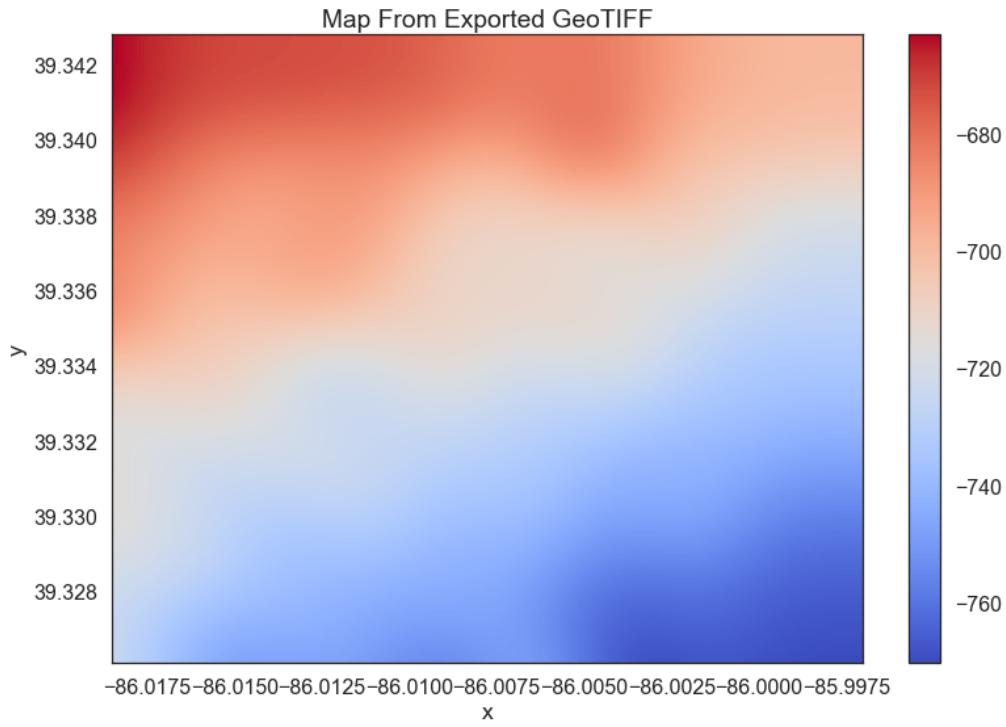
```

Interpolating survey anomaly data to map coordinates  
Running radial basis function (RBF) interpolation for all map pixels

[ ]: Text(0.5, 1.0, 'Map From Exported GeoTIFF')





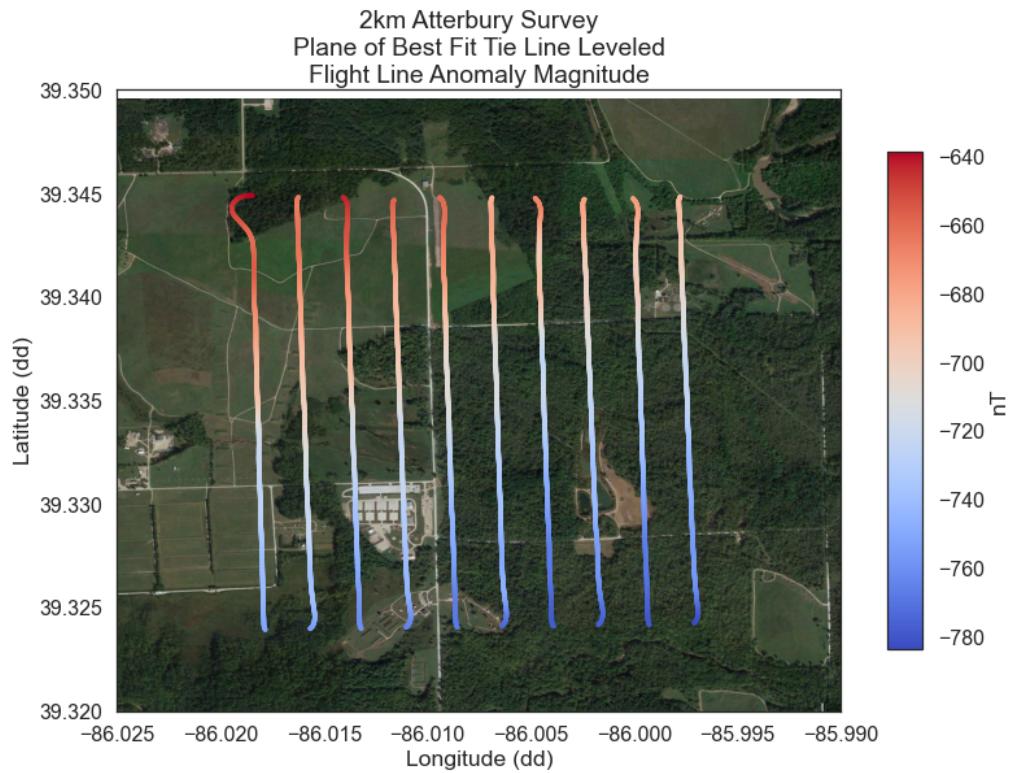


## 12 Apply Plane of Best Fit Tie Line-Based Flight Line Leveling

```
[ ]: lvld_survey_df = tieLvl.tie_lvl(survey_df = log_df,
                                     approach = 'lsq')

plt.figure()
plt.title('{} \nPlane of Best Fit Tie Line Leveled \nFlight Line Anomaly\n Magnitude'.format(SURVEY_NAME))
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.scatter(lvld_survey_df.LONG[lvld_survey_df.LINE_TYPE == 1],
                 lvld_survey_df.LAT[lvld_survey_df.LINE_TYPE == 1],
                 s=s,
                 c=lvld_survey_df.F[lvld_survey_df.LINE_TYPE == 1],
                 cmap=cmap)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)
```

```
[ ]: (39.32, 39.35)
```



### 13 Create Map Based on Plane of Best Fit Tie Line-Leveled Flight Lines

```
[ ]: interp_type = 'RBF'  
interp_df = pu.interp_flight_lines(anomaly_df  
    ↪lvld_survey_df[(lvld_survey_df.LAT >= MIN_MAP_LAT) & (lvld_survey_df.LAT <=  
    ↪MAX_MAP_LAT)],  
    dx = DX,  
    dy = DY,  
    max_terrain_msl = MAX_TERRAIN_MSL,  
    buffer = 0,  
    interp_type = interp_type,  
    neighbors = None,  
    skip_na_mask = True)  
  
interp_lats = interp_df['LAT']  
interp_lons = interp_df['LONG']
```

```

interp_scalar = interp_df['F']
interp_heights = interp_df['ALT']
interp_std     = interp_df['STD']

interp_scalar_LPF = filt.lpf2(interp_scalar,
                               MAX_SURVEY_AGL,
                               DX,
                               DY)

map_title = '{}\nPlane of Best Fit Tie Line Leveled Anomaly Map\nUsing {}_{}  
↳Interpolation'.format(SURVEY_NAME, interp_type)

plt.figure()
plt.title(map_title)
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.pcolormesh(interp_lons,
                     interp_lats,
                     interp_scalar,
                     cmap=cmap,
                     alpha=alpha)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)

map_title = '{}\nPlane of Best Fit Tie Line Leveled Anomaly Map\nUsing {}_{}  
↳Interpolation (Filtered)'.format(SURVEY_NAME, interp_type)

plt.figure()
plt.title(map_title)
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.pcolormesh(interp_lons,
                     interp_lats,
                     interp_scalar_LPF,
                     cmap=cmap,
                     alpha=alpha)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)

process_dict['F_CAL_IGRF_TEMPORAL_FILTER_LEVEL'] = lvld_survey_df.F

# Export map as a GeoTIFF
map = mu.export_map(out_dir           = SURVEY_DIR,

```

```

location      = ' '.join(map_title.split()),
date         = log_df.datetime[0],
lats          = interp_lats,
lons          = interp_lons,
scalar        = interp_scalar_LPF,
heights       = interp_heights,
process_df    = pd.DataFrame(process_dict),
process_app   = PROCESS_APP,
stds          = interp_std,
vector         = None,
scalar_type   = SCALAR_TYPE,
vector_type   = VECTOR_TYPE,
scalar_var    = np.nan,
vector_var    = np.nan,
poc            = POC,
flight_path   = flight_path,
area_polys    = area_polys,
osm_path       = None,
level_type    = 'Plane of best fit tie line leveling',
tl_coeff_types = TL_COEFF_TYPES,
tl_coeffs     = TL_C,
interp_type   = interp_type,
final_filt_cut = FINAL_FILT_CUT,
final_filt_order = FINAL_FILT_ORDER)

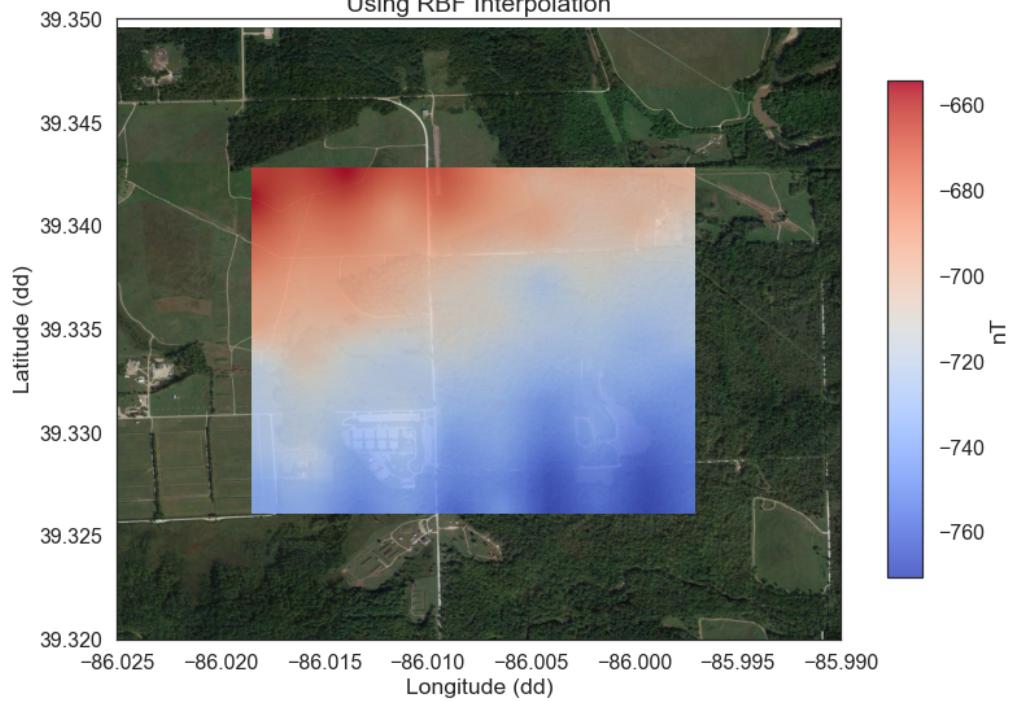
plt.figure()
map[0].plot(cmap=cm.coolwarm)
plt.title('Map From Exported GeoTIFF')

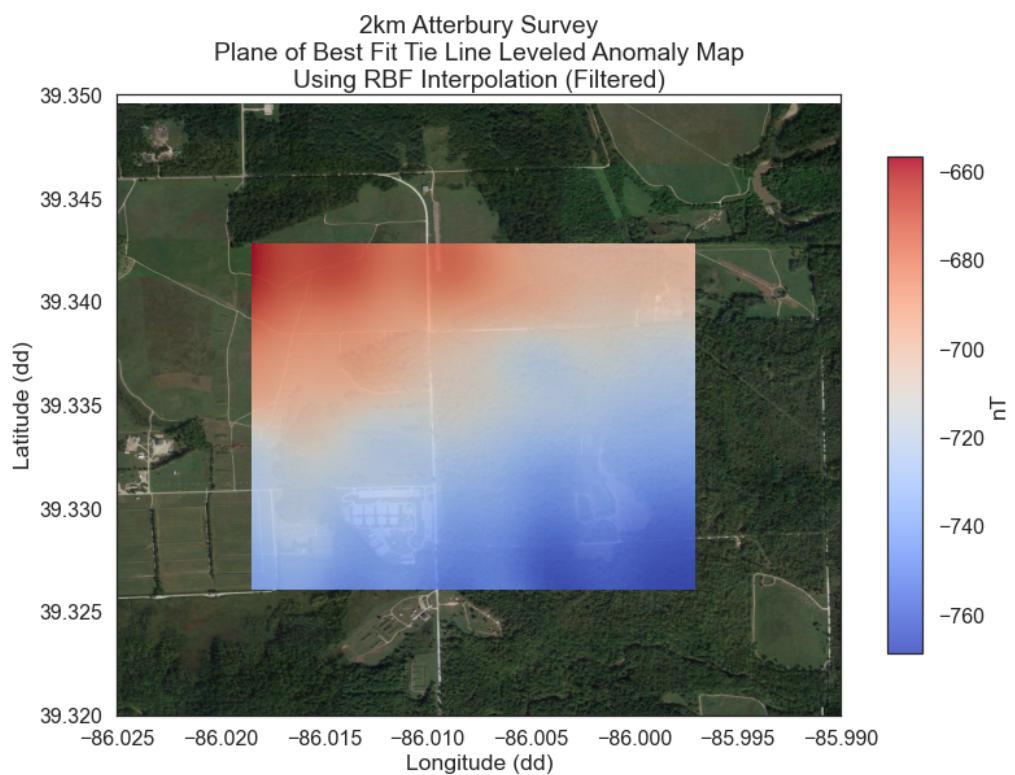
```

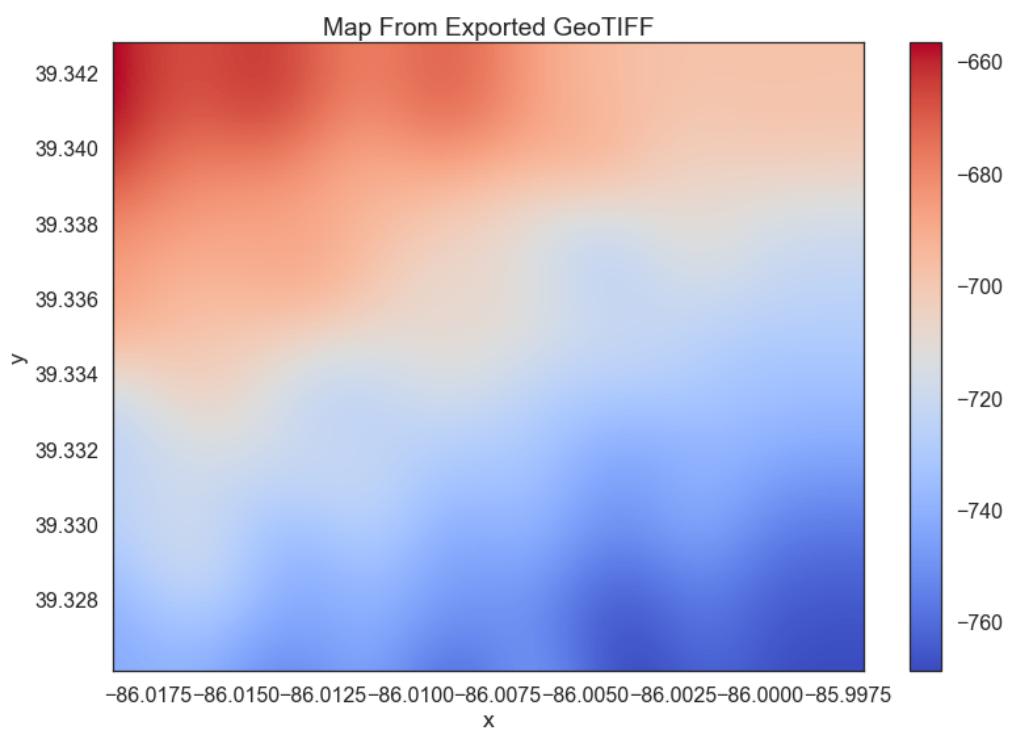
Interpolating survey anomaly data to map coordinates  
Running radial basis function (RBF) interpolation for all map pixels

[ ]: Text(0.5, 1.0, 'Map From Exported GeoTIFF')

2km Atterbury Survey  
Plane of Best Fit Tie Line Leveled Anomaly Map  
Using RBF Interpolation







## \_\_\_\_\_process\_4k\_atterbury\_survey\_\_\_\_\_

December 20, 2022

```
[ ]: import sys
from os.path import join

import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
import pandas as pd
import scipy.linalg as la
from matplotlib import cm

PROJ_DIR = r'C:\Users\ltber\Desktop\files\AFIT\Research\WPAFB_Survey\mammal'
sys.path.append(PROJ_DIR)

from MAMMAL import Diurnal
from MAMMAL.MapLvl import pcaLvl
from MAMMAL.MapLvl import tieLvl
from MAMMAL.Parse import parseGSMP as pgsm
from MAMMAL.Parse import parseIM as pim
from MAMMAL.Parse import parseLCM as plcm
from MAMMAL.Utils import coordinateUtils as cu
from MAMMAL.Utils import Filters as filt
from MAMMAL.Utils import mapUtils as mu
from MAMMAL.Utils import ProcessingUtils as pu
from MAMMAL.VehicleCal import magUtilsTL as magtl

%matplotlib inline
plt.rcParams["figure.figsize"] = (30, 15) # (w, h)
plt.style.use(['seaborn-poster', 'seaborn-white'])
pd.set_option('mode.chained_assignment', None)

shrink = 0.8
aspect = 20 * 0.7
xlims = [-86.028, -85.9927]
ylims = [39.3050, 39.35]
cmap = cm.coolwarm
s = 10
```

```

alpha = 0.85

GMAP_EXTENT = [-86.067298, -85.952393, 39.302228, 39.349337]

SURVEY_DIR = r'D:\__atterbury_data__\4k_atterbury_survey_'
GMAP_FNAME = r'D:\__atterbury_data__\atterbury_large.jpg'
REF_FNAME = r'D:\__atterbury_data__\4k_atterbury_survey_\Ground_Reference.csv'
LOG_FNAME = r'D:
    \__atterbury_data__\4k_atterbury_survey_\__4k_atterbury_survey__.csv'
KML_FNAME = r'4k_Atterbury_Survey.kml'

SURVEY_NAME = '4km Atterbury Survey'

MAX_TERRAIN_MSL = 230 # (m)
MAX_SURVEY_AGL = 400 # (m)
MAX_SURVEY_CRUISE = 30 # (m/s)

MAX_EXPECTED_FREQ = MAX_SURVEY_CRUISE / MAX_SURVEY_AGL # (hz)

TL_C = np.array([-1.86687725e+01, 1.33975396e+02, -1.80762945e+02, 1.
    -69023832e-01,
    -3.92262356e-03, -1.84382741e-03, 1.71830230e-01, -1.
    -61173781e-04,
    1.72575427e-01, -4.31927864e-04, -8.21512835e-05, -4.
    -37609432e-05,
    -1.06838978e-04, -1.22444017e-04, -2.76294434e-04, -8.
    -51727772e-05,
    3.16374022e-05, -2.77441572e-05])

TL_TERMS = magtl.DEFAULT_TL_TERMS
TL_COEFF_TYPES = ['Permanent', 'Induced', 'Eddy']

SCALAR_TYPE = 'Geometrics MFAM optically pumped caesium scalar magnetometer
    with 2 sensor heads'
VECTOR_TYPE = 'TwinLeaf VMR anisotropic magnetoresistive vector magnetometer'

FINAL_filt_CUT = MAX_SURVEY_AGL
FINAL_filt_ORDER = 6

DX = 10 # Map pixel width (m)
DY = 10 # Map pixel height (m)

POC = '''Autonomy and Navigation Technology Center
    Air Force Institute of Technology
    Graduate School of Engineering and Management
    2950 Hobson Way, Bldg 646, Rm 205

```

```

Wright Patterson AFB, Ohio 45433

Telephone: (937) 255-3636 Ext. 4671'''

PROCESS_APP = 'MAMMAL 0.0.1'

```

## 1 Load Survey Data

```

[ ]: img = mpimg.imread(GMAP_FNAME)

ref_df = pd.read_csv(REF_FNAME, parse_dates=['datetime'])

log_df      = pd.read_csv(LOG_FNAME, parse_dates=['datetime'])
datetimes   = log_df.datetime
timestamps  = np.array(log_df.epoch_sec)
sample_rate = 1.0 / np.diff(timestamps).mean()

```

## 2 Interpolate Reference Data

```

[ ]: # Despike reference data
span  = 10
delta = 0.5

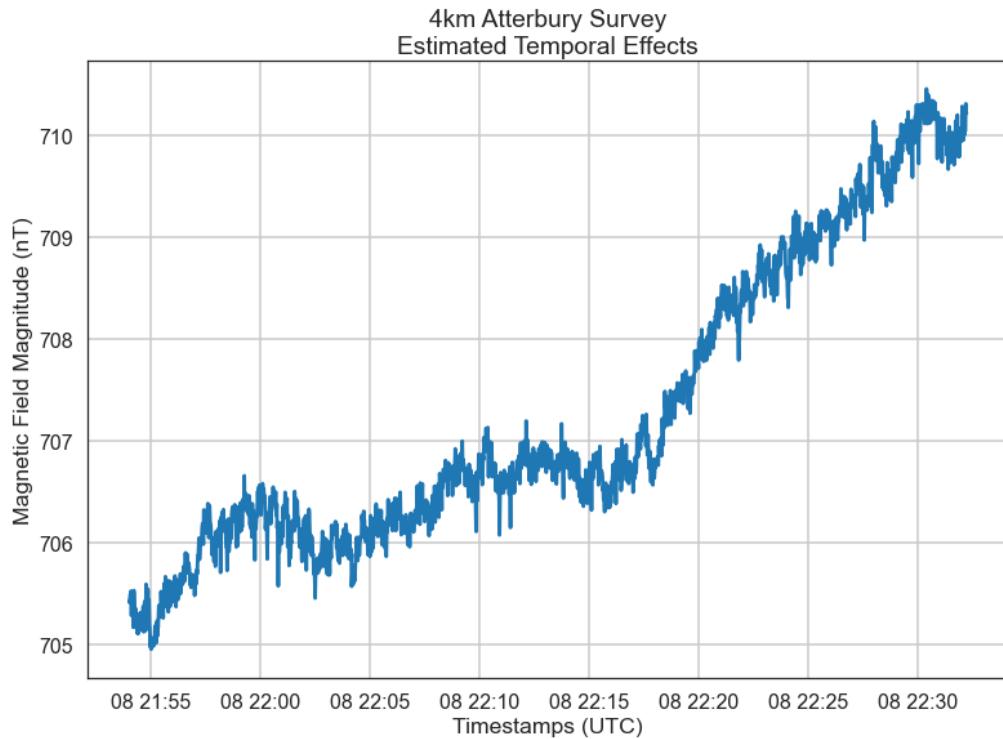
ref_despiked = pu.remove_outliers(ref_df.F, pu.ewma_fb(ref_df.F, span), delta)

ref_df.F = ref_despiked
ref_df.F = ref_df.F.interpolate()

# Interpolate reference data
_, ref_mag = Diurnal.interp_reference_df(df           = ref_df,
                                           timestamps = timestamps,
                                           survey_lon = log_df.LONG.mean(),
                                           subtract_core = True)

plt.figure()
plt.title('{}\\nEstimated Temporal Effects'.format(SURVEY_NAME))
plt.xlabel('Timestamps (UTC)')
plt.ylabel('Magnetic Field Magnitude (nT)')
plt.plot(log_df.datetime, ref_mag)
plt.grid()

```



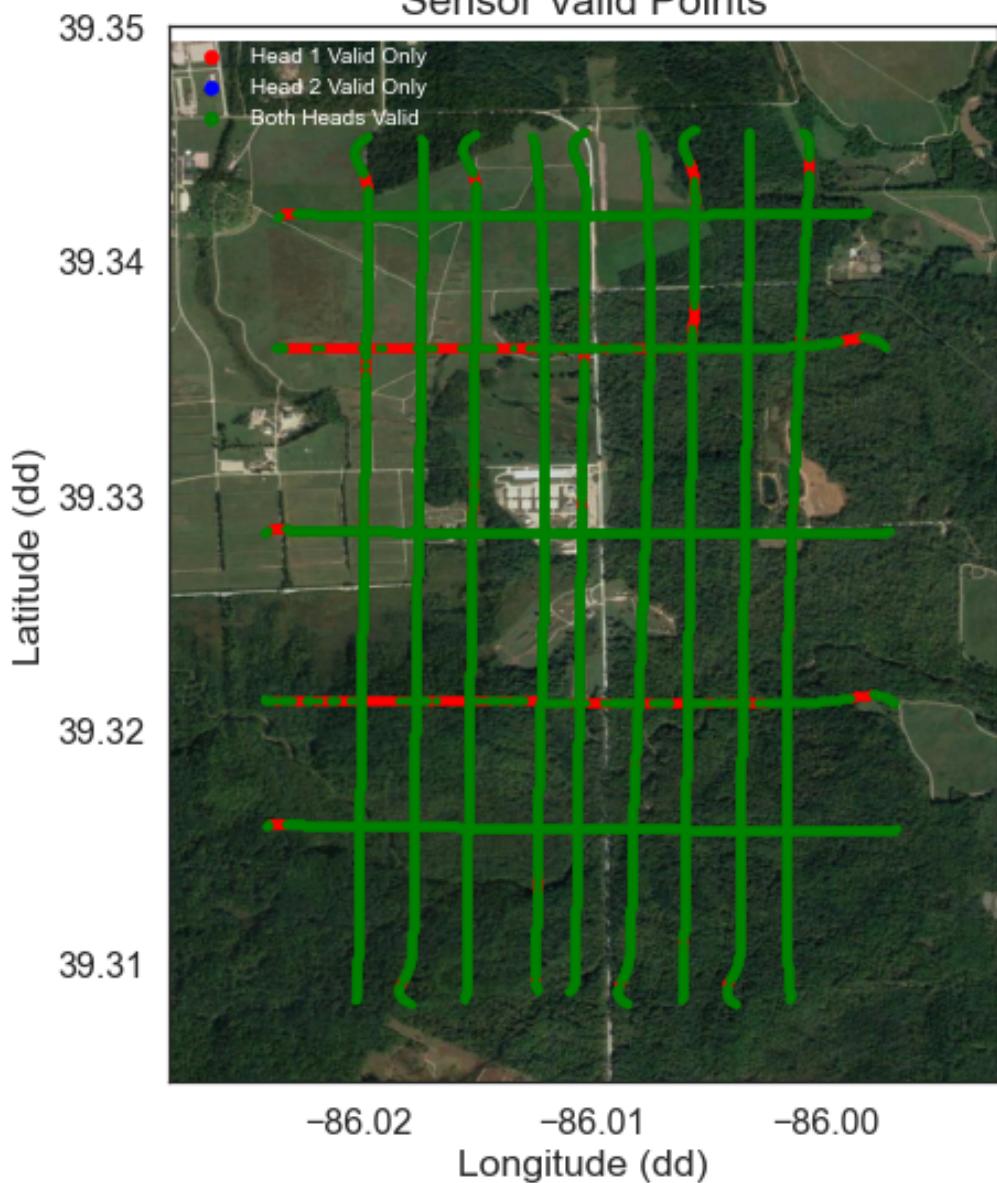
### 3 Determine When Each Sensor Head was Valid During the Survey

```
[ ]: plt.figure()
plt.title('{}\nSensor Valid Points'.format(SURVEY_NAME))
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
plt.scatter(log_df.LONG[(log_df.SCALAR_1_VALID == 1) & (log_df.SCALAR_2_VALID == 0) & (log_df.LINE_TYPE != 0)],
            log_df.LAT[(log_df.SCALAR_1_VALID == 1) & (log_df.SCALAR_2_VALID == 0) & (log_df.LINE_TYPE != 0)],
            s=s,
            c='r',
            label='Head 1 Valid Only')
plt.scatter(log_df.LONG[(log_df.SCALAR_1_VALID == 0) & (log_df.SCALAR_2_VALID == 1) & (log_df.LINE_TYPE != 0)],
            log_df.LAT[(log_df.SCALAR_1_VALID == 0) & (log_df.SCALAR_2_VALID == 1) & (log_df.LINE_TYPE != 0)],
```

```
s=s,
c='b',
label='Head 2 Valid Only')
plt.scatter(log_df.LONG[(log_df.SCALAR_1_VALID == 1) & (log_df.SCALAR_2_VALID== 1) & (log_df.LINE_TYPE != 0)],
            log_df.LAT[(log_df.SCALAR_1_VALID == 1) & (log_df.SCALAR_2_VALID == 1) & (log_df.LINE_TYPE != 0)],
            s=s,
            c='g',
            label='Both Heads Valid')
plt.legend(labelcolor='white', fontsize='medium', markerscale=2)
plt.xlim(xlims)
plt.ylim(ylims)
```

[ ]: (39.305, 39.35)

## 4km Atterbury Survey Sensor Valid Points



## 4 Apply Tolles Lawson Calibration

```
[ ]: # Compile vector data
b_vector = np.hstack([np.array(log_df.X)[:, np.newaxis],
                      np.array(log_df.Y)[:, np.newaxis],
                      np.array(log_df.Z)[:, np.newaxis]])

dcs  = b_vector / la.norm(b_vector, axis=1)[:, np.newaxis]
dc_x = dcs[:, 0]
dc_y = dcs[:, 1]
dc_z = dcs[:, 2]

# Calibrate sensor head
f = (log_df.SCALAR_1_LPF + log_df.SCALAR_2_LPF) / 2.0

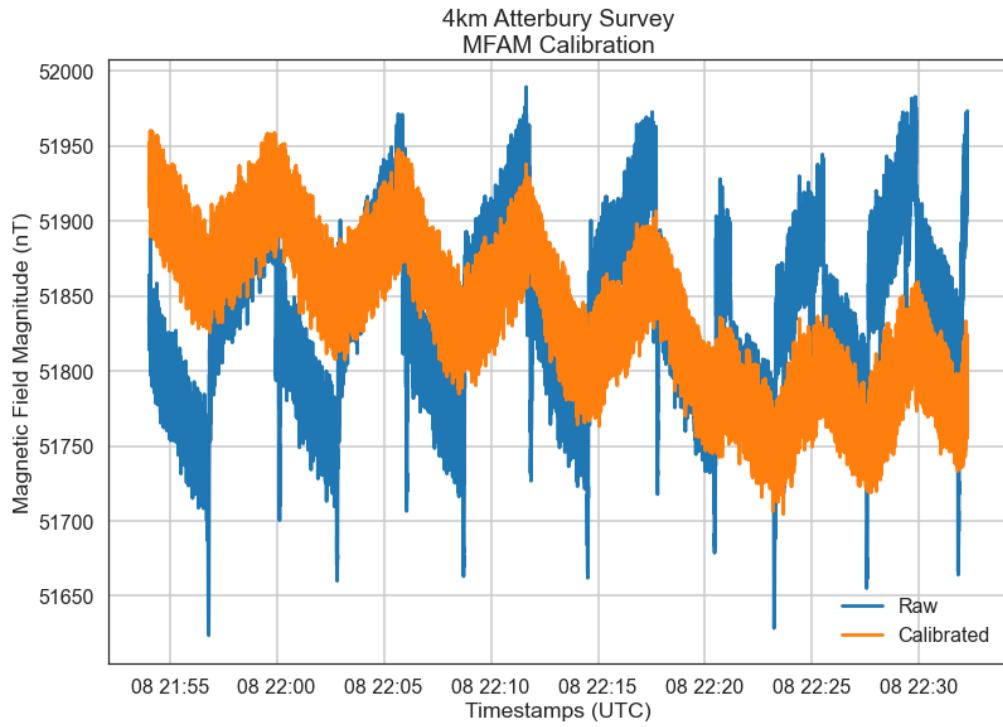
body_effects_scalar = magtl.tlc_compensation(vector = b_vector,
                                              tlc      = TL_C,
                                              terms    = TL_TERMS)

f_cal = f - body_effects_scalar

f_cal += (f.mean() - f_cal.mean())

# Update dataframe with calibrated data
log_df['SCALAR_CAL'] = f_cal
log_df['F']           = log_df.SCALAR_CAL

plt.figure()
plt.title('{}\\nMFAM Calibration'.format(SURVEY_NAME))
plt.xlabel('Timestamps (UTC)')
plt.ylabel('Magnetic Field Magnitude (nT)')
plt.plot(log_df.datetime, f,      label='Raw')
plt.plot(log_df.datetime, f_cal, label='Calibrated')
plt.legend()
plt.grid()
```



## 5 Find Estimated Magnetic Anomaly Values

```
[ ]: f_cal_igrf = f_cal - log_df.IGRF_F
f_cal_igrf_temporal = f_cal_igrf - ref_mag
f_cal_igrf_temporal_filt = filt.lpf(f_cal_igrf_temporal, MAX_EXPECTED_FREQ,
                                     sample_rate)

process_dict = {'TIMESTAMP': timestamps,
                'LAT': log_df.LAT,
                'LONG': log_df.LONG,
                'ALT': log_df.ALT,
                'DC_X': dc_x,
                'DC_Y': dc_y,
                'DC_Z': dc_z,
                'F': f,
                'F_CAL': f_cal,
                'F_CAL_IGRF': f_cal_igrf,
                'F_CAL_IGRF_TEMPORAL': f_cal_igrf_temporal,
                'F_CAL_IGRF_TEMPORAL_FILTER': f_cal_igrf_temporal_filt}
```

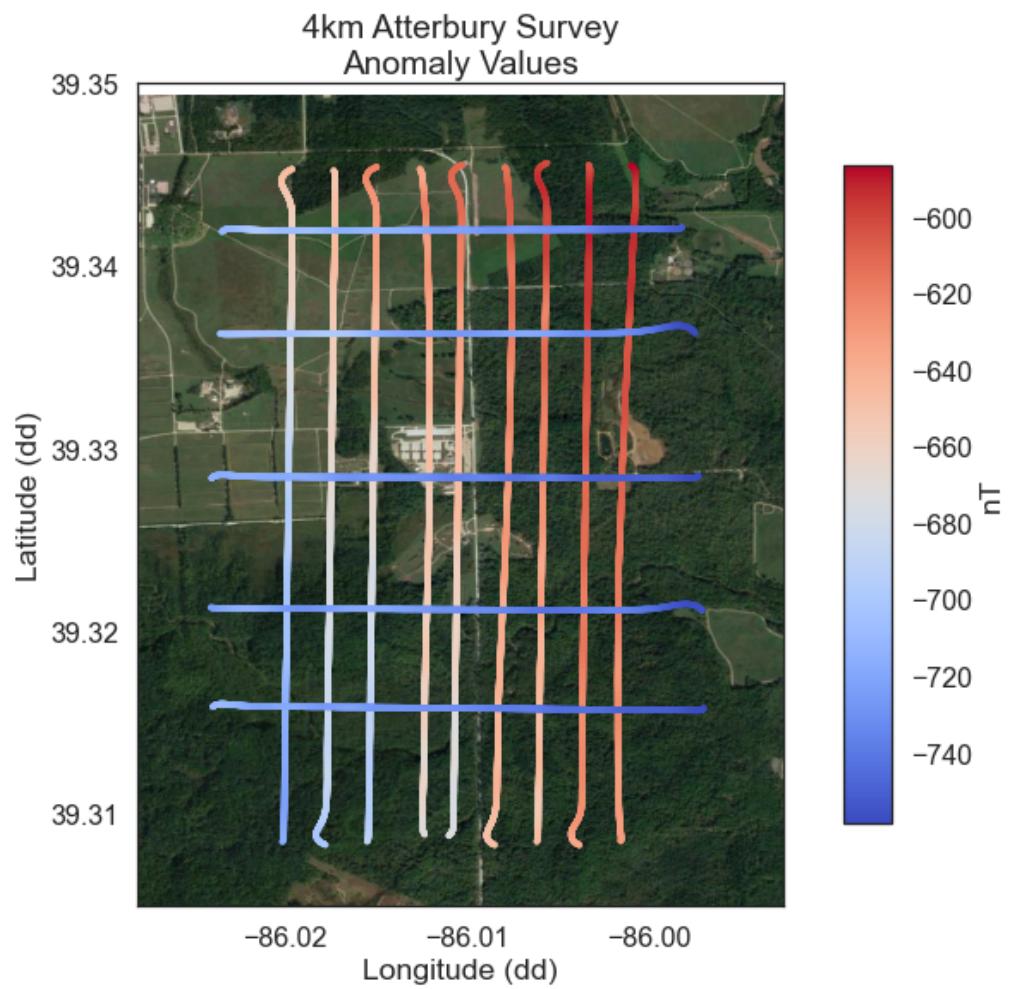
```

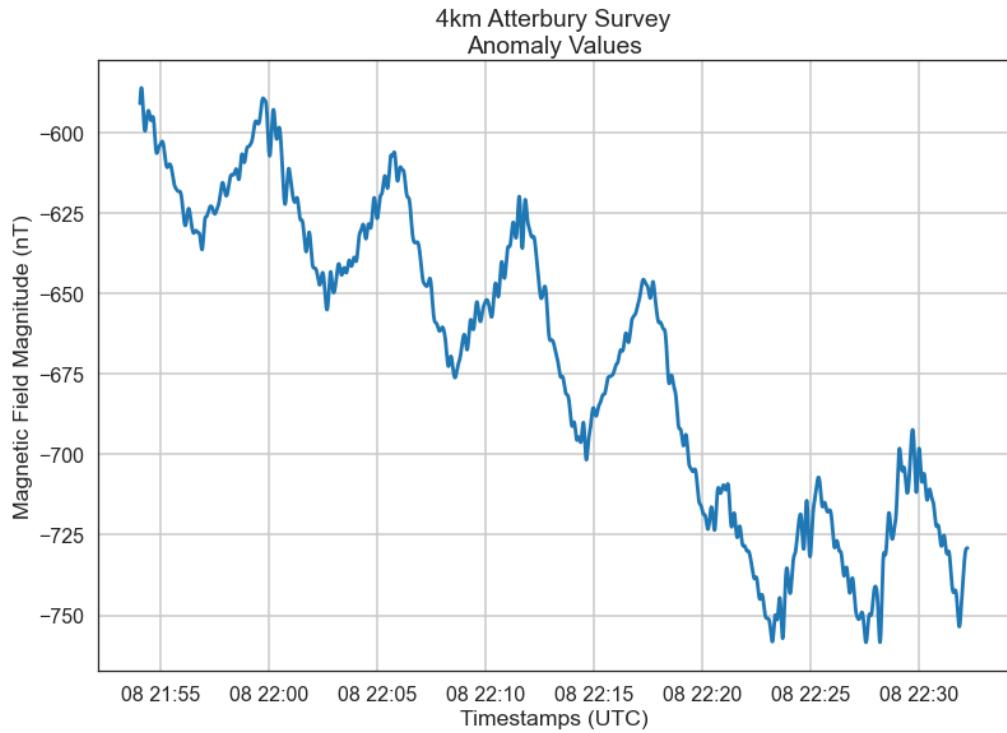
log_df.F = f_cal_igrf_temporal_filt

plt.figure()
plt.title('{}\\nAnomaly Values'.format(SURVEY_NAME))
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.scatter(log_df.LONG[log_df.LINE_TYPE != 0],
                  log_df.LAT[log_df.LINE_TYPE != 0],
                  s=s,
                  c=log_df.F[log_df.LINE_TYPE != 0],
                  cmap=cmap)
plt.xlim(xlims)
plt.ylim(ylims)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)

plt.figure()
plt.title('{}\\nAnomaly Values'.format(SURVEY_NAME))
plt.xlabel('Timestamps (UTC)')
plt.ylabel('Magnetic Field Magnitude (nT)')
plt.plot(log_df.datetime, log_df.F)
plt.grid()

```





## 6 Crop Map Extent

```
[ ]: MIN_MAP_LAT = 39.3127
      MAX_MAP_LAT = 39.3427
```

## 7 Create Maps Based on Non-Leveled Flight Lines

```
[ ]: interp_type = 'RBF'
interp_df = pu.interp_flight_lines(anomaly_df      = log_df[(log_df.LAT >= MIN_MAP_LAT) & (log_df.LAT <= MAX_MAP_LAT)],
                                    dx              = DX,
                                    dy              = DY,
                                    max_terrain_msl = MAX_TERRAIN_MSL,
                                    buffer          = 0,
                                    interp_type     = interp_type,
                                    neighbors       = None,
                                    skip_na_mask   = True)

interp_lats = interp_df['LAT']
```

```

interp_lons    = interp_df['LONG']
interp_scalar  = interp_df['F']
interp_heights = interp_df['ALT']
interp_std     = interp_df['STD']

interp_scalar_LPF = filt.lpf2(interp_scalar,
                               MAX_SURVEY_AGL,
                               DX,
                               DY)

map_title = '{}\nNon-Leveled Anomaly Map\nUsing {} Interpolation'.
            .format(SURVEY_NAME, interp_type)

plt.figure()
plt.title(map_title)
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.pcolormesh(interp_lons,
                     interp_lats,
                     interp_scalar,
                     cmap=cmap,
                     alpha=alpha)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)

map_title = '{}\nNon-Leveled Anomaly Map\nUsing {} Interpolation (Filtered)'.
            .format(SURVEY_NAME, interp_type)

plt.figure()
plt.title(map_title)
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.pcolormesh(interp_lons,
                     interp_lats,
                     interp_scalar_LPF,
                     cmap=cmap,
                     alpha=alpha)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)

extent = [interp_lons.min(), interp_lats.min(), interp_lons.max(), interp_lats.
          .max()]
xmin, ymin, xmax, ymax = extent

```

```

flight_path = np.hstack([np.array(log_df.LAT)[:, np.newaxis],
                        np.array(log_df.LONG)[:, np.newaxis],
                        np.array(log_df.ALT)[:, np.newaxis],
                        np.array(log_df.epoch_sec)[:, np.newaxis]]))

area_polys = [ {'NAME': 'Survey Area',
                 'FL_DIR': 0,
                 'FL_DIST': 200,
                 'TL_DIR': 90,
                 'TL_DIST': 200,
                 'LAT': [ymax, ymax, ymin, ymin, ymax],
                 'LONG': [xmin, xmax, xmax, xmin, xmin],
                 'ALT': [MAX_TERRAIN_MSL + MAX_SURVEY_AGL] * 5}]

process_dict['F_CAL_IGRF_TEMPORAL_FILT_LEVEL'] = f_cal_igrf_temporal_filt

# Export map as a GeoTIFF
map = mu.export_map(out_dir = SURVEY_DIR,
                     location = ' '.join(map_title.split()),
                     date = log_df.datetime[0],
                     lats = interp_lats,
                     lons = interp_lons,
                     scalar = interp_scalar_LPF,
                     heights = interp_heights,
                     process_df = pd.DataFrame(process_dict),
                     process_app = PROCESS_APP,
                     stds = interp_std,
                     vector = None,
                     scalar_type = SCALAR_TYPE,
                     vector_type = VECTOR_TYPE,
                     scalar_var = np.nan,
                     vector_var = np.nan,
                     poc = POC,
                     flight_path = flight_path,
                     area_polys = area_polys,
                     osm_path = None,
                     level_type = 'No leveling',
                     tl_coeff_types = TL_COEFF_TYPES,
                     tl_coeffs = TL_C,
                     interp_type = interp_type,
                     final_filt_cut = FINAL_FILT_CUT,
                     final_filt_order = FINAL_FILT_ORDER)

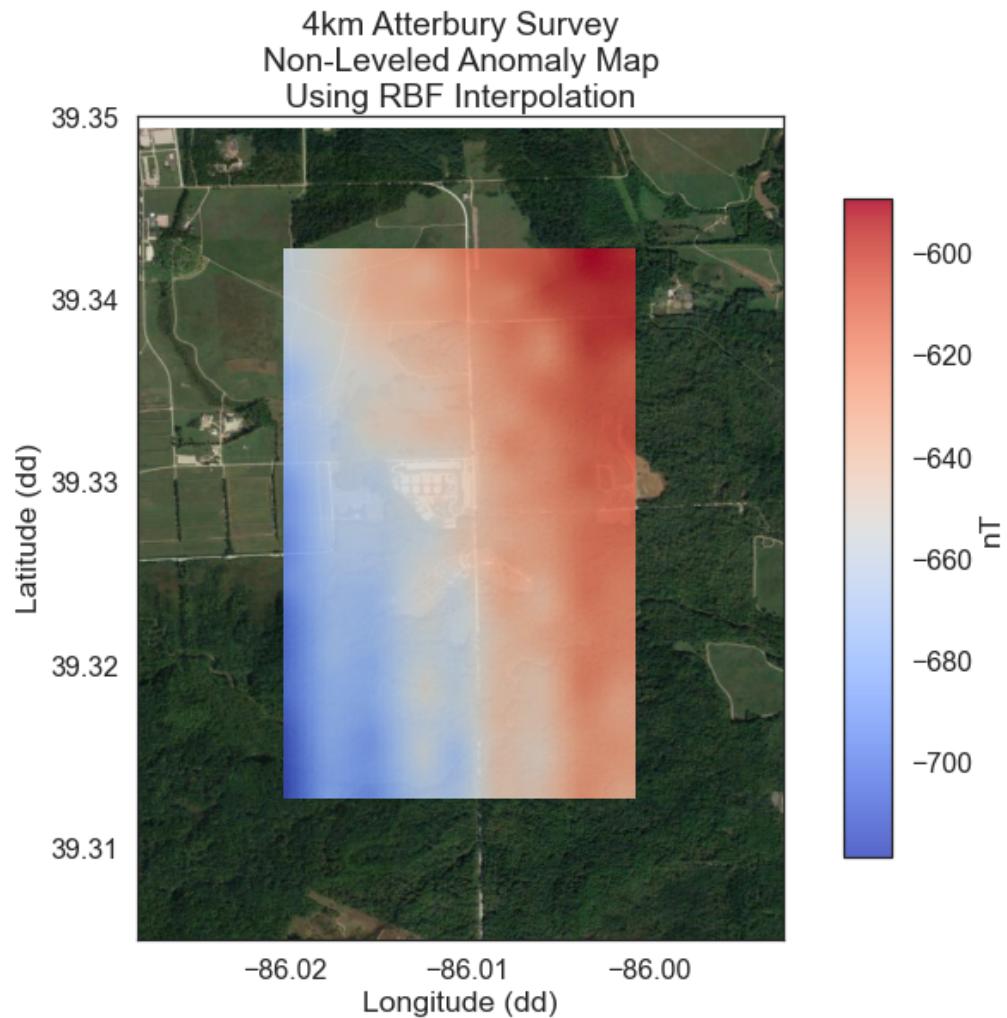
plt.figure()
map[0].plot(cmap=cm.coolwarm)
plt.title('Map From Exported GeoTIFF')

```

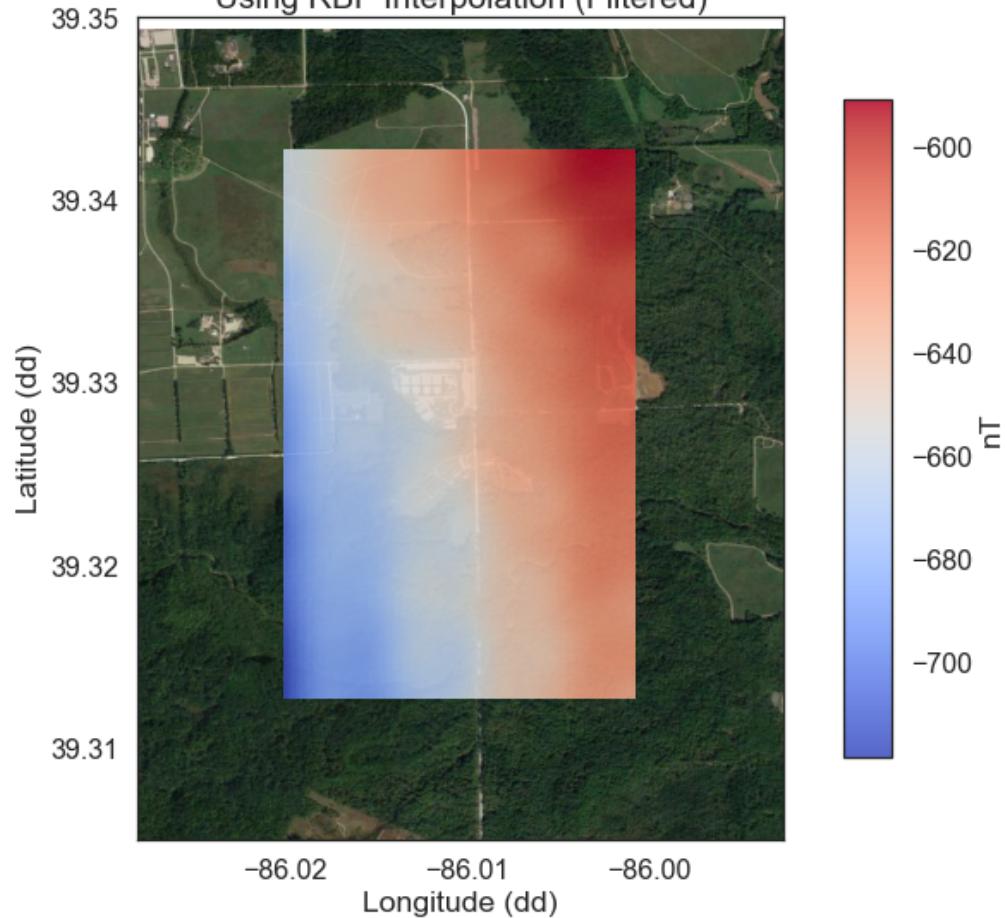
Interpolating survey anomaly data to map coordinates

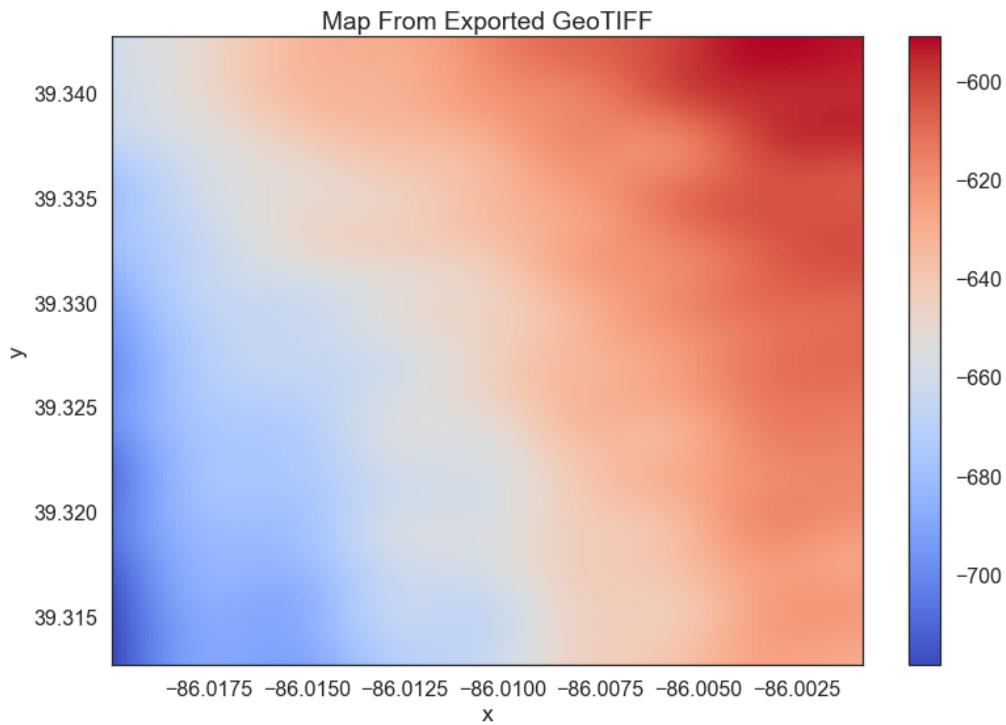
Running radial basis function (RBF) interpolation for all map pixels

[ ]: Text(0.5, 1.0, 'Map From Exported GeoTIFF')



4km Atterbury Survey  
Non-Leveled Anomaly Map  
Using RBF Interpolation (Filtered)



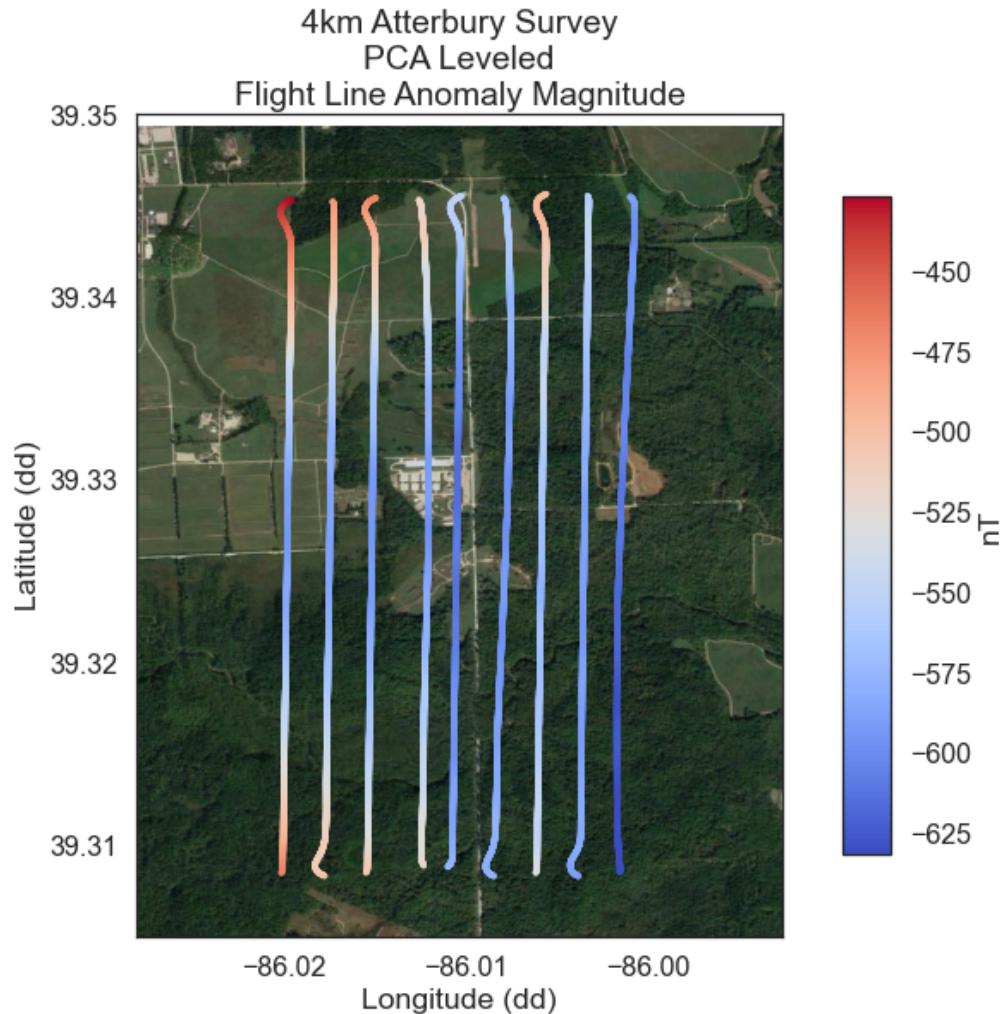


## 8 Apply PCA-Based Flight Line Leveling

```
[ ]: lvld_survey_df = pcaLvl.pca_lvl(survey_df = log_df,
                                      num_ptls = 2,
                                      ptl_locs = np.array([0.25, 0.75]))

plt.figure()
plt.title('PCA Leveled\nFlight Line Anomaly Magnitude'.format(SURVEY_NAME))
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.scatter(lvld_survey_df.LONG[lvld_survey_df.LINE_TYPE == 1],
                 lvld_survey_df.LAT[lvld_survey_df.LINE_TYPE == 1],
                 s=s,
                 c=lvld_survey_df.F[lvld_survey_df.LINE_TYPE == 1],
                 cmap=cmap)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)
```

[ ]: (39.305, 39.35)



## 9 Create Maps Based on PCA-Leveled Flight Lines

```
[ ]: interp_type = 'RBF'  
interp_df = pu.interp_flight_lines(anomaly_df  
    ↪lvld_survey_df[(lvld_survey_df.LAT >= MIN_MAP_LAT) & (lvld_survey_df.LAT <=  
    ↪MAX_MAP_LAT)],  
        dx = DX,  
        dy = DY,  
        max_terrain_msl = MAX_TERRAIN_MSL,
```

```

                buffer      = 0,
                interp_type = interp_type,
                neighbors   = None,
                skip_na_mask = True)

interp_lats    = interp_df['LAT']
interp_lons    = interp_df['LONG']
interp_scalar  = interp_df['F']
interp_heights = interp_df['ALT']
interp_std     = interp_df['STD']

interp_scalar_LPF = filt.lpf2(interp_scalar,
                               MAX_SURVEY_AGL,
                               DX,
                               DY)

map_title = '{}\nPCA Leveled Anomaly Map\nUsing {} Interpolation'.
            .format(SURVEY_NAME, interp_type)

plt.figure()
plt.title(map_title)
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.pcolormesh(interp_lons,
                     interp_lats,
                     interp_scalar,
                     cmap=cmap,
                     alpha=alpha)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)

map_title = '{}\nPCA Leveled Anomaly Map\nUsing {} Interpolation (Filtered)'.
            .format(SURVEY_NAME, interp_type)

plt.figure()
plt.title(map_title)
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.pcolormesh(interp_lons,
                     interp_lats,
                     interp_scalar_LPF,
                     cmap=cmap,
                     alpha=alpha)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)

```

```

plt.xlim(xlims)
plt.ylim(ylims)

process_dict['F_CAL_IGRF_TEMPORAL_FILT_LEVEL'] = lvld_survey_df.F

# Export map as a GeoTIFF
map = mu.export_map(out_dir
                     location = SURVEY_DIR,
                     date = ' '.join(map_title.split()),
                     lats = interp_lats,
                     lons = interp_lons,
                     scalar = interp_scalar_LPF,
                     heights = interp_heights,
                     process_df = pd.DataFrame(process_dict),
                     process_app = PROCESS_APP,
                     stds = interp_std,
                     vector = None,
                     scalar_type = SCALAR_TYPE,
                     vector_type = VECTOR_TYPE,
                     scalar_var = np.nan,
                     vector_var = np.nan,
                     poc = POC,
                     flight_path = flight_path,
                     area_polys = area_polys,
                     osm_path = None,
                     level_type = 'PCA pseudo tie line leveling',
                     tl_coeff_types = TL_COEFF_TYPES,
                     tl_coeffs = TL_C,
                     interp_type = interp_type,
                     final_filt_cut = FINAL_FILT_CUT,
                     final_filt_order = FINAL_FILT_ORDER)

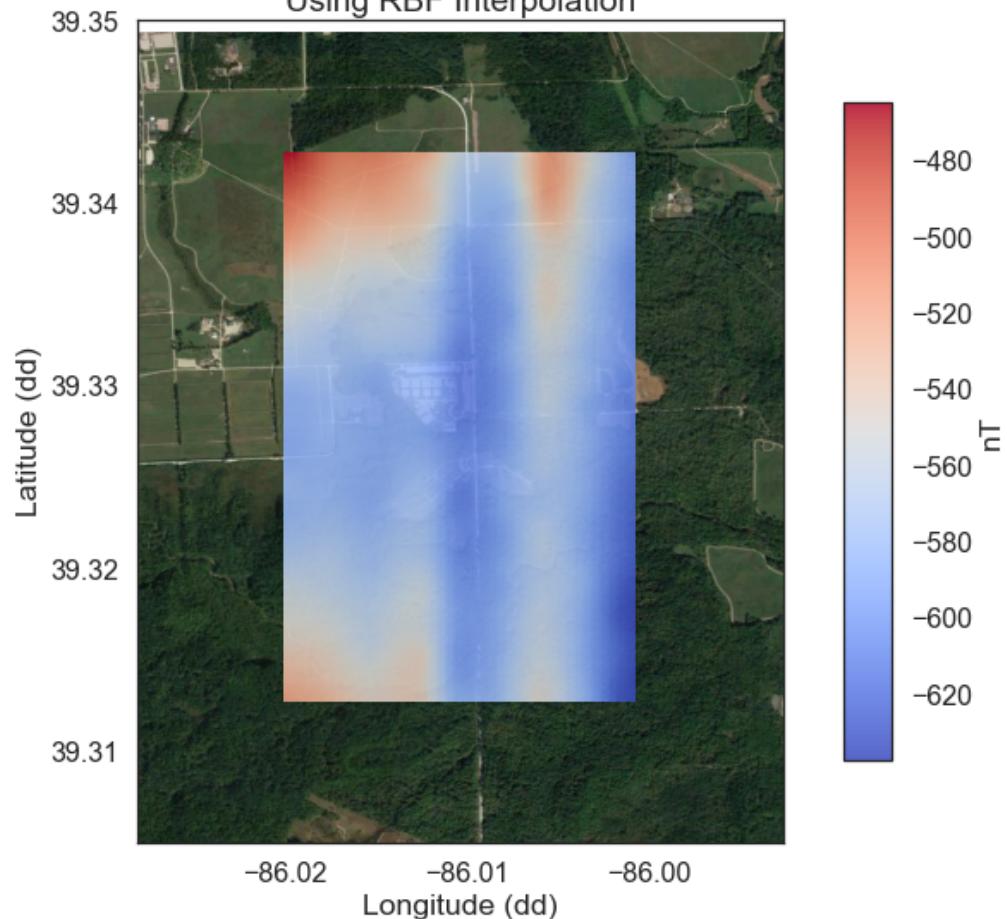
plt.figure()
map[0].plot(cmap=cm.coolwarm)
plt.title('Map From Exported GeoTIFF')

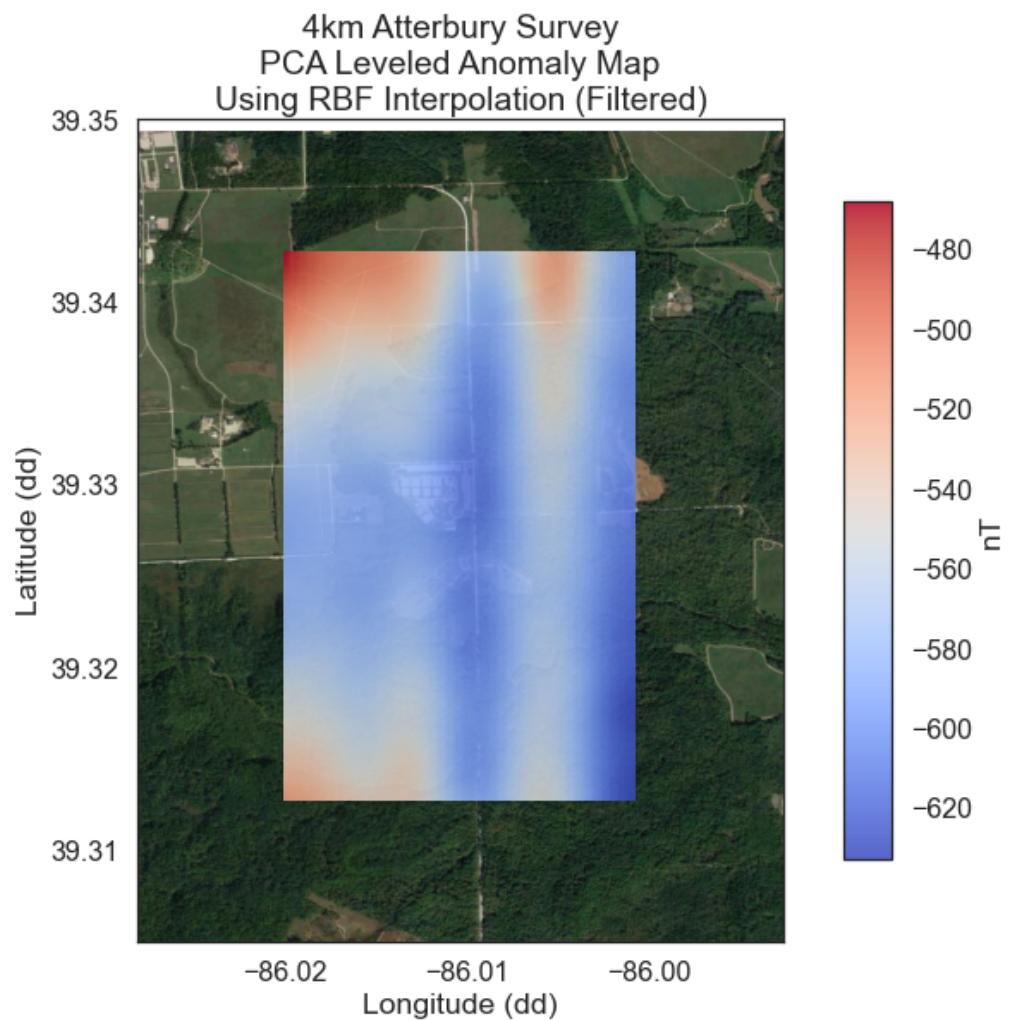
```

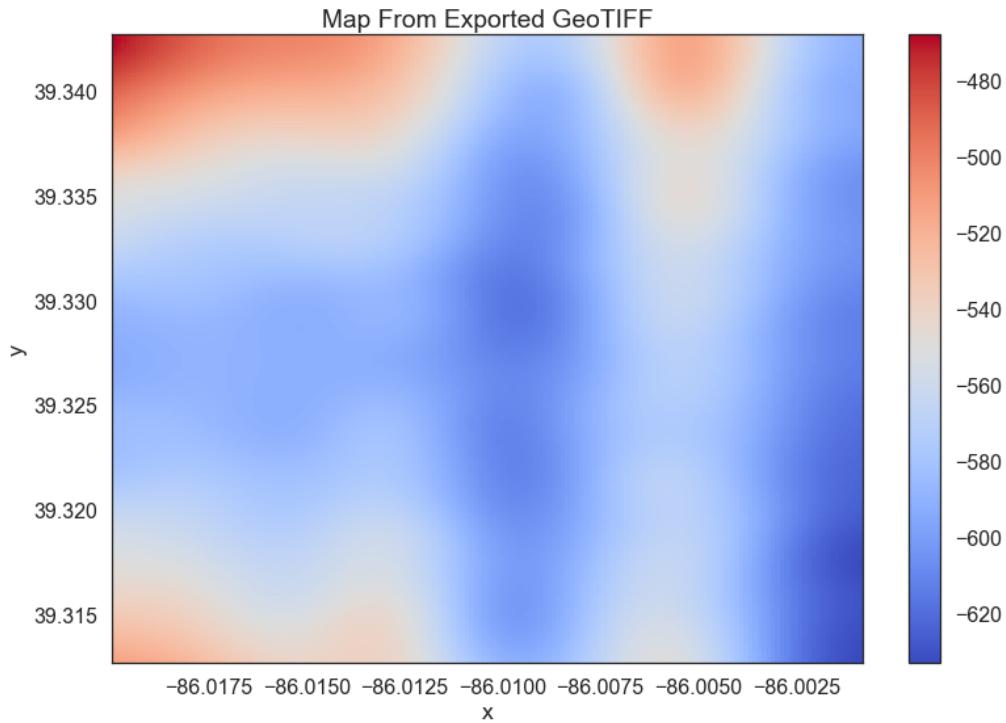
Interpolating survey anomaly data to map coordinates  
Running radial basis function (RBF) interpolation for all map pixels

[ ]: Text(0.5, 1.0, 'Map From Exported GeoTIFF')

4km Atterbury Survey  
PCA Leveled Anomaly Map  
Using RBF Interpolation





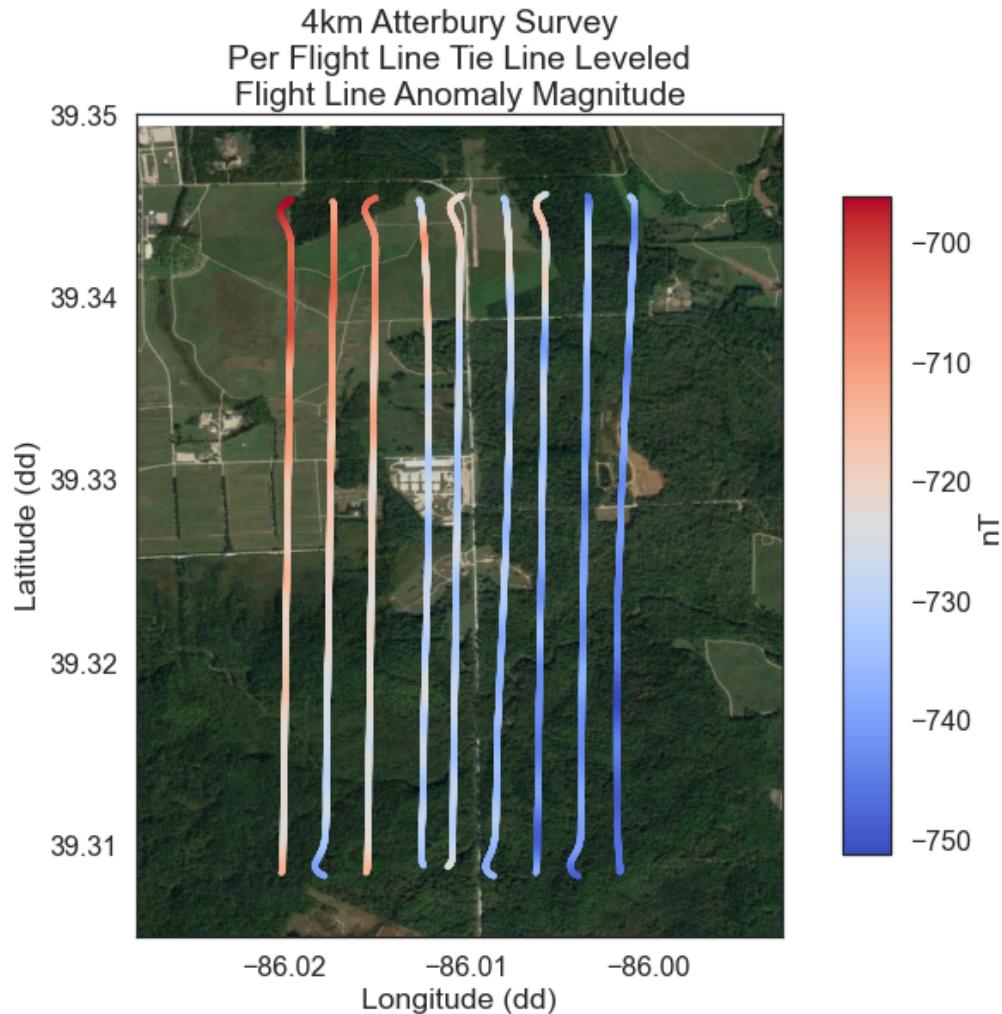


## 10 Apply Per Flight Line Tie Line-Based Flight Line Leveling

```
[ ]: lvld_survey_df = tieLvl.tie_lvl(survey_df = log_df,
                                     approach = 'lobf')

plt.figure()
plt.title('{}\\nPer Flight Line Tie Line Leveled\\nFlight Line Anomaly Magnitude'.
          format(SURVEY_NAME))
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.scatter(lvld_survey_df.LONG[lvld_survey_df.LINE_TYPE == 1],
                 lvld_survey_df.LAT[lvld_survey_df.LINE_TYPE == 1],
                 s=s,
                 c=lvld_survey_df.F[lvld_survey_df.LINE_TYPE == 1],
                 cmap=cmap)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)
```

[ ]: (39.305, 39.35)



## 11 Create Map Based on Per Flight Line Tie Line-Leveled Flight Lines

```
[ ]: interp_type = 'RBF'  
interp_df = pu.interp_flight_lines(anomaly_df  
    ↪ lvld_survey_df[(lvld_survey_df.LAT >= MIN_MAP_LAT) & (lvld_survey_df.LAT <=  
    ↪ MAX_MAP_LAT)],  
    dx = DX,  
    dy = DY,
```

```

        max_terrain_msl = MAX_TERRAIN_MSL,
        buffer          = 0,
        interp_type     = interp_type,
        neighbors       = None,
        skip_na_mask    = True)

interp_lats   = interp_df['LAT']
interp_lons   = interp_df['LONG']
interp_scalar = interp_df['F']
interp_heights = interp_df['ALT']
interp_std    = interp_df['STD']

interp_scalar_LPF = filt.lpf2(interp_scalar,
                               MAX_SURVEY_AGL,
                               DX,
                               DY)

map_title = '{}\nPer Flight Line Tie Line Leveled Anomaly Map\nUsing {}  
↳Interpolation'.format(SURVEY_NAME, interp_type)

plt.figure()
plt.title(map_title)
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.pcolormesh(interp_lons,
                     interp_lats,
                     interp_scalar,
                     cmap=cmap,
                     alpha=alpha)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)

map_title = '{}\nPer Flight Line Tie Line Leveled Anomaly Map\nUsing {}  
↳Interpolation (Filtered)'.format(SURVEY_NAME, interp_type)

plt.figure()
plt.title(map_title)
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.pcolormesh(interp_lons,
                     interp_lats,
                     interp_scalar_LPF,
                     cmap=cmap,
                     alpha=alpha)

```

```

plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)

process_dict['F_CAL_IGRF_TEMPORAL_FILT_LEVEL'] = lvld_survey_df.F

# Export map as a GeoTIFF
map = mu.export_map(out_dir
                     location      = SURVEY_DIR,
                     date         = ' '.join(map_title.split()),
                     lats          = interp_lats,
                     lons          = interp_lons,
                     scalar        = interp_scalar_LPF,
                     heights       = interp_heights,
                     process_df    = pd.DataFrame(process_dict),
                     process_app   = PROCESS_APP,
                     stds          = interp_std,
                     vector         = None,
                     scalar_type   = SCALAR_TYPE,
                     vector_type   = VECTOR_TYPE,
                     scalar_var    = np.nan,
                     vector_var    = np.nan,
                     poc            = POC,
                     flight_path   = flight_path,
                     area_polys    = area_polys,
                     osm_path       = None,
                     level_type    = 'Per flight line tie line leveling',
                     tl_coeff_types = TL_COEFF_TYPES,
                     tl_coeffs     = TL_C,
                     interp_type   = interp_type,
                     final_filt_cut = FINAL_FILT_CUT,
                     final_filt_order = FINAL_FILT_ORDER)

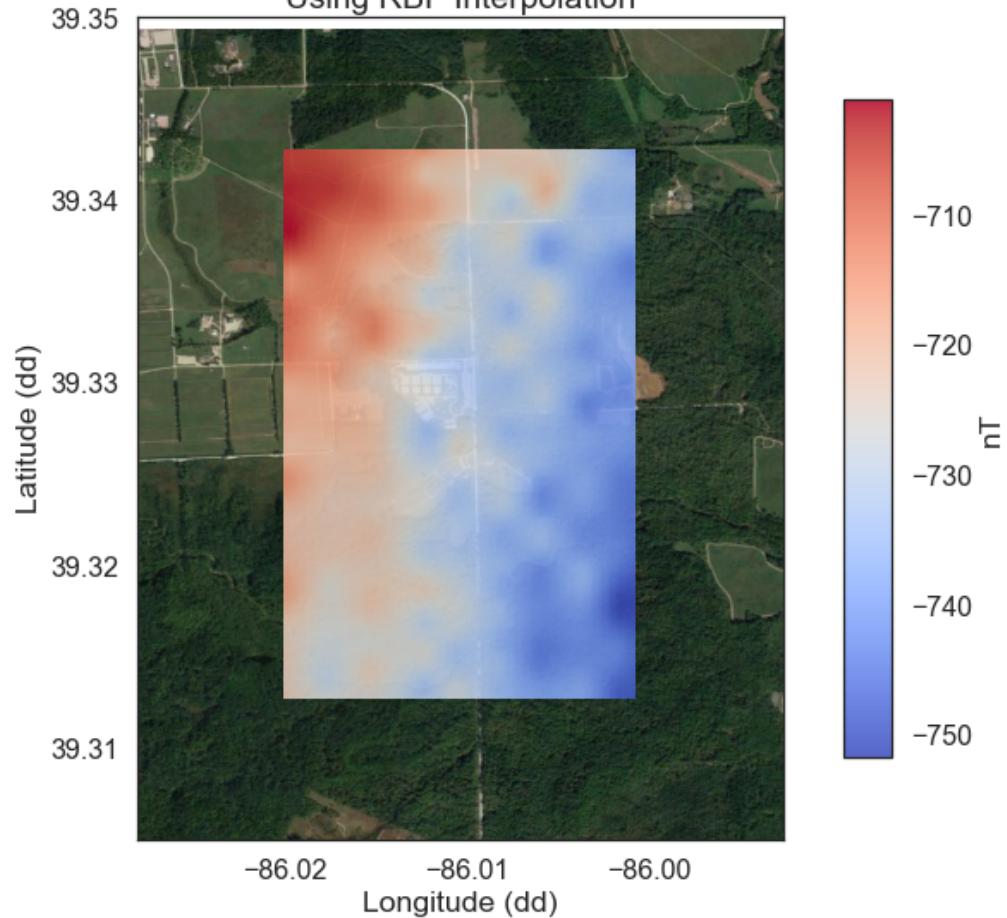
plt.figure()
map[0].plot(cmap=cm.coolwarm)
plt.title('Map From Exported GeoTIFF')

```

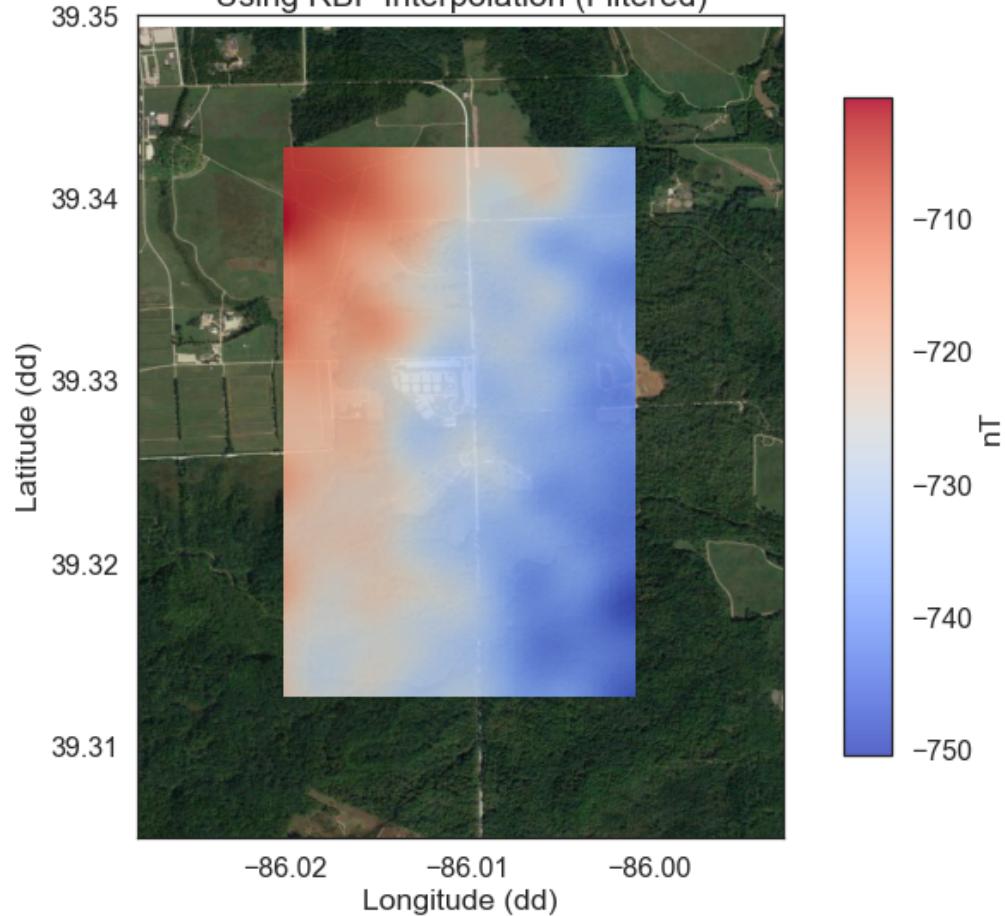
Interpolating survey anomaly data to map coordinates  
Running radial basis function (RBF) interpolation for all map pixels

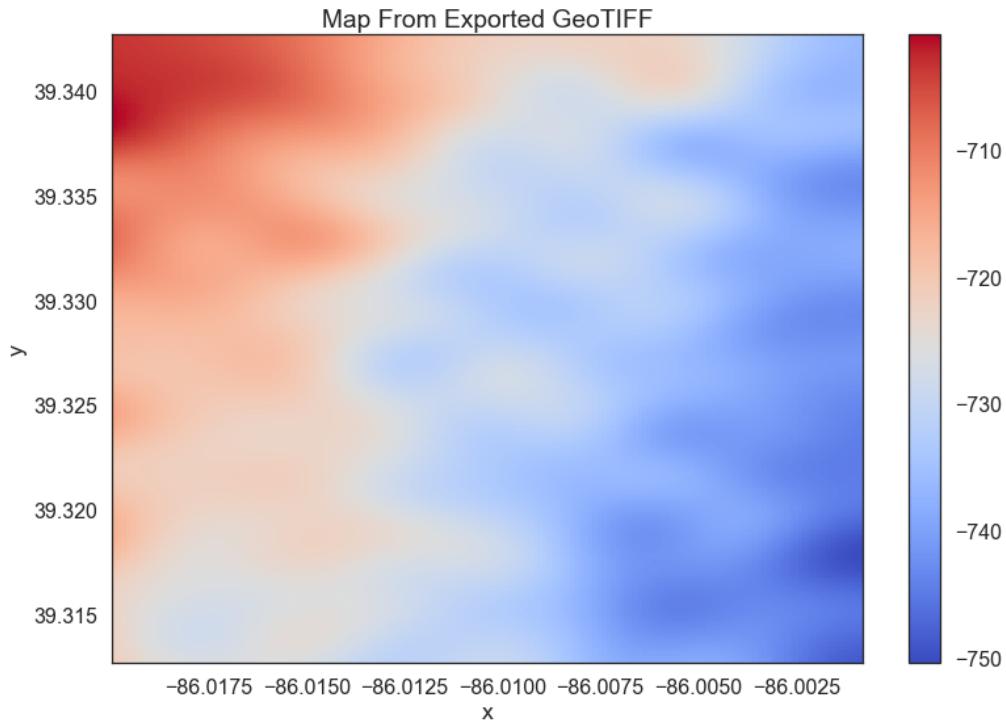
[ ]: Text(0.5, 1.0, 'Map From Exported GeoTIFF')

4km Atterbury Survey  
Per Flight Line Tie Line Leveled Anomaly Map  
Using RBF Interpolation



4km Atterbury Survey  
Per Flight Line Tie Line Leveled Anomaly Map  
Using RBF Interpolation (Filtered)



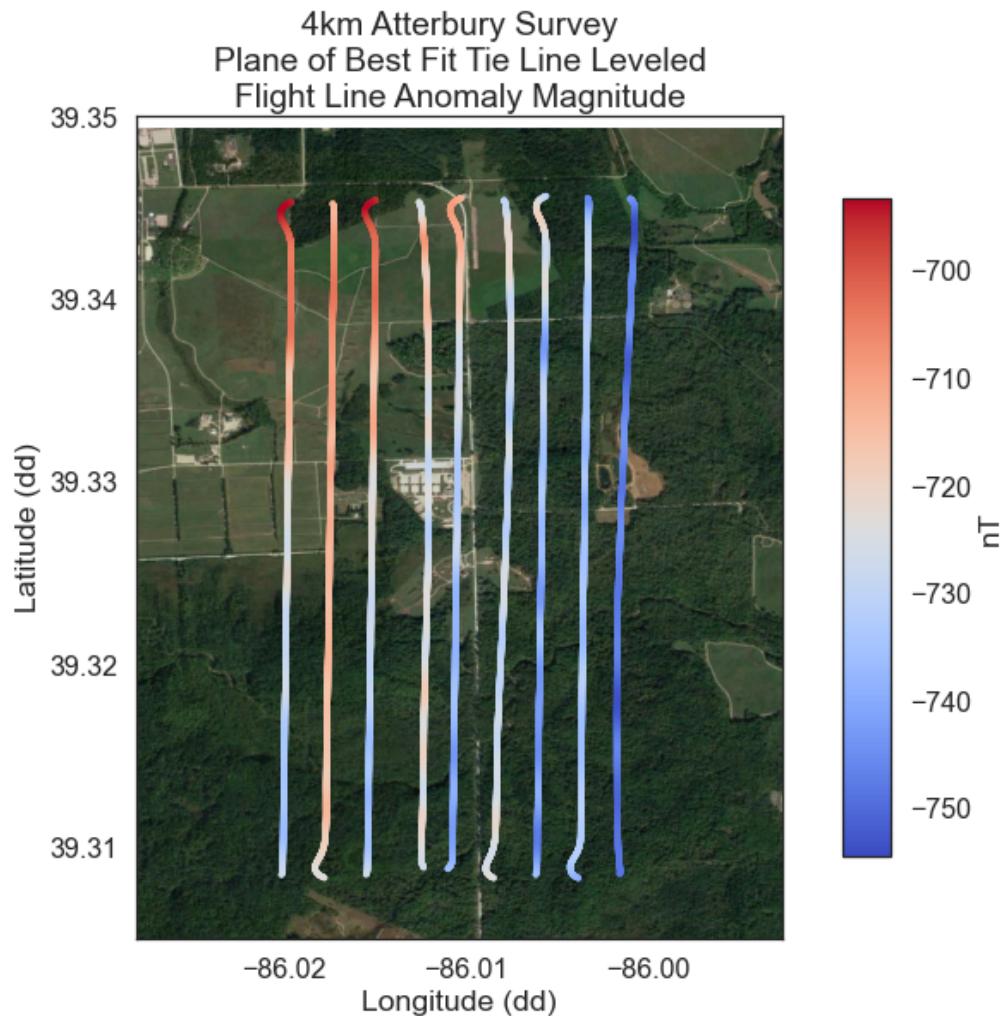


## 12 Apply Plane of Best Fit Tie Line-Based Flight Line Leveling

```
[ ]: lvld_survey_df = tieLvl.tie_lvl(survey_df = log_df,
                                     approach = 'lsq')

plt.figure()
plt.title('{} \nPlane of Best Fit Tie Line Leveled \nFlight Line Anomaly\n Magnitude'.format(SURVEY_NAME))
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.scatter(lvld_survey_df.LONG[lvld_survey_df.LINE_TYPE == 1],
                 lvld_survey_df.LAT[lvld_survey_df.LINE_TYPE == 1],
                 s=s,
                 c=lvld_survey_df.F[lvld_survey_df.LINE_TYPE == 1],
                 cmap=cmap)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)
```

[ ]: (39.305, 39.35)



### 13 Create Map Based on Plane of Best Fit Tie Line-Leveled Flight Lines

```
[ ]: interp_type = 'RBF'  
interp_df = pu.interp_flight_lines(anomaly_df  
    ↪ lvld_survey_df[(lvld_survey_df.LAT >= MIN_MAP_LAT) & (lvld_survey_df.LAT <=  
    ↪ MAX_MAP_LAT)],  
                                    dx = DX,  
                                    dy = DY,
```

```

        max_terrain_msl = MAX_TERRAIN_MSL,
        buffer          = 0,
        interp_type     = interp_type,
        neighbors       = None,
        skip_na_mask    = True)

interp_lats   = interp_df['LAT']
interp_lons   = interp_df['LONG']
interp_scalar = interp_df['F']
interp_heights = interp_df['ALT']
interp_std    = interp_df['STD']

interp_scalar_LPF = filt.lpf2(interp_scalar,
                               MAX_SURVEY_AGL,
                               DX,
                               DY)

map_title = '{}\nPlane of Best Fit Tie Line Leveled Anomaly Map\nUsing {}_{}'
            .format(SURVEY_NAME, interp_type)

plt.figure()
plt.title(map_title)
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.pcolormesh(interp_lons,
                     interp_lats,
                     interp_scalar,
                     cmap=cmap,
                     alpha=alpha)
plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)

map_title = '{}\nPlane of Best Fit Tie Line Leveled Anomaly Map\nUsing {}_{}'
            .format(SURVEY_NAME, interp_type)

plt.figure()
plt.title(map_title)
plt.xlabel('Longitude (dd)')
plt.ylabel('Latitude (dd)')
plt.imshow(img, extent=GMAP_EXTENT)
cb = plt.pcolormesh(interp_lons,
                     interp_lats,
                     interp_scalar_LPF,
                     cmap=cmap,
                     alpha=alpha)

```

```

plt.colorbar(cb, label='nT', shrink=shrink, aspect=aspect)
plt.xlim(xlims)
plt.ylim(ylims)

process_dict['F_CAL_IGRF_TEMPORAL_FILT_LEVEL'] = lvld_survey_df.F

# Export map as a GeoTIFF
map = mu.export_map(out_dir
                     location      = SURVEY_DIR,
                     date         = ' '.join(map_title.split()),
                     lats          = interp_lats,
                     lons          = interp_lons,
                     scalar        = interp_scalar_LPF,
                     heights       = interp_heights,
                     process_df    = pd.DataFrame(process_dict),
                     process_app   = PROCESS_APP,
                     stds          = interp_std,
                     vector         = None,
                     scalar_type   = SCALAR_TYPE,
                     vector_type   = VECTOR_TYPE,
                     scalar_var    = np.nan,
                     vector_var    = np.nan,
                     poc            = POC,
                     flight_path   = flight_path,
                     area_polys    = area_polys,
                     osm_path       = None,
                     level_type    = 'Plane of best fit tie line leveling',
                     tl_coeff_types = TL_COEFF_TYPES,
                     tl_coeffs     = TL_C,
                     interp_type   = interp_type,
                     final_filt_cut = FINAL_FILT_CUT,
                     final_filt_order = FINAL_FILT_ORDER)

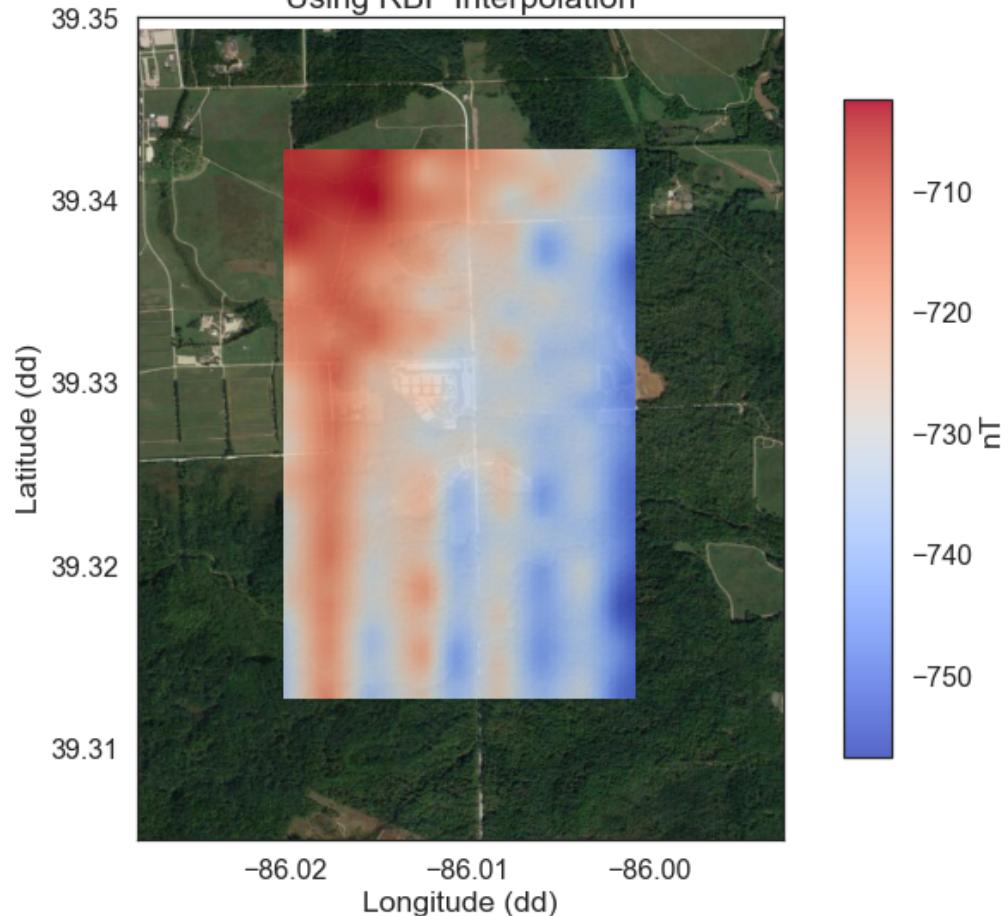
plt.figure()
map[0].plot(cmap=cm.coolwarm)
plt.title('Map From Exported GeoTIFF')

```

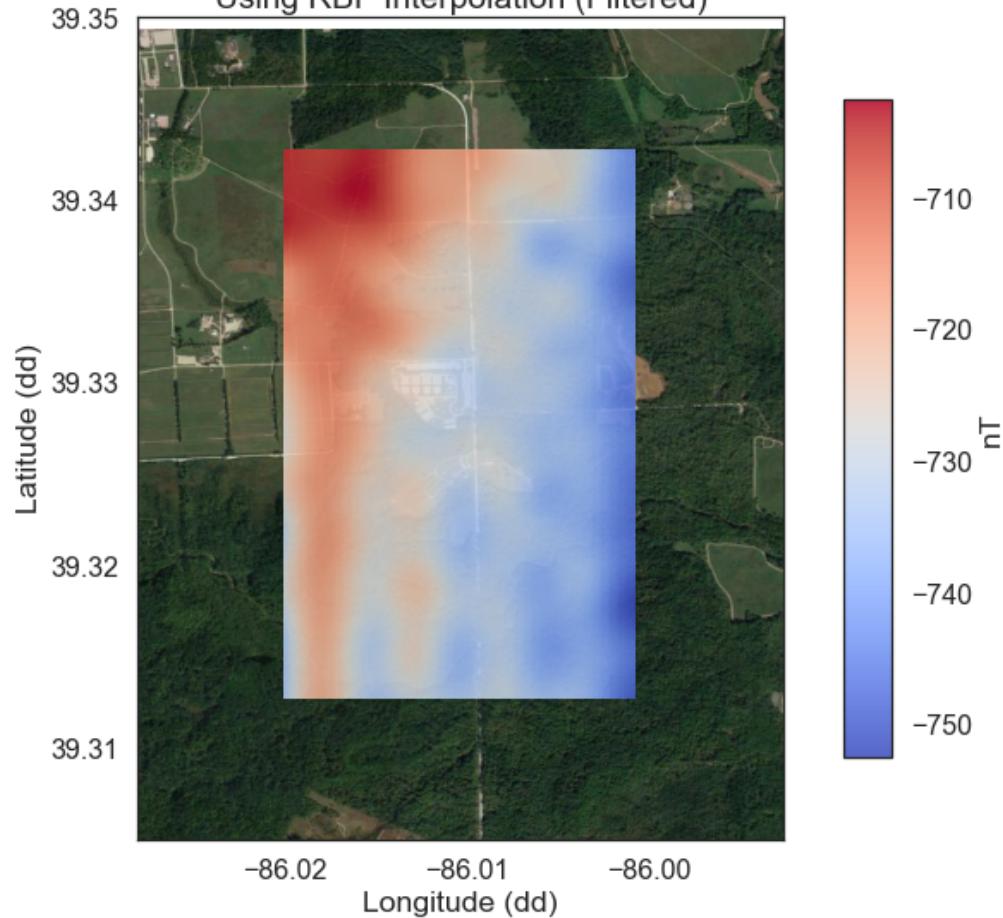
Interpolating survey anomaly data to map coordinates  
Running radial basis function (RBF) interpolation for all map pixels

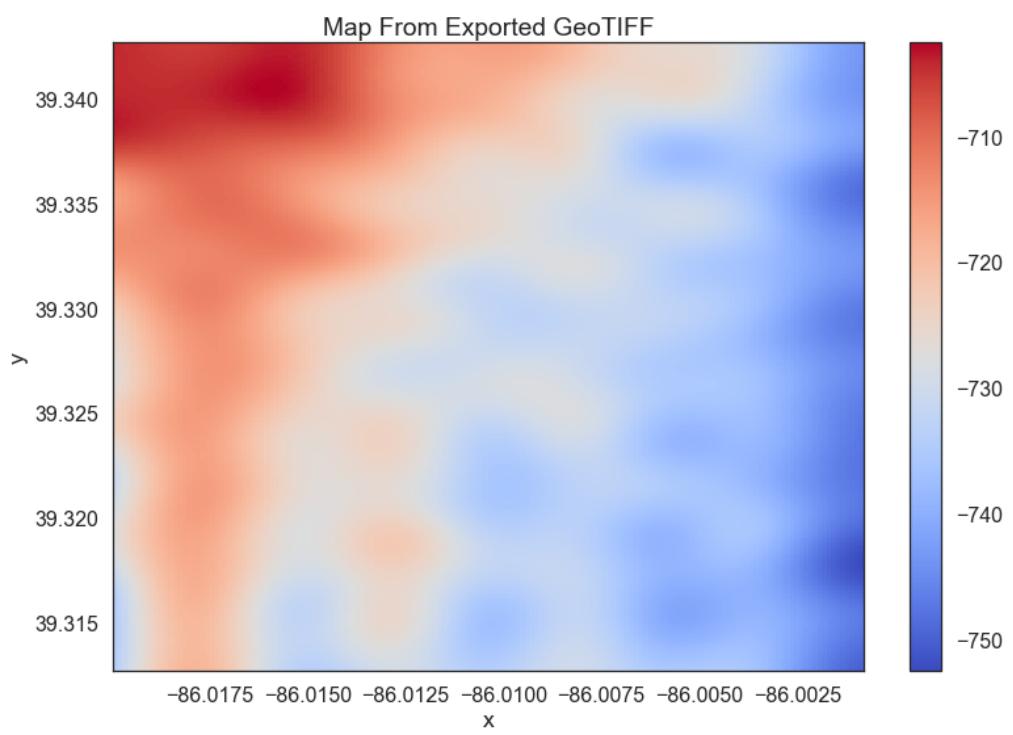
[ ]: Text(0.5, 1.0, 'Map From Exported GeoTIFF')

4km Atterbury Survey  
Plane of Best Fit Tie Line Leveled Anomaly Map  
Using RBF Interpolation



4km Atterbury Survey  
Plane of Best Fit Tie Line Leveled Anomaly Map  
Using RBF Interpolation (Filtered)





## compare

January 3, 2023

```
[ ]: from copy import deepcopy

import matplotlib.pyplot as plt
import numpy as np
import rioxarray as rxr
from matplotlib import cm
from numpy import fft
from scipy import signal
from scipy.interpolate import interp1d
from scipy.signal import butter, sosfiltfilt

%matplotlib inline
plt.rcParams['figure.figsize'] = (30, 15) # (w, h)
plt.style.use(['seaborn-poster', 'seaborn-white'])

shrink      = 1.0
aspect       = 20 * 0.7
cmap         = cm.coolwarm
corr_lims   = [-1, 1]

SCALAR_BAND = 0
DX_BAND     = 6
DY_BAND     = 7

def filt(data: np.ndarray,
          cutoff: float,
          fs:      float,
          btype:   str,
          order:   int=6,
          axis:    int=-1) -> np.ndarray:
    sos = butter(order,
                 cutoff,
                 fs=fs,
                 btype=btype,
```

```

        analog=False,
        output='sos')

    return sosfiltfilt(sos,
                        data,
                        axis=axis)

def lpf(data: np.ndarray,
        cutoff: float,
        fs: float,
        order: int=6,
        axis: int=-1) -> np.ndarray:
    return filt(data,
                cutoff,
                fs,
                'low',
                order,
                axis)

def hpf(data: np.ndarray,
        cutoff: float,
        fs: float,
        order: int=6,
        axis: int=-1) -> np.ndarray:
    return filt(data,
                cutoff,
                fs,
                'high',
                order,
                axis)

def hpf2(data: np.ndarray,
         cutoff: float,
         dx: float,
         dy: float,
         order: int=6) -> np.ndarray:
    data_copy = deepcopy(data)

    # Filter in Y direction
    data_hpf_y = hpf(data = data_copy,
                      cutoff = 1/cutoff,
                      fs = 1/dy,
                      order = order,
                      axis = 0)

    # Filter in X direction
    data_hpf = hpf(data = data_hpf_y,

```

```

        cutoff = 1/cutoff,
        fs      = 1/dx,
        order   = order,
        axis    = 1)

return data_hpf

def find_correagation_fom(map:      rxr.rioxarray.raster_dataset.xarray.DataArray,
                           cutoffs: list[float],
                           dp:       float,
                           band:    int=0,
                           axis:    int=1) -> float:
    map_cpy = deepcopy(map)
    data    = map_cpy[band].data

    high_cutoff = 1 / cutoffs[0]
    low_cutoff  = 1 / cutoffs[1]

    if axis == 1:
        rev_axis = 0
    else:
        rev_axis = 1

    data_zero_mean = data - data.mean()
    data_hpf      = hpf(data_zero_mean, low_cutoff, dp, 10, axis)
    hpf_zero_mean = data_hpf - data_hpf.mean()
    hpf_compressed = np.mean(hpf_zero_mean, axis=rev_axis)

    hpf_compressed_zero_mean = hpf_compressed - hpf_compressed.mean()
    hpf_compressed_fft      = np.fft.fft(np.
    ↪nan_to_num(hpf_compressed_zero_mean))

    fft_mag   = np.abs(hpf_compressed_fft)
    n         = hpf_compressed_zero_mean.size
    fft_mag   = fft_mag[:int(n/2)]
    fft_freqs = np.fft.fftfreq(n, d=dp)[:int(n/2)] # Only use positive freqs
    f         = interp1d(fft_freqs, 20 * np.log10(fft_mag))

    corrugation_fom = 1 / f(np.linspace(low_cutoff, high_cutoff, 100)).max()

    # plt.figure()
    # map[band].plot(cmap=cm.coolwarm)

    # plt.figure()
    # plt.imshow(data_zero_mean, cmap=cm.coolwarm)
    # plt.colorbar()

```

```

# plt.figure()
# plt.imshow(hpf_zero_mean, cmap=cm.coolwarm)
# plt.colorbar()

# plt.figure()
# plt.plot(hpf_compressed)

# plt.figure()
# plt.plot(fft_freqs, fft_mag)
# plt.plot(np.linspace(low_cutoff, high_cutoff, 100), f(np.
linspace(low_cutoff, high_cutoff, 100)))
# plt.axhline(high_cutoff)
# plt.axhline(low_cutoff)

return corrugation_fom

def compare_maps(ref_map, comp_map, ref_map_name='', comp_map_name=''):
    print('*' * 50)
    print('2D Correlation Between\n{}\nand\n{}\nScalar Magnitudes'.
format(ref_map_name, comp_map_name))

    # Copy in data
    ref_map_cpy = deepcopy(ref_map)
    comp_map_cpy = deepcopy(comp_map).interp(x=ref_map_cpy.x, y=ref_map_cpy.y)

    # Set original, reference coords
    x = ref_map_cpy.x.data
    y = ref_map_cpy.y.data

    # Interpolate comparison map to reference map resolution
    comp_map_cpy = comp_map_cpy.interp(x=x, y=y)

    # Alias reference map band data
    ref_map_scalar_data = ref_map_cpy[SCALAR_BAND].data
    ref_map_dx_data     = ref_map_cpy[DX_BAND].data
    ref_map_dy_data     = ref_map_cpy[DY_BAND].data

    # Alias comparison map band data
    comp_map_scalar_data = comp_map_cpy[SCALAR_BAND].data
    comp_map_dx_data     = comp_map_cpy[DX_BAND].data
    comp_map_dy_data     = comp_map_cpy[DY_BAND].data

    # Mask out all Nan pixels
    mask = np.isfinite(ref_map_scalar_data) & np.isfinite(comp_map_scalar_data)
    x_mask = (mask == 1).any(axis=0)
    y_mask = (mask == 1).any(axis=1)

```

```

# Crop coords
x = x[x_mask]
y = y[y_mask]

# Find the correlation lags
x_lags = signal.correlation_lags(len(x), len(x), mode='same')
y_lags = signal.correlation_lags(len(y), len(y), mode='same')

# Find correlation x and y indices with no lag
x_zero = np.where(x_lags == 0)[0].item()
y_zero = np.where(y_lags == 0)[0].item()

# Crop reference map bands
ref_map_scalar_data = ref_map_scalar_data[mask].reshape((len(y), len(x)))
ref_map_dx_data = ref_map_dx_data[mask].reshape((len(y), len(x)))
ref_map_dy_data = ref_map_dy_data[mask].reshape((len(y), len(x)))

# Crop comparison map bands
comp_map_scalar_data = comp_map_scalar_data[mask].reshape((len(y), len(x)))
comp_map_dx_data = comp_map_dx_data[mask].reshape((len(y), len(x)))
comp_map_dy_data = comp_map_dy_data[mask].reshape((len(y), len(x)))

# Calc zero-mean reference bands
ref_map_scalar_0mean_data = ref_map_scalar_data - ref_map_scalar_data.mean()
ref_map_dx_0mean_data = ref_map_dx_data - ref_map_dx_data.mean()
ref_map_dy_0mean_data = ref_map_dy_data - ref_map_dy_data.mean()

# Calc zero-mean comparison bands
comp_map_scalar_0mean_data = comp_map_scalar_data - comp_map_scalar_data.mean()
comp_map_dx_0mean_data = comp_map_dx_data - comp_map_dx_data.mean()
comp_map_dy_0mean_data = comp_map_dy_data - comp_map_dy_data.mean()

# Calculate Pearson correlation number for scalar band
cref = signal.correlate2d(ref_map_scalar_0mean_data, □
ref_map_scalar_0mean_data, mode='same')
ccomp = signal.correlate2d(comp_map_scalar_0mean_data, □
comp_map_scalar_0mean_data, mode='same')
ccros = signal.correlate2d(ref_map_scalar_0mean_data, □
comp_map_scalar_0mean_data, mode='same')

ps = ccros / (np.sqrt(cref.max()) * np.sqrt(ccomp.max()))
p_max = ccros.max() / (np.sqrt(cref.max()) * np.sqrt(ccomp.max()))
p_max_loc = [x_lags[np.where(ccros == ccros.max())[1].item()],
             y_lags[np.where(ccros == ccros.max())[0].item()]]
print('Max Scalar Pearson:', p_max)
print('Max Scalar Pearson Lags:', p_max_loc)

```

```

print('Zero Lag Scalar Pearson:', ps[y_zero, x_zero])

plt.figure()
cb = plt.pcolormesh(x_lags, y_lags, ps, cmap=cm.coolwarm)
plt.title('2D Correlation Between\n{}\\n{}\\nScalar Magnitudes'.
format(ref_map_name, comp_map_name))
plt.xlabel('East-West Lags')
plt.ylabel('North-South Lags')
plt.colorbar(cb, shrink=shrink, aspect=aspect)
plt.clim(corr_lims)
plt.scatter(*p_max_loc, s=500, c='g', marker='*', label='Global Maximum')
plt.legend()

# Calculate Pearson correlation number for scalar band
cref = signal.correlate2d(ref_map_dx_0mean_data, ref_map_dx_0mean_data, mode='same')
ccomp = signal.correlate2d(comp_map_dx_0mean_data, comp_map_dx_0mean_data, mode='same')
ccros = signal.correlate2d(ref_map_dx_0mean_data, comp_map_dx_0mean_data, mode='same')

ps = ccros / (np.sqrt(cref.max()) * np.sqrt(ccomp.max()))
p_max = ccros.max() / (np.sqrt(cref.max()) * np.sqrt(ccomp.max()))
p_max_loc = [x_lags[np.where(ccros == ccros.max())[1].item()],
             y_lags[np.where(ccros == ccros.max())[0].item()]]
print('Max dX Pearson:', p_max)
print('Max dX Pearson Lags:', p_max_loc)
print('Zero Lag dX Pearson:', ps[y_zero, x_zero])

plt.figure()
cb = plt.pcolormesh(x_lags, y_lags, ps, cmap=cm.coolwarm)
plt.title('2D Correlation Between\n{}\\n{}\\nEasterly Gradients'.
format(ref_map_name, comp_map_name))
plt.xlabel('East-West Lags')
plt.ylabel('North-South Lags')
plt.colorbar(cb, shrink=shrink, aspect=aspect)
plt.clim(corr_lims)
plt.scatter(*p_max_loc, s=500, c='g', marker='*', label='Global Maximum')
plt.legend()

# Calculate Pearson correlation number for scalar band
cref = signal.correlate2d(ref_map_dy_0mean_data, ref_map_dy_0mean_data, mode='same')
ccomp = signal.correlate2d(comp_map_dy_0mean_data, comp_map_dy_0mean_data, mode='same')

```

```

ccros = signal.correlate2d(ref_map_dy_0mean_data, comp_map_dy_0mean_data, mode='same')

ps      = ccros / (np.sqrt(cref.max()) * np.sqrt(ccomp.max()))
p_max   = ccros.max() / (np.sqrt(cref.max()) * np.sqrt(ccomp.max()))
p_max_loc = [x_lags[np.where(ccros == ccros.max())[1].item()],
              y_lags[np.where(ccros == ccros.max())[0].item()]]
print('Max dY Pearson:', p_max)
print('Max dY Pearson Lags:', p_max_loc)
print('Zero Lag dY Pearson:', ps[y_zero, x_zero])

plt.figure()
cb = plt.pcolormesh(x_lags, y_lags, ps, cmap=cm.coolwarm)
plt.title('2D Correlation Between\n{}\\n{}\\nNortherly Gradients'.
           format(ref_map_name, comp_map_name))
plt.xlabel('East-West Lags')
plt.ylabel('North-South Lags')
plt.colorbar(cb, shrink=shrink, aspect=aspect)
plt.clim(corr_lims)
plt.scatter(*p_max_loc, s=500, c='g', marker='*', label='Global Maximum')
plt.legend()

```

## 1 Read in Maps

```

[ ]: _1km_no_lvl_survey_1      = rxr.open_rasterio('1km Atterbury Survey (First Attempt) Non-Leveled Anomaly Map Using RBF Interpolation' (Filtered)_619m_2022_11_3_0.tif')
_1km_pca_lvl_survey_1      = rxr.open_rasterio('1km Atterbury Survey (First Attempt) PCA Leveled Anomaly Map Using RBF Interpolation' (Filtered)_619m_2022_11_3_0.tif')
_1km_per_flt_lvl_survey_1 = rxr.open_rasterio('1km Atterbury Survey (First Attempt) Per Flight Line Tie Line Leveled Anomaly Map Using RBF Interpolation (Filtered)_619m_2022_11_3_0.tif')
_1km_plane_lvl_survey_1   = rxr.open_rasterio('1km Atterbury Survey (First Attempt) Plane of Best Fit Tie Line Leveled Anomaly Map Using RBF Interpolation (Filtered)_619m_2022_11_3_0.tif')

_1km_no_lvl_survey_2      = rxr.open_rasterio('1km Atterbury Survey (Second Attempt) Non-Leveled Anomaly Map Using RBF Interpolation' (Filtered)_619m_2022_11_8_0.tif')
_1km_pca_lvl_survey_2      = rxr.open_rasterio('1km Atterbury Survey (Second Attempt) PCA Leveled Anomaly Map Using RBF Interpolation' (Filtered)_619m_2022_11_8_0.tif')
_1km_per_flt_lvl_survey_2 = rxr.open_rasterio('1km Atterbury Survey (Second Attempt) Per Flight Line Tie Line Leveled Anomaly Map Using RBF Interpolation (Filtered)_619m_2022_11_8_0.tif')

```

```

_1km_plane_lvl_survey_2 = rxr.open_rasterio('1km Atterbury Survey (Second
    ↪Attempt) Plane of Best Fit Tie Line Leveled Anomaly Map Using RBF
    ↪Interpolation (Filtered)_619m_2022_11_8_0.tif')

_2km_no_lvl_survey      = rxr.open_rasterio('2km Atterbury Survey Non-Leveled
    ↪Anomaly Map Using RBF Interpolation (Filtered)_617m_2022_11_8_0.tif')
_2km_pca_lvl_survey     = rxr.open_rasterio('2km Atterbury Survey PCA Leveled
    ↪Anomaly Map Using RBF Interpolation (Filtered)_617m_2022_11_8_0.tif')
_2km_per_flt_lvl_survey = rxr.open_rasterio('2km Atterbury Survey Per Flight
    ↪Line Tie Line Leveled Anomaly Map Using RBF Interpolation
    ↪(Filtered)_617m_2022_11_8_0.tif')
_2km_plane_lvl_survey   = rxr.open_rasterio('2km Atterbury Survey Plane of Best
    ↪Fit Tie Line Leveled Anomaly Map Using RBF Interpolation
    ↪(Filtered)_617m_2022_11_8_0.tif')

_4km_no_lvl_survey      = rxr.open_rasterio('4km Atterbury Survey Non-Leveled
    ↪Anomaly Map Using RBF Interpolation (Filtered)_615m_2022_11_8_0.tif')
_4km_pca_lvl_survey     = rxr.open_rasterio('4km Atterbury Survey PCA Leveled
    ↪Anomaly Map Using RBF Interpolation (Filtered)_615m_2022_11_8_0.tif')
_4km_per_flt_lvl_survey = rxr.open_rasterio('4km Atterbury Survey Per Flight
    ↪Line Tie Line Leveled Anomaly Map Using RBF Interpolation
    ↪(Filtered)_615m_2022_11_8_0.tif')
_4km_plane_lvl_survey   = rxr.open_rasterio('4km Atterbury Survey Plane of Best
    ↪Fit Tie Line Leveled Anomaly Map Using RBF Interpolation
    ↪(Filtered)_615m_2022_11_8_0.tif')

```

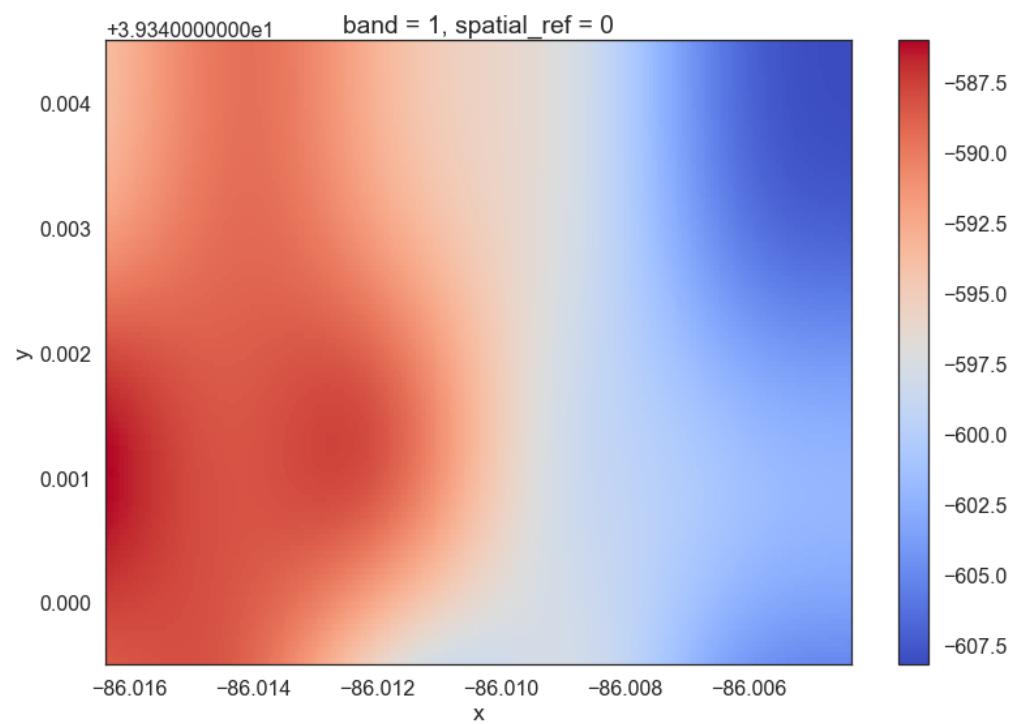
```

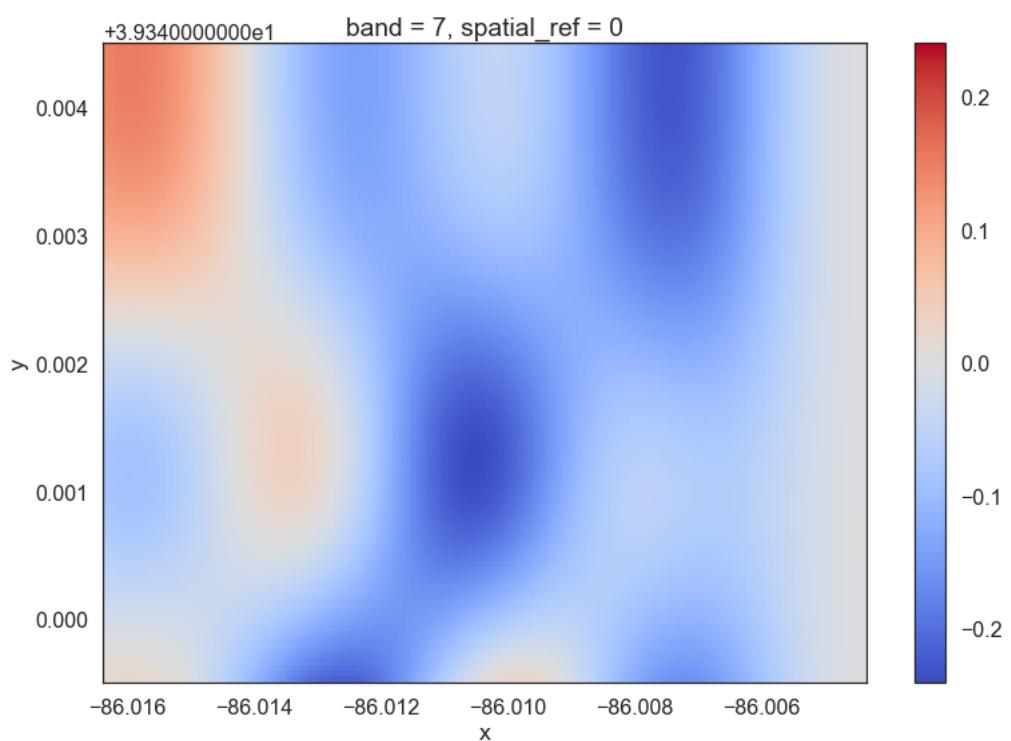
[ ]: plt.figure()
_1km_plane_lvl_survey_2[SCALAR_BAND].plot(cmap=cm.coolwarm)
plt.figure()
_1km_plane_lvl_survey_2[DX_BAND].plot(cmap=cm.coolwarm)
plt.figure()
_1km_plane_lvl_survey_2[DY_BAND].plot(cmap=cm.coolwarm)

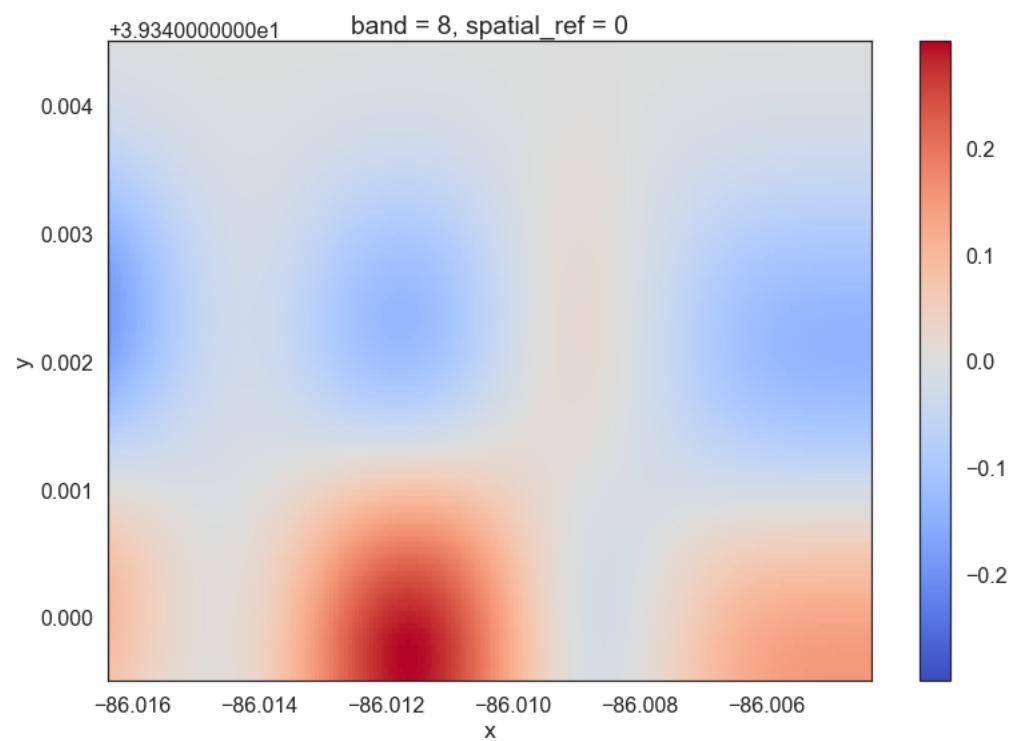
plt.figure()
_4km_per_flt_lvl_survey[SCALAR_BAND].interp(x=_1km_plane_lvl_survey_2.x,
    ↪y=_1km_plane_lvl_survey_2.y).plot(cmap=cm.coolwarm)
plt.figure()
_4km_per_flt_lvl_survey[DX_BAND].interp(x=_1km_plane_lvl_survey_2.x,
    ↪y=_1km_plane_lvl_survey_2.y).plot(cmap=cm.coolwarm)
plt.figure()
_4km_per_flt_lvl_survey[DY_BAND].interp(x=_1km_plane_lvl_survey_2.x,
    ↪y=_1km_plane_lvl_survey_2.y).plot(cmap=cm.coolwarm)

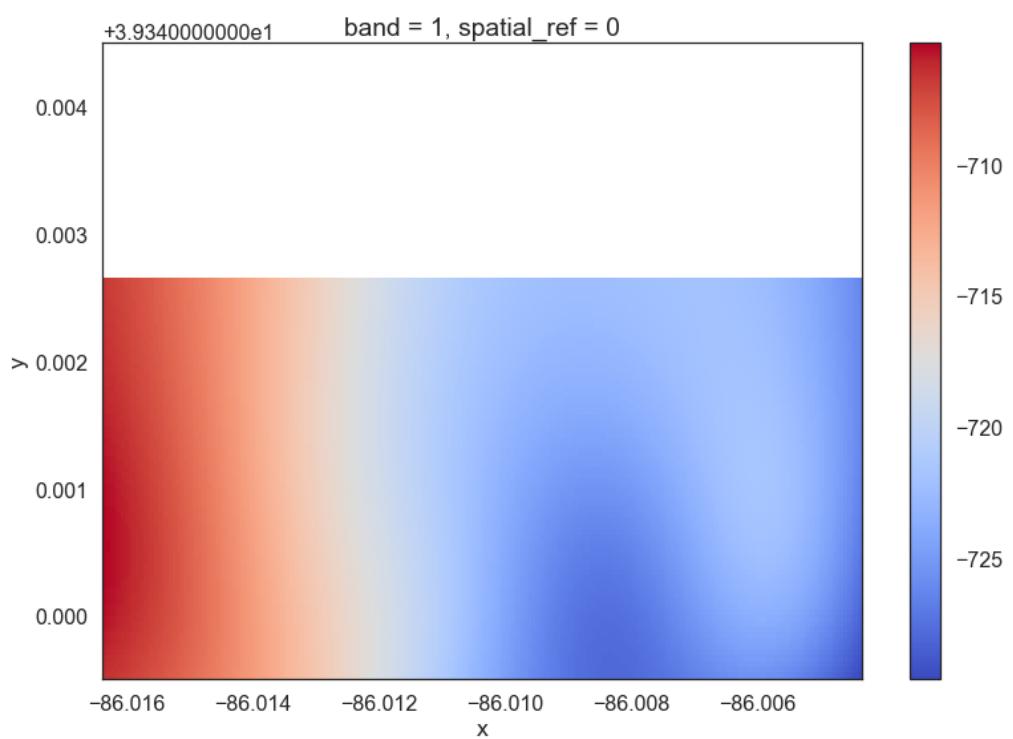
```

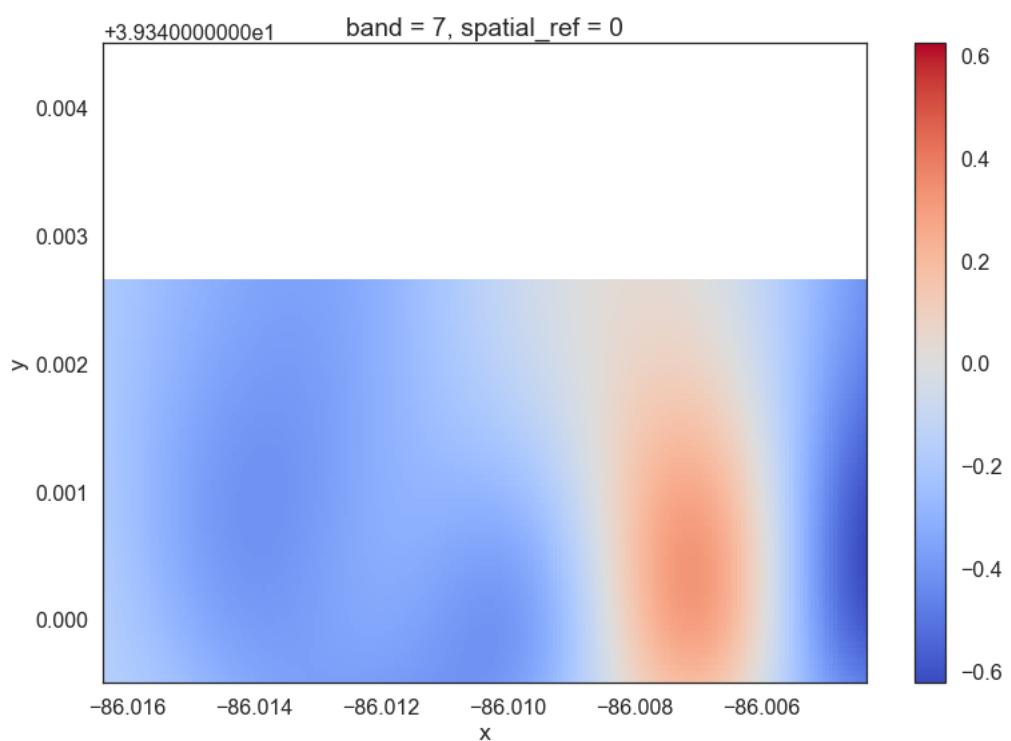
```
[ ]: <matplotlib.collections.QuadMesh at 0x21115ca33a0>
```

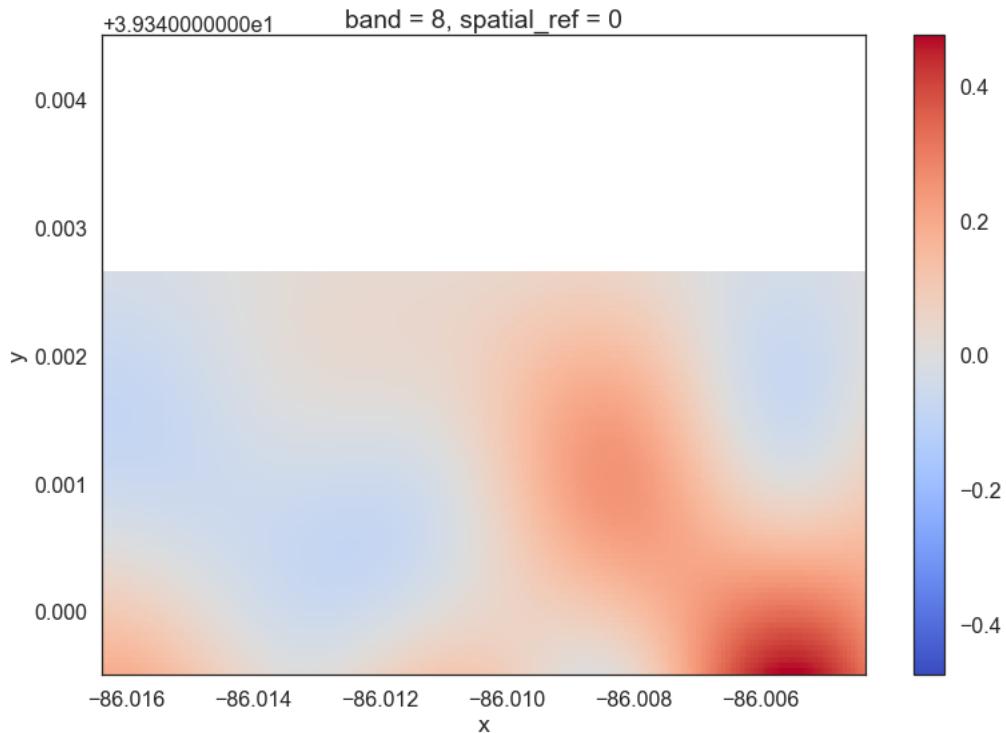












## 2 Find Corrugation FOM for Each Map

```
[ ]: cutoffs = [300, 500]
band      = SCALAR_BAND
axis      = 1

_1km_no_lvl_survey_1_fom      = find_corregation_fom(_1km_no_lvl_survey_1,      ↴
    ↵cutoffs, 5, band, axis)
_1km_pca_lvl_survey_1_fom     = find_corregation_fom(_1km_pca_lvl_survey_1,     ↴
    ↵cutoffs, 5, band, axis)
_1km_per_flt_lvl_survey_1_fom = find_corregation_fom(_1km_per_flt_lvl_survey_1, ↴
    ↵cutoffs, 5, band, axis)
_1km_plane_lvl_survey_1_fom   = find_corregation_fom(_1km_plane_lvl_survey_1,   ↴
    ↵cutoffs, 5, band, axis)

_1km_no_lvl_survey_2_fom      = find_corregation_fom(_1km_no_lvl_survey_2,      ↴
    ↵cutoffs, 5, band, axis)
_1km_pca_lvl_survey_2_fom     = find_corregation_fom(_1km_pca_lvl_survey_2,     ↴
    ↵cutoffs, 5, band, axis)
```

```

_1km_per_flt_lvl_survey_2_fom = find_corregation_fom(_1km_per_flt_lvl_survey_2,
    ↪cutoffs, 5, band, axis)
_1km_plane_lvl_survey_2_fom   = find_corregation_fom(_1km_plane_lvl_survey_2,   ↪
    ↪cutoffs, 5, band, axis)

_2km_no_lvl_survey_fom       = find_corregation_fom(_2km_no_lvl_survey,      ↪
    ↪cutoffs, 5, band, axis)
_2km_pca_lvl_survey_fom     = find_corregation_fom(_2km_pca_lvl_survey,     ↪
    ↪cutoffs, 5, band, axis)
_2km_per_flt_lvl_survey_fom = find_corregation_fom(_2km_per_flt_lvl_survey, ↪
    ↪cutoffs, 5, band, axis)
_2km_plane_lvl_survey_fom   = find_corregation_fom(_2km_plane_lvl_survey,   ↪
    ↪cutoffs, 5, band, axis)

_4km_no_lvl_survey_fom       = find_corregation_fom(_4km_no_lvl_survey,      ↪
    ↪cutoffs, 10, band, axis)
_4km_pca_lvl_survey_fom     = find_corregation_fom(_4km_pca_lvl_survey,     ↪
    ↪cutoffs, 10, band, axis)
_4km_per_flt_lvl_survey_fom = find_corregation_fom(_4km_per_flt_lvl_survey, ↪
    ↪cutoffs, 10, band, axis)
_4km_plane_lvl_survey_fom   = find_corregation_fom(_4km_plane_lvl_survey,   ↪
    ↪cutoffs, 10, band, axis)

print('_1km_no_lvl_survey_1      FOM:', _1km_no_lvl_survey_1_fom)
print('_1km_pca_lvl_survey_1      FOM:', _1km_pca_lvl_survey_1_fom)
print('_1km_per_flt_lvl_survey_1 FOM:', _1km_per_flt_lvl_survey_1_fom)
print('_1km_plane_lvl_survey_1    FOM:', _1km_plane_lvl_survey_1_fom)

print('_1km_no_lvl_survey_2      FOM:', _1km_no_lvl_survey_2_fom)
print('_1km_pca_lvl_survey_2      FOM:', _1km_pca_lvl_survey_2_fom)
print('_1km_per_flt_lvl_survey_2 FOM:', _1km_per_flt_lvl_survey_2_fom)
print('_1km_plane_lvl_survey_2    FOM:', _1km_plane_lvl_survey_2_fom)

print('_2km_no_lvl_survey        FOM:', _2km_no_lvl_survey_fom)
print('_2km_pca_lvl_survey        FOM:', _2km_pca_lvl_survey_fom)
print('_2km_per_flt_lvl_survey    FOM:', _2km_per_flt_lvl_survey_fom)
print('_2km_plane_lvl_survey      FOM:', _2km_plane_lvl_survey_fom)

print('_4km_no_lvl_survey        FOM:', _4km_no_lvl_survey_fom)
print('_4km_pca_lvl_survey        FOM:', _4km_pca_lvl_survey_fom)
print('_4km_per_flt_lvl_survey    FOM:', _4km_per_flt_lvl_survey_fom)
print('_4km_plane_lvl_survey      FOM:', _4km_plane_lvl_survey_fom)

_1km_no_lvl_survey_1      FOM: 0.01340355613087892
_1km_pca_lvl_survey_1      FOM: 0.01494787449893193
_1km_per_flt_lvl_survey_1 FOM: 0.01473053477231277

```

```

_1km_plane_lvl_survey_1    FOM: 0.013348248403973485
_1km_no_lvl_survey_2      FOM: 0.02354461923387489
_1km_pca_lvl_survey_2     FOM: 0.01575741818799143
_1km_per_flt_lvl_survey_2 FOM: 0.0224194196054789
_1km_plane_lvl_survey_2   FOM: 0.025328450125894207
_2km_no_lvl_survey        FOM: 0.01831323241324738
_2km_pca_lvl_survey       FOM: 0.014422441753486772
_2km_per_flt_lvl_survey  FOM: 0.022972464525653435
_2km_plane_lvl_survey     FOM: 0.021801233581354402
_4km_no_lvl_survey        FOM: 0.021360770281794013
_4km_pca_lvl_survey       FOM: 0.019616981110879766
_4km_per_flt_lvl_survey  FOM: 0.02786507685896278
_4km_plane_lvl_survey     FOM: 0.02296736377986826

```

### 3 Make FOM Comparison Plot

```

[ ]: barWidth = 0.15
fig = plt.subplots(figsize =(12, 8))

none_bars    = [_1km_no_lvl_survey_1_fom,
                _1km_no_lvl_survey_2_fom,
                _2km_no_lvl_survey_fom,
                _4km_no_lvl_survey_fom]
pca_bars     = [_1km_pca_lvl_survey_1_fom,
                _1km_pca_lvl_survey_2_fom,
                _2km_pca_lvl_survey_fom,
                _4km_pca_lvl_survey_fom]
per_flt_bars = [_1km_per_flt_lvl_survey_1_fom,
                 _1km_per_flt_lvl_survey_2_fom,
                 _2km_per_flt_lvl_survey_fom,
                 _4km_per_flt_lvl_survey_fom]
plane_bars   = [_1km_plane_lvl_survey_1_fom,
                 _1km_plane_lvl_survey_2_fom,
                 _2km_plane_lvl_survey_fom,
                 _4km_plane_lvl_survey_fom]

br1 = np.arange(len(none_bars))
br2 = [x + barWidth for x in br1]
br3 = [x + barWidth for x in br2]
br4 = [x + barWidth for x in br3]

# Make the plot
plt.bar(br1, none_bars,   color ='r', width = barWidth, edgecolor ='grey', u
        ↪label ='No Leveling')
plt.bar(br2, pca_bars,    color ='g', width = barWidth, edgecolor ='grey', u
        ↪label ='PCA Leveling')

```

```

plt.bar(br3, per_fl_bars, color ='b', width = barWidth, edgecolor ='grey',  

        label ='Per Flight Line Leveling')  

plt.bar(br4, plane_bars, color ='m', width = barWidth, edgecolor ='grey',  

        label ='Plane of Best Fit Leveling')  

# Adding Xticks  

plt.xlabel('Aeromagnetic Surveys', fontweight ='bold', fontsize = 15)  

plt.ylabel('Corrugation FOM ($dB^{-1}$)', fontweight ='bold', fontsize =  

          15)  

plt.xticks([r + barWidth for r in range(len(none_bars))],  

          ['1km Atterbury Survey\n(First Attempt)', '1km Atterbury  

           Survey\n(Second Attempt)', '2km Atterbury Survey', '4km Atterbury Survey'])  

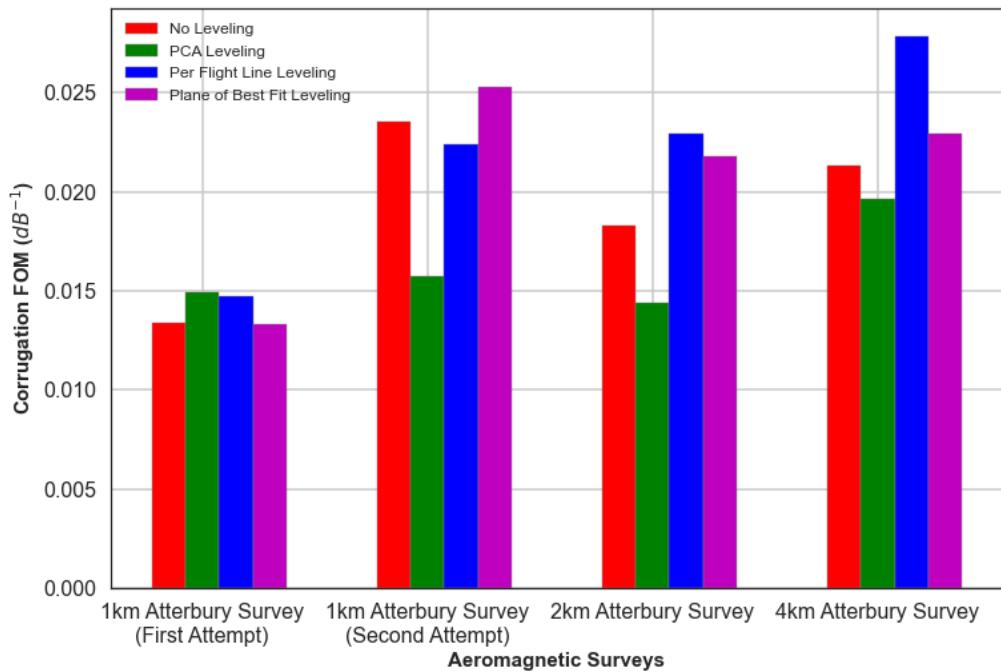
  

plt.grid()  

plt.legend(fontsize='large')  

plt.show()

```



## 4 Map Correlations

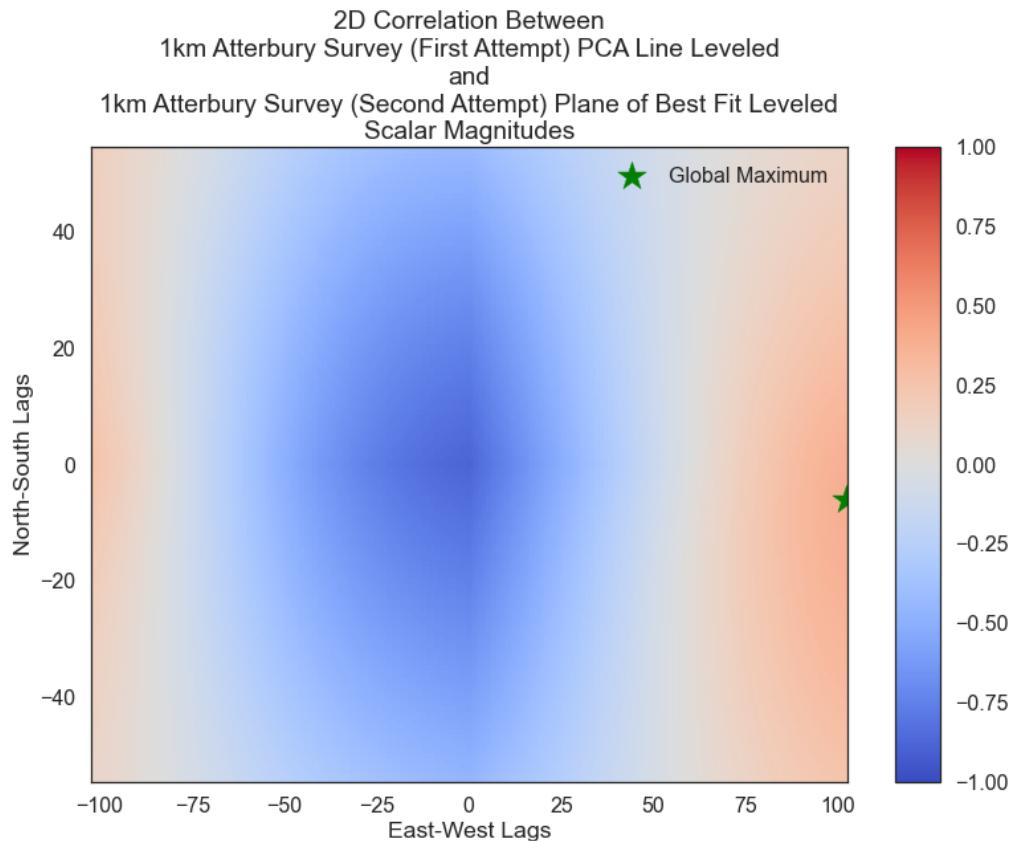
```
[ ]: compare_maps(_1km_pca_lvl_survey_1,
                  _1km_plane_lvl_survey_2,
                  ref_map_name='1km Atterbury Survey (First Attempt) PCA Line Leveled',
                  comp_map_name='1km Atterbury Survey (Second Attempt) Plane of Best Fit Leveled')
compare_maps(_2km_per_flt_lvl_survey,
              _4km_per_flt_lvl_survey,
              ref_map_name='2km Atterbury Survey Per Flight Line Leveled',
              comp_map_name='4km Atterbury Survey Per Flight Line Leveled')
compare_maps(_1km_per_flt_lvl_survey_2,
              _4km_per_flt_lvl_survey,
              ref_map_name='1km Atterbury Survey (Second Attempt) Plane of Best Fit Leveled',
              comp_map_name='4km Atterbury Survey Per Flight Line Leveled')

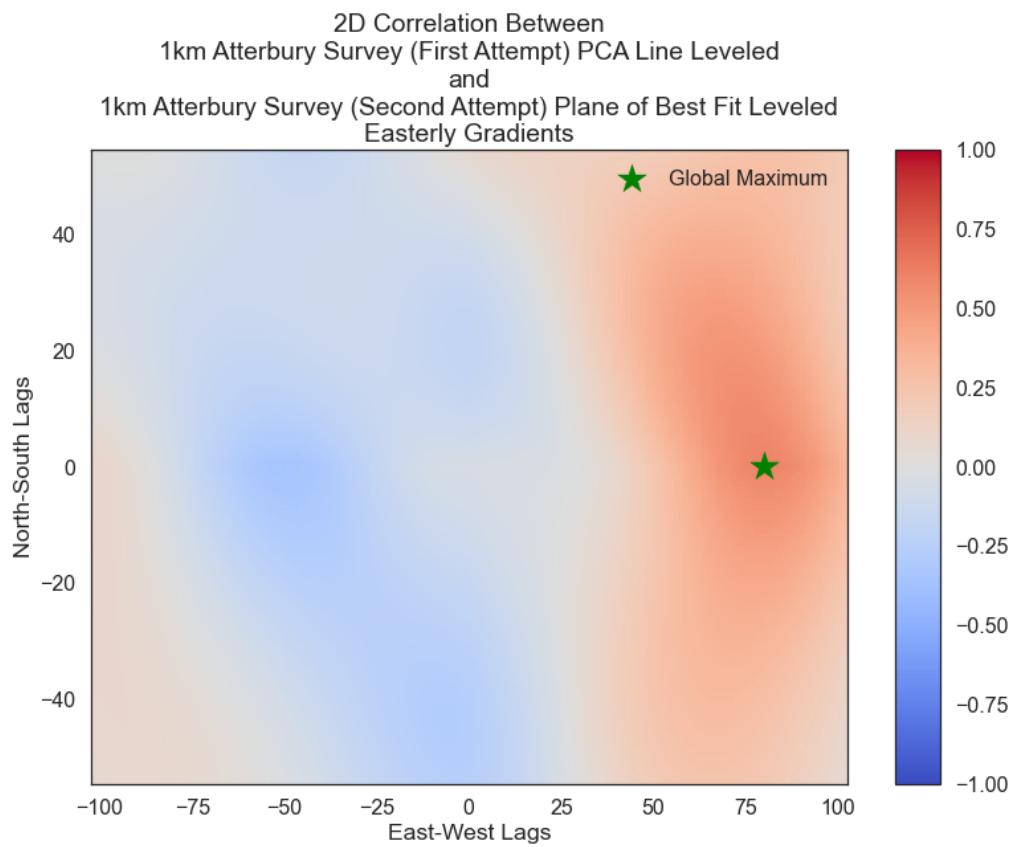
*****
2D Correlation Between
1km Atterbury Survey (First Attempt) PCA Line Leveled
and
1km Atterbury Survey (Second Attempt) Plane of Best Fit Leveled
Scalar Magnitudes
Max Scalar Pearson: 0.3881907465354701
Max Scalar Pearson Lags: [102, -6]
Zero Lag Scalar Pearson: -0.8834348885114223
Max dX Pearson: 0.5961616240255544
Max dX Pearson Lags: [80, 0]
Zero Lag dX Pearson: -0.05339229631899242
Max dY Pearson: 0.7580391323065421
Max dY Pearson Lags: [0, 0]
Zero Lag dY Pearson: 0.7580391323065421
*****
2D Correlation Between
2km Atterbury Survey Per Flight Line Leveled
and
4km Atterbury Survey Per Flight Line Leveled
Scalar Magnitudes
Max Scalar Pearson: 0.6445899665944941
Max Scalar Pearson Lags: [0, -1]
Zero Lag Scalar Pearson: 0.6424424039327695
Max dX Pearson: 0.2655640797643319
Max dX Pearson Lags: [-85, -142]
Zero Lag dX Pearson: 0.041969658021214065
Max dY Pearson: 0.2979138377382414
Max dY Pearson Lags: [151, -28]
```

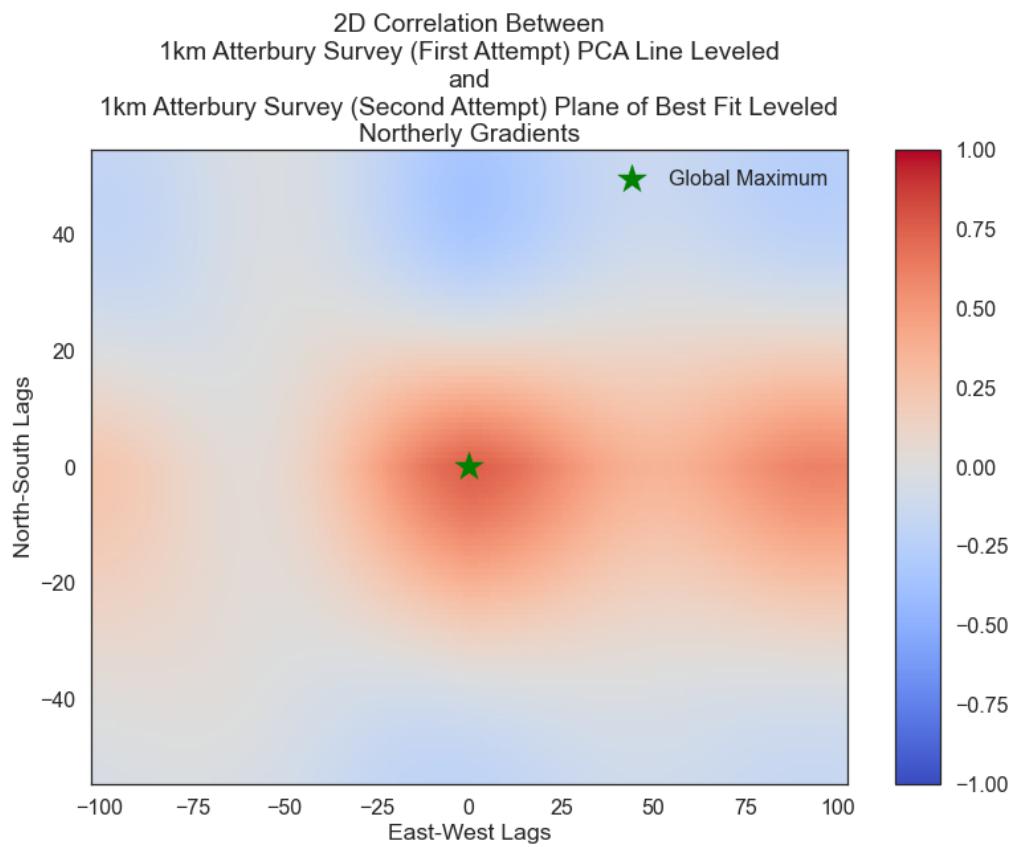
```

Zero Lag dY Pearson: 0.16419254135497013
*****
2D Correlation Between
1km Atterbury Survey (Second Attempt) Plane of Best Fit Leveled
and
4km Atterbury Survey Per Flight Line Leveled
Scalar Magnitudes
Max Scalar Pearson: 0.8272013876125689
Max Scalar Pearson Lags: [0, -1]
Zero Lag Scalar Pearson: 0.811971750034992
Max dX Pearson: 0.3752179698116518
Max dX Pearson Lags: [44, -1]
Zero Lag dX Pearson: -0.46037233350385615
Max dY Pearson: 0.4026901580006161
Max dY Pearson Lags: [-103, -1]
Zero Lag dY Pearson: 0.21633015106616563

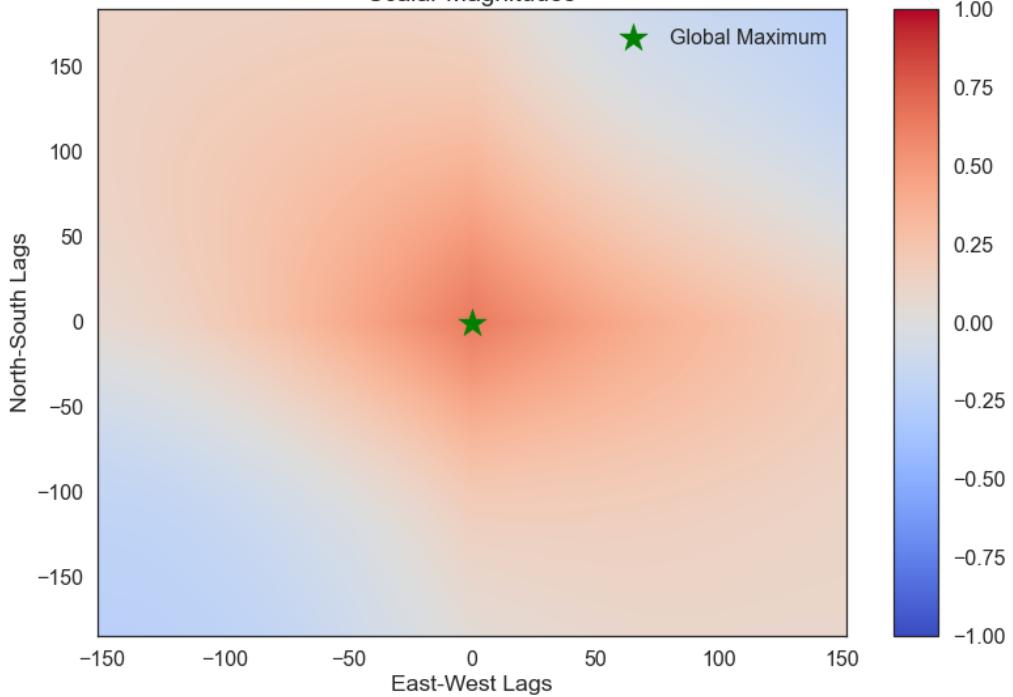
```

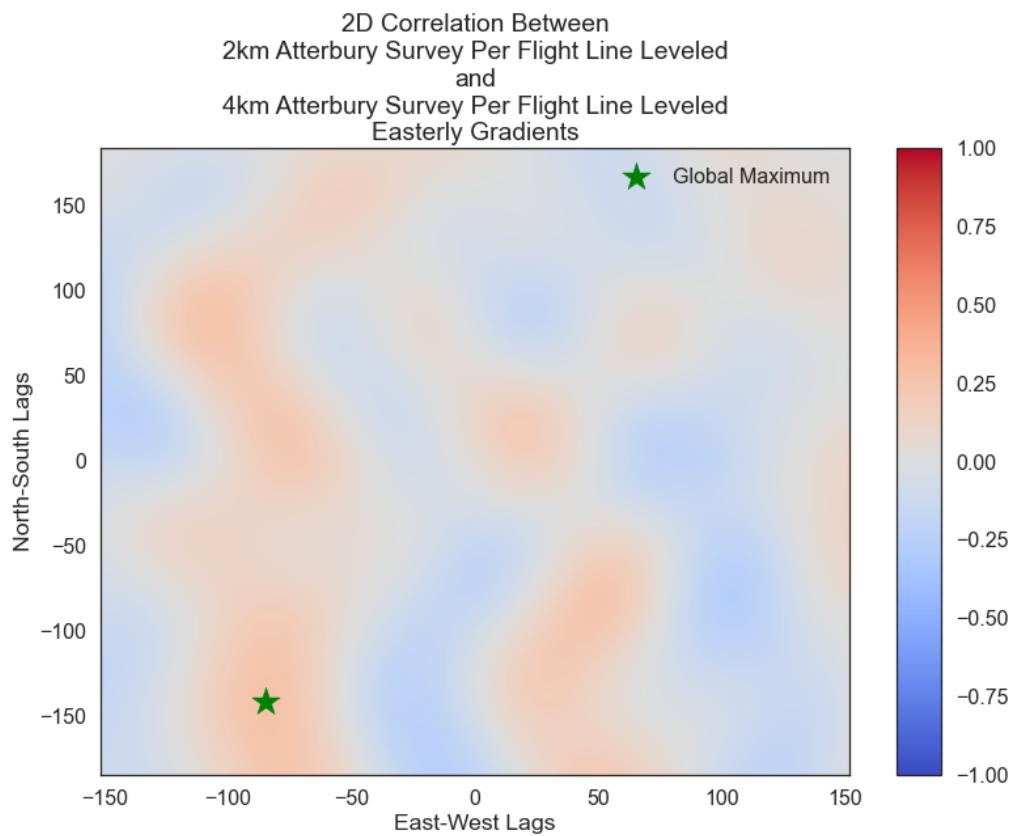


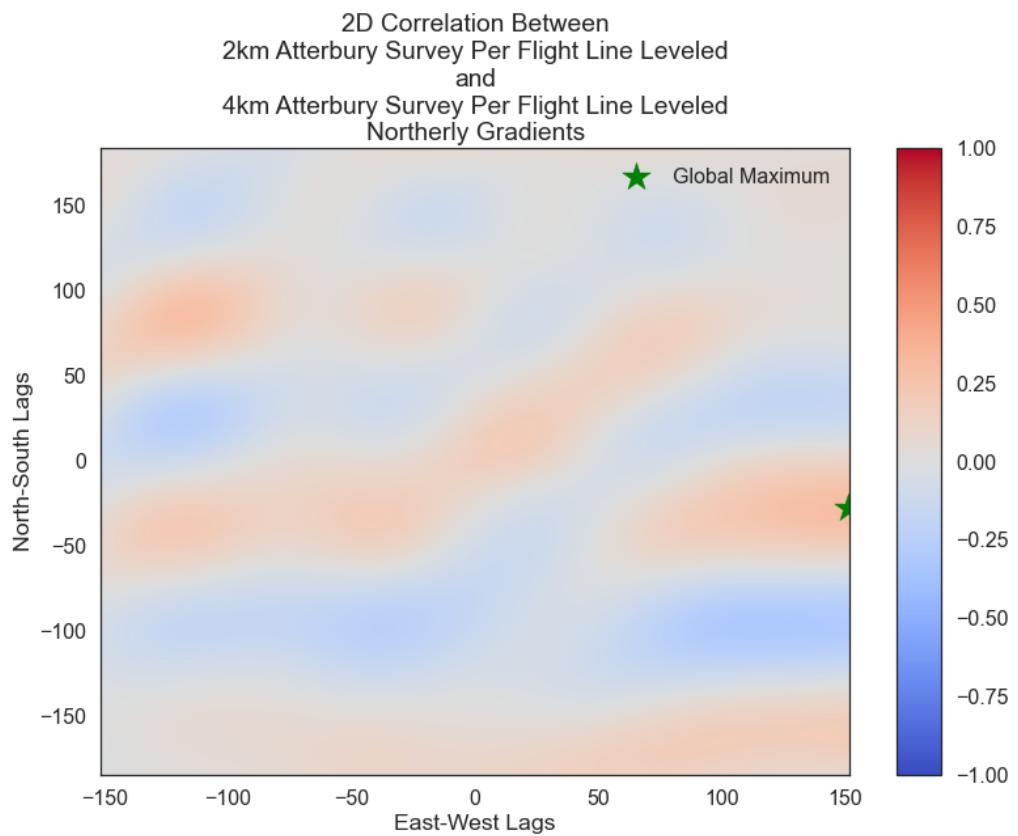




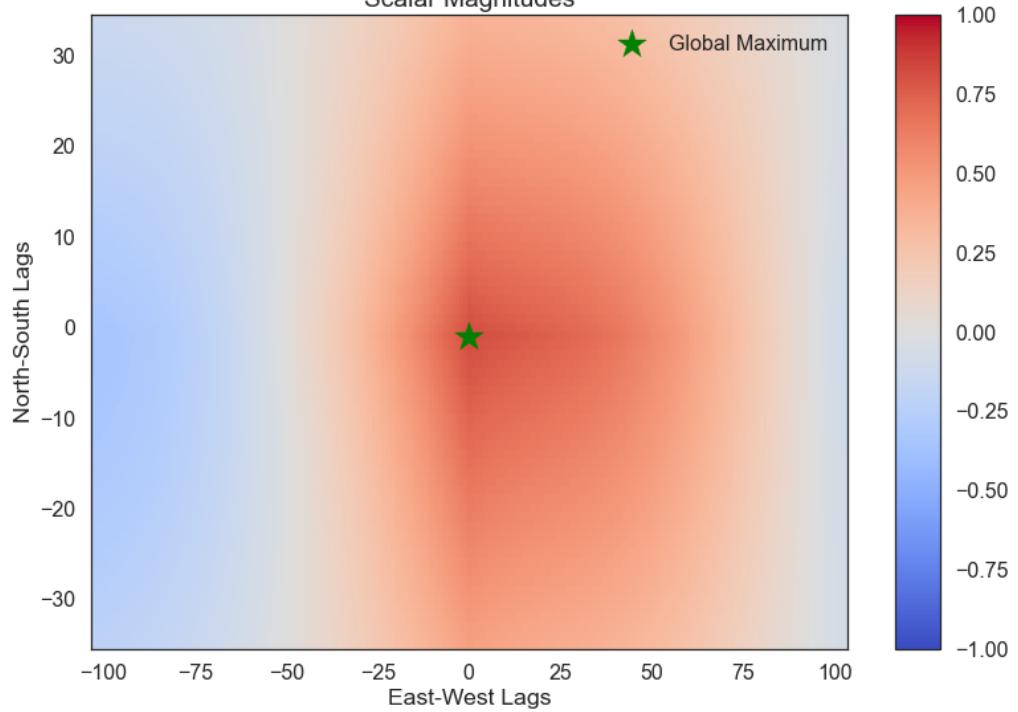
2D Correlation Between  
2km Atterbury Survey Per Flight Line Leveled  
and  
4km Atterbury Survey Per Flight Line Leveled  
Scalar Magnitudes



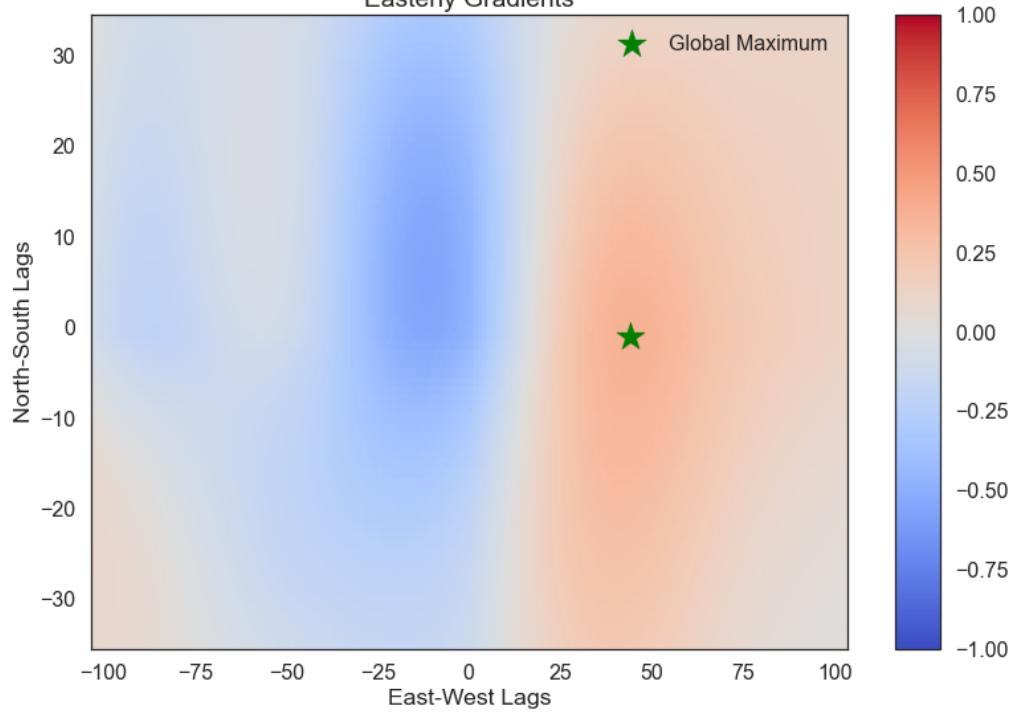


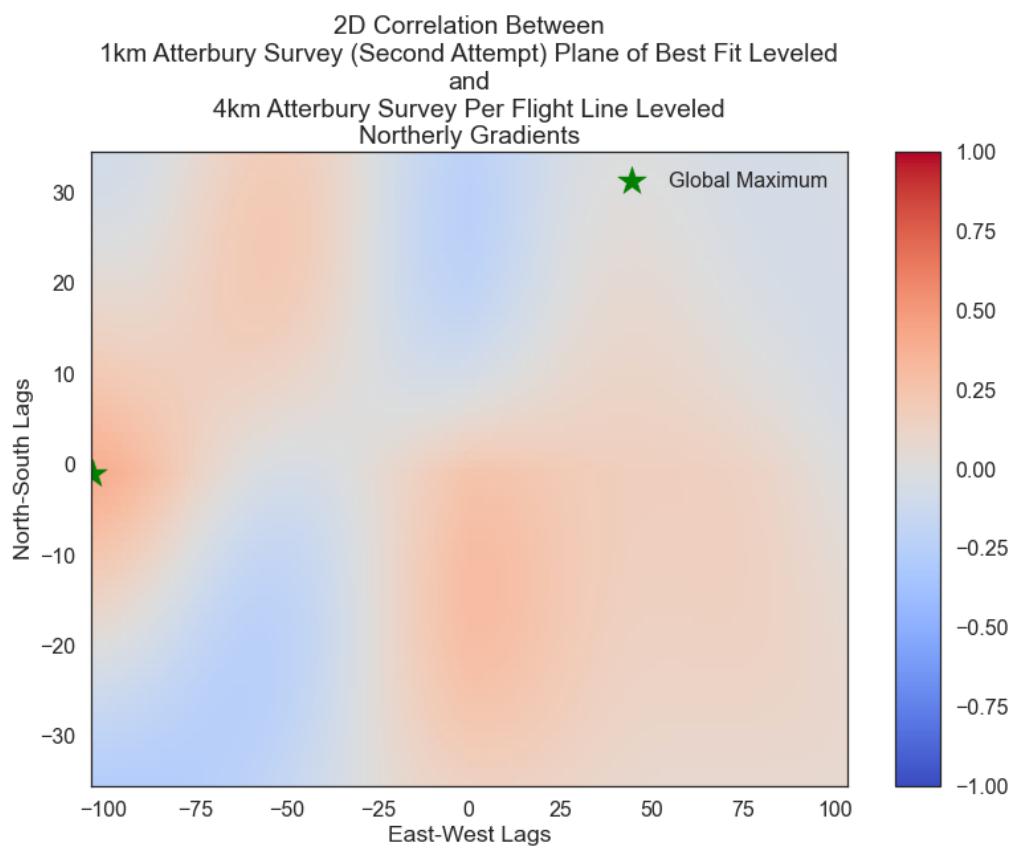


2D Correlation Between  
1km Atterbury Survey (Second Attempt) Plane of Best Fit Leveled  
and  
4km Atterbury Survey Per Flight Line Leveled  
Scalar Magnitudes



2D Correlation Between  
1km Atterbury Survey (Second Attempt) Plane of Best Fit Leveled  
and  
4km Atterbury Survey Per Flight Line Leveled  
Easterly Gradients





## Appendix F. MAMMAL Python Library

mammal/setup.py

```
1 from setuptools import find_packages, setup
2
3 with open('README.md', "r", encoding = "utf-8") as fh:
4     long_description = fh.read()
5
6 setup(
7     name          = 'MAMMAL',
8     packages      = find_packages(),
9     version       = '0.0.1',
10    description   = 'Python package used to create aeromagnetic anomaly maps for Magnetic Navigation (MagNav)',
11    long_description = long_description,
12    long_description_content_type = "text/markdown",
13    author        = '',
14    author_email  = '',
15    url          = '',
16    download_url = '',
17    keywords      = [''],
18    classifiers   = [],
19    install_requires = [
20        'numpy==1.20.3',
21        'pandas==1.3.4',
22        'scipy==1.7.1',
23        'matplotlib==3.4.3',
24        'tqdm==4.62.3',
25        #'gdal==3.4.3', # Must be installed via conda
26        'xarray==2022.3.0',
27        'rioxarray==0.10.3',
28        'scikit-learn==0.24.2',
29        'ppigrf==1.0.0',
30        #'geoscraper==0.0.1',
31        'simplekml==1.3.3'
32    ]
33)
```

mammal/MAMMAL/\_\_init\_\_.py

```
1 import sys
2 import datetime as dt
3 from os.path import dirname, realpath, join
4
5 import numpy as np
6 import pandas as pd
7 import rioxarray as rxr
8
9 SRC_DIR      = dirname(realpath(__file__))
10 PROJ_DIR     = dirname(SRC_DIR)
11 DOCS_DIR     = join(PROJ_DIR, 'docs')
12 EXAMPLES_DIR = join(PROJ_DIR, 'examples')
13 DATA_DIR     = join(PROJ_DIR, 'data')
14 TEST_DIR     = join(DATA_DIR, 'test')
```

```

15
16     sys.path.append(SRC_DIR)
17
18     import Simulator as sim
19     from Parse import parseIM as pim
20     from Utils import ProcessingUtils as pu
21     from VehicleCal import TL as tl
22
23
24     class BaseClass():
25         ...
26
27         ...
28
29         def __init__(self,
30             map_loc_name: str='test',
31             center_lat_dd: float=0,
32             center_lon_dd: float=0,
33             alt_m_msl: float=0,
34             alt_m_agl: float=0) -> None:
35             ...
36
37             ...
38
39             # Spin Test
40             self.spin_lat = center_lat_dd
41             self.spin_lon = center_lon_dd
42             self.spin_height_m = 0
43             self.spin_a = np.eye(3)
44             self.spin_b = np.zeros(3)
45
46             # Tolles-Lawson
47             self.tl_center_lat = center_lat_dd
48             self.tl_center_lon = center_lon_dd
49             self.tl_height_m = alt_m_msl
50             self.tl_box_xlen_m = 100
51             self.tl_box_ylen_m = 100
52             self.tl_c = np.zeros(18)
53             self.tl_dither_hz = 1
54             self.tl_dither_amp = 10
55             self.tl_terms = tl.ALL_TERMS
56
57             # Map
58             self.map_loc_name = map_loc_name
59             self.map_center_lat = center_lat_dd
60             self.map_center_lon = center_lon_dd
61             self.map_height_m = alt_m_msl
62             self.map_height_agl_m = alt_m_agl
63             self.map_dx_m = self.map_height_agl_m / 20
64             self.map_dy_m = self.map_height_agl_m / 20
65
66
67     class SurveySim(BaseClass):
68         ...

```

```

69
70     ,
71
72     def __init__(self,
73             map_loc_name: str='MAMMAL',
74             center_lat_dd: float=0,
75             center_lon_dd: float=0,
76             start_dt_utc: dt.datetime=dt.datetime.utcnow(),
77             alt_m_msl: float=0,
78             alt_m_agl: float=0,
79             vel_mps: float=20,
80             samp_hz: float=10,
81             data_dir: str=None,
82             debug: bool=False) -> None:
83         ,
84
85         ,
86
87         super().__init__(map_loc_name,
88                         center_lat_dd,
89                         center_lon_dd,
90                         alt_m_msl,
91                         alt_m_agl)
92
93         # General
94         self.vel_mps = vel_mps
95         self.samp_hz = samp_hz
96         self.data_dir = data_dir
97         self.debug = debug
98
99         # Spin Test
100        self.spin_df = None
101        self.spin_start_dt = start_dt_utc
102        self.spin_headings = np.linspace(0, 720, 1000)
103        self.spin_elevations = np.linspace(0, 7200, 1000)
104
105        # Tolles-Lawson
106        self.tl_df = None
107        self.tl_start_dt = start_dt_utc
108        self.tl_vel_mps = self.vel_mps
109        self.tl_sample_hz = self.samp_hz
110
111        # Reference Station
112        self.ref_df = None
113        self.ref_file_df = None
114        self.ref_lat = center_lat_dd
115        self.ref_lon = center_lon_dd
116        self.ref_height_m = 0
117        self.ref_start_dt = start_dt_utc
118        self.ref_dur_s = 10000
119        self.ref_scale = 1
120        self.ref_offset = 0
121        self.ref_awgn_std = 0
122        self.ref_sample_hz = 1

```

```

123     self.ref_id      = 'FRD'
124
125     # Map
126     self.sim_map      = None
127     self.map_upcontinue = False
128     self.map_x_dist_m = 300
129     self.map_y_dist_m = 300
130     self.map_start_dt = start_dt_utc
131     self.map_anomaly_locs = np.array([[self.map_center_lat], # dd
132                                         [self.map_center_lon]]) # dd
133     self.map_anomaly_scales = np.array([20]) # nT
134     self.map_anomaly_covs = np.zeros((1, 2, 2))
135     self.map_anomaly_covs[0, :, :] = np.diag([0.000001, 0.000002])
136
137     # Survey
138     self.survey_height_m = self.map_height_m
139     self.survey_start_dt = start_dt_utc
140     self.survey_vel_mps = self.vel_mps
141     self.survey_e_buff_m = 15
142     self.survey_w_buff_m = 15
143     self.survey_n_buff_m = 15
144     self.survey_s_buff_m = 15
145     self.survey_sample_hz = self.samp_hz
146     self.survey_ft_line_dist_m = self.map_height_agl_m / 2
147     self.survey_ft_line_dir = sim.HORIZ
148     self.survey_scalar_awgn_std = 0
149     self.survey_diurnal_dist = np.array([0, 1]), # [offset (nT), scale]
150     self.survey_use_tie_lines = True
151     self.survey_tie_dist_m = self.survey_ft_line_dist_m * 5
152
153     def gen_spin_data(self) -> pd.DataFrame:
154         """
155
156         """
157
158         self.spin_df = sim.gen_spin_data(out_dir = self.data_dir,
159                                         lat = self.spin_lat,
160                                         lon = self.spin_lon,
161                                         height = self.spin_height_m,
162                                         date = self.spin_start_dt,
163                                         headings = self.spin_headings,
164                                         elevations = self.spin_elevations,
165                                         a = self.spin_a,
166                                         b = self.spin_b,
167                                         debug = self.debug)
168
169         return self.spin_df
170
171     def gen_TL_data(self) -> pd.DataFrame:
172         """
173
174         """
175         self.tl_df = sim.gen_TL_data(out_dir = self.data_dir,
176                                     center_lat = self.tl_center_lat,

```

```

177         center_lon = self.tl_center_lon,
178         height      = self.tl_height_m,
179         start_dt   = self.tl_start_dt,
180         box_xlen_m = self.tl_box_xlen_m,
181         box_ylen_m = self.tl_box_ylen_m,
182         c          = self.tl_c,
183         vel_mps    = self.tl_vel_mps,
184         sample_hz  = self.tl_sample_hz,
185         dither_hz  = self.tl_dither_hz,
186         dither_amp = self.tl_dither_amp,
187         terms      = self.tl_terms,
188         a          = self.spin_a,
189         b          = self.spin_b,
190         debug      = self.debug)
191
192     return self.tl_df
193
194     def gen_ref_station_data(self) -> pd.DataFrame:
195         """
196         """
197
198         if self.ref_id is not None and self.data_dir is not None:
199             self.ref_file_df = pim.loadInterMagData(self.data_dir)[self.ref_id]
200         else:
201             self.ref_file_df = None
202
203         self.ref_df = sim.gen_ref_station_data(out_dir = self.data_dir,
204                                              lat      = self.ref_lat,
205                                              lon      = self.ref_lon,
206                                              height   = self.ref_height_m,
207                                              start_dt = self.ref_start_dt,
208                                              dur_s    = self.ref_dur_s,
209                                              scale    = self.ref_scale,
210                                              offset   = self.ref_offset,
211                                              awgn_std = self.ref_awgn_std,
212                                              sample_hz = self.ref_sample_hz,
213                                              file_df   = self.ref_file_df,
214                                              debug     = self.debug)
215
216
217     return self.ref_df
218
219     def gen_sim_map(self) -> pd.DataFrame:
220         """
221         """
222
223         self.sim_map = sim.gen_sim_map(out_dir      = self.data_dir,
224                                       location     = self.map_loc_name,
225                                       center_lat  = self.map_center_lat,
226                                       center_lon  = self.map_center_lon,
227                                       dx_m        = self.map_dx_m,
228                                       dy_m        = self.map_dy_m,
229                                       x_dist_m   = self.map_x_dist_m,
230                                       y_dist_m   = self.map_y_dist_m,
```

```

231             height      = self.map_height_m,
232             date       = self.map_start_dt,
233             anomaly_locs = self.map_anomaly_locs,
234             anomaly_scales = self.map_anomaly_scales,
235             anomaly_covs = self.map_anomaly_covs,
236             upcontinue   = self.map_upcontinue,
237             debug        = self.debug)
238
239     return self.sim_map
240
241
242
243
244
245     self.survey_df = sim.gen_survey_data(out_dir           = self.data_dir,
246                                         map            = self.sim_map,
247                                         survey_height_m = self.survey_height_m,
248                                         survey_start_dt = self.survey_start_dt,
249                                         survey_vel_mps = self.survey_vel_mps,
250                                         survey_e_buff_m = self.survey_e_buff_m,
251                                         survey_w_buff_m = self.survey_w_buff_m,
252                                         survey_n_buff_m = self.survey_n_buff_m,
253                                         survey_s_buff_m = self.survey_s_buff_m,
254                                         sample_hz      = self.survey_sample_hz,
255                                         ft_line_dist_m = self.survey_ft_line_dist_m,
256                                         ft_line_dir    = self.survey_ft_line_dir,
257                                         a              = self.spin_a,
258                                         b              = self.spin_b,
259                                         c              = self.tl_c,
260                                         terms          = self.tl_terms,
261                                         scalar_awgn_std = self.survey_scalar_awgn_std,
262                                         diurnal_df     = self.ref_df,
263                                         diurnal_dist   = self.survey_diurnal_dist,
264                                         use_tie_lines  = self.survey_use_tie_lines,
265                                         tie_dist_m     = self.survey_tie_dist_m,
266                                         debug          = self.debug)
267
268
269
270 class MapMaker(BaseClass):
271
272
273
274
275     def __init__(self,
276                  map_loc_name: str=MAMMAL',
277                  alt_m_msl:    float=0,
278                  alt_m_agl:    float=0,
279                  data_dir:     str=None,
280                  debug:        bool=False) -> None:
281
282
283
284

```

```

285     super().__init__(map_loc_name,
286                      0,
287                      0,
288                      alt_m_msl,
289                      alt_m_agl)
290
291     # General
292     self.data_dir = data_dir
293     self.debug = debug
294
295     # Spin Test
296     self.spin_fname = None
297     self.spin_df = None
298     self.a = np.eye(3)
299     self.b = np.zeros(3)
300
301     # Tolles-Lawson
302     self.tl_fname = None
303     self.tl_df = None
304     self.c = np.zeros(18)
305     self.use_filter = True
306     self.fstart = 0.1
307     self.fstop = 1
308     self.terms = tl.ALL_TERMS
309
310     # Reference Station
311     self.ref_fname = None
312     self.ref_df = None
313     self.ref_scale = 1
314     self.ref_offset = 0
315     self.enable_lon_norm = False
316
317     # Survey
318     self.survey_fname = None
319     self.survey_df = None
320
321     # Map
322     self.map = None
323     self.lvl_type = None
324     self.num_ptls = None
325     self.ptl_locs = None
326     self.percent_thresh = 0.85
327     self.sensor_sigma = 0
328     self.interp_type = 'bicubic'
329
330     def spin_params(self,
331                     spin_df: pd.DataFrame=None,
332                     use_internal: bool=False) -> list:
333         ...
334
335         ...
336
337         if (spin_df is None) and (use_internal is False):
338             self.spin_df = pd.read_csv(self.spin_fname, parse_dates=['datetime'])

```

```

339         elif spin_df is not None:
340             self.spin_df = spin_df
341
342             self.a, self.b = pu.cal_spin_df(self.spin_df)
343
344         return self.a, self.b
345
346     def tl_params(self,
347                 tl_df: pd.DataFrame=None,
348                 use_internal: bool=False) -> np.ndarray:
349         ...
350
351         ...
352
353         if (tl_df is None) and (use_internal is False):
354             self.tl_df = pd.read_csv(self.tl_fname, parse_dates=['datetime'])
355         elif tl_df is not None:
356             self.tl_df = tl_df
357
358             self.c = pu.cal_tl_df(self.tl_df,
359                                 self.use_filter,
360                                 self.fstart,
361                                 self.fstop,
362                                 self.a,
363                                 self.b,
364                                 self.terms)
365
366         return self.c
367
368     def gen_map(self,
369                 survey_df: pd.DataFrame=None,
370                 survey_use_internal: bool=False,
371                 ref_df: pd.DataFrame=None,
372                 ref_use_internal: bool=False) -> rxr.rioxarray.raster_dataset.xarray.DataArray:
373
374         ...
375
376         if (survey_df is None) and (survey_use_internal is False):
377             self.survey_df = pd.read_csv(self.survey_fname, parse_dates=['datetime'])
378         elif survey_df is not None:
379             self.survey_df = survey_df
380
381         if (ref_df is None) and (ref_use_internal is False):
382             self.ref_df = pd.read_csv(self.ref_fname, parse_dates=['datetime'])
383         elif ref_df is not None:
384             self.ref_df = ref_df
385
386         self.map = pu.gen_map(out_dir = self.data_dir,
387                               map_name = self.map_loc_name,
388                               survey_df = self.survey_df,
389                               ref_df = self.ref_df,
390                               dx = self.map_dx_m,
391                               dy = self.map_dy_m,
392                               min_alt_agl = self.map_height_agl_m,

```

```

393             ref_scale      = self.ref_scale,
394             ref_offset     = self.ref_offset,
395             enable_lon_norm = self.enable_lon_norm,
396             a              = self.a,
397             b              = self.b,
398             c              = self.c,
399             terms          = self.terms,
400             lvl_type       = self.lvl_type,
401             num_ptls       = self.num_ptls,
402             ptl_locs       = self.ptl_locs,
403             percent_thresh = self.percent_thresh,
404             sensor_sigma   = self.sensor_sigma,
405             interp_type    = self.interp_type,
406             debug          = self.debug)
407
408         return self.map

```

### mammal/MAMMAL/Simulator/\_\_init\_\_.py

```

1 import datetime as dt
2 import sys
3 from os.path import dirname, join, realpath, exists
4 from typing import Union
5
6 import matplotlib.pyplot as plt
7 import numpy as np
8 import pandas as pd
9 import rioxarray as rxr
10 import scipy.linalg as la
11 import scipy.stats as stat
12 import xarray as xr
13 from matplotlib import cm
14 from tqdm import tqdm
15 from ppigrf import igrf
16
17 sys.path.append(dirname(realpath(__file__)))
18 sys.path.append(join(dirname(realpath(__file__)), ''))
19
20 import Diurnal
21 from SensorCal import spinCal as sc
22 from Utils import coordinateUtils as cu
23 from Utils import mapUtils as mu
24 from VehicleCal import TL as tl
25
26
27 # Enumerate line directions
28 HORIZ = 0 # Horizontal
29 VERT = 1 # Vertical
30
31 # Enumerate noise types
32 ZERO = 0
33 BIAS = 1 << 0
34 POLY = 1 << 1
35 AWCN = 1 << 2

```

```

36 FILE = 1 << 3
37
38 M2FT = 3.280839895 # m to ft conversion
39 FT2M = 1 / M2FT    # ft to m conversion
40 KM2M = 1000         # km to m conversion
41 MKM = 1 / KM2M     # m to km conversion
42
43
44 def save_dataset(type: str,
45                  out_dir: str,
46                  date: Union[dt.date, dt.datetime],
47                  data: dict,
48                  debug: bool=True) -> pd.DataFrame:
49      ...
50
51      Save dictionary of data into a .csv file and return
52      said data as a Pandas DataFrame
53
54      Parameters
55      -----
56      type
57          Custom field that will be prepended to the .csv's file name
58      out_dir
59          Path to the directory to save the .csv file to
60      date
61          Date/datetime object that corresponds to the dataset
62      data
63          Dictionary of data to save
64      debug
65          Whether or not debug prints/plot should be generated
66
67      Returns
68      -----
69      df
70          Pandas DataFrame of the given data
71
72      i = 0
73      fname = '{type}_{year}_{month}_{day}_{num}.csv'.format(type = type,
74                                                               year = date.year,
75                                                               month = date.month,
76                                                               day = date.day,
77                                                               num = i)
78      full_path = join(out_dir, fname)
79
80      while(exists(full_path)):
81          i += 1
82          fname = '{type}_{year}_{month}_{day}_{num}.csv'.format(type = type,
83                                                               year = date.year,
84                                                               month = date.month,
85                                                               day = date.day,
86                                                               num = i)
87          full_path = join(out_dir, fname)
88
89      df = pd.DataFrame(data)

```

```

90     df.to_csv(full_path)
91
92     if debug:
93         print('Saved_data to ', full_path)
94
95     return df
96
97 def gen_parallel_search(lon_min:           float,
98                         lon_max:            float,
99                         lat_min:            float,
100                        lat_max:            float,
101                        lon_total_dist_m: float,
102                        lat_total_dist_m: float,
103                        lon_sub_dist_m:   float,
104                        lat_sub_dist_m:   float,
105                        line_dir:          int,
106                        last_line_num:    int=0) -> list:
107     """
108     Generate an array of lat/lon coordinates of a parallel
109     search pattern lines within a given box boundary
110
111     Parameters
112     -----
113     lon_min
114         Minimum longitude of the generated lines
115     lon_max
116         Maximum longitude of the generated lines
117     lat_min
118         Minimum latitude of the generated lines
119     lat_max
120         Maximum latitude of the generated lines
121     lon_total_dist_m
122         Total longitudinal distance of the spread
123         of the generated lines (m)
124     lat_total_dist_m
125         Total latitudinal distance of the spread
126         of the generated lines (m)
127     lon_sub_dist_m
128         Distance between samples in the
129         longitudinal direction of the lines
130     lat_sub_dist_m
131         Distance between samples in the
132         latitudinal direction of the lines
133     line_dir
134         Direction the generated lines run. Options include:
135
136         - HORIZ: Horizontal - East/West
137         - VERT: Vertical - North/South
138
139     last_line_num
140         Number of the last line "surveyed"
141
142     Returns
143     -----

```

```

144      list
145          2xN array of lat/lon coordinates of a parallel
146          search pattern within a given box boundary and
147          1xN array of line numbers (flight or tie) —> [[ coord 0 lat (dd), coord 1 lat (dd), ... ]
148          ]
149          [[ coord 0 lon (dd), coord 1 lon (dd), ... ], [coord 0 line number
150          , coord 1 line number, ...]]
151      ...
152
153
154      x_num = int(lon_total_dist_m / lon_sub_dist_m) + 1
155      y_num = int(lat_total_dist_m / lat_sub_dist_m) + 1
156
157      x_coords = np.linspace(lon_min, lon_max, x_num)
158      y_coords = np.linspace(lat_min, lat_max, y_num)
159
160      if line_dir == HORIZ:
161          x_coord_mesh = np.zeros((y_num, x_num))
162          x_coord_mesh[:,::2, :] = x_coords
163          x_coord_mesh[1::2, :] = x_coords[:-1]
164          x_coord_mesh = x_coord_mesh.flatten()
165
166          y_coord_mesh = np.tile(y_coords, (x_num, 1)).T.flatten()
167
168      else:
169          y_coord_mesh = np.zeros((x_num, y_num))
170          y_coord_mesh[:,::2, :] = y_coords
171          y_coord_mesh[1::2, :] = y_coords[:-1]
172          y_coord_mesh = y_coord_mesh.flatten()
173
174          x_coord_mesh = np.tile(x_coords, (y_num, 1)).T.flatten()
175
176          line_nums = np.sort(np.tile(np.arange(y_num), x_num)) + last_line_num + 1
177
178      return [np.vstack([y_coord_mesh,
179                         x_coord_mesh]), line_nums]
180
181  def gen_timestamps(survey_coords: np.ndarray,
182                      survey_start_dt: dt.datetime,
183                      survey_vel_mps: float) -> np.ndarray:
184      ...
185
186      Generate Unix epoch timestamps for each survey sample based on
187      the survey's start time, path coordinates, and vehicle velocity
188
189      Parameters
190      -----
191      survey_coords
192          2xN array of survey coordinates in lat/lon (dd) —> [ coord 0 lat (dd), coord 1 lat (dd), ... ]
193          [ coord 0 lon (dd), coord 1 lon (dd), ... ]
194      survey_start_dt
195          Start date/time of survey (UTC)
196      survey_vel_mps
197          Velocity of survey vehicle (m/s)

```

```

196
197     Returns
198     -----
199     timestamps
200         Unix epoch timestamps of survey data points
201     ,
202
203     timestamps = np.zeros(survey_coords.shape[1])
204
205     timestamps[0] = survey_start_dt.timestamp() # Unix epoch timestamp
206
207     for i in tqdm(range(1, survey_coords.shape[1])):
208         prev_lat = survey_coords[0, i-1]
209         prev_lon = survey_coords[1, i-1]
210         cur_lat = survey_coords[0, i]
211         cur_lon = survey_coords[1, i]
212
213         timestamps[i] = timestamps[i-1] + cu.coord_dist(prev_lat,
214                                         prev_lon,
215                                         cur_lat,
216                                         cur_lon,) * KM2M / survey_vel_mps
217
218     return timestamps
219
220 def gen_spin_data(out_dir: str,
221                   lat: float,
222                   lon: float,
223                   height: float,
224                   date: dt.datetime,
225                   headings: np.ndarray=np.linspace(0, 720, 1000),
226                   elevations: np.ndarray=np.linspace(0, 7200, 1000),
227                   a: np.ndarray=np.eye(3),
228                   b: np.ndarray=np.zeros(3),
229                   debug: bool=True) -> pd.DataFrame:
230     ,
231     Generate simulated spin test data, save it to a .csv
232     file, and return the data as a Pandas DataFrame
233
234     Parameters
235     -----
236     out_dir
237         Path to directory where the simulated spin test DataFrame will be exported to
238         (set to None to prevent writing out the data to disk)
239     lat
240         Latitude of the simulated spin test (dd)
241     lon
242         Longitude of the simulated spin test (dd)
243     height
244         Height of the simulated spin test above MSL (m)
245     date
246         Date and time of when the simulated spin test began (UTC)
247     headings
248         1xN array of heading angles (Degrees)
249     elevations

```

```

250      1xN array of elevation angles (Degrees)
251      a
252      3x3 vector magnetometer distortion matrix to be applied to the data
253      b
254      1x3 vector magnetometer bias vector to be applied to the data (nT)
255      debug
256      Whether or not debug prints/plot should be generated
257
258      Returns
259      -----
260      pd.DataFrame
261      DataFrame of the simulated spin test data
262      ,,
263
264      if debug:
265          print('Generating_simulated_spin_test_data')
266
267      rolls = np.zeros(headings.shape)
268      eulers = np.hstack([rolls[:, np.newaxis],
269                          elevations[:, np.newaxis],
270                          headings[:, np.newaxis]])
271
272      if debug:
273          print('Generating_perfect_simulated_spin_test_measurements')
274
275      b_true = sc.gen_b_truth_euler(lat, lon, height, date, eulers)
276
277      if debug:
278          print('Applying_vector_distortion_to_simulated_spin_test_data')
279
280      b_dist = sc.apply_dist_to_vec(b_true, a, b)
281
282      b_dist_x = b_dist[:, 0]
283      b_dist_y = b_dist[:, 1]
284      b_dist_z = b_dist[:, 2]
285      b_dist_f = la.norm(b_dist, axis=1)
286
287      if debug:
288          print('Calculating_IGRF_values_for_simulated_spin_test')
289
290      IGRF = sc.b_earth_ned_igrf(lat, lon, height, date)
291
292      IGRF_x = IGRF[0]
293      IGRF_y = IGRF[1]
294      IGRF_z = IGRF[2]
295      IGRF_f = la.norm(IGRF)
296
297      if debug:
298          sc.plot_spin_data(b_true,
299                             b_dist)
300
301      print('Exporting_simulated_spin_test_data_as_a_CSV')
302
303      data = {'datetime': date,

```

```

304         'epoch_sec': date.timestamp(),
305         'LAT': lat,
306         'LONG': lon,
307         'ALT': height,
308         'PITCH': elevations,
309         'ROLL': rolls,
310         'AZIMUTH': headings,
311         'X': b_dist_x,
312         'Y': b_dist_y,
313         'Z': b_dist_z,
314         'F': b_dist_f,
315         'IGRF_X': IGRF_x,
316         'IGRF_Y': IGRF_y,
317         'IGRF_Z': IGRF_z,
318         'IGRF_F': IGRF_f}
319
320     if out_dir is not None:
321         return save_dataset(type = 'spin',
322                             out_dir = out_dir,
323                             date = date,
324                             data = data)
325
326
327     def gen_TL_data(out_dir: str,
328                     center_lat: float,
329                     center_lon: float,
330                     height: float,
331                     start_dt: dt.datetime,
332                     box_xlen_m: float,
333                     box_ylen_m: float,
334                     c: np.ndarray,
335                     vel_mps: float=20,
336                     sample_hz: float=50,
337                     dither_hz: float=1,
338                     dither_amp: float=10,
339                     terms: int=tl.ALL_TERMS,
340                     a: np.ndarray=np.eye(3),
341                     b: np.ndarray=np.zeros(3),
342                     debug: bool=True) -> pd.DataFrame:
343
344     """
345     Generate simulated Tolles-Lawson flight data, save it to a .csv
346     file, and return the data as a Pandas DataFrame
347
348     Parameters
349     -----
350         out_dir
351             Path to directory where the simulated Tolles-Lawson flight DataFrame will be exported to
352             (set to None to prevent writing out the data to disk)
353         center_lat
354             Latitude of the center of the simulated Tolles-Lawson box flight (dd)
355         center_lon
356             Longitude of the center of the simulated Tolles-Lawson box flight (dd)
357         height
358             Height above MSL of the simulated Tolles-Lawson box flight (m)

```

```

358     start_dt
359         Date and time of when the simulated Tolles–Lawson flight began "collecting" data (UTC)
360     box_xlen_m
361         Total distance simulated Tolles–Lawson box flight spans in the x/East direction (m)
362     box_ylen_m
363         Total distance simulated Tolles–Lawson box flight spans in the y/North direction (m)
364     c
365         Tolles–Lawson coefficients used to distort the scalar magnetometer data. The number
366         of coefficients must correspond to the terms specified by 'terms'
367     vel_mps
368         Velocity of the plane in the simulated Tolles–Lawson flight (m/s)
369     sample_hz
370         Sample rate of the sensors for the simulated Tolles–Lawson flight (Hz)
371     dither_hz
372         Frequency at which the "plane" oscilates pitch, roll, and azimuth for the simulated
373         Tolles–Lawson flight (Hz)
374     dither_amp
375         Amplitude at which the "plane" oscilates pitch, roll, and azimuth for the simulated
376         Tolles–Lawson flight (Degrees)
377     terms
378         Specify which terms to use to distort the data. Options include:
379
380         - ALL_TERMS
381         - PERMANENT
382         - INDUCED
383         - EDDY
384
385     a
386         3x3 distortion matrix
387     b
388         1x3 bias vector
389     debug
390         Whether or not debug prints/plot should be generated
391
392     Returns
393     -----
394     pd.DataFrame
395         DataFrame of the simulated Tolles–Lawson flight data
396     ,
397
398     if debug:
399         print('Generating simulated TL calibration flight data')
400
401     # General setup
402     delta_t      = 1 / sample_hz
403     sample_dist_m = vel_mps / sample_hz
404     num_xsamples = int(box_xlen_m / sample_dist_m)
405     num_ysamples = int(box_ylen_m / sample_dist_m)
406     num_samples   = (2 * num_xsamples) + (2 * num_ysamples)
407     datetimes    = [start_dt + dt.timedelta(seconds=(delta_t * i)) for i in range(num_samples)]
408     timestamps   = [datetime.timestamp() for datetime in datetimes]
409
410     # Calculate lat/lon coordinates for box flight
411     _, x_min = cu.coord_coord(center_lat, center_lon, box_xlen_m * MKM / 2, 270)

```

```

412     __, x_max = cu.coord_coord(center_lat, center_lon, box_xlen_m * MKM / 2, 90)
413     y_min, __ = cu.coord_coord(center_lat, center_lon, box_ylen_m * MKM / 2, 180)
414     y_max, __ = cu.coord_coord(center_lat, center_lon, box_ylen_m * MKM / 2, 0)
415
416     x_coords = np.linspace(x_min, x_max, num_xsamples)
417     y_coords = np.linspace(y_min, y_max, num_ysamples)
418
419     #           Fly LLHC to ULHC          Fly ULHC to URHC          Fly URHC to LRHC          Fly LRHC to LLHC
420     #--> LLHC = Lower Left Hand Corner, etc...
421     lats = np.array(list(y_coords) + ([y_max] * num_xsamples) + list(y_coords[::-1])) + ([y_min] *
422     num_xsamples))
422
423     if debug:
424         print('Calculating IGRF values for simulated TL calibration flight')
425
426     # Calculate IGRF values for flight
427     IGRF = sc.b_earth_ned_igrf(lats, lons, height, start_dt)
428
429     IGRF_x = IGRF[:, 0]
430     IGRF_y = IGRF[:, 1]
431     IGRF_z = IGRF[:, 2]
432     IGRF_f = la.norm(IGRF, axis=1)
433
434     # Calculate euler angles for flight
435     dither_xlen_m = box_xlen_m / 3 # Divide each side of the box into 3 sections since we need to independently dither
436     pitch, roll, and azimuth
437     dither_ylen_m = box_ylen_m / 3
438
439     xdither_num_samples = int(dither_xlen_m / sample_dist_m) - 1
440     ydither_num_samples = int(dither_ylen_m / sample_dist_m) - 1
441
442     xdither_t = np.linspace(0,
443                             xdither_num_samples * delta_t,
444                             xdither_num_samples)
445     xdither = dither_amp * np.sin(2 * np.pi * dither_hz * xdither_t)
446
447     ydither_t = np.linspace(0,
448                             ydither_num_samples * delta_t,
449                             ydither_num_samples)
450     ydither = dither_amp * np.sin(2 * np.pi * dither_hz * ydither_t)
451
452     if debug:
453         print('Dithering orientation angles ({}.Hz, ±{}°)'.format(dither_hz, dither_amp))
454         print('Dithering pitch angles')
455
456     xpitch = np.zeros(num_xsamples)
457     ypitch = np.zeros(num_ysamples)
458
459     xpitch[0:len(xdither)] = xdither
460     ypitch[0:len(ydither)] = ydither
461
462     pitch = np.hstack([xpitch, ypitch, xpitch, ypitch])

```

```

462
463     if debug:
464         print('Dithering_roll_angles')
465
466     xroll = np.zeros(num_xsamples)
467     yroll = np.zeros(num_ysamples)
468
469     xroll[len(xdither):len(xdither)*2] = xdither
470     yroll[len(ydither):len(ydither)*2] = ydither
471
472     roll = np.hstack([xroll, yroll, xroll, yroll])
473
474     if debug:
475         print('Dithering_azimuth_angles')
476
477     xazimuth = np.zeros(num_xsamples)
478     yazimuth = np.zeros(num_ysamples)
479
480     xazimuth[len(xdither)*2:len(xdither)*3] = xdither
481     yazimuth[len(ydither)*2:len(ydither)*3] = ydither
482
483     azimuth = np.hstack([xazimuth + 0, # Add biases for the fact that each side of the box has the plane fly different
484                          average azimuths
485                          yazimuth + 90,
486                          xazimuth + 180,
487                          yazimuth + 270])
488
489     eulers = np.vstack([roll, pitch, azimuth]).T
490
491     if debug:
492         print('Generating_true_TL_readings (assuming_no_anomaly_only_IGRF_is_used)')
493
494     b_scalar_true = IGRF_f
495     b_vector      = mu.ned2body(IGRF, eulers)
496
497     if debug:
498         print('Applying_TL_distortion_to_simulated_calibration_flight_data')
499
500     b_scalar_dist = tl.apply_tl_dist(c,
501                                         b_scalar_true,
502                                         b_vector,
503                                         delta_t,
504                                         None,
505                                         terms=terms)
506
507     if debug:
508         print('Applying_spin_test_distortion_to_simulated_simulated_TL_readings')
509
510     IGRF = sc.apply_dist_to_vec(IGRF, a, b)
511
512     b_dist_x = b_vector[:, 0]
513     b_dist_y = b_vector[:, 1]
514     b_dist_z = b_vector[:, 2]

```

```

515     if debug:
516         _, (ax1, ax2) = plt.subplots(1, 2)
517
518         ax1.set_title('TL_Box_Flight_Angles')
519         ax1.plot(datetimes, pitch, label='Pitch')
520         ax1.plot(datetimes, roll, label='Roll')
521         ax1.plot(datetimes, azimuth, label='Azimuth')
522         ax1.set_ylabel('')
523         ax1.set_xlabel('Date')
524         ax1.legend()
525         ax1.grid()
526
527         ax2.set_title('TL_Magnetometer_Data')
528         ax2.plot(datetimes, b_scalar_true, label='Original_Scalar_Data')
529         ax2.plot(datetimes, b_scalar_dist, linestyle='dashed', label='Distorted_Scalar_Data')
530         ax2.set_ylabel('nT')
531         ax2.set_xlabel('Date')
532         ax2.legend()
533         ax2.grid()
534
535     print('Exporting_simulated_TL_flight_data_as_a_CSV')
536
537     # Save results
538     data = { 'datetime': datetimes,
539             'epoch_sec': timestamps,
540             'LAT': lats,
541             'LONG': lons,
542             'ALT': height,
543             'PITCH': pitch,
544             'ROLL': roll,
545             'AZIMUTH': azimuth,
546             'X': b_dist_x,
547             'Y': b_dist_y,
548             'Z': b_dist_z,
549             'F': b_scalar_dist,
550             'IGRF_X': IGRF_x,
551             'IGRF_Y': IGRF_y,
552             'IGRF_Z': IGRF_z,
553             'IGRF_F': IGRF_f}
554
555     if out_dir is not None:
556         return save_dataset(type = 'tl',
557                             out_dir = out_dir,
558                             date = start_dt,
559                             data = data)
560
561     return pd.DataFrame(data)
562
562 def gen_ref_station_data(out_dir: str,
563                           lat: float,
564                           lon: float,
565                           height: float,
566                           start_dt: dt.datetime,
567                           dur_s: float,
568                           scale: float=1,

```

```

569             offset: float=0,
570             awgn_std: float=0.1,
571             sample_hz: float=1,
572             file_df: pd.DataFrame=None,
573             debug: bool=True) -> pd.DataFrame:
574     """
575     Generate simulated reference station data, save it to a .csv file, and return
576     the data as a Pandas DataFrame
577
578     Parameters
579     -----
580     out_dir
581         Path to directory where the simulated reference station DataFrame will be exported to
582         (set to None to prevent writing out the data to disk)
583     lat
584         Latitude of the simulated reference station (dd)
585     lon
586         Longitude of the simulated reference station (dd)
587     height
588         Height of the simulated reference station above MSL (m)
589     start_dt
590         Date of when the simulated reference station began "collecting" data (UTC)
591     dur_s
592         Number of seconds the simulated reference station "collected" data (s)
593     scale
594         Inversely scale the simulated reference data by this factor
595     offset
596         Negatively bias the simulated reference data by this amount (mT)
597     awgn_std
598         If 'type & AWGN', add random noise with this standard deviation to the data
599     sample_hz
600         Sample rate of simulated reference station (Hz)
601     file_df
602         If 'type & FILE', add data from this DataFrame to the diurnal data
603     debug
604         Whether or not debug prints/plot should be generated
605
606     Returns
607     -----
608     pd.DataFrame
609         DataFrame of the simulated reference station data
610     ...
611
612     if debug:
613         print('Generating_simulated_reference_station_data')
614
615     t_start      = start_dt.timestamp()
616     t_stop       = t_start + dur_s
617     timestamps   = np.linspace(t_start, t_stop, (dur_s * sample_hz) + 1)
618     num_timestamps = len(timestamps)
619     datetimes    = np.array([dt.datetime.fromtimestamp(t) for t in timestamps])
620
621     ref_vec = np.zeros((num_timestamps, 3))
622     ref_mag = np.zeros(num_timestamps)

```

```

623
624     if file_df is not None:
625         _, file_ref_mag = Diurnal.interp_reference_df(df
626                                         = file_df,
627                                         timestamps = timestamps,
628                                         survey_lon = None,
629                                         subtract_core = True)
630
631     if debug:
632         print('Incorporating_file_data into simulated_reference_data')
633
634     ref_mag += file_ref_mag
635
636     if debug:
637         print('Incorporating_scale_and_offset_into_simulated_reference_data')
638
639     ref_mag = Diurnal.apply_dist(x = [offset, scale],
640                                 data = ref_mag)
641
642     if debug:
643         print('Incorporating AWGN into simulated_reference_data')
644
645     awgn_ref_vec = np.random.normal(0, awgn_std, ref_vec.shape)
646     awgn_ref_mag = la.norm(awgn_ref_vec, axis=1)
647
648     ref_mag += awgn_ref_mag
649
650     if debug:
651         print('Calculating IGRF values at simulated reference station')
652
653     IGRF = sc.b_earth_ned_igrf(lat, lon, height, start_dt)
654
655     IGRF_x = IGRF[0]
656     IGRF_y = IGRF[1]
657     IGRF_z = IGRF[2]
658     IGRF_f = la.norm(IGRF)
659
660     IGRF_dcs = IGRF / IGRF_f
661
662     if debug:
663         print('Adding IGRF core field to simulated reference station scalar values')
664
665     ref_mag += IGRF_f
666
667     if debug:
668         print('Projecting simulated reference station scalar values into vector values using IGRF direction cosines')
669
670     x = ref_mag * IGRF_dcs[0]
671     y = ref_mag * IGRF_dcs[1]
672     z = ref_mag * IGRF_dcs[2]
673     f = ref_mag
674
675     if debug:
676         plt.plot(datetimes, f)
677         plt.title('Total Simulated Reference Station Scalar Values')

```

```

677         plt.ylabel('nT')
678         plt.xlabel('Date')
679         plt.grid()
680
681     print('Exporting simulated reference station data as a CSV')
682
683     data = { 'datetime': datetimes,
684             'epoch_sec': timestamps,
685             'LAT': lat,
686             'LONG': lon,
687             'ALT': height,
688             'PITCH': 0,
689             'ROLL': 0,
690             'AZIMUTH': 0,
691             'X': x,
692             'Y': y,
693             'Z': z,
694             'F': f,
695             'IGRF_X': IGRF_x,
696             'IGRF_Y': IGRF_y,
697             'IGRF_Z': IGRF_z,
698             'IGRF_F': IGRF_f}
699
700     if out_dir is not None:
701         return save_dataset(type = 'ref',
702                             out_dir = out_dir,
703                             date = start_dt,
704                             data = data)
705
706     return pd.DataFrame(data)
707
708     def gen_sim_map(out_dir: str,
709                     location: str,
710                     center_lat: float,
711                     center_lon: float,
712                     dx_m: float,
713                     dy_m: float,
714                     x_dist_m: float,
715                     y_dist_m: float,
716                     height: float,
717                     date: dt.datetime,
718                     anomaly_locs: np.ndarray,
719                     anomaly_scales: np.ndarray,
720                     anomaly_covs: np.ndarray,
721                     upcontinue: bool=False,
722                     debug: bool=True) -> rxr.rioxarray.raster_dataset.xarray.DataArray:
723
724     Generate a simulated anomaly map as a multi-band GeoTIFF file in
725     WGS-84 coordinates. Bands include:
726
727     - Band 0: Scalar anomaly values (nT)
728     - Band 1: x/North vector anomaly values (nT)
729     - Band 2: y/East vector anomaly values (nT)
730     - Band 3: z/Down vector anomaly values (nT)
731     - Band 3: Pixel height values (m)

```

```

731
732     Parameters
733     -----
734     out_dir
735         Path to directory where the GeoTIFF will be exported to
736         (set to None to prevent writing out the data to disk)
737     location
738         Description of survey area
739     center_lat
740         Latitude of the center of the map (dd)
741     center_lon
742         Longitude of the center of the map (dd)
743     dx_m
744         X pixel size (m)
745     dy_m
746         Y pixel size (m)
747     x_dist_m
748         Total distance map spans in the x/East direction (m)
749     y_dist_m
750         Total distance map spans in the y/North direction (m)
751     height
752         Height above MSL of the level map (m)
753     date
754         Date of when the survey was "collected" (UTC)
755     anomaly_locs
756         2xN array of lat/lon coordinates of all magnetic anomalies
757         in the map area --> [anomaly 0 lat (dd), anomaly 1 lat (dd), ...]
758         [anomaly 0 lon (dd), anomaly 1 lon (dd), ...]
759     anomaly_scales
760         1xN array of scale values to control the amplitudes of
761         all magnetic anomalies in the map area at MSL (NOT at 'height')
762     anomaly_covs
763         Nx2x2 array of covariance matrices that control the spread
764         direction and spread amount of all magnetic anomalies in the
765         map area
766     upcontinue
767         Whether or not to create the initial map at MSL and upcontinue
768         to the specified height (otherwise simply leave the initial
769         map as the final map)
770     debug
771         Whether or not debug prints/plot should be generated
772
773     Returns
774     -----
775     rrx.rioxarray.raster_dataset.xarray.DataArray
776         Map as a rioxarray
777         ...
778
779     if debug:
780         print('Generating_simulated_anomaly_map')
781
782     # Calculate number of samples and lat/lon coordinates for map
783     num_xsamples = int(x_dist_m / dx_m)
784     num_ysamples = int(y_dist_m / dy_m)

```

```

785
786     _, min_lon = cu.coord_coord(center_lat, center_lon, x_dist_m * MKM / 2, 270)
787     _, max_lon = cu.coord_coord(center_lat, center_lon, x_dist_m * MKM / 2, 90)
788     min_lat, _ = cu.coord_coord(center_lat, center_lon, y_dist_m * MKM / 2, 180)
789     max_lat, _ = cu.coord_coord(center_lat, center_lon, y_dist_m * MKM / 2, 0)
790
791     lons = np.linspace(min_lon, max_lon, num_xsamples)
792     lats = np.linspace(min_lat, max_lat, num_ysamples)
793
794     num_anomalies = anomaly_locs.shape[1]
795     anomaly_lats = anomaly_locs[0]
796     anomaly_lons = anomaly_locs[1]
797
798     x, y = np.meshgrid(lons, lats)
799
800     scalar = np.zeros((len(lats), len(lons)))
801
802     if debug:
803         print('Processing simulated anomaly structures at MSL')
804
805     # Add each anomaly field to the map individually by spatially
806     # sampling a 2D Gaussian distribution that represents the
807     # given anomaly
808     for i in range(num_anomalies):
809         pos = np.zeros((*scalar.shape, 2))
810         pos[:, :, 0] = x
811         pos[:, :, 1] = y
812
813         rv = stat.multivariate_normal([anomaly_lons[i], anomaly_lats[i]], anomaly_covs[i])
814         rv_norm = rv.pdf(pos) / rv.pdf(pos).max()
815
816         new_anomaly_vals = anomaly_scales[i] * rv_norm
817
818         scalar += new_anomaly_vals
819
820     if upcontinue:
821         if debug:
822             plt.figure()
823             plt.title('Simulated_Total_Anomaly_Map_at_MSL')
824             plt.pcolor(x, y, scalar, shading='nearest', cmap=cm.coolwarm)
825             plt.ylabel('Latitude(dd)')
826             plt.xlabel('Longitude(dd)')
827             plt.colorbar()
828
829         print('Upward_continuing_map_to_{0}m'.format(height))
830
831         scalar = mu.upcontinue(scalar, dx_m, dy_m, height)
832
833         if debug:
834             plt.figure()
835             plt.title('Simulated_Total_Anomaly_Map_at_{0}m_MSL'.format(height))
836             plt.pcolor(x, y, scalar, shading='nearest', cmap=cm.coolwarm)
837             plt.ylabel('Latitude(dd)')
838             plt.xlabel('Longitude(dd)')

```

```

839         plt.colorbar()
840
841     else:
842         plt.figure()
843         plt.title('Simulated_Total_Anomaly_Map_at_{}m MSL'.format(height))
844         plt.pcolormesh(x, y, scalar, shading='nearest', cmap=cm.coolwarm)
845         plt.ylabel('Latitude_(dd)')
846         plt.xlabel('Longitude_(dd)')
847         plt.colorbar()
848
849     if debug:
850         print('Calculating_IGRF_values_for_simulated_map')
851
852     # Calculate IGRF values for map
853     mesh_lons, mesh_lats = np.meshgrid(lons, lats)
854     Be, Bn, Bu = igrf(mesh_lons, mesh_lats, height * MM, date)
855
856     IGRF = np.zeros((3, *scalar.shape))
857     IGRF[0, :, :] = Bn.squeeze()
858     IGRF[1, :, :] = Be.squeeze()
859     IGRF[2, :, :] = -Bu.squeeze()
860
861     IGRF_x = IGRF[0, :, :]
862     IGRF_y = IGRF[1, :, :]
863     IGRF_z = IGRF[2, :, :]
864     IGRF_f = la.norm(IGRF, axis=0)
865
866     if debug:
867         print('Projecting_simulated_map_scalar_measurements_into_vector_measurements_using_IGRF_direction_cosines')
868
869     IGRF_dcs = np.zeros(IGRF.shape)
870     IGRF_dcs[0, :, :] = IGRF_x / IGRF_f
871     IGRF_dcs[1, :, :] = IGRF_y / IGRF_f
872     IGRF_dcs[2, :, :] = IGRF_z / IGRF_f
873
874     vector = np.zeros(IGRF.shape)
875     vector[0, :, :] = scalar * IGRF_dcs[0, :, :]
876     vector[1, :, :] = scalar * IGRF_dcs[1, :, :]
877     vector[2, :, :] = scalar * IGRF_dcs[2, :, :]
878
879     if debug:
880         print('Exporting_simulated_map_as_a_GeoTIFF')
881
882     map = mu.export_map(out_dir = out_dir,
883                         location = location,
884                         date = date,
885                         lats = lats,
886                         lons = lons,
887                         scalar = scalar,
888                         heights = height,
889                         stds = None,
890                         vector = vector)
891
892     if debug:

```

```

893     plt.figure()
894     map[0].plot(cmap=cm.coolwarm)
895     plt.title('Final_Simulated_Scalar_Anomaly_Map_From_GeoTIFF')
896
897     plt.figure()
898     map[6].plot(cmap=cm.coolwarm)
899     plt.title('Final_Simulated_X-Gradient_Map_From_GeoTIFF')
900
901     plt.figure()
902     map[7].plot(cmap=cm.coolwarm)
903     plt.title('Final_Simulated_Y-Gradient_Map_From_GeoTIFF')
904
905     plt.figure()
906     map[4].plot(cmap=cm.coolwarm)
907     plt.title('Final_Simulated_Height_Map_From_GeoTIFF')
908
909     return map
910
911 def gen_survey_data(out_dir: str,
912                     map: rxr.rioxarray.raster_dataset.xarray.DataArray,
913                     survey_height_m: float,
914                     survey_start_dt: dt.datetime,
915                     survey_vel_mps: float,
916                     survey_e_buff_m: float,
917                     survey_w_buff_m: float,
918                     survey_n_buff_m: float,
919                     survey_s_buff_m: float,
920                     sample_hz: float,
921                     ft_line_dist_m: float,
922                     ft_line_dir: int=HORIZ,
923                     a: np.ndarray=np.eye(3),
924                     b: np.ndarray=np.zeros(3),
925                     c: np.ndarray=np.zeros(18),
926                     terms: int=t1.ALL_TERMS,
927                     scalar_awgn_std: float=0,
928                     diurnal_df: pd.DataFrame=None,
929                     diurnal_dist: np.ndarray=None,
930                     use_tie_lines: bool=False,
931                     tie_dist_m: float=None,
932                     debug: bool=True) -> pd.DataFrame:
933     ...
934
935     Generate simulated flight survey data based on several
936     parameters
937
938     Parameters
939     -----
940     out_dir
941         Path to directory where the simulated survey data
942         will be exported to
943         (set to None to prevent writing out the data to disk)
944     map
945         Rioxarray of the "truth map" used to generate the
946         simulated anomaly samples
947     survey_height_m

```

```

947      Height of the simulated survey MSL (m)
948      survey_start_dt
949          Start datetime of the simulated survey (UTC)
950      survey_vel_mps
951          Simulated survey vehicle velocity (m/s)
952      survey_e_buff_m
953          Buffer distance between the East extreme of the map
954          and the East extreme of the survey samples (m)
955      survey_w_buff_m
956          Buffer distance between the West extreme of the map
957          and the West extreme of the survey samples (m)
958      survey_n_buff_m
959          Buffer distance between the North extreme of the map
960          and the North extreme of the survey samples (m)
961      survey_s_buff_m
962          Buffer distance between the South extreme of the map
963          and the South extreme of the survey samples (m)
964      sample_hz
965          Sample rate of the simulated magnetometers (hz)
966      ft_line_dist_m
967          Distance between flight lines (m)
968      ft_line_dir
969          Direction the flight lines run. Tie lines (if used)
970          will be oriented in the orthogonal direction.
971          Options include:
972
973          - HORIZ: Horizontal - East/West
974          - VERT: Vertical - North/South
975
976      a
977          3x3 distortion matrix
978      b
979          1x3 bias vector
980      c
981          Tolles-Lawson coefficients used to distort the scalar magnetometer data. The number
982          of coefficients must correspond to the terms specified by 'terms'
983      terms
984          Terms to include in A-matrix. Options include:
985
986          - ALL_TERMS
987          - PERMANENT
988          - INDUCED
989          - EDDY
990
991      scalar_awgn_std
992          Standard deviation of the AWGN to add to the survey
993          magnetometer measurements
994      diurnal_df
995          DataFrame of reference station data
996      diurnal_dist
997          1x2 array of distortion parameters to apply to
998          diurnal/reference station data —> [offset (nT), scale]
999      use_tie_lines
1000         Whether or not to generate tie line samples

```

```

1001     tie_dist_m
1002         Distance between tie lines (m)
1003     debug
1004         Whether or not debug prints/plot should be generated
1005
1006     Returns
1007     -----
1008     pd.DataFrame
1009         DataFrame containing the simulated survey data
1010     ,
1011
1012     if debug:
1013         print('Generating_simulated_survey_data')
1014
1015     # General setup
1016     delta_t      = 1 / sample_hz
1017     sample_dist_m = survey_vel_mps / sample_hz
1018
1019     if debug:
1020         print('Generating_simulated_survey_flight_path')
1021
1022     map_min_lat = map.y.min().item()
1023     map_max_lat = map.y.max().item()
1024     map_min_lon = map.x.min().item()
1025     map_max_lon = map.x.max().item()
1026
1027     __, suvey_min_lon = cu.coord_coord(lat      = map_min_lat,
1028                                         lon      = map_min_lon,
1029                                         dist    = survey_w_buff_m * MKM
1030                                         bearing = 90)
1031     __, suvey_max_lon = cu.coord_coord(lat      = map_min_lat,
1032                                         lon      = map_max_lon,
1033                                         dist    = survey_e_buff_m * MKM
1034                                         bearing = 270)
1035     suvey_min_lat, __ = cu.coord_coord(lat      = map_min_lat,
1036                                         lon      = map_min_lon,
1037                                         dist    = survey_s_buff_m * MKM
1038                                         bearing = 0)
1039     suvey_max_lat, __ = cu.coord_coord(lat      = map_max_lat,
1040                                         lon      = map_min_lon,
1041                                         dist    = survey_n_buff_m * MKM
1042                                         bearing = 180)
1043
1044     lon_total_dist_m = cu.coord_dist(lat_1 = suvey_min_lat,
1045                                         lon_1 = suvey_min_lon,
1046                                         lat_2 = suvey_min_lat,
1047                                         lon_2 = suvey_max_lon) * KM
1048     lat_total_dist_m = cu.coord_dist(lat_1 = suvey_min_lat,
1049                                         lon_1 = suvey_min_lon,
1050                                         lat_2 = suvey_max_lat,
1051                                         lon_2 = suvey_min_lon) * KM
1052
1053     # Create survey "flight path"
1054     if ft_line_dir == HORIZ: # Flight lines run horizontal

```

```

1055     ft_line_pts, ft_line_nums = gen_parallel_search(lon_min           = suvey_min_lon,
1056                                         lon_max            = suvey_max_lon,
1057                                         lat_min            = suvey_min_lat,
1058                                         lat_max            = suvey_max_lat,
1059                                         lon_total_dist_m = lon_total_dist_m,
1060                                         lat_total_dist_m = lat_total_dist_m,
1061                                         lon_sub_dist_m   = sample_dist_m,
1062                                         lat_sub_dist_m   = ft_line_dist_m,
1063                                         line_dir          = HORIZ)
1064
1065     if use_tie_lines:
1066         tie_line_pts, tie_line_nums = gen_parallel_search(lon_min           = suvey_min_lon,
1067                                         lon_max            = suvey_max_lon,
1068                                         lat_min            = suvey_min_lat,
1069                                         lat_max            = suvey_max_lat,
1070                                         lon_total_dist_m = lon_total_dist_m,
1071                                         lat_total_dist_m = lat_total_dist_m,
1072                                         lon_sub_dist_m   = tie_dist_m,
1073                                         lat_sub_dist_m   = sample_dist_m,
1074                                         line_dir          = VERT,
1075                                         last_line_num    = ft_line_nums[-1])
1076
1077     elif ft_line_dir == VERT: # Flight lines run vertical
1078         ft_line_pts, ft_line_nums = gen_parallel_search(lon_min           = suvey_min_lon,
1079                                         lon_max            = suvey_max_lon,
1080                                         lat_min            = suvey_min_lat,
1081                                         lat_max            = suvey_max_lat,
1082                                         lon_total_dist_m = lon_total_dist_m,
1083                                         lat_total_dist_m = lat_total_dist_m,
1084                                         lon_sub_dist_m   = ft_line_dist_m,
1085                                         lat_sub_dist_m   = sample_dist_m,
1086                                         line_dir          = VERT)
1087
1088     if use_tie_lines:
1089         tie_line_pts, tie_line_nums = gen_parallel_search(lon_min           = suvey_min_lon,
1090                                         lon_max            = suvey_max_lon,
1091                                         lat_min            = suvey_min_lat,
1092                                         lat_max            = suvey_max_lat,
1093                                         lon_total_dist_m = lon_total_dist_m,
1094                                         lat_total_dist_m = lat_total_dist_m,
1095                                         lon_sub_dist_m   = sample_dist_m,
1096                                         lat_sub_dist_m   = tie_dist_m,
1097                                         line_dir          = HORIZ,
1098                                         last_line_num    = ft_line_nums[-1])
1099
1100    if use_tie_lines:
1101        survey_coords    = np.hstack([ft_line_pts, tie_line_pts])
1102        survey_line_nums = np.hstack([ft_line_nums, tie_line_nums])
1103        line_types       = np.hstack([np.ones(len(ft_line_nums)), np.zeros(len(tie_line_nums))])
1104
1105    else:
1106        survey_coords    = ft_line_pts
1107        survey_line_nums = ft_line_nums
1108        line_types       = np.ones(len(ft_line_nums))

```

```

1109
1110     lats = survey_coords[0, :]
1111     lons = survey_coords[1, :]
1112
1113     if debug:
1114         plt.figure()
1115         map[0].plot(cmap=cm.coolwarm)
1116         plt.title('Survey Flight Path')
1117         plt.plot(lons, lats, label='Flight Path')
1118         plt.plot(lons, lats, 'x', label='Sample Points')
1119         plt.legend()
1120
1121     # More setup stuff
1122     num_samples = survey_coords.shape[1]
1123     datetimes = [survey_start_dt + dt.timedelta(seconds=(delta_t * i)) for i in range(num_samples)]
1124
1125     survey_mag = np.zeros(num_samples)
1126     pitches = np.zeros(num_samples)
1127     rolls = np.zeros(num_samples)
1128     azimuths = np.zeros(num_samples)
1129
1130     # Generate a raster map of the IGRF field in the survey area
1131     IGRF_map = mu.igrf_WGS84(map_WGS84 = map,
1132                               map_alt_m = survey_height_m,
1133                               survey_date = survey_start_dt)
1134     IGRF_vec = np.zeros((survey_coords.shape[1], 3))
1135
1136     if debug:
1137         print('Calculating simulated survey scalar anomaly, azimuth, and IGRF values')
1138
1139     # This loop samples both the anomaly and IGRF field maps and also
1140     # calculates the azimuth in degrees between the current and next
1141     # flight coordinate
1142     for i in tqdm(range(num_samples)):
1143         x = lons[i]
1144         y = lats[i]
1145
1146         survey_mag[i] = mu.sample_map(map = map,
1147                                       x = x,
1148                                       y = y,
1149                                       band = mu.SCALAR)
1150
1151     try:
1152         next_x = lons[i + 1]
1153         next_y = lats[i + 1]
1154
1155         azimuths[i] = cu.coord_bearing(lat_1 = y,
1156                                         lon_1 = x,
1157                                         lat_2 = next_y,
1158                                         lon_2 = next_x)
1159     except IndexError:
1160         azimuths[i] = azimuths[i - 1]
1161
1162     IGRF_vec[i, 0] = mu.sample_map(map = IGRF_map,

```

```

1163             x     = x,
1164             y     = y,
1165             band = mu.VEC_X) # x/North IGRF component (nT)
1166             IGRF_vec[i, 1] = mu.sample_map(map = IGRF_map,
1167                                         x     = x,
1168                                         y     = y,
1169                                         band = mu.VEC_Y) # y/East IGRF component (nT)
1170             IGRF_vec[i, 2] = mu.sample_map(map = IGRF_map,
1171                                         x     = x,
1172                                         y     = y,
1173                                         band = mu.VEC_Z) # z/Down IGRF component (nT)
1174
1175         if debug:
1176             plt.figure()
1177             plt.title('Perfect_Scalar_Anomaly_Survey_Measurements')
1178             plt.plot(datetimes, survey_mag)
1179             plt.ylabel('nT')
1180             plt.xlabel('Date')
1181             plt.grid()
1182
1183         IGRF_x = IGRF_vec[:, 0]
1184         IGRF_y = IGRF_vec[:, 1]
1185         IGRF_z = IGRF_vec[:, 2]
1186         IGRF_f = la.norm(IGRF_vec, axis=1)
1187
1188         IGRF_dcs = np.zeros(IGRF_vec.shape)
1189         IGRF_dcs[:, 0] = IGRF_x / IGRF_f
1190         IGRF_dcs[:, 1] = IGRF_y / IGRF_f
1191         IGRF_dcs[:, 2] = IGRF_z / IGRF_f
1192
1193         if debug:
1194             print('Adding_core_field_to_simulated_survey_scalar_measurements')
1195
1196         survey_mag += IGRF_f
1197
1198         if debug:
1199             plt.figure()
1200             plt.title('Perfect_Scalar_Total_Field_Survey_Measurements')
1201             plt.plot(datetimes, survey_mag)
1202             plt.ylabel('nT')
1203             plt.xlabel('Date')
1204             plt.grid()
1205
1206         timestamps = gen_timestamps(survey_coords = survey_coords,
1207                                     survey_start_dt = survey_start_dt,
1208                                     survey_vel_mps = survey_vel_mps)
1209
1210         # Apply diurnal noise if diurnal data is provided
1211         if diurnal_df is not None:
1212             _, diurnal_mag = Diurnal.interp_reference_df(df = diurnal_df,
1213                                               timestamps = timestamps,
1214                                               survey_lon = None,
1215                                               subtract_core = True)
1216

```

```

1217     # Apply scale and offset to diurnal data if given
1218     if diurnal_dist is not None:
1219         diurnal_mag = Diurnal.apply_dist(x    = diurnal_dist,
1220                                         data = diurnal_mag)
1221
1222     if debug:
1223         print('Adding_diurnal_to_simulated_survey_scalar_measurements')
1224
1225     survey_mag += diurnal_mag
1226
1227     if debug:
1228         plt.figure()
1229         plt.title('Scalar_Total_Field_Survey_Measurements_Durnal')
1230         plt.plot(datetimes, survey_mag)
1231         plt.ylabel('nT')
1232         plt.xlabel('Date')
1233         plt.grid()
1234
1235     if scalar_awgn_std != 0:
1236         if debug:
1237             print('Adding_AWGN_to_simulated_survey_scalar_measurements')
1238
1239         scalar_awgn = np.random.normal(0, scalar_awgn_std, survey_mag.shape)
1240         survey_mag += scalar_awgn
1241
1242     if debug:
1243         plt.figure()
1244         plt.title('Scalar_Total_Field_Survey_Measurements_Durnal_AWGN')
1245         plt.plot(datetimes, survey_mag)
1246         plt.ylabel('nT')
1247         plt.xlabel('Date')
1248         plt.grid()
1249
1250     print('Applying_TL_distortion_to_simulated_survey_scalar_measurements')
1251
1252     survey_mag = tl.apply_tl_dist(c          = c,
1253                                   b_scalar   = survey_mag,
1254                                   b_vector   = IGRF_vec,
1255                                   delta_t    = delta_t,
1256                                   b_external = None,
1257                                   terms      = terms)
1258
1259     if debug:
1260         plt.figure()
1261         plt.title('Scalar_Total_Field_Survey_Measurements_Durnal_AWGN_TL_Distortion')
1262         plt.plot(datetimes, survey_mag)
1263         plt.ylabel('nT')
1264         plt.xlabel('Date')
1265         plt.grid()
1266
1267     if debug:
1268         print('Projecting_simulated_survey_scalar_measurements_into_NED_vector_measurements_using_IGRF_direction_cosines')
1269

```

```

1270      # Compute vector readings by rotating the scalar "measurements" to
1271      # point in the same direction as the IGRF field at that date/time
1272      # and coordinate
1273      survey_vec = np.zeros(IGRF_dcs.shape)
1274      survey_vec[:, 0] = survey_mag * IGRF_dcs[:, 0]
1275      survey_vec[:, 1] = survey_mag * IGRF_dcs[:, 1]
1276      survey_vec[:, 2] = survey_mag * IGRF_dcs[:, 2]
1277
1278      if debug:
1279          print('Rotating_NED_vector(measurements) into sensor\'s body frame')
1280
1281      eulers = np.hstack([rolls[:, np.newaxis],
1282                          pitches[:, np.newaxis],
1283                          azimuths[:, np.newaxis]])
1284
1285      survey_vec = mu.ned2body(ned_vecs=survey_vec,
1286                               eulers=eulers)
1287
1288      if debug:
1289          print('Applying_spin test distortion to simulated survey vector measurements')
1290
1291      survey_vec = sc.apply_dist_to_vec(vec=survey_vec,
1292                                         a=a,
1293                                         b=b)
1294
1295      survey_vec_x = survey_vec[:, 0]
1296      survey_vec_y = survey_vec[:, 1]
1297      survey_vec_z = survey_vec[:, 2]
1298
1299      if debug:
1300          print('Exporting simulated survey data as a CSV')
1301
1302      if debug:
1303          print('Survey_start_datetime/timestamp: {} / {}'.format(datetimes[0], timestamps[0]))
1304          print('Survey_end_datetime/timestamp: {} / {}'.format(datetimes[-1], timestamps[-1]))
1305          print('Flight_line_samples_end_at_timestamp: {} s'.format(timestamps[len(ft_line_pts)])))
1306          print('Survey Duration: {} s'.format(timestamps[-1] - timestamps[0]))
1307
1308      # Save results
1309      data = {'datetime': datetimes,
1310              'epoch_sec': timestamps,
1311              'LAT': lats,
1312              'LONG': lons,
1313              'ALT': survey_height_m,
1314              'LINE': survey_line_nums,
1315              'LINE_TYPE': line_types,
1316              'PITCH': pitches,
1317              'ROLL': rolls,
1318              'AZIMUTH': azimuths,
1319              'X': survey_vec_x,
1320              'Y': survey_vec_y,
1321              'Z': survey_vec_z,
1322              'F': survey_mag,
1323              'IGRF_X': IGRF_x,
1324              'IGRF_Y': IGRF_y,

```

```

1324         'IGRF_Z':     IGRF_z,
1325         'IGRF_F':     IGRF_f}
1326
1327     if out_dir is not None:
1328         return save_dataset(type    = 'survey',
1329                             out_dir = out_dir,
1330                             date    = survey_start_dt,
1331                             data    = data)
1332
1333     return pd.DataFrame(data)

```

## mammal/MAMMAL/Diurnal/\_\_init\_\_.py

```

1 import sys
2 from os.path import dirname, realpath
3
4 import numpy as np
5 import pandas as pd
6 from scipy import interpolate
7 from scipy.optimize import minimize
8
9 sys.path.append(dirname(realpath(__file__)))
10 sys.path.append(dirname(dirname(realpath(__file__))))
11
12 from Utils import Filters
13
14
15 E_ROT_DEG_S = 360.9856 / 24 / 60 / 60 # Earth's rotation rate in degrees/s
16
17
18 def interp_reference_df(df: pd.DataFrame,
19                         timestamps: np.ndarray,
20                         survey_lon: float=None,
21                         ref_scale: float=1,
22                         ref_offset: float=0,
23                         subtract_core: bool=False) -> list:
24     """
25     Generate interpolated reference station data from a given magnetic DataFrame
26
27     Parameters
28     -----
29     df
30         DataFrame of reference station data. DataFram must include the
31         following columns:
32
33         - LONG:      Longitude (dd)
34         - epoch_sec: UNIX epoch timestamp (s)
35         - X:         Magnetic field measurement in the North direction (nT)
36         - Y:         Magnetic field measurement in the East direction (nT)
37         - Z:         Magnetic field measurement in the Down direction (nT)
38         - IGRF_X:   IGRF magnetic field in the North direction (nT)
39         - IGRF_y:   IGRF magnetic field in the East direction (nT)
40         - IGRF_z:   IGRF magnetic field in the Down direction (nT)
41

```

```

42     survey_lon
43         Approximate longitude of the survey area (dd)
44     timestamps
45         1xN array of UNIX epoch timestamps to interpolate the INTERMAGNET
46         data at (s)
47     ref_scale
48         Amount to scale the reference station data by
49     ref_offset
50         Amount to bias the reference station data by (nT)
51     subtract_core
52         Whether or not to subtract IGRF Earth core field from returned reference data
53
54     Returns
55     -----
56     list
57         Interpolated vector and scalar reference station
58         measurements --> [Nx3 vector array (nT), 1xN scalar array (nT)]
59     ...
60
61     if survey_lon is not None:
62         t, f, x, y, z = longitude_norm(df, survey_lon)
63     else:
64         t = np.array(df.epoch_sec)
65
66         x = np.array(df.X)
67         y = np.array(df.Y)
68         z = np.array(df.Z)
69         f = np.array(df.F)
70
71     IGRF_x = np.array(df.IGRF_X)
72     IGRF_y = np.array(df.IGRF_Y)
73     IGRF_z = np.array(df.IGRF_Z)
74     IGRF_f = np.array(df.IGRF_F)
75
76     min_timestamp = timestamps.min()
77     max_timestamp = timestamps.max()
78     min_t = t.min()
79     max_t = t.max()
80
81     assert min_t <= min_timestamp, 'Reference data must start before the start of the survey, is currently off by {}s'.format(min_t - min_timestamp)
82     assert max_t >= max_timestamp, 'Reference data must extend beyond the end of the survey, is currently off by {}s'.format(max_timestamp - max_t)
83
84     interp_x = interpolate.interp1d(t, x, 'linear')
85     interp_y = interpolate.interp1d(t, y, 'linear')
86     interp_z = interpolate.interp1d(t, z, 'linear')
87     interp_f = interpolate.interp1d(t, f, 'linear')
88
89     interp_IGRF_x = interpolate.interp1d(df.epoch_sec, IGRF_x, 'linear')
90     interp_IGRF_y = interpolate.interp1d(df.epoch_sec, IGRF_y, 'linear')
91     interp_IGRF_z = interpolate.interp1d(df.epoch_sec, IGRF_z, 'linear')
92     interp_IGRF_f = interpolate.interp1d(df.epoch_sec, IGRF_f, 'linear')
93

```

```

94     IGRF_x_interp = interp_IGRF_x(timestamps)
95     IGRF_y_interp = interp_IGRF_y(timestamps)
96     IGRF_z_interp = interp_IGRF_z(timestamps)
97     IGRF_f_interp = interp_IGRF_f(timestamps)
98
99     x_interp_no_core = interp_x(timestamps) - IGRF_x_interp
100    y_interp_no_core = interp_y(timestamps) - IGRF_y_interp
101    z_interp_no_core = interp_z(timestamps) - IGRF_z_interp
102    f_interp_no_core = interp_f(timestamps) - IGRF_f_interp
103
104    x_interp_cal_no_core = apply_cal([ref_offset, ref_scale], x_interp_no_core)
105    y_interp_cal_no_core = apply_cal([ref_offset, ref_scale], y_interp_no_core)
106    z_interp_cal_no_core = apply_cal([ref_offset, ref_scale], z_interp_no_core)
107    f_interp_cal_no_core = apply_cal([ref_offset, ref_scale], f_interp_no_core)
108
109    x_interp_cal = x_interp_cal_no_core + IGRF_x_interp
110    y_interp_cal = y_interp_cal_no_core + IGRF_y_interp
111    z_interp_cal = z_interp_cal_no_core + IGRF_z_interp
112    f_interp_cal = f_interp_cal_no_core + IGRF_f_interp
113
114    if subtract_core:
115        ref_vec = np.hstack([x_interp_cal_no_core[:, np.newaxis],
116                             y_interp_cal_no_core[:, np.newaxis],
117                             z_interp_cal_no_core[:, np.newaxis]]))
118        ref_mag = f_interp_cal_no_core
119
120    else:
121        ref_vec = np.hstack([x_interp_cal[:, np.newaxis],
122                             y_interp_cal[:, np.newaxis],
123                             z_interp_cal[:, np.newaxis]])
124        ref_mag = f_interp_cal
125
126    return [ref_vec, ref_mag]
127
128 def longitude_norm(ref_df: pd.DataFrame,
129                      survey_lon: float,
130                      corner_frq: float=1/(3600*3)) -> list:
131    ...
132
133    When using data from a reference station with sufficient
134    difference in longitude from the survey site, the
135    difference in longitude can be accounted for. This is
136    done by time shifting the low frequency content of the
137    reference station data
138
139    Parameters
140    -----
141    ref_df
142        DataFrame of reference station data
143    survey_lon
144        Average longitude of the survey site
145    corner_frq
146        Cutoff frequency that separates which frequencies
147        will be time shifted and which frequencies will
148        retain their original timestamps

```

```

148
149     Returns
150     -----
151     list
152         List of new timestamps and scalar measurements
153         based on the required time shift -> [timestamps (s), scalar (nT), x/North vector (nT), y/East vector (nT), z/
154             Down vector (nT)]
155     ...
156     t = np.array(ref_df.epoch_sec)
157     f = np.array(ref_df.F)
158     x = np.array(ref_df.X)
159     y = np.array(ref_df.Y)
160     z = np.array(ref_df.Z)
161
162     from2toLonDiff = ref_df.LONG.mean() - survey_lon
163     lon_t_offset = pd.Timedelta(seconds=from2toLonDiff / E_ROT_DEG_S)
164
165     lpf_f = Filters.lpf(f, corner_frq, 1, 1)
166     hpf_f = Filters.hpf(f, corner_frq, 1, 1)
167
168     lpf_x = Filters.lpf(x, corner_frq, 1, 1)
169     hpf_x = Filters.hpf(x, corner_frq, 1, 1)
170
171     lpf_y = Filters.lpf(y, corner_frq, 1, 1)
172     hpf_y = Filters.hpf(y, corner_frq, 1, 1)
173
174     lpf_z = Filters.lpf(z, corner_frq, 1, 1)
175     hpf_z = Filters.hpf(z, corner_frq, 1, 1)
176
177     lpf_t = t + lon_t_offset.total_seconds()
178
179     interp_lpf_f = interpolate.interp1d(lpf_t, lpf_f, 'cubic')
180     interp_hpf_f = interpolate.interp1d(t, hpf_f, 'cubic')
181
182     interp_lpf_x = interpolate.interp1d(lpf_t, lpf_x, 'cubic')
183     interp_hpf_x = interpolate.interp1d(t, hpf_x, 'cubic')
184
185     interp_lpf_y = interpolate.interp1d(lpf_t, lpf_y, 'cubic')
186     interp_hpf_y = interpolate.interp1d(t, hpf_y, 'cubic')
187
188     interp_lpf_z = interpolate.interp1d(lpf_t, lpf_z, 'cubic')
189     interp_hpf_z = interpolate.interp1d(t, hpf_z, 'cubic')
190
191     combined_t = lpf_t[np.logical_and(lpf_t >= t.min(), lpf_t <= t.max())] # Clip interpolation times
192
193     lpf_interp_f = interp_lpf_f(combined_t)
194     hpf_interp_f = interp_hpf_f(combined_t)
195
196     lpf_interp_x = interp_lpf_x(combined_t)
197     hpf_interp_x = interp_hpf_x(combined_t)
198
199     lpf_interp_y = interp_lpf_y(combined_t)
200     hpf_interp_y = interp_hpf_y(combined_t)

```

```

201     lpf_interp_z = interp_lpf_z(combined_t)
202     hpf_interp_z = interp_hpf_z(combined_t)
203
204     combined_f = lpf_interp_f + hpf_interp_f
205     combined_x = lpf_interp_x + hpf_interp_x
206     combined_y = lpf_interp_y + hpf_interp_y
207     combined_z = lpf_interp_z + hpf_interp_z
208
209     return [combined_t, combined_f, combined_x, combined_y, combined_z]
210
211
212 def apply_dist(x: np.ndarray,
213                 data: np.ndarray) -> np.ndarray:
214     """
215     Apply distortion to diurnal data
216
217     Parameters
218     -----
219     x
220         1xN array of diurnal measurements
221     data
222         1x2 distortion vector -> [offset (nT), scale]
223
224     Returns
225     -----
226     np.ndarray
227         1xN array of distorted diurnal measurements
228     """
229
230     if type(x) == tuple: # Idk why, but the code sometimes makes x a tuple of one element...
231         x = x[0]
232
233     return (data - x[0]) / x[1]
234
235 def apply_cal(x: np.ndarray,
236               data: np.ndarray) -> np.ndarray:
237     """
238     Apply calibration to distorted diurnal data
239
240     Parameters
241     -----
242     x
243         1xN array of distorted diurnal measurements
244     data
245         1x2 calibration vector -> [offset (nT), scale]
246
247     Returns
248     -----
249     np.ndarray
250         1xN array of calibrated diurnal measurements
251     """
252
253     return (x[1] * data) + x[0]
254

```

```

255 def min_func(x: np.ndarray,
256             *args,
257             **kwargs) -> float:
258     """
259     Cost function to minimize when calibrating diurnal measurements
260
261     Parameters
262     -----
263     x
264         1x2 calibration vector -> [offset (nT), scale]
265     *args
266         list of distorted and "true" diurnal measurements -> [b_distorted (nT), b_true (nT)]
267
268     Returns
269     -----
270     float
271         Calibration cost
272     """
273
274     data = args[0]
275     target = args[1]
276
277     guess = apply_cal(x, data)
278     diff = (target - guess)**2
279
280     return diff.sum()
281
282 def calibrate(x0: np.ndarray,
283               localRef: np.ndarray,
284               intMag: np.ndarray) -> np.ndarray:
285     """
286     Find the calibration vector that minimizes the given cost function
287
288     Parameters
289     -----
290     x
291         1x2 calibration vector "initial guess" -> [offset (nT), scale]
292     *args
293         list of distorted and "true" diurnal measurements -> [b_distorted (nT), b_true (nT)]
294
295     Returns
296     -----
297     np.ndarray
298         1x2 calibration vector -> [offset (nT), scale]
299     """
300
301     return minimize(min_func, x0, args=(localRef, intMag), method='Nelder-Mead').x

```

mammal/MAMMAL/VehicleCal/\_\_init\_\_.py

```

1 import os, sys
2
3 sys.path.append(os.path.dirname(os.path.realpath(__file__)))

```

```
4     sys.path.append(os.path.dirname(os.path.dirname(os.path.realpath(__file__))))
```

## mammal/MAMMAL/VehicleCal/magUtilsTL.py

```
1  """
2  tolles_lawson.py
3
4  Implement a standard Tolles-Lawson magnetometer compensation.
5
6  Aaron Nielsen, Didactex, LLC, apn@didactex.com
7  ANT Center, Air Force Institute of Technology
8  """
9  import itertools
10 import typing
11 from enum import Enum
12
13 import numpy as np
14 import scipy.signal
15 from sklearn.linear_model import Ridge
16 from scipy import signal
17
18
19 class Filter:
20     """ this is a simple wrapper class around the scipy.signal filter tools
21     to provide plotting and application methods in a single object """
22
23     def __init__(self, order, fcut, btype='low', ftype='butter', fs=1.0):
24         """
25             argument nomenclature follows scipy.signal.iirfilter
26         """
27         self._order = order
28         self._fcut = fcut
29         self._btype = btype
30         self._ftype = ftype
31         self._fs = fs
32         self._gensos()
33
34     @property
35     def order(self): # pylint: disable=missing-function-docstring
36         return self._order
37
38     @property
39     def fcut(self): # pylint: disable=missing-function-docstring
40         return self._fcut
41
42     @property
43     def btype(self): # pylint: disable=missing-function-docstring
44         return self._btype
45
46     @property
47     def ftype(self): # pylint: disable=missing-function-docstring
48         return self._ftype
49
```

```

50     @property
51     def fs(self): # pylint: disable=missing-function-docstring
52         return self._fs
53
54     @property
55     def sos(self): # pylint: disable=missing-function-docstring
56         return self._sos
57
58     @sos.setter
59     def sos(self, sos):
60         self._sos = sos
61
62     def _gensos(self):
63         sos = signal.iirfilter(self.order, self.fcut, btype=self.btype, ftype=self.ftype, fs=self.fs, output='sos')
64         self.sos = sos
65
66     def filter_response(self, npts=2048):
67         """ return the filter response """
68         w, h = signal.sosfreqz(self.sos, npts, fs=self.fs)
69         return w, h
70
71     def filtfilt(self, sig: np.ndarray, axis=0):
72         """ filter the input signal forward and backward """
73         return signal.sosfiltfilt(self.sos, sig, axis=axis)
74
75
76
77 # list of the default Tolles-Lawson terms
78 class TollesLawsonTerms(Enum):
79     """
80     Enumeration type for the Tolles Lawson Terms
81     """
82     PERMANENT = 0
83     INDUCED = 1
84     EDDY = 2
85     INDUCED_REDUCED = 3
86     EDDY_REDUCED = 4
87     CUBIC = 5
88     CUBIC_REDUCED = 6
89
90
91 DEFAULT_TL_TERMS = (TollesLawsonTerms.PERMANENT, TollesLawsonTerms.INDUCED, TollesLawsonTerms.EDDY)
92
93 # mapping of the Tolles Lawson term to the number of coefficients
94 TollesLawsonTermsToCoefficientNumbers = {
95     TollesLawsonTerms.PERMANENT: 3,
96     TollesLawsonTerms.INDUCED: 6,
97     TollesLawsonTerms.INDUCED_REDUCED: 5,
98     TollesLawsonTerms.EDDY: 9,
99     TollesLawsonTerms.EDDY_REDUCED: 8,
100    TollesLawsonTerms.CUBIC: 10,
101    TollesLawsonTerms.CUBIC_REDUCED: 7
102 }
103

```

```

104
105 def get_number_of_tl_coefficient_from_terms(tolles_lawson_terms: typing.Iterable[TollesLawsonTerms]) -> int:
106     """
107         Get the total number of tolles lawson coefficient from a list of terms
108
109         Parameters
110         -----
111         tolles_lawson_terms
112             the list of terms to use for calculating the total
113
114         Returns
115         -----
116         int
117             the total number of coefficients for the input list of terms
118         """
119         num_tl_coefficients = 0
120         for tl_term in tolles_lawson_terms:
121             num_tl_coefficients += TollesLawsonTermsToCoefficientNumbers[tl_term]
122         return num_tl_coefficients
123
124
125     class TollesLawsonError(Exception):
126         """
127             TollesLawsonError is the exception class to be used when an error
128             specific to the tolles-lawson model is generated.
129         """
130
131
132     class TLFilter(Filter):
133         """
134             default filter to use, band-pass cutoff at 0.1 Hz to 0.9 Hz.
135         """
136
137     def __init__(self, fs: float):
138         order = 10
139         fcut = [0.1, 0.9]
140         btype = 'band'
141         ftype = 'butter'
142         super().__init__(order, fcut, btype=btype, ftype=ftype, fs=fs)
143
144
145     def calculate_direction_cosines(
146         vector: np.ndarray,
147         f_total: np.ndarray = None,
148     ) -> np.ndarray:
149         """
150             compute the direction cosines from a vector array
151
152             Parameters
153             -----
154             vector
155                 Nx3 array of vector components. Also work with Nxm...x3
156             f_total
157                 N size array of total magnetic field, it will be synthesized from vector if None

```

```

158     this should normally be set to None for default value
159
160     Returns
161     -----
162     np.ndarray:
163         Nx3 array of direction cosine components. Also can return same shape as vector
164         """
165
166         # if no total field is input, use the vector magnitude
167         if f_total is None:
168             f_total = np.linalg.norm(vector, axis=-1)
169
170         direction_cosines = vector / f_total [..., np.newaxis]
171
172     return direction_cosines
173
174
175     def compute_A(
176         direction_cosines: np.ndarray,
177         diff_direction_cosines: typing.Optional[np.ndarray],
178         B_t: np.ndarray,
179         terms: typing.Sequence[TollesLawsonTerms] = DEFAULT_TL_TERMS
180     ) -> np.ndarray:
181         """
182             compute the Tolles Lawson A for vector compensations.
183
184     Parameters
185     -----
186     direction_cosines
187         Nx3 vector magnetometer direction cosine values ( $\cos(x/B_t)$ ,  $\cos(y/B_t)$ , and  $\cos(z/B_t)$ ) named (cosX, cosY, cosZ
188         )
189     diff_direction_cosines
190         Nx3 vector magnetometer direction cosine gradient values ( $d(\cos(x/B_t))/dt$ ,  $d(\cos(y/B_t))/dt$ , and  $d(\cos(z/B_t))/dt$ ) named
191         (cosX_dot, cosY_dot, cosZ_dot)
192     B_t:
193         N size array of the total measured magnetic intensity for each measurement
194     terms:
195         list of the terms in the Tolles-Lawson model to include, defaults to all three fields
196         for the permanent, induced and eddy current terms mode
197
198     Returns
199     -----
200     np.ndarray
201         A calibration vector with order of first dimension dependent on the terms:
202         PERMANENT
203             0: cosX
204             1: cosY
205             2: cosZ
206
207         INDUCED
208             3: cosX ** 2
209             4: cosX * cosY
210             5: cosX * cosZ

```

```

210      6: cosY ** 2
211      7: cosY * cosZ
212      8: cosZ ** 2
213
214      EDDY
215      9: cosX * cosX_dot
216      10: cosX * cosy_dot
217      11: cosX * cosz_dot
218      12: cosY * cosX_dot
219      13: cosY * cosy_dot
220      14: cosY * cosz_dot
221      15: cosZ * cosX_dot
222      16: cosZ * cosy_dot
223      17: cosZ * cosz_dot
224
225      INDUCED_REDUCED
226      3: cosX ** 2
227      4: cosX * cosy
228      5: cosX * cosZ
229      6: cosY ** 2
230      7: cosY * cosZ
231
232      EDDY_REDUCED
233      9: cosX * cosX_dot
234      10: cosX * cosy_dot
235      11: cosX * cosz_dot
236      12: cosY * cosX_dot
237      13: cosY * cosy_dot
238      14: cosY * cosz_dot
239      15: cosZ * cosX_dot
240      16: cosZ * cosy_dot
241
242      CUBIC
243      18: XXX
244      19: XXY
245      20: XXZ
246      21: YY
247      22: XYZ
248      23: XZZ
249      24: YY
250      25: YYZ
251      26: YZZ
252      27: ZZZ
253
254      CUBIC_REDUCED
255      18: XXX
256      19: XXY
257      20: XXZ
258      21: YY
259      22: XYZ
260      24: YY
261      25: YYZ
262      """
263      # check that at least one output term is requested

```

```

264     if len(terms) == 0:
265         raise TollesLawsonError("At least one term must be included in Tolles-Lawson model")
266
267     # check if requested terms are in list of supported terms
268     for term in terms:
269         if not isinstance(term, TollesLawsonTerms):
270             raise TollesLawsonError(f"Requested model term {term} must be of type {TollesLawsonTerms}")
271     if TollesLawsonTerms.INDUCED in terms and TollesLawsonTerms.INDUCED_REDUCED in terms:
272         raise TollesLawsonError("Cannot have both TollesLawsonTerms.INDUCED and TollesLawsonTerms.INDUCED_REDUCED terms")
273     )
274     if TollesLawsonTerms.EDDY in terms and TollesLawsonTerms.EDDY_REDUCED in terms:
275         raise TollesLawsonError("Cannot have both TollesLawsonTerms.EDDY and TollesLawsonTerms.EDDY_REDUCED terms")
276     if TollesLawsonTerms.CUBIC in terms and TollesLawsonTerms.CUBIC_REDUCED in terms:
277         raise TollesLawsonError("Cannot have both TollesLawsonTerms.CUBIC and TollesLawsonTerms.CUBIC_REDUCED terms")
278
279     # Create A matrix components of zero size to be replaced later if needed
280     Ap: np.ndarray
281     Ae: np.ndarray
282     Ai: np.ndarray
283     Ac: np.ndarray
284     Ap = Ae = Ai = Ac = np.array([], dtype=np.float64).reshape(direction_cosines.shape[0], 0)
285
286     # compute the requested components of the A matrix
287
288     # permanent moments
289     # cosX, cosY, cosZ
290     if TollesLawsonTerms.PERMANENT in terms:
291         Ap = direction_cosines
292
293     # induced moments
294     if TollesLawsonTerms.INDUCED in terms or TollesLawsonTerms.INDUCED_REDUCED in terms:
295         Ai = np.zeros(shape=(direction_cosines.shape[0], 6))
296         # cosX ** 2, cosX * cosY, cosX * cosZ
297         Ai[:, 0:3] = direction_cosines[:, 0:1] * direction_cosines
298         # cosY ** 2, cosY * cosZ
299         Ai[:, 3:5] = direction_cosines[:, 1:2] * direction_cosines[:, 1:]
300         # cosZ ** 2
301         Ai[:, 5:6] = direction_cosines[:, 2:3]**2
302         # multiply the induced and eddy currents by the measured magnetic intensity
303         Ai *= B_t[:, np.newaxis]
304
305         if TollesLawsonTerms.INDUCED_REDUCED in terms:
306             # remove the last coefficient to get a reduced set of induced terms
307             Ai = Ai[:, :-1]
308
309     # eddy current moments
310     if TollesLawsonTerms.EDDY in terms or TollesLawsonTerms.EDDY_REDUCED in terms:
311         if diff_direction_cosines is None:
312             raise TollesLawsonError("direction_cosines must be supplied to use EDDY or EDDY_REDUCED terms")
313         Ae = np.zeros(shape=(direction_cosines.shape[0], 9))
314         # cosX * cosX_dot, cosX * cosY_dot, cosX * cosZ_dot
315         Ae[:, 0:3] = direction_cosines[:, 0:1] * diff_direction_cosines
316         # cosY * cosX_dot, cosY * cosY_dot, cosY * cosZ_dot
317         Ae[:, 3:6] = direction_cosines[:, 1:2] * diff_direction_cosines

```

```

317     # cosZ * cosX_dot, cosZ * cosY_dot, cosZ * cosZ_dot
318     Ae[..., 6:9] = direction_cosines[..., 2:3] * diff_direction_cosines
319     # multiply the induced and eddy currents by the measured magnetic intensity
320     Ae *= B_t[..., np.newaxis]
321     if TollesLawsonTerms.HDDY_REDUCED in terms:
322         Ae = Ae[..., :-1]
323
324     # cubic terms
325     if TollesLawsonTerms.CUBIC in terms or TollesLawsonTerms.CUBC_REDUCED in terms:
326         Ac = np.zeros(shape=(direction_cosines.shape[0], 10))
327         combinations: typing.List[typing.Tuple[int, int,
328                                         int]] = list(itertools.combinations_with_replacement([0, 1, 2], r=3)) # type: ignore
329         for idx, comb in enumerate(combinations):
330             Ac[..., idx] = np.product([direction_cosines[..., ii:ii + 1] for ii in comb])
331
332         if TollesLawsonTerms.CUBC_REDUCED in terms:
333             # remove the redundant coefficient to get a reduced set of induced terms
334             good_indices = list(range(10))
335             remove_indices = [5, 8, 9]
336             for remove_idx in remove_indices:
337                 good_indices.remove(remove_idx)
338             Ac = Ac[..., good_indices]
339
340         # put all the terms together
341         A = np.concatenate([Ap, Ai, Ae, Ac], axis=1)
342
343     return A
344
345
346     def tlc_matrix(
347         vector: np.ndarray,
348         f_total: np.ndarray = None,
349         time_delta: float = 1.0,
350         window_length: int = 3,
351         norm: bool = False,
352         terms: typing.Sequence[TollesLawsonTerms] = DEFAULT_TL_TERMS
353     ) -> np.ndarray:
354         """
355             computes the matrix of values required to solve for the
356             Tolles-Lawson coefficients
357
358         Parameters
359         -----
360         vector
361             is a Nx3 matrix of the vector magnetometer data
362         f_total
363             N size array of total magnetic field, it will be synthesized from vector if None
364             this should normally be set to None for default value
365         time_delta
366             is the time between samples defaults to 1.0
367             this is only used for computing the numerical derivative for the eddy current
368             terms. Sometimes this is ignore (set to 1.0), but setting it explicitly
369             allows for comparison with data sampled at different rates.

```

```

370     window_length
371         is the window length to use for the differentiation filter in samples
372         defaults to 3
373     norm
374         if True will normalized the mean total field to have an average value of 1
375     terms
376         list of Tolles-Lawson terms to include in the model, available options are
377         permanent, induced, and/or eddy, at least one must be included.
378         defaults to all three terms.
379
380     Returns
381     -----
382     np.ndarray
383         Tolles-Lawson A-matrix, Nx18 ndarray
384
385     """
386
387     # if no total field is input, use the vector magnitude
388     if f_total is None:
389         f_total = np.linalg.norm(vector, axis=-1)
390
391     # find the mean total field and normalize the induced and eddy current
392     # moments to the mean value if requested
393     mean_total_field: typing.Union[float, np.ndarray]
394     if norm:
395         mean_total_field = typing.cast(np.ndarray, np.mean(f_total))
396     else:
397         mean_total_field = 1.0
398     B_t: np.ndarray = f_total / mean_total_field
399
400     # compute direction cosines
401     direction_cosines = calculate_direction_cosines(vector, f_total=f_total)
402
403     if TollesLawsonTerms.EDDY in terms or TollesLawsonTerms.EDDY_REDUCED in terms:
404         # compute the time derivatives
405         # This is a differentiating filter that is zero lag, using a
406         # Savitzgy-Golay filter with padding on each end.
407         diff_direction_cosines = scipy.signal.savgol_filter(
408             direction_cosines, window_length=window_length, polyorder=2, deriv=1, axis=0, delta=time_delta
409         )
410     else:
411         diff_direction_cosines = None
412
413     # calculate the A matrix using the direction cosines and the diffs
414     A = compute_A(direction_cosines=direction_cosines, diff_direction_cosines=diff_direction_cosines, B_t=B_t, terms=
415                 terms)
416
417     return A
418
419 def tolles_lawson_coefficients(
420     vector: np.ndarray,
421     y_value: np.ndarray,
422     f_total: typing.Optional[np.ndarray] = None,

```

```

423     time_delta: float = 1.0,
424     window_length: int = 3,
425     apply_filter: bool = True,
426     mag_filter: typing.Optional[Filter] = None,
427     cut_length: typing.Optional[int] = None,
428     ridge_parameters: typing.Optional[typing.Dict[str, typing.Any]] = None,
429     terms: typing.Sequence[TollesLawsonTerms] = DEFAULT_TL_TERMS
430 ) -> np.ndarray:
431     """
432         compute the Tolles Lawson coefficients using the input vector data and target y_value data
433
434     Parameters
435     -----
436     vector
437         Nx3 array of vector data describing the direction of the external field.
438         typically the values of a vector magnetometer
439     y_value
440         N size array of the total-field magnetometer that we wish
441         to compensate with the resulting coefficients.
442         This value can have the geomagnetic elements pre-removed to improve accuracy
443     f_total
444         N size array of the total field as recorded by a scalar magnetometer
445         determined to be more accurate compared to using the lengths of vector
446         defaults to None which will use the vector values
447     time_delta
448         the sample spacing between each sample in seconds
449         required for eddy current coefficient generation
450     window_length
451         the window length in samples to use for the differentiation filter
452     apply_filter
453         if True (default) will apply the mag_filter. if False then no filter will be used
454     mag_filter
455         an optional filter to apply to the A matrix and y_value before computing
456         the TLC, should be of type Filter from compensation.filter
457         Ignored if apply_filter is False
458     cut_length
459         is the length of samples to cut from each end to avoid filter artifacts
460         at the edges, will be set to 2 seconds on each end (empirical) + window_length if None
461         defaults to None
462     ridge_parameters
463         The parameters given to the ridge regression
464         if None is given defaults to the built in parameters with the following exceptions:
465         alpha=0, do not do a ridge regression and just do a least squares fit
466         fit_intercept=False, assume the user can add an intercept if desired
467         for more info see: https://scikit-learn.org/stable/modules/generated/sklearn.linear\_model.Ridge.html
468     terms
469         list of Tolles-Lawson terms to include in the model, available options are
470         permanent, induced, and/or eddy, at least one must be included.
471         defaults to all three terms.
472
473     Returns
474     -----
475     np.ndarray
476         array of size 18 which is the computed tolles lawson coefficients

```

```

477     """
478     augmented_values: np.ndarray = np.empty(shape=(vector.shape[0], 0))
479     return augmented_tolles_lawson_coefficients(
480         vector=vector,
481         augmented_values=augmented_values,
482         y_value=y_value,
483         f_total=f_total,
484         time_delta=time_delta,
485         window_length=window_length,
486         apply_filter=apply_filter,
487         apply_filter_augmented_values=False,
488         mag_filter=mag_filter,
489         cut_length=cut_length,
490         ridge_parameters=ridge_parameters,
491         terms=terms
492     )
493
494
495 def augmented_tolles_lawson_fitting_matrices(
496     vector: np.ndarray,
497     augmented_values: np.ndarray,
498     y_value: np.ndarray,
499     f_total: typing.Optional[np.ndarray] = None,
500     time_delta: float = 1.0,
501     window_length: int = 3,
502     apply_filter: bool = True,
503     apply_filter_augmented_values: bool = True,
504     mag_filter: typing.Optional[Filter] = None,
505     cut_length: typing.Optional[int] = None,
506     terms: typing.Sequence[TollesLawsonTerms] = DEFAULT_TL_TERMS
507 ) -> typing.Tuple[np.ndarray, np.ndarray]:
508     """
509     Compute the augmented Tolles Lawson fitting matrices ( $A$ ,  $y$ )
510     using the input vector data augmented data and target  $y\_value$  data
511     Does preprocessing on the data such as direction cosines calculation, clipping, filtering and rearranging
512
513     Parameters
514     -----
515     vector
516         Nx3 array of vector data describing the direction of the external field.
517         typically the values of a vector magnetometer
518     augmented_values
519         NxD array of augmented values to concat to the  $A$  matrix so the final fit uses
520         augmented data to predict the  $y\_values$ 
521     y_value
522         N size array of the total-field magnetometer that we wish
523         to compensate with the resulting coefficients.
524         This value can have the geomagnetic elements pre-removed to improve accuracy
525     f_total
526         N size array of the total field as recorded by a scalar magnetometer
527         determined to be more accurate compared to using the lengths of vector
528         defaults to None which will use the vector values
529     time_delta
530         the sample spacing between each sample in seconds

```

```

531     required for eddy current coefficient generation
532     window_length
533         the window length in samples to use for the differentiation filter
534     apply_filter
535         if True (default) will apply the mag_filter to the A matrix and y_values
536     apply_filter_augmented_values
537         if True (default) will apply the mag_filter to the augmented values
538     mag_filter
539         an optional filter to apply to the A matrix and y_value before computing
540         the TLC; should be of type Filter from compensation.filter
541         Ignored if apply_filter is False
542     cut_length
543         is the length of samples to cut from each end to avoid filter artifacts
544         at the edges, will be set to 2 seconds on each end (empirical) + window_length if None
545         defaults to None
546     terms
547         list of Tolles-Lawson terms to include in the model, available options are
548         permanent, induced, and/or eddy, at least one must be included.
549         defaults to all three terms.
550
551     Returns
552     -----
553     typing.Tuple[np.ndarray, np.ndarray]
554         the two matrices for fitting via some optimizer, the tuple is (A, y) where A is the A matrix of the tolles
555         lawson algorithm
556         with the direction cosines (and any augmented data) and y is the target values to fit to.
557         This is meant to be used in solving for the matrix x in y=Ax
558     """
559     # if the filter is None get our default filter
560     if mag_filter is None:
561         mag_filter = TLFFilter(fs=1.0 / time_delta)
562
563     # compute the components of the A matrix
564     A = tlc_matrix(vector=vector, f_total=f_total, time_delta=time_delta, window_length=window_length, terms=terms)
565
566     if apply_filter:
567         # apply the filter to the A matrix and y_values
568         A = mag_filter.filtfilt(A, axis=0)
569         y_value = mag_filter.filtfilt(y_value, axis=0)
570
571     if apply_filter_augmented_values:
572         # apply the filter to our augmented data
573         augmented_values = mag_filter.filtfilt(augmented_values, axis=0)
574
575     # concat the A matrix and augmented data
576     A_augmented = np.hstack([A, augmented_values])
577
578     # if cut_length is None use the default
579     if cut_length is None:
580         cut_length = int(2.0 / time_delta) + window_length
581
582     # if cut_length is above 0 then apply the cut
583     if cut_length > 0:
584         A_augmented = A_augmented[cut_length:-cut_length] # pylint: disable=invalid-unary-operand-type

```

```

584         y_value = y_value[cut_length:-cut_length] # pylint: disable=invalid-unary-operand-type
585
586     return A_augmented, y_value
587
588
589     def augmented_tolles_lawson_coefficients(
590         vector: np.ndarray,
591         augmented_values: np.ndarray,
592         y_value: np.ndarray,
593         f_total: typing.Optional[np.ndarray] = None,
594         time_delta: float = 1.0,
595         window_length: int = 3,
596         apply_filter: bool = True,
597         apply_filter_augmented_values: bool = True,
598         mag_filter: typing.Optional[Filter] = None,
599         cut_length: typing.Optional[int] = None,
600         ridge_parameters: typing.Optional[typing.Dict[str, typing.Any]] = None,
601         terms: typing.Sequence[TollesLawsonTerms] = DEFAULT_TL_TERMS
602     ) -> np.ndarray:
603         """
604             compute the augmented Tolles Lawson coefficients using the input vector data augmented data
605             and target y_value data
606
607             Parameters
608             -----
609             vector
610                 Nx3 array of vector data describing the direction of the external field.
611                 typically the values of a vector magnetometer
612             augmented_values
613                 NzM array of augmented values to concat to the A matrix so the final fit uses
614                 augmented data to predict the y_values
615             y_value
616                 N size array of the total-field magnetometer that we wish
617                 to compensate with the resulting coefficients.
618                 This value can have the geomagnetic elements pre-removed to improve accuracy
619             f_total
620                 N size array of the total field as recorded by a scalar magnetometer
621                 determined to be more accurate compared to using the lengths of vector
622                 defaults to None which will use the vector values
623             time_delta
624                 the sample spacing between each sample in seconds
625                 required for eddy current coefficient generation
626             window_length
627                 the window length in samples to use for the differentiation filter
628             apply_filter
629                 if True (default) will apply the mag_filter to the A matrix and y_values
630             apply_filter_augmented_values
631                 if True (default) will apply the mag_filter to the augmented values
632             mag_filter
633                 an optional filter to apply to the A matrix and y_value before computing
634                 the TLC, should be of type Filter from compensation.filter
635                 Ignored if apply_filter is False
636             cut_length
637                 is the length of samples to cut from each end to avoid filter artifacts

```

```

638     at the edges, will be set to 2 seconds on each end (empirical) + window_length if None
639     defaults to None
640
641     ridge_parameters
642         The parameters given to the ridge regression
643         if None is given defaults to the built in parameters with the following exceptions:
644         alpha=0, do not do a ridge regression and just do a least squares fit
645         fit_intercept=False, assume the user can add an intercept if desired
646         for more info see: https://scikit-learn.org/stable/modules/generated/sklearn.linear\_model.Ridge.html
647
648     terms
649         list of Tolles-Lawson terms to include in the model, available options are
650         permanent, induced, and/or eddy, at least one must be included.
651         defaults to all three terms.
652
653     Returns
654     -----
655     np.ndarray
656         array of size 18+M which is the computed augmented tolles lawson coefficients
657         """
658
659     A_augmented, y_value = augmented_tolles_lawson_fitting_matrices(
660         vector=vector,
661         augmented_values=augmented_values,
662         y_value=y_value,
663         f_total=f_total,
664         time_delta=time_delta,
665         window_length=window_length,
666         apply_filter=apply_filter,
667         apply_filter_augmented_values=apply_filter_augmented_values,
668         mag_filter=mag_filter,
669         cut_length=cut_length,
670         terms=terms
671     )
672
673     # make our ridge regression model
674     if ridge_parameters is None:
675         ridge_parameters = {
676             'alpha': 0,
677             'fit_intercept': False,
678         }
679     ridge = Ridge(**ridge_parameters)
680
681     # fit our ridge regression
682     ridge.fit(A_augmented, y_value)
683
684     # return the coefficients which are the tolles lawson coefficients
685     return ridge.coef_
686
687     def tlc_compensation(
688         vector: np.ndarray,
689         tlc: np.ndarray,
690         f_total: typing.Optional[np.ndarray] = None,
691         time_delta: float = 1.0,
692         window_length: int = 3,

```

```

692     terms: typing.Sequence[TollesLawsonTerms] = DEFAULT_TL_TERMS
693 ) -> np.ndarray:
694     """
695     determine the Tolles–Lawson disturbance field to remove from the total field magnetometer
696
697     Parameters
698     -----
699     vector
700         Nx3 array of vector data describing the direction of the external field.
701         typically the values of a vector magnetometer
702     tlc
703         18 size array of the tolles lawson coefficients that correspond to the columns of the
704         A matrix calculated from the tlc_matrix function
705     f_total
706         N sized array of the total field as recorded by a scalar magnetometer
707         determined to be more accurate compared to using the lengths of vector
708         defaults to None which will use the vector values
709     time_delta
710         the sample spacing between each sample in seconds
711         required for eddy current coefficient generation
712     window_length
713         the window length in samples to use for the differentiation filter
714     terms
715         list of Tolles–Lawson terms to include in the model, available options are
716         permanent, induced, and/or eddy, at least one must be included.
717         defaults to all three terms.
718
719     Returns
720     -----
721     np.ndarray
722         returns the T-L field to be removed
723         """
724     if f_total is None:
725         f_total = np.linalg.norm(vector, axis=-1)
726
727     # calculate our A matrix
728     A = tlc_matrix(vector=vector, f_total=f_total, time_delta=time_delta, window_length=window_length, terms=terms)
729
730     # do the dot product
731     return A @ tlc
732
733
734     def augmented_tlc_compensation_matrix(
735         vector: np.ndarray,
736         augmented_values: np.ndarray,
737         f_total: typing.Optional[np.ndarray] = None,
738         time_delta: float = 1.0,
739         window_length: int = 3,
740         terms: typing.Sequence[TollesLawsonTerms] = DEFAULT_TL_TERMS
741     ) -> np.ndarray:
742         """
743         determine the Tolles–Lawson A matrix to use for removing the aircraft field
744
745         Parameters

```

```

746
747     -----
748     vector
749         Nx3 array of vector data describing the direction of the external field.
750         typically the values of a vector magnetometer
750     augmented_values
751         NxM array of augmented values to concat to the A matrix so the final fit uses
752         augmented data to predict the y_values
753     f_total
754         N sized array of the total field as recorded by a scalar magnetometer
755         determined to be more accurate compared to using the lengths of vector
756         defaults to None which will use the vector values
757     time_delta
758         the sample spacing between each sample in seconds
759         required for eddy current coefficient generation
760     window_length
761         the window length in samples to use for the differentiation filter
762     terms
763         list of Tolles-Lawson terms to include in the model, available options are
764         permanent, induced, and/or eddy, at least one must be included.
765         defaults to all three terms.
766
767     Returns
768     -----
769     np.ndarray
770         returns the T-L field to be removed
771         """
772     if f_total is None:
773         f_total = np.linalg.norm(vector, axis=-1)
774
775     # calculate our A matrix
776     A = tlc_matrix(vector=vector, f_total=f_total, time_delta=time_delta, window_length=window_length, terms=terms)
777
778     # concat the A matrix and augmented data
779     A_augmented = np.hstack([A, augmented_values])
780
781     return A_augmented
782
783
784     def augmented_tlc_compensation(
785         vector: np.ndarray,
786         augmented_values: np.ndarray,
787         tlc: np.ndarray,
788         f_total: typing.Optional[np.ndarray] = None,
789         time_delta: float = 1.0,
790         window_length: int = 3,
791         terms: typing.Sequence[TollesLawsonTerms] = DEFAULT_TL_TERMS
792     ) -> np.ndarray:
793         """
794             determine the Tolles-Lawson disturbance field to remove from the total field magnetometer
795
796     Parameters
797     -----
798     vector
799         Nx3 array of vector data describing the direction of the external field.

```

```

800      typically the values of a vector magnetometer
801      augmented_values
802          Nxm array of augmented values to concat to the A matrix so the final fit uses
803          augmented data to predict the y_values
804      tlc
805          18 size array of the tolles lawson coefficients that correspond to the columns of the
806          A matrix calculated from the tlc_matrix function
807      f_total
808          N sized array of the total field as recorded by a scalar magnetometer
809          determined to be more accurate compared to using the lengths of vector
810          defaults to None which will use the vector values
811      time_delta
812          the sample spacing between each sample in seconds
813          required for eddy current coefficient generation
814      window_length
815          the window length in samples to use for the differentiation filter
816      terms
817          list of Tolles-Lawson terms to include in the model, available options are
818          permanent, induced, and/or eddy, at least one must be included.
819          defaults to all three terms.
820
821      Returns
822      -----
823      np.ndarray
824          returns the T-L field to be removed
825      """
826
827      # get the augmented matrix
828      A_augmented = augmented_tlc_compensation_matrix(
829          vector=vector, augmented_values=augmented_values, f_total=f_total, time_delta=time_delta, window_length=
830          window_length, terms=terms
831      )
832
833      # do the dot product
834      return A_augmented @ tlc

```

## mammal/MAMMAL/VehicleCal/TL.py

```

1 import sys
2 from os.path import dirname, join, realpath
3
4 import numpy as np
5 import scipy.linalg as la
6 import scipy.signal as sps
7 import scipy.interpolate as interp
8 from sklearn.linear_model import Ridge
9
10 sys.path.append(dirname(realpath(__file__)))
11 sys.path.append(dirname(dirname(realpath(__file__))))
12
13 from Utils import Filters
14
15

```

```

16  # Enumerate A matrix terms
17  ALL_TERMS = 0
18  PERMANENT = 1
19  INDUCED = 2
20  EDDY = 3
21
22
23  def direction_cosines(vec: np.ndarray) -> np.ndarray:
24      ...
25      Compute direction cosines from vector magnetometer values
26
27      Parameters
28      -----
29      vec
30          Nx3 array of vector measurements (nT)
31
32      Returns
33      -----
34      np.ndarray
35          Nx3 array of direction cosines
36      ...
37
38      magnitude = la.norm(vec, axis=1)[:, np.newaxis] # Use np.newaxis to keep magnitude a col vector
39
40      return vec / magnitude
41
42  def A_matrix(dir_cos: np.ndarray,
43              b_external: np.ndarray,
44              delta_t: float=1.0,
45              terms: int=ALL_TERMS) -> np.ndarray:
46      ...
47      Compute the A_matrix from direction cosines. Filtering
48      the columns with the input filter.
49
50      Parameters
51      -----
52      direction_cosines
53          Nx3 numpy array of direciton cosines
54      b_external
55          External mag field magnitude (nT) to use for a-matrix, if none,
56          try to compute from average of b_scalar
57      terms
58          Terms to include in A-matrix. Options include:
59
60          - ALL_TERMS
61          - PERMANENT
62          - INDUCED
63          - EDDY
64
65      Returns
66      -----
67      A
68      ...
69

```

```

70
71     if terms == ALL_TERMS:
72         terms = [PERMANENT, INDUCED, EDDY]
73
74     A = np.array([]).reshape([len(dir_cos), 0])
75
76     # Direction cosines
77     cx = dir_cos[:, [0]]
78     cy = dir_cos[:, [1]]
79     cz = dir_cos[:, [2]]
80
81     # Compute the gradient of the direction cosines in the time direction
82     t = np.linspace(0, int((A.shape[0] + delta_t) * delta_t), A.shape[0]) # Relative timestamps
83
84     cx_prime = interp.splev(t, interp.splrep(t, cx), der=1)[:, np.newaxis]
85     cy_prime = interp.splev(t, interp.splrep(t, cy), der=1)[:, np.newaxis]
86     cz_prime = interp.splev(t, interp.splrep(t, cz), der=1)[:, np.newaxis]
87
88     if PERMANENT in terms:
89         A = np.hstack([A, cx, cy, cz])
90
91     if INDUCED in terms:
92         A = np.hstack([A,
93                         b_external * cx**2,
94                         b_external * cy**2,
95                         b_external * cz**2,
96                         b_external * cx * cy,
97                         b_external * cx * cz,
98                         b_external * cy * cz])
99
100    if EDDY in terms:
101        A = np.hstack([A,
102                      b_external * cx * cx_prime,
103                      b_external * cy * cy_prime,
104                      b_external * cz * cz_prime,
105                      b_external * cx_prime * cy,
106                      b_external * cx_prime * cz,
107                      b_external * cy_prime * cx,
108                      b_external * cy_prime * cz,
109                      b_external * cz_prime * cx,
110                      b_external * cz_prime * cy])
111
112    return A
113
114 def tlc(b_scalar: np.ndarray,
115         b_vector: np.ndarray,
116         delta_t: float,
117         use_filter: bool=False,
118         fstart: float=0.1,
119         fstop: float=1.0,
120         order: int=5,
121         b_external: float=None,
122         alpha: float=0,
123         fit_intcpt: bool=False,

```

```

124     terms:      int=ALL_TERMS):
125     '',
126     Solve for Tolles-Lawson coefficients using all of the input data
127
128     Parameters
129     -----
130     b_scalar
131         Nx1 array of scalar measurements (nT)
132     b_vector
133         Nx3 array of vector measurements (nT)
134     delta_t
135         Time between data samples (s)
136     use_filter
137         Use band pass filter if true
138     fstart
139         Start frequency for the band (Hz)
140     fstop
141         Stop frequency for the band (Hz)
142     order
143         Order of the filter
144     b_external
145         External mag field magnitude (nT) to use for a-matrix, if none,
146         try to compute from average of b_scalar
147     alpha
148         Ridge regression alpha tuning parameter - set to
149         0 to disable ridge regression
150     fit_intcpt
151         Ridge regression fit intercept flag
152     terms
153         Terms to include in A-matrix. Options include:
154
155         - ALL_TERMS
156         - PERMANENT
157         - INDUCED
158         - EDDY
159
160     Returns
161     -----
162     np.ndarray
163         1xK T-L calibration coefficients where K is the
164         number of A matrix terms to use for calibration
165         ,
166
167     if terms == ALL_TERMS:
168         terms = [PERMANENT, INDUCED, EDDY]
169
170     if b_external is None:
171         b_external = np.mean(b_scalar)
172
173     dc = direction_cosines(b_vector)
174     A = A_matrix(dc,
175                  b_external,
176                  delta_t,
177                  terms)

```

```

178
179     if use_filter:
180         fs = 1.0 / delta_t
181
182     if (PERMANENT in terms) and (len(terms) > 1):
183         A[:, 3:] = Filters.bpf(data = A[:, 3:],
184                                fstart = fstart,
185                                fstop = fstop,
186                                fs = fs,
187                                order = order,
188                                axis = 0)
189
190     elif PERMANENT not in terms:
191         A = Filters.bpf(data = A,
192                           fstart = fstart,
193                           fstop = fstop,
194                           fs = fs,
195                           order = order,
196                           axis = 0)
197
198     else:
199         A = Filters.lpf(data = A,
200                           cutoff = fstart,
201                           fs = fs,
202                           axis = 0)
203
204     b_scalar = Filters.bpf(b_scalar, fstart, fstop, fs, order)
205
206     if alpha == 0:
207         return np.linalg.lstsq(A, b_scalar, rcond=None)[0]
208
209     ridge = Ridge(alpha=alpha, fit_intercept=fit_intcpt)
210     ridge.fit(A, b_scalar)
211
212     return ridge.coef_
213
214     def apply_tlc(c: np.ndarray,
215                  b_scalar: np.ndarray,
216                  b_vector: np.ndarray,
217                  delta_t: float,
218                  b_external: float=None,
219                  terms: int=ALL_TERMS):
220         ...
221
222         Apply T-L calibration coefficients to data
223
224         -----
225         c
226
227         1xK T-L calibration coefficients where K is the
228         number of A matrix terms to use for calibration
229         b_scalar
230
231         Nx1 array of scalar measurements (nT)
232         b_vector
233
234         Nx3 array of vector measurements (nT)

```

```

232     delta_t
233         Time between data samples (s)
234     b_external
235         External mag field magnitude (nT) to use for a-matrix, if none,
236         try to compute from average of b_scalar
237     terms
238         Terms to include in A-matrix. Options include:
239
240         - ALL_TERMS
241         - PERMANENT
242         - INDUCED
243         - EDDY
244
245     Returns
246     -----
247     np.ndarray
248         Nx1 array of calibrated scalar measurements (nT)
249     ,
250
251     if b_external is None:
252         b_external = np.mean(b_scalar)
253
254     dc = direction_cosines(b_vector)
255     A = A_matrix(dc,
256                 b_external,
257                 delta_t,
258                 terms)
259
260     return A @ c
261
262 def apply_tl_dist(c:          np.ndarray,
263                    b_scalar:   np.ndarray,
264                    b_vector:   np.ndarray,
265                    delta_t:    float,
266                    b_external: float=None,
267                    terms:      int=ALL_TERMS) -> np.ndarray:
268     ,
269
270     Apply distortion to the scalar magnetometer data given a set of Tolles-Lawson
271     coefficients
272
273     Parameters
274     -----
275     c
276         Tolles-Lawson coefficients used to distort the scalar magnetometer data. The number
277         of coefficients must correspond to the terms specified by 'terms'
278     b_scalar
279         Nx1 array of scalar measurements (nT)
280     b_vector
281         Nx3 array of vector measurements (nT)
282     delta_t
283         Time between data samples (s)
284     b_external
285         External mag field magnitude (nT) to use for a-matrix, if none,
286         try to compute from average of b_scalar

```

```

286      terms
287          Specify which terms to use to distort the data. Options include:
288
289          - ALL_TERMS
290          - PERMANENT
291          - INDUCED
292          - EDDY
293
294      Returns
295  -----
296      np.ndarray
297          Nx1 array of distorted scalar measurements (nT)
298      ,
299
300      return b_scalar + apply_tlc(c,
301                                  b_scalar,
302                                  b_vector,
303                                  delta_t,
304                                  b_external,
305                                  terms)
306
307
308  if __name__ == '__main__':
309      import sys
310      from os.path import dirname, join
311
312      import matplotlib.pyplot as plt
313
314      SRC_DIR = dirname(dirname(__file__))
315      sys.path.append(SRC_DIR)
316
317      from Parse import parseSGL as psgl
318
319
320      BASE_DIR = dirname(SRC_DIR)
321      DATA_DIR = join(BASE_DIR, 'data')
322      TEST_DIR = join(DATA_DIR, 'test')
323
324      FNAME = join(TEST_DIR, '10Hz_Mag_INS_Aux_Flt1002.xyz')
325
326
327      df = psgl.parse_xyz(FNAME,
328                           drop_dups=False,
329                           drop_cust=['DRAPE', 'OGS_MAG', 'OGS_HGT'])
330      print(df)
331
332      alt     = np.array(df.BARO)
333      t       = np.array(df.epoch_sec)
334      vec_x   = np.array(df.FLUXB_X)
335      vec_y   = np.array(df.FLUXB_Y)
336      vec_z   = np.array(df.FLUXB_Z)
337
338      b_vector = np.vstack((vec_x, vec_y, vec_z)).T
339      b_mag1   = np.array(df.UNCOMPMAG1)

```

```

340     b_mag2 = np.array(df.UNCOMPMAG2)
341     b_mag3 = np.array(df.UNCOMPMAG3)
342     b_mag4 = np.array(df.UNCOMPMAG4)
343     b_mag5 = np.array(df.UNCOMPMAG5)
344
345     truth = np.array(df.COMPMAG1)
346     delta_t = 0.1
347
348     cal_mask = np.logical_and(alt > 2925, t < 1.59266e9) # Calibration alt > 2925m and Unix epoch timestamp < 1.59266e9s
349
350     t_cal = t[cal_mask]
351     b_mag_cal = b_mag3[cal_mask]
352     b_vec_cal = b_vector[cal_mask, :]
353     truth_cal = df.COMPMAG1[cal_mask]
354
355     c = tlc(b_scalar = b_mag_cal,
356               b_vector = b_vec_cal,
357               delta_t = delta_t,
358               use_filter = True,
359               fstart = 0.1,
360               fstop = 1.0,
361               order = 5,
362               b_external = None,
363               alpha = 0,
364               fit_intcpt = False,
365               terms = ALL_TERMS)
366
367     cal_data = apply_tlc(c = c,
368                           b_scalar = b_mag_cal,
369                           b_vector = b_vec_cal,
370                           delta_t = delta_t,
371                           terms = ALL_TERMS)
372
373     plt.figure()
374     plt.title('T-L Example')
375     plt.plot(t_cal, b_mag_cal, label='Original')
376     plt.plot(t_cal, b_mag_cal-cal_data, label='Calibrated')
377     plt.plot(t_cal, truth_cal, label='Truth')
378     plt.grid()
379     plt.legend()
380     plt.show()

```

## mammal/MAMMAL/SensorCal/\_\_init\_\_.py

```

1 import os, sys
2
3 sys.path.append(os.path.dirname(os.path.realpath(__file__)))
4 sys.path.append(os.path.dirname(os.path.dirname(os.path.realpath(__file__))))

```

## mammal/MAMMAL/SensorCal/spinCal.py

```

1 import datetime as dt

```

```

2
3 import matplotlib.pyplot as plt
4 import numpy as np
5 import scipy.linalg as la
6 from scipy.optimize import minimize
7 from scipy.spatial.transform import Rotation as R
8 from ppigrf import igrf
9
10
11 def b_earth_ned_igrf(lats: np.ndarray,
12                      lons: np.ndarray,
13                      heights: np.ndarray,
14                      dates: list) -> np.ndarray:
15
16     """Find the IGRF Earth field in the NED frame
17
18     Parameters
19     -----
20
21     lats
22         Latitude (dd) or 1xN array of latitudes (dd)
23
24     lons
25         Longitude (dd) or 1xN array of longitudes (dd)
26
27     heights
28         Height MSL (m) or 1xN array of heights MSL (m)
29
30     dates
31         DateTime (UTC) object or 1xN List of DateTime (UTC) objects
32
33     Returns
34     -----
35
36     np.ndarray
37         Nx3 array of IGRF Earth field (nT) -> [Bn, Be, Ba] <- 1st time sample
38
39     [., ., .]
40
41     [., ., .]
42
43     [., ., .] <- Nth time sample
44
45     """
46
47     Be, Bn, Bu = igrf(lons, lats, heights / 1000, dates)
48
49     if type(lats) is not np.ndarray:
50         return np.hstack((Bn.squeeze(), Be.squeeze(), -Bu.squeeze()))
51     elif (len(lats) > 1) or (len(lons) > 1):
52         return np.hstack((Bn.squeeze()[:, np.newaxis], Be.squeeze()[:, np.newaxis], -Bu.squeeze()[:, np.newaxis]))
53     else:
54         return np.hstack((Bn.squeeze(), Be.squeeze(), -Bu.squeeze()))
55
56
57     def gen_b_truth_dcm(lat: float,
58                         lon: float,
59                         height: float,
60                         date: dt.datetime,
61                         dcms: np.ndarray) -> np.ndarray:
62
63
64     """Rotate the IGRF Earth field into the sensor frame across
65     entire calibration using a tensor of direction cosine
66     matrices

```

```

56
57     Parameters
58     -----
59     lat
60         Latitude (dd)
61     lon
62         Longitude (dd)
63     height
64         Height MSL (m)
65     date
66         datetime.datetime (UTC) object
67     dcms
68         Nx3x3 Tensor array of direction cosine matrices
69         where N is the number of samples taken in the
70         calibration
71
72     Returns
73     -----
74     np.ndarray
75         Nx3 array of IGRF Earth field (nT) rotated
76         into sensor frame -> [Bx, By, Bz] <- 1st time sample
77             [..., ., .]
78             [..., ., .]
79             [..., ., .] <- Nth time sample
80         where N is the number of samples taken in the
81         calibration
82         ...
83
84     b_earth = b_earth_ned_igrf(lat, lon, height, date)
85
86     return dcms @ b_earth
87
88     def gen_b_truth_euler(lat:      float,
89                           lon:      float,
90                           height:   float,
91                           date:     dt.datetime,
92                           eulers:   np.ndarray,
93                           rot_seq:  str='zyx',
94                           degrees:  bool=True) -> np.ndarray:
95         ...
96
97         Rotate the IGRF Earth field into the sensor frame across
98         entire calibration using an array of euler angles
99
100    Parameters
101    -----
102    lat
103        Latitude (dd)
104        Longitude (dd)
105    height
106        Height MSL (m)
107    date
108        datetime.datetime (UTC) object
109    eulers

```

```

110      Nx3 array of sensor pitch, roll, yaw euler angles
111      (rotates sensor frame —not measurement vector— from NED to body)
112      where N is the number of samples taken in the
113      calibration -> [roll, pitch, yaw] <- 1st time sample
114          [. . .]
115          [. . .]
116          [. . .] <- Nth time sample
117
118      rot_seq
119      Rotation sequence for the euler angles
120      degrees
121      Whether the euler angles are in degrees or not
122
123      Returns
124      -----
125      np.ndarray
126      Nx3 array of IGRF Earth field (nT) rotated
127      into sensor frame -> [Bx, By, Bz] <- 1st time sample
128          [. . .]
129          [. . .]
130          [. . .] <- Nth time sample
131      where N is the number of samples taken in the
132      calibration
133      ,
134      flpd_angs = np.flip(-eulers, axis=1) # Negate angles because we want to rotate NED vectors into body frame and flip
135      angle order cause scipy rotations are stupid
136      dcms     = R.from_euler(rot_seq, flpd_angs, degrees=degrees).as_matrix()
137
138      return gen_b_truth_dcm(lat, lon, height, date, dcms)
139
140  def ab_to_x(a: np.ndarray,
141              b: np.ndarray) -> np.ndarray:
142
143      Combine distortion matrix and bias vector into a single calibration vector
144
145      Parameters
146      -----
147      a
148          3x3 distortion matrix
149      b
150          1x3 bias vector
151
152      Returns
153      -----
154      np.ndarray
155      1x12 array of elements where the first 9 are from 'a' and the last 3 are from 'b'
156      ,
157      return np.concatenate([a.flatten(), b.flatten()])
158
159  def x_to_ab(x: np.ndarray) -> np.ndarray:
160
161      Split calibration vector into distortion matrix and bias vector
162

```

```

163     Parameters
164     -----
165     x
166         1x12 array of elements where the first 9 are from 'a' and the last 3 are from 'b'
167
168     Returns
169     -----
170     a
171         3x3 distortion matrix
172     b
173         1x3 bias vector
174     ...
175
176     a = x[0:9].reshape(3,3)
177     b = x[9:12]
178
179     return a, b
180
181 def apply_dist_to_vec(vec: np.ndarray,
182                     a: np.ndarray,
183                     b: np.ndarray) -> np.ndarray:
184     ...
185     Apply distortion to vector data
186
187     Parameters
188     -----
189     vec
190         Nx3 array of vector magnetometer measurements
191         where N is the number of samples taken in the
192         calibration
193     a
194         3x3 distortion matrix
195     b
196         1x3 bias vector
197
198     Returns
199     -----
200     np.ndarray
201         Nx3 array of distorted vector measurements (nT)
202         where N is the number of samples taken in the
203         calibration
204     ...
205
206     return (a @ (vec + b).T).T
207
208 def apply_distx_to_vec(vec: np.ndarray,
209                     x: np.ndarray) -> np.ndarray:
210     ...
211     Apply distortion vector to vector data
212
213     Parameters
214     -----
215     vec
216         Nx3 array of vector magnetometer measurements

```

```

217     where  $N$  is the number of samples taken in the
218     calibration
219      $x$ 
220      $1 \times 12$  calibration vector
221
222     Returns
223     -----
224      $np.ndarray$ 
225      $N \times 3$  array of distorted vector measurements ( $nT$ )
226     where  $N$  is the number of samples taken in the
227     calibration
228     ,
229
230     return apply_dist_to_vec(vec, *x_to_ab(x))
231
232 def apply_cal_to_vec(vec: np.ndarray,
233                      a: np.ndarray,
234                      b: np.ndarray) -> np.ndarray:
235
236     Apply distortion and bias calibration to vector data
237
238     Parameters
239     -----
240     vec
241          $N \times 3$  array of vector magnetometer measurements
242         where  $N$  is the number of samples taken in the
243         calibration
244     a
245          $3 \times 3$  distortion matrix
246     b
247          $1 \times 3$  bias vector
248
249     Returns
250     -----
251      $np.ndarray$ 
252          $N \times 3$  array of calibrated vector measurements ( $nT$ )
253         where  $N$  is the number of samples taken in the
254         calibration
255     ,
256
257     return la.solve(a, vec.T).T - b
258
259 def apply_calx_to_vec(vec: np.ndarray,
260                      x: np.ndarray) -> np.ndarray:
261
262     Apply calibration vector to vector data
263
264     Parameters
265     -----
266     vec
267          $N \times 3$  array of vector magnetometer measurements
268         where  $N$  is the number of samples taken in the
269         calibration
270     x

```

```

271      1x12 calibration vector
272
273      Returns
274      -----
275      np.ndarray
276          Nx3 array of calibrated vector measurements (nT)
277          where N is the number of samples taken in the
278          calibration
279          ...
280
281      return apply_cal_to_vec(vec, *x_to_ab(x))
282
283 def min_func_vec(x: np.ndarray,
284                  *args,
285                  **kwargs) -> float:
286      ...
287      Cost function to minimize when calibrating vector magnetometer
288
289      Parameters
290      -----
291      x
292          1x12 calibration vector
293      *args
294          list of distorted and "true" measurements -> [b_distorted (nT), b_true (nT)]
295
296      Returns
297      -----
298      float
299          Calibration cost
300          ...
301
302      b_distorted = args[0]
303      b_true      = args[1]
304
305      guess      = apply_calx_to_vec(b_distorted, x)
306      vec_diff = np.linalg.norm((b_true - guess)**2, axis=1)
307
308      return vec_diff.sum()
309
310 def calx(x0:           np.ndarray,
311           b_distorted: np.ndarray,
312           b_true:       np.ndarray,
313           min_func) -> np.ndarray:
314      ...
315      Find the calibration vector that minimizes the given cost function
316
317      Parameters
318      -----
319      x
320          1x12 calibration vector
321      *args
322          list of distorted and "true" measurements -> [b_distorted (nT), b_true (nT)]
323
324      Returns

```

```

325      _____
326      np.ndarray
327          1x12 calibration vector
328      ...
329
330      return minimize(min_func, x0, args=(b_distorted, b_true), method='Nelder-Mead').x
331
332  def calibrate_vec(x0: np.ndarray,
333                      b_distorted: np.ndarray,
334                      b_true: np.ndarray,
335                      max_iters: int=100,
336                      converge_lim: float=0.1) -> np.ndarray:
337      ...
338      Iterative approach to calibrate vector magnetometer data
339
340      Parameters
341      _____
342      x0
343          1x12 calibration vector (initial guess)
344      b_distorted
345          Nx3 array of uncalibrated vector measurements (nT)
346          where N is the number of samples taken in the
347          calibration
348      b_true
349          Nx3 array of the IGRF Earth field (nT) rotated into the sensor frame
350          where N is the number of samples taken in the
351          calibration
352      max_iters
353          Maximum number of iterations the calibration is allowed to run
354      converge_lim
355          The minimum magnitude difference between the previous and current
356          calibration iterations that denote convergence
357
358      Returns
359      _____
360      calibrated
361          Nx3 array of calibrated vector measurements
362          where N is the number of samples taken in the
363          calibration
364      x
365          1x12 calibration vector (final estimate)
366      ...
367
368      distorted = b_distorted
369      x = calx(x0, b_distorted, b_true, min_func_vec)
370      calibrated = apply_calx_to_vec(b_distorted, x)
371
372      i = 0
373      dx = la.norm(distorted - calibrated)
374
375      while (dx > converge_lim) and (i < max_iters):
376          distorted = b_distorted
377          x = calx(x, b_distorted, b_true, min_func_vec)
378          calibrated = apply_calx_to_vec(b_distorted, x)

```

```

379
380     i += 1
381     dx = la.norm(distorted - calibrated)
382
383     if i > max_iters:
384         print('calibrate_vec_took_too_many_iterations_to_converge')
385
386     return calibrated, x
387
388 def plot_sphere(ax,
389                 radius: float,
390                 color: str='r'):
391     """
392     Add a wireframe sphere to an axis ax
393
394     Parameters
395     -----
396     ax
397         Axis object to draw in, must support 3d projections
398     radius
399         Radius of sphere to draw
400     color
401         Color of sphere to draw, defaults to red
402     """
403
404     u, v = np.mgrid[0:2*np.pi:20j, 0:np.pi:10j]
405
406     x = np.cos(u) * np.sin(v) * radius
407     y = np.sin(u) * np.sin(v) * radius
408     z = np.cos(v) * radius
409
410     ax.plot_wireframe(x, y, z, color=color)
411
412 def plot_spin_data(b_vec_true: np.ndarray,
413                     b_vec_dist: np.ndarray,
414                     b_vec_cal: np.ndarray=None):
415     """
416     Plot vector data from spin test (before or after calibration)
417
418     Parameters
419     -----
420     b_vec_true
421         Nx3 array of ideal vector measurements (Earth's
422         field rotated in the sensor's frame with no
423         distortion)
424     b_vec_dist
425         Nx3 array of distorted vector measurements (Earth's
426         field rotated in the sensor's frame with distortion)
427     b_vec_cal
428         Nx3 array of calibrated vector measurements
429     """
430
431     fig = plt.figure()
432     ax = fig.add_subplot(111, projection='3d')

```

```

433
434     b_earth_mag = la.norm(b_vec_true, axis=1).mean()
435
436     plot_sphere(ax, b_earth_mag)
437
438     b_vec_true_xs = b_vec_true[:, 0]
439     b_vec_true_ys = b_vec_true[:, 1]
440     b_vec_true_zs = b_vec_true[:, 2]
441
442     ax.scatter(b_vec_true_xs,
443                 b_vec_true_ys,
444                 b_vec_true_zs,
445                 label="True_Earth_Field")
446
447     b_vec_dist_xs = b_vec_dist[:, 0]
448     b_vec_dist_ys = b_vec_dist[:, 1]
449     b_vec_dist_zs = b_vec_dist[:, 2]
450
451     ax.scatter(b_vec_dist_xs,
452                 b_vec_dist_ys,
453                 b_vec_dist_zs,
454                 label="Distorted_Measurements")
455
456     if b_vec_cal is not None:
457         b_vec_cal_xs = b_vec_cal[:, 0]
458         b_vec_cal_ys = b_vec_cal[:, 1]
459         b_vec_cal_zs = b_vec_cal[:, 2]
460
461         ax.scatter(b_vec_cal_xs,
462                     b_vec_cal_ys,
463                     b_vec_cal_zs,
464                     marker='x',
465                     label="Calibrated_Measurements")
466
467     plt.title('Spin_Test_Data')
468     ax.legend()
469
470
471 if __name__ == '__main__':
472     lat    = 39.775784
473     lon    = -84.109811
474     height = 160
475     date   = dt.datetime.utcnow()
476
477     print('B_Earth_NED_IGRF:\n', b_earth_ned_igrf(lat, lon, height, date))
478
479     eulers = np.array([[ 0,  0,  0],  # No rotation
480                       [90,  0,  0],  # 90 deg roll right
481                       [ 0, 90,  0],  # 90 deg pitch up
482                       [ 0,  0, 90],  # 90 deg yaw right
483                       [90, 90,  0],  # 90 deg pitch up then 90 deg roll right
484                       [ 0, 90, 90],  # 90 deg yaw right then 90 deg pitch up
485                       [90,  0, 90],  # 90 deg yaw right then 90 deg roll right
486                       [90, 90, 90]]) # 90 deg yaw right then 90 deg pitch up then 90 deg roll right

```

```

487
488     b_true = gen_b_truth_euler(lat, lon, height, date, eulers)
489
490     print('True:\n', b_true)
491     print('')
492
493     a = np.array([[1.1, 0, 0],
494                   [0, 1.2, 0],
495                   [0, 0, 1.3]])
496     b = np.array([3, 4, 2])
497
498     b_distorted = apply_dist_to_vec(b_true, a, b)
499
500     print('Distorted:\n', b_distorted)
501     print('Distortion\u2022Vector:\n', ab_to_x(a, b))
502     print('')
503
504     a = np.eye(3)
505     b = np.zeros(3)
506     x0 = ab_to_x(a, b)
507
508     calibrated, x = calibrate_vec(x0, b_distorted, b_true)
509
510     print('Estimated\u2022Distortion\u2022Vector:\n', x)
511     print('Calibrated:\n', calibrated)
512     print('True\u2022vs\u2022Calibrated\u2022Difference:\n', b_true - calibrated)
513     print('')
514
515     plot_spin_data(b_true,
516                      b_distorted,
517                      calibrated)
518     plt.show()

```

### mammal/MAMMAL/Parse/\_\_init\_\_.py

```

1 import os, sys
2
3 sys.path.append(os.path.dirname(os.path.realpath(__file__)))
4 sys.path.append(os.path.dirname(os.path.dirname(os.path.realpath(__file__))))

```

### mammal/MAMMAL/Parse/parseGeometrics.py

```

1 import sys
2 import datetime as dt
3 from os import listdir
4 from os.path import dirname, join, realpath
5
6 import numpy as np
7 import pandas as pd
8 import re
9 from tqdm import tqdm
10

```

```

11     sys.path.append(dirname(realpath(__file__)))
12     sys.path.append(dirname(dirname(realpath(__file__))))
13
14     from Utils.ProcessingUtils import add_igrf_cols
15
16
17     def natural_sort(l):
18         ...
19
20         Credits:
21             - https://stackoverflow.com/a/4836734/9860973
22             ...
23
24         convert      = lambda text: int(text) if text.isdigit() else text.lower()
25         alphanum_key = lambda key: [convert(c) for c in re.split('([0-9]+)', key)]
26
27         return sorted(l, key=alphanum_key)
28
29     def parse_devLog(fname: str,
30                      start_dt: dt.datetime=dt.datetime.utcnow(),
31                      lat: float=0,
32                      lon: float=0,
33                      alt: float=0,
34                      rej_thresh: float=500,
35                      fast_mode: bool=True,
36                      chunk: int=1000) -> pd.DataFrame:
37
38         Parse the MFAM dev kit SD log file and return a
39         pandas data frame with the resulting parsed data
40
41         Parameters
42         -----
43         fname
44             File path/name to the dev kit log file to parse
45         start_dt
46             Start date/time of the log in UTC
47         lat
48             Approximate geodetic latitude of the collect (dd)
49         lon
50             Approximate geodetic longitude of the collect (dd)
51         alt
52             Approximate altitude of the collect above MSL (m)
53         rej_thresh
54             Core field rejection threshold (nT). Scalar samples outside the range
55             determined by the calculated IGRF field +/- this threshold will
56             be removed from the dataset. For instance, if this threshold is
57             500nT and the IGRF field is 50000nT, a scalar sample of 7000nT
58             would be rejected, where a sample of 50300nT would not.
59         fast_mode
60             Calculate IGRF values based only on the first coordinate
61             in the dev kit log file (speeds up parsing considerably)
62         chunk
63             If not parsing in 'fast_mode', set the number of samples
64             to simultaneously calculate IGRF values for

```

```

65      Returns
66  -----
67  pd.DataFrame
68      Dataframe of data parsed from the given dev kit log
69      file – includes the following columns/fields at
70      a minimum:
71
72      – F:          Magnetic field measurement magnitude (nT)
73      – datetime:   Datetime object (UTC)
74      – epoch_sec:  UNIX epoch timestamp (s)
75      – ID:         Global millisecond counter
76      – FIDUCIAL:   Number of samples since last PPS pulse
77      – FRAME_ID:   Subset of the FrameID value
78      – SYS_STATUS: See manual
79      – SCALAR_1:   Sensor head 1 value (nT)
80      – SCALAR_1_STATUS: Sensor head 1 status
81      – SCALAR_2:   Sensor head 2 value (nT)
82      – SCALAR_2_STATUS: Sensor head 1 status
83      – AUX_0:      See manual
84      – AUX_1:      See manual
85      – AUX_2:      See manual
86      – AUX_3:      See manual
87      – LAT:        Latitude (dd)
88      – LONG:       Longitude (dd)
89      – ALT:        Altitude MSL (km)
90      – IGRF_X:    IGRF magnetic field in the North direction (nT)
91      – IGRF_Y:    IGRF magnetic field in the East direction (nT)
92      – IGRF_Z:    IGRF magnetic field in the Down direction (nT)
93      – IGRF_F:    IGRF magnetic field magnitude (nT)
94      ,,
95
96  pps_id = None
97  pps_dt = None
98
99  true_lat = None
100 true_lon = None
101 true_alt = None
102
103 ID          = []
104 FIDUCIAL    = []
105 FRAME_ID    = []
106 SYS_STATUS  = []
107 SCALAR_1    = []
108 SCALAR_1_STATUS = []
109 SCALAR_2    = []
110 SCALAR_2_STATUS = []
111 AUX_0       = []
112 AUX_1       = []
113 AUX_2       = []
114 AUX_3       = []
115
116 with open(fname, 'r') as log:
117     contents = log.read()
118

```

```

119     for _, line in enumerate(contents.split('\n')):
120         line_pieces = line.split(',')
121
122         if len(line_pieces) == 12:
123             try:
124                 int(line_pieces[0])
125                 int(line_pieces[1])
126                 int(line_pieces[2])
127                 int(line_pieces[3])
128                 float(line_pieces[4])
129                 int(line_pieces[5])
130                 float(line_pieces[6])
131                 int(line_pieces[7])
132                 int(line_pieces[8])
133                 int(line_pieces[9])
134                 int(line_pieces[10])
135                 int(line_pieces[11])
136
137                 ID.append(int(line_pieces[0]))
138                 FIDUCIAL.append(int(line_pieces[1]))
139                 FRAME_ID.append(int(line_pieces[2]))
140                 SYS_STATUS.append(int(line_pieces[3]))
141                 SCALAR_1.append(float(line_pieces[4]))
142                 SCALAR_1_STATUS.append(int(line_pieces[5]))
143                 SCALAR_2.append(float(line_pieces[6]))
144                 SCALAR_2_STATUS.append(int(line_pieces[7]))
145                 AUX_0.append(int(line_pieces[8]))
146                 AUX_1.append(int(line_pieces[9]))
147                 AUX_2.append(int(line_pieces[10]))
148                 AUX_3.append(int(line_pieces[11]))
149             except ValueError:
150                 pass
151
152         elif line.startswith('PPS:'):
153             subline = line[line.index('PPS:'):]
154             pieces = subline.split(', ')
155
156             pps_id = int(pieces[1])
157             pps_str = pieces[2]
158
159             pps_str = ''.join(pps_str.split())
160
161             try:
162                 pps_dt = dt.datetime.strptime(pps_str, r'%a %b %d %H%M%S %Y').replace(tzinfo=dt.timezone.utc)
163             except ValueError:
164                 pass
165
166         elif '$GPGGA' in line:
167             subline = line[line.index('$GPGGA'):]
168             pieces = subline.split(',')
169
170             if int(pieces[6]) > 0: # Process if GNSS data is valid
171                 lat_dm = float(pieces[2])
172                 lat_dir = pieces[3]

```

```

173     lat_d      = int(lat_dm / 100)
174     lat_m      = lat_dm - (lat_d * 100)
175     true_lat = lat_d + (lat_m / 60.0)
176
177     if lat_dir == 'S':
178         true_lat *= -1
179
180     lon_dm    = float(pieces[4])
181     lon_dir   = pieces[5]
182     lon_d     = int(lon_dm / 100)
183     lon_m     = lon_dm - (lon_d * 100)
184     true_lon = lon_d + (lon_m / 60.0)
185
186     if lon_dir == 'W':
187         true_lon *= -1
188
189     true_alt = float(pieces[9])
190
191     # Create/condition the DataFrame
192     df = pd.DataFrame({ 'ID': ID[1:] ,
193                         'FIDUCIAL': FIDUCIAL[1:] ,
194                         'FRAME_ID': FRAME_ID[1:] ,
195                         'SYS_STATUS': SYS_STATUS[1:] ,
196                         'SCALAR_1': SCALAR_1[1:] ,
197                         'SCALAR_1_STATUS': SCALAR_1_STATUS[1:] ,
198                         'SCALAR_2': SCALAR_2[1:] ,
199                         'SCALAR_2_STATUS': SCALAR_2_STATUS[1:] ,
200                         'AUX_0': AUX_0[1:] ,
201                         'AUX_1': AUX_1[1:] ,
202                         'AUX_2': AUX_2[1:] ,
203                         'AUX_3': AUX_3[1:] })
204
205     df.SCALAR_1 = df.SCALAR_1.astype(float)
206     df.SCALAR_2 = df.SCALAR_2.astype(float)
207     df.ID       = df.ID.astype(int)
208
209     # Add columns for compatibility
210     if pps_id is None:
211         df['epoch_sec'] = ((df.ID - df.ID.iloc[0]) / 1000.0) + start_dt.timestamp()
212     else:
213         df['epoch_sec'] = ((df.ID - pps_id) / 1000.0) + pps_dt.timestamp()
214
215     if true_lat is None:
216         df[ 'LAT' ] = np.ones(len(df.ID)) * lat
217     else:
218         df[ 'LAT' ] = np.ones(len(df.ID)) * true_lat
219
220     if true_lon is None:
221         df[ 'LONG' ] = np.ones(len(df.ID)) * lon
222     else:
223         df[ 'LONG' ] = np.ones(len(df.ID)) * true_lon
224
225     if true_alt is None:
226         df[ 'ALT' ] = np.ones(len(df.ID)) * alt

```

```

227     df[ 'ALT' ] = np.ones(len(df.ID)) * true_alt
228
229     df[ 'datetime' ] = pd.to_datetime(df.epoch_sec, unit='s')
230     df[ 'SCALAR_1_VALID' ] = np.ones(len(df.ID)) * np.nan
231     df[ 'SCALAR_2_VALID' ] = np.ones(len(df.ID)) * np.nan
232     df[ 'X' ] = np.ones(len(df.ID)) * np.nan
233     df[ 'Y' ] = np.ones(len(df.ID)) * np.nan
234     df[ 'Z' ] = np.ones(len(df.ID)) * np.nan
235     df[ 'F' ] = df.SCALAR_1
236
237     df = add_igrf_cols(df, fast_mode, chunk)
238
239     # Find valid min/max values
240     min_F = df.IGRF_F - rej_thresh
241     max_F = df.IGRF_F + rej_thresh
242
243     # Find rows that have valid samples for each sensor head
244     scalar_1_valid_mask = (df.SCALAR_1 >= min_F) & (df.SCALAR_1 <= max_F)
245     scalar_2_valid_mask = (df.SCALAR_2 >= min_F) & (df.SCALAR_2 <= max_F)
246
247     df[ 'SCALAR_1_VALID' ] = scalar_1_valid_mask
248     df[ 'SCALAR_2_VALID' ] = scalar_2_valid_mask
249
250     # Find when both and neither sensor head values are valid
251     both_valid_mask = scalar_1_valid_mask & scalar_2_valid_mask
252     neither_valid_mask = ~(scalar_1_valid_mask | scalar_2_valid_mask)
253
254     # Test if all data is "bad"
255     if neither_valid_mask.all():
256         both_valid_mask = ~both_valid_mask
257         neither_valid_mask = ~neither_valid_mask
258
259     print('WARNING: All scalar data was found to be outside the acceptable range from the expected IGRF magnitude, no data will be clipped!')
260
261     # Find when only sensor head 2 is valid
262     only_scalar_2_valid_mask = scalar_2_valid_mask & (~scalar_1_valid_mask)
263
264     # Use sensor head 2 when sensor head 1 is invalid
265     df.F.iloc[only_scalar_2_valid_mask] = df.SCALAR_2[only_scalar_2_valid_mask]
266
267     # Average sensor values when both are valid
268     df.F.iloc[both_valid_mask] = (df.SCALAR_1[both_valid_mask] + df.SCALAR_2[both_valid_mask]) / 2.0
269
270     # Drop when both heads are invalid
271     df.F.iloc[neither_valid_mask] = 0.0
272
273     return df
274
275 def parse_devACQU(dir: str,
276                     start_dt: dt.datetime=dt.datetime.utcnow(),
277                     lat: float=0,
278                     lon: float=0,
279                     alt: float=0,

```

```

280             rej_thresh: float=500,
281             fast_mode: bool=True,
282             chunk:      int=1000) -> pd.DataFrame:
283             ...
284
285             Parse all logs from a single acquisition on a MFAM
286             dev kit SD log and return a pandas data frame with
287             the resulting parsed data
288
289             Parameters
290             -----
291             dir
292                 File path to the dev kit survey acquisition directory
293             start_dt
294                 Start date/time of the log in UTC
295             lat
296                 Approximate geodetic latitude of the collect (dd)
297             lon
298                 Approximate geodetic longitude of the collect (dd)
299             alt
300                 Approximate altitude of the collect above MSL (m)
301             rej_thresh
302                 Core field rejection threshold (nT). Scalar samples outside the range
303                 determined by the calculated IGRF field +/- this threshold will
304                 be removed from the dataset. For instance, if this threshold is
305                 500nT and the IGRF field is 50000nT, a scalar sample of 7000nT
306                 would be rejected, where a sample of 50300nT would not.
307             fast_mode
308                 Calculate IGRF values based only on the first coordinate
309                 in the dev kit log file (speeds up parsing considerably)
310             chunk
311                 If not parsing in 'fast_mode', set the number of samples
312                 to simultaneously calculate IGRF values for
313
314             Returns
315             -----
316             pd.DataFrame
317
318                 Dataframe of data parsed from the given dev kit log
319                 file - includes the following columns/fields at
320                 a minimum:
321
322                 - F:          Magnetic field measurement magnitude (nT)
323                 - datetime:   Datetime object (UTC)
324                 - epoch_sec:  UNIX epoch timestamp (s)
325                 - ID:         Global millisecond counter
326                 - FIDUCIAL:   Number of samples since last PPS pulse
327                 - FRAME_ID:   Subset of the FrameID value
328                 - SYS_STATUS: See manual
329                 - SCALAR_1:    Sensor head 1 value (nT)
330                 - SCALAR_1_STATUS: Sensor head 1 status
331                 - SCALAR_2:    Sensor head 2 value (nT)
332                 - SCALAR_2_STATUS: Sensor head 1 status
333                 - AUX_0:       See manual
334                 - AUX_1:       See manual
335                 - AUX_2:       See manual

```

```

334      - AUX_3           See manual
335      - LAT:            Latitude (dd)
336      - LONG:           Longitude (dd)
337      - ALT:             Altitude MSL (km)
338      - IGRF_X:         IGRF magnetic field in the North direction (nT)
339      - IGRF_Y:         IGRF magnetic field in the East direction (nT)
340      - IGRF_Z:         IGRF magnetic field in the Down direction (nT)
341      - IGRF_F:          IGRF magnetic field magnitude (nT)
342      ...
343
344      df_list = []
345
346      file_list = natural_sort(listdir(dir))
347
348      for log in tqdm(file_list):
349          if not log.lower().endswith('.txt'):
350              continue
351
352          try:
353              int(log.split('.')[0])
354          except:
355              continue
356
357          if len(df_list) > 0:
358              start_dt = df_list[-1].datetime.iloc[-1]
359
360          df_list.append(parse_devLog(join(dir, log),
361                               start_dt,
362                               lat,
363                               lon,
364                               alt,
365                               rej_thresh,
366                               fast_mode,
367                               chunk))
368
369      return pd.concat(df_list)

```

## mammal/MAMMAL/Parse/parseGSMP.py

```

1 import sys
2 import datetime as dt
3 from os.path import dirname, realpath
4
5 import numpy as np
6 import pandas as pd
7
8 sys.path.append(dirname(realpath(__file__)))
9 sys.path.append(dirname(dirname(realpath(__file__))))
10
11 from Utils.ProcessingUtils import add_igrf_cols
12
13
14 def parse_date(fname: str) -> dt.datetime:

```

```

15      """
16      Find collection start date in GSMP log file
17
18      Parameters
19      -----
20      fname
21          File path/name to the GSMP log file
22
23      Returns
24      -----
25      dt.datetime
26          Collection start date in GSMP log file
27      ...
28
29      with open(fname, 'r') as inFile:
30          contents = inFile.readlines()
31
32      for i, line in enumerate(contents):
33          if line.strip() == 'G':
34              return dt.datetime.strptime(contents[i + 1][:-1], '%d.%m.%Y')
35      raise Exception('Could not find start date')
36
37  def skip_lines(fname: str) -> int:
38      ...
39      Find number of header lines to skip when reading GSMP log file
40      as a dataframe
41
42      Parameters
43      -----
44      fname
45          File path/name to the GSMP log file
46
47      Returns
48      -----
49      int
50          Number of header lines to skip when reading GSMP log file
51          as a dataframe
52      ...
53
54      with open(fname, 'r') as inFile:
55          contents = inFile.readlines()
56
57      for i, line in enumerate(contents):
58          if 'hhmmss.s' in line:
59              return i
60      return 0
61
62  def fix_datetime(df: dt.datetime) -> pd.DataFrame:
63      ...
64      Corrected TIME and DATE columns in dataframe of
65      GSMP log file data, plus add datetime and
66      epoch_sec columns
67
68      Parameters

```

```

69      _____
70      df
71          Dataframe of GSMP log file data
72
73      Returns
74      _____
75      pd.DataFrame
76          Dataframe of GSMP log file data with corrected
77          TIME and DATE columns, plus additional datetime
78          and epoch_sec columns
79      ...
80
81      # Fix TIME column to a pure seconds counter
82      hrs = np.floor(df.TIME / 10000)
83      mins = np.floor(df.TIME / 100) % 100
84      secs = df.TIME % 100
85
86      df.TIME = (hrs * 3600) + (mins * 60) + secs
87      time_diff = np.diff(df.TIME, prepend=df.TIME[0])
88      time_diff[time_diff < 0] = 1
89
90      # Create the datetime column while handling rollover
91      df['datetime'] = df.DATE + \
92                      pd.to_timedelta(df.TIME[0], unit='s') + \
93                      pd.to_timedelta(np.cumsum(time_diff), unit='s')
94
95      # Add column for total number of seconds after epoch to make it easier
96      # to interpolate between readings of multiple datasets
97      df['epoch_sec'] = (df['datetime'] - pd.Timestamp('1970-01-01')).dt.total_seconds()
98
99      # Fix DATE column to handle rollover
100     sec_offset = np.array(df.epoch_sec - df.epoch_sec[0] + df.TIME[0], dtype=int)
101     day_offset = np.array(sec_offset / 86400, dtype=int)
102
103     df.DATE = df.DATE + pd.to_timedelta(day_offset, unit='d')
104
105     return df
106
107 def parse_GSMP(fname: str,
108                 fast_mode: bool=True,
109                 chunk: int=1000) -> pd.DataFrame:
110     ...
111
112     Parse the GSMP log file and return a
113     pandas data frame with the resulting parsed data
114
115     Parameters
116     _____
117     fname
118         File path/name to the GSMP log file to parse
119         fast_mode
120             Calculate IGRF values based only on the first coordinate
121             in the GSMP log file (speeds up parsing considerably)
122         chunk
123             If not parsing in 'fast_mode', set the number of samples

```

```

123      to simultaneously calculate IGRF values for
124
125      Returns
126      -----
127      pd.DataFrame
128          Dataframe of data parsed from the given GSMP log
129          file – includes the following columns/fields at
130          a minimum:
131
132          - DATE: Date object (UTC)
133          - TIME: Number of seconds past UTC midnight
134          - F: Magnetic field measurement magnitude (nT)
135          - datetime: Datetime object (UTC)
136          - epoch_sec: UNIX epoch timestamp (s)
137          - LAT: Latitude (dd)
138          - LONG: Longitude (dd)
139          - ALT: Altitude MSL (km)
140          - IGRF_X: IGRF magnetic field in the North direction (nT)
141          - IGRF_Y: IGRF magnetic field in the East direction (nT)
142          - IGRF_Z: IGRF magnetic field in the Down direction (nT)
143          - IGRF_F: IGRF magnetic field magnitude (nT)
144
145      ,
146      date = parse_date(fname)
147      skip = skip_lines(fname)
148      df = pd.read_csv(fname, delim_whitespace=True, skiprows=skip)
149
150      # Renaming columns for compatibility
151      df.rename(columns={'nT': 'F'}, inplace=True)
152      df.rename(columns={'hhmmss.s': 'TIME'}, inplace=True)
153      df.rename(columns={'lat': 'LAT'}, inplace=True)
154      df.rename(columns={'lon': 'LONG'}, inplace=True)
155      df.rename(columns={'alt': 'ALT'}, inplace=True)
156
157      # Add empty vector columns for compatibility
158      df['X'] = np.ones(len(df.F)) * np.nan
159      df['Y'] = np.ones(len(df.F)) * np.nan
160      df['Z'] = np.ones(len(df.F)) * np.nan
161
162      df['DATE'] = date
163
164      fix_datetime(df)
165
166      # Remove all rows of data with invalid GNSS fixes
167      df = df[df.sat >= 4]
168
169      return add_igrf_cols(df, fast_mode, chunk)

```

mammal/MAMMAL/Parse/parseSGL.py

```

1 import numpy as np
2 import pandas as pd
3

```

```

4
5  def parse_xyz(fname:      str,
6          drop_dups:  bool=False,
7          drop_cust:  list=[]) -> pd.DataFrame:
8      """
9          Parse the xyz sensor file name and return a pandas data
10         frame with the resulting parsed data
11
12     Parameters
13     -----
14     fname
15         File path/name to the .xyz file to parse
16     drop_dups
17         Drop rows with duplicate 'TIME' values if set to True
18     drop_cust
19         String or list of strings of column names to drop
20
21     Returns
22     -----
23     pd.DataFrame
24         Dataframe of SGL data parsed from the given .xyz
25         file - *usually* includes the following columns/fields:
26
27         - LINE:      Line Number XXXX.YY where XXXX is line number and YY is segment number
28         - FLT:       Flight Number
29         - YEAR:     Year
30         - DOY:      Julian day of year (UTC)
31         - TIME:     Number of seconds past UTC midnight
32         - UTM-X:    X coordinate, WGS-84 UTM ZONE 11N (m)
33         - UTM-Y:    Y coordinate, WGS-84 UTM ZONE 11N (m)
34         - UTM-Z:    GPS Elevation (above WGS-84 Ellipsoid) (m)
35         - MSL-Z:    GPS Elevation (above EGM2008 Geoid) (m)
36         - LAT:      Latitude, WGS-84 (dd)
37         - LONG:     Longitude, WGS-84 (dd)
38         - BARO:     Barometric Altimeter (m)
39         - RADAR:    Filtered Radar Altimeter (m)
40         - TOPO:     Radar Topography (above EGM2008 Geoid) (m)
41         - DEM:      Digital Elevation Model from SRTM (above EGM2008 Geoid) (m)
42         - PITCH:    System computed aircraft pitch (deg)
43         - ROLL:     System computed aircraft roll (deg)
44         - AZIMUTH:   System computed aircraft azimuth (deg)
45         - DIURNAL:   Diurnal Magnetic Field from reference station (nT)
46         - FLUX_X:    Fluxgate x-axis (nT)
47         - FLUX_Y:    Fluxgate y-axis (nT)
48         - FLUX_Z:    Fluxgate z-axis (nT)
49         - FLUX_TOT:  Flugate total (nT)
50         - UNCOMPMAG: Uncompensated Airborne Magnetic Field (nT)
51         - COMPMAG:   Compensated Airborne Magnetic Field (nT)
52         - DCMAG:     Diurnal Corrected Airborne Magnetic Field (nT)
53         - IGRFMAG:   IGRF and Diurnal Corrected Airborne Magnetic Field (nT)
54         - LVLDmag:   Levelled Magnetic Anomaly (nT)
55         - datetime:  Datetime object (UTC)
56         - epoch_sec: UNIX epoch timestamp (s)
57     """

```

```

58
59     df = pd.read_csv(fname, header=2, delim_whitespace=True, na_values='*')
60     df.columns = np.hstack(([df.columns[1:].values, np.array(['extra_header'])])) # shift over the header names and pad an
       extra for the N/A column of data
61     df = df.iloc[:, :-1] # drop the last N/A column of data
62
63     if 'FTIME' in df.columns:
64         df.rename(columns={'FTIME': 'TIME'}, inplace=True)
65
66     for col in drop_cust:
67         df.drop(col, axis=1, inplace=True)
68
69     # Remove rows with nan-date/time values for the datetime column creation to work
70     df.drop(df.index[df['YEAR'].isnull()], axis=0, inplace=True)
71     df.drop(df.index[df['DOY'].isnull()], axis=0, inplace=True)
72     df.drop(df.index[df['TIME'].isnull()], axis=0, inplace=True)
73
74     # Reindex DataFrame
75     df.index = range(len(df))
76
77     # Want some columns as string for timestamp parsing YEAR, DOY, and day-seconds
78     # Want them as native types otherwise for eas of use
79     df['datetime'] = pd.to_datetime(df['YEAR'].astype(int).astype(str) + df['DOY'].astype(int).astype(str),
80                                    format='%Y%j',
81                                    errors='coerce') + pd.to_timedelta(df['TIME'], unit='s')
82
83     # Fix these fields to be true UTC julian day/day-seconds
84     df['DOY'] = df['datetime'].dt.dayofyear
85     df['TIME'] = df['TIME'] % 86399
86
87     # Add column for total number of seconds after epoch to make it easier
88     # to interpolate between readings of multiple datasets
89     df['epoch_sec'] = (df['datetime'] - pd.Timestamp('1970-01-01')).dt.total_seconds()
90
91     # There are sometimes duplicated rows -- they are associated with different lines, but the
92     # other information is duplicate
93     if drop_dups:
94         df.drop_duplicates(subset='TIME', inplace=True, keep='first')
95
96     return df.sort_values(by=['datetime'])

```

## mammal/MAMMAL/Parse/parseIM.py

```

1 import os
2 import sys
3 from os.path import join, dirname, realpath
4
5 import pandas as pd
6
7 sys.path.append(dirname(realpath(__file__)))
8 sys.path.append(dirname(dirname(realpath(__file__))))
9
10 from Utils.ProcessingUtils import add_igrf_cols

```

```

11
12
13 def skip_lines(fname: str) -> int:
14     """
15         Find number of header lines to skip when reading .sec file
16         as a dataframe
17
18     Parameters
19     -----
20     fname
21         File path/name to the .sec file
22
23     Returns
24     -----
25     int
26         Number of header lines to skip when reading .sec file
27         as a dataframe
28     """
29
30     with open(fname, 'r') as inFile:
31         contents = inFile.readlines()
32
33     for i, line in enumerate(contents):
34         if '|' not in line:
35             return i - 1
36
37     return 0
38
39 def add_lls_cols(df: pd.DataFrame,
40                  fname: str) -> pd.DataFrame:
41     """
42         Add latitude, longitude, and altitude columns for all samples
43         in the dataframe (lat/lon in dd and alt in km above MSL)
44
45     Parameters
46     -----
47     df
48         Dataframe of INTERMAGNET data
49     fast_mode
50         Calculate IGRF values based only on the first coordinate
51         in the .sec file (speeds up parsing considerably)
52     chunk
53         If not parsing in 'fast_mode', set the number of samples
54         to simultaneously calculate IGRF values for
55
56     Returns
57     -----
58     pd.DataFrame
59         Dataframe of INTERMAGNET data with new latitude,
60         longitude, and altitude columns
61     """
62
63     with open(fname, 'r') as inFile:
64         header = inFile.readlines()[:25]

```

```

65     header = ''.join(header)
66
67     lat = float(header.split()[header.split().index('Latitude') + 1]) # (dd)
68     lon = float(header.split()[header.split().index('Longitude') + 1]) # (dd)
69     alt = float(header.split()[header.split().index('Elevation') + 1]) # (m)
70
71     if lon > 180:
72         lon -= 360
73
74     df[ 'LAT' ] = lat
75     df[ 'LONG' ] = lon
76     df[ 'ALT' ] = alt
77
78     return df
79
80 def parse_sec(fname:      str,
81                 fast_mode: bool=True,
82                 chunk:      int=1000) -> pd.DataFrame:
83     """
84     Parse the INTERMAGNET (.sec) sensor file and return a
85     pandas data frame with the resulting parsed data
86
87     Parameters
88     -----
89     fname
90         File path/name to the .sec file to parse
91     fast_mode
92         Calculate IGRF values based only on the first coordinate
93         in the .sec file (speeds up parsing considerably)
94     chunk
95         If not parsing in 'fast_mode', set the number of samples
96         to simultaneously calculate IGRF values for
97
98     Returns
99     -----
100    pd.DataFrame
101
102        Dataframe of INTERMAGNET data parsed from the given .sec
103        file - includes the following columns/fields:
104
105        - DATE: Date object (UTC)
106        - TIME: Number of seconds past UTC midnight
107        - DOY: Julian day of year (UTC)
108        - X: Magnetic field measurement in the North direction (nT)
109        - Y: Magnetic field measurement in the East direction (nT)
110        - Z: Magnetic field measurement in the Down direction (nT)
111        - F: Magnetic field measurement magnitude (nT)
112        - datetime: Datetime object (UTC)
113        - epoch_sec: UNIX epoch timestamp (s)
114        - LAT: Latitude (dd)
115        - LONG: Longitude (dd)
116        - ALT: Altitude MSL (km)
117        - IGRF_X: IGRF magnetic field in the North direction (nT)
118        - IGRF_Y: IGRF magnetic field in the East direction (nT)
119        - IGRF_Z: IGRF magnetic field in the Down direction (nT)

```

```

119      - IGRF_F:    IGRF magnetic field magnitude (nT)
120      '',
121
122      if fname.endswith('.sec'):
123          skip = skip_lines(fname)
124
125      df = pd.read_csv(fname,
126                         header=skip,
127                         delim_whitespace=True,
128                         na_values='99999.0')
129
130      del df['|'] # Get rid of extra column (artifact of how .sec file headers are formatted)
131
132      df.columns = [ 'DATE', 'TIME', 'DOY', 'X', 'Y', 'Z', 'F']
133      df[ 'TIME'] = pd.to_timedelta(df[ 'TIME']).dt.total_seconds()
134
135      df = df.dropna() # Must drop NaNs to get datetime column creation to work
136
137      # want some columns as string for timestamp parsing DOY, and day-seconds
138      # want them as native types otherwise for easof use
139      date = pd.to_datetime(df[ 'DATE'])
140      df[ 'datetime'] = pd.to_datetime(date.dt.year.astype(str) + df[ 'DOY'].astype(int).astype(str),
141                                         format='%Y%j',
142                                         errors='coerce') + pd.to_timedelta(df[ 'TIME'], unit='s')
143
144      # add column for total number of seconds after epoch to make it easier
145      # to interpolate between readings of multiple datasets
146      df[ 'epoch_sec'] = (df[ 'datetime'] - pd.Timestamp('1970-01-01')).dt.total_seconds()
147
148      df = add_lls_cols(df, fname)
149      df = add_igrf_cols(df, fast_mode, chunk)
150
151      return df.sort_values(by=['datetime'])
152
153  return None
154
155  def loadInterMagData(data_dir: str,
156                        fast_mode: bool=True,
157                        chunk: int=1000) -> dict:
158      ''
159      Walks through all the files saved in the InterMagnet data storage folder,
160      and saves the data from all '.sec' files in that folder/subfolders. For
161      each '.sec' file, the function decodes the file's data into a Pandas
162      dataframe, determines the location of the data source (i.e. data is from
163      Boulder CO or Touscon AZ, etc.), combines the data from the current '.sec'
164      file with the previously saved data for that location, sorts the entire
165      dataset of that location by day of year (DOY/Julian Day) and by
166      day-seconds, and drops all rows with NaNs. After all '.sec' files are
167      processed, a dictionary with combined data from each location is returned.
168
169      Parameters
170  -----
171      data_dir
172          Path to directory that holds all INTERMAGNET .sec files

```

```

173      to be parsed
174      fast_mode
175          Calculate IGRF values based only on the first coordinate
176          in the .sec file (speeds up parsing considerably)
177      chunk
178          If not parsing in 'fast_mode', set the number of samples
179          to simultaneously calculate IGRF values for
180
181      Returns
182      -----
183      dict
184          Dictionary that includes all INTERMAGNET data parsed from .sec files
185          found in 'data_dir'
186      ...
187
188      data = {}
189
190      for root, _, files in os.walk(data_dir):
191          for file in files:
192              if file.lower().endswith('.sec'):
193                  id = file[:3].upper()
194
195                  if id not in data.keys():
196                      data[id] = parse_sec(join(root, file),
197                                           fast_mode,
198                                           chunk)
199
200          else:
201              data[id] = pd.concat([data[id],
202                                   parse_sec(join(root, file),
203                                           fast_mode,
204                                           chunk)]).sort_values(by=['DOY', 'TIME']).dropna()
205
206          print('Loaded', file)
207
208      return data

```

## mammal/MAMMAL/Parse/parsePixhawk.py

```

1 import datetime as dt
2
3 import pandas as pd
4 from scipy import interpolate
5 from tqdm import tqdm
6
7
8 LEAP_SEC = 18
9
10
11 def gps2utc(gpsweek, gpsseconds, leapseconds=LEAP_SEC):
12     ...
13     https://gist.github.com/jeremiahjohnson/eca97484db88bcf6b124
14     ...

```

```

15
16     epoch = dt.datetime.strptime("1980-01-06_00:00:00", "%Y-%m-%d_%H%M%S").replace(tzinfo=dt.timezone.utc)
17     elapsed = dt.timedelta(days=(gpsweek * 7), seconds=(gpsseconds - leapseconds))
18
19     return epoch + elapsed
20
21 def parsePix(fname, alt_min=None, leapsec=LEAP_SEC):
22     """
23     Create Pandas DataFrame of timestamped LLA coordinates from log
24     """
25
26     gps_dict = { 'TimeUS': [] ,
27                  'GMS' : [] ,
28                  'GWk' : [] }
29     ekf_dict = { 'TimeUS': [] ,
30                  'LAT' : [] ,
31                  'LONG' : [] ,
32                  'ALT' : [] ,
33                  'PITCH' : [] ,
34                  'ROLL' : [] ,
35                  'AZIMUTH' : [] }
36
37     with open(fname, 'r') as inFile:
38         lines = inFile.readlines()
39
40     for line in tqdm(lines):
41         if line.startswith('GPS'):
42             pieces = line.split(',')
43
44             if len(pieces) == 16:
45                 gps_dict[ 'TimeUS'].append(float(pieces[1]))
46                 gps_dict[ 'GMS'].append(float(pieces[4]))
47                 gps_dict[ 'GWk'].append(int(pieces[5]))
48
49             elif line.startswith('AHR2'):
50                 pieces = line.split(',')
51
52                 if len(pieces) == 12:
53                     ekf_dict[ 'TimeUS'].append(float(pieces[1]))
54                     ekf_dict[ 'ROLL'].append(float(pieces[2]))
55                     ekf_dict[ 'PITCH'].append(float(pieces[3]))
56                     ekf_dict[ 'AZIMUTH'].append(float(pieces[4]))
57                     ekf_dict[ 'ALT'].append(float(pieces[5]))
58                     ekf_dict[ 'LAT'].append(float(pieces[6]))
59                     ekf_dict[ 'LONG'].append(float(pieces[7]))
60
61     gps_df = pd.DataFrame(gps_dict)
62     ekf_df = pd.DataFrame(ekf_dict)
63
64     gps_timeus = gps_df.TimeUS
65     gms        = gps_df.GMS
66     gwk        = gps_df.GWk
67
68     datetime  = [gps2utc(float(gwk[i]), float(gms[i]) / 1000.0, leapsec) for i in range(len(gps_df))]
```

```

69     epoch_sec = [datetime[i].timestamp() for i in range(len(gps_df))]
70
71     ekf_timeus = ekf_df.TimeUS
72
73     f = interpolate.interp1d(gps_timeus, epoch_sec, fill_value='extrapolate')
74     ekf_epoch_sec = f(ekf_timeus)
75
76     ekf_df['epoch_sec'] = ekf_epoch_sec
77     ekf_df['datetime'] = [dt.datetime.utcnow().timestamp() for ts in ekf_epoch_sec]
78
79     ekf_df = ekf_df[['epoch_sec', 'datetime', 'LAT', 'LONG', 'ALT', 'PITCH', 'ROLL', 'AZIMUTH']]
80
81     if alt_min is not None:
82         ekf_df = ekf_df[ekf_df.ALT >= alt_min]
83
84     return ekf_df

```

### mammal/MAMMAL/Parse/parseRaster.py

```

1 import rioxarray as rxr
2
3
4 WGS84_EPSG = 'EPSG:4326' # The EPSG for WGS-84 lat/lon is 4326
5
6
7 def parse_raster(fname: str,
8                  x_lim: list=None,
9                  y_lim: list=None) -> rxr.rioxarray.raster_dataset.xarray.DataArray:
10    ...
11
12    Read-in raster file, crop if x or y limits are provided and return
13    as a rioxarray object
14
15    Parameters
16    -----
17    fname
18        File name/path of the raster file
19    x_lim
20        Optional 1x2 array-like object that defines the minimum
21        and maximum x coordinates the raster can have -> [min x, max x]
22    y_lim
23        Optional 1x2 array-like object that defines the minimum
24        and maximum y coordinates the raster can have -> [min y, max y]
25
26    Returns
27    -----
28    rxr.rioxarray.raster_dataset.xarray.DataArray
29        DataArray of the (optionally) cropped raster
30    ...
31
32    ds = rxr.open_rasterio(fname, masked=True, decode_coords="all")
33
34    if x_lim is not None:
35        x_min = x_lim[0]

```

```

35         x_max = x_lim[1]
36         mask_x = (ds.x >= x_min) & (ds.x <= x_max)
37     else:
38         mask_x = ds.all()
39
40     if y_lim is not None:
41         y_min = y_lim[0]
42         y_max = y_lim[1]
43         mask_y = (ds.y >= y_min) & (ds.y <= y_max)
44     else:
45         mask_y = ds.all()
46
47     return ds.where(mask_x & mask_y, drop=True)

```

### mammal/MAMMAL/Utils/\_\_init\_\_.py

```

1 import os, sys
2
3 sys.path.append(os.path.dirname(os.path.realpath(__file__)))
4 sys.path.append(os.path.dirname(os.path.dirname(os.path.realpath(__file__))))

```

### mammal/MAMMAL/Utils/ProcessingUtils.py

```

1 import sys
2 from copy import deepcopy
3 from os.path import dirname, realpath
4
5 import numpy as np
6 import pandas as pd
7 import rioxarray as rxr
8 import scipy.linalg as la
9 from numpy import sin, cos
10 from scipy import interpolate
11 from scipy.spatial import distance
12 from sklearn.gaussian_process import GaussianProcessRegressor as GPR
13 from sklearn.gaussian_process.kernels import RBF, WhiteKernel as WK
14 from tqdm import tqdm
15 from ppigrf import igrf
16
17 sys.path.append(dirname(realpath(__file__)))
18 sys.path.append(dirname(dirname(realpath(__file__))))
19
20 import Diurnal
21 from MapLvl import pcaLvl
22 from MapLvl import tieLvl
23 from SensorCal import spinCal as sc
24 from VehicleCal import TL as tl
25 from Utils import coordinateUtils as cu
26 from Utils import Filters as filt
27 from Utils import mapUtils as mu
28
29

```

```

30  # Enumerate line directions
31  HORIZ = 0 # Horizontal
32  VERT  = 1 # Vertical
33
34
35  def rmse(arr_1, arr_2, axis=None):
36      mask = np.logical_and(~np.isnan(arr_1), ~np.isnan(arr_2))
37      return np.sqrt(np.mean(np.square(arr_1[mask] - arr_2[mask]), axis=axis))
38
39  def clip_data(unclipped, high_clip, low_clip):
40      """
41          Clip unclipped between high_clip and low_clip.
42          unclipped contains a single column of unclipped data.
43
44          Credit:
45          https://stackoverflow.com/a/71230493/9860973
46      """
47
48      # convert to np.array to access the np.where method
49      np_unclipped = np.array(unclipped)
50
51      # clip data above HIGH_CLIP or below LOW_CLIP
52      cond_high_clip = (np_unclipped > high_clip) | (np_unclipped < low_clip)
53      np_clipped     = np.where(cond_high_clip, np.nan, np_unclipped)
54
55      return np_clipped.tolist()
56
57
58  def ewma_fb(df_column, span):
59      """
60          Apply forwards, backwards exponential weighted moving average (EWMA) to df_column.
61
62          Credit:
63          https://stackoverflow.com/a/71230493/9860973
64      """
65
66      # Forwards EWMA.
67      fwd = pd.Series.ewm(df_column, span=span).mean()
68
69      # Backwards EWMA.
70      bwd = pd.Series.ewm(df_column[::-1], span=span).mean()
71
72      # Add and take the mean of the forwards and backwards EWMA.
73      stacked_ewma = np.vstack((fwd, bwd[::-1]))
74      fb_ewma     = np.mean(stacked_ewma, axis=0)
75
76      return fb_ewma
77
78
79  def remove_outliers(spikey, fbewma, delta):
80      """
81          Remove data from df_spikey that is > delta from fbewma.
82
83          Credit:

```

```

84      https://stackoverflow.com/a/71230493/9860973
85      ,
86
87      np_spikey = np.array(spikey)
88      np_fbewma = np.array(fbewma)
89
90      cond_delta      = (np.abs(np_spikey-np_fbewma) > delta)
91      np_remove_outliers = np.where(cond_delta, np.nan, np_spikey)
92
93      return np_remove_outliers
94
95  def reject_outliers(df: pd.DataFrame,
96                      window_size: int=100,
97                      std_lim: float=3,
98                      col: str='F') -> pd.DataFrame:
99      ,
100
101      Reject outliers within the scalar data
102
103      **NOTE**: This assumes 'F' column is already
104      included in the dataframe
105
106      Parameters
107      -----
108      df
109          Dataframe of INTERMAGNET data
110
111          Number of scalar samples within the window used for outlier
112          rejection. Set to 'None' to prevent outlier rejection
113
114          std_lim
115
116          Any values outside this number of standard deviations within
117          the given window of scalar values will be discarded
118
119          col
120
121          Name of column of data to process
122
123
124      Returns
125      -----
126      pd.DataFrame
127
128      Dataframe without outlier datapoints
129
130
131      new_df = deepcopy(df)
132
133      for i in tqdm(range(0, len(new_df[col]), window_size)):
134          window_df = deepcopy(new_df.iloc[i:i+window_size])
135          window_df[(window_df[col] - window_df[col].mean()).abs() >= (std_lim * window_df[col].std())] = np.nan
136          new_df.iloc[i:i+window_size] = window_df
137
138      return new_df.dropna()
139
140
141  def add_igrf_cols(df: pd.DataFrame,
142                     fast_mode: bool=True,
143                     chunk: int=1000) -> pd.DataFrame:
144
145      Add IGRF vector and magnitude columns for all samples

```

```

138      in the dataframe
139
140      **NOTE**: This assumes 'LONG', 'LAT', 'ALT', and
141      'datetime' columns are already included in the
142      dataframe
143
144      Parameters
145      -----
146      df
147          Dataframe of INTERMAGNET data
148      fast_mode
149          Calculate IGRF values based only on the first coordinate
150          in the .sec file (speeds up parsing considerably)
151      chunk
152          If not parsing in 'fast_mode', set the number of samples
153          to simultaneously calculate IGRF values for
154
155      Returns
156      -----
157      pd.DataFrame
158          Dataframe of INTERMAGNET data with new IGRF columns
159      ,
160
161      if fast_mode:
162          Be, Bn, Bu = igrf(df.LONG[~np.isnan(df.LONG)].mean(),
163                               df.LAT[~np.isnan(df.LAT)].mean(),
164                               df.ALT[~np.isnan(df.ALT)].mean() / 1000,
165                               df.datetime.mean())
166
167          Bn = Bn.squeeze()
168          Be = Be.squeeze()
169          Bd = -Bu.squeeze()
170
171          df['IGRF_X'] = Bn
172          df['IGRF_Y'] = Be
173          df['IGRF_Z'] = Bd
174          df['IGRF_F'] = la.norm([Bn, Be, Bd], axis=0)
175
176      else:
177          df['IGRF_X'] = ''
178          df['IGRF_Y'] = ''
179          df['IGRF_Z'] = ''
180          df['IGRF_F'] = ''
181
182      for i in range(0, len(df['datetime']), chunk):
183          start = i
184          stop = i + chunk
185
186          Be, Bn, Bu = igrf(df.LONG.iloc[start:stop],
187                             df.LAT.iloc[start:stop],
188                             df.ALT.iloc[start:stop] / 1000,
189                             df.datetime.iloc[start:stop]))
190
191          Bn = np.diagonal(Bn) # Must use np.diagonal because passing multiple locations to ppigrf is for calculating

```

```

    IGRF values over a grid (provides more values than we want)
192     Be = np.diagonal( Be)
193     Bd = np.diagonal(-Bu)
194
195     df.IGRF_X.iloc[start:stop] = Bn
196     df.IGRF_Y.iloc[start:stop] = Be
197     df.IGRF_Z.iloc[start:stop] = Bd
198     df.IGRF_F.iloc[start:stop] = la.norm([Bn, Be, Bd], axis=0)
199
200     return df
201
202 def angle2dcm(angles:           np.ndarray,
203                 angle_unit:      str='degrees',
204                 NED_to_body:     bool=True,
205                 rotation_sequence: int=321) -> np.ndarray:
206     """
207         Convert euler angles to direction cosine matrix (DCM)
208
209     Parameters
210     -----
211     angles
212         Nx2 array of euler angles (can be in rad or deg) -> [roll, pitch, yaw]
213     angle_unit
214         'degrees' if angles are in degrees, 'radians' if angles are in radians
215     NED_to_body
216         Set to True if rotation is North-East-Down (NED) to body frame
217     rotation_sequence
218         321 for standard ZYX rotation sequence
219
220     Returns
221     -----
222     dcm
223         Direction cosine matrix that corresponds to the given euler angles
224     """
225
226     if len(angles.shape) == 1:
227         angles = angles.reshape(1, 3)
228
229     num_angles = angles.shape[0]
230
231     if angle_unit.lower() == 'degrees':
232         roll = np.radians(angles[:, 0])
233         pitch = np.radians(angles[:, 1])
234         yaw = np.radians(angles[:, 2])
235     else:
236         roll = angles[:, 0]
237         pitch = angles[:, 1]
238         yaw = angles[:, 2]
239
240     # For a single angle, DCM R1 would be:
241     # R1 = np.array([[1, 0, 0],
242     #                 [0, cos(roll), sin(roll)],
243     #                 [0, -sin(roll), cos(roll)]])
244

```

```

245     R1 = np.zeros((num_angles, 3, 3))
246     R1[:, 0, 0] = 1
247     R1[:, 1, 1] = cos(roll)
248     R1[:, 1, 2] = sin(roll)
249     R1[:, 2, 1] = -sin(roll)
250     R1[:, 2, 2] = cos(roll)
251
252     # For a single angle, DCM R2 would be:
253     # R2 = np.array([[cos(pitch), 0, -sin(pitch)],
254     #                 [0, 1, 0],
255     #                 [sin(pitch), 0, cos(pitch)]])
256
257     R2 = np.zeros((num_angles, 3, 3))
258     R2[:, 0, 0] = cos(pitch)
259     R2[:, 0, 2] = -sin(pitch)
260     R2[:, 1, 1] = 1
261     R2[:, 2, 0] = sin(pitch)
262     R2[:, 2, 2] = cos(pitch)
263
264     # For a single angle, DCM R3 would be:
265     # R3 = np.array([[cos(yaw), sin(yaw), 0],
266     #                 [-sin(yaw), cos(yaw), 0],
267     #                 [0, 0, 1]])
268
269     R3 = np.zeros((num_angles, 3, 3))
270     R3[:, 0, 0] = cos(yaw)
271     R3[:, 0, 1] = sin(yaw)
272     R3[:, 1, 0] = -sin(yaw)
273     R3[:, 1, 1] = cos(yaw)
274     R3[:, 2, 2] = 1
275
276     if rotation_sequence == 321:
277         dcms = R1 @ R2 @ R3
278     elif rotation_sequence == 312:
279         dcms = R2 @ R1 @ R3
280     elif rotation_sequence == 231:
281         dcms = R1 @ R3 @ R2
282     elif rotation_sequence == 213:
283         dcms = R3 @ R1 @ R2
284     elif rotation_sequence == 132:
285         dcms = R2 @ R3 @ R1
286     elif rotation_sequence == 123:
287         dcms = R3 @ R2 @ R1
288     else:
289         dcms = R1 @ R2 @ R3
290
291     if not NED_to_body:
292         return np.transpose(dcms, axes=(0, 2, 1))
293
294     return dcms
295
296 def cal_spin_df(df: pd.DataFrame) -> list:
297     ,
298     Determine distortion matrix and bias vector used

```

```

299      to calibrate the vector magnetometer
300
301      Parameters
302      -----
303      df
304          Dataframe of spin test calibration data
305
306      Returns
307      -----
308      a
309          3x3 distortion matrix
310      b
311          1x3 bias vector
312      ...
313
314      vec    = np.hstack([np.array(df.X)[:, np.newaxis],
315                           np.array(df.Y)[:, np.newaxis],
316                           np.array(df.Z)[:, np.newaxis]])
317      lat    = df.LAT.mean()
318      lon    = df.LONG.mean()
319      height = df.ALT.mean()
320      date   = df.datetime[0]
321      eulers = np.hstack([np.array(df.ROLL)[:, np.newaxis],
322                           np.array(df.PITCH)[:, np.newaxis],
323                           np.array(df.AZIMUTH)[:, np.newaxis]])
324
325      a    = np.eye(3)
326      b    = np.zeros(3)
327      x0  = sc.ab_to_x(a, b)
328
329      vec_true = sc.gen_b_truth_euler(lat    = lat,
330                                      lon    = lon,
331                                      height = height,
332                                      date   = date,
333                                      eulers = eulers)
334
335      _, xf = sc.calibrate_vec(x0           = x0,
336                                b_distorted = vec,
337                                b_true     = vec_true)
338
339      return sc.x_to_ab(xf)
340
341  def cal_tl_df(df: pd.DataFrame,
342                 use_filter: bool,
343                 fstart: float=0,
344                 fstop: float=1,
345                 a: np.ndarray=np.eye(3),
346                 b: np.ndarray=np.zeros(3),
347                 terms: int=t1.ALL_TERMS->np.ndarray:
348                 ...,
349
350      Determine Tolles-Lawson calibration coefficients used
351      to calibrate the scalar magnetometer
352
353      Parameters

```

```

353   -----
354   df
355   a
356   use_filter
357   Use band pass filter if true
358   fstart
359   Start frequency for the band (Hz)
360   fstop
361   Stop frequency for the band (Hz)
362   a
363   3x3 distortion matrix
364   b
365   1x3 bias vector
366   terms
367   Terms to include in A-matrix. Options include:
368
369   - ALL_TERMS
370   - PERMANENT
371   - INDUCED
372   - EDDY
373
374   Returns
375   -----
376   np.ndarray
377   1xK T-L calibration coefficients where K is the
378   number of A matrix terms to use for calibration
379   ,,
380
381   b_scalar = df.F
382   b_vector = np.hstack([np.array(df.X)[:, np.newaxis],
383                         np.array(df.Y)[:, np.newaxis],
384                         np.array(df.Z)[:, np.newaxis]])
385   b_vector = sc.apply_cal_to_vec(vec = b_vector,
386                                 a = a,
387                                 b = b)
388
389   delta_t = np.diff(df.epoch_sec).mean()
390
391   return tl.tlc(b_scalar = b_scalar,
392                  b_vector = b_vector,
393                  delta_t = delta_t,
394                  use_filter = use_filter,
395                  fstart = fstart,
396                  fstop = fstop,
397                  terms = terms)
398
399   def calibrate(survey_df: pd.DataFrame,
400                 a: np.ndarray=np.eye(3),
401                 b: np.ndarray=np.zeros(3),
402                 c: np.ndarray=np.zeros(18),
403                 terms: int=tl.ALL_TERMS,
404                 debug: bool=True):
405   ,,
406   Applies given calibration terms to the scalar sensor data

```

```

407
408     Parameters
409     -----
410     survey_df
411         Dataframe containing flight data from the survey
412         Minimum required columns include:
413
414             - datetime
415             - epoch_sec
416             - LAT
417             - LONG
418             - ALT
419             - F
420             - X
421             - Y
422             - Z
423             - LINE
424             - LINE_TYPE
425
426     a
427         3x3 distortion matrix
428     b
429         1x3 bias vector
430     c
431         1xK T-L calibration coefficients where K is the
432         number of A matrix terms to use for calibration
433         terms
434         Terms to include in A-matrix. Options include:
435
436             - ALL_TERMS
437             - PERMANENT
438             - INDUCED
439             - EDDY
440
441     debug
442         Whether or not debug prints/plot should be generated
443
444     Returns
445     -----
446     calibrated_df
447         Copy of the survey DataFrame, but with calibrated total
448         field values in the 'F' column
449         ...
450
451     timestamps = np.array(survey_df.epoch_sec)
452     b_scalar = np.array(survey_df.F)
453     b_vector = np.hstack([np.array(survey_df.X)[:, np.newaxis],
454                           np.array(survey_df.Y)[:, np.newaxis],
455                           np.array(survey_df.Z)[:, np.newaxis]])
456
457     if debug:
458         print('Applying_spin-test_calibration_to_vector_magnetometer_measurements')
459
460     b_vector = sc.apply_cal_to_vec(vec = b_vector,

```

```

461             a    = a,
462             b    = b)
463     delta_t  = np.diff(timestamps).mean()
464
465     if debug:
466         print('Applying Tolles-Lawson calibration to scalar magnetometer measurements')
467
468     b_body = t1.apply_tlc(c      = c,
469                           b_scalar = b_scalar,
470                           b_vector = b_vector,
471                           delta_t   = delta_t,
472                           terms     = terms)
473
474     b_scalar == b_body
475
476     calibrated_df     = deepcopy(survey_df)
477     calibrated_df[ 'F'] = b_scalar
478
479     return calibrated_df
480
481 def find_anomaly(survey_df: pd.DataFrame,
482                   ref_df: pd.DataFrame,
483                   ref_scale: float=1,
484                   ref_offset: float=0,
485                   enable_lon_norm: bool=False,
486                   filt_cutoff: float=None,
487                   debug: bool=True):
488     """
489     Finds magnetic anomaly values by subtracting out the
490     core field and temporal effects from the *calibrated*
491     scalar magnetometer values
492
493     Parameters
494     -----
495     survey_df
496         Dataframe containing flight data from the survey
497         Minimum required columns include:
498
499         - epoch_sec
500         - LAT
501         - LONG
502         - ALT
503         - F
504
505     ref_df
506         DataFrame containing reference station data coinciding
507         with the flight
508     ref_scale
509         Amount to scale the reference station data by
510     ref_offset
511         Amount to bias the reference station data by (nT)
512     enable_lon_norm
513         Whether or not to conduct frequency based longitudinal
514         normalization to provided reference station data

```

```

515     filt_cutoff
516         Low pass filter cutoff frequency for scalar data. Set to None
517         to prevent filtering
518     debug
519         Whether or not debug prints/plot should be generated
520
521     Returns
522     -----
523     anomaly_df
524         Copy of the survey DataFrame, but with anomaly values in
525         the 'F' column
526     ...
527
528     timestamps = np.array(survey_df.epoch_sec)
529     lats      = np.array(survey_df.LAT)
530     lons      = np.array(survey_df.LONG)
531     heights   = np.array(survey_df.ALT)
532     b_scalar  = np.array(survey_df.F)
533
534     if debug:
535         print('Removing core field from measurements')
536
537     try:
538         b_scalar == survey_df.IGRF_F
539
540     except:
541         Be, Bn, Bu = igrf(lons,
542                            lats,
543                            heights / 1000,
544                            survey_df.datetime[0])
545
546         IGRF = np.hstack((Bn.squeeze()[:, np.newaxis],
547                           Be.squeeze()[:, np.newaxis],
548                           -Bu.squeeze()[:, np.newaxis]))
549         IGRF_F = la.norm(IGRF, axis=1)
550
551         b_scalar == IGRF_F
552
553     if ref_df is not None:
554         if debug:
555             print('Removing diurnal/space/weather effects with reference station data')
556
557         if enable_lon_norm:
558             survey_lon = lons.mean()
559         else:
560             survey_lon = None
561
562         _, ref_mag = Diurnal.interp_reference_df(df          = ref_df,
563                                                 timestamps = timestamps,
564                                                 survey_lon = survey_lon,
565                                                 ref_scale  = ref_scale,
566                                                 ref_offset = ref_offset,
567                                                 subtract_core = True)
568         b_scalar == ref_mag

```

```

569
570     if filt_cutoff is not None:
571         if debug:
572             print('Low_pass_filtering_scalar_anomaly_measurements_w/{}Hz_cutoff'.format(filt_cutoff))
573
574     filt_fs = 1.0 / np.diff(survey_df.epoch_sec).mean() # Find avg sample frequency
575     b_scalar = filt.lpf(data = b_scalar,
576                          cutoff = filt_cutoff,
577                          fs      = filt_fs)
578
579     anomaly_df = deepcopy(survey_df)
580     anomaly_df[ 'F' ] = b_scalar
581
582     return anomaly_df
583
584 def meshgrid_from_lines(survey_df: pd.DataFrame,
585                         dx:          float,
586                         dy:          float,
587                         buffer:       float,
588                         line_type:    int=1) -> list:
589     """
590     Creates a latitude/longitude meshgrid of coordinates based
591     on the max/min coordinates of the given line data (either
592     flight or tie lines)
593
594     Parameters
595     -----
596     survey_df
597         Dataframe containing flight data from the survey
598         Minimum required columns include:
599
600         - LAT
601         - LONG
602         - LINE_TYPE
603
604     dx
605         Pixel distance in the E/W direction (m)
606     dy
607         Pixel distance in the N/S direction (m)
608     buffer
609         The amount of extra distance to include around the edges of
610         the min/max line coordinates (m)
611     line_type
612         Set to 1 to make a meshgrid of flight line data and set
613         to 2 for a meshgrid of tie line data
614
615     Returns
616     -----
617     list of np.ndarrays
618         Two NxD meshgridded latitude and longitude arrays -> [latitude (dd) grid, longitude (dd) grid]
619     ...
620
621     mask = (survey_df.LINE_TYPE == line_type)
622

```

```

623     lats = np.array(survey_df.LAT[mask], dtype=np.float64)
624     lons = np.array(survey_df.LONG[mask], dtype=np.float64)
625
626     min_lat = lats.min()
627     min_lon = lons.min()
628     max_lat = lats.max()
629     max_lon = lons.max()
630     center_lat = (min_lat + max_lat) / 2
631     center_lon = (min_lon + max_lon) / 2
632
633     # Create a meshgrid of pixel coordinates to interpolate data at plus add min(h AGL)/2 buffer on edges
634     __, min_lon = cu.coord_coord(lat = center_lat,
635                               lon = min_lon,
636                               dist = buffer / 1000,
637                               bearing = 270)
638     __, max_lon = cu.coord_coord(lat = center_lat,
639                               lon = max_lon,
640                               dist = buffer / 1000,
641                               bearing = 90)
642     min_lat, __ = cu.coord_coord(lat = min_lat,
643                               lon = center_lon,
644                               dist = buffer / 1000,
645                               bearing = 180)
646     max_lat, __ = cu.coord_coord(lat = max_lat,
647                               lon = center_lon,
648                               dist = buffer / 1000,
649                               bearing = 0)
650
651     dist_x = cu.coord_dist(lat_1 = center_lat,
652                             lon_1 = min_lon,
653                             lat_2 = center_lat,
654                             lon_2 = max_lon) * 1000
655     dist_y = cu.coord_dist(lat_1 = min_lat,
656                             lon_1 = center_lon,
657                             lat_2 = max_lat,
658                             lon_2 = center_lon) * 1000
659     num_lon = int(dist_x / dx)
660     num_lat = int(dist_y / dy)
661
662     interp_lons = np.linspace(min_lon, max_lon, num_lon)
663     interp_lats = np.linspace(min_lat, max_lat, num_lat)
664
665     return np.meshgrid(interp_lons, interp_lats)
666
667 def normalize_alts(anomaly_df: pd.DataFrame,
668                     dx: float,
669                     dy: float,
670                     buffer: float=0) -> pd.DataFrame:
671     """
672     Normalizes the flight and tie line samples (if given)
673     to a constant altitude. The altitude of the highest
674     sample within the flight/tie lines is the new constant
675     altitude for all samples
676

```

```

677     Parameters
678     -----
679     anomaly_df
680         Dataframe containing flight data from the survey
681         Minimum required columns include:
682             - LAT
683             - LONG
684             - ALT
685             - F (anomaly values)
686             - LINE_TYPE
687
688     dx
689         Pixel distance in the E/W direction (m)
690     dy
691         Pixel distance in the N/S direction (m)
692     buffer
693         The amount of extra distance to include around the edges of
694         the min/max line coordinates (m)
695     line_type
696         Set to 1 to make a meshgrid of flight line data and set
697         to 2 for a meshgrid of tie line data
698
699     Returns
700     -----
701
702     norm_alt_anom_df
703         Copy of the anomaly DataFrame, but with altitude-normalized
704         anomaly values in the 'F' column, plus a new constant
705         altitude in the 'ALT' column
706         ...
707
708     # Create a DataFrame copy to be returned at the end
709     norm_alt_anom_df = deepcopy(anomaly_df)
710
711     #####
712     # Mask out flight line data
713     #####
714     fl_mask = (anomaly_df.LINE_TYPE == 1)
715
716     fl_scalar = np.array(anomaly_df.F[fl_mask], dtype=np.float64)
717     fl_lats = np.array(anomaly_df.LAT[fl_mask], dtype=np.float64)
718     fl_lons = np.array(anomaly_df.LONG[fl_mask], dtype=np.float64)
719     fl_heights = np.array(anomaly_df.ALT[fl_mask], dtype=np.float64)
720
721     fl_fit_points = np.hstack([fl_lons[:, np.newaxis],
722                               fl_lats[:, np.newaxis]])
723
724     max_fl_height = fl_heights.max()
725     cartesian_height = max_fl_height
726
727     grid_fl_lon, grid_fl_lat = meshgrid_from_lines(survey_df = anomaly_df,
728                                                    dx = dx,
729                                                    dy = dy,
730                                                    buffer = buffer,

```

```

731                         line_type = 1)
732
733     # Interpolate pixel altitudes
734     interp_fl_heights = interpolate.griddata(fl_fit_points,
735                                              fl_heights,
736                                              (grid_fl_lon, grid_fl_lat),
737                                              method='linear')
738
739     # Clamp pixel heights
740     interp_fl_heights(interp_fl_heights < fl_heights.min()) = fl_heights.min()
741     interp_fl_heights(interp_fl_heights > fl_heights.max()) = fl_heights.max()
742
743     #####
744     # Mask out tie line data (if available)
745     #####
746     tl_mask = (anomaly_df.LINE_TYPE == 2)
747
748     if len(anomaly_df[tl_mask]) > 0:
749         tl_scalar = np.array(anomaly_df.F[tl_mask], dtype=np.float64)
750         tl_lats = np.array(anomaly_df.LAT[tl_mask], dtype=np.float64)
751         tl_lons = np.array(anomaly_df.LONG[tl_mask], dtype=np.float64)
752         tl_heights = np.array(anomaly_df.ALT[tl_mask], dtype=np.float64)
753
754         tl_fit_points = np.hstack([tl_lons[:, np.newaxis],
755                                    tl_lats[:, np.newaxis]])
756
757         max_tl_height = tl_heights.max()
758
759         if max_tl_height > cartesian_height:
760             cartesian_height = max_tl_height
761
762             grid_tl_lon, grid_tl_lat = meshgrid_from_lines(survey_df = anomaly_df,
763                                                dx = dx,
764                                                dy = dy,
765                                                buffer = buffer,
766                                                line_type = 1)
767
768             # Interpolate pixel altitudes
769             interp_tl_heights = interpolate.griddata(tl_fit_points,
770                                              tl_heights,
771                                              (grid_tl_lon, grid_tl_lat),
772                                              method='linear')
773
774             # Clamp pixel heights
775             interp_tl_heights(interp_tl_heights < tl_heights.min()) = tl_heights.min()
776             interp_tl_heights(interp_tl_heights > tl_heights.max()) = tl_heights.max()
777
778             #####
779             # Altitude-normalize flight line samples
780             #####
781             # Interpolate pixel values
782             interp_fl_scalar = interpolate.griddata(fl_fit_points,
783                                              fl_scalar,
784                                              (grid_fl_lon, grid_fl_lat),

```

```

785                         method='cubic')
786
787     # Altitude-normalize interpolated flight line data
788     cartesian_fl_scalar = mu.drape2lvl(drape_map)      = interp_fl_scalar,
789                           drape_heights   = interp_fl_heights,
790                           delta_x        = dx,
791                           delta_y        = dy,
792                           cartesian_height = cartesian_height)
793
794     # Remove NaNs in input data so that interpolate.griddata doesn't return all NaNs
795     flat_cartesian_fl_scalar = cartesian_fl_scalar.flatten()
796     no_nan_fl_mask          = ~np.isnan(flat_cartesian_fl_scalar)
797     no_nan_flat_cartesian_fl_scalar = flat_cartesian_fl_scalar[no_nan_fl_mask]
798
799     flat_cartesian_fl_points = np.hstack([grid_fl_lon.flatten()[:, np.newaxis],
800                                            grid_fl_lat.flatten()[:, np.newaxis]])
801     no_nan_flat_cartesian_fl_points = flat_cartesian_fl_points[no_nan_fl_mask, :]
802
803
804     # Resample altitude-normalized flight line data where the flight line samples are located
805     norm_alt_fl_scalar = interpolate.griddata(no_nan_flat_cartesian_fl_points,
806                                                no_nan_flat_cartesian_fl_scalar,
807                                                fl_fit_points,
808                                                method='nearest')
809
810     # Save the altitude-normalized flight line samples
811     norm_alt_anom_df['F'].loc[fl_mask] = norm_alt_fl_scalar
812
813 ######
814     # Altitude-normalize tie line samples (if available)
815 ######
816     if len(anomaly_df[tl_mask]) > 0:
817         # Interpolate pixel altitudes
818         interp_tl_heights = interpolate.griddata(tl_fit_points,
819                                                 tl_heights,
820                                                 (grid_tl_lon, grid_tl_lat),
821                                                 method='linear')
822
823         # Interpolate pixel values
824         interp_tl_scalar = interpolate.griddata(tl_fit_points,
825                                                 tl_scalar,
826                                                 (grid_tl_lon, grid_tl_lat),
827                                                 method='cubic')
828
829         # Altitude-normalize interpolated tie line data
830         cartesian_tl_scalar = mu.drape2lvl(drape_map)      = interp_tl_scalar,
831                           drape_heights   = interp_tl_heights,
832                           delta_x        = dx,
833                           delta_y        = dy,
834                           cartesian_height = cartesian_height)
835
836         # Remove NaNs in input data so that interpolate.griddata doesn't return all NaNs
837         flat_cartesian_tl_scalar = cartesian_tl_scalar.flatten()
838         no_nan_tl_mask          = ~np.isnan(flat_cartesian_tl_scalar)

```

```

839     no_nan_flat_cartesian_tl_scalar = flat_cartesian_tl_scalar[no_nan_tl_mask]
840
841     flat_cartesian_tl_points      = np.hstack([grid_tl_lon.flatten()[:, np.newaxis],
842                                              grid_tl_lat.flatten()[:, np.newaxis]])
843     no_nan_flat_cartesian_tl_points = flat_cartesian_tl_points[no_nan_tl_mask, :]
844
845     # Resample altitude-normalized tie line data where the tie line samples are located
846     norm_alt_tl_scalar = interpolate.griddata(no_nan_flat_cartesian_tl_points,
847                                                no_nan_flat_cartesian_tl_scalar,
848                                                tl_fit_points,
849                                                method='nearest')
850
851     # Save the altitude-normalized tie line samples
852     norm_alt_anom_df['F'].loc[tl_mask] = norm_alt_tl_scalar
853
854     # Save the normalized altitude values for all samples
855     norm_alt_anom_df['ALT'] = cartesian_height
856
857     return norm_alt_anom_df
858
859 def lvl_flight_lines(anomaly_df: pd.DataFrame,
860                      lvl_type: str=None,
861                      num_ptls: int=5,
862                      ptl_locs: np.ndarray=None,
863                      percent_thresh: float=0.85) -> pd.DataFrame:
864     """
865         Applies the given flight line leveling technique and
866         returns a DataFrame of the leveled data
867
868     Parameters
869     -----
870     anomaly_df
871         Dataframe containing flight data from the survey
872         Minimum required columns include:
873
874             - LAT
875             - LONG
876             - F (anomaly values)
877             - LINE
878             - LINE_TYPE
879
880     lvl_type
881         Specify map leveling approach, set to None to disable. Options
882         include:
883
884             - 'pca'
885             - 'tie'
886
887     num_ptls
888         Only used if lvl_type is set to 'pca'.
889         Number of pseudo tie lines to use for leveling
890     ptl_locs
891         Only used if lvl_type is set to 'pca'.
892         Kx1 array of relative locations of the pseudo tie lines.

```

```

893     Each relative location is a percent distance from the
894     edge of the survey area where the first sample of the
895     first flight line was taken. For example, in order to
896     set two pseudo tie lines at opposite ends of the dataset,
897     set ptl_locs = [0.0, 1.0]. Values must be between 0 and 1
898     percent_thresh
899         Only used if lvl_type is set to 'pca'.
900         Value ranging from 0 to 1 (not inclusive) that
901         specifies the minimum cumulative contribution
902         rate of the components to use for the PCA
903         reconstruction
904
905     Returns
906     -----
907     pd.DataFrame
908         Copy of anomaly DataFrame, but with leveled flight line
909         anomaly values in the 'F' column
910         ...
911
912     assert lvl_type.lower() in ['pca', 'tie'], 'Unknown_map_leveling_type specified'
913
914     if lvl_type.lower() == 'pca':
915         return pcaLvl.pca_lvl(survey_df      = anomaly_df,
916                               num_ptls      = num_ptls,
917                               ptl_locs      = ptl_locs,
918                               percent_thresh = percent_thresh)
919
920     elif lvl_type.lower() == 'tie':
921         return tieLvl.tie_lvl(survey_df = anomaly_df,
922                               approach    = 'lobf') # Hard code Line Of Best Fit (lobf) per individual flight line
923                                         approach
924
925     def interp_flight_lines(anomaly_df: pd.DataFrame,
926                            dx: float,
927                            dy: float,
928                            max_terrain_msl: float,
929                            buffer: float=0,
930                            sensor_sigma: float=0,
931                            interp_type: str='bicubic',
932                            neighbors: int=None,
933                            skip_na_mask: bool=False,
934                            debug: bool=True) -> dict:
935
936         Interpolate flight line data into a grid array for easy
937         map creation
938
939         Parameters
940         -----
941         anomaly_df
942             Dataframe containing flight data from the survey
943             Minimum required columns include:
944             - LAT (dd)
945             - LONG (dd)

```

```

946      - F      (anomaly values in nT)
947      - ALT    (m above MSL)
948      - LINE_TYPE
949
950      dx
951          Pixel distance in the E/W direction (m)
952      dy
953          Pixel distance in the N/S direction (m)
954      max_terrain_msl
955          Maximum altitude above MSL of the survey terrain
956          across all survey points. This parameter is
957          mainly used to determine the distance at which
958          map pixels are too far away from all survey
959          samples and therefore need to be converted
960          to NaN values
961      buffer
962          The amount of extra distance to include around the edges of
963          the min/max line coordinates (m)
964      sensor_sigma
965          Scalar magnetometer noise variance (nT)
966      interp_type
967          Specify interpolator to use for final map value interpolation.
968          Options include:
969
970          - 'bicubic': Bicubic spline interpolator
971          - 'rbf':     Radial basis function interpolator
972          - 'gpr':    Gaussian process regression
973
974      neighbors
975          Only used for RBF interpolation: If specified, the value of the
976          interpolant at each evaluation point will be computed using only
977          this many nearest data points. All the data points are used by default
978      skip_na_mask
979          Prevent the NaN masking of pixels that are considered too far
980          from the survey data (saves computation time for large datasets)
981      debug
982          Whether or not debug prints/plot should be generated
983
984      Returns
985  -----
986      interp_dict
987          Dictionary of meshgridded/interpolated coordinate, scalar anomaly,
988          altitude, and standard deviation values. Columns include:
989
990          - LAT
991          - LONG
992          - F
993          - ALT
994          - STD
995      ...
996
997      fl_mask = (anomaly_df.LINE_TYPE == 1)
998
999      fl_scalar = np.array(anomaly_df.F[fl_mask],      dtype=np.float64)

```

```

1000     fl_lats    = np.array(anomaly_df.LAT[fl_mask],   dtype=np.float64)
1001     fl_lons    = np.array(anomaly_df.LONG[fl_mask],  dtype=np.float64)
1002     fl_heights = np.array(anomaly_df.ALT[fl_mask],   dtype=np.float64)
1003
1004     fl_fit_points = np.hstack([fl_lons[:, np.newaxis],
1005                                fl_lats[:, np.newaxis]])
1006
1007     grid_lon, grid_lat = meshgrid_from_lines(survey_df = anomaly_df,
1008                                                dx      = dx,
1009                                                dy      = dy,
1010                                                buffer  = buffer,
1011                                                line_type = 1)
1012     grid_coords = np.hstack([grid_lon.flatten()[:, np.newaxis],
1013                               grid_lat.flatten()[:, np.newaxis]])
1014
1015     interp_shape = grid_lon.shape
1016
1017     dist_thresh_m = fl_heights.min() - max_terrain_msl
1018
1019     if debug:
1020         print('Interpolating survey anomaly data to map coordinates')
1021
1022     if interp_type.lower() == 'bicubic':
1023         if debug:
1024             print('Running bicubic spline interpolation for all map pixels')
1025
1026         interp_scalar = interpolate.griddata(fl_fit_points,
1027                                              fl_scalar,
1028                                              (grid_lon, grid_lat),
1029                                              method='cubic')
1030
1031         interp_std = np.ones(interp_shape) * np.nan
1032
1033     elif interp_type.lower() == 'rbf':
1034         if debug:
1035             print('Running radial basis function (RBF) interpolation for all map pixels')
1036
1037         rbfScalar = interpolate.RBFInterpolator(fl_fit_points,
1038                                               fl_scalar,
1039                                               neighbors=neighbors,
1040                                               kernel='linear',
1041                                               smoothing=0)
1042
1043         interp_scalar = rbfScalar(grid_coords).reshape(interp_shape)
1044         interp_std    = np.ones(interp_shape) * np.nan
1045
1046     elif interp_type.lower() == 'gpr':
1047         if debug:
1048             print('Running Gaussian Process Regression (GPR) interpolation for all map pixels')
1049
1050             # Find average distance between samples (m)
1051             unique_fl_nums = np.unique(anomaly_df.LINE[anomaly_df.LINE_TYPE == 1])
1052             avg_samp_dists = np.zeros(len(unique_fl_nums))
1053
1054             for i, line in enumerate(unique_fl_nums):
1055                 mask = np.where(anomaly_df.LINE == line)[0]

```

```

1054
1055     lats = np.array(anomaly_df.LAT[mask])
1056     lons = np.array(anomaly_df.LONG[mask])
1057
1058     line_len = cu.coord_dist(lat_1 = lats[0],
1059                               lon_1 = lons[0],
1060                               lat_2 = lats[-1],
1061                               lon_2 = lons[-1]) * 1000
1062     avg_samp_dists[i] = line_len / len(lats)
1063
1064     avg_samp_dist = avg_samp_dists.mean()
1065
1066     # GPR interpolate
1067     kernel = WNK(sensor_sigma) + RBF([dist_thresh_m, avg_samp_dist])
1068     gpr = GPR(kernel=kernel)
1069
1070     gpr.fit(fit_points, scalar)
1071
1072     interp_points = np.hstack([grid_lon.flatten()[:, np.newaxis],
1073                                grid_lat.flatten()[:, np.newaxis]])
1074
1075     interp_scalar, interp_std = gpr.predict(interp_points, return_std=True)
1076
1077     interp_scalar = interp_scalar.reshape(interp_shape)
1078     interp_std = interp_std.reshape(interp_shape)
1079
1080     if not skip_na_mask:
1081         if debug:
1082             print('NaN\'ing-out pixels that are at least min(h_AGL)/2 dist away from all sample locations')
1083
1084     angle_thresh = cu.arc_angle(dist_thresh_m / 1000)
1085
1086     fl_grid_lon = fl_lons.flatten()[:, np.newaxis]
1087     fl_grid_lat = fl_lats.flatten()[:, np.newaxis]
1088     fl_grid_coords = np.hstack([fl_grid_lon, fl_grid_lat])
1089
1090     flat_grid_lon = grid_lon.flatten()[:, np.newaxis]
1091     flat_grid_lat = grid_lat.flatten()[:, np.newaxis]
1092     flat_grid_coords = np.hstack([flat_grid_lon, flat_grid_lat])
1093
1094     for grid_coord in tqdm(flat_grid_coords):
1095         min_angle = distance.cdist(flat_grid_coords,
1096                                     [grid_coord],
1097                                     'euclidean').min()
1098
1099         if min_angle > angle_thresh:
1100             nan_mask = np.logical_and(grid_lon == grid_coord[0],
1101                                       grid_lat == grid_coord[1])
1102
1103             interp_scalar[nan_mask] = np.nan
1104
1105             if interp_std is not None:
1106                 interp_std[nan_mask] = np.nan
1107

```

```

1108     interp_heights = np.ones(interp_shape) * fl_heights.mean() # Interpolate pixel heights?
1109
1110     interp_dict = {'LAT': grid_lat,
1111                 'LONG': grid_lon,
1112                 'F': interp_scalar,
1113                 'ALT': interp_heights,
1114                 'STD': interp_std}
1115
1116     return interp_dict
1117
1118 def gen_map(out_dir: str,
1119             map_name: str,
1120             survey_df: pd.DataFrame,
1121             ref_df: pd.DataFrame,
1122             dx: float,
1123             dy: float,
1124             max_terrain_msl: float=0,
1125             ref_scale: float=1,
1126             ref_offset: float=0,
1127             enable_lon_norm: bool=False,
1128             a: np.ndarray=np.eye(3),
1129             b: np.ndarray=np.zeros(3),
1130             c: np.ndarray=np.zeros(18),
1131             terms: int=t1.ALL_TERMS,
1132             enable_alt_norm: bool=False,
1133             lvl_type: str=None,
1134             num_ptls: int=5,
1135             ptl_locs: np.ndarray=None,
1136             percent_thresh: float=0.85,
1137             sensor_sigma: float=0.0,
1138             interp_type: str='bicubic',
1139             filt_cutoff: float=None,
1140             debug: bool=True) -> rxr.rioxarray.raster_dataset.xarray.DataArray:
1141     ...
1142     Generate a map based on the provided sensor calibration
1143     parameters, reference station data (if given), and flight survey
1144     data
1145
1146     Parameters
1147     -----
1148     out_dir
1149         Path to directory where the map (as a GeoTIFF) will
1150         be exported to
1151     map_name
1152         Description of survey area or other identifier
1153     survey_df
1154         Dataframe containing flight data from the survey
1155         Minimum required columns include:
1156
1157         - datetime
1158         - epoch_sec
1159         - LAT
1160         - LONG
1161         - ALT

```

```

1162      - F (total field)
1163      - X
1164      - Y
1165      - Z
1166      - LINE
1167      - LINE_TYPE
1168
1169      ref_df
1170      DataFrame containing reference station data coinciding
1171      with the flight
1172      dx
1173      Pixel distance in the E/W direction (m)
1174      dy
1175      Pixel distance in the N/S direction (m)
1176      max_terrain_msl
1177      Maximum altitude above MSL of the survey terrain
1178      across all survey points. This parameter is
1179      mainly used to determine the distance at which
1180      map pixels are too far away from all survey
1181      samples and therefore need to be converted
1182      to NaN values
1183      ref_scale
1184      Amount to scale the reference station data by
1185      ref_offset
1186      Amount to bias the reference station data by (nT)
1187      enable_lon_norm
1188      Whether or not to conduct frequency based longitudinal
1189      normalization to provided reference station data
1190      a
1191      3x3 distortion matrix
1192      b
1193      1x3 bias vector
1194      c
1195      lxK T-L calibration coefficients where K is the
1196      number of A matrix terms to use for calibration
1197      terms
1198      Terms to include in A-matrix. Options include:
1199
1200      - ALL_TERMS
1201      - PERMANENT
1202      - INDUCED
1203      - EDDY
1204
1205      enable_alt_norm
1206      Whether or not to normalize survey data to a constant
1207      altitude. Useful for creating a level map from a drape
1208      survey or a survey with high altitude variance
1209      lvl_type
1210      Specify map leveling approach, set to None to disable. Options
1211      include:
1212
1213      - 'pca'
1214      - 'tie' (Currently unsupported)
1215

```

```

1216     num_ptls
1217         Only used if lvl_type is set to 'pca'.
1218         Number of pseudo tie lines to use for leveling
1219     ptl_locs
1220         Only used if lvl_type is set to 'pca'.
1221         Kx1 array of relative locations of the pseudo tie lines.
1222         Each relative location is a percent distance from the
1223         edge of the survey area where the first sample of the
1224         first flight line was taken. For example, in order to
1225         set two pseudo tie lines at opposite ends of the dataset,
1226         set ptl_locs = [0.0, 1.0]. Values must be between 0 and 1
1227         percent_thresh
1228             Only used if lvl_type is set to 'pca'.
1229             Value ranging from 0 to 1 (not inclusive) that
1230             specifies the minimum cumulative contribution
1231             rate of the components to use for the PCA
1232             reconstruction
1233     sensor_sigma
1234         Scalar magnetometer noise variance (nT)
1235     interp_type
1236         Specify interpolator to use for final map value interpolation.
1237         Options include:
1238             - 'bicubic': Bicubic spline interpolator
1239             - 'rbf': Radial basis function interpolator
1240             - 'gpr': Gaussian process regression
1241
1242     filt_cutoff
1243         Low pass filter cutoff frequency for scalar data. Set to None
1244         to prevent filtering
1245     debug
1246         Whether or not debug prints/plot should be generated
1247
1248     Returns
1249     -----
1250
1251     rxr.rioxarray.raster_dataset.xarray.DataArray
1252         Generated map as a rioxarray
1253         ,,
1254
1255     interp_type = interp_type.lower()
1256
1257     if debug:
1258         print('Generating_map_from_survey_data')
1259
1260     # Calibrate only if the calibration terms are not all 0
1261     if ((~(a == 0).all()) or (~(b == 0).all()) or (~(c == 0).all())):
1262         if debug:
1263             print('Calibrating_survey_data')
1264
1265     calibrated_df = calibrate(survey_df = survey_df,
1266                               a = a,
1267                               b = b,
1268                               c = c,
1269                               terms = terms,

```

```

1270                 debug      = debug)
1271         else:
1272             calibrated_df = survey_df
1273
1274     if debug:
1275         print('Finding_survey_anomaly_scalar_values')
1276
1277     anomaly_df = find_anomaly(survey_df      = calibrated_df,
1278                                ref_df       = ref_df,
1279                                ref_scale   = ref_scale,
1280                                ref_offset  = ref_offset,
1281                                enable_lon_norm = enable_lon_norm,
1282                                filt_cutoff = filt_cutoff,
1283                                debug       = debug)
1284
1285
1286     if enable_alt_norm:
1287         if debug:
1288             print('Altitude-normalizing_survey_anomaly_scalar_values')
1289
1290         alt_norm_df = normalize_alts(anomaly_df = anomaly_df,
1291                                       dx          = dx,
1292                                       dy          = dy)
1293     else:
1294         alt_norm_df = anomaly_df
1295
1296     if lvl_type is not None:
1297         if debug:
1298             print('Leveling_flight_line_anomaly_scalar_values_using_{}_{method}'.format(lvl_type))
1299
1300     lvl_df = lvl_flight_lines(anomaly_df      = alt_norm_df,
1301                               lvl_type       = lvl_type,
1302                               num_ptls      = num_ptls,
1303                               ptl_locs      = ptl_locs,
1304                               percent_thresh = percent_thresh)
1305
1306     else:
1307         lvl_df = alt_norm_df
1308
1309     if debug:
1310         print('Interpolating_anomaly_scalar_values')
1311
1312     interp_df = interp_flight_lines(anomaly_df      = lvl_df,
1313                                     dx            = dx,
1314                                     dy            = dy,
1315                                     max_terrain_msl = max_terrain_msl,
1316                                     buffer        = (lvl_df.ALT.min() - max_terrain_msl) / 2.0,
1317                                     sensor_sigma  = sensor_sigma,
1318                                     interp_type   = interp_type,
1319                                     debug         = debug)
1320
1321     interp_lats  = interp_df['LAT']
1322     interp_lons  = interp_df['LONG']
1323     interp_scalar = interp_df['F']
1324     interp_heights = interp_df['ALT']

```

```

1324     interp_std      = interp_df['STD']
1325
1326     if debug:
1327         print('Applying a final low pass filter to the interpolated anomaly scalar values')
1328
1329     filt_interp_scalar = filt.lpf2(data    = interp_scalar,
1330                                     cutoff = max_terrain_msl - interp_heights.min(),
1331                                     dx     = dx,
1332                                     dy     = dy)
1333
1334     if debug:
1335         print('Exporting map as a GeoTIFF')
1336
1337     return mu.export_map(out_dir = out_dir,
1338                           location = map_name,
1339                           date    = survey_df.datetime[0],
1340                           lats    = interp_lats,
1341                           lons    = interp_lons,
1342                           scalar   = filt_interp_scalar,
1343                           heights  = interp_heights,
1344                           stds    = interp_std,
1345                           vector   = None)

```

### mammal/MAMMAL/Utils/mapUtils.py

```

1 import sys
2 import datetime as dt
3 from copy import deepcopy
4 from os.path import dirname, join, exists, realpath
5 from typing import Union
6
7 import matplotlib.pyplot as plt
8 import numpy as np
9 import pandas as pd
10 import rioxarray as rxr
11 import xarray as xr
12 from GeoScraper import OSM_URL_Wizard, GeoScraper, uri_validator
13 from numpy import fft
14 from osgeo import gdal
15 from osgeo import osr
16 from scipy.interpolate import interp1d
17 from scipy.spatial.transform import Rotation as R
18 from simplekml import Kml, Color
19 from ppigrf import igrf
20
21 sys.path.append(dirname(realpath(__file__)))
22 sys.path.append(dirname(dirname(realpath(__file__))))
23
24 import coordinateUtils as cu
25 import Filters
26
27
28 WGS84_EPSG = "EPSG:4326" # The EPSG for WGS-84 lat/lon is 4326

```

```

29
30 # Enumerate band numbers
31 SCALAR = 0
32 VEC_X = 1
33 VEC_Y = 2
34 VEC_Z = 3
35 ALT = 4
36 STD = 5
37
38
39 def plt_freqs(map: rxr.rioxarray.raster_dataset.xarray.DataArray,
40             map_name: str='',
41             mag_thresh: float=le-2):
42     """
43     """
44
45
46     map_utm = map.rio.reproject(map.rio.estimate_utm_crs())
47     map_utm.data = np.nan_to_num(num_to_nan(map_utm).data)
48
49     map_fft = fft.fft2(np.nan_to_num(np.squeeze(map_utm.data)))
50
51     kx = fft.freq(n=len(map_utm.x), d=np.diff(map_utm.x).mean()) # Cycles/m
52     ky = fft.freq(n=len(map_utm.y), d=np.diff(map_utm.y).mean()) # Cycles/m
53
54     _, (ax1, ax2) = plt.subplots(1, 2)
55     ew_freqs = np.mean(abs(fft.fftshift(map_fft)), axis=0)
56     shift_kx = fft.fftshift(kx)
57     ax1.set_title('{}_{Map_E/W/Marginal_FFT}'.format(map_name))
58     ax1.set_xlabel('Wavenumber_(cycles/m)')
59     ax1.set_ylabel('Magnitude_(nT)')
60     ax1.plot(shift_kx, ew_freqs, label='E/W_Map_Freq_Content')
61     ax1.legend()
62     ax1.grid()
63
64     ns_freqs = np.mean(abs(fft.fftshift(map_fft)), axis=1)
65     shift_ky = fft.fftshift(ky)
66     ax2.set_title('{}_{Map_N/S/Marginal_FFT}'.format(map_name))
67     ax2.set_xlabel('Wavenumber_(cycles/m)')
68     ax2.set_ylabel('Magnitude_(nT)')
69     ax2.plot(shift_ky, ns_freqs, label='N/S_Map_Freq_Content')
70     ax2.legend()
71     ax2.grid()
72
73 def ned2body(ned_vecs: np.ndarray,
74             eulers: np.ndarray,
75             rot_seq: str='zyx',
76             degrees: bool=True) -> np.ndarray:
77     """
78     Rotate vectors from the NED to sensor body frame
79     using an array of euler angles
80
81     Parameters
82     -----

```

```

83     ned_vecs
84         Nx3 array of vectors in the NED frame to be
85         rotated -> [North, East, Down] <- 1st vector
86             [. . .]
87             [. . .]
88             [. . .] <- Nth vector
89
90     eulers
91         Nx3 array of sensor pitch, roll, yaw euler angles
92         (rotates sensor frame – _not measurement vector_ – from NED to body)
93         where N is the number of vectors
94         to be rotated -> [roll, pitch, yaw] <- 1st vector
95             [. . .]
96             [. . .]
97             [. . .] <- Nth vector
98
99     rot_seq
100        Rotation sequence for the euler angles
101        degrees
102        Whether the euler angles are in degrees or not
103
104    Returns
105    -----
106    np.ndarray
107        Nx3 array of the original vectors rotated
108        into sensor's body frame -> [x, y, z] <- 1st vector
109            [. . .]
110            [. . .]
111            [. . .] <- Nth vector
112
113        where N is the number of vectors
114        ,
115
116        flpd_angs = np.flip(-eulers, axis=1) # Negate angles because we want to rotate NED vectors into body frame and flip
117        angle order cause scipy rotations are stupid
118        dcms = R.from_euler(rot_seq, flpd_angs, degrees=degrees).as_matrix()
119
120        return np.array([dcms[i] @ vec for i, vec in enumerate(ned_vecs)])
121
122    def num_to_nan(map: np.ndarray,
123                  nan_lims: list=[-1e6, 1e6]) -> np.ndarray:
124        ,
125
126        Set all values of a map's data array that exceed the values
127        given in 'nan_lims' to NaNs
128
129        Parameters
130        -----
131        map
132            MxN magnetic map values
133        nan_lims
134            Min and max value thresholds for num to NaN
135            conversion -> [min, max]
136
137        Returns
138        -----
139        map
140            MxN magnetic map values with applied NaN mask

```

```

136      """
137
138      map.data[np.logical_or(map.data < nan_lims[0],
139                             map.data > nan_lims[1])] = np.nan
140
141      return map
142
143  def stack_bands(bands: list) -> rxr.rioxarray.raster_dataset.xarray.DataArray:
144      """
145          Create a combined rioxarray with bands as listed in 'bands'
146
147          Parameters
148          -----
149          bands
150              List of rioxarrays whos data/bands are to be combined. Note
151              that the coordinates of each rioxarray MUST match for this
152              to work
153
154          Returns
155          -----
156          rxr.rioxarray.raster_dataset.xarray.DataArray
157              Combined rioxarray
158              ...
159
160          return xr.concat(bands, dim='band')
161
162  def igrf_WGS84(map_WGS84: rxr.rioxarray.raster_dataset.xarray.DataArray,
163                 map_alt_m: float,
164                 survey_date: Union[dt.date, dt.datetime]) -> np.ndarray:
165      """
166          Create a rioxarray object that holds both scalar and vector IGRF
167          values that correspond to a given WGS-84 anomaly survey. The
168          coordinates will be in lat/lon (dd), and the band list is as
169          follows:
170
171          - Band 0 -> x/North IGRF Component Values (nT)
172          - Band 1 -> y/East IGRF Component Values (nT)
173          - Band 2 -> z/Dowm IGRF Component Values (nT)
174          - Band 3 -> f/Magnitude IGRF Values (nT)
175
176          Parameters
177          -----
178          map_WGS84
179              Anomaly map rioxarray with WGS-84 coordinates
180          map_alt_m
181              Altitude MSL survey was collected (m)
182          survey_date
183              Date survey was collected
184
185          Returns
186          -----
187          igrf_map_WGS84
188              4 Band rioxarray of vector and scalar IGRF map
189              with WGS-84 coordinates

```

```

190      , ,
191
192      igrf_Be, igrf_Bn, igrf_Bu = igrf(np.array(map_WGS84.x),
193                                         np.array(map_WGS84.y)[:, np.newaxis],
194                                         map_alt_m / 1000,
195                                         survey_date)
196
197      igrf_mag = np.sqrt(igrf_Be**2 + igrf_Bn**2 + igrf_Bu**2)
198
199      if map_WGS84.data.shape[0] == 1:
200          igrf_Bn_map = deepcopy(map_WGS84)
201          igrf_Bn_map.data = igrf_Bn
202
203          igrf_Be_map = deepcopy(map_WGS84)
204          igrf_Be_map.data = igrf_Be
205
206          igrf_Bd_map = deepcopy(map_WGS84)
207          igrf_Bd_map.data = -igrf_Bu
208
209          igrf_mag_map = deepcopy(map_WGS84)
210          igrf_mag_map.data = igrf_mag
211
212          igrf_map_WGS84 = stack_bands([igrf_Bn_map,
213                                         igrf_Be_map,
214                                         igrf_Bd_map,
215                                         igrf_mag_map])
216
217      else:
218          igrf_Bn_map = deepcopy(map_WGS84[0])
219          igrf_Bn_map.data = igrf_Bn.squeeze()
220
221          igrf_Be_map = deepcopy(map_WGS84[0])
222          igrf_Be_map.data = igrf_Be.squeeze()
223
224          igrf_Bd_map = deepcopy(map_WGS84[0])
225          igrf_Bd_map.data = -igrf_Bu.squeeze()
226
227          igrf_mag_map = deepcopy(map_WGS84[0])
228          igrf_mag_map.data = igrf_mag.squeeze()
229
230          igrf_map_WGS84 = stack_bands([igrf_mag_map,
231                                         igrf_Bn_map,
232                                         igrf_Be_map,
233                                         igrf_Bd_map])
234
235      return igrf_map_WGS84
236
237  def upcontinue(map: np.ndarray,
238                 delta_x: float,
239                 delta_y: float,
240                 delta_h: float) -> np.ndarray:
241      ,
242
243      This function upward continues a scalar magnetic anomaly map
244      by an altitude different delta_h.

```

```

244
245     Parameters
246     -----
247     map
248         MnN array of scalar magnetic anomaly map values
249     delta_x
250         Size of pixels in x-dimension (m)
251     delta_y
252         Size of pixels in y-dimension (m)
253     delta_h
254         Number of meters (MSL) to upward continue the original
255         map by
256
257     Returns
258     -----
259     upmap
260         MnN array of upward continued scalar anomaly map values
261     ,
262
263     upmap = deepcopy(map)
264     mean = upmap[~np.isnan(upmap)].mean()
265     upmap -= mean # Remove DC bias
266
267     kx2d, ky2d = Filters.compute_wavevectors(delta_x,
268                                         delta_y,
269                                         upmap.shape[1],
270                                         upmap.shape[0])
271
272     fftMap = np.fft.fft2(np.nan_to_num(upmap))
273     scaledMap = fftMap * (np.e**(-delta_h * np.sqrt(kx2d**2 + ky2d**2)))
274     upmap = np.real(np.fft.ifft2(scaledMap))
275
276     upmap[map == np.nan] = np.nan # Keep original NaNs
277     upmap += mean # Keep DC bias
278
279     return upmap
280
281 def drape2lvl(drape_map: np.ndarray,
282                 drape_heights: np.ndarray,
283                 delta_x: float,
284                 delta_y: float,
285                 cartesian_height: float) -> np.ndarray:
286     ,
287
288     This function takes a draped scalar magnetic anomaly map and
289     converts it to a cartesian grid.
290
291     Parameters
292     -----
293     drape_map
294         MnN array of draped scalar magnetic map values (nT)
295     drape_heights
296         MnN array of pixel drape heights MSL (m)
297     delta_x
298         Size of pixels in x-dimension (m)

```

```

298     delta_y
299         Size of pixels in y-dimension (m)
300     cartesian_height
301         Height of output grid – must be greater than max(drape_height) MSL (m)
302
303     Returns
304     -----
305     cartesian_grid
306         MxN array of level scalar magnetic anomaly map values (nT)
307         ,
308
309     maxHeight = cartesian_height - np.min(drape_heights[~np.isnan(drape_heights)])
310     minHeight = cartesian_height - np.max(drape_heights[~np.isnan(drape_heights)])
311
312     topLim = upcontinue(drape_map,
313                           delta_x,
314                           delta_y,
315                           maxHeight)
316     topHeights = drape_heights + maxHeight
317
318     bottomLim = upcontinue(drape_map,
319                           delta_x,
320                           delta_y,
321                           minHeight)
322     bottomHeights = drape_heights + minHeight
323
324     # Flatten all arrays and then turn them into column vectors
325     topLimFlat      = topLim.flatten()[:, np.newaxis]
326     topHeightsFlat   = topHeights.flatten()[:, np.newaxis]
327     bottomLimFlat    = bottomLim.flatten()[:, np.newaxis]
328     bottomHeightsFlat = bottomHeights.flatten()[:, np.newaxis]
329
330     # Horizontally tack the upward continued map readings
331     lims      = np.hstack([bottomLimFlat, topLimFlat])
332     heights   = np.hstack([bottomHeightsFlat, topHeightsFlat])
333
334     cartesian_grid = np.array([np.interp(cartesian_height, heights[i, :], lims[i, :]) for i in range(lims.shape[0])])
335
336     return cartesian_grid.reshape(drape_map.shape)
337
338 def sample_map(map: rxr.rioxarray.raster_dataset.xarray.DataArray,
339                 x: float,
340                 y: float,
341                 band: int=SCALAR) -> Union[float, np.ndarray]:
342     ,
343     Evaluate map at a given coordinate using
344     the 2D cubic interpolation method. Set either
345     x or y to None to get all map values in that
346     direction (i.e. y=200 and x=None will return
347     an array with all map values in the row where
348     the Northing=200n)
349
350     Parameters
351     -----

```

```

352     map
353         DataArray of the magnetic map
354     x
355         x location of the point on
356         the map to estimate/interpolate in (m)
357     y
358         y location of the point on
359         the map to estimate/interpolate (m)
360     band
361         Index of the band to interpolate
362
363     Returns
364     -----
365     float / np.array:
366         Estimated/interpolated magnetic
367         field strength(s) of the map at the
368         given coordinate and band (nT)
369     ,
370
371     if x is None and y is not None: # Grab row of data
372         return map[band].interp(y=y, method='cubic').data.squeeze()
373
374     elif x is not None and y is None: # Grab row of data
375         return map[band].interp(x=x, method='cubic').data.squeeze()
376
377     elif x is not None and y is not None: # Grab single sample
378         return map[band].interp(x=x, y=y, method='cubic').data.item()
379
380 def find_corrugation_fom(map: rxr.rioxarray.raster_dataset.xarray.DataArray,
381                         cutoffs: list[float],
382                         dp: float,
383                         band: int=SCALAR,
384                         axis: int=1) -> float:
385     ,
386
387     Estimate the severity of corrugation present in a given map
388     by producing a FOM (Figure of Merit) based on
389
390     Parameters
391     -----
392     map
393         DataArray of the magnetic map
394     cutoffs
395         List of wavelength cutoffs in the tie line
396         direction -> [lowest wavelength (m), highest wavelength (m)]
397     dp
398         Pixel length in the tie line direction (m)
399     band
400         Band of the map DataArray of which to calculate the FOM
401     axis
402         Array axis in the tie line direction (i.e. if the map is
403         North up and the tie lines are in the East direction,
404         'axis' should be set to 1)
405
406     Returns

```

```

406      -----
407      corrugation_fom
408          A figure of merit where the larger the value, the more corrugation is
409          present in the given map
410          ...
411
412      map_cpy = deepcopy(map)
413      data    = map_cpy[band].data
414
415      high_cutoff = 1 / cutoffs[0]
416      low_cutoff = 1 / cutoffs[1]
417
418      if axis == 1:
419          rev_axis = 0
420      else:
421          rev_axis = 1
422
423      data_zero_mean = data - data.mean()
424      data_hpf      = Filters.hpf(data_zero_mean, low_cutoff, dp, 10, axis)
425      hpf_zero_mean = data_hpf - data_hpf.mean()
426      hpf_compressed = np.mean(hpf_zero_mean, axis=rev_axis)
427
428      hpf_compressed_zero_mean = hpf_compressed - hpf_compressed.mean()
429      hpf_compressed_fft       = np.fft.fft(np.nan_to_num(hpf_compressed_zero_mean))
430
431      fft_mag    = np.abs(hpf_compressed_fft)
432      n         = hpf_compressed_zero_mean.size
433      fft_mag   = fft_mag[:int(n/2)]
434      fft_freqs = np.fft.freq(n, d=dp)[:int(n/2)] # Only use positive freqs
435      f         = interp1d(fft_freqs, fft_mag)
436
437      corrugation_fom = f(np.linspace(low_cutoff, high_cutoff, 100)).max()
438
439      return corrugation_fom
440
441  def add_kml_flight_path(kml: Kml,
442                           flight_path: np.ndarray=None) -> Kml:
443      ...
444
445      Credits:
446          https://simplekml.readthedocs.io/en/latest/geometries.html#gxtrack
447
448      Using the given flight path, add a gxtrack to the given Kml object
449
450      Parameters
451      -----
452      kml
453          Kml object
454      flight_path
455          Nx4 array of timestamped survey sample
456          geolocations -> [latitude (dd), longitude (dd), altitude above MSL (m), UTC timestamp (s)]
457
458      Returns
459      -----
460      kml_cpy

```

```

460      An updated Kml object with the flightpath added (if given)
461      ,
462
463      kml_cpy = deepcopy(kml)
464
465      if flight_path is not None:
466          lats      = flight_path[:, 0]
467          lons      = flight_path[:, 1]
468          alts      = flight_path[:, 2]
469          timestamps = flight_path[:, 3]
470          datetimes = [dt.datetime.fromtimestamp(stamp).strftime('%Y-%m-%d %H:%M:%S.%fZ') for stamp in timestamps]
471
472          when = list(datetimes)
473          coord = list(zip(lons,
474                            lats,
475                            alts))
476
477          trk = kml_cpy.newgxtrack(name='Flight_Path')
478
479          trk.newwhen(when)
480          trk.newgxcoord(coord)
481
482          trk.altitudemode = 'absolute'
483
484          trk.stylemap.normalstyle.iconstyle.icon.href    = 'http://earth.google.com/images/kml-icons/track-directional/
485                                         track-0.png'
486          trk.stylemap.normalstyle.linestyle.color        = '99ffac59'
487          trk.stylemap.normalstyle.linestyle.width       = 6
488          trk.stylemap.highlightstyle.iconstyle.icon.href = 'http://earth.google.com/images/kml-icons/track-directional/
489                                         track-0.png'
490          trk.stylemap.highlightstyle.iconstyle.scale     = 1.2
491          trk.stylemap.highlightstyle.linestyle.color     = '99ffac59'
492          trk.stylemap.highlightstyle.linestyle.width     = 8
493
494      return kml_cpy
495
496
497  def add_kml_areas(kml: Kml,
498                      area_polys: list[dict]=None) -> Kml:
499
500      """
501      Add the sub-survey area polygons to the given Kml object
502
503      Parameters
504      -----
505      kml
506          Kml object
507      area_polys
508          List of dicts where each dict corresponds to
509          a single sub-survey area with the following
510          contents:
511
512          - NAME: str
513              - Name of sub-survey area
514          - FL_DIR: float
515              - Direction (degrees) of flight
516                  lines in the area (must be
517                  within the range [0, 180])

```

```

512      - FL_DIST: float           - Distance (m) between flight lines
513      - TL_DIR: float          - Direction (degrees) of tie
514                           lines in the area (must be
515                           within the range [0, 180)). Set
516                           to -90 if tie lines not flown in
517                           the area
518      - TL_DIST: float          - Distance (m) between tie lines.
519                           Set to 0 if tie lines not flown in
520                           the area
521      - LAT:      list[float] - Boundary latitudes (dd)
522      - LONG:     list[float] - Boundary longitudes (dd)
523      - ALT:      list[float] - Boundary altitudes (m) above MSL
524
525      Default value of None will cause a single polygon
526      generated with boundary points at the map extent
527      with the average altitude of the pixel points. Line
528      directions will be set to -90 and line distances
529      will be set to 0
530
531      Returns
532      -----
533      kml_cpy
534      An updated Kml object with the sub-survey area polygons added (if given)
535      ...
536
537      kml_cpy = deepcopy(kml)
538
539      if area_polys is not None:
540          fol = kml_cpy.newfolder(name='SubSurveyAreas')
541
542          for area in area_polys:
543              name      = area[ 'NAME' ]
544              fl_dir    = area[ 'FL_DIR' ] % 180.0 # [0, 180]
545              fl_dist   = area[ 'FL_DIST' ]
546              tl_dir    = area[ 'TL_DIR' ] % 180.0 # [0, 180]
547              tl_dist   = area[ 'TL_DIST' ]
548              lats      = area[ 'LAT' ]
549              lons      = area[ 'LONG' ]
550              alts      = area[ 'ALT' ]
551              avg_alts = np.array(alts).mean()
552
553              if np.isnan(fl_dir):
554                  fl_dir = -90
555
556              if np.isnan(tl_dir):
557                  tl_dir = -90
558
559              coords = list(zip(lons,
560                               lats,
561                               alts))
562
563              pol = fol.newpolygon(name=name)
564
565              pol.outerboundaryis = coords

```

```

566     pol.altitudemode = 'absolute'
567     pol.description = 'FLDir:{}°,FLDist:{}m,TLDir:{}°,TLDist:{}m,Alt:{}m,aboveMSL'.format(
568         fl_dir,
569                                     fl_dist
570                                     ,
571                                     tl_dir
572                                     ,
573                                     tl_dist
574                                     ,
575                                     avg_alts
576                                     )
577
578     pol.style.linestyle.color = Color.green
579     pol.style.linestyle.width = 5
580     pol.style.polystyle.color = Color.changealphaint(100, Color.green)
581
582     return kml_cpy
583
584
585     def add_kml_features(kml: Kml,
586                           osm_path: str=None) -> Kml:
587         ,
588
589         Using OpenSourceMap (OSM), add road/power line linestrings and
590         substation polygons within the survey extent to the given
591         Kml object
592
593         Parameters
594
595         kml
596             Kml object
597         osm_path
598             Either path to an OpenStreetMap *.osm XML file or valid
599             OpenStreetMap API query URL. Devault value of None
600             will cause the function to query the OpenStreetMap API
601             with a bbox set to the map extent
602
603         Returns
604
605         kml_cpy
606             An updated Kml object with the road/power line
607             linestrings and substation polygons added (if valid OSM
608             path given)
609             ,
610
611         kml_cpy = deepcopy(kml)
612
613         if osm_path is not None:
614             scraper = GeoScraper()
615
616             if uri_validator(osm_path):
617                 scraper.from_url(osm_path)
618             else:

```

```

611     scraper.from_file(osm_path)
612
613     kml_roads      = kml_cpy.newmultigeometry(name='Roads')
614     kml_power_lines = kml_cpy.newmultigeometry(name='PowerLines')
615     kml_sub_stations = kml_cpy.newmultigeometry(name='Substations')
616
617     for road in scraper.highways():
618         kml_roads.newlinestring(coords=road['coords'])
619
620     for power in scraper.power_locations():
621         if 'line' in power['tag_vals']:
622             kml_power_lines.newlinestring(coords=power['coords'])
623
624         elif 'substation' in power['tag_vals']:
625             kml_sub_stations.newpolygon(outerboundaryis=power['coords'])
626
627     kml_roads.style.linestyle.color      = Color.black
628     kml_roads.style.linestyle.width     = 2
629     kml_power_lines.style.linestyle.color = Color.yellow
630     kml_power_lines.style.linestyle.width = 2
631     kml_sub_stations.style.polystyle.color = Color.orange
632
633     return kml_cpy
634
635 def read_map_metadata(map_fname: str) -> dict:
636     """
637     Read map GeoTIFF metadata fields and return as a dictionary with
638     the following top-level keys:
639
640     - AREA_OR_POINT
641     - CSV
642     - Description
643     - ExtentDD
644     - FinalFiltCut
645     - FinalFiltOrder
646     - InterpType
647     - KML
648     - LevelType
649     - POC
650     - SampleDistM
651     - ScalarSensorVar
652     - ScalarType
653     - SurveyDateUTC
654     - TLCoeffs
655     - TLCoeffTypes
656     - VectorSensorVar
657     - VectorType
658     - xResolutionM
659     - yResolutionM
660     - Band_1
661     - Band_2
662     - Band_3
663     - Band_4
664     - Band_5

```

```

665      — Band_6
666      — Band_7
667      — Band_8
668
669      Parameters
670      -----
671      map_fname
672          File path/name to the map GeoTIFF
673
674      Returns
675      -----
676      metadata
677          Dictionary containing all metadata fields of the
678          given map GeoTIFF
679          ...
680
681      gdal.UseExceptions()
682      map = gdal.Open(map_fname)
683
684      metadata = map.GetMetadata()
685
686      metadata[ 'Band_1' ] = map.GetRasterBand(1).GetMetadata()
687      metadata[ 'Band_2' ] = map.GetRasterBand(2).GetMetadata()
688      metadata[ 'Band_3' ] = map.GetRasterBand(3).GetMetadata()
689      metadata[ 'Band_4' ] = map.GetRasterBand(4).GetMetadata()
690      metadata[ 'Band_5' ] = map.GetRasterBand(5).GetMetadata()
691      metadata[ 'Band_6' ] = map.GetRasterBand(6).GetMetadata()
692      metadata[ 'Band_7' ] = map.GetRasterBand(7).GetMetadata()
693      metadata[ 'Band_8' ] = map.GetRasterBand(8).GetMetadata()
694
695      return metadata
696
697  def export_map(out_dir: str,
698                 location: str,
699                 date: Union[dt.date, dt.datetime],
700                 lats: np.ndarray,
701                 lons: np.ndarray,
702                 scalar: np.ndarray,
703                 heights: Union[np.ndarray, float],
704                 process_df: pd.DataFrame,
705                 process_app: str,
706                 stds: np.ndarray=None,
707                 vector: np.ndarray=None,
708                 scalar_type: str='Not_Given',
709                 vector_type: str='Not_Given',
710                 scalar_var: float=np.nan,
711                 vector_var: float=np.nan,
712                 poc: str='Not_Given',
713                 flight_path: np.ndarray=None,
714                 area_polys: list[dict]=None,
715                 osm_path: str=None,
716                 level_type: str='Not_Given',
717                 tl_coeff_types: list=[] ,
718                 tl_coeffs: np.ndarray=np.array([]) ,

```

```

719     interp_type:      str='Not_Given',
720     final_filt_cut:   float=0,
721     final_filt_order: int=1) -> rxr.rioxarray.raster_dataset.xarray.DataArray:
722     ,
723     Credits:
724     https://stackoverflow.com/a/33950009/9860973
725
726     Exports survey data to a multi-band GeoTIFF file in WGS-84
727     coordinates. Bands include:
728
729     - Band 0: Scalar anomaly values (nT)
730     - Band 1: x/North anomaly vector values (nT)
731     - Band 2: y/East anomaly vector values (nT)
732     - Band 3: z/Down anomaly vector values (nT)
733     - Band 4: Pixel height values (m)
734     - Band 5: Pixel scalar standard deviation values
735     - Band 6: Pixel scalar x/North gradient values (nT)
736     - Band 7: Pixel scalar y/East gradient values (nT)
737
738     **NOTE**: rioxarray can't read GeoTIFF metadata correctly. In
739     order to read and parse both the top-level and band-level
740     metadata, you must use gdal like so::
741
742     >>> from pprint import pprint
743     >>> from osgeo import gdal
744     >>> gdal.UseExceptions()
745     >>> map = gdal.Open('map.tif') # Read in map
746     >>> pprint(map.GetMetadata()) # Print top-level metadata
747     >>> pprint(map.GetRasterBand(1).GetMetadata()) # Print first band metadata
748
749     Parameters
750     -----
751     out_dir
752         Path to directory where the GeoTIFF will be exported to
753         location
754         Description of survey area
755         date
756             Date of when the survey was collected (UTC)
757         lats
758             1xM array of pixel latitudes (dd)
759         lons
760             1xN array of pixel longitudes (dd)
761         scalar
762             MnN scalar magnetic anomaly values (nT)
763         heights
764             MnN array of heights (if map is drape) or float
765             map height MSL (m)
766         process_df
767             Pandas DataFrame of all pertinent survey data points and
768             data processing steps. Minimum required columns include:
769
770             - TIMESTAMP: UTC timestamps (s)
771             - LAT:      Latitudes (dd)
772             - LONG:    Longitudes (dd)

```

```

773      - ALT:          Altitudes above MSL (m)
774      - DC_X:         Direction cosine X-Components (n/a)
775      - DC_Y:         Direction cosine Y-Components (n/a)
776      - DC_Z:         Direction cosine Z-Components (n/a)
777      - F:            Raw scalar measurements (nT)
778      - F_CAL:        Calibrated scalar measurements (nT)
779      - F_CAL_IGRF:   Calibrated scalar measurements without
780          core field (nT)
781      - F_CAL_IGRF_TEMPORAL: Calibrated scalar measurements
782          without core field or temporal
783          corrections (nT)
784      - F_CAL_IGRF_TEMPORAL_FILT: Calibrated scalar measurements
785          without core field or temporal
786          corrections after low pass
787          filtering (nT)
788      - F_CAL_IGRF_TEMPORAL_FILT_LEVEL: Calibrated scalar
789          measurements without
790          core field or temporal
791          corrections after low
792          pass filtering and map
793          leveling (nT)
794
795      process_app
796          String describing the application name and version used
797          to generate the map file
798      stds
799          Pixel standard deviation values
800      vector
801          3xMxN array of NED vector magnetic anomaly values
802          (first page is x/North values, etc.) (nT)
803      scalar_type
804          String describing the make/model/type of scalar
805          magnetometer used
806      vector_type
807          String describing the make/model/type of vector
808          magnetometer used
809      scalar_var
810          Scalar magnetometer's assessed noise variance
811      vector_var
812          Vector magnetometer's assessed noise variance
813      poc
814          Point of contact information
815      flight_path
816          Nx4 array of timestamped survey sample
817          geolocations -> [latitude (dd), longitude (dd), altitude above MSL (m), UTC timestamp (s)]
818      area_polys
819          List of dicts where each dict corresponds to
820          a single sub-survey area with the following
821          contents:
822
823      - NAME:    str          - Name of sub-survey area
824      - FL_DIR:  float        - Direction (degrees) of flight
825          lines in the area (must be
826          within the range [0, 180))

```

```

827      - FL_DIST: float           - Distance (m) between flight lines
828      - TL_DIR: float          - Direction (degrees) of tie
829                                lines in the area (must be
830                                within the range [0, 180]). Set
831                                to -90 if tie lines not flown in
832                                the area
833      - TL_DIST: float          - Distance (m) between tie lines.
834                                Set to 0 if tie lines not flown in
835                                the area
836      - LAT:      list[float] - Boundary latitudes (dd)
837      - LONG:     list[float] - Boundary longitudes (dd)
838      - ALT:      list[float] - Boundary altitudes (m) above MSL
839
840      Default value of None will cause a single polygon
841      generated with boundary points at the map extent
842      with the average altitude of the pixel points. Line
843      directions will be set to -90 and line distances
844      will be set to 0
845
846      osm_path
847      Either path to an OpenStreetMap *.osm XML file or valid
848      OpenStreetMap API query URL. Default value of None
849      will cause the function to query the OpenStreetMap API
850      with a bbox set to the map extent and a value of -1 will
851      skip the OpenStreetMap data processing completely
852      level_type
853      String describing how the flight lines were leveled
854      tl_coeff_types
855      List of TL coefficient types used in the same order as
856      listed in ‘tl_coeffs’
857      tl_coeffs
858      1xN array of TL coefficients in teh same order as listed
859      in ‘tl_coeff_types’
860      interp_type
861      String describing algorithm used to interpolate pixel values
862      final_filt_cut
863      Cutoff wavelength of the 2D LPF applied to the interpolated
864      scalar pixel values. The output of this filter are the pixel
865      values in raster band 1
866      final_filt_order
867      Order of the 2D LPF applied to the interpolated
868      scalar pixel values. The output of this filter are the pixel
869      values in raster band 1
870
871      Returns
872      -----
873      rxr.rioxarray.raster_dataset.xarray.DataArray
874      Map as a rioxarray
875      , , ,
876
877      ######
878      # Handle sample distance
879      #####
880      samp_dist = np.nan

```

```

881
882     if flight_path is not None:
883         lats_1 = flight_path[:, 0]
884         lons_1 = flight_path[:, 1]
885
886         lats_2 = np.roll(lats_1, -1)
887         lons_2 = np.roll(lons_1, -1)
888
889         lats_1 = lats_1[1:-1]
890         lons_1 = lons_1[1:-1]
891
892         lats_2 = lats_2[1:-1]
893         lons_2 = lons_2[1:-1]
894
895         samp_dist = cu.coord_dist(lats_1,
896                                     lons_1,
897                                     lats_2,
898                                     lons_2).mean() * 1000 # km to m conversion
899
900 #####
901 # Determine number of bands, image size, and
902 # verify rasters and image size all match
903 #####
904 lats = np.sort(np.unique(lats))
905 lons = np.sort(np.unique(lons))
906
907 num_bands = 8 # 1 scalar, 3 vector, 1 height, 1 scalar std, and 2 scalar gradient bands
908 image_size = (len(lats), len(lons))
909
910 assert scalar.shape == image_size, 'Map_scalar_dimensions_must_match_coordinate_dimensions'
911
912 #####
913 # Handle various input stuff
914 #####
915 if type(heights) == np.ndarray:
916     height = int(heights.mean())
917 else:
918     height = int(heights)
919     heights = np.ones(image_size) * height
920
921 if vector is None:
922     vector = np.ones((3, *image_size)) * np.nan
923
924 if stds is None:
925     stds = np.ones(scalar.shape) * np.nan
926
927 grad_x = np.gradient(scalar, axis=1)
928 grad_y = np.gradient(scalar, axis=0)
929
930 vec_x = vector[0, :, :]
931 vec_y = vector[1, :, :]
932 vec_z = vector[2, :, :]
933 #####
934

```

```

935     # Handle pixel and transform stuff
936     #####
937     ny = image_size[0]
938     nx = image_size[1]
939
940     extent = [min(lons), min(lats), max(lons), max(lats)]
941     xmin, ymin, xmax, ymax = extent
942
943     xres = (xmax - xmin) / float(nx)
944     yres = (ymax - ymin) / float(ny)
945
946     geotransform = (xmin, xres, 0, ymax, 0, -yres)
947
948     #####
949     # Handle file name stuff
950     #####
951     i = 0
952     fname = '{loc}_{height}m_{year}_{month}_{day}_{num}.tiff'.format(loc = location,
953                                         height = height,
954                                         year = date.year,
955                                         month = date.month,
956                                         day = date.day,
957                                         num = i)
958     full_path = join(out_dir, fname)
959
960     while exists(full_path):
961         i += 1
962         fname = '{loc}_{height}m_{year}_{month}_{day}_{num}.tiff'.format(loc = location,
963                                         height = height,
964                                         year = date.year,
965                                         month = date.month,
966                                         day = date.day,
967                                         num = i)
968         full_path = join(out_dir, fname)
969
970     #####
971     # Handle KML stuff
972     #####
973     kml_fname = fname.split('.')[0] + '.kml' # Use same name as GeoTIFF, but .kml extension
974     kml_full_path = join(out_dir, kml_fname)
975
976     kml = Kml(name = kml_fname,
977                open = 1)
978
979     kml = add_kml_flight_path(kml, flight_path)
980
981     if area_polys is None:
982         area_polys = [{NAME: 'Main_Survey_Area',
983                       'FL_DIR': np.nan,
984                       'FL_DIST': 0,
985                       'TL_DIR': np.nan,
986                       'TL_DIST': 0,
987                       'LAT': [ymax, ymax, ymin, ymin],
988                       'LONG': [xmin, xmax, xmax, xmin],
```

```

989             'ALT' :      [height, height, height, height]]]
990
991     kml = add_kml_areas(kml, area_polys)
992
993     if osm_path is None:
994         url_wiz = OSM_URL_Wizard()
995         osm_path = url_wiz.bbox_url(*extent)
996
997     if osm_path != -1:
998         kml = add_kml_features(kml, osm_path)
999
1000    kml_text = kml.kml()
1001    kml.save(kml_full_path)
1002
1003    ######
1004    # Handle CSV stuff
1005    #####
1006    csv_fname     = fname.split('.')[0] + '.csv' # Use same name as GeoTIFF, but .csv extension
1007    csv_full_path = join(out_dir, csv_fname)
1008    process_df.to_csv(csv_full_path, index=False)
1009
1010    #####
1011    # Handle GeoTIFF stuff
1012    #####
1013    dst_ds = gdal.GetDriverByName('GTiff').Create(full_path,
1014                                              nx,
1015                                              ny,
1016                                              num_bands,
1017                                              gdal.GDT_Float64)
1018
1019    # Note that when writing out the bands, we
1020    # have to reverse the data's row order for
1021    # the GeoTIFF to be correct
1022    dst_ds.SetGeoTransform(geotransform) # Specify coords
1023    srs = osr.SpatialReference() # Establish encoding
1024    srs.ImportFromEPSG(4326) # WGS84 lat/long
1025    dst_ds.SetProjection(srs.ExportToWkt()) # Export coords to file
1026
1027    dst_ds.SetMetadata({ 'Description' :      'MagNav_Aeromagnetic_Anomaly_Map', # Set general map metadata fields
1028                         'ProcessingApp' : str(process_app),
1029                         'SurveyDateUTC' : str(date.isoformat()),
1030                         'SampleDistM' :   str(samp_dist),
1031                         'xResolutionM' : str(xres),
1032                         'yResolutionM' : str(yres),
1033                         'ExtentDD' :     str(extent),
1034                         'ScalarType' :   str(scalar_type),
1035                         'VectorType' :   str(vector_type),
1036                         'ScalarSensorVar' : str(scalar_var),
1037                         'VectorSensorVar' : str(vector_var),
1038                         'POC' :          str(poc),
1039                         'KML' :          str(kml_text),
1040                         'LevelType' :    str(level_type),
1041                         'TLCoeffTypes' : str(tl_coeff_types),
1042                         'TLCoeffs' :     str(tl_coeffs),

```

```

1043             'CSV':           str(process_df.to_csv(index=False)),
1044             'InterpType':      str(interp_type),
1045             'FinalFiltCut':   str(final_filt_cut),
1046             'FinalFiltOrder': str(final_filt_order)})

1047
1048     dst_ds.GetRasterBand(1).WriteArray(scalar[:, :-1, :])      # Write scalar anomaly values to the raster (nT)
1049     dst_ds.GetRasterBand(1).SetMetadata({ 'Type':      'F', # Set band-specific metadata
1050                                         'Units':     'nT',
1051                                         'Direction': 'n/a'})}

1052
1053     dst_ds.GetRasterBand(2).WriteArray(vec_x[:, :-1, :])      # Write x/North anomaly vector values to the raster (nT)
1054     dst_ds.GetRasterBand(2).SetMetadata({ 'Type':      'X', # Set band-specific metadata
1055                                         'Units':     'nT',
1056                                         'Direction': 'North'})}

1057
1058     dst_ds.GetRasterBand(3).WriteArray(vec_y[:, :-1, :])      # Write y/East anomaly vector values to the raster (nT)
1059     dst_ds.GetRasterBand(3).SetMetadata({ 'Type':      'Y', # Set band-specific metadata
1060                                         'Units':     'nT',
1061                                         'Direction': 'East'})}

1062
1063     dst_ds.GetRasterBand(4).WriteArray(vec_z[:, :-1, :])      # Write z/Down anomaly vector values to the raster (nT)
1064     dst_ds.GetRasterBand(4).SetMetadata({ 'Type':      'Z', # Set band-specific metadata
1065                                         'Units':     'nT',
1066                                         'Direction': 'Down'})}

1067
1068     dst_ds.GetRasterBand(5).WriteArray(heights[:, :-1, :])    # Write height MSL values to the raster (m)
1069     dst_ds.GetRasterBand(5).SetMetadata({ 'Type':      'ALT', # Set band-specific metadata
1070                                         'Units':     'm_MSL',
1071                                         'Direction': 'n/a'})}

1072
1073     dst_ds.GetRasterBand(6).WriteArray(stds[:, :-1, :])       # Write standard deviation values to the raster (nT)
1074     dst_ds.GetRasterBand(6).SetMetadata({ 'Type':      'STD', # Set band-specific metadata
1075                                         'Units':     'nT',
1076                                         'Direction': 'n/a'})}

1077
1078     dst_ds.GetRasterBand(7).WriteArray(grad_x[:, :-1, :])    # Write scalar x-gradient values to the raster (nT)
1079     dst_ds.GetRasterBand(7).SetMetadata({ 'Type':      'dX', # Set band-specific metadata
1080                                         'Units':     'nT',
1081                                         'Direction': 'East'})}

1082
1083     dst_ds.GetRasterBand(8).WriteArray(grad_y[:, :-1, :])    # Write scalar y-gradient values to the raster (nT)
1084     dst_ds.GetRasterBand(8).SetMetadata({ 'Type':      'dY', # Set band-specific metadata
1085                                         'Units':     'nT',
1086                                         'Direction': 'North'})}

1087
1088     dst_ds.FlushCache() # Write to disk
1089     dst_ds = None
1090
1091     while not exists(full_path):
1092         pass
1093
1094     return rxr.open_rasterio(full_path)

```

mammal/MAMMAL/Utils/Filters.py

```
1  from copy import deepcopy
2
3  import numpy as np
4  from numpy import fft
5  from scipy.signal import butter, sosfilt, sosfiltfilt, sosfilt_zi
6
7
8  def compute_wavevectors(dx: float,
9                         dy: float,
10                        nx: int,
11                        ny: int) -> list:
12    """
13      Compute the wave vectors for a spatial fourier transfrom in
14      Cartesian coordinates
15
16      Parameters
17      -----
18      dx
19          x-pixel size (m)
20      dy
21          y-pixel size (m)
22      nx
23          Number of pixels in x
24      ny
25          Number of pixesl in y
26
27      Returns
28      -----
29      list
30          Spatial wavevectors kx, ky as 2D grids -> [kx, ky]
31    """
32
33    kx = 2 * np.pi * fft.freq(n=nx, d=dx)
34    ky = 2 * np.pi * fft.freq(n=ny, d=dy)
35
36    kx2d, ky2d = np.meshgrid(kx, ky)
37
38    return [kx2d, ky2d]
39
40  def filt(data: np.ndarray,
41            cutoff: float,
42            fs: float,
43            btype: str,
44            order: int=6,
45            axis: int=-1) -> np.ndarray:
46    """
47      Filter an MxN array of data in a single direction
48      without phase shift
49
50      Parameters
51      -----
52      data
```

```

53      MnN array of original data
54      cutoff
55          Filter's cutoff frequency
56      fs
57          Sample frequency of dataset
58      btype
59          Filter type (i.e. 'low' for LPF)
60      order
61          Filter order
62      axis
63          Axis along which to filter
64
65      Returns
66  -----
67      np.ndarray
68          MnN array of filtered data
69      ...
70
71      sos = butter(order,
72                      cutoff,
73                      fs=fs,
74                      btype=btype,
75                      analog=False,
76                      output='sos')
77
78      return sosfiltfilt(sos,
79                            data,
80                            axis=axis)
81
82  def bpf(data: np.ndarray,
83           fstart: float,
84           fstop: float,
85           fs: float,
86           order: int=6,
87           axis: int=-1) -> np.ndarray:
88      ...
89      Band pass filter data array
90
91      Parameters
92  -----
93      data
94          NmM array of data to be filtered
95      fstart
96          Start frequency for the band (Hz)
97      fstop
98          Stop frequency for the band (Hz)
99      fs
100         Data sample frequency (Hz)
101     order
102         Order of the filter
103
104     Returns
105  -----
106     x_filt

```

```

107      Band pass filtered data
108      ,
109
110      return filt(data,
111                  [fstart, fstop],
112                  fs,
113                  'band',
114                  order,
115                  axis)
116
117  def lpf(data: np.ndarray,
118          cutoff: float,
119          fs: float,
120          order: int=6,
121          axis: int=-1) -> np.ndarray:
122      ,
123
124      Low pass filter an MxN array of data in a single direction
125
126      Parameters
127      -----
128      data
129          MxN array of original data
130      cutoff
131          Filter's cutoff frequency
132      fs
133          Sample frequency of dataset
134      order
135          Filter order
136      axis
137          Axis along which to filter
138
139      Returns
140      -----
141      np.ndarray
142          MxN array of filtered data
143
144      return filt(data,
145                  cutoff,
146                  fs,
147                  'low',
148                  order,
149                  axis)
150
151  def lpf2(data: np.ndarray,
152          cutoff: float,
153          dx: float,
154          dy: float,
155          order: float=6) -> np.ndarray:
156      ,
157
158      Low pass filter a 2D array of data
159
160      Parameters
161      -----

```

```

161     data
162         MnN array of original data
163     cutoff
164         Filter's max spacial wavelength
165     dx
166         Difference in x coordinates (i.e. meters
167         per pixel in the x direction)
168     dy
169         Difference in y coordinates (i.e. meters
170         per pixel in the y direction)
171
172     Returns
173     -----
174     data_lpf
175         MnN array of low pass filtered data
176     ,
177
178     data_copy = deepcopy(data)
179
180     # Filter in Y direction
181     data_lpf_y = lpf(data = data_copy,
182                         cutoff = 1/cutoff,
183                         fs      = 1/dy,
184                         order   = order,
185                         axis    = 0)
186
187     # Filter in X direction
188     data_lpf = lpf(data = data_lpf_y,
189                     cutoff = 1/cutoff,
190                     fs      = 1/dx,
191                     order   = order,
192                     axis    = 1)
193
194     return data_lpf
195
196 def hpf(data: np.ndarray,
197          cutoff: float,
198          fs: float,
199          order: int=6,
200          axis: int=-1) -> np.ndarray:
201     ,
202
203     High pass filter an MnN array of data in a single direction
204
205     Parameters
206     -----
207     data
208         MnN array of original data
209     cutoff
210         Filter's cutoff frequency
211     fs
212         Sample frequency of dataset
213     order
214         Filter order
215     axis

```

```

215      Axis along which to filter
216
217      Returns
218      -----
219      np.ndarray
220      MnN array of filtered data
221      ,
222
223      return filt(data,
224          cutoff,
225          fs,
226          'high',
227          order,
228          axis)

```

### mammal/MAMMAL/Utils/coordinateUtils.py

```

1  from typing import Union
2
3  import numpy as np
4  from numpy import degrees, radians, sqrt, sin, cos, arcsin, arccos, arctan2, pi
5  from scipy.spatial import distance
6
7
8  EARTH_RADIUS_KM = 6378.137
9
10
11 def coord_bearing(lat_1: Union[float, np.ndarray],
12                     lon_1: Union[float, np.ndarray],
13                     lat_2: Union[float, np.ndarray],
14                     lon_2: Union[float, np.ndarray]) -> Union[float, np.ndarray]:
15     ,
16
17     Credits:
18         - http://www.movable-type.co.uk/scripts/latlong.html
19         - https://github.com/PowerBroker2/WarThunder/blob/master/WarThunder/mapinfo.py
20
21     Find the bearing (in degrees) between two lat/lon coordinates (dd)
22
23     Parameters
24     -----
25     lat_1
26         First point's latitude (dd)
27     lon_1
28         First point's longitude (dd)
29     lat_2
30         Second point's latitude (dd)
31     lon_2
32         Second point's longitude (dd)
33
34     Returns
35     -----
36     float / np.ndarray
37         Bearing between point 1 and 2 (Degrees)

```

```

37      """
38
39      deltaLon_r = radians(lon_2 - lon_1)
40      lat_1_r     = radians(lat_1)
41      lat_2_r     = radians(lat_2)
42
43      x = cos(lat_2_r) * sin(deltaLon_r)
44      y = cos(lat_1_r) * sin(lat_2_r) - sin(lat_1_r) * cos(lat_2_r) * cos(deltaLon_r)
45
46      return (degrees(arctan2(x, y)) + 360) % 360
47
48  def coord_dist(lat_1: Union[float, np.ndarray],
49                  lon_1: Union[float, np.ndarray],
50                  lat_2: Union[float, np.ndarray],
51                  lon_2: Union[float, np.ndarray]) -> Union[float, np.ndarray]:
52      """
53
54      Credits:
55          - http://www.movable-type.co.uk/scripts/latlong.html
56          - https://github.com/PowerBroker2/WarThunder/blob/master/WarThunder/mapinfo.py
57
58      Find the total distance (in km) between two lat/lon coordinates (dd)
59
60      Parameters
61      -----
62      lat_1
63          First point's latitude (dd)
64      lon_1
65          First point's longitude (dd)
66      lat_2
67          Second point's latitude (dd)
68      lon_2
69          Second point's longitude (dd)
70
71      Returns
72      -----
73      float / np.ndarray
74          Distance between point 1 and 2 (km)
75
76      lat_1_rad = radians(lat_1)
77      lon_1_rad = radians(lon_1)
78      lat_2_rad = radians(lat_2)
79      lon_2_rad = radians(lon_2)
80
81      d_lat = lat_2_rad - lat_1_rad
82      d_lon = lon_2_rad - lon_1_rad
83
84      a = (sin(d_lat / 2) ** 2) + cos(lat_1_rad) * cos(lat_2_rad) * (sin(d_lon / 2) ** 2)
85
86      return 2 * EARTH_RADIUS_KM * arctan2(sqrt(a), sqrt(1 - a))
87
88  def coord_coord(lat:      Union[float, np.ndarray],
89                  lon:      Union[float, np.ndarray],
90                  dist:    Union[float, np.ndarray],

```

```

91         bearing: Union[float, np.ndarray]) -> np.ndarray:
92     """
93     Credits:
94     - http://www.movable-type.co.uk/scripts/latlong.html
95     - https://github.com/PowerBroker2/WarThunder/blob/master/WarThunder/mapinfo.py
96
97     Finds the lat/lon coordinates "dist" km away from the given "lat" and "lon"
98     coordinate along the given compass "bearing"
99
100    Parameters
101    -----
102    lat
103        First point's latitude (dd)
104    lon
105        First point's longitude (dd)
106    dist
107        Distance in km the second point should be from the first point
108    bearing
109        Bearing in degrees from the first point to the second
110
111    Returns
112    -----
113    np.ndarray
114        Latitude and longitude in DD of the second point -> [lat (dd), lon (dd)]
115    ...
116
117    brng = radians(bearing)
118    lat_1 = radians(lat)
119    lon_1 = radians(lon)
120
121    lat_2 = arcsin(sin(lat_1) * cos(dist / EARTH_RADIUS_KM) + cos(lat_1) * sin(dist / EARTH_RADIUS_KM) * cos(brng))
122    lon_2 = lon_1 + arctan2(sin(brng) * sin(dist / EARTH_RADIUS_KM) * cos(lat_1), cos(dist / EARTH_RADIUS_KM) - sin(
123        lat_1) * sin(lat_2))
124
125    try:
126        return np.hstack([degrees(lat_2)[:, np.newaxis], degrees(lon_2)[:, np.newaxis]])
127    except IndexError:
128        return np.array([degrees(lat_2), degrees(lon_2)])
129
130    def coord_intercept(lat_1: Union[float, np.ndarray],
131                         lon_1: Union[float, np.ndarray],
132                         bearing_1: Union[float, np.ndarray],
133                         lat_2: Union[float, np.ndarray],
134                         lon_2: Union[float, np.ndarray],
135                         bearing_2: Union[float, np.ndarray]) -> Union[float, np.ndarray]:
136
137    Credits:
138        - http://www.movable-type.co.uk/scripts/latlong.html
139
140        Given two points and two bearings, find the great circle intersection coordinate
141
142    Parameters
143    -----
144    lat_1

```

```

144      First point's latitude (dd)
145      lon_1
146      First point's longitude (dd)
147      bearing_1
148      First point's bearing to intersection point (degrees)
149      lat_2
150      Second point's latitude (dd)
151      lon_2
152      Second point's longitude (dd)
153      bearing_2
154      Second point's bearing to intersection point (degrees)
155
156      Returns
157  -----
158      np.ndarray
159      Latitude and longitude in DD of the intersection point -> [lat (dd), lon (dd)]
160      ,
161
162      brng_1 = radians(bearing_1)
163      lat_1 = radians(lat_1)
164      lon_1 = radians(lon_1)
165
166      brng_2 = radians(bearing_2)
167      lat_2 = radians(lat_2)
168      lon_2 = radians(lon_2)
169
170      del_lat = lat_2 - lat_1
171      del_lon = lon_2 - lon_1
172
173      del_12 = 2 * arcsin(sqrt(sin(del_lat / 2.0)**2 + (cos(lat_1) * cos(lat_2) * (sin(del_lon / 2.0)**2))))
174      theta_a = arccos((sin(lat_2) - (sin(lat_1) * cos(del_12))) / (sin(del_12) * cos(lat_1)))
175      theta_b = arccos((sin(lat_1) - (sin(lat_2) * cos(del_12))) / (sin(del_12) * cos(lat_2)))
176
177      if sin(del_lon) > 0:
178          theta_12 = theta_a
179          theta_21 = (2 * pi) - theta_b
180
181      else:
182          theta_12 = (2 * pi) - theta_a
183          theta_21 = theta_b
184
185      alpha_1 = brng_1 - theta_12
186      alpha_2 = theta_21 - brng_2
187
188      alpha_3 = arccos((-cos(alpha_1) * cos(alpha_2)) + (sin(alpha_1) * sin(alpha_2) * cos(del_12)))
189      del_13 = arctan2(sin(del_12) * sin(alpha_1) * sin(alpha_2), cos(alpha_2) + (cos(alpha_1) * cos(alpha_3)))
190      lat_3 = arcsin((sin(lat_1) * cos(del_13)) + (cos(lat_1) * sin(del_13) * cos(brng_1)))
191      del_lon_13 = arctan2(sin(brng_1) * sin(del_13) * cos(lat_1), cos(del_13) - (sin(lat_1) * sin(lat_3)))
192      lon_3 = lon_1 + del_lon_13
193
194      try:
195          return np.hstack([degrees(lat_3)[:, np.newaxis], degrees(lon_3)[:, np.newaxis]])
196      except IndexError:
197          return np.array([degrees(lat_3), degrees(lon_3)])

```

```

198
199  def path_intersection(line_1_lats: np.ndarray,
200                  line_1_lons: np.ndarray,
201                  line_2_lats: np.ndarray,
202                  line_2_lons: np.ndarray) -> np.ndarray:
203      """
204      Given two path, find the great circle intersection coordinate
205
206      Parameters
207      -----
208      lat_1
209          First path's latitude (dd)
210      lon_1
211          First path's longitude (dd)
212      lat_2
213          Second path's latitude (dd)
214      lon_2
215          Second path's longitude (dd)
216
217      Returns
218      -----
219      np.ndarray
220          Latitude and longitude in DD of the intersection point -> [lat (dd), lon (dd)]
221      ...
222
223      if type(line_1_lats) is not np.ndarray:
224          line_1_lats = np.array(line_1_lats)
225
226      if type(line_1_lons) is not np.ndarray:
227          line_1_lons = np.array(line_1_lons)
228
229      if type(line_2_lats) is not np.ndarray:
230          line_2_lats = np.array(line_2_lats)
231
232      if type(line_2_lons) is not np.ndarray:
233          line_2_lons = np.array(line_2_lons)
234
235      line_1_coords = np.hstack([np.array(line_1_lons)[:, np.newaxis],
236                                np.array(line_1_lats)[:, np.newaxis]])
237      line_2_coords = np.hstack([np.array(line_2_lons)[:, np.newaxis],
238                                np.array(line_2_lats)[:, np.newaxis]])
239
240      dists = distance.cdist(line_1_coords,
241                            line_2_coords,
242                            'euclidean')
243
244      row, col = np.where(dists == dists.min())
245      row_idx = row[0]
246      col_idx = col[0]
247
248
249      line_1_lat_1 = line_1_lats[row_idx]
250      line_1_lon_1 = line_1_lons[row_idx]
251

```

```

252     try:
253         line_1_lat_2 = line_1_lats[row_idx + 1]
254         line_1_lon_2 = line_1_lons[row_idx + 1]
255     except IndexError:
256         line_1_lat_2 = line_1_lats[row_idx - 1]
257         line_1_lon_2 = line_1_lons[row_idx - 1]
258
259     line_2_lat_1 = line_2_lats[col_idx]
260     line_2_lon_1 = line_2_lons[col_idx]
261
262     try:
263         line_2_lat_2 = line_2_lats[col_idx + 1]
264         line_2_lon_2 = line_2_lons[col_idx + 1]
265     except IndexError:
266         line_2_lat_2 = line_2_lats[col_idx - 1]
267         line_2_lon_2 = line_2_lons[col_idx - 1]
268
269     line_1_bearing_1 = coord_bearing(line_1_lat_1,
270                                     line_1_lon_1,
271                                     line_1_lat_2,
272                                     line_1_lon_2)
273     line_1_bearing_2 = (line_1_bearing_1 + 180) % 360
274
275     line_2_bearing_1 = coord_bearing(line_2_lat_1,
276                                     line_2_lon_1,
277                                     line_2_lat_2,
278                                     line_2_lon_2)
279     line_2_bearing_2 = (line_2_bearing_1 + 180) % 360
280
281     intercept_coord_1 = coord_intercept(line_1_lat_1,
282                                         line_1_lon_1,
283                                         line_1_bearing_1,
284                                         line_2_lat_1,
285                                         line_2_lon_1,
286                                         line_2_bearing_1)
287     intercept_coord_2 = coord_intercept(line_1_lat_1,
288                                         line_1_lon_1,
289                                         line_1_bearing_2,
290                                         line_2_lat_1,
291                                         line_2_lon_1,
292                                         line_2_bearing_2)
293
294     test_dist_1 = distance.cdist([[line_1_lat_1, line_1_lon_1]],
295                                 [intercept_coord_1],
296                                 'euclidean').item()
297     test_dist_2 = distance.cdist([[line_1_lat_1, line_1_lon_1]],
298                                 [intercept_coord_2],
299                                 'euclidean').item()
300
301     if test_dist_1 < test_dist_2:
302         return intercept_coord_1
303     return intercept_coord_2
304
305 def arc_angle(dist: Union[float, np.ndarray]) -> Union[float, np.ndarray]:

```

```

306      """
307      Find Arc angle between two points at the given distance
308      on Earth's surface (or near it)
309
310      Parameters
311      -----
312      dist
313          Arc distance between two points on Earth (km)
314
315      Returns
316      -----
317      float / np.ndarray
318          Arc angle between two points at the given distance (Degrees)
319      ...
320
321      return degrees(dist / EARTH_RADIUS_KM)

```

mammal/MAMMAL/MapLvl/\_\_\_\_init\_\_\_\_.py

```

1 import os, sys
2
3 sys.path.append(os.path.dirname(os.path.realpath(__file__)))
4 sys.path.append(os.path.dirname(os.path.dirname(os.path.realpath(__file__))))

```

mammal/MAMMAL/MapLvl/pcaLvl.py

```

1 import sys
2 from copy import deepcopy
3 from os.path import dirname
4
5 import numpy as np
6 import pandas as pd
7 from scipy import interpolate
8
9 SRC_DIR = dirname(dirname(__file__))
10 sys.path.append(SRC_DIR)
11
12 from Utils import coordinateUtils as cu
13
14
15 def pca_lvl(survey_df: pd.DataFrame,
16             num_ptls: int=5,
17             ptl_locs: np.ndarray=None,
18             percent_thresh: float=0.85) -> pd.DataFrame:
19     ...
20
21     Credits:
22         - Zhang, Q., Peng, C., Lu, Y., Wang, H., & Zhu, K. (2018).
23             Airborne electromagnetic data levelling using principal
24             component analysis based on flight line difference.
25             Journal of Applied Geophysics, 151, -290297.
26             https://doi.org/10.1016/j.jappgeo.2018.02.023

```

```

27      Level survey flight lines using principal component analysis
28      on pseudo tie lines (ptls) of differenced flight line data
29
30      Parameters
31      -----
32      survey_df
33          Dataframe containing magnetic anomaly flight data from the survey
34          Minimum required columns include:
35
36          - LAT
37          - LONG
38          - F (magnetic anomaly scalar values, NOT RAW!)
39          - LINE
40          - LINE_TYPE
41
42      num_ptls
43          Number of pseudo tie lines to use for leveling
44      ptl_locs
45          Kx1 array of relative locations of the pseudo tie lines.
46          Each relative location is a percent distance from the
47          edge of the survey area where the first sample of the
48          first flight line was taken. For example, in order to
49          set two pseudo tie lines at opposite ends of the dataset,
50          set ptl_locs = [0.0, 1.0]. Values must be between 0 and 1
51      percent_thresh
52          Value ranging from 0 to 1 (not inclusive) that
53          specifies the minimum cumulative contribution
54          rate of the components to use for the PCA
55          reconstruction
56
57      Returns
58      -----
59      lvld_survey_df
60          Dataframe containing leveled flight data from the survey
61      ...
62
63      assert num_ptls >= 2, 'Must use at least 2 pseudo tie-lines'
64      assert (ptl_locs is None) or \
65          ((ptl_locs.min() >= 0) and (ptl_locs.max() <= 1)), 'Invalid pseudo tie-line locations detected'
66
67      if ptl_locs is not None:
68          assert num_ptls == len(ptl_locs), 'Number of pseudo tie-lines must equal number of given pseudo tie-line \
69          locations indicies'
70
71      # Calculate coordinate to coordinate azimuths for all data points
72      num_samples = len(survey_df.F)
73      azimuths = np.zeros(num_samples)
74
75      orig_pts = np.hstack([np.array(survey_df.LAT)[:, np.newaxis],
76                            np.array(survey_df.LONG)[:, np.newaxis]])
77      rolled_pts = np.roll(orig_pts, 1, axis=0)
78      azimuths = cu.coord_bearing(orig_pts[:, 0], orig_pts[:, 1], rolled_pts[:, 0], rolled_pts[:, 1])
79
80      # Sample Flight Lines at Pseudo Tie Line Locations Before Leveling

```

```

80     unique_fl_nums = np.unique(survey_df.LINE[survey_df.LINE_TYPE == 1])
81
82     avg_line_azs = np.zeros(len(unique_fl_nums))
83     avg_line_lens = np.zeros(len(unique_fl_nums))
84     avg_samp_dists = np.zeros(len(unique_fl_nums))
85
86     for i, line in enumerate(unique_fl_nums):
87         mask = survey_df.LINE == line
88
89         lats = np.array(survey_df.LAT[mask])
90         lons = np.array(survey_df.LONG[mask])
91         azs = np.array(azimuths[mask])[1:-1]
92
93         avg_line_azs[i] = azs.mean()
94         avg_line_lens[i] = cu.coord_dist(lats[0], lons[0], lats[-1], lons[-1])
95         avg_samp_dists[i] = avg_line_lens[i] / len(lats)
96
97         avg_line_az = (avg_line_azs % 180).mean() # Keep flight line azimuth between 0 to 180 degrees for simplicity
98         avg_line_len = avg_line_lens.mean()
99
100        ptl_locs = (ptl_locs * avg_line_len) - (avg_line_len / 2.0)
101
102        sample_shape = (len(unique_fl_nums), len(ptl_locs))
103
104        sampled_fl_scalar = np.zeros(sample_shape)
105        sampled_fl_lats = np.zeros(sample_shape)
106        sampled_fl_lons = np.zeros(sample_shape)
107
108        for i, loc in enumerate(ptl_locs):
109            for j, line in enumerate(unique_fl_nums):
110                mask = survey_df.LINE == line
111
112                scalar = np.array(survey_df.F[mask], dtype=np.float64)
113                lats = np.array(survey_df.LAT[mask], dtype=np.float64)
114                lons = np.array(survey_df.LONG[mask], dtype=np.float64)
115
116                center_lat = (lats.max() + lats.min()) / 2
117                center_lon = (lons.max() + lons.min()) / 2
118
119                dist = loc
120                az = avg_line_az
121
122                if dist < 0: # Negative distance means 180 degree azimuth shift
123                    dist *= -1
124                    az += 180
125
126                sampled_lat, sampled_lon = cu.coord_coord(center_lat, center_lon, dist, az)
127
128                rbfi_scalar = interpolate.Rbf(lons, # TODO: Pull this out of the loop somehow?
129                                              lats,
130                                              scalar,
131                                              function='linear',
132                                              smooth=0)
133

```

```

134         interp_scalar = rbf1_scalar(sampled_lon, sampled_lat)
135
136         sampled_fl_scalar[j, i] = interp_scalar
137         sampled_fl_lats[j, i] = sampled_lat
138         sampled_fl_lons[j, i] = sampled_lon
139
140     # Find the approximate error between adjacent flight lines
141     fl_diff = np.zeros(sample_shape).T
142     fl_diff[:, 1:] = (sampled_fl_scalar.T - np.roll(sampled_fl_scalar.T, -1))[:, :-1]
143
144     # Create pseudo tie-lines of the flight line difference data
145     ptls = fl_diff
146
147     # Find pseudo tie-line covariance matrix
148     diff_cov = np.cov(ptls)
149
150     # Find SVD of pseudo tie-line covariance matrix
151     R, eigvals, _ = np.linalg.svd(diff_cov)
152
153     # Find minimum required components for PCA reconstruction
154     val_cum_sum = np.cumsum(np.real(np.abs(eigvals))) / np.sum(np.real(np.abs(eigvals))))
155     last_comp_idx = np.where(val_cum_sum >= percent_thresh)[0].min()
156     components = R.T @ ptls
157     reconstruct_comps = components[:, last_comp_idx + 1, :]
158
159     # Calculate pseudo tie-line correction terms
160     ptl_corrs = R[:, :last_comp_idx + 1] @ reconstruct_comps
161     ptl_corrs = np.cumsum(ptl_corrs, axis=1)
162
163     # Interpolate pseudo tie-line correction terms and evaluate flight
164     # line corrections at all flight line sample locations and apply
165     # interpolated corrections to flight line data and return output
166     ptl_lats = sampled_fl_lats
167     ptl_lons = sampled_fl_lons
168
169     anomaly_F = np.array(survey_df.F)
170
171     for i, fl_num in enumerate(unique_fl_nums):
172         mask = survey_df.LINE == fl_num
173         fl_lats = survey_df.loc[mask]['LAT']
174         fl_lons = survey_df.loc[mask]['LONG']
175
176         rbf = interpolate.Rbf(ptl_lons[i, :], ptl_lats[i, :], ptl_corrs.T[i, :], function='linear', smooth=0)
177         interp_ptl_corrs = rbf(fl_lons, fl_lats)
178         anomaly_F[mask] += interp_ptl_corrs
179
180     lvld_survey_df = deepcopy(survey_df)
181     lvld_survey_df['F'] = anomaly_F
182
183
184     return lvld_survey_df

```

mammal/MAMMAL/MapLvl/tieLvl.py

```
1  from copy import deepcopy
2  import sys
3  from os.path import dirname, realpath
4
5  import numpy as np
6  import pandas as pd
7  import scipy.linalg as la
8  import scipy.stats as stats
9  from scipy.spatial import distance
10 from scipy import interpolate
11
12 sys.path.append(dirname(realpath(__file__)))
13 sys.path.append(dirname(dirname(realpath(__file__))))
14
15 from Utils import coordinateUtils as cu
16
17
18 def tie_lvl(survey_df: pd.DataFrame,
19             approach: str='lsq') -> pd.DataFrame:
20     """
21
22     Parameters
23     -----
24
25
26     Returns
27     -----
28
29     """
30
31
32     fl_mask = survey_df.LINE_TYPE == 1
33     unique_fl_nums = np.unique(survey_df.LINE[fl_mask])
34     num_fl = len(unique_fl_nums)
35     fl_scalar = np.array(survey_df.F[fl_mask], dtype=np.float64)
36     fl_lats = np.array(survey_df.LAT[fl_mask], dtype=np.float64)
37     fl_lons = np.array(survey_df.LONG[fl_mask], dtype=np.float64)
38
39     tl_mask = survey_df.LINE_TYPE == 2
40     unique_tl_nums = np.unique(survey_df.LINE[tl_mask])
41     num_tl = len(unique_tl_nums)
42     tl_scalar = np.array(survey_df.F[tl_mask], dtype=np.float64)
43     tl_lats = np.array(survey_df.LAT[tl_mask], dtype=np.float64)
44     tl_lons = np.array(survey_df.LONG[tl_mask], dtype=np.float64)
45
46     # Find intersection coordinates
47     num_ints = num_fl * num_tl
48     int_lats = np.zeros(num_ints)
49     int_lons = np.zeros(num_ints)
50
51     combinations = np.array(np.meshgrid(unique_fl_nums, unique_tl_nums)).T.reshape(-1, 2)
```

```

53     for i, combination in enumerate(combinations):
54         fl, tl = combination
55
56         int_fl_lats = survey_df.LAT[fl_mask & (survey_df.LINE == fl)]
57         int_fl_lons = survey_df.LONG[fl_mask & (survey_df.LINE == fl)]
58
59         int_tl_lats = survey_df.LAT[tl_mask & (survey_df.LINE == tl)]
60         int_tl_lons = survey_df.LONG[tl_mask & (survey_df.LINE == tl)]
61
62         int_lat, int_lon = cu.path_intersection(int_fl_lats,
63                                              int_fl_lons,
64                                              int_tl_lats,
65                                              int_tl_lons)
66
67         int_lats[i] = int_lat
68         int_lons[i] = int_lon
69
70     # Interpolate flight and tie lines at the intersection locations
71     interp_fl = interpolate.Rbf(fl_lons,
72                               fl_lats,
73                               fl_scalar,
74                               function='linear')
75     fl_interp = interp_fl(int_lons, int_lats)
76
77     interp_tl = interpolate.Rbf(tl_lons,
78                               tl_lats,
79                               tl_scalar,
80                               function='linear')
81     tl_interp = interp_tl(int_lons, int_lats)
82
83     # Find the difference between the flight line and interpolated flight line data
84     diff = (fl_interp - tl_interp).reshape(num_fl, num_tl)
85     diff_lats = int_lats.reshape(num_fl, num_tl)
86     diff_lons = int_lons.reshape(num_fl, num_tl)
87
88     lvld_survey_df = deepcopy(survey_df)
89
90     if approach.lower() == 'lsq':
91         # Find least squares optomized plane of best fit of the difference data at the intercept points
92         A = np.hstack([int_lons[:, np.newaxis],
93                       int_lats[:, np.newaxis],
94                       np.ones((len(int_lons), 1))])
95         C, _, _, _ = la.lstsq(A, diff.flatten())
96
97         # Find the corrections for all flight line data points based on the optomized plane of best fit
98         B = np.hstack([fl_lons[:, np.newaxis],
99                       fl_lats[:, np.newaxis],
100                      np.ones((len(fl_lons), 1))])
101        corrections = B @ C
102
103        # Apply the leveling corrections to all flight line data
104        lvld_survey_df['F'].loc[fl_mask] = fl_scalar - corrections
105
106    else:

```

```

107     # Find the first order line of best fit of the difference data
108     # for each flight line
109     for i, k in enumerate(range(diff.shape[0])):
110         fl_num      = unique_fl_nums[i]
111         fl_samp_mask = (fl_mask) & (survey_df.LINE == fl_num)
112         fl_samp_lats = survey_df.LAT[fl_samp_mask]
113         fl_samp_lons = survey_df.LONG[fl_samp_mask]
114         fl_samp_coords = np.hstack([np.array(fl_samp_lats)[:, np.newaxis],
115                                     np.array(fl_samp_lons)[:, np.newaxis]])
116         ref_coord    = fl_samp_coords[0]
117         fl_samp_dists = distance.cdist(fl_samp_coords,
118                                         [ref_coord],
119                                         'euclidean').flatten()
120
121         diff_line      = diff[k]
122         diff_line_lats = diff_lats[k]
123         diff_line_lons = diff_lons[k]
124         diff_coords    = np.hstack([diff_line_lats[:, np.newaxis],
125                                     diff_line_lons[:, np.newaxis]])
126         diff_dists     = distance.cdist(diff_coords,
127                                         [ref_coord],
128                                         'euclidean').flatten()
129
130         res = stats.linregress(diff_dists, diff_line)
131
132         corrections = res.intercept + (res.slope * fl_samp_dists)
133
134         # Apply the leveling corrections for the given flight line
135         lvld_survey_df['F'].loc[fl_samp_mask] = survey_df['F'].loc[fl_samp_mask] - corrections
136
137     return lvld_survey_df
138
139
140 if __name__ == '__main__':
141     import matplotlib.pyplot as plt
142
143
144     # Flight lines are stacked column-wise:
145     flight_lines = np.array([[1, 1, 1, 1, 1, 1],
146                             [0, 0, 0, 0, 0, 0],
147                             [1, 1, 1, 1, 1, 1],
148                             [0, 0, 0, 0, 0, 0],
149                             [1, 1, 1, 1, 1, 1]])
150
151     lats = np.array([[0, 0, 0, 0, 0, 0],
152                     [1, 1, 1, 1, 1, 1],
153                     [2, 2, 2, 2, 2, 2],
154                     [3, 3, 3, 3, 3, 3],
155                     [4, 4, 4, 4, 4, 4]])
156
157     lons = np.array([[0, 1, 2, 3, 4, 5],
158                     [0, 1, 2, 3, 4, 5],
159                     [0, 1, 2, 3, 4, 5],
160                     [0, 1, 2, 3, 4, 5],

```

```

161             [0, 1, 2, 3, 4, 5]])
162
163     tie_lines = np.array([[1, 1, 1, 1, 1],
164                           [1, 1, 1, 1, 1],
165                           [1, 1, 1, 1, 1]])
166
167     tie_lats = np.array([-0.005, 1, 2, 3, 4.005],
168                           [-0.005, 1, 2, 3, 4.005],
169                           [-0.005, 1, 2, 3, 4.005]])
170
171     tie_lons = np.array([0.005, 0.005, 0.005, 0.005, 0.005],
172                           [2.005, 2.005, 2.005, 2.005, 2.005],
173                           [4.005, 4.005, 4.005, 4.005, 4.005])
174
175     survey_df = pd.DataFrame({'F': np.hstack([flight_lines.flatten(), tie_lines.flatten()]),
176                               'LAT': np.hstack([lats.flatten(), tie_lats.flatten()]),
177                               'LONG': np.hstack([lons.flatten(), tie_lons.flatten()]),
178                               'LINE': np.hstack([np.tile(np.arange(flight_lines.shape[0]) + 1, [flight_lines.shape
179                                         [1], 1]).T.flatten(), np.tile(np.arange(tie_lines.shape[0]) + 1, [tie_lines.shape
180                                         [1], 1]).T.flatten()]),
181                               'LINE_TYPE': np.array(np.hstack([np.ones(len(flight_lines.flatten())), np.ones(len(
182                                         tie_lines.flatten())) * 2]), dtype=int)})
183
184     lvld_df = tie_lvl(survey_df = survey_df,
185                         approach = 'lobf')
186
187
188     plt.figure()
189     plt.scatter(lons, lats, c=flight_lines)
190     plt.clim(0, 1)
191
192     plt.figure()
193     plt.scatter(lvld_df.LONG[lvld_df.LINE_TYPE == 1], lvld_df.LAT[lvld_df.LINE_TYPE == 1], c=lvld_df.F[lvld_df.LINE_TYPE
194                                         == 1])
195     plt.clim(0, 1)
196
197     plt.show()

```

mammal/examples/calibration/data/combine.py

```

1  from os.path import dirname, join
2
3  import pandas as pd
4  import numpy as np
5  from scipy.spatial import distance
6  from scipy import interpolate
7
8  from MAMMAL_Utils import Filters as filt
9  from MAMMAL_Utils import ProcessingUtils as pu
10
11
12 REJ_THRESH = 500 # nT
13 FILT_CUTOFF = 5 # Hz
14 INTERP_FS = 10 # Hz

```

```

15 OUT_FNAME = 'ground_cal.csv'
16 LEVER_ARM = np.array([-0.8000, -0.0508, 0.0635]) # [m, m, m] (body frame)
17
18 LAT = 39.342638
19 LONG = -86.009488
20 ALT = 160
21
22 SRC_DIR = dirname(__file__)
23
24
25 def reindex(df: pd.DataFrame) -> pd.DataFrame:
26     df.index = pd.RangeIndex(len(df.index))
27     return df
28
29 def add_dt_cols(df: pd.DataFrame) -> pd.DataFrame:
30     df['epoch_sec'] = df['timestamp']
31     df['datetime'] = pd.to_datetime(df.epoch_sec, unit='s')
32     return df
33
34 def add_line_cols(log_df: pd.DataFrame,
35                   wpt_df: pd.DataFrame) -> pd.DataFrame:
36     """
37     Add columns denoting line type and line number (if applicable)
38
39     Parameters
40     -----
41     log_df
42         Dataframe compiled from flight log data
43     wpt_df
44         Dataframe of survey waypoints for each line. Columns include:
45
46         - LINE_TYPE
47             Type of line (0 for none, 1 for flight line, and 2 for tie line)
48         - LINE
49             Line number
50         - START_LONG
51             Line start longitude (dd)
52         - START_LAT
53             Line start latitude (dd)
54         - END_LONG
55             Line end longitude (dd)
56         - END_LAT
57             Line end latitude (dd)
58
59     Returns
60     -----
61     log_df
62         Log dataframe with added 'LINE_TYPE' and 'LINE' columns based
63         on survey waypoints
64     """
65
66     log_df['LINE_TYPE'] = 0
67     log_df['LINE'] = 0
68

```

```

69     log_coords = np.hstack([np.array(log_df.LONG)[:, np.newaxis],
70                             np.array(log_df.LAT)[:, np.newaxis]])
71
72     for _, row in wpt_df.iterrows():
73         start_idx = distance.cdist(log_coords,
74                                     [[row.START_LONG, row.START_LAT]],
75                                     'euclidean').argmin()
76         end_idx = distance.cdist(log_coords,
77                                     [[row.END_LONG, row.END_LAT]],
78                                     'euclidean').argmin()
79
80         log_df.LINE_TYPE[start_idx:end_idx] = int(row.LINE_TYPE)
81         log_df.LINE[start_idx:end_idx] = int(row.LINE)
82
83     return log_df
84
85
86 if __name__ == '__main__':
87     pd.set_option('mode.chained_assignment', None)
88
89     # Read in datasets
90     scalar_1_df = add_dt_cols(pd.read_csv(join(SRC_DIR, 'MAG_HEAD_0.csv')))
91     scalar_2_df = add_dt_cols(pd.read_csv(join(SRC_DIR, 'MAG_HEAD_1.csv')))
92     vector_df = add_dt_cols(pd.read_csv(join(SRC_DIR, 'threeaxismagnetometer.csv')))
93     compass_df = add_dt_cols(pd.read_csv(join(SRC_DIR, 'COMPASS.csv')))
94
95
96     # Vector data
97     vector_ts = np.array(vector_df.epoch_sec)
98     vector_x = np.array(vector_df['X']) # nT
99     vector_y = np.array(vector_df['Y']) # nT
100    vector_z = np.array(vector_df['Z']) # nT
101
102    compass_ts = np.array(compass_df.epoch_sec)
103    compass_x = np.array(compass_df['X']) # nT
104    compass_y = np.array(compass_df['Y']) # nT
105    compass_z = np.array(compass_df['Z']) # nT
106
107    interp_vector_x = interpolate.interp1d(vector_ts, vector_x, 'linear', fill_value='extrapolate')
108    interp_vector_y = interpolate.interp1d(vector_ts, vector_y, 'linear', fill_value='extrapolate')
109    interp_vector_z = interpolate.interp1d(vector_ts, vector_z, 'linear', fill_value='extrapolate')
110
111    interp_compass_x = interpolate.interp1d(compass_ts, compass_x, 'linear', fill_value='extrapolate')
112    interp_compass_y = interpolate.interp1d(compass_ts, compass_y, 'linear', fill_value='extrapolate')
113    interp_compass_z = interpolate.interp1d(compass_ts, compass_z, 'linear', fill_value='extrapolate')
114
115
116     # Scalar data
117     scalar_1_ts = np.array(scalar_1_df.epoch_sec)
118     scalar_2_ts = np.array(scalar_2_df.epoch_sec)
119
120     interp_scalar_1 = interpolate.interp1d(scalar_1_ts,
121                                           scalar_1_df.magnitude,
122                                           'linear',

```

```

123             fill_value='extrapolate')
124     interp_scalar_2 = interpolate.interp1d(scalar_2_ts,
125                                         scalar_2_df.magnitude,
126                                         'linear',
127                                         fill_value='extrapolate')
128
129     # Time-align data from both MFAM sensor heads
130     min_t = max([scalar_1_ts.min(), scalar_2_ts.min()])
131     max_t = min([scalar_1_ts.max(), scalar_2_ts.max()])
132
133     scalar_1_df = scalar_1_df[scalar_1_df.epoch_sec >= min_t]
134     scalar_1_df = scalar_1_df[scalar_1_df.epoch_sec <= max_t]
135
136     scalar_2_df = scalar_2_df[scalar_2_df.epoch_sec >= min_t]
137     scalar_2_df = scalar_2_df[scalar_2_df.epoch_sec <= max_t]
138
139     if len(scalar_1_df) > len(scalar_2_df):
140         scalar_1_df = scalar_1_df[:len(scalar_2_df) - len(scalar_1_df)]
141     elif len(scalar_1_df) < len(scalar_2_df):
142         scalar_2_df = scalar_2_df[:len(scalar_1_df) - len(scalar_2_df)]
143
144     scalar_1_df = reindex(scalar_1_df)
145     scalar_1_ts = np.array(scalar_1_df.epoch_sec)
146
147     scalar_2_df = reindex(scalar_2_df)
148     scalar_2_ts = np.array(scalar_2_df.epoch_sec)
149
150     # Add approximate LLA data to calculate IGRF
151     scalar_1_df['LAT'] = LAT
152     scalar_1_df['LONG'] = LONG
153     scalar_1_df['ALT'] = ALT
154
155     scalar_2_df['LAT'] = LAT
156     scalar_2_df['LONG'] = LONG
157     scalar_2_df['ALT'] = ALT
158
159     # Use IGRF field to find when each sensor head values are valid
160     scalar_1_df = pu.add_igrf_cols(scalar_1_df)
161     scalar_2_df = pu.add_igrf_cols(scalar_2_df)
162
163     min_F          = scalar_1_df.IGRF_F - REJ_THRESH
164     max_F          = scalar_1_df.IGRF_F + REJ_THRESH
165     scalar_1_valid_mask = (scalar_1_df.magnitude >= min_F) & (scalar_1_df.magnitude <= max_F)
166     interp_scalar_1_valid = interpolate.interp1d(scalar_1_ts,
167                                                 scalar_1_valid_mask,
168                                                 'linear',
169                                                 fill_value='extrapolate')
170
171     min_F          = scalar_2_df.IGRF_F - REJ_THRESH
172     max_F          = scalar_2_df.IGRF_F + REJ_THRESH
173     scalar_2_valid_mask = (scalar_2_df.magnitude >= min_F) & (scalar_2_df.magnitude <= max_F)
174     interp_scalar_2_valid = interpolate.interp1d(scalar_2_ts,
175                                                 scalar_2_valid_mask,
176                                                 'linear',

```

```

177             fill_value='extrapolate')
178
179     # Find when both and neither sensor head values are valid
180     both_valid_mask = scalar_1_valid_mask & scalar_2_valid_mask
181     neither_valid_mask = ~(scalar_1_valid_mask | scalar_2_valid_mask)
182
183     # Test if all data is "bad"
184     if neither_valid_mask.all():
185         both_valid_mask = ~both_valid_mask
186         neither_valid_mask = ~neither_valid_mask
187
188     print('WARNING: All scalar data was found to be outside the acceptable range from the expected IGRF magnitude, no data will be clipped!')
189
190     # Find when only one sensor head is valid
191     only_scalar_1_valid_mask = scalar_1_valid_mask & (~scalar_2_valid_mask)
192     only_scalar_2_valid_mask = scalar_2_valid_mask & (~scalar_1_valid_mask)
193
194     # Interpolate and filter valid sensor head data
195     interp_scalar_1_lpf = interpolate.interp1d(scalar_1_ts[scalar_1_valid_mask],
196                                                 scalar_1_df.magnitude[scalar_1_valid_mask],
197                                                 'linear',
198                                                 fill_value='extrapolate')
199     interp_scalar_2_lpf = interpolate.interp1d(scalar_2_ts[scalar_2_valid_mask],
200                                                 scalar_2_df.magnitude[scalar_2_valid_mask],
201                                                 'linear',
202                                                 fill_value='extrapolate')
203     scalar_1_lpf = interp_scalar_1_lpf(scalar_1_ts)
204     scalar_2_lpf = interp_scalar_2_lpf(scalar_2_ts)
205
206     if FILT_CUTOFF is not None:
207         scalar_1_lpf = filt.lpf(scalar_1_lpf, FILT_CUTOFF, 1.0 / np.diff(scalar_1_ts).mean())
208         scalar_2_lpf = filt.lpf(scalar_2_lpf, FILT_CUTOFF, 1.0 / np.diff(scalar_2_ts).mean())
209
210
211     # Compile final log DataFrame
212     log_df = pd.DataFrame()
213
214     # Create timestamps at the desired frequency
215     min_t = scalar_1_ts.min()
216     max_t = scalar_1_ts.max()
217     log_ts = np.linspace(min_t, max_t, int((max_t - min_t) * INTERP_FS))
218
219     log_df['epoch_sec'] = log_ts
220     log_df['datetime'] = pd.to_datetime(log_df.epoch_sec, unit='s')
221
222     # Add raw columns for each sensor head
223     log_df['SCALAR_1'] = interp_scalar_1(log_ts)
224     log_df['SCALAR_2'] = interp_scalar_2(log_ts)
225
226     # Add filtered columns for each sensor head
227     log_df['SCALAR_1_LPF'] = interp_scalar_1_lpf(log_ts)
228     log_df['SCALAR_2_LPF'] = interp_scalar_2_lpf(log_ts)
229

```

```

230     # Add columns to specify when each sensor head had valid data
231     log_scalar_1_valid = np.array(interp_scalar_1_valid(log_ts))
232     log_scalar_2_valid = np.array(interp_scalar_2_valid(log_ts))
233
234     # Any valid flag < 1 should be clamped to zero because of interpolation issues
235     log_scalar_1_valid[log_scalar_1_valid != 1] = 0
236     log_scalar_2_valid[log_scalar_2_valid != 1] = 0
237
238     # Save valid flags as bool columns
239     log_df['SCALAR_1_VALID'] = log_scalar_1_valid
240     log_df['SCALAR_2_VALID'] = log_scalar_2_valid
241
242     log_df['SCALAR_1_VALID'] = log_df['SCALAR_1_VALID'].astype('bool')
243     log_df['SCALAR_2_VALID'] = log_df['SCALAR_2_VALID'].astype('bool')
244
245     # Save location data
246     log_df['LAT'] = LAT
247     log_df['LONG'] = LONG
248     log_df['ALT'] = ALT
249
250     # Save vector data
251     log_df['X'] = interp_vector_x(log_ts)
252     log_df['Y'] = interp_vector_y(log_ts)
253     log_df['Z'] = interp_vector_z(log_ts)
254
255     log_df['X_MEAM'] = interp_compass_x(log_ts)
256     log_df['Y_MEAM'] = interp_compass_y(log_ts)
257     log_df['Z_MEAM'] = interp_compass_z(log_ts)
258
259     # Add IGRF data
260     log_df = pu.add_igrf_cols(log_df, fast_mode=True)
261
262
263     # Save the final log DataFrame as a CSV
264     log_df.to_csv(join(SRC_DIR, OUT_FNAME), index=False)

```

mammal/examples/camp\_atterbury\_surveys/\_2k\_atterbury\_survey\_/input\_-  
data/geolocate.py

```

1  from os.path import dirname, join
2
3  import matplotlib.pyplot as plt
4  import matplotlib.cm as cm
5
6  from MAMMAL.Parse import parsePixhawk as ppix
7
8
9  LOG_FNAME = join(dirname(__file__), '00000027.log')
10 CSV_FNAME = join(dirname(__file__), 'EKF.csv')
11
12 MIN_ALT_MSL = 610 # (m)
13
14

```

```

15  if __name__ == '__main__':
16      df = ppix.parsePix(LOG_FNAME, alt_min=MIN_ALT_MSL)
17      df.to_csv(CSV_FNAME, index=False)
18
19      plt.figure()
20      plt.title('Latitude_(dd)')
21      plt.scatter(df.datetime, df.LAT, s=1)
22      plt.grid()
23
24      plt.figure()
25      plt.title('Longitude_(dd)')
26      plt.scatter(df.datetime, df.LONG, s=1)
27      plt.grid()
28
29      plt.figure()
30      plt.title('Altitude_(m)')
31      plt.scatter(df.datetime, df.ALT, s=1)
32      plt.grid()
33
34      plt.figure()
35      plt.title('Path')
36      plt.scatter(df.LONG, df.LAT, s=1, c=df.ALT, cmap=cm.coolwarm)
37      plt.grid()
38
39      plt.show()

```

mammal/examples/camp\_atterbury\_surveys/\_2k\_atterbury\_survey\_/input\_-  
data/combine.py

```

1  from os.path import dirname, join
2
3  import pandas as pd
4  import numpy as np
5  from scipy.spatial import distance
6  from scipy import interpolate
7
8  from MAMMAL_Utils import coordinateUtils as cu
9  from MAMMAL_Utils import Filters as filt
10 from MAMMAL_Utils import ProcessingUtils as pu
11
12
13 REJ_THRESH = 500 # nT
14 FILT_CUTOFF = 5 # Hz
15 INTERP_FS = 10 # Hz
16 OUT_FNAME = '_2k_atterbury_survey_.csv'
17 LEVER_ARM = np.array([-0.8000, -0.0508, 0.0635]) # [m, m, m] (body frame)
18
19 SRC_DIR = dirname(__file__)
20
21
22 def reindex(df: pd.DataFrame) -> pd.DataFrame:
23     df.index = pd.RangeIndex(len(df.index))
24     return df

```

```

25
26 def add_dt_cols(df: pd.DataFrame) -> pd.DataFrame:
27     df['epoch_sec'] = df['timestamp']
28     df['datetime'] = pd.to_datetime(df.epoch_sec, unit='s')
29     return df
30
31 def add_line_cols(log_df: pd.DataFrame,
32                   wpt_df: pd.DataFrame) -> pd.DataFrame:
33     ...
34     Add columns denoting line type and line number (if applicable)
35
36     Parameters
37     -----
38     log_df
39         Dataframe compiled from flight log data
40     wpt_df
41         Dataframe of survey waypoints for each line. Columns include:
42
43         - LINE_TYPE
44             Type of line (0 for none, 1 for flight line, and 2 for tie line)
45         - LINE
46             Line number
47         - START_LONG
48             Line start longitude (dd)
49         - START_LAT
50             Line start latitude (dd)
51         - END_LONG
52             Line end longitude (dd)
53         - END_LAT
54             Line end latitude (dd)
55
56     Returns
57     -----
58     log_df
59         Log dataframe with added 'LINE_TYPE' and 'LINE' columns based
60         on survey waypoints
61     ...
62
63     log_df['LINE_TYPE'] = 0
64     log_df['LINE'] = 0
65
66     log_coords = np.hstack([np.array(log_df.LONG)[:, np.newaxis],
67                            np.array(log_df.LAT)[:, np.newaxis]])
68
69     for _, row in wpt_df.iterrows():
70         start_idx = distance.cdist(log_coords,
71                                     [[row.START_LONG, row.START_LAT]],
72                                     'euclidean').argmin()
73         end_idx = distance.cdist(log_coords,
74                                     [[row.END_LONG, row.END_LAT]],
75                                     'euclidean').argmin()
76
77         log_df.LINE_TYPE[start_idx:end_idx] = int(row.LINE_TYPE)
78         log_df.LINE[start_idx:end_idx] = int(row.LINE)

```

```

79
80     return log_df
81
82
83     if __name__ == '__main__':
84         pd.set_option('mode.chained_assignment', None)
85
86     # Read in datasets
87     wpts_df = pd.read_csv(join(SRC_DIR, 'wpts.csv'))
88     ekf_df = pd.read_csv(join(SRC_DIR, 'EKF.csv'), parse_dates=['datetime'])
89     scalar_1_df = add_dt_cols(pd.read_csv(join(SRC_DIR, 'MAG_HEAD_0.csv')))
90     scalar_2_df = add_dt_cols(pd.read_csv(join(SRC_DIR, 'MAG_HEAD_1.csv')))
91     vector_df = add_dt_cols(pd.read_csv(join(SRC_DIR, 'threeaxismagnetometer.csv')))
92     compass_df = add_dt_cols(pd.read_csv(join(SRC_DIR, 'COMPASS.csv')))
93
94
95     # Positional data
96     ekf_ts = np.array(ekf_df.epoch_sec)
97     lat = np.array(ekf_df['LAT'])      # dd
98     lon = np.array(ekf_df['LONG'])    # dd
99     alt = np.array(ekf_df['ALT'])     # m above MSL
100    pitch = np.array(ekf_df['PITCH'])  # degrees
101    roll = np.array(ekf_df['ROLL'])   # degrees
102    azimuth = np.array(ekf_df['AZMUTH']) # degrees
103
104    interp_lat = interpolate.interp1d(ekf_ts, lat, 'linear', fill_value='extrapolate')
105    interp_lon = interpolate.interp1d(ekf_ts, lon, 'linear', fill_value='extrapolate')
106    interp_alt = interpolate.interp1d(ekf_ts, alt, 'linear', fill_value='extrapolate')
107    interp_pitch = interpolate.interp1d(ekf_ts, pitch, 'linear', fill_value='extrapolate')
108    interp_roll = interpolate.interp1d(ekf_ts, roll, 'linear', fill_value='extrapolate')
109    interp_azimuth = interpolate.interp1d(ekf_ts, azimuth, 'linear', fill_value='extrapolate')
110
111
112     # Vector data
113     vector_ts = np.array(vector_df.epoch_sec)
114     vector_x = np.array(vector_df['X']) # nT
115     vector_y = np.array(vector_df['Y']) # nT
116     vector_z = np.array(vector_df['Z']) # nT
117
118     compass_ts = np.array(compass_df.epoch_sec)
119     compass_x = np.array(compass_df['X']) # nT
120     compass_y = np.array(compass_df['Y']) # nT
121     compass_z = np.array(compass_df['Z']) # nT
122
123     interp_vector_x = interpolate.interp1d(vector_ts, vector_x, 'linear', fill_value='extrapolate')
124     interp_vector_y = interpolate.interp1d(vector_ts, vector_y, 'linear', fill_value='extrapolate')
125     interp_vector_z = interpolate.interp1d(vector_ts, vector_z, 'linear', fill_value='extrapolate')
126
127     interp_compass_x = interpolate.interp1d(compass_ts, compass_x, 'linear', fill_value='extrapolate')
128     interp_compass_y = interpolate.interp1d(compass_ts, compass_y, 'linear', fill_value='extrapolate')
129     interp_compass_z = interpolate.interp1d(compass_ts, compass_z, 'linear', fill_value='extrapolate')
130
131
132     # Scalar data

```

```

133     scalar_1_ts = np.array(scalar_1_df.epoch_sec)
134     scalar_2_ts = np.array(scalar_2_df.epoch_sec)
135
136     interp_scalar_1 = interpolate.interp1d(scalar_1_ts,
137                                             scalar_1_df.magnitude,
138                                             'linear',
139                                             fill_value='extrapolate')
140     interp_scalar_2 = interpolate.interp1d(scalar_2_ts,
141                                             scalar_2_df.magnitude,
142                                             'linear',
143                                             fill_value='extrapolate')
144
145     # Time-align data from both MFAM sensor heads
146     min_t = max([scalar_1_ts.min(), scalar_2_ts.min(), ekf_ts.min()])
147     max_t = min([scalar_1_ts.max(), scalar_2_ts.max(), ekf_ts.max()])
148
149     scalar_1_df = scalar_1_df[scalar_1_df.epoch_sec >= min_t]
150     scalar_1_df = scalar_1_df[scalar_1_df.epoch_sec <= max_t]
151
152     scalar_2_df = scalar_2_df[scalar_2_df.epoch_sec >= min_t]
153     scalar_2_df = scalar_2_df[scalar_2_df.epoch_sec <= max_t]
154
155     if len(scalar_1_df) > len(scalar_2_df):
156         scalar_1_df = scalar_1_df[:len(scalar_2_df) - len(scalar_1_df)]
157     elif len(scalar_1_df) < len(scalar_2_df):
158         scalar_2_df = scalar_2_df[:len(scalar_1_df) - len(scalar_2_df)]
159
160     scalar_1_df = reindex(scalar_1_df)
161     scalar_1_ts = np.array(scalar_1_df.epoch_sec)
162
163     scalar_2_df = reindex(scalar_2_df)
164     scalar_2_ts = np.array(scalar_2_df.epoch_sec)
165
166     # Add approximate LLA data to calculate IGRF
167     scalar_1_df['LAT'] = interp_lat(scalar_1_ts)
168     scalar_1_df['LONG'] = interp_lon(scalar_1_ts)
169     scalar_1_df['ALT'] = interp_alt(scalar_1_ts)
170
171     scalar_2_df['LAT'] = interp_lat(scalar_2_ts)
172     scalar_2_df['LONG'] = interp_lon(scalar_2_ts)
173     scalar_2_df['ALT'] = interp_alt(scalar_2_ts)
174
175     # Use IGRF field to find when each sensor head values are valid
176     scalar_1_df = pu.add_igrf_cols(scalar_1_df)
177     scalar_2_df = pu.add_igrf_cols(scalar_2_df)
178
179     min_F           = scalar_1_df.IGRF_F - REJ_THRESH
180     max_F           = scalar_1_df.IGRF_F + REJ_THRESH
181     scalar_1_valid_mask = (scalar_1_df.magnitude >= min_F) & (scalar_1_df.magnitude <= max_F)
182     interp_scalar_1_valid = interpolate.interp1d(scalar_1_ts,
183                                                   scalar_1_valid_mask,
184                                                   'linear',
185                                                   fill_value='extrapolate')
186

```

```

187     min_F          = scalar_2_df.IGRF_F - REL_THRESH
188     max_F          = scalar_2_df.IGRF_F + REL_THRESH
189     scalar_2_valid_mask = (scalar_2_df.magnitude >= min_F) & (scalar_2_df.magnitude <= max_F)
190     interp_scalar_2_valid = interpolate.interp1d(scalar_2_ts,
191                                               scalar_2_valid_mask,
192                                               'linear',
193                                               fill_value='extrapolate')
194
195     # Find when both and neither sensor head values are valid
196     both_valid_mask = scalar_1_valid_mask & scalar_2_valid_mask
197     neither_valid_mask = ~(scalar_1_valid_mask | scalar_2_valid_mask)
198
199     # Test if all data is "bad"
200     if neither_valid_mask.all():
201         both_valid_mask = ~both_valid_mask
202         neither_valid_mask = ~neither_valid_mask
203
204     print('WARNING: All scalar data was found to be outside the acceptable range from the expected IGRF magnitude, no data will be clipped!')
205
206     # Find when only one sensor head is valid
207     only_scalar_1_valid_mask = scalar_1_valid_mask & (~scalar_2_valid_mask)
208     only_scalar_2_valid_mask = scalar_2_valid_mask & (~scalar_1_valid_mask)
209
210     # Interpolate and filter valid sensor head data
211     interp_scalar_1_lpf = interpolate.interp1d(scalar_1_ts[scalar_1_valid_mask],
212                                              scalar_1_df.magnitude[scalar_1_valid_mask],
213                                              'linear',
214                                              fill_value='extrapolate')
215     interp_scalar_2_lpf = interpolate.interp1d(scalar_2_ts[scalar_2_valid_mask],
216                                              scalar_2_df.magnitude[scalar_2_valid_mask],
217                                              'linear',
218                                              fill_value='extrapolate')
219     scalar_1_lpf = interp_scalar_1_lpf(scalar_1_ts)
220     scalar_2_lpf = interp_scalar_2_lpf(scalar_2_ts)
221
222     if FILT_CUTOFF is not None:
223         scalar_1_lpf = filt.lpf(scalar_1_lpf, FILT_CUTOFF, 1.0 / np.diff(scalar_1_ts).mean())
224         scalar_2_lpf = filt.lpf(scalar_2_lpf, FILT_CUTOFF, 1.0 / np.diff(scalar_2_ts).mean())
225
226
227     # Compile final log DataFrame
228     log_df = pd.DataFrame()
229
230     # Create timestamps at the desired frequency
231     min_t = scalar_1_ts.min()
232     max_t = scalar_1_ts.max()
233     log_ts = np.linspace(min_t, max_t, int((max_t - min_t) * INTERP_FS))
234
235     log_df['epoch_sec'] = log_ts
236     log_df['datetime'] = pd.to_datetime(log_df.epoch_sec, unit='s')
237
238     # Add raw columns for each sensor head
239     log_df['SCALAR_1'] = interp_scalar_1(log_ts)

```

```

240     log_df['SCALAR_2'] = interp_scalar_2(log_ts)
241
242     # Add filtered columns for each sensor head
243     log_df['SCALAR_1_LPF'] = interp_scalar_1_lpf(log_ts)
244     log_df['SCALAR_2_LPF'] = interp_scalar_2_lpf(log_ts)
245
246     # Add columns to specify when each sensor head had valid data
247     log_scalar_1_valid = np.array(interp_scalar_1_valid(log_ts))
248     log_scalar_2_valid = np.array(interp_scalar_2_valid(log_ts))
249
250     # Any valid flag < 1 should be clamped to zero because of interpolation issues
251     log_scalar_1_valid[log_scalar_1_valid != 1] = 0
252     log_scalar_2_valid[log_scalar_2_valid != 1] = 0
253
254     # Save valid flags as bool columns
255     log_df['SCALAR_1_VALID'] = log_scalar_1_valid
256     log_df['SCALAR_2_VALID'] = log_scalar_2_valid
257
258     log_df['SCALAR_1_VALID'] = log_df['SCALAR_1_VALID'].astype('bool')
259     log_df['SCALAR_2_VALID'] = log_df['SCALAR_2_VALID'].astype('bool')
260
261     # Save navigation location data
262     log_df['NAV_LAT'] = interp_lat(log_ts)
263     log_df['NAV_LONG'] = interp_lon(log_ts)
264     log_df['NAV_ALT'] = interp_alt(log_ts)
265
266     # Apply lever arm to navigation LLA data to find true magnetometer LLA
267     nav_lats = np.array(log_df['NAV_LAT'])
268     nav_lons = np.array(log_df['NAV_LONG'])
269     nav_alts = np.array(log_df['NAV_ALT'])
270
271     pitch = interp_pitch(log_ts)
272     roll = interp_roll(log_ts)
273     yaw = interp_azimuth(log_ts)
274
275     eulers = np.hstack([roll[:, np.newaxis],
276                         pitch[:, np.newaxis],
277                         yaw[:, np.newaxis]])
278
279     lats = nav_lats.copy()
280     lons = nav_lons.copy()
281     alts = nav_alts.copy()
282
283     if ~LEVER_ARM == np.zeros(3).all():
284         dcms = pu.angle2dcm(eulers,
285                             angle_unit='degrees',
286                             NED_to_body=False,
287                             rotation_sequence=321)
288         offsets = dcms @ LEVER_ARM
289
290         n = offsets[:, 0]
291         e = offsets[:, 1]
292         d = offsets[:, 2]
293

```

```

294     lats = cu.coord_coord(nav_lats, nav_lons, n / 1000, np.zeros(len(n)))[ :, 0]
295     lons = cu.coord_coord(nav_lats, nav_lons, e / 1000, np.ones(len(n)) * 90)[ :, 1]
296     alts = nav_alts + d
297
298     log_df[ 'LAT' ] = lats
299     log_df[ 'LONG' ] = lons
300     log_df[ 'ALT' ] = alts
301
302     # Save vector data
303     log_df[ 'X' ] = interp_vector_x(log_ts)
304     log_df[ 'Y' ] = interp_vector_y(log_ts)
305     log_df[ 'Z' ] = interp_vector_z(log_ts)
306
307     log_df[ 'X_MEAM' ] = interp_compass_x(log_ts)
308     log_df[ 'Y_MEAM' ] = interp_compass_y(log_ts)
309     log_df[ 'Z_MEAM' ] = interp_compass_z(log_ts)
310
311     # Add IGRF data
312     log_df = pu.add_igrf_cols(log_df, fast_mode=False)
313
314     # Save line data
315     log_df = add_line_cols(log_df, wpts_df)
316
317
318     # Save the final log DataFrame as a CSV
319     log_df.to_csv(join(SRC_DIR, OUT_FNAME), index=False)

```

mammal/examples/camp\_atterbury\_surveys/\_mini\_atterbury\_survey\_/input\_-  
data/geolocate.py

```

1  from os.path import dirname, join
2
3  import matplotlib.pyplot as plt
4  import matplotlib.cm as cm
5
6  from MAMMAL.Parse import parsePixhawk as ppix
7
8
9  LOG_FNAME = join(dirname(__file__), '00000024.log')
10 CSV_FNAME = join(dirname(__file__), 'EKF.csv')
11
12 MIN_ALT_MSL = 610 # (m)
13
14
15 if __name__ == '__main__':
16     df = ppix.parsePix(LOG_FNAME, alt_min=MIN_ALT_MSL)
17     df.to_csv(CSV_FNAME, index=False)
18
19     plt.figure()
20     plt.title('Latitude_(dd)')
21     plt.scatter(df.datetime, df.LAT, s=1)
22     plt.grid()
23

```

```

24     plt.figure()
25     plt.title('Longitude_(dd)')
26     plt.scatter(df.datetime, df.LONG, s=1)
27     plt.grid()
28
29     plt.figure()
30     plt.title('Altitude_(m)')
31     plt.scatter(df.datetime, df.ALT, s=1)
32     plt.grid()
33
34     plt.figure()
35     plt.title('Path')
36     plt.scatter(df.LONG, df.LAT, s=1, c=df.ALT, cmap=cm.coolwarm)
37     plt.grid()
38
39     plt.show()

```

mammal/examples/camp\_atterbury\_surveys/\_mini\_atterbury\_survey\_/input\_-  
data/combine.py

```

1  from os.path import dirname, join
2
3  import pandas as pd
4  import numpy as np
5  from scipy.spatial import distance
6  from scipy import interpolate
7
8  from MAMMAL_Utils import coordinateUtils as cu
9  from MAMMAL_Utils import Filters as filt
10 from MAMMAL_Utils import ProcessingUtils as pu
11
12
13 REJ_THRESH = 500 # nT
14 FILT_CUTOFF = 5 # Hz
15 INTERP_FS = 10 # Hz
16 OUT_FNAME = '_mini_atterbury_survey_.csv'
17 LEVER_ARM = np.array([-0.8000, -0.0508, 0.0635]) # [m, m, m] (body frame)
18
19 SRC_DIR = dirname(__file__)
20
21
22 def reindex(df: pd.DataFrame) -> pd.DataFrame:
23     df.index = pd.RangeIndex(len(df.index))
24     return df
25
26 def add_dt_cols(df: pd.DataFrame) -> pd.DataFrame:
27     df['epoch_sec'] = df['timestamp']
28     df['datetime'] = pd.to_datetime(df.epoch_sec, unit='s')
29     return df
30
31 def add_line_cols(log_df: pd.DataFrame,
32                   wpt_df: pd.DataFrame) -> pd.DataFrame:
33     ...

```

```

34      Add columns denoting line type and line number (if applicable)
35
36      Parameters
37      -----
38      log_df
39          Dataframe compiled from flight log data
40      wpt_df
41          Dataframe of survey waypoints for each line. Columns include:
42
43          - LINE_TYPE
44              Type of line (0 for none, 1 for flight line, and 2 for tie line)
45          - LINE
46              Line number
47          - START_LONG
48              Line start longitude (dd)
49          - START_LAT
50              Line start latitude (dd)
51          - END_LONG
52              Line end longitude (dd)
53          - END_LAT
54              Line end latitude (dd)
55
56      Returns
57      -----
58      log_df
59          Log dataframe with added 'LINE_TYPE' and 'LINE' columns based
60          on survey waypoints
61      ,
62
63      log_df[ 'LINE_TYPE' ] = 0
64      log_df[ 'LINE' ]       = 0
65
66      log_coords = np.hstack([np.array(log_df.LONG)[:, np.newaxis] ,
67                               np.array(log_df.LAT)[:, np.newaxis]])
68
69      for _, row in wpt_df.iterrows():
70          start_idx = distance.cdist(log_coords,
71                                      [[row.START_LONG, row.START_LAT]],
72                                      'euclidean').argmin()
73          end_idx = distance.cdist(log_coords,
74                                      [[row.END_LONG, row.END_LAT]],
75                                      'euclidean').argmin()
76
77          log_df.LINE_TYPE[start_idx:end_idx] = int(row.LINE_TYPE)
78          log_df.LINE[start_idx:end_idx]     = int(row.LINE)
79
80      return log_df
81
82
83  if __name__ == '__main__':
84      pd.set_option('mode.chained_assignment', None)
85
86      # Read in datasets
87      wpts_df = pd.read_csv(join(SRC_DIR, 'wpts.csv'))

```

```

88      ekf_df      = pd.read_csv(join(SRC_DIR, 'EKF.csv'), parse_dates=['datetime'])
89      scalar_1_df = add_dt_cols(pd.read_csv(join(SRC_DIR, 'MAG_HEAD_0.csv')))
90      scalar_2_df = add_dt_cols(pd.read_csv(join(SRC_DIR, 'MAG_HEAD_1.csv')))
91      vector_df   = add_dt_cols(pd.read_csv(join(SRC_DIR, 'threeaxismagnetometer.csv')))
92      compass_df  = add_dt_cols(pd.read_csv(join(SRC_DIR, 'COMPASS.csv')))

93
94
95      # Positional data
96      ekf_ts     = np.array(ekf_df.epoch_sec)
97      lat        = np.array(ekf_df['LAT'])          # dd
98      lon        = np.array(ekf_df['LONG'])         # dd
99      alt        = np.array(ekf_df['ALT'])          # m above MSL
100     pitch      = np.array(ekf_df['PITCH'])        # degrees
101     roll       = np.array(ekf_df['ROLL'])         # degrees
102     azimuth    = np.array(ekf_df['AZIMUTH'])       # degrees

103    interp_lat  = interpolate.interp1d(ekf_ts, lat,      'linear', fill_value='extrapolate')
104    interp_lon  = interpolate.interp1d(ekf_ts, lon,      'linear', fill_value='extrapolate')
105    interp_alt  = interpolate.interp1d(ekf_ts, alt,      'linear', fill_value='extrapolate')
106    interp_pitch = interpolate.interp1d(ekf_ts, pitch,    'linear', fill_value='extrapolate')
107    interp_roll  = interpolate.interp1d(ekf_ts, roll,     'linear', fill_value='extrapolate')
108    interp_azimuth = interpolate.interp1d(ekf_ts, azimuth, 'linear', fill_value='extrapolate')

109
110
111
112      # Vector data
113      vector_ts = np.array(vector_df.epoch_sec)
114      vector_x  = np.array(vector_df['X']) # nT
115      vector_y  = np.array(vector_df['Y']) # nT
116      vector_z  = np.array(vector_df['Z']) # nT

117
118      compass_ts = np.array(compass_df.epoch_sec)
119      compass_x  = np.array(compass_df['X']) # nT
120      compass_y  = np.array(compass_df['Y']) # nT
121      compass_z  = np.array(compass_df['Z']) # nT

122
123      interp_vector_x = interpolate.interp1d(vector_ts, vector_x, 'linear', fill_value='extrapolate')
124      interp_vector_y = interpolate.interp1d(vector_ts, vector_y, 'linear', fill_value='extrapolate')
125      interp_vector_z = interpolate.interp1d(vector_ts, vector_z, 'linear', fill_value='extrapolate')

126
127      interp_compass_x = interpolate.interp1d(compass_ts, compass_x, 'linear', fill_value='extrapolate')
128      interp_compass_y = interpolate.interp1d(compass_ts, compass_y, 'linear', fill_value='extrapolate')
129      interp_compass_z = interpolate.interp1d(compass_ts, compass_z, 'linear', fill_value='extrapolate')

130
131
132      # Scalar data
133      scalar_1_ts = np.array(scalar_1_df.epoch_sec)
134      scalar_2_ts = np.array(scalar_2_df.epoch_sec)

135
136      interp_scalar_1 = interpolate.interp1d(scalar_1_ts,
137                                              scalar_1_df.magnitude,
138                                              'linear',
139                                              fill_value='extrapolate')
140      interp_scalar_2 = interpolate.interp1d(scalar_2_ts,
141                                              scalar_2_df.magnitude,

```

```

142             'linear',
143             fill_value='extrapolate')
144
145     # Time-align data from both MFAM sensor heads
146     min_t = max([scalar_1_ts.min(), scalar_2_ts.min(), ekf_ts.min()])
147     max_t = min([scalar_1_ts.max(), scalar_2_ts.max(), ekf_ts.max()])
148
149     scalar_1_df = scalar_1_df[scalar_1_df.epoch_sec >= min_t]
150     scalar_1_df = scalar_1_df[scalar_1_df.epoch_sec <= max_t]
151
152     scalar_2_df = scalar_2_df[scalar_2_df.epoch_sec >= min_t]
153     scalar_2_df = scalar_2_df[scalar_2_df.epoch_sec <= max_t]
154
155     if len(scalar_1_df) > len(scalar_2_df):
156         scalar_1_df = scalar_1_df[:len(scalar_2_df) - len(scalar_1_df)]
157     elif len(scalar_1_df) < len(scalar_2_df):
158         scalar_2_df = scalar_2_df[:len(scalar_1_df) - len(scalar_2_df)]
159
160     scalar_1_df = reindex(scalar_1_df)
161     scalar_1_ts = np.array(scalar_1_df.epoch_sec)
162
163     scalar_2_df = reindex(scalar_2_df)
164     scalar_2_ts = np.array(scalar_2_df.epoch_sec)
165
166     # Add approximate LLA data to calculate IGRF
167     scalar_1_df['LAT'] = interp_lat(scalar_1_ts)
168     scalar_1_df['LONG'] = interp_lon(scalar_1_ts)
169     scalar_1_df['ALT'] = interp_alt(scalar_1_ts)
170
171     scalar_2_df['LAT'] = interp_lat(scalar_2_ts)
172     scalar_2_df['LONG'] = interp_lon(scalar_2_ts)
173     scalar_2_df['ALT'] = interp_alt(scalar_2_ts)
174
175     # Use IGRF field to find when each sensor head values are valid
176     scalar_1_df = pu.add_igrf_cols(scalar_1_df)
177     scalar_2_df = pu.add_igrf_cols(scalar_2_df)
178
179     min_F           = scalar_1_df.IGRF_F - REJ_THRESH
180     max_F           = scalar_1_df.IGRF_F + REJ_THRESH
181     scalar_1_valid_mask = (scalar_1_df.magnitude >= min_F) & (scalar_1_df.magnitude <= max_F)
182     interp_scalar_1_valid = interpolate.interp1d(scalar_1_ts,
183                                                 scalar_1_valid_mask,
184                                                 'linear',
185                                                 fill_value='extrapolate')
186
187     min_F           = scalar_2_df.IGRF_F - REJ_THRESH
188     max_F           = scalar_2_df.IGRF_F + REJ_THRESH
189     scalar_2_valid_mask = (scalar_2_df.magnitude >= min_F) & (scalar_2_df.magnitude <= max_F)
190     interp_scalar_2_valid = interpolate.interp1d(scalar_2_ts,
191                                                 scalar_2_valid_mask,
192                                                 'linear',
193                                                 fill_value='extrapolate')
194
195     # Find when both and neither sensor head values are valid

```

```

196     both_valid_mask = scalar_1_valid_mask & scalar_2_valid_mask
197     neither_valid_mask = ~(scalar_1_valid_mask | scalar_2_valid_mask)
198
199     # Test if all data is "bad"
200     if neither_valid_mask.all():
201         both_valid_mask = ~both_valid_mask
202         neither_valid_mask = ~neither_valid_mask
203
204     print('WARNING: All scalar data was found to be outside the acceptable range from the expected IGRF magnitude, no data will be clipped!')
205
206     # Find when only one sensor head is valid
207     only_scalar_1_valid_mask = scalar_1_valid_mask & (~scalar_2_valid_mask)
208     only_scalar_2_valid_mask = scalar_2_valid_mask & (~scalar_1_valid_mask)
209
210     # Interpolate and filter valid sensor head data
211     interp_scalar_1_lpf = interpolate.interp1d(scalar_1_ts[scalar_1_valid_mask],
212                                                 scalar_1_df.magnitude[scalar_1_valid_mask],
213                                                 'linear',
214                                                 fill_value='extrapolate')
215     interp_scalar_2_lpf = interpolate.interp1d(scalar_2_ts[scalar_2_valid_mask],
216                                                 scalar_2_df.magnitude[scalar_2_valid_mask],
217                                                 'linear',
218                                                 fill_value='extrapolate')
219     scalar_1_lpf = interp_scalar_1_lpf(scalar_1_ts)
220     scalar_2_lpf = interp_scalar_2_lpf(scalar_2_ts)
221
222     if FILT_CUTOFF is not None:
223         scalar_1_lpf = filt.lpf(scalar_1_lpf, FILT_CUTOFF, 1.0 / np.diff(scalar_1_ts).mean())
224         scalar_2_lpf = filt.lpf(scalar_2_lpf, FILT_CUTOFF, 1.0 / np.diff(scalar_2_ts).mean())
225
226
227     # Compile final log DataFrame
228     log_df = pd.DataFrame()
229
230     # Create timestamps at the desired frequency
231     min_t = scalar_1_ts.min()
232     max_t = scalar_1_ts.max()
233     log_ts = np.linspace(min_t, max_t, int((max_t - min_t) * INTERP_FS))
234
235     log_df['epoch_sec'] = log_ts
236     log_df['datetime'] = pd.to_datetime(log_df.epoch_sec, unit='s')
237
238     # Add raw columns for each sensor head
239     log_df['SCALAR_1'] = interp_scalar_1(log_ts)
240     log_df['SCALAR_2'] = interp_scalar_2(log_ts)
241
242     # Add filtered columns for each sensor head
243     log_df['SCALAR_1_LPF'] = interp_scalar_1_lpf(log_ts)
244     log_df['SCALAR_2_LPF'] = interp_scalar_2_lpf(log_ts)
245
246     # Add columns to specify when each sensor head had valid data
247     log_scalar_1_valid = np.array(interp_scalar_1_valid(log_ts))
248     log_scalar_2_valid = np.array(interp_scalar_2_valid(log_ts))

```

```

249
250     # Any valid flag < 1 should be clamped to zero because of interpolation issues
251     log_scalar_1_valid[log_scalar_1_valid != 1] = 0
252     log_scalar_2_valid[log_scalar_2_valid != 1] = 0
253
254     # Save valid flags as bool columns
255     log_df['SCALAR_1_VALID'] = log_scalar_1_valid
256     log_df['SCALAR_2_VALID'] = log_scalar_2_valid
257
258     log_df['SCALAR_1_VALID'] = log_df['SCALAR_1_VALID'].astype('bool')
259     log_df['SCALAR_2_VALID'] = log_df['SCALAR_2_VALID'].astype('bool')
260
261     # Save navigation location data
262     log_df['NAV_LAT'] = interp_lat(log_ts)
263     log_df['NAV_LONG'] = interp_lon(log_ts)
264     log_df['NAV_ALT'] = interp_alt(log_ts)
265
266     # Apply lever arm to navigation LLA data to find true magnetometer LLA
267     nav_lats = np.array(log_df['NAV_LAT'])
268     nav_lons = np.array(log_df['NAV_LONG'])
269     nav_alts = np.array(log_df['NAV_ALT'])
270
271     pitch = interp_pitch(log_ts)
272     roll = interp_roll(log_ts)
273     yaw = interp_azimuth(log_ts)
274
275     eulers = np.hstack([roll[:, np.newaxis],
276                         pitch[:, np.newaxis],
277                         yaw[:, np.newaxis]]))
278
279     lats = nav_lats.copy()
280     lons = nav_lons.copy()
281     alts = nav_alts.copy()
282
283     if ~LEVER_ARM == np.zeros(3).all():
284         dcms = pu.angle2dcm(eulers,
285                             angle_unit='degrees',
286                             NED_to_body=False,
287                             rotation_sequence=321)
288         offsets = dcms @ LEVER_ARM
289
290         n = offsets[:, 0]
291         e = offsets[:, 1]
292         d = offsets[:, 2]
293
294         lats = cu.coord_coord(nav_lats, nav_lons, n / 1000, np.zeros(len(n)))[ :, 0]
295         lons = cu.coord_coord(nav_lats, nav_lons, e / 1000, np.ones(len(n)) * 90)[ :, 1]
296         alts = nav_alts + d
297
298         log_df['LAT'] = lats
299         log_df['LONG'] = lons
300         log_df['ALT'] = alts
301
302     # Save vector data

```

```

303     log_df['X'] = interp_vector_x(log_ts)
304     log_df['Y'] = interp_vector_y(log_ts)
305     log_df['Z'] = interp_vector_z(log_ts)
306
307     log_df[ 'X_MEAM' ] = interp_compass_x(log_ts)
308     log_df[ 'Y_MEAM' ] = interp_compass_y(log_ts)
309     log_df[ 'Z_MEAM' ] = interp_compass_z(log_ts)
310
311     # Add IGRF data
312     log_df = pu.add_igrf_cols(log_df, fast_mode=False)
313
314     # Save line data
315     log_df = add_line_cols(log_df, wpts_df)
316
317
318     # Save the final log DataFrame as a CSV
319     log_df.to_csv(join(SRC_DIR, OUT_FNAME), index=False)

```

mammal/examples/camp\_atterbury\_surveys/\_mini\_atterbury\_survey\_2\_/input\_data/geolocate.py

```

1  from os.path import dirname, join
2
3  import matplotlib.pyplot as plt
4  import matplotlib.cm as cm
5
6  from MAMMAL.Parse import parsePixhawk as ppix
7
8
9  LOG_FNAME = join(dirname(__file__), '00000026.log')
10 CSV_FNAME = join(dirname(__file__), 'EKF.csv')
11
12 MIN_ALT_MSL = 610 # (m)
13
14
15 if __name__ == '__main__':
16     df = ppix.parsePix(LOG_FNAME, alt_min=MIN_ALT_MSL)
17     df.to_csv(CSV_FNAME, index=False)
18
19     plt.figure()
20     plt.title('Latitude_(dd)')
21     plt.scatter(df.datetime, df.LAT, s=1)
22     plt.grid()
23
24     plt.figure()
25     plt.title('Longitude_(dd)')
26     plt.scatter(df.datetime, df.LONG, s=1)
27     plt.grid()
28
29     plt.figure()
30     plt.title('Altitude_(m)')
31     plt.scatter(df.datetime, df.ALT, s=1)
32     plt.grid()

```

```

33
34     plt.figure()
35     plt.title('Path')
36     plt.scatter(df.LONG, df.LAT, s=1, c=df.ALT, cmap=cm.coolwarm)
37     plt.grid()
38
39     plt.show()

```

mammal/examples/camp\_atterbury\_surveys/\_mini\_atterbury\_survey\_2\_/input\_data/combine.py

```

1  from os.path import dirname, join
2
3  import pandas as pd
4  import numpy as np
5  from scipy.spatial import distance
6  from scipy import interpolate
7
8  from MAMMAL_Utils import coordinateUtils as cu
9  from MAMMAL_Utils import Filters as filt
10 from MAMMAL_Utils import ProcessingUtils as pu
11
12
13 REJ_THRESH = 500 # nT
14 FILT_CUTOFF = 5 # Hz
15 INTERP_FS = 10 # Hz
16 OUT_FNAME = '__mini_atterbury_survey_2__.csv'
17 LEVER_ARM = np.array([-0.8000, -0.0508, 0.0635]) # [m, m, m] (body frame)
18
19 SRC_DIR = dirname(__file__)
20
21
22 def reindex(df: pd.DataFrame) -> pd.DataFrame:
23     df.index = pd.RangeIndex(len(df.index))
24     return df
25
26 def add_dt_cols(df: pd.DataFrame) -> pd.DataFrame:
27     df['epoch_sec'] = df['timestamp']
28     df['datetime'] = pd.to_datetime(df.epoch_sec, unit='s')
29     return df
30
31 def add_line_cols(log_df: pd.DataFrame,
32                   wpt_df: pd.DataFrame) -> pd.DataFrame:
33     """
34     Add columns denoting line type and line number (if applicable)
35
36     Parameters
37     -----
38     log_df
39         Dataframe compiled from flight log data
40     wpt_df
41         Dataframe of survey waypoints for each line. Columns include:
42

```

```

43      - LINE_TYPE
44          Type of line (0 for none, 1 for flight line, and 2 for tie line)
45      - LINE
46          Line number
47      - START_LONG
48          Line start longitude (dd)
49      - START_LAT
50          Line start latitude (dd)
51      - END_LONG
52          Line end longitude (dd)
53      - END_LAT
54          Line end latitude (dd)
55
56      Returns
57  -----
58      log_df
59          Log dataframe with added 'LINE_TYPE' and 'LINE' columns based
60          on survey waypoints
61      ,
62
63      log_df[ 'LINE_TYPE' ] = 0
64      log_df[ 'LINE' ]       = 0
65
66      log_coords = np.hstack([np.array(log_df.LONG)[:, np.newaxis],
67                               np.array(log_df.LAT)[:, np.newaxis]])
68
69      for __, row in wpt_df.iterrows():
70          start_idx = distance.cdist(log_coords,
71                                      [[row.START_LONG, row.START_LAT]],
72                                      'euclidean').argmin()
73          end_idx = distance.cdist(log_coords,
74                                      [[row.END_LONG, row.END_LAT]],
75                                      'euclidean').argmin()
76
77          log_df.LINE_TYPE[start_idx:end_idx] = int(row.LINE_TYPE)
78          log_df.LINE[start_idx:end_idx]     = int(row.LINE)
79
80      return log_df
81
82
83  if __name__ == '__main__':
84      pd.set_option('mode.chained_assignment', None)
85
86      # Read in datasets
87      wpts_df    = pd.read_csv(join(SRC_DIR, 'wpts.csv'))
88      ekf_df     = pd.read_csv(join(SRC_DIR, 'EKF.csv'), parse_dates=['datetime'])
89      scalar_1_df = add_dt_cols(pd.read_csv(join(SRC_DIR, 'MAG_HEAD_0.csv')))
90      scalar_2_df = add_dt_cols(pd.read_csv(join(SRC_DIR, 'MAG_HEAD_1.csv')))
91      vector_df   = add_dt_cols(pd.read_csv(join(SRC_DIR, 'threeaxismagnetometer.csv')))
92      compass_df  = add_dt_cols(pd.read_csv(join(SRC_DIR, 'COMPASS.csv')))
93
94
95      # Positional data
96      ekf_ts = np.array(ekf_df.epoch_sec)

```

```

97     lat      = np.array(ekf_df[ 'LAT' ])      # dd
98     lon      = np.array(ekf_df[ 'LONG' ])      # dd
99     alt      = np.array(ekf_df[ 'ALT' ])       # m above MSL
100    pitch    = np.array(ekf_df[ 'PITCH' ])     # degrees
101    roll     = np.array(ekf_df[ 'ROLL' ])      # degrees
102    azimuth  = np.array(ekf_df[ 'AZIMUTH' ])   # degrees
103
104    interp_lat = interpolate.interp1d(ekf_ts, lat,      'linear', fill_value='extrapolate')
105    interp_lon = interpolate.interp1d(ekf_ts, lon,      'linear', fill_value='extrapolate')
106    interp_alt = interpolate.interp1d(ekf_ts, alt,      'linear', fill_value='extrapolate')
107    interp_pitch = interpolate.interp1d(ekf_ts, pitch,   'linear', fill_value='extrapolate')
108    interp_roll = interpolate.interp1d(ekf_ts, roll,    'linear', fill_value='extrapolate')
109    interp_azimuth = interpolate.interp1d(ekf_ts, azimuth, 'linear', fill_value='extrapolate')
110
111    # Vector data
112    vector_ts = np.array(vector_df.epoch_sec)
113    vector_x  = np.array(vector_df[ 'X' ]) # nT
114    vector_y  = np.array(vector_df[ 'Y' ]) # nT
115    vector_z  = np.array(vector_df[ 'Z' ]) # nT
116
117
118    compass_ts = np.array(compass_df.epoch_sec)
119    compass_x  = np.array(compass_df[ 'X' ]) # nT
120    compass_y  = np.array(compass_df[ 'Y' ]) # nT
121    compass_z  = np.array(compass_df[ 'Z' ]) # nT
122
123    interp_vector_x = interpolate.interp1d(vector_ts, vector_x, 'linear', fill_value='extrapolate')
124    interp_vector_y = interpolate.interp1d(vector_ts, vector_y, 'linear', fill_value='extrapolate')
125    interp_vector_z = interpolate.interp1d(vector_ts, vector_z, 'linear', fill_value='extrapolate')
126
127    interp_compass_x = interpolate.interp1d(compass_ts, compass_x, 'linear', fill_value='extrapolate')
128    interp_compass_y = interpolate.interp1d(compass_ts, compass_y, 'linear', fill_value='extrapolate')
129    interp_compass_z = interpolate.interp1d(compass_ts, compass_z, 'linear', fill_value='extrapolate')
130
131    # Scalar data
132    scalar_1_ts = np.array(scalar_1_df.epoch_sec)
133    scalar_2_ts = np.array(scalar_2_df.epoch_sec)
134
135
136    interp_scalar_1 = interpolate.interp1d(scalar_1_ts,
137                                           scalar_1_df.magnitude,
138                                           'linear',
139                                           fill_value='extrapolate')
140    interp_scalar_2 = interpolate.interp1d(scalar_2_ts,
141                                           scalar_2_df.magnitude,
142                                           'linear',
143                                           fill_value='extrapolate')
144
145    # Time-align data from both MFAM sensor heads
146    min_t = max([scalar_1_ts.min(), scalar_2_ts.min(), ekf_ts.min()])
147    max_t = min([scalar_1_ts.max(), scalar_2_ts.max(), ekf_ts.max()])
148
149    scalar_1_df = scalar_1_df[scalar_1_df.epoch_sec >= min_t]
150    scalar_1_df = scalar_1_df[scalar_1_df.epoch_sec <= max_t]

```

```

151
152     scalar_2_df = scalar_2_df[scalar_2_df.epoch_sec >= min_t]
153     scalar_2_df = scalar_2_df[scalar_2_df.epoch_sec <= max_t]
154
155     if len(scalar_1_df) > len(scalar_2_df):
156         scalar_1_df = scalar_1_df[:len(scalar_2_df) - len(scalar_1_df)]
157     elif len(scalar_1_df) < len(scalar_2_df):
158         scalar_2_df = scalar_2_df[:len(scalar_1_df) - len(scalar_2_df)]
159
160     scalar_1_df = reindex(scalar_1_df)
161     scalar_1_ts = np.array(scalar_1_df.epoch_sec)
162
163     scalar_2_df = reindex(scalar_2_df)
164     scalar_2_ts = np.array(scalar_2_df.epoch_sec)
165
166     # Add approximate LLA data to calculate IGRF
167     scalar_1_df['LAT'] = interp_lat(scalar_1_ts)
168     scalar_1_df['LONG'] = interp_lon(scalar_1_ts)
169     scalar_1_df['ALT'] = interp_alt(scalar_1_ts)
170
171     scalar_2_df['LAT'] = interp_lat(scalar_2_ts)
172     scalar_2_df['LONG'] = interp_lon(scalar_2_ts)
173     scalar_2_df['ALT'] = interp_alt(scalar_2_ts)
174
175     # Use IGRF field to find when each sensor head values are valid
176     scalar_1_df = pu.add_igrf_cols(scalar_1_df)
177     scalar_2_df = pu.add_igrf_cols(scalar_2_df)
178
179     min_F           = scalar_1_df.IGRF_F - REJ_THRESH
180     max_F           = scalar_1_df.IGRF_F + REJ_THRESH
181     scalar_1_valid_mask = (scalar_1_df.magnitude >= min_F) & (scalar_1_df.magnitude <= max_F)
182     interp_scalar_1_valid = interpolate.interp1d(scalar_1_ts,
183                                                 scalar_1_valid_mask,
184                                                 'linear',
185                                                 fill_value='extrapolate')
186
187     min_F           = scalar_2_df.IGRF_F - REJ_THRESH
188     max_F           = scalar_2_df.IGRF_F + REJ_THRESH
189     scalar_2_valid_mask = (scalar_2_df.magnitude >= min_F) & (scalar_2_df.magnitude <= max_F)
190     interp_scalar_2_valid = interpolate.interp1d(scalar_2_ts,
191                                                 scalar_2_valid_mask,
192                                                 'linear',
193                                                 fill_value='extrapolate')
194
195     # Find when both and neither sensor head values are valid
196     both_valid_mask = scalar_1_valid_mask & scalar_2_valid_mask
197     neither_valid_mask = ~(scalar_1_valid_mask | scalar_2_valid_mask)
198
199     # Test if all data is "bad"
200     if neither_valid_mask.all():
201         both_valid_mask = ~both_valid_mask
202         neither_valid_mask = ~neither_valid_mask
203
204     print('WARNING: All scalar data was found to be outside the acceptable range from the expected IGRF magnitude,')

```

```

    no_data_will_be_clipped!')
205
206     # Find when only one sensor head is valid
207     only_scalar_1_valid_mask = scalar_1_valid_mask & (~scalar_2_valid_mask)
208     only_scalar_2_valid_mask = scalar_2_valid_mask & (~scalar_1_valid_mask)
209
210     # Interpolate and filter valid sensor head data
211     interp_scalar_1_lpf = interpolate.interp1d(scalar_1_ts[scalar_1_valid_mask],
212                                                 scalar_1_df.magnitude[scalar_1_valid_mask],
213                                                 'linear',
214                                                 fill_value='extrapolate')
215     interp_scalar_2_lpf = interpolate.interp1d(scalar_2_ts[scalar_2_valid_mask],
216                                                 scalar_2_df.magnitude[scalar_2_valid_mask],
217                                                 'linear',
218                                                 fill_value='extrapolate')
219     scalar_1_lpf = interp_scalar_1_lpf(scalar_1_ts)
220     scalar_2_lpf = interp_scalar_2_lpf(scalar_2_ts)
221
222     if FILT_CUTOFF is not None:
223         scalar_1_lpf = filt.lpf(scalar_1_lpf, FILT_CUTOFF, 1.0 / np.diff(scalar_1_ts).mean())
224         scalar_2_lpf = filt.lpf(scalar_2_lpf, FILT_CUTOFF, 1.0 / np.diff(scalar_2_ts).mean())
225
226
227     # Compile final log DataFrame
228     log_df = pd.DataFrame()
229
230     # Create timestamps at the desired frequency
231     min_t = scalar_1_ts.min()
232     max_t = scalar_1_ts.max()
233     log_ts = np.linspace(min_t, max_t, int((max_t - min_t) * INTERP_FS))
234
235     log_df['epoch_sec'] = log_ts
236     log_df['datetime'] = pd.to_datetime(log_df.epoch_sec, unit='s')
237
238     # Add raw columns for each sensor head
239     log_df['SCALAR_1'] = interp_scalar_1(log_ts)
240     log_df['SCALAR_2'] = interp_scalar_2(log_ts)
241
242     # Add filtered columns for each sensor head
243     log_df['SCALAR_1_LPF'] = interp_scalar_1_lpf(log_ts)
244     log_df['SCALAR_2_LPF'] = interp_scalar_2_lpf(log_ts)
245
246     # Add columns to specify when each sensor head had valid data
247     log_scalar_1_valid = np.array(interp_scalar_1_valid(log_ts))
248     log_scalar_2_valid = np.array(interp_scalar_2_valid(log_ts))
249
250     # Any valid flag < 1 should be clamped to zero because of interpolation issues
251     log_scalar_1_valid[log_scalar_1_valid != 1] = 0
252     log_scalar_2_valid[log_scalar_2_valid != 1] = 0
253
254     # Save valid flags as bool columns
255     log_df['SCALAR_1_VALID'] = log_scalar_1_valid
256     log_df['SCALAR_2_VALID'] = log_scalar_2_valid
257

```

```

258     log_df[ 'SCALAR_1_VALID' ] = log_df[ 'SCALAR_1_VALID' ].astype( 'bool' )
259     log_df[ 'SCALAR_2_VALID' ] = log_df[ 'SCALAR_2_VALID' ].astype( 'bool' )
260
261     # Save navigation location data
262     log_df[ 'NAV_LAT' ] = interp_lat(log_ts)
263     log_df[ 'NAV_LONG' ] = interp_lon(log_ts)
264     log_df[ 'NAV_ALT' ] = interp_alt(log_ts)
265
266     # Apply lever arm to navigation LLA data to find true magnetometer LLA
267     nav_lats = np.array(log_df[ 'NAV_LAT' ])
268     nav_lons = np.array(log_df[ 'NAV_LONG' ])
269     nav_alts = np.array(log_df[ 'NAV_ALT' ])
270
271     pitch = interp_pitch(log_ts)
272     roll = interp_roll(log_ts)
273     yaw = interp_azimuth(log_ts)
274
275     eulers = np.hstack([roll[:, np.newaxis],
276                         pitch[:, np.newaxis],
277                         yaw[:, np.newaxis]])
278
279     lats = nav_lats.copy()
280     lons = nav_lons.copy()
281     alts = nav_alts.copy()
282
283     if ~ (LEVER_ARM == np.zeros(3)).all():
284         dcms = pu.angle2dcm(eulers,
285                               angle_unit='degrees',
286                               NED_to_body=False,
287                               rotation_sequence=321)
288         offsets = dcms @ LEVER_ARM
289
290         n = offsets[:, 0]
291         e = offsets[:, 1]
292         d = offsets[:, 2]
293
294         lats = cu.coord_coord(nav_lats, nav_lons, n / 1000, np.zeros(len(n)))[ :, 0]
295         lons = cu.coord_coord(nav_lats, nav_lons, e / 1000, np.ones(len(n)) * 90)[ :, 1]
296         alts = nav_alts + d
297
298     log_df[ 'LAT' ] = lats
299     log_df[ 'LONG' ] = lons
300     log_df[ 'ALT' ] = alts
301
302     # Save vector data
303     log_df[ 'X' ] = interp_vector_x(log_ts)
304     log_df[ 'Y' ] = interp_vector_y(log_ts)
305     log_df[ 'Z' ] = interp_vector_z(log_ts)
306
307     log_df[ 'X_MFAM' ] = interp_compass_x(log_ts)
308     log_df[ 'Y_MFAM' ] = interp_compass_y(log_ts)
309     log_df[ 'Z_MFAM' ] = interp_compass_z(log_ts)
310
311     # Add IGRF data

```

```

312     log_df = pu.add_igrf_cols(log_df, fast_mode=False)
313
314     # Save line data
315     log_df = add_line_cols(log_df, wpts_df)
316
317
318     # Save the final log DataFrame as a CSV
319     log_df.to_csv(join(SRC_DIR, OUT_FNAME), index=False)

```

mammal/examples/camp\_atterbury\_surveys/\_4k\_atterbury\_survey\_/input\_-  
data/geolocate.py

```

1  from os.path import dirname, join
2
3  import matplotlib.pyplot as plt
4  import matplotlib.cm as cm
5
6  from MAMMAL.Parse import parsePixhawk as ppix
7
8
9  LOG_FNAME = join(dirname(__file__), '00000028.log')
10 CSV_FNAME = join(dirname(__file__), 'EKF.csv')
11
12 MIN_ALT_MSL = 600 # (m)
13
14
15 if __name__ == '__main__':
16     df = ppix.parsePix(LOG_FNAME, alt_min=MIN_ALT_MSL)
17     df.to_csv(CSV_FNAME, index=False)
18
19     plt.figure()
20     plt.title('Latitude_(dd)')
21     plt.scatter(df.datetime, df.LAT, s=1)
22     plt.grid()
23
24     plt.figure()
25     plt.title('Longitude_(dd)')
26     plt.scatter(df.datetime, df.LONG, s=1)
27     plt.grid()
28
29     plt.figure()
30     plt.title('Altitude_(m)')
31     plt.scatter(df.datetime, df.ALT, s=1)
32     plt.grid()
33
34     plt.figure()
35     plt.title('Path')
36     plt.scatter(df.LONG, df.LAT, s=1, c=df.ALT, cmap=cm.coolwarm)
37     plt.grid()
38
39     plt.show()

```

mammal/examples/camp\_atterbury\_surveys/\_4k\_atterbury\_survey\_/input\_-  
data/combine.py

```
1  from os.path import dirname, join
2
3  import pandas as pd
4  import numpy as np
5  from scipy.spatial import distance
6  from scipy import interpolate
7
8  from MAMMAL_Utils import coordinateUtils as cu
9  from MAMMAL_Utils import Filters as filt
10 from MAMMAL_Utils import ProcessingUtils as pu
11
12
13 REJ_THRESH = 500 # nT
14 FILT_CUTOFF = 5 # Hz
15 INTERP_FS = 10 # Hz
16 OUT_FNAME = '_4k_atterbury_survey_.csv'
17 LEVER_ARM = np.array([-0.8000, -0.0508, 0.0635]) # [m, m, m] (body frame)
18
19 SRC_DIR = dirname(__file__)
20
21
22 def reindex(df: pd.DataFrame) -> pd.DataFrame:
23     df.index = pd.RangeIndex(len(df.index))
24     return df
25
26 def add_dt_cols(df: pd.DataFrame) -> pd.DataFrame:
27     df['epoch_sec'] = df['timestamp']
28     df['datetime'] = pd.to_datetime(df.epoch_sec, unit='s')
29     return df
30
31 def add_line_cols(log_df: pd.DataFrame,
32                   wpt_df: pd.DataFrame) -> pd.DataFrame:
33     """
34     Add columns denoting line type and line number (if applicable)
35
36     Parameters
37     -----
38     log_df
39         Dataframe compiled from flight log data
40     wpt_df
41         Dataframe of survey waypoints for each line. Columns include:
42
43         - LINE_TYPE
44             Type of line (0 for none, 1 for flight line, and 2 for tie line)
45         - LINE
46             Line number
47         - START_LONG
48             Line start longitude (dd)
49         - START_LAT
50             Line start latitude (dd)
```

```

51      - END_LONG
52          Line end longitude (dd)
53      - END_LAT
54          Line end latitude (dd)
55
56      Returns
57  -----
58  log_df
59      Log dataframe with added 'LINE_TYPE' and 'LINE' columns based
60      on survey waypoints
61  ,
62
63  log_df[ 'LINE_TYPE' ] = 0
64  log_df[ 'LINE' ]       = 0
65
66  log_coords = np.hstack([np.array(log_df.LONG)[:, np.newaxis] ,
67                         np.array(log_df.LAT)[:, np.newaxis]])
68
69  for _, row in wpt_df.iterrows():
70      start_idx = distance.cdist(log_coords,
71                                  [[row.START_LONG, row.START_LAT]],
72                                  'euclidean').argmin()
73
74      end_idx = distance.cdist(log_coords,
75                                [[row.END_LONG, row.END_LAT]],
76                                'euclidean').argmin()
77
78      log_df.LINE_TYPE[start_idx:end_idx] = int(row.LINE_TYPE)
79      log_df.LINE[start_idx:end_idx]     = int(row.LINE)
80
81
82  if __name__ == '__main__':
83      pd.set_option('mode.chained_assignment', None)
84
85
86      # Read in datasets
87      wpts_df    = pd.read_csv(join(SRC_DIR, 'wpts.csv'))
88      ekf_df     = pd.read_csv(join(SRC_DIR, 'EKF.csv'), parse_dates=['datetime'])
89      scalar_1_df = add_dt_cols(pd.read_csv(join(SRC_DIR, 'MAG_HEAD_0.csv')))
90      scalar_2_df = add_dt_cols(pd.read_csv(join(SRC_DIR, 'MAG_HEAD_1.csv')))
91      vector_df   = add_dt_cols(pd.read_csv(join(SRC_DIR, 'threeaxismagnetometer.csv')))
92      compass_df  = add_dt_cols(pd.read_csv(join(SRC_DIR, 'COMPASS.csv')))
93
94
95      # Positional data
96      ekf_ts    = np.array(ekf_df.epoch_sec)
97      lat        = np.array(ekf_df[ 'LAT' ])      # dd
98      lon        = np.array(ekf_df[ 'LONG' ])     # dd
99      alt        = np.array(ekf_df[ 'ALT' ])      # m above MSL
100     pitch      = np.array(ekf_df[ 'PITCH' ])    # degrees
101     roll       = np.array(ekf_df[ 'ROLL' ])     # degrees
102     azimuth    = np.array(ekf_df[ 'AZIMUTH' ])  # degrees
103
104     interp_lat = interpolate.interp1d(ekf_ts, lat,      'linear', fill_value='extrapolate')

```

```

105     interp_lon = interpolate.interp1d(ekf_ts, lon,      'linear', fill_value='extrapolate')
106     interp_alt = interpolate.interp1d(ekf_ts, alt,      'linear', fill_value='extrapolate')
107     interp_pitch = interpolate.interp1d(ekf_ts, pitch,   'linear', fill_value='extrapolate')
108     interp_roll = interpolate.interp1d(ekf_ts, roll,    'linear', fill_value='extrapolate')
109     interp_azimuth = interpolate.interp1d(ekf_ts, azimuth, 'linear', fill_value='extrapolate')
110
111
112     # Vector data
113     vector_ts = np.array(vector_df.epoch_sec)
114     vector_x = np.array(vector_df['X']) # nT
115     vector_y = np.array(vector_df['Y']) # nT
116     vector_z = np.array(vector_df['Z']) # nT
117
118     compass_ts = np.array(compass_df.epoch_sec)
119     compass_x = np.array(compass_df['X']) # nT
120     compass_y = np.array(compass_df['Y']) # nT
121     compass_z = np.array(compass_df['Z']) # nT
122
123     interp_vector_x = interpolate.interp1d(vector_ts, vector_x, 'linear', fill_value='extrapolate')
124     interp_vector_y = interpolate.interp1d(vector_ts, vector_y, 'linear', fill_value='extrapolate')
125     interp_vector_z = interpolate.interp1d(vector_ts, vector_z, 'linear', fill_value='extrapolate')
126
127     interp_compass_x = interpolate.interp1d(compass_ts, compass_x, 'linear', fill_value='extrapolate')
128     interp_compass_y = interpolate.interp1d(compass_ts, compass_y, 'linear', fill_value='extrapolate')
129     interp_compass_z = interpolate.interp1d(compass_ts, compass_z, 'linear', fill_value='extrapolate')
130
131
132     # Scalar data
133     scalar_1_ts = np.array(scalar_1_df.epoch_sec)
134     scalar_2_ts = np.array(scalar_2_df.epoch_sec)
135
136     interp_scalar_1 = interpolate.interp1d(scalar_1_ts,
137                                             scalar_1_df.magnitude,
138                                             'linear',
139                                             fill_value='extrapolate')
140     interp_scalar_2 = interpolate.interp1d(scalar_2_ts,
141                                             scalar_2_df.magnitude,
142                                             'linear',
143                                             fill_value='extrapolate')
144
145     # Time-align data from both MFAM sensor heads
146     min_t = max([scalar_1_ts.min(), scalar_2_ts.min(), ekf_ts.min()])
147     max_t = min([scalar_1_ts.max(), scalar_2_ts.max(), ekf_ts.max()])
148
149     scalar_1_df = scalar_1_df[scalar_1_df.epoch_sec >= min_t]
150     scalar_1_df = scalar_1_df[scalar_1_df.epoch_sec <= max_t]
151
152     scalar_2_df = scalar_2_df[scalar_2_df.epoch_sec >= min_t]
153     scalar_2_df = scalar_2_df[scalar_2_df.epoch_sec <= max_t]
154
155     if len(scalar_1_df) > len(scalar_2_df):
156         scalar_1_df = scalar_1_df[:len(scalar_2_df) - len(scalar_1_df)]
157     elif len(scalar_1_df) < len(scalar_2_df):
158         scalar_2_df = scalar_2_df[:len(scalar_1_df) - len(scalar_2_df)]

```

```

159
160     scalar_1_df = reindex(scalar_1_df)
161     scalar_1_ts = np.array(scalar_1_df.epoch_sec)
162
163     scalar_2_df = reindex(scalar_2_df)
164     scalar_2_ts = np.array(scalar_2_df.epoch_sec)
165
166     # Add approximate LLA data to calculate IGRF
167     scalar_1_df[ 'LAT' ] = interp_lat(scalar_1_ts)
168     scalar_1_df[ 'LONG' ] = interp_lon(scalar_1_ts)
169     scalar_1_df[ 'ALT' ] = interp_alt(scalar_1_ts)
170
171     scalar_2_df[ 'LAT' ] = interp_lat(scalar_2_ts)
172     scalar_2_df[ 'LONG' ] = interp_lon(scalar_2_ts)
173     scalar_2_df[ 'ALT' ] = interp_alt(scalar_2_ts)
174
175     # Use IGRF field to find when each sensor head values are valid
176     scalar_1_df = pu.add_igrf_cols(scalar_1_df)
177     scalar_2_df = pu.add_igrf_cols(scalar_2_df)
178
179     min_F           = scalar_1_df.IGRF_F - REJ_THRESH
180     max_F           = scalar_1_df.IGRF_F + REJ_THRESH
181     scalar_1_valid_mask = (scalar_1_df.magnitude >= min_F) & (scalar_1_df.magnitude <= max_F)
182     interp_scalar_1_valid = interpolate.interp1d(scalar_1_ts,
183                                                 scalar_1_valid_mask,
184                                                 'linear',
185                                                 fill_value='extrapolate')
186
187     min_F           = scalar_2_df.IGRF_F - REJ_THRESH
188     max_F           = scalar_2_df.IGRF_F + REJ_THRESH
189     scalar_2_valid_mask = (scalar_2_df.magnitude >= min_F) & (scalar_2_df.magnitude <= max_F)
190     interp_scalar_2_valid = interpolate.interp1d(scalar_2_ts,
191                                                 scalar_2_valid_mask,
192                                                 'linear',
193                                                 fill_value='extrapolate')
194
195     # Find when both and neither sensor head values are valid
196     both_valid_mask = scalar_1_valid_mask & scalar_2_valid_mask
197     neither_valid_mask = ~(scalar_1_valid_mask | scalar_2_valid_mask)
198
199     # Test if all data is "bad"
200     if neither_valid_mask.all():
201         both_valid_mask = ~both_valid_mask
202         neither_valid_mask = ~neither_valid_mask
203
204     print('WARNING: All scalar data was found to be outside the acceptable range from the expected IGRF magnitude, no data will be clipped!')
205
206     # Find when only one sensor head is valid
207     only_scalar_1_valid_mask = scalar_1_valid_mask & (~scalar_2_valid_mask)
208     only_scalar_2_valid_mask = scalar_2_valid_mask & (~scalar_1_valid_mask)
209
210     # Interpolate and filter valid sensor head data
211     interp_scalar_1_lpf = interpolate.interp1d(scalar_1_ts[scalar_1_valid_mask],

```

```

212             scalar_1_df.magnitude[scalar_1_valid_mask] ,
213                 'linear' ,
214                 fill_value='extrapolate')
215         interp_scalar_2_lpf = interpolate.interp1d(scalar_2_ts[scalar_2_valid_mask] ,
216                                         scalar_2_df.magnitude[scalar_2_valid_mask] ,
217                                         'linear' ,
218                                         fill_value='extrapolate')
219         scalar_1_lpf = interp_scalar_1_lpf(scalar_1_ts)
220         scalar_2_lpf = interp_scalar_2_lpf(scalar_2_ts)
221
222     if FILT_CUTOFF is not None:
223         scalar_1_lpf = filt.lpf(scalar_1_lpf, FILT_CUTOFF, 1.0 / np.diff(scalar_1_ts).mean())
224         scalar_2_lpf = filt.lpf(scalar_2_lpf, FILT_CUTOFF, 1.0 / np.diff(scalar_2_ts).mean())
225
226
227     # Compile final log DataFrame
228     log_df = pd.DataFrame()
229
230     # Create timestamps at the desired frequency
231     min_t = scalar_1_ts.min()
232     max_t = scalar_1_ts.max()
233     log_ts = np.linspace(min_t, max_t, int((max_t - min_t) * INTERP_FS))
234
235     log_df['epoch_sec'] = log_ts
236     log_df['datetime'] = pd.to_datetime(log_df.epoch_sec, unit='s')
237
238     # Add raw columns for each sensor head
239     log_df['SCALAR_1'] = interp_scalar_1(log_ts)
240     log_df['SCALAR_2'] = interp_scalar_2(log_ts)
241
242     # Add filtered columns for each sensor head
243     log_df['SCALAR_1_LPF'] = interp_scalar_1_lpf(log_ts)
244     log_df['SCALAR_2_LPF'] = interp_scalar_2_lpf(log_ts)
245
246     # Add columns to specify when each sensor head had valid data
247     log_scalar_1_valid = np.array(interp_scalar_1_valid(log_ts))
248     log_scalar_2_valid = np.array(interp_scalar_2_valid(log_ts))
249
250     # Any valid flag < 1 should be clamped to zero because of interpolation issues
251     log_scalar_1_valid[log_scalar_1_valid != 1] = 0
252     log_scalar_2_valid[log_scalar_2_valid != 1] = 0
253
254     # Save valid flags as bool columns
255     log_df['SCALAR_1_VALID'] = log_scalar_1_valid
256     log_df['SCALAR_2_VALID'] = log_scalar_2_valid
257
258     log_df['SCALAR_1_VALID'] = log_df['SCALAR_1_VALID'].astype('bool')
259     log_df['SCALAR_2_VALID'] = log_df['SCALAR_2_VALID'].astype('bool')
260
261     # Save navigation location data
262     log_df['NAV_LAT'] = interp_lat(log_ts)
263     log_df['NAV_LONG'] = interp_lon(log_ts)
264     log_df['NAV_ALT'] = interp_alt(log_ts)
265

```

```

266     # Apply lever arm to navigation LLA data to find true magnetometer LLA
267     nav_lats = np.array(log_df[ 'NAV_LAT' ])
268     nav_lons = np.array(log_df[ 'NAV_LONG' ])
269     nav_alts = np.array(log_df[ 'NAV_ALT' ])
270
271     pitch = interp_pitch(log_ts)
272     roll = interp_roll(log_ts)
273     yaw = interp_azimuth(log_ts)
274
275     eulers = np.hstack([roll[:, np.newaxis],
276                         pitch[:, np.newaxis],
277                         yaw[:, np.newaxis]])
278
279     lats = nav_lats.copy()
280     lons = nav_lons.copy()
281     alts = nav_alts.copy()
282
283     if ~LEVER_ARM == np.zeros(3).all():
284         dcms = pu.angle2dcm(eulers,
285                             angle_unit='degrees',
286                             NED_to_body=False,
287                             rotation_sequence=321)
288         offsets = dcms @ LEVER_ARM
289
290         n = offsets[:, 0]
291         e = offsets[:, 1]
292         d = offsets[:, 2]
293
294         lats = cu.coord_coord(nav_lats, nav_lons, n / 1000, np.zeros(len(n)))[ :, 0]
295         lons = cu.coord_coord(nav_lats, nav_lons, e / 1000, np.ones(len(n)) * 90)[ :, 1]
296         alts = nav_alts + d
297
298     log_df[ 'LAT' ] = lats
299     log_df[ 'LONG' ] = lons
300     log_df[ 'ALT' ] = alts
301
302     # Save vector data
303     log_df[ 'X' ] = interp_vector_x(log_ts)
304     log_df[ 'Y' ] = interp_vector_y(log_ts)
305     log_df[ 'Z' ] = interp_vector_z(log_ts)
306
307     log_df[ 'X_MEAM' ] = interp_compass_x(log_ts)
308     log_df[ 'Y_MEAM' ] = interp_compass_y(log_ts)
309     log_df[ 'Z_MEAM' ] = interp_compass_z(log_ts)
310
311     # Add IGRF data
312     log_df = pu.add_igrf_cols(log_df, fast_mode=False)
313
314     # Save line data
315     log_df = add_line_cols(log_df, wpts_df)
316
317
318     # Save the final log DataFrame as a CSV
319     log_df.to_csv(join(SRC_DIR, OUT_FNAME), index=False)

```

## mammal/requirements.txt

```
1 # python dependencies
2 # for exact match use numpy==1.21.2
3 # for a soft match (match first two numbers) use numpy~=1.21.2
4 # also can use matches for later or earlier versions like numpy>=1.21.2 or numpy<1.21.2
5 numpy==1.20.3
6 pandas==1.3.4
7 scipy==1.7.1
8 matplotlib==3.4.3
9 tqdm==4.62.3
10 #gdal==3.4.3 # Must be installed via conda
11 xarray==2022.3.0
12 rioxarray==0.10.3
13 scikit-learn==0.24.2
14 ppigrf==1.0.0
15 simplekml==1.3.3
16 pyyaml
17 types-PyYAML
18 #geoscraper==0.0.1
19
20 # test requirements
21 # formatters
22 yapf
23 isort
24 # linters
25 mypy
26 flake8
27 flake8-docstrings
28 pylint
29 #testers
30 pytest
31 pytest-cov
32 pytest-order
33 flaky
34 # pre-commit
35 pre-commit
36 pre-commit-hooks
37 blacken-docs
38 bashate
39
40 # docs requirements
41 mkdocs
42 mkdocs-git-revision-date-localized-plugin
43 mkdocs-macros-plugin
44 mkdocs-material
45 mkdocs-mermaid2-plugin
46 mkdocs-pdf-export-plugin
47 # currently only the latest version works for mkdocs serve. TODO remove temp version when new tag is made
48 mktheapidocs[plugin]>=0.2.0.post0.dev21
49 pymdown-extensions
```

## mammal/mkdocs.yml

```

1 # this mkdocs.yml is used to automatically make documentation for modules
2 # for a list of configurations see https://www.mkdocs.org/user-guide/configuration/
3
4 # Project information
5 site_name: 'MAMMAL'
6 site_url: "https://ant.antcenter.io/mammal"
7 repo_url: "https://git.antcenter.net/lbergeron/mammal"
8 repo_name: "GitLab"
9 site_description: 'MAMMAL— Magnetic Anomaly Map Matching Airborne and Land'
10 site_author: "ANT Center"
11
12 # Documentation Layout
13 nav:
14   - 'index.md'
15   - 'container.md'
16   - 'vscode.md'
17   - 'setup_py.md'
18   - 'application.md'
19   - 'kubernetes.md'
20   - 'api-docs-Docstrings'
21   - 'about.md'
22
23 # Build Directories
24 theme:
25   name: readthedocs
26   docs_dir: "docs"
27   site_dir: "site"
28   extra_javascript:
29     - https://unpkg.com/mermaid/dist/mermaid.min.js
30
31 # Formatting options
32 markdown_extensions:
33   - codehilite
34   - def_list
35   - pymdownx.superfences
36   - pymdownx.tasklist:
37     custom_checkbox: true
38
39 plugins:
40   - search
41   - mermaid2
42   - macros
43   - git-revision-date-localized:
44     locale: en
45     type: timeago
46     fallback_to_build_date: true

```

## mammal/README.md

```

1 #MAMMAL
2
3 MAMMAL— Magnetic Anomaly Map Matching Airborne and Land
4

```

```

5 A Python package for simulating and processing aeromagnetic anomaly survey data. It can be used to create magnetic
6 anomaly maps for Magnetic Navigation solutions (MagNav).
7
8 ## Install
9
10 To install MAMMAL, clone the repository to your machine and open a terminal in the folder containing 'setup.py'. Lastly,
11 run the following commands:
12
13
14
15
16 You will also need to download and install the GeoScraper package. Navigate to the [GeoScraper repository](https://git.
17 antcenter.net/lbergeron/geoscraper), clone the repository to your machine, and open a terminal in the folder
18 containing 'setup.py'. Lastly, run the following commands:
19
20
21
22 #### If the osgeo (GDAL) package is not importing correctly on Windows:
23
24 1. Download and install GDAL core and Python binding binaries from https://www.gisinternals.com/release.php
25 2. Find folder where GDAL was installed (usually 'C:\Program Files (x86)\GDAL')
26 3. Create a new environment variable named 'GDAL' and set its value to the GDAL install folder path
27 4. Download the GDAL wheel from https://www.lfd.uci.edu/~gohlke/pythonlibs/#gdal for your CPU type and Python version
28 5. Navigate to the folder the wheel was saved to
29 6. Open a command terminal and run the following:
30
31
32
33
34
35 7. Test installation by opening a python/ipython terminal and trying:
36
37
38 import osgeo
39
40
41 #### If the rioxarray/rasterio packages are not importing correctly on Windows
42
43 If rioxarray is erroring on import, it might be because rasterio was installed incorrectly. If this is the case:
44
45 1. Install rasterio manually by downloading the rasterio wheel from https://www.lfd.uci.edu/~gohlke/pythonlibs/#rasterio
46 for your CPU type and Python version
47 2. Navigate to the folder the wheel was saved to
48 3. Open a command terminal and run the following:
49
50
51
52
53 ## Usage

```

```

54
55  ##### Parsing Log Files
56
57  -----
58
59  To process a single MEAM Dev Kit log and save as a CSV:
60
61  '''Python
62  from MAMMAL.Parse import parseGeometrics as pg
63
64
65  LOG_FNAME = r'dev_kit_log.txt'
66  CSV_FNAME = r'dev_kit_log.csv'
67
68
69  df = pg.parse_devLog(LOG_FNAME)
70  print(df)
71
72  df.to_csv(CSV_FNAME, index=False)
73  '''
74
75  To process an entire acquisition of MEAM Dev Kit logs and save as a CSV:
76
77  '''Python
78  from MAMMAL.Parse import parseGeometrics as pg
79
80
81  LOG_PATH = r'dev_kit_acqu_folder_path'
82  CSV_FNAME = r'dev_kit_log.csv'
83
84
85  df = pg.parse_devACQU(LOG_FNAME)
86  print(df)
87
88  df.to_csv(CSV_FNAME, index=False)
89  '''
90
91  To process a GSMP sensor log and save as a CSV:
92
93  '''Python
94  from MAMMAL.Parse import parseGSMP as pgsmp
95
96
97  LOG_FNAME = r'gsmp_log.txt'
98  CSV_FNAME = r'gsmp_log.csv'
99
100
101  df = pgsmp.parse_GSMP(LOG_FNAME)
102  print(df)
103
104  df.to_csv(CSV_FNAME, index=False)
105  '''
106
107  To process an INTERMAGNET ground reference station log and save as a CSV:

```

```

108
109  '''Python
110  from MAMMAL.Parse import parseIM as pim
111
112
113  LOG_FNAME = r'intermagnet_log.sec'
114  CSV_FNAME = r'intermagnet_log.csv'
115
116
117  df = pim.parse_sec(LOG_FNAME)
118  print(df)
119
120  df.to_csv(CSV_FNAME, index=False)
121  ...
122
123  To process a Pixhawk flight log and save as a CSV:
124
125  '''Python
126  from MAMMAL.Parse import parsePixhawk as pp
127
128
129  LOG_FNAME = r'pix_log.txt'
130  CSV_FNAME = r'pix_log.csv'
131
132
133  df = pp.parsePix(LOG_FNAME)
134  print(df)
135
136  df.to_csv(CSV_FNAME, index=False)
137  ...
138
139  #### Data Processing
140
141  -----
142
143  To find temporal variations after reading-in flight and magnetic reference datasets:
144
145  '''Python
146  import pandas as pd
147
148  from MAMMAL import Diurnal
149
150
151  REF_FNAME = r'ref_log.csv'
152  LOG_FNAME = r'flight_log.csv'
153
154
155  ref_df = pd.read_csv(REF_FNAME, parse_dates=['datetime'])
156
157  log_df      = pd.read_csv(LOG_FNAME, parse_dates=['datetime'])
158  timestamps  = np.array(log_df.epoch_sec)
159
160  __, ref_mag = Diurnal.interp_reference_df(df           = ref_df,
161                                             timestamps = timestamps,

```

```

162             survey_lon    = log_df.LONG.mean() ,
163             subtract_core = True)
164     ...
165
166 To calibrate airborne scalar data:
167
168     '''Python
169     import pandas as pd
170
171     from MAMMAL.VehicleCal import magUtilsTL as magtl
172
173
174 LOG_FNAME = r'flight_log.csv'
175
176 TL_C      = np.array([-1.86687725e+01,  1.33975396e+02, -1.80762945e+02,  1.69023832e-01,
177                         -3.92262356e-03, -1.84382741e-03,  1.71830230e-01, -1.61173781e-04,
178                         1.72575427e-01, -4.31927864e-04, -8.21512835e-05, -4.37609432e-05,
179                         -1.06838978e-04, -1.22444017e-04, -2.76294434e-04, -8.51727772e-05,
180                         3.16374022e-05, -2.77441572e-05])
181 TL_TERMS = magtl.DEFAULT_TL_TERMS
182
183
184 log_df = pd.read_csv(LOG_FNAME, parse_dates=['datetime'])
185 f      = log_df.F
186
187 b_vector = np.hstack([np.array(log_df.X)[:, np.newaxis],
188                       np.array(log_df.Y)[:, np.newaxis],
189                       np.array(log_df.Z)[:, np.newaxis]])
190
191 body_effects_scalar = magtl.tlc_compensation(vector = b_vector,
192                                               tlc      = TL_C,
193                                               terms   = TL_TERMS)
194 f_cal  = f - body_effects_scalar
195 f_cal += (f.mean() - f_cal.mean())
196 ...
197
198 To level scalar anomaly data:
199
200     '''Python
201     import pandas as pd
202
203     from MAMMAL_Utils import ProcessingUtils as pu
204
205
206
207 log_df = pd.DataFrame() # Replace with df where df.F are the scalar anomaly values
208
209 # PCA leveling
210 lvld_survey_df = pcaLvl.pca_lvl(survey_df = log_df,
211                                   num_ptls  = 2,
212                                   ptl_locs  = np.array([0.25, 0.75]))
213
214 # Per flight line leveling
215 lvld_survey_df = tieLvl.tie_lvl(survey_df = log_df,

```

```

216                     approach = 'lobf')
217
218 # Plane of best fit leveling
219 lvld_survey_df = tieLvl.tie_lvl(survey_df = log_df,
220                                 approach = 'lsq')
221 ...
222
223 To interpolate scalar anomaly data:
224
225 '''Python
226 import pandas as pd
227
228 from MAMMAL_Utils import ProcessingUtils as pu
229
230
231 DX = 5 # meters
232 DY = 5 # meters
233
234 MAX_TERRAIN_MSL = 630 # meters
235
236
237 log_df = pd.DataFrame() # Replace with df where df.F are the scalar anomaly values
238
239 interp_type = 'RBF'
240 interp_df = pu.interp_flight_lines(anomaly_df      = log_df,
241                                     dx            = DX,
242                                     dy            = DY,
243                                     max_terrain_msl = MAX_TERRAIN_MSL,
244                                     buffer        = 0,
245                                     interp_type   = interp_type,
246                                     neighbors     = None,
247                                     skip_na_mask  = True)
248 ...
249
250 To create and export a magnetic anomaly map:
251
252 '''Python
253 from MAMMAL_Utils import mapUtils as mu
254
255
256 # Replace each argument with the appropriate value for your use-case
257 # **See export_map doc string for argument details**
258 map = mu.export_map(out_dir       = SURVEY_DIR,
259                      location     = map_title,
260                      date         = log_df.datetime[0],
261                      lats         = interp_lats,
262                      lons         = interp_lons,
263                      scalar        = interp_scalar_LPF,
264                      heights      = interp_heights,
265                      process_df    = pd.DataFrame(process_dict),
266                      process_app   = PROCESS_APP,
267                      stds          = interp_std,
268                      vector        = None,
269                      scalar_type   = SCALAR_TYPE

```

```

270         vector_type      = VECTOR_TYPE
271         scalar_var       = np.nan,
272         vector_var       = np.nan,
273         poc              = POC,
274         flight_path      = flight_path,
275         area_polys       = area_polys,
276         osm_path          = None,
277         level_type        = 'No leveling',
278         tl_coeff_types   = TL_COEFF_TYPES
279         tl_coeffs         = TL_C,
280         interp_type       = interp_type,
281         final_filt_cut   = FINAL_FILT_CUT,
282         final_filt_order  = FINAL_FILT_ORDER)
283     ...
284
285     ##### Map Metadata
286
287     -----
288
289     Magnetic anomaly maps for magnetic navigation (MagNav) must be standardized
290     in a easy to use, common file format with consistent use of units. This will ensure
291     plug-and-play interoperability between all future MagNav filters and maps generated
292     by various sources.
293
294     The GeoTIFF format is a highly versatile extension designed to represent various
295     geospatial data and is ubiquitous in the geospatial data processing discipline with
296     many mapping tools and software already supporting the file format. For this reason,
297     all MagNav survey maps should be published as GeoTIFF files with the following
298     metadata and fields:
299
300     * Coordinate reference system:
301         * WGS84
302     * Orientation of raster bands:
303         * North up
304     * Invalid pixel value:
305         * NaN
306     * Top level metadata:
307         * Metadata field name: "Description"
308             * Standardized value: "MagNav Aeromagnetic Anomaly "Map"
309         * Metadata field name: "ProcessingApp"
310             * Description of the application name and version used to generate the map file
311         * Metadata field name: "SurveyDateUTC"
312             * Approximate UTC data of the survey in an ISO 8601 formatted string
313         * Metadata field name: "SampleDistM"
314             * Approximate distance between each magnetic reading along a given flight line in meters
315         * Metadata field name: "xResolutionM"
316             * Pixel width in meters
317         * Metadata field name: "yResolutionM"
318             * Pixel height in meters
319         * Metadata field name: "ExtentDD"
320             * Extent of map in degrees decimal
321             * Example: "[-84.0958, 39.7617, -84.0484, 39.7823]"
322         * Metadata field name: "ScalarType"
323             * Description of the make/model/type of scalar magnetometer used

```

```

324    * Metadata field name: "VectorType
325        * Description of the make/model/type of vector magnetometer used
326    * Metadata field name: "ScalarSensorVar
327        * Survey scalar magnetometer variance in nT
328    * Metadata field name: "VectorSensorVar
329        * Survey vector magnetometer variance in nT
330    * Metadata field name: "POC
331        * Point of contact information about the organization who conducted the survey and produced the map (no standard
332            format for the information in this metadata field)
333    * Metadata field name: "KML
334        * Keyhole Markup Language (KML) document text that specifies the timestamped survey sample locations; flight/tie
335            line average directions, distances, and altitudes for each sub-survey area; and location of roads, power lines
336            , and substations
337        * The timestamped survey sample locations must be represented by a top-level GxTimeSpan named "FlightPath with
338            UTC timestamps; WGS84 coordinates; and the altitude mode set to "absolute.
339        * The sub-survey areas must be represented by a top-level folder of polygons named "SubSurveyAreas. Each sub-
340            survey area polygon must have the following description: "FL Dir: (fldir) $\circ$ , FL Dist: (fldist)m, TL Dir: (tldir) $\circ$ , TL Dist: (tldist)m, Alt: (alt)m above "MSL where:
341            * "(fldir)" is replaced with the average flight line direction in degrees off North
342            * "(fldist)" is replaced with the average flight line distance in meters
343            * "(tldir)" is replaced with the average tie line direction in degrees off North (if tie lines not present, set
344                to -90)
345            * "(tldist)" is replaced with the average tie line distance in meters (if tie lines not present, set to 0)
346            * "(alt)" is replaced with the average altitude in meters above mean sea level (MSL)
347        * Directions must within the range [0 $\circ$ , 180 $\circ$ ] except for the tie line direction if tie lines are not present (set
348            value to -90)
349        * The road locations must be represented by a top-level multigeometry of line strings named "Roads with WGS84
350            coordinates and the altitude mode to "clampToGround
351        * The power line locations must be represented by a top-level multigeometry of line strings named "PowerLines
352            with WGS84 coordinates and the altitude mode set to "clampToGround
353        * The substation locations must be represented by a top-level multigeometry of polygons named "Substations with
354            WGS84 coordinates and the altitude mode set to "clampToGround
355    * Metadata field name: "LevelType
356        * Description of the algorithm used for map leveling
357    * Metadata field name: "TLCoeffTypes
358        * Ordered list of Tolles-Lawson coefficient types used
359        * Example: "[Permanent, Induced, Eddy]"
360    * Metadata field name: "TLCoeffs
361        * Ordered list of Tolles-Lawson coefficients used
362        * Example: "[0.62, 0.70, 0.55, 0.24, 0.49, 0.28, 0.43, 0.57, 0.90, 0.80, 0.84, 0.14, 0.42, 0.58, 0.85, 0.86, 0.80,
363            0.73]"
364    * Metadata field name: "CSV
365        * Comma-separated values (CSV) document text that includes all pertinent survey data points and data processing
366            steps.
367        * Minimum required columns include:
368            * TIMESTAMP: Coordinated Universal Time (UTC) timestamps (s)
369            * LAT: Latitudes (dd)
370            * LONG: Longitudes (dd)
371            * ALT: Altitudes above MSL (m)
372            * DC_X: Direction cosine X-Components (dimensionless)
373            * DC_Y: Direction cosine Y-Components (dimensionless)
374            * DC_Z: Direction cosine Z-Components (dimensionless)
375            * F: Raw scalar measurements (nT)
376        * Suggested columns include (may vary depending on exact steps used to produce the original map values):

```

```

365 * F_CAL: Calibrated scalar measurements (nT)
366 * F_CAL_IGRF: Calibrated scalar measurements without core field (nT)
367 * F_CAL_IGRF_TEMPORAL: Calibrated scalar measurements without core field or temporal corrections (nT)
368 * F_CAL_IGRF_TEMPORAL_FILT: Calibrated scalar measurements without core field or temporal corrections after low
      pass filtering (nT)
369 * F_CAL_IGRF_TEMPORAL_FILT_LEVEL: Calibrated scalar measurements without core field or temporal corrections after
      low pass filtering and map leveling (nT)
370 * Metadata field name: ""InterpType
371   * Description of the algorithm used for map pixel interpolation
372 * Metadata field name: ""FinalFiltCut
373   * Cutoff wavelength of the 2D low pass filter applied to the interpolated scalar pixel values
374 * Metadata field name: ""FinalFiltOrder
375   * Order number of the 2D low pass filter applied to the interpolated scalar pixel values
376 * Band 1:
377   * NxM raster array of scalar magnetic anomaly values in nT
378   * Band metadata:
379     * Metadata field name: ""Type
380   · Standardized value: ""F
381     * Metadata field name: ""Units
382   · Standardized value: ""nT
383     * Metadata field name: ""Direction
384   · Standardized value: "n/"a
385 * Band 2:
386   * NxM raster array of North magnetic anomaly component values in nT (optional – if no data provided, fill band with
      NaNs)
387   * Band metadata:
388     * Metadata field name: ""Type
389   · Standardized value: ""X
390     * Metadata field name: ""Units
391   · Standardized value: ""nT
392     * Metadata field name: ""Direction
393   · Standardized value: ""North
394 * Band 3:
395   * NxM raster array of East magnetic anomaly component values in nT (optional – if no data provided, fill band with
      NaNs)
396   * Band metadata:
397     * Metadata field name: ""Type
398       * Standardized value: ""Y
399     * Metadata field name: ""Units
400       * Standardized value: ""nT
401     * Metadata field name: ""Direction
402       * Standardized value: ""East
403 * Band 4
404   * NxM raster array of downward magnetic anomaly component values in nT (optional – if no data provided, fill band with
      NaNs)
405   * Band metadata:
406     * Metadata field name: ""Type
407       * Standardized value: ""Z
408     * Metadata field name: ""Units
409       * Standardized value: ""nT
410     * Metadata field name: ""Direction
411       * Standardized value: ""Down
412 * Band 5
413   * NxM raster array of pixel altitudes in meters above MSL

```

```

414     * Band metadata:
415         * Metadata field name: "Type
416             * Standardized value: "ALT
417         * Metadata field name: "Units
418             * Standardized value: "m 'MSL
419         * Metadata field name: "Direction
420             * Standardized value: "n/"a
421     * Band 6
422         * NxM raster array of pixel interpolation standard deviation values in nT (optional – if no data provided, fill band
423             with NaNs)
423     * Band metadata:
424         * Metadata field name: "Type
425             * Standardized value: "STD
426         * Metadata field name: "Units
427             * Standardized value: "nT
428         * Metadata field name: "Direction
429             * Standardized value: "n/"a
430     * Band 7
431         * NxM raster array of pixel easterly gradients
432     * Band metadata:
433         * Metadata field name: "Type
434             * Standardized value: "dX
435         * Metadata field name: "Units
436             * Standardized value: "nT
437         * Metadata field name: "Direction
438             * Standardized value: "East
439     * Band 8
440         * NxM raster array of pixel northerly gradients
441     * Band metadata:
442         * Metadata field name: "Type
443             * Standardized value: "dY
444         * Metadata field name: "Units
445             * Standardized value: "nT
446         * Metadata field name: "Direction
447             * Standardized value: "North

```

## mammal/setup.cfg

```

1 [metadata]
2 description-file = README.md

```

## Appendix G. GeoScraper Python Library

### GeoScraper/setup.py

```
1 from setuptools import setup
2
3 with open('README.md', "r") as fh:
4     long_description = fh.read()
5
6 setup(
7     name          = 'GeoScraper',
8     packages      = ['GeoScraper'],
9     version       = '0.0.1',
10    description   = 'Very user friendly library to parse OpenSourceMap data either from an XML file (i.e. *.osm),\n        from an OSM API URL, or from a user specified bounding box',
11    long_description = long_description,
12    long_description_content_type = "text/markdown",
13    author        = 'Power_Broker',
14    author_email  = 'gitstuff2@gmail.com',
15    url          = 'https://github.com/PowerBroker2/GeoScraper',
16    download_url  = 'https://github.com/PowerBroker2/GeoScraper/archive/0.0.1.tar.gz',
17    keywords      = ['geospatial', 'map', 'maps', 'mapping', 'osm', 'openstreetmap'],
18    classifiers   = [],
19    install_requires = ['xmltodict', 'simplekml']
20 )
```

### GeoScraper/GeoScraper/\_\_init\_\_.py

```
1 """
2 geoscraper
3
4 https://wiki.openstreetmap.org/wiki/API
5 https://wiki.openstreetmap.org/wiki/OSM_XML/XSD
6 https://wiki.openstreetmap.org/wiki/Map_features
7 ...
8
9 import os
10 import requests
11 from copy import deepcopy
12 from typing import Union
13 from urllib.parse import urlparse
14
15 import xmltodict
16
17
18 class GeoScraper_Path_Exception(Exception):
19     pass
20
21
22 class GeoScraper_File_Exception(Exception):
23     pass
```

```

25
26     class GeoScraper_URL_Exception(Exception):
27         pass
28
29
30     class GeoScraper_Node_Exception(Exception):
31         pass
32
33
34     class OSM_URL_Wizard(object):
35         def __init__(self,
36             api_ver: Union[str, float] = '0.6'):
37             self.api_ver = api_ver
38
39         def api_version_url(self) -> str:
40             return r'https://api.openstreetmap.org/api/versions'
41
42         def capabilities_url(self) -> str:
43             return r'https://api.openstreetmap.org/api/capabilities'
44
45         def bbox_url(self,
46             left: Union[float, str],
47             bottom: Union[float, str],
48             right: Union[float, str],
49             top: Union[float, str]) -> str:
50             return r'https://api.openstreetmap.org/api/{api_ver}/map?bbox={left},{bottom},{right},{top}'.format(api_ver =
51                         self.api_ver,
52                                         left      =
53                                         left,
54                                         bottom   =
55                                         bottom,
56                                         ,
57                                         right    =
58                                         right,
59                                         top      =
59                                         top)
56
57         def permissions_url(self) -> str:
58             return r'https://api.openstreetmap.org/api/{api_ver}/permissions'.format(api_ver = self.api_ver)
59
60         def get_node_url(self,
61             id: Union[str, int]) -> str:
62             return r'https://api.openstreetmap.org/api/{api_ver}/node/{id}'.format(api_ver = self.api_ver,
63                                         id      = id)
64
65         def get_way_url(self,
66             id: Union[str, int]) -> str:
67             return r'https://api.openstreetmap.org/api/{api_ver}/way/{id}'.format(api_ver = self.api_ver,
68                                         id      = id)
69
70         def get_relation_url(self,
71             id: Union[str, int]) -> str:
72             return r'https://api.openstreetmap.org/api/{api_ver}/relation/{id}'.format(api_ver = self.api_ver,
72                                         id      = id)

```

```

73
74     def get_node_history_url(self,
75             id: Union[str, int]) -> str:
76         return r'https://api.openstreetmap.org/api/{api_ver}/node/{id}/history'.format(api_ver = self.api_ver,
77                                         id = id)
78
79     def get_way_history_url(self,
80             id: Union[str, int]) -> str:
81         return r'https://api.openstreetmap.org/api/{api_ver}/way/{id}/history'.format(api_ver = self.api_ver,
82                                         id = id)
83
84     def get_relation_history_url(self,
85             id: Union[str, int]) -> str:
86         return r'https://api.openstreetmap.org/api/{api_ver}/relation/{id}/history'.format(api_ver = self.api_ver,
87                                         id = id)
88
89     def get_node_version_url(self,
90             id: Union[str, int],
91             version: Union[str, int]) -> str:
92         return r'https://api.openstreetmap.org/api/{api_ver}/node/{id}/{version}'.format(api_ver = self.api_ver,
93                                         id = id,
94                                         version = version)
95
96     def get_way_version_url(self,
97             id: Union[str, int],
98             version: Union[str, int]) -> str:
99         return r'https://api.openstreetmap.org/api/{api_ver}/way/{id}/{version}'.format(api_ver = self.api_ver,
100                                         id = id,
101                                         version = version)
102
103    def get_relation_version_url(self,
104            id: Union[str, int],
105            version: Union[str, int]) -> str:
106        return r'https://api.openstreetmap.org/api/{api_ver}/relation/{id}/{version}'.format(api_ver = self.api_ver,
107                                         id = id,
108                                         version = version)
109
110    def get_nodes_url(self,
111        ids: list[Union[str, int]]) -> str:
112        return r'https://api.openstreetmap.org/api/{api_ver}/nodes?{ids}'.format(api_ver = self.api_ver,
113                                         ids = ','.join(ids))
114
115    def get_ways_url(self,
116        ids: list[Union[str, int]]) -> str:
117        return r'https://api.openstreetmap.org/api/{api_ver}/ways?{ids}'.format(api_ver = self.api_ver,
118                                         ids = ','.join(ids))
119
120    def get_relations_url(self,
121        ids: list[Union[str, int]]) -> str:
122        return r'https://api.openstreetmap.org/api/{api_ver}/relations?{ids}'.format(api_ver = self.api_ver,
123                                         ids = ','.join(ids))
124
125    def get_node_relations_url(self,
126        id: Union[str, int]) -> str:

```

```

127     return r'https://api.openstreetmap.org/api/{api_ver}/node/{id}/relations'.format(api_ver = self.api_ver,
128                                         id = id)
129
130     def get_way_relations_url(self,
131                               id: Union[str, int]) -> str:
132         return r'https://api.openstreetmap.org/api/{api_ver}/way/{id}/relations'.format(api_ver = self.api_ver,
133                                         id = id)
134
135     def get_relation_relations_url(self,
136                                   id: Union[str, int]) -> str:
137         return r'https://api.openstreetmap.org/api/{api_ver}/relation/{id}/relations'.format(api_ver = self.api_ver,
138                                         id = id)
139
140     def get_ways_for_node_url(self,
141                               id: Union[str, int]) -> str:
142         return r'https://api.openstreetmap.org/api/{api_ver}/node/{id}/ways'.format(api_ver = self.api_ver,
143                                         id = id)
144
145     def get_way_full_data_url(self,
146                               id: Union[str, int]) -> str:
147         return r'https://api.openstreetmap.org/api/{api_ver}/way/{id}/full'.format(api_ver = self.api_ver,
148                                         id = id)
149
150     def get_relation_full_data_url(self,
151                                   id: Union[str, int]) -> str:
152         return r'https://api.openstreetmap.org/api/{api_ver}/relation/{id}/full'.format(api_ver = self.api_ver,
153                                         id = id)
154
155     def get_user_details_url(self,
156                               id: Union[str, int]) -> str:
157         return r'https://api.openstreetmap.org/api/{api_ver}/user/{id}'.format(api_ver = self.api_ver,
158                                         id = id)
159
160     def get_users_details_url(self,
161                               ids: list[Union[str, int]]) -> str:
162         return r'https://api.openstreetmap.org/api/{api_ver}/users?users={ids}'.format(api_ver = self.api_ver,
163                                         ids = ','.join(ids))
164
165     def get_your_details_url(self) -> str:
166         return r'https://api.openstreetmap.org/api/{api_ver}/user/details'.format(api_ver = self.api_ver)
167
168     def get_your_preferences_url(self) -> str:
169         return r'https://api.openstreetmap.org/api/{api_ver}/user/preferences'.format(api_ver = self.api_ver)
170
171     def get_your_preference_url(self,
172                                preference: str) -> str:
173         return r'https://api.openstreetmap.org/api/{api_ver}/user/preferences/{preference}'.format(api_ver = self.
174                                         api_ver,
175                                         preference =
176                                         preference)
177
178     class OSM_XML(object):
179         def __init__(self) -> None:

```

```

179         self.__xml = ''
180         self.__clear_datasets()
181
182     def __clear_datasets(self) -> None:
183         self.__raw_data      = {}
184         self.__data          = {}
185         self.__Aerialway    = []
186         self.__Aeroway       = []
187         self.__Amenity      = []
188         self.__Barrier       = []
189         self.__Boundary      = []
190         self.__Building      = []
191         self.__Craft         = []
192         self.__Emergency     = []
193         self.__Geological    = []
194         self.__Healthcare    = []
195         self.__Highway        = []
196         self.__Historic      = []
197         self.__Landuse        = []
198         self.__Leisure        = []
199         self.__Man_made      = []
200         self.__Military       = []
201         self.__Natural        = []
202         self.__Office         = []
203         self.__Place          = []
204         self.__Power          = []
205         self.__Public_transport = []
206         self.__Railway        = []
207         self.__Route          = []
208         self.__Shop           = []
209         self.__Sport          = []
210         self.__Telecom         = []
211         self.__Tourism         = []
212         self.__Water          = []
213         self.__Waterway       = []
214
215     def set_xml(self,
216                 xml: str) -> None:
217         self.__xml = xml
218
219     def __parse_file(self) -> None:
220         self.__raw_data = xmltodict.parse(self.__xml)
221
222     def __link_nd_tag(self,
223                       node_id: str) -> dict:
224         try:
225             match_idx = self.__node_keys.index(node_id)
226             return self.__nodes[match_idx]
227         except ValueError:
228             self.__clear_datasets()
229             raise GeoScraper_Node_Exception('Could not find a matching node with ID: {}'.format(node_id))
230
231     def __link_data(self) -> None:
232         if 'osm' not in self.__raw_data.keys():

```

```

233         self.__clear_datasets()
234         raise GeoScraper_File_Exception( 'Invalid_file\osm\element_not_found' )
235
236     try:
237         self.__nodes      = self.__raw_data[ 'osm' ][ 'node' ]
238         self.__node_keys = [ node[key] for node in self.__nodes for key in node if key == '@id' ]
239     except KeyError:
240         self.__clear_datasets()
241         raise GeoScraper_Node_Exception( 'No_nodes_found_in_file' )
242
243     self.__data_types = [key for key in self.__raw_data[ 'osm' ].keys() if not key.startswith( '@' )]
244     self.__data       = deepcopy( self.__raw_data[ 'osm' ] )
245
246     to_pop = []
247
248     for key in self.__data:
249         if key not in self.__data_types:
250             to_pop.append(key)
251
252     for key in to_pop:
253         self.__data.pop(key, None)
254
255     for element_name in self.__data_types:
256         element = self.__data[element_name]
257
258         if (element_name == 'node') or (element is dict):
259             continue
260
261         for feature_num, feature in enumerate(element):
262             if type(feature) is not dict:
263                 continue
264
265             orig_feat_keys = feature.keys()
266
267             for key in orig_feat_keys:
268                 if type(feature[key]) == list:
269                     self.__data[element_name][feature_num][key] = compress_dicts(feature[key])
270
271                 if 'nd' not in feature.keys():
272                     continue
273
274                 matching_node_data = [ self.__link_nd_tag(nd_ref) for nd_ref in feature[ 'nd' ][ '@ref' ] ]
275                 self.__data[element_name][feature_num][ 'node' ] = matching_node_data
276
277     def __sort_ways(self) -> None:
278         elements = []
279
280         if 'way' in self.__data.keys():
281             elements.extend(self.__data[ 'way' ])
282
283         if 'node' in self.__data.keys():
284             elements.extend(self.__data[ 'node' ])
285
286         for element in elements:

```

```

287     if 'tag' not in element.keys():
288         continue
289
290     if type(element['tag']) is list:
291         element['tag'] = compress_dicts(element['tag'])
292
293     element_keys      = element.keys()
294     element_tag_keys = element['tag'][ '@k' ]
295
296     node_ids = []
297
298     if 'nd' in element_keys:
299         node_ids = element['nd'][ '@ref' ]
300
301     tags      = element['tag']
302     tag_keys = tags[ '@k' ]
303     tag_vals = tags[ '@v' ]
304     tags      = list(zip(tag_keys, tag_vals))
305
306     lats = []
307     lons = []
308
309     if 'node' in element_keys:
310         for node in element['node']:
311             lats.append(node[ '@lat' ])
312             lons.append(node[ '@lon' ])
313
314     elif ('@lat' in element_keys) and ('@lon' in element_keys):
315         lats.append(element[ '@lat' ])
316         lons.append(element[ '@lon' ])
317
318     coords = list(zip(lons, lats))
319
320     entry = { 'id':           element[ '@id' ] ,
321               'uid':          element[ '@uid' ] ,
322               'timestamp':    element[ '@timestamp' ] ,
323               'node_ids':     node_ids,
324               'tag_keys':     tag_keys,
325               'tag_vals':     tag_vals,
326               'tags':          tags,
327               'lats':          lats,
328               'lons':          lons,
329               'coords':        coords}
330
331     if 'aerialway' in element_tag_keys:
332         self.__Aerialway.append(entry)
333
334     if 'aeroway' in element_tag_keys:
335         self.__Aeroway.append(entry)
336
337     if 'amenity' in element_tag_keys:
338         self.__Amenity.append(entry)
339
340     if 'barrier' in element_tag_keys:

```

```

341         self.__Barrier.append(entry)
342
343     if 'boundary' in element_tag_keys:
344         self.__Boundary.append(entry)
345
346     if 'building' in element_tag_keys:
347         self.__Building.append(entry)
348
349     if 'craft' in element_tag_keys:
350         self.__Craft.append(entry)
351
352     if 'emergency' in element_tag_keys:
353         self.__Emergency.append(entry)
354
355     if 'geological' in element_tag_keys:
356         self.__Geological.append(entry)
357
358     if 'healthcare' in element_tag_keys:
359         self.__Healthcare.append(entry)
360
361     if 'highway' in element_tag_keys:
362         self.__Highway.append(entry)
363
364     if 'historic' in element_tag_keys:
365         self.__Historic.append(entry)
366
367     if 'landuse' in element_tag_keys:
368         self.__Landuse.append(entry)
369
370     if 'leisure' in element_tag_keys:
371         self.__Leisure.append(entry)
372
373     if 'man_made' in element_tag_keys:
374         self.__Man_made.append(entry)
375
376     if 'military' in element_tag_keys:
377         self.__Military.append(entry)
378
379     if 'natural' in element_tag_keys:
380         self.__Natural.append(entry)
381
382     if 'office' in element_tag_keys:
383         self.__Office.append(entry)
384
385     if 'place' in element_tag_keys:
386         self.__Place.append(entry)
387
388     if 'power' in element_tag_keys:
389         self.__Power.append(entry)
390
391     if 'public_transport' in element_tag_keys:
392         self.__Public_transport.append(entry)
393
394     if 'railway' in element_tag_keys:

```

```

395         self.__Railway.append(entry)
396
397     if 'route' in element_tag_keys:
398         self.__Route.append(entry)
399
400     if 'shop' in element_tag_keys:
401         self.__Shop.append(entry)
402
403     if 'sport' in element_tag_keys:
404         self.__Sport.append(entry)
405
406     if 'telecom' in element_tag_keys:
407         self.__Telecom.append(entry)
408
409     if 'tourism' in element_tag_keys:
410         self.__Tourism.append(entry)
411
412     if 'water' in element_tag_keys:
413         self.__Water.append(entry)
414
415     if 'waterway' in element_tag_keys:
416         self.__Waterway.append(entry)
417
418     def parse(self,
419             xml: str=None) -> dict:
420         self.__clear_datasets()
421
422         if xml is not None:
423             self.set_xml(xml)
424
425         self.__parse_file()
426         self.__link_data()
427         self.__sort_ways()
428
429         return self.__data
430
431     def raw_data(self) -> dict:
432         return self.__raw_data
433
434     def data(self) -> dict:
435         return self.__data
436
437     def nodes(self) -> dict:
438         return self.__data['node']
439
440     def ways(self) -> dict:
441         return self.__data['way']
442
443     def relations(self) -> dict:
444         return self.__data['relation']
445
446     def aerialways(self) -> list[dict]:
447         return self.__Aerialway
448

```

```

449     def aereways(self) -> list[dict]:
450         return self.__Aeroway
451
452     def amenities(self) -> list[dict]:
453         return self.__Amenity
454
455     def barriers(self) -> list[dict]:
456         return self.__Barrier
457
458     def boundaries(self) -> list[dict]:
459         return self.__Boundary
460
461     def buildings(self) -> list[dict]:
462         return self.__Building
463
464     def crafts(self) -> list[dict]:
465         return self.__Craft
466
467     def emergency_locations(self) -> list[dict]:
468         return self.__Emergency
469
470     def geological_locations(self) -> list[dict]:
471         return self.__Geological
472
473     def healthcare_locations(self) -> list[dict]:
474         return self.__Healthcare
475
476     def highways(self) -> list[dict]:
477         return self.__Highway
478
479     def historic_locations(self) -> list[dict]:
480         return self.__Historic
481
482     def landuse(self) -> list[dict]:
483         return self.__Landuse
484
485     def leisure_locations(self) -> list[dict]:
486         return self.__Leisure
487
488     def man_made_locations(self) -> list[dict]:
489         return self.__Man_made
490
491     def military_locations(self) -> list[dict]:
492         return self.__Military
493
494     def natural_locations(self) -> list[dict]:
495         return self.__Natural
496
497     def offices(self) -> list[dict]:
498         return self.__Office
499
500     def places(self) -> list[dict]:
501         return self.__Place
502

```

```

503     def power_locations(self) -> list[dict]:
504         return self.__Power
505
506     def public_transport_locations(self) -> list[dict]:
507         return self.__Public_transport
508
509     def railways(self) -> list[dict]:
510         return self.__Railway
511
512     def routes(self) -> list[dict]:
513         return self.__Route
514
515     def shops(self) -> list[dict]:
516         return self.__Shop
517
518     def sport_locations(self) -> list[dict]:
519         return self.__Sport
520
521     def telecom_locations(self) -> list[dict]:
522         return self.__Telecom
523
524     def tourism_locations(self) -> list[dict]:
525         return self.__Tourism
526
527     def water_locations(self) -> list[dict]:
528         return self.__Water
529
530     def waterways(self) -> list[dict]:
531         return self.__Waterway
532
533
534     class GeoScraper(OSM_URL_Wizard, OSM_XML):
535         def __init__(self,
536                     api_ver: Union[str, float] = '0.6'):
537             super().__init__(api_ver)
538
539         def from_str(self,
540                     xml_text: str) -> dict:
541             return self.parse(xml_text)
542
543         def from_file(self,
544                     path: str) -> dict:
545             if not os.path.exists(path):
546                 raise GeoScraper_File_Exception('File does not exist')
547
548             with open(path, 'r') as xml:
549                 xml_text = xml.read()
550
551             return self.from_str(xml_text)
552
553         def from_url(self,
554                     url: str) -> dict:
555             if not uri_validator(url):
556                 raise GeoScraper_URL_Exception('URL is invalid')

```

```

557     r = requests.get(url)
559
560     if not r.ok:
561         raise GeoScraper_URL_Exception( 'URL_Request_Error_Code:{}\n{}'.format(r.status_code,
562                                         r.text))
563
564     return self.from_str(r.text)
565
566     def from_bbox(self,
567                 left: Union[float, str],
568                 bottom: Union[float, str],
569                 right: Union[float, str],
570                 top: Union[float, str]) -> dict:
571
572         url = self.bbox_url(left,
573                             bottom,
574                             right,
575                             top)
576
577         if not uri_validator(url):
578             raise GeoScraper_URL_Exception( 'URL_is_invalid')
579
580         r = requests.get(url)
581
582         if not r.ok:
583             raise GeoScraper_URL_Exception( 'URL_Request_Error_Code:{}'.format(r.status_code))
584
585         return self.from_str(r.text)
586
587
588     def uri_validator(x: str) -> bool:
589         """
590         https://stackoverflow.com/a/38020041/9860973
591         """
592
593         try:
594             result = urlparse(x)
595             return all([result.scheme, result.netloc])
596         except:
597             return False
598
599     def compress_dicts(dict_list: list[dict]) -> Union[dict, list[dict]]:
600         compress = True
601         ref_keys = dict_list[0].keys()
602
603         for dict in dict_list:
604             if not dict.keys() == ref_keys:
605                 compress = False
606                 break
607
608         if compress:
609             compressed = {}
610

```

```

611     for dict in dict_list:
612         for key, value in dict.items():
613             if key not in compressed.keys():
614                 compressed[key] = []
615                 compressed[key].append(value)
616
617     return compressed
618
619 else:
620     return dict_list

```

### GeoScraper/examples/basic\_example.py

```

1 import os
2 from pprint import pprint
3
4 from GeoScraper import GeoScraper
5
6
7 XML_FNAME= os.path.join(os.path.dirname(__file__),
8                     'map.osm')
9
10
11 if __name__ == '__main__':
12     scraper = GeoScraper()
13
14     # print('=' * 50)
15     # print('Using file :')
16     # print('=' * 50)
17     # scraper.from_file(XML_FNAME)
18     # pprint(scraper.highways()[:3])
19     # print('\n')
20
21     print('=' * 50)
22     print('Using_bbox: ')
23     print('=' * 50)
24     scraper.from_bbox(left    = -84.0958000,
25                         bottom   = 39.7617000,
26                         right    = -84.0484000,
27                         top      = 39.7823000)
28     pprint(scraper.highways()[:3])
29     print('\n')
30     print('=' * 50)
31     print('Done')
32     print('=' * 50)

```

### GeoScraper/examples/osm\_to\_kml.py

```

1 import os
2
3 from simplekml import Kml, Color
4 from GeoScraper import GeoScraper

```

```

5
6
7 if __name__ == '__main__':
8     scraper = GeoScraper()
9     scraper.from_bbox(left    = -84.0958000,
10                      bottom   = 39.7617000,
11                      right    = -84.0484000,
12                      top      = 39.7823000)
13
14 KML_FNAME = r'osm_example.kml'
15
16 kml = Kml(name=KML_FNAME, open=1)
17
18 kml_roads      = kml.newmultigeometry(name='roads')
19 kml_power_lines = kml.newmultigeometry(name='power_lines')
20 kml_sub_stations = kml.newmultigeometry(name='sub_stations')
21
22 for road in scraper.highways():
23     kml_roads.newlinestring(coords=road['coords'])
24
25 for power in scraper.power_locations():
26     if 'line' in power['tag_vals']:
27         kml_power_lines.newlinestring(coords=power['coords'])
28
29     elif 'substation' in power['tag_vals']:
30         kml_sub_stations.newpolygon(outerboundaryis=power['coords'])
31
32 kml_roads.style.linestyle.color      = Color.black
33 kml_roads.style.linestyle.width     = 2
34 kml_power_lines.style.linestyle.color = Color.red
35 kml_power_lines.style.linestyle.width = 2
36 kml_sub_stations.style.polystyle.color = Color.orange
37
38 kml.save(KML_FNAME)
39 os.system(KML_FNAME)

```

### GeoScraper/update\_live\_pypi.bat

```

1 python setup.py sdist
2 twine upload dist/*
3 PAUSE

```

### GeoScraper/create\_live\_pypi.bat

```

1 python -m pip install --user --upgrade setuptools wheel
2 python setup.py sdist bdist_wheel
3 python -m pip install --user --upgrade twine
4 python -m twine upload dist/*
5 PAUSE

```

### GeoScraper/create\_test\_pypi.bat

```
1 python -m pip install --user --upgrade setuptools wheel
2 python setup.py sdist bdist_wheel
3 python -m pip install --user --upgrade twine
4 python -m twine upload --repository-url https://test.pypi.org/legacy/ dist/*
5 PAUSE
```

## GeoScraper/LICENSE

```
1 MIT License
2
3 Copyright (c) 2020 PB2
4
5 Permission is hereby granted, free of charge, to any person obtaining a copy
6 of this software and associated documentation files (the "Software"), to deal
7 in the Software without restriction, including without limitation the rights
8 to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
9 copies of the Software, and to permit persons to whom the Software is
10 furnished to do so, subject to the following conditions:
11
12 The above copyright notice and this permission notice shall be included in all
13 copies or substantial portions of the Software.
14
15 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
16 IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
17 FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
18 AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
19 LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
20 OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
21 SOFTWARE.
```

## GeoScraper/README.md

```
1 # GeoScraper
```

## GeoScraper/setup.cfg

```
1 [metadata]
2 description-file = README.md
```

## Bibliography

1. A. Canciani, *Absolute Positioning Using the Earth's Magnetic Anomaly Field*. PhD thesis, Air Force Institute of Technology, 09 2016.
2. R. J. Blakely, *Potential Theory in Gravity and Magnetic Applications*. Cambridge University Press, 1996.
3. E. Thébault, M. Purucker, K. A. Whaler, B. Langlais, and T. J. Sabaka, “The magnetic field of the earth’s lithosphere,” *Space Science Reviews*, vol. 155, pp. 95–127, 07 2010.
4. J. D. Bonifaz, “Magnetic navigation using online calibration filter analysis,” Master’s thesis, Air Force Institute of Technology, 03 2022.
5. C. Reeves, *Aeromagnetic Surveys; Principles, Practice & Interpretation*. Geosoft, 08 2005.
6. Y. Zheng, S. Li, K. Xing, and X. Zhang, “Unmanned aerial vehicles for magnetic surveys: A review on platform selection and interference suppression,” *Drones*, vol. 5, 09 2021.
7. J. Lenz and S. Edelstein, “Magnetic sensors and their applications,” *IEEE Sensors Journal*, vol. 6, pp. 631–649, 06 2006.
8. Geometrics, “G-823a cesium magnetometer data sheet,” 04 2013.
9. M. Munsch, D. Boulanger, P. Ulrich, and M. Bouiflane, “Magnetic mapping for the detection and characterization of uxo: Use of multi-sensor fluxgate 3-axis magnetometers and methods of interpretation,” *Journal of Applied Geophysics*, vol. 61, pp. 168–183, 03 2007.

10. A. Canciani and J. Raquet, “Airborne magnetic anomaly navigation,” *IEEE Transactions on Aerospace and Electronic Systems*, vol. 53, pp. 67–80, 02 2017.
11. G. S. Shenwary, A. H. Kohistany, S. Hussain, S. Ashan, A. S. Mutty, M. A. Daud, M. D. Wussow, R. E. Sweeney, J. D. Phillips, C. R. Lindsay, R. P. Kucks, C. A. Finn, B. J. Drenth, E. D. Anderson, J. D. Abraham, R. T. Liang, J. L. Jarvis, J. M. Gardner, V. A. Childers, D. C. Ball, and J. M. Brozena, “Aeromagnetic surveys in afghanistan: An updated website for distribution of data,” 2011.
12. J.-F. Oehler, D. Rouxel, and M.-F. Lequentrec-Lalancette, “Comparison of global geomagnetic field models and evaluation using marine datasets in the north-eastern atlantic ocean and western mediterranean sea,” *Earth, Planets and Space*, vol. 70, 06 2018.
13. A. Luyendyk, “Processing of airborne magnetic data,” *ASGO Journal of Australian Geology & Geophysics*, vol. 17, pp. 31–38, 1997.
14. T. J. Sabaka, N. Olsen, and R. A. Langel, “A comprehensive model of the quiet-time, near-earth magnetic field: phase 3,” *Geophysical Journal International*, vol. 151, pp. 32–68, 10 2002.
15. W. Tolles, “Magnetic field compensation system,” 04 1955.
16. B. Gavazzi, P. Le Maire, J. Mercier de Lépinay, P. Calou, and M. Munsch, “Fluxgate three-component magnetometers for cost-effective ground, uav and airborne magnetic surveys for industrial and academic geoscience applications and comparison with current industrial standards through case studies,” *Geomechanics for Energy and the Environment*, vol. 20, p. 100117, 12 2019.
17. F. Primdahl, “Temperature compensation of fluxgate magnetometers,” *IEEE Transactions on Magnetics*, vol. 6, pp. 819–822, 12 1970.

18. P. Leliak, “Identification and evaluation of magnetic-field sources of magnetic airborne detector equipped aircraft,” *IRE Transactions on Aeronautical and Navigational Electronics*, vol. ANE-8, pp. 95–105, 09 1961.
19. I. O. Committee and E. Council, “Intermagnet technical reference manual,” 2020.
20. R. Hansen, “Magnetic survey design,” 06 2007.
21. J. C. White and D. Beamish, “Levelling aeromagnetic survey data without the need for tie-lines,” *Geophysical Prospecting*, vol. 63, pp. 451–460, 11 2015.
22. Q. Zhang, C. Peng, Y. Lu, H. Wang, and K. Zhu, “Airborne electromagnetic data levelling using principal component analysis based on flight line difference,” *Journal of Applied Geophysics*, vol. 151, pp. 290–297, 04 2018.
23. T. Ishihara, “A new leveling method without the direct use of crossover data and its application in marine magnetic surveys: weighted spatial averaging and temporal filtering,” *Earth, Planets and Space*, vol. 67, p. 11, 2015.
24. T. L.L.C., “Twinleaf — vmr,” 2023.

## Acronyms

**AFIT** Air Force Institute of Technology. 47, 91

**AGL** above ground level. 29, 56, 57, 58

**ANT** Autonomy and Navigation Technology. 91

**COTS** commercial off-the-shelf. iv

**CSV** comma-separated values. 53, 54, 55, 102

**ERSM** Extended Reference Station Modeling. x, xi, xiv, 43, 59, 61, 62, 63, 66, 67, 68, 88, 89

**EWMA** exponential weighted moving average. 55

**FFT** fast Fourier transform. 58

**FOM** Figure of Merit. ix, xii, xiv, 25, 26, 27, 37, 57, 58, 81, 82

**GNSS** Global Navigation Satellite System. 1, 46, 47, 54

**IGRF** International Geomagnetic Reference Field. 6, 27, 31, 35, 41, 48, 53, 54, 55

**IMU** inertial measurement unit. 91

**INTERMAGNET** International Real-time Magnetic Observatory Network. ix, x, 5, 27, 28, 34, 37, 42, 43, 59, 60, 61, 62, 63, 66

**KML** Keyhole Markup Language. 101

**LCM** Lightweight Communications and Marshalling. 53

**MagNav** magnetic navigation. iv, v, 1, 2, 3, 4, 5, 12, 13, 29, 33, 43, 63, 64, 87, 89, 91, 99

**MAMMAL** Magnetic Anomaly Map Making for Air and Land. 3, 87

**MSL** mean sea level. 35, 39, 50, 101, 102, 105

**PCA** principal component analysis. xiii, 31, 33, 56, 80, 81, 82, 93, 94, 95, 96

**RF** Radio Frequency. 11, 18

**RMSE** Root Mean Square Error. 37, 43, 49, 59, 61, 64, 66, 67, 70, 73, 74

**UAV** unmanned aerial vehicle. ix, x, xiv, 3, 4, 11, 12, 13, 14, 34, 44, 45, 47, 48, 49, 50, 51, 52, 53, 54, 72, 89, 90, 92

**US** United States. 1

**USGS** United States Geological Survey. viii, 9, 34

**UTC** Coordinated Universal Time. 54, 102

**WDMAM** World Digital Magnetic Anomaly Map. viii, 7

**WMM** World Magnetic Model. 6, 27

# REPORT DOCUMENTATION PAGE

*Form Approved  
OMB No. 0704-0188*

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

<b>1. REPORT DATE (DD-MM-YYYY)</b> 23-03-2023	<b>2. REPORT TYPE</b> Master's Thesis	<b>3. DATES COVERED (From — To)</b> Sept 2021 — Mar 2023		
<b>4. TITLE AND SUBTITLE</b>  Magnetic Anomaly Mapping for Navigation		<b>5a. CONTRACT NUMBER</b>  <b>5b. GRANT NUMBER</b>  <b>5c. PROGRAM ELEMENT NUMBER</b>		
<b>6. AUTHOR(S)</b>  Bergeron, Luke T., Capt		<b>5d. PROJECT NUMBER</b>  <b>5e. TASK NUMBER</b>  <b>5f. WORK UNIT NUMBER</b>		
<b>7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)</b>  Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way WPAFB OH 45433-7765		<b>8. PERFORMING ORGANIZATION REPORT NUMBER</b>  AFIT-ENG-MS-23-M-010		
<b>9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)</b>  National Geospatial-Intelligence Agency Research Directorate Mail Stop N74-IB 7500 GEOINT Drive Springfield, Virginia 22150 COMM 571-557-8547 Email: Nikolaos.K.Pavlis@nga.mil		<b>10. SPONSOR/MONITOR'S ACRONYM(S)</b>  NGA		
<b>12. DISTRIBUTION / AVAILABILITY STATEMENT</b>  Distribution Statement A. Approved for Public Release; Distribution Unlimited.		<b>11. SPONSOR/MONITOR'S REPORT NUMBER(S)</b>		
<b>13. SUPPLEMENTARY NOTES</b>				
<b>14. ABSTRACT</b>  Magnetic navigation (MagNav) has the potential to provide a global form of navigation that uses magnetic measurements of the Earth's anomaly magnetic field and compares those measurements to a magnetic anomaly map in order to determine the user's position. Widespread use of MagNav will require a database of fully-sampled, low-altitude magnetic anomaly maps. Existing magnetic anomaly map databases usually come from under- or poorly-sampled surveys. In this work, we provide an easy to follow MagNav anomaly map generation framework and set of survey collection metrics/requirements in an effort to help facilitate and standardize the creation of such a database. We further explore the necessary equipment, sensors, and software algorithms required to conduct a magnetic anomaly survey and generate a map for use with MagNav by generating maps from both simulated and real-world aeromagnetic surveys.				
<b>15. SUBJECT TERMS</b>  magnetic anomaly, magnetic navigation, MagNav, aeromagnetic survey, map-making, mapping				
<b>16. SECURITY CLASSIFICATION OF:</b>  <b>a. REPORT</b> U <b>b. ABSTRACT</b> U <b>c. THIS PAGE</b> U		<b>17. LIMITATION OF ABSTRACT</b>  UU	<b>18. NUMBER OF PAGES</b>  576	<b>19a. NAME OF RESPONSIBLE PERSON</b> Dr. Aaron P. Nielsen AFIT/ENG  <b>19b. TELEPHONE NUMBER (include area code)</b> (937) 255-3636 x4583 Aaron.Nielsen@afit.edu