University of Stavanger

Faculty of Science and Technology

MASTER'S THESIS

Study program/ Specialization: **Computer Science**	Spring semester, **2015** Open / Restricted access
Writer: **Kristian S. Stangeland**	…… (Writer's signature)
Faculty supervisor: **Tom Ryen** External supervisor(s): **Åge Andersen**	
Thesis title: **Positioning in Electromagnetic Fields**	
Credits (ECTS): **30**	
Key words: **Drone, UAV, Overhead Power Line, Automatic Inspection, Magnetic Field Positioning, FFT, CMA-ES**	Pages: ……**82**.…… + enclosure: …**3**…… Stavanger, …**15.06.2015**…….. Date/year

Front page for master thesis
Faculty of Science and Technology
Decision made by the Dean October 30th 2009

Positioning in Electromagnetic Fields

Kristian S. Stangeland

June 15, 2015

Abstract

Drones (or Unmanned Aerial Vehicles) can become an important tool for performing the regular inspection of overhead power lines in difficult terrain. This currently requires a pilot, but if we could detect the position of the power line using magnetic field sensors, we might be able to automatically navigate alongside it. This would also be useful in remotely measuring the amount of current flowing through the conductors.

We then focus on the magnetic field around a three-phase single-circuit power line. Using either a mathematical model, or a FFT (Fast-Fourier Transform) of 8 samples of the field, we compute two values per component that are invariant to the unknown current, but depend on the location of the drone. Using CMA-ES (Covariance Matrix Adaptation Evolution Strategy), we estimate the corresponding point given at least two sensors and four invariants per sensor. As suggested by experimental data, we also consider the possibility that each conductor is carrying a different amount of current.

Using this approach, we successfully navigate a simulated drone using three magnetic field sensors, where two is the minimum. The estimated position is accurate down to about 1.8 mm, needing about 350 ms of computational time on a desktop computer.

This enables us to position a drone with better accuracy than using a GPS alone. We can also use this method to cancel drift in an INS (Inertial Navigation System).

Acknowledgements

I would like to thank to Verico their interest and support in this thesis, and for allowing me time off to focus all of my attention on this paper.

I'd like to especially thank Åge Andersen for his invaluable technical expertise, and Tom Ryen for his continued support and guidance. I'd also like to thank Arild Kramme Berstad in Statnett for supplying accurate magnetic field simulations in exploratory phase of the thesis.

Contents

List of Figures **4**

List of Tables **6**

1 Introduction **7**
1.1 Background . . . 7
1.2 Motivation . . . 7
1.2.1 Problem Statement . . . 7
1.3 Related Work . . . 8
1.3.1 Mechanical Solution . . . 8
1.4 Report Outline . . . 9

2 Magnetic Field of a Transmission Line **10**
2.1 Overview . . . 10
2.2 Magnetic Field of a Single Wire . . . 10
2.2.1 Cartesian Representation . . . 11
2.2.2 Polar Representation . . . 12
2.2.3 Magnetic Field Inside a Wire . . . 13
2.3 Magnetic Field of Multiple Wires . . . 13
2.3.1 Alternating Current in Power Lines . . . 15
2.3.2 Changing Magnetic Field . . . 16
2.3.3 Sinusoidal Vector Sum . . . 19

3 Drone Positioning **22**
3.1 Positioning with a Single Sensor . . . 22
3.1.1 Amplitude of the Magnetic Field . . . 22
3.1.2 Phase of the Magnetic Field . . . 24
3.1.3 Algebraic Solution? . . . 26
3.2 Positioning with Multiple Sensors . . . 27
3.3 Determine Orientation . . . 28
3.3.1 Rotation of a Rigid Body . . . 29
3.3.2 Drone Rotation . . . 30
3.3.3 GPS Assisted Orientation . . . 30
3.3.4 Magnetic Field Orientation . . . 30
3.4 Phase Sequence . . . 32
3.4.1 Effect on Field Invariants . . . 33
3.5 Real-World Considerations . . . 33
3.5.1 Frequency Deviations . . . 34
3.5.2 Current Imbalance . . . 34
3.6 Estimating Conductor Currents . . . 36

4 Computer Simulation **38**
4.1 Implementation in Mathematica . . . 38
4.2 Overview of Java Implementation . . . 40
4.2.1 Build Tool . . . 40
4.2.2 Command Line Interface . . . 40
4.2.3 Logarithm of Both Positive and Negative Values . . . 41
4.2.4 Plotting TSV data . . . 43
4.3 Code Abstractions . . . 43
4.3.1 Scalar Field Abstractions . . . 44
4.3.2 Vector Field Abstractions . . . 46
4.3.3 Positioning Abstractions . . . 48
4.3.4 Power Line Model . . . 50
4.3.5 Plotting System . . . 50

5 Algorithms **55**
5.1 Fast-Fourier Transform . . . 55
5.2 Covariance Matrix Adaptation Evolution Strategy . . . 56
5.2.1 Initial Parameters . . . 57
5.2.2 Main Iteration . . . 57

6 Results **60**
6.1 Single Sensor . . . 60
6.1.1 Evaluating BOBYQA and CMA-ES . . . 60
6.1.2 Points of High Inaccuracy . . . 62
6.2 Multiple Sensors . . . 62
6.2.1 Problematic Points . . . 63
6.3 Improving Accuracy . . . 63
6.3.1 Select the Lowest Invariant Distance . . . 63
6.3.2 Restricting the Search Area . . . 64
6.3.3 Virtual Sensors . . . 64
6.4 Simulating Navigation . . . 66
6.4.1 Proposed Algorithm . . . 67
6.4.2 Effect of Frequency Deviation . . . 68
6.4.3 Effect of Current Imbalance . . . 69

7 Conclusion **72**

8 Future Work **73**

A Code Listings **74**
A.1 Mathematica . . . 74
A.2 Java . . . 75
A.2.1 Field Interfaces . . . 75
A.3 PlotCSV Commands . . . 78
A.4 Attached Files . . . 78

B Bibliography **81**

List of Figures

2.1 Example power line with three conductors 11
2.2 Magnetic field of an infinitely long wire 11
2.3 Polar Representation of **B**, with respect to **r**. 13
2.4 Magnitude (in Tesla) of a magnetic field around a wire 14
2.5 Magnitude (in Tesla) of a magnetic field around three wires 15
2.6 The current of the wires in a three-phase system 16
2.7 The magnetic field in the x-axis experienced by point (0, 30). 17
2.8 The magnetic field in the y-axis experienced by point (0, 30). 17
2.9 Magnitude of the magnetic field 1/4 into the 50 Hertz cycle 18
2.10 Direction of the magnetic field 1/4 into the 50 Hertz cycle (1/200 s). . . 19

3.1 Amplitudes of each sinusoidal component in the magnetic field 23
3.2 Weekly variation in average power consumption in Norway 23
3.3 Magnetic field vector over a reflected point 25
3.4 The phases associated with each sinusoidal component in a magnetic field 26
3.5 Summation of the amplitudes as vectors. 27
3.6 The sinusoidal phase in x subtracted from the one in y 28
3.7 The pitch, yaw and roll that define an arbitrary rotation of a drone. . . 29
3.8 A selection of orthogonal vectors in $\mathbb{R}^3$ 31
3.9 Effect of rotating a reference frame around a static environment 32
3.10 A transposition scheme applied to a power line. 33
3.11 The phase difference when the phase sequence is negative. The color scale goes from 0 to 2π radians. 34
3.12 The effect of changing the system frequency 35
3.13 The current in three conductors over time 36

4.1 The inaccuracy of listing 4.1, in meters. 39
4.2 Comparison of the log-modulus transformation 43
4.3 The magnetic field data provided by Statnett 44
4.4 Data structure used by *fillComponents* 48
4.5 Log-modulus plot of the x and y components of the magnetic field . . . 49
4.6 Plot of the angle difference of the second and fourth component 49
4.7 Mapping of the red, green and blue color channel in *RainbowGradient* . 51

5.1 Convergence to the global optimum using the CMA-ES algorithm 56

6.1 Visualization of both the phase difference and the amplitude angle . . . 62
6.2 Plot of miss distances when using CMAES and the two field invariants. 64
6.3 The miss distance contra the invariant distance 66
6.4 Position of the drone, and the estimated position. 68

6.5 Distance between the estimated position and the true position 69
6.6 Comparison of attempting to navigate in a field with a high current imbalance (HCD), using $a = b = 1$, or by allowing CMA-ES to estimate a and b. 71

List of Tables

2.1 Position and current of each wire. 18

3.1 Sequences of phase offsets in a three-phase system 33
3.2 Average and SD (Standard Deviation) of the current in the three conductors (L1, L2 and L3). 35
3.3 Times when the SD of the conductor currents at a instant is lowest or highest. 35

4.1 Table of common flags in our CLI. 41
4.2 Table of flags for customizing the magnetic field generator. 42

6.1 Testing BOBYQA and CMAES with 10000 random points 61
6.2 The ten most inaccurate results in a CMAES run 65
6.3 The effect of adding additional sensor inputs with known relative offsets. 65
6.4 Selecting the lowest invariant distance over repeat runs 66
6.5 Accuracy after restricting the search area 67
6.6 The accuracy of virtual sensors with a set repeat count 67
6.7 Effect of Frequency Deviation 69
6.8 Current Imbalance Tests 70
6.9 Including a and b as dimensions in the search space 70

A.1 Assorted Commands 79

Chapter 1

Introduction

1.1 Background

In recent years, we have seen a rise in the application of drones (unmanned aerial vehicle) to perform tasks that would otherwise be too costly, dangerous or boring for humans to perform. Early development was primarily spearheaded by the military (and the government), much like a great deal of other novel technologies, but now commercial interests have also begun to follow suit [9].

One such application is visual inspection of power lines to detect signs of early failure, or the encroachment of growing vegetation. This must be performed regularly, at least once a year. Complicating matters further, many sections are located in rugged or mountainous terrain that can only be accessed by foot or helicopter (dangerous in bad weather), at considerable cost. However, recently a number of electric utilities have invested in drone technology to augment their inspection process. The drones are typically equipped with a live video feed and a GPS, allowing a pilot to operate them remotely (RPA).

1.2 Motivation

Unfortunately, the accuracy of GPS is variable, usually no better than approximately 7 meters horizontally, and certainly not ideal in the vertical direction. If we can use the electromagnetic field for positioning, it might be better at keeping the drone centred over the power line and ensuring the drone is at a safe distance from the live conductors. There are also many cheap and accurate magnetic field sensors small enough to be installed on a drone (see [18]).

It might even be theoretically possible to extract more than just positional data from the magnetic field, such as the amount of current flowing through each wire. This kind of data might prove useful during an inspection, but it could just as well be implemented as a mobile application for human inspectors.

1.2.1 Problem Statement

Given a mathematical model or a data set of the electromagnetic field emanating from a power line, is it possible estimate the current position of an arbitrary point within range by using direct measurements of the field in nearby points?

- What is the minimum number of sensors needed get valid results? What is their optimal placement on a drone?

- What kind of positional information and at what accuracy is it possible to achieve in this manner?
- Given the characteristics of electromagnetic sensors what are the minimum distance between sensors to get valid results. What kind of computer algorithms can be used for recognizing the patterns of the electromagnetic fields?

1.3 Related Work

The concept of using a changing or static magnetic field for positioning is not new. We will give a short run down of the systems currently in development or in production that use magnetic field for positioning, particularly those that rely on existing magnetic fields.

As an example of a recent commercial effort, the Finnish start-up IndoorAtlas has developed an indoor navigation system based solely on local distortions of Earth's magnetic field, which are unique but static in a specific indoor location, and a function of the permeability of the different materials surrounding it. By surveying these static distortions in advance, the company claims to achieve an accuracy less than 3 meters using only consumer grade sensors on smartphones [3].

There is also a theoretical and experimental study of measuring the magnetic field of low-voltage power lines to estimate the position and orientation of a bird-scale drone, in order to potentially move within close range and extract power from the electric field using induction [18]. Using cheap low-weight hardware, the authors performed a rudimentary test on a magnetic field produced by a wire loop to verify the feasibility of the concept in practice.

In simulations, they were successful in tracking a drone cruising at 8 m/s at a distance of 4 meters from the power line, though its unclear how accurate this would be in practice. They did not, however, extend their simulation to three conductors, or consider the implication of current imbalance in the conductors, as we shall see in section 6.4.3.

Unrelated to their work, there is also a patent in the US covering a system of estimating the position and orientation of a drone, and the amount of current carried by the power line [27].

1.3.1 Mechanical Solution

An alternative approach to automate power line inspection, would be to construct a robot that is capable of physically traversing the conductors or the earth wire themselves, and then somehow bypass the suspender clamps in each transmission tower [17]. Expliner [7] is one intriguing example of such a solution, which uses movable wheel axles to drive along a bundle of conductors.

Restricting the degrees of freedom to one dimension is an appealing simplification, though one should first evaluate any potential risks associated with permitting a heavy semi-automated drone physical access to the structure of the power line. Granted, the complexity of navigating a drone might incur a greater chance of failure, but that must be weighted against the amount of damage it can actually cause.

Keep also in mind that the design of a line walking robot may not general enough to traverse any type of power line, unlike a drone, nor can it position the sensors and cameras at an arbitrary nearby point to capture the best possible data.

1.4 Report Outline

We will first develop the mathematical foundations for computing a static magnetic field generated by the current of multiple wires in chapter 2. Next, we introduce alternating current (assuming a low frequency), and show that this results in a sinusoidal magnetic field components with equations for computing their amplitude and phase.

In chapter 3, we transform these sinusoids into invariants that are independent on the conductor currents, allowing us to look up the current position in the magnetic field based on measured invariants. The drone orientation is found using the vector cross product of two magnetic field vectors from the same point. Finally, we discuss potential real-world sources of inaccuracies in the final algorithm, such as frequency deviation or current imbalance.

The algorithms needed to extract the sinusoid from the magnetic field and looking up the corresponding point in the model, are defined in chapter 5. We then delve into the Java implementation of the magnetic field model, the plotting and finally, the positioning algorithm.

Finally, chapter 6 presents the accuracy of the positioning algorithm.

Chapter 2

Magnetic Field of a Transmission Line

2.1 Overview

Consider an overhead three-phase power line with three conductors, as illustrated in figure 2.1. To approximate its magnetic field, we first disregard the presence of each metallic tower structure and the earth wires along the top of the power line. Then the magnetic fields of each conducting wire is considered separately, before taking the total vector sum using the principle of superposition [20].

When suspended between multiple towers, each conductor is a sequence of straight wire segments when viewed from above [12]. Though these segments make parabolas of variable height (depending on temperature) relative to the ground, they are nevertheless at a fairly shallow angle. Over distance, the vertical displacement can be significant, but this will only aid our goal of navigating along the power line at a constant distance.

At the scale we are studying ($< 8m$, or the accuracy of GPS [2]), the wire segments just reduce to an infinitely long straight wire. This is the model we will use for the magnetic field.

2.2 Magnetic Field of a Single Wire

Given an infinite straight wire parallel to the z-axis in $\mathbb{R}^3$, any conducting current will generate a magnetic vector field. If the current is constant (DC), the magnetic flux density ($\mathbf{B}$) experienced by a point $\mathbf{p}$ can be calculated using the Biot-Savart law [**?**]:

$$\mathbf{B}(\mathbf{r}) = \frac{\mu_o I_w \ \hat{\mathbf{z}} \times (\mathbf{r} - \mathbf{r_w})}{2\pi \left|\mathbf{r} - \mathbf{r_w}\right|^2} \tag{2.1}$$

The derivation of this expression will be skipped, as it is well known and covered in most elementary introductions to the subject.

In 2.1, μ_o is the magnetic permeability of free space ($4\pi\, 10^{-7}\ {}^{\mathrm{N}}/_{A^2}$), I_w is the constant current conducted by the wire, and $\hat{\mathbf{z}}$ is a unit vector in the direction of the conducted current (parallel to the z-axis). Projecting $\mathbf{p}$ onto the x–y plane yields position vector $\mathbf{r}$, and vector $\mathbf{r_w}$ is the point where the wire and the x–y plane intersects. Lastly, the $\times$ operator is the cross product. Let us define this explicitly:

$$\mathbf{r} = \mathbf{p_x}\hat{x} + \mathbf{p_y}\hat{y} \tag{2.2}$$

$$\hat{\mathbf{z}} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T \tag{2.3}$$

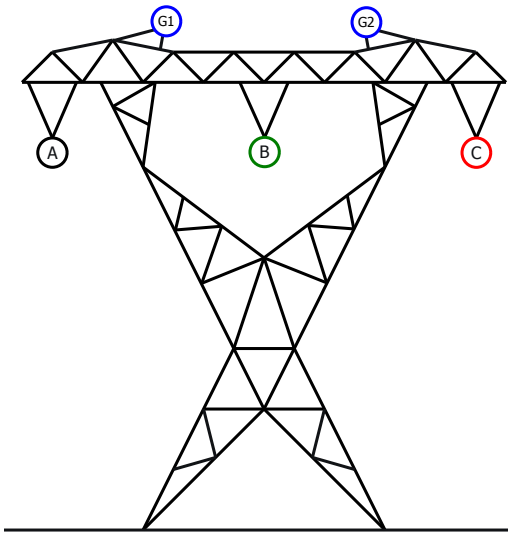


Figure 2.1: Example of a power line construction containing three conductors (A, B, C) in a three-phase system, with two overhead earth wires (G1 and G2) at the top. The wires travel through the plane of this paper.

Using the right-hand rule, we determine that the field lines of the resulting magnetic field are consecutive circles centred on the wire (see figure 2.2), where the magnetic flux density decreases quadratically by the radius of the circle. Note that we can ignore the

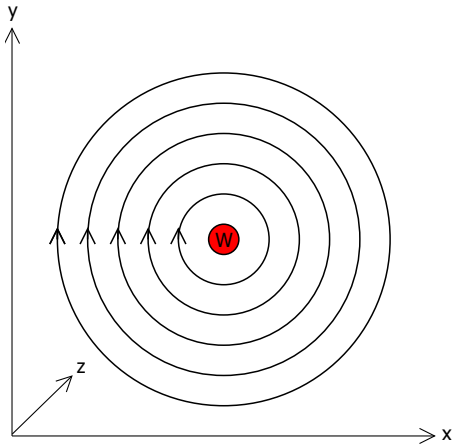


Figure 2.2: Magnetic field of an infinitely long wire (W) parallel to the z-axis. The current is flowing in positive z direction.

magnetic permeability of air, as it almost identical to the permeability of free space ($u = 1.00000037\, u_0$).

2.2.1 Cartesian Representation

Next, we can determine the exact components of $\mathbf{B}$ by evaluating the cross product of $\hat{\mathbf{z}}$ and the distance vector $\mathbf{r} - \mathbf{r_w}$. Also note the definition of the cross product in terms

of its components [5]:

$$\mathbf{a} \times \mathbf{b} = (a_y b_z - a_z b_y)\hat{\mathbf{x}} + (a_z b_x - a_x b_z)\hat{\mathbf{y}} + (a_x b_y - a_y b_x)\hat{\mathbf{z}} \tag{2.4}$$

Applying the definition in 2.4 to the last part of the nominator in 2.1, we find:

$$\hat{\mathbf{z}} \times (\mathbf{r} - \mathbf{r_w}) = \hat{\mathbf{z}} \times [(r_x - r_w x)\hat{\mathbf{x}} + (r_y - r_w y)\hat{\mathbf{y}}] \tag{2.5}$$

$$= (r_y - r_w y)\hat{\mathbf{x}} - (r_x - r_w x)\hat{\mathbf{y}} \tag{2.6}$$

$$= \begin{bmatrix} r_y - r_w y \\ -(r_x - r_w x) \\ 0 \end{bmatrix} \tag{2.7}$$

To simplify the final expression, we define the distance vector $\mathbf{d}$, which is the distance between the field point we are computing, and the center of the wire:

$$\mathbf{d} = \mathbf{r} - \mathbf{r_w} = \begin{bmatrix} r_x - r_w x \\ r_y - r_w y \\ 0 \end{bmatrix} \tag{2.8}$$

Combining 2.8 and 2.7 with 2.1, we end up with the following expressions for the magnetic field density in the x-axis and y-axis:

$$B_x = \frac{\mu_o I_w \ d_y}{2\pi \left(d_x^2 + d_y^2\right)} \tag{2.9}$$

$$B_y = -\frac{\mu_o I_w \ d_x}{2\pi \left(d_x^2 + d_y^2\right)} \tag{2.10}$$

This formulation will be particularly useful when we need to compute the magnetic field efficiently, as it only involves simple algebraic operations.

2.2.2 Polar Representation

The magnetic field equation in 2.1 can also be expressed in a polar coordinate system, using a similar approach.

But, we will use the conventional definition of the cross product this time:

$$\mathbf{a} \times \mathbf{b} = |\mathbf{a}||\mathbf{b}| \sin\theta \ \hat{\mathbf{n}} \tag{2.11}$$

Here $|\mathbf{a}|$ and $|\mathbf{b}|$ is the magnitude of the vectors, θ is the angle between the vectors, and $\hat{\mathbf{n}}$ is a unit-vector orthogonal to both vectors (as specified by the right-hand rule).

We then proceed like in equation 2.7, except using defintion 2.11. Due to the right-hand rule, the angle of $\hat{\mathbf{n}}$ must be in a 90° clockwise rotation off from $\mathbf{d}$, which makes it a tangent on the magnetic field line. This is illustrated in figure 2.3.

The angle between $\mathbf{d}$ and $\hat{\mathbf{z}}$ is always 90°, so the magnitude of $|B|$ reduces to:

$$|\mathbf{B}| = \left| \frac{\mu_o I_w \ \hat{\mathbf{z}} \times (\mathbf{r} - \mathbf{r_w})}{2\pi \left|\mathbf{r} - \mathbf{r_w}\right|^2} \right| \tag{2.12}$$

$$= \frac{\mu_o I_w \ \cancel{|\hat{\mathbf{z}}|}\,|\mathbf{d}|\cancel{\sin 90^\circ}\,\cancel{|\hat{\mathbf{n}}|}}{2\pi \left|\mathbf{d}\right|^2} \tag{2.13}$$

$$= \frac{\mu_o I_w \ \cancel{|\mathbf{d}|}}{2\pi \left|\mathbf{d}\right|^{\cancel{2}}} \tag{2.14}$$

$$= \frac{\mu_o I_w}{2\pi \left|\mathbf{d}\right|} \tag{2.15}$$

Finally, the angle of $\mathbf{B}$, $\mathbf{B}_\theta$, is the angle of the distance vector $\mathbf{d}$ rotated 90° clockwise:

$$\mathbf{B}_\theta = \mathbf{d}_\theta - 90^\circ \tag{2.16}$$

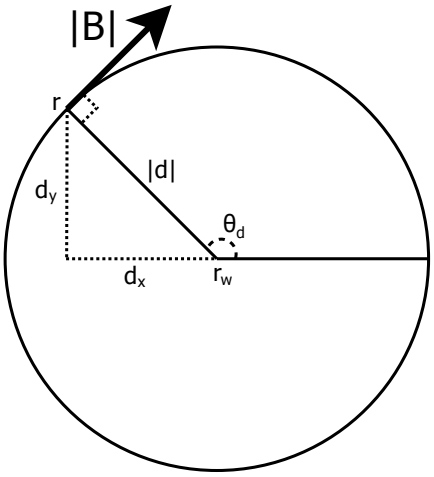


Figure 2.3: Polar Representation of **B**, with respect to **r**.

2.2.3 Magnetic Field Inside a Wire

As a side note, although the magnetic field as defined by 2.15 does suggest the magnitude of the field tends towards infinity as the distance to the wire center r_w approaches zero, this does not apply to wires in reality. Unlike our wire point w, they possess a non-zero radius where the current spreads evenly throughout the cross section (assuming DC):

$$\lim_{|\mathbf{d}|\to 0} \frac{\mu_o I_w}{2\pi\,|\mathbf{d}|} = +\infty$$

Using Ampere's law, which can be derived from the Biot-Savart law, one can show that the magnetic field inside a wire of radius R decreases linearly from the wire surface to the inner core [16]:

$$|\mathbf{B}| = \begin{cases} \dfrac{\mu_o I_w}{2\pi\,R^2}|\mathbf{d}|, & \text{if } |\mathbf{d}| \le R \\ \dfrac{\mu_o I_w}{2\pi\,|\mathbf{d}|}, & \text{otherwise} \end{cases} \tag{2.17}$$

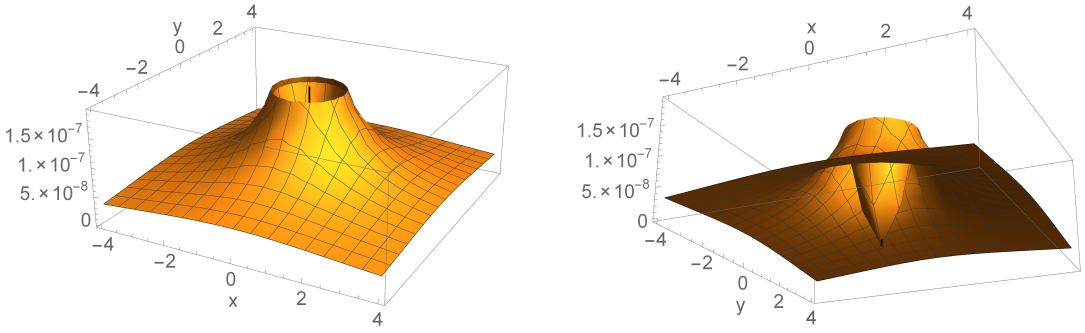
This result can be useful if we need to avoid singularities in our model, though we typically don't need to know the exact magnitudes in this case, as the drone will never be in a position to measure the inside of a wire.

The magnitude of the magnetic field around a wire travelling through z is illustrated in figure 2.4.

2.3 Magnetic Field of Multiple Wires

In order to determine the combined magnetic field of the wires in our example (2.1), we apply the principle of superposition [20] and perform a vector addition of the contributions from each wire. Note that we ignore any optional neutral wires, as the voltage sum of the three conductors usually is zero, meaning no current will flow through neutral.

Let us begin by extending the definition of **B** (2.1) with this in mind. Assume that $\mathbf{w}_i$ and $\mathbf{I_i}$ is the position and current of the i-th wire respectively, given n wires in total.



(a) Plot from above

(b) Plot from below

Figure 2.4: Magnitude (in Tesla) of a magnetic field around a wire whose center is at (0, 0) (black line) and has a radius of 1 unit.

Furthermore, $\hat{z}_i$ is a unit vector representing the direction of the current, which in our example is identical for all wires. Then we end up with the following expression:

$$\mathbf{B_n}(\mathbf{r}) = \sum_{i=1}^{n} \frac{\mu_o I_i \ \hat{\mathbf{z}}_\mathbf{i} \times (\mathbf{r} - \mathbf{r_i})}{2\pi \left|\mathbf{r} - \mathbf{r_i}\right|^2} \tag{2.18}$$

Similarly, we parametrize the distance vector $\mathbf{d}$:

$$\mathbf{d}(\mathbf{i}) = \mathbf{r} - \mathbf{r_i} \tag{2.19}$$

And the Cartesian representation in 2.9 and 2.10:

$$B_{nx} = \sum_{i=1}^{n} \frac{\mu_o I_i \ d(i)_y}{2\pi \left(d(i)_x^2 + d(i)_y^2\right)} \tag{2.20}$$

$$B_{ny} = -\sum_{i=1}^{n} \frac{\mu_o I_i \ d(i)_x}{2\pi \left(d(i)_x^2 + d(i)_y^2\right)} \tag{2.21}$$

Unfortunately, there is no such simple extension for polar coordinates, so we're forced to base it on 2.20 and 2.21 instead

$$|\mathbf{B_n}| = \sqrt{B_{nx}^2 + B_{ny}^2} \tag{2.22}$$

$$B_{n\theta} = atan2(B_{ny}, B_{nx}) \tag{2.23}$$

Here, $atan2(y, x)$ is a modified version of $\tan^{-1}$ that computes the angle of a vector $\begin{bmatrix} x & y \end{bmatrix}$ in the same quadrant as the vector. The following definition assumes all angles are expressed in radians:

$$\text{atan2}(y, x) = \begin{cases} \tan^{-1}\frac{y}{x} & x > 0 \\ \tan^{-1}\frac{y}{x} + \pi & y \geq 0, x < 0 \\ \tan^{-1}\frac{y}{x} - \pi & y < 0, x < 0 \\ +\frac{\pi}{2} & y > 0, x = 0 \\ -\frac{\pi}{2} & y < 0, x = 0 \\ \text{undefined} & y = 0, x = 0 \end{cases} \tag{2.24}$$

To illustrate the effect of this vector sum, consider figure 2.5. It displays the magnitude of the total magnetic field resulting from three conductors at (-8, 0), (0, 0) and (8, 0), each having a radius of 1 unit, and conducting 1 Ampers of current.

Note the two valleys between the middle conductor at (0, 0), and the two conductors at (-8, 0) and (8, 0) respectively. This is because the direction of the magnetic field of a wire in a point to the left is exactly opposite of a mirrored point to the right (see figure 2.2), and thus the magnetic fields cancel out.

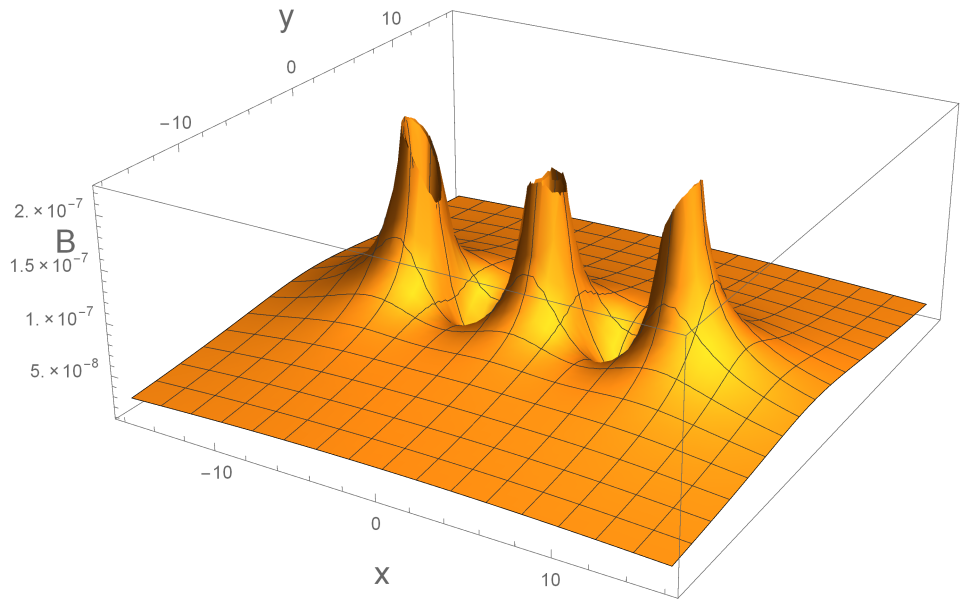


Figure 2.5: Magnitude (in Tesla) of a magnetic field around three wires whose center is at (-8, 0), (0, 0) and (0, 0) - and all have a radius of 1 unit.

2.3.1 Alternating Current in Power Lines

Up until this point, we have restricted our view to magnetic fields generated by the flow of direct current (DC), yet we're dealing with alternating current in reality. Both the classical formulation of Ampere's circuital law and the Biot-Savart law presupposes a steady flow current, and may produce incorrect results if this assumption is violated.

Maxwell's extension of Ampere's law [16] solves this limitation by adding a new quantity - the displacement current, which is related to the rate of change of the electric displacement field.

Quasi-Static Approximation

However, by taking into account the low frequency of the alternating current (50/60 Hz), and thus rate of change of the electric field, we can find an approximate distance where the contribution of the displacement current can be ignored [20]:

$$|d| \ll \delta = 503\sqrt{\frac{p_g}{f}} \tag{2.25}$$

Here, p_g is the earth resistivity in Ωm, and f is the frequency of the current in Hertz. Typical values of p_g reside in the range of 10 - 1000 Ωm, which suggests δ is in the range of 223 m to 2249 m (assuming f is 50 Hertz).

Thus, as long as the maximum distance to a conductor is much less than 223 m, which we expect as the GPS will be used beyond 20 meters, the displacement current can be justifiably ignored and we're left with Ampere's original formulation (the magnetostatic part). This simplification is also known as the quasi-static approximation.

Modelling

Per the quasi-static approximation, we assume the system is in equilibrium at all times, and simply calculate the magnetic field as a function of a sinusoidal current:

$$I_i = A_i \sin(2\pi f t + \varphi_i) \tag{2.26}$$

Here, f is the aforementioned 50 Hertz frequency, t is time in seconds and φ_i is the phase-offset specific to the current wire.

Most modern electrical grids are based on three-phase alternating current in order to achieve balanced loads and constant power transfer. This requires three conductors, each offset at 0°, 120° and 240° respectively:

$$\begin{aligned} I_1 &= I_p \sin(\omega t + 0°) \\ I_2 &= I_p \sin(\omega t + 120°) \\ I_3 &= I_p \sin(\omega t + 240°) \end{aligned}$$

This is illustrated in figure 2.6. The angular frequency ω is $2\pi f$ radians per second, or $360f$ degrees per second - which in the case of 50 Hertz is expected to be 18000 degrees per second (barring any deviations due to imbalance in power generation and the load). The shared amplitude I_p is the peak current of the power line, which can deviate depending on the overall load of the connected grid.

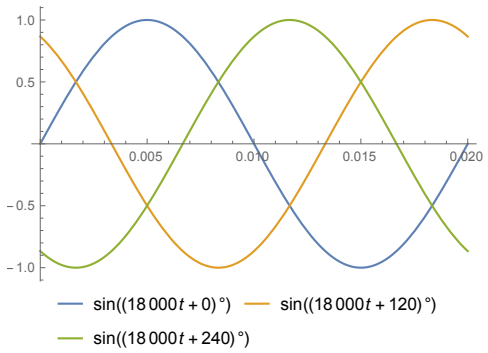


Figure 2.6: The current of three wires (blue, yellow, green) in a three-phase system over 1/50 of a second, assuming I_p is 1 A.

2.3.2 Changing Magnetic Field

The sinusoidal currents causes the overall magnetic field to change in both magnitude and direction over a single cycle ($\frac{1}{50}$ second).

In order to get a better understanding just how the field changes, let us re-examine the example in 2.1 and specify the position and currents of each conductor in $x-y$ space, and let the radius of each wire be 0.04 meters. See table 2.1 for the exact positions.

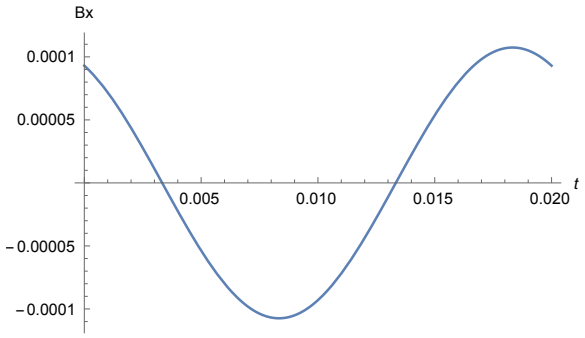


Figure 2.7: The magnetic field in the x-axis experienced by point (0, 30).

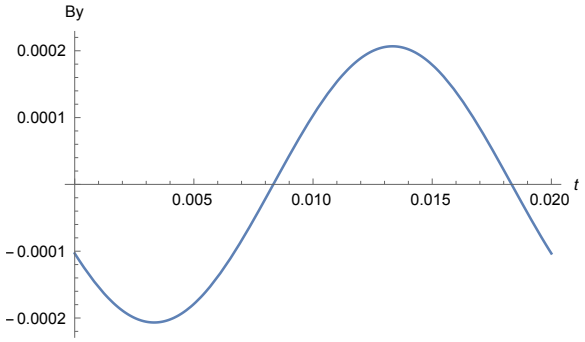


Figure 2.8: The magnetic field in the y-axis experienced by point (0, 30).

Next, pick the point (0, 30), which is 10 meters above wire B, for our investigation. Substituting the current for our changing sinusoid, we can plot the magnetic field in the x- and y-axis, as seen in figure 2.7 and 2.8.
It is very evident from these figures that we are dealing with sinusoids, an observation that will be substantiated in the next section.

As this is how the magnetic field appears to a drone sensor in a given point, it must contain some information about the current position if our task is achievable. The features we extract from the sinusoid should also be independent of the time we started to measure the magnetic field, as we cannot predict the exact phase of the current in each wire, only that they differ by 120°.

Figure 2.9 and figure 2.10, which is a plot of the magnitude and direction of the magnetic field from (-20, 0) to (20, 40) at 0.005 s, show just how the magnetic field changes throughout the target area. To see an animation of the magnetic field over a full AC period, please refer to the **YouTube link** in reference [25], or link [24].

The magnitude is shaped much like in figure 2.5, especially near the wire centres with the same recurring tall peaks. However, as evident by figure 2.6, the current of the left-most wire is flowing in the opposite direction to the two other wires at $t = 0.005$, causing the area in-between to plateau instead of turning into a deep valley. This is because the inverse direction of the current results in magnetic field lines (see 2.2) going counter-clockwise instead of clockwise, ensuring that vectors on opposite sides of two wires contribute instead of cancelling each other out.

Figure 2.10 displays the angle of the magnetic field as a hue in the color spectrum, going from 0 to 2π radians. The hue ensures that the transition between 0 and 2π is not a sharp divide, unlike a simple RGB scale.

Wire	X Position	Y Position	Wire Radius	Current [A]
A	-9 m	20 m	0.04 m	$12000\sin(18000t + 0°)$
B	0 m	20 m	0.04 m	$12000\sin(18000t + 120°)$
C	9 m	20 m	0.04 m	$12000\sin(18000t + 240°)$

Table 2.1: Position and current of each wire.

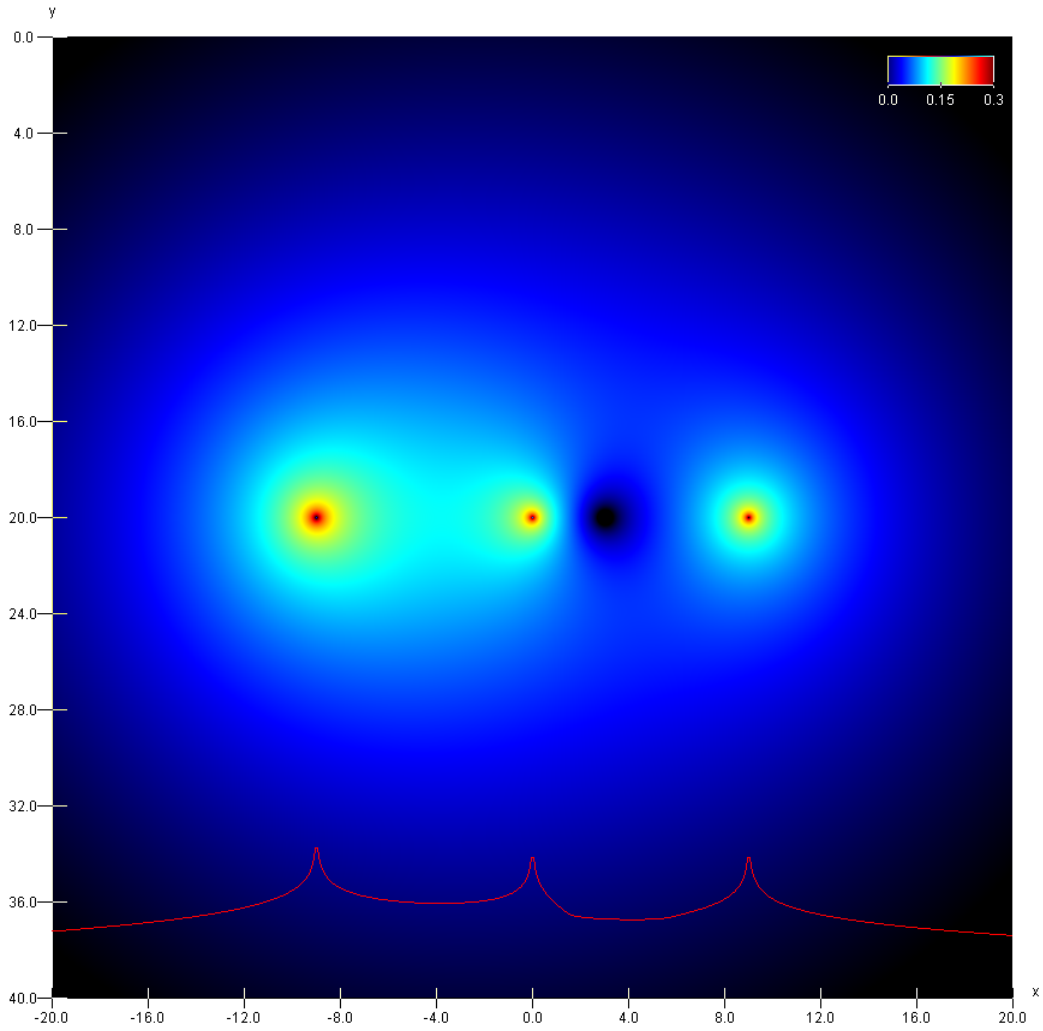


Figure 2.9: Log-plot of the magnitude of the magnetic field 1/4 into the 50 Hertz cycle (0.005 s). The red curve displays the maximum magnitude of every point with a given x position.

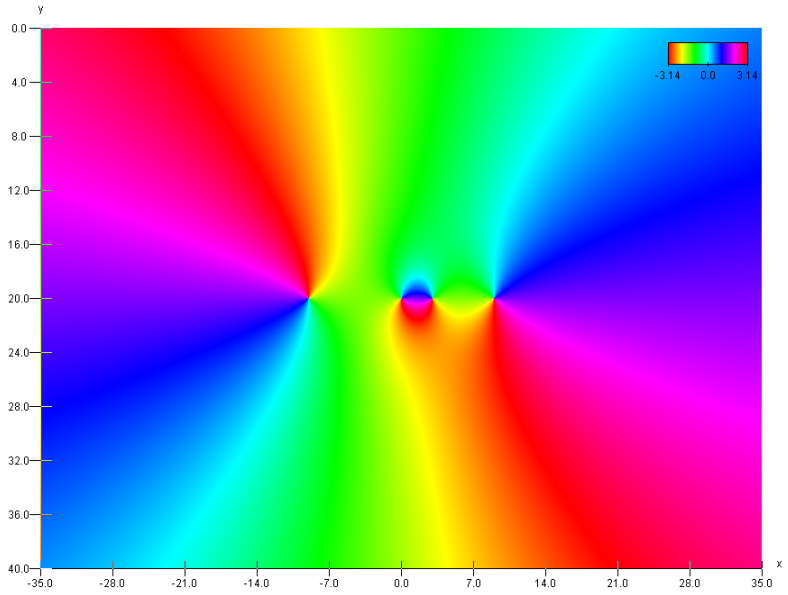


Figure 2.10: Direction of the magnetic field 1/4 into the 50 Hertz cycle (1/200 s).

2.3.3 Sinusoidal Vector Sum

Combining the vector summation in 2.18 with the definition of alternating current in 2.26, we end up with a summation of sinusoids in each axis:

$$B_{nx} = \sum_{i=1}^{n} \frac{\mu_o \; d(i)_y}{2\pi \left(d(i)_x^2 + d(i)_y^2\right)} I_p \sin(\omega t + \varphi_i) \tag{2.27}$$

$$= \sum_{i=1}^{n} A_{ix} \sin(\omega t + \varphi_i) \tag{2.28}$$

$$= A_x \sin(\omega t + \delta_x) \tag{2.29}$$

$$B_{ny} = -\sum_{i=1}^{n} \frac{\mu_o \; d(i)_x}{2\pi \left(d(i)_x^2 + d(i)_y^2\right)} I_p \sin(\omega t + \varphi_i) \tag{2.30}$$

$$= \sum_{i=1}^{n} A_{iy} \sin(\omega t + \varphi_i) \tag{2.31}$$

$$= A_y \sin(\omega t + \delta_y) \tag{2.32}$$

Note that A_{ix} and A_{iy} is the amplitude of the magnetic field of the i-th wire.

Moreover, the sum of these sinusoids can always be expressed as another sinusoid [29], as we saw in the previous section. This shall become important later on, seeing how the phase and amplitude of the final sinusoid can be determined efficiently using the Harmonic Addition Theorem.

Harmonic Addition Theorem

The Harmonic Addition Theorem enables the summation of N sinusoids of the following form:

$$\Psi = \sum_{i=1}^{N} A_i \cos(\omega t + \delta_i) = A \cos(\omega t + \delta) \tag{2.33}$$

The final sinusoid is characterized by a new amplitude ...:

$$A^2 = \sum_{i=1}^{N}\sum_{j=1}^{N} A_i A_j \cos(\delta_i - \delta_j) \tag{2.34}$$

$$= \sum_{i=1}^{N} A_i^2 + 2\sum_{i=1}^{N}\sum_{j>i}^{N} A_i A_j \cos(\delta_i - \delta_j) \tag{2.35}$$

... and a new phase:

$$\tan\delta = \frac{\sum_{i=1}^{N} A_i \sin\delta_i}{\sum_{i=1}^{N} A_i \cos\delta_i} \tag{2.36}$$

Sinusoidal Sum The definition of a sinusoidal used by this theorem is based on the cos function, while our current (and thus the magnetic field) is defined using the sin function. However, we can easily convert between the two representations using the following identities:

$$A\sin(\omega t + \delta) = A\cos(\omega t + \hat{\delta}) \tag{2.37}$$

$$\hat{\delta} = \delta - \frac{\pi}{2} \tag{2.38}$$

Sum of Three Conductors

If we are measuring the magnetic field of a simple three-phase power line, such as example 2.1, it may prove difficult or impossible to know the exact phases of each conductor in advance. But we can most certainly rely on the fact that each conductor is be offset from each other by 120°, or expressed as an unknown initial offset Ψ:

$$\delta_1 = \Psi + 0° \tag{2.39}$$

$$\delta_2 = \Psi + 120° \tag{2.40}$$

$$\delta_3 = \Psi + 240° \tag{2.41}$$

$$\tag{2.42}$$

Amplitude Assuming we have these three conductors, the amplitude in 2.28 and 2.31 then reduces to:

$$\begin{aligned} A_x^2 &= \sum_{i}^{3} A_{ix}^2 + 2[A_{1x}A_{2x}\cos(\delta_1 - \delta_2) + A_{1x}A_{3x}\cos(\delta_1 - \delta_3) \\ &\qquad + A_{2x}A_{3x}\cos(\delta_2 - \delta_3)] \\ &= \sum_{i}^{3} A_{ix}^2 + 2[A_{1x}A_{2x}\cos(-120°) + A_{1x}A_{3x}\cos(-240°) \\ &\qquad + A_{2x}A_{3x}\cos(-120°)] \\ &= A_{1x}^2 + A_{2x}^2 + A_{3x}^2 - A_{1x}A_{2x} - A_{1x}A_{3x} - A_{2x}A_{3x} \end{aligned} \tag{2.43}$$

Recall that A_{ix} and A_{iy} is the amplitude of the magnetic field of the i-th wire in the x-axis and y-axis:

$$A_{ix} = \frac{\mu_o\ d(i)_y I_p}{2\pi\ (d(i)_x^2 + d(i)_y^2)} \tag{2.44}$$

$$A_{iy} = -\frac{\mu_o\ d(i)_x I_p}{2\pi\ (d(i)_x^2 + d(i)_y^2)} \tag{2.45}$$

A similar approach leads us to the expression for A_y:

$$A_y^2 = A_{1y}^2 + A_{2y}^2 + A_{3y}^2 - A_{1y}A_{2y} - A_{1y}A_{3y} - A_{2y}A_{3y} \tag{2.46}$$

Phase We can also determine the final phase of 2.28 and 2.31 for these three conductors, using 2.36:

$$\tan \delta_x = \frac{A_{1x} \sin \delta_{1x} + A_{2x} \sin \delta_{2x} + A_{3x} \sin \delta_{3x}}{A_{1x} \cos \delta_{1x} + A_{2x} \cos \delta_{2x} + A_{3x} \cos \delta_{3x}} \tag{2.47}$$

Substituting in the phases of the wires, we get:

$$\tan \delta_x = \frac{A_{1x} \sin \Psi + A_{2x} \sin(\Psi + 120°) + A_{3x} \sin(\Psi + 240°)}{A_{1x} \cos \Psi + A_{2x} \cos(\Psi + 120°) + A_{3x} \cos(\Psi + 240°)} \tag{2.48}$$

And the same for δ_y:

$$\tan \delta_y = \frac{A_{1y} \sin \Psi + A_{2y} \sin(\Psi + 120°) + A_{3y} \sin(\Psi + 240°)}{A_{1y} \cos \Psi + A_{2y} \cos(\Psi + 120°) + A_{3y} \cos(\Psi + 240°)} \tag{2.49}$$

We could elect to use $\tan^{-1}$ to compute the angles δ_x and δ_y, but we would lose the quadrant position in the process. It is better to use $atan2$ from equation 2.24 instead.

Chapter 3

Drone Positioning

Now that we have a more detailed understanding of the magnetic field around a power line, we can begin to pin down a strategy for determining the current position using direct measurements from the field; in particular on a drone approximately 5 - 20 meters or more from the closest conductor in the power line.

The magnetic field will need to be measured from one or more magnetometer sensors on-board, capable of detecting both the strength and direction of the field. While we can adjust the number of sensors installed, as well as their individual positions for maximum accuracy, we'll first focus on the simple case of a single sensor.

3.1 Positioning with a Single Sensor

As seen in section 2.3.3, the magnetic field produced by a power line conducting alternating current is a sinusoid (see equation 2.28 and 2.31) in the x and y component. If the sensor can sample the field at a discrete interval at least twice the frequency of the alternating current (Nyquist-frequency) [21], then an application of the FFT should be able to reconstruct the amplitude (A_x) and phase (δ_x) of the sinusoids. Note that in practice, one might aim for more than twice frequency to avoid aliasing induced by high-frequency noise.

The amplitudes of the x and y sinusoidal components is seen in figure 3.1, but illustrated using the magnitude ($\sqrt{A_x^2 + A_y^2}$) and the angle ($\tan^{-1} \frac{A_y}{A_x}$) instead of A_x and A_y directly.

3.1.1 Amplitude of the Magnetic Field

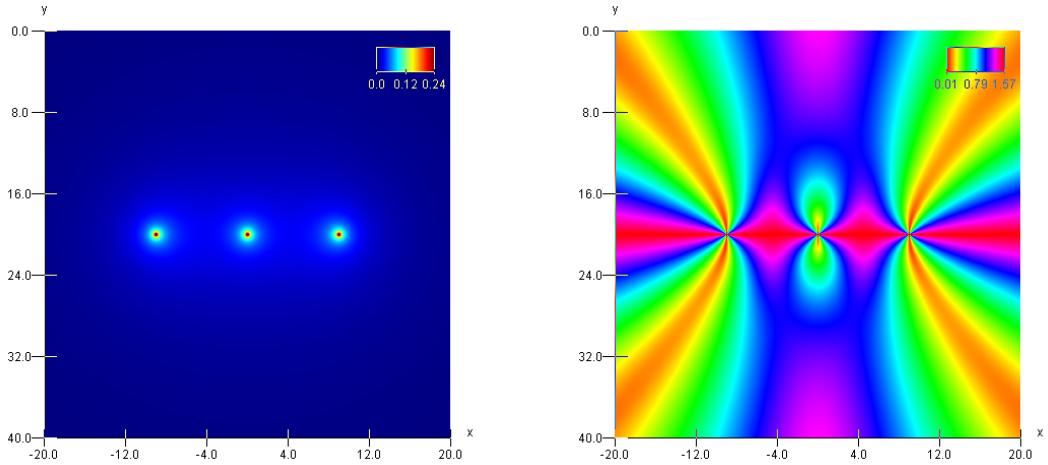
Initially, it might be tempting to use the amplitude A_x and A_y, as an input for our algorithm. However, here we encounter a significant hurdle; the amplitudes A_x (see 2.43) and A_y are proportional to the magnetic fields or each wire, which in turn depend on the amount of current flowing through each wire (see 2.1):

$$A_x^2 = A_{1x}^2 + A_{2x}^2 + A_{3x}^2 - A_{1x}A_{2x} - A_{1x}A_{3x} - A_{2x}A_{3x} \tag{3.1}$$

$$= I_p^2 \left[N_{1x}^2 + N_{2x}^2 + N_{3x}^2 - N_{1x}N_{2x} - N_{1x}N_{3x} - N_{2x}N_{3x}\right] \tag{3.2}$$

Where N_{xi} is a normalized amplitude of the magnetic field of a wire conducting a current I_p of one Ampere:

$$N_{ix} = \frac{\mu_o \; d(i)_y}{2\pi \left(d(i)_x^2 + d(i)_y^2\right)} \tag{3.3}$$



(a) Log-plot of Magnitude

(b) Plot of Angle

Figure 3.1: Plot of the amplitudes and angles (in radians) associated with each sinusoidal component in the magnetic field from (-20 m, 0 m) to (20 m, 40 m).

This suggests that both A_x and A_y are functions of the current I_p. Unfortunately, predicting this current is a non-trivial task, as this depend on the exact demand experienced by the power grid at any given time, which in turn can vary significantly from peak load (usually at 9 PM) to the lowest load. This variation in load can be seen in figure 3.2. In addition to weekly variation, the load profile also depend on the current season and the distribution of holidays.

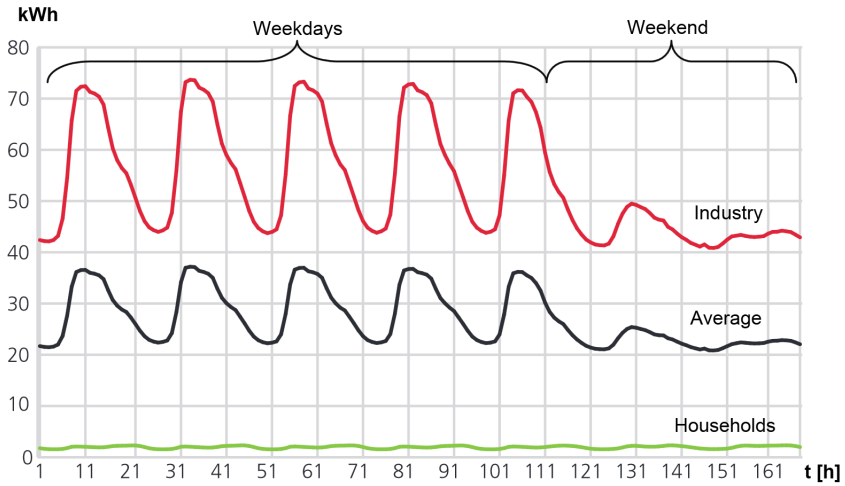


Figure 3.2: Weekly variation in average power consumption (168 hours in one week) for each customer connected to a Norwegian grid, separated into households and general industry. [8]

Magnitude The same can be said of the magnitude of $\mathbf{A} = [A_x, A_y]$, which has been plotted in figure 3.1:

$$\begin{aligned} |\mathbf{A}| = \sqrt{A_x^2 + A_y^2} &= \sqrt{I_p^2(N_{1x}^2 + \dots) + I_p^2(N_{1y}^2 + \dots)} \\ &= I_p\sqrt{N_{1x}^2 + \dots + N_{1y}^2 + \dots} \end{aligned}$$

This has the effect of scaling the magnitude of the amplitude $\mathbf{A}$ by the square of I_p, unsurprisingly, as we expect the peak of the sinusoid to increase as the magnetic fields from each wire increase in strength.

Angle Looking at figure 3.1, one might postulate that the angle is independent of I_p, given that it is divided into four reflected quadrants containing every angle from 0 to $pi/2$ radians. We'll determine if this is actually the case:

$$\begin{aligned} A_\theta &= \tan^{-1}\frac{A_y}{A_x} && (3.4) \\ &= \tan^{-1}\frac{\cancel{I_p^2}(N_{1y}^2 + \dots)}{\cancel{I_p^2}(N_{1x}^2 + \dots)} \\ &= \boxed{\tan^{-1}\frac{N_{1y}^2 + N_{2y}^2 + N_{3y}^2 - N_{1y}N_{2y} - N_{1y}N_{3y} - N_{2y}N_{3y}}{N_{1x}^2 + N_{2x}^2 + N_{3x}^2 - N_{1x}N_{2x} - N_{1x}N_{3x} - N_{2x}N_{3x}}} && (3.5) \end{aligned}$$

As we suspected, A_θ is not a function of I_p. That enables it to be used in our position algorithm. Note also that we will use $atan2$ in place of tan^{-1} in our final implementation.

3.1.2 Phase of the Magnetic Field

Next, let us consider the phase of the sinusoid $\mathbf{B}$. Unfortunately, after applying the FFT algorithm, the resulting phase will be shifted by an unknown angle Ψ. For instance, consider the magnetic field strength in figure 2.7, and assume we start sampling at time $t_0 = 0.005$. In that case the sinusoid will be shifted backwards by 1/4 of a period, making the unknown angle $\Psi = \frac{\pi}{2}$ radians. Starting at any other time relative to the beginning of the period results in different values of Ψ.

It may be difficult to determine Ψ in advance, especially seeing how we have no convenient way of measuring the network voltage directly, unlike automatic systems that synchronize power generators against the grid. Theoretically, one could use a high-precision clock (such as GPS with 4 or more satellites) and a frequently updated database to supply Ψ given an exact location and time, but this is a rather complex and brittle solution. Notably given all the unpredictable frequency fluctuations that occur when the demand exceeds the supply of electric power, causing Ψ to drift over time. Granted, the frequency is adjusted daily to match the ideal frequency (some devices use the AC frequency for time-keeping), but this might prove too infrequent for our purposes.

Figure 3.4 displays the phase of each the sinusoid in B_x and B_y. Curiously, that there is an extremely sharp transition between the line $y = 20$, where points at the top right (green) and bottom right (magenta) are clearly approximately π radians apart.

Sharp Transition To understand this, consider a point p in the green section, and its reflected counterpart $\hat{p}$ across the dividing line $y = 20$. Figure 3.3 displays these points in relation to one of the conducting wires w. Both points are at the same distance r to the wire w, and share the same magnitude as predicted by the polar representation in

equation 2.15. The angle of the field vector in $\hat{p}$ has been rotated by 90° clockwise, and as a result, A_x and $\hat{A}_x$ has the opposite sign.

The last observation is crucial, as it every the magnitude from A_{1x} to A_{3x} in equation 2.48 to switch sign. This inversion is cancelled by the fraction, but it does effect the angle if we use $atan2$ instead (see definition 2.24), as evident by the following identity:

$$\text{atan2}(y, -x) = \begin{cases} \pi - \text{atan2}(y, x) & y > 0 \\ -\pi + \text{atan2}(y, x) & y < 0 \end{cases} \tag{3.6}$$

This is the source of the transition.

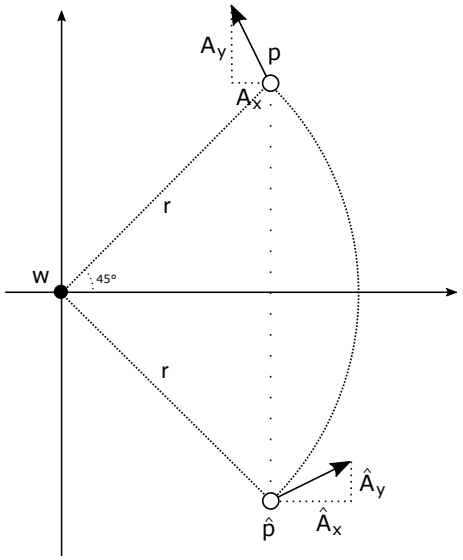


Figure 3.3: Comparing the magnetic field vector of a reflected point across $y = 20$.

Finding an Invariant Before we discard the idea of relying on the sinusoidal phase, let us examine just how it changes due to different values of Ψ.

Recall that the final phase δ_x is defined by equation 2.48. One way to interpret this equation, is to think of the amplitude A_{ix} as a vector of length A_{ix} and angle δ_{ix}:

$$\mathbf{V_i} = A_{ix}\underline{/\delta_{ix}} = \begin{bmatrix} A_{ix}\cos\delta_{ix} \\ A_{ix}\sin\delta_{ix} \end{bmatrix} \tag{3.7}$$

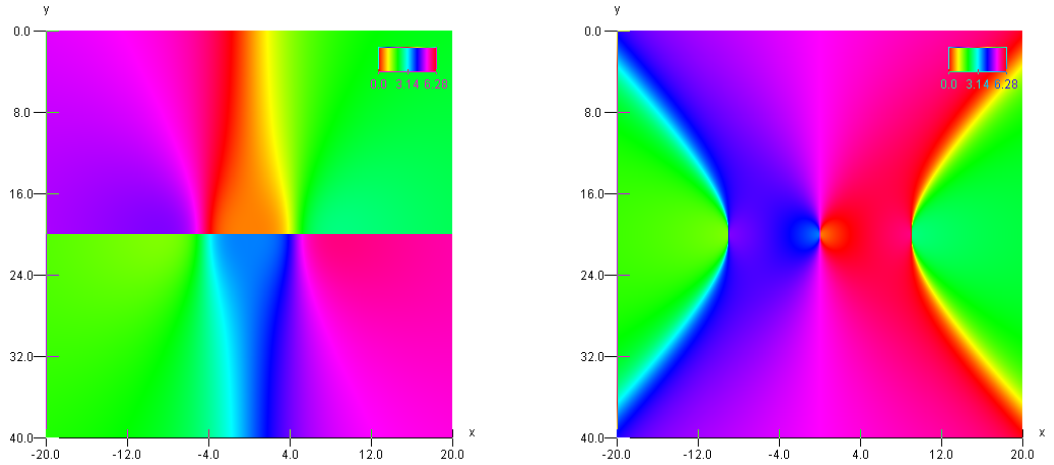
And similarly for A_{iy}:

$$\mathbf{W_i} = A_{iy}\underline{/\delta_{iy}} = \begin{bmatrix} A_{iy}\cos\delta_{iy} \\ A_{iy}\sin\delta_{iy} \end{bmatrix} \tag{3.8}$$

Then the numerator in equation 2.48 is simply the sum of the y components of $\mathbf{V_1}$, $\mathbf{V_2}$ and $\mathbf{V_3}$; while the denominator is the sum of the x components of said vectors.

Taking this one step further, one can interpret the equation as the angle of the vector sum $\mathbf{V} = \mathbf{V_1} + \mathbf{V_2} + \mathbf{V_3}$:

$$\begin{aligned} \tan V_\theta &= \frac{V_y}{V_x} = \frac{V_{1y} + V_{2y} + V_{3y}}{V_{1x} + V_{2x} + V_{3x}} \\ &= \frac{A_{1x}\sin\delta_{1x} + A_{2x}\sin\delta_{2x} + A_{3x}\sin\delta_{3x}}{A_{1x}\cos\delta_{1x} + A_{2x}\cos\delta_{2x} + A_{3x}\cos\delta_{3x}} \\ &= \frac{\cancel{I\!/_p}\left[N_{1x}\sin\delta_{1x} + N_{2x}\sin\delta_{2x} + N_{3x}\sin\delta_{3x}\right]}{\cancel{I\!/_p}\left[N_{1x}\cos\delta_{1x} + N_{2x}\cos\delta_{2x} + N_{3x}\cos\delta_{3x}\right]} \end{aligned}$$



(a) x component

(b) y component

Figure 3.4: Plot of the phase angles associated with each sinusoidal component in the magnetic field from (-20 m, 0 m) to (20 m, 40 m), assuming $\Psi = 0$. The angles are all in radians.

Keep in mind that the current I_p is also cancelled in the expression. Now, the unknown offset Ψ then becomes the initial angle of vector V_1, while the angle between each subsequent vector v_i is 120°. This is all illustrated in figure 3.5. Any increase or decrease of Ψ causes each vector to be rotated by a corresponding amount, which is of course self-evident from 2.42. However, because rotation is a linear transformation [26], the final vector $\mathbf{V}$ will also be rotated by the same amount.

$$\begin{aligned} \Delta &= \Psi_1 - \Psi_0 \\ V_{\theta 1} &= V_{\theta 0} + \Delta \\ W_{\theta 1} &= W_{\theta 0} + \Delta \end{aligned}$$

Here is the interesting part - because both $V_{\theta 1}$ and $W_{\theta 1}$ is shifted by the same Δ, the angle difference between the two will be constant and independent of Ψ:

$$\boxed{W_{\theta 1} - V_{\theta 1} = \text{AngleDiff}(\delta_y, \delta_x) = Z_\theta} \tag{3.9}$$

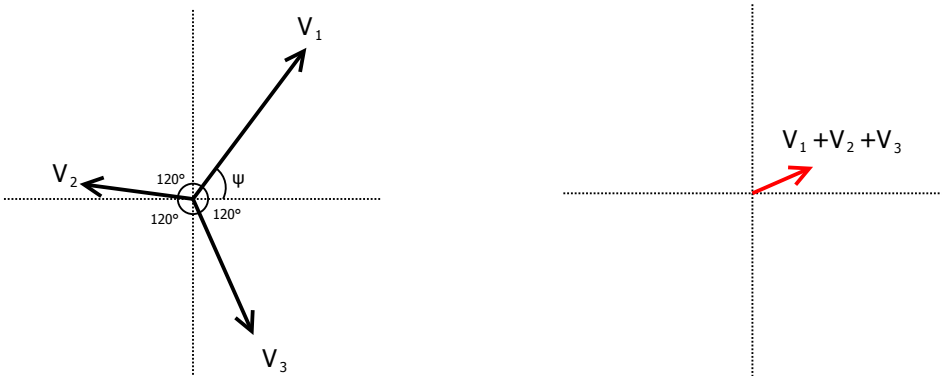
The angle difference function (AngleDiff) is necessary here, as there are two distinct ways we can measure the difference between two angles, either counter clockwise or clockwise. The most common solution is just select the smallest possible angle; one computationally efficient definition that satisfies this property is as follows:

$$\text{AngleDiff}(a, b) = \begin{cases} 2\pi - a + b & \text{if } a > b \\ b - a & \text{if } a \leq b \end{cases} \tag{3.10}$$

Z_θ can thus be used an another input in our position algorithm. Figure 3.6 displays Z_θ over the usual area of interest.

3.1.3 Algebraic Solution?

Given the invariant found in the amplitude angle (3.5) and phase difference (3.9), we do have a system of two equations and two unknowns - x and y - which is likely determined and thus solvable.



(a) Vectors $\mathbf{V_1}$, $\mathbf{V_2}$ and $\mathbf{V_3}$ (b) The vector sum $\mathbf{V}$

Figure 3.5: Summation of the amplitudes as vectors.

A algebraic solution would be even more desirable, but it seems very unlikely that one would exist. Many parts of the equations (B, δ and A) are transcendental, with a rather complex layered structure. Resorting to a CAS, such as the Solve[f] method in Mathematica (requires code listing A.1), did not yield any promising results either:

$$\text{Solve}[\{\tan^{-1}\left(\frac{\text{amplitudey}(x,y)}{\text{amplitudex}(x,y)}\right) = A,$$
$$\text{angleDifference}(\text{phasey}(x,y,0), \text{phasex}(x,y,0)) = B\}, \{x,y\}, \mathbb{R}]$$

Solve::inex: Solve was unable to solve the system with inexact coefficients or the system obtained by direct rationalization of inexact numbers present in the system. Since many of the methods used by Solve require exact input, providing Solve with an exact version of the system may help.

Unfortunately, substituting exact values for A and B as suggested did not change anything.

It is beyond the scope of this thesis to prove or disprove the existence of an analytical solution, or if an approximate solution is possible. Though if the latter is the case, it might still be more accurate or efficient to use numerical optimization or a root finding algorithm. We will explore the use of numerical algorithms in chapter 4.

3.2 Positioning with Multiple Sensors

It is fairly straight forward to incorporate the input of multiple sensors in our model. First, install N sensors on the drone at a relative offset to a given reference point (center of the drone), and supply these offsets to the algorithm responsible for lookup of the current position.

Each of the N sensors output two magnetic field invariants. We can call the resulting array of invariants the feature vector of point $\mathbf{p}$:

$$\mathbf{f}(\mathbf{p}) = \begin{bmatrix} Z_\theta(1) & A_\theta(1) & \cdots & Z_\theta(N) & A_\theta(N) \end{bmatrix} \tag{3.11}$$

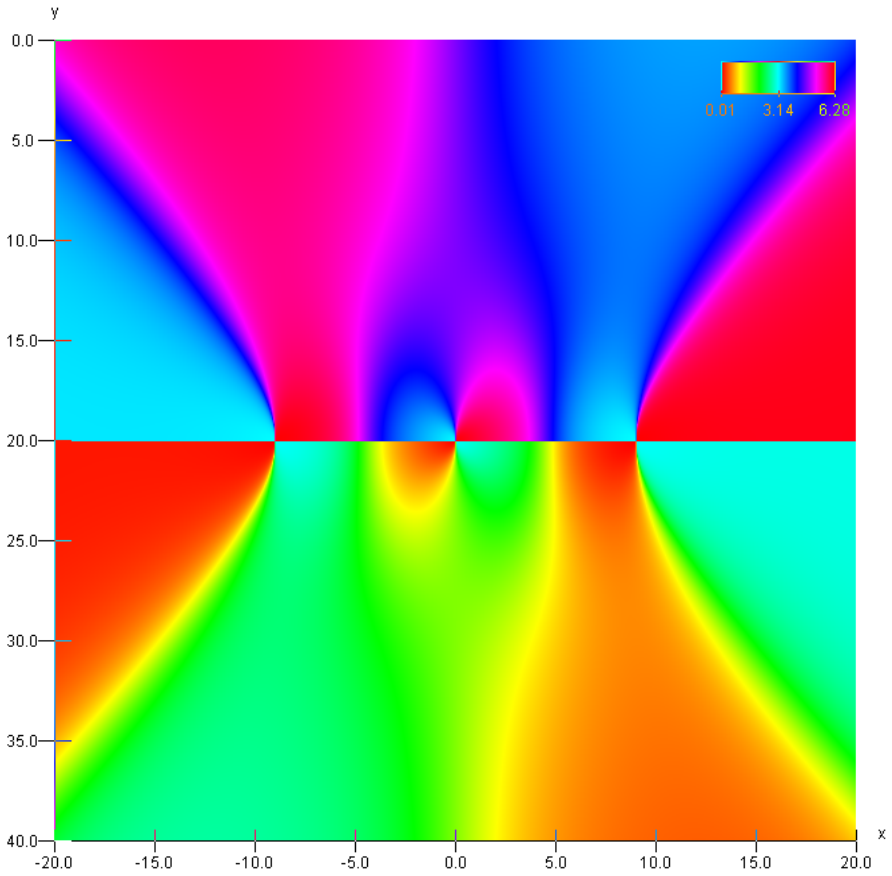


Figure 3.6: The phase of the sinusoid in the y component subtracted from the phase in the x component, using the AngleDiff function.

Here, $Z_\theta(i)$ and $A_\theta(i)$ is the phase difference of i-th sensor. The optimal search algorithm (CMA-ES) is given a feature vector $\mathbf{x_f}$ and the offsets of the corresponding sensors, and attempts to find the true point $\mathbf{x_p}$ by looking at points with the nearest feature vector. However, most algorithms expect a scalar field, so we have to minimize the invariant distance to the target. The invariant distance is just the distance between the two vectors:

$$d(\mathbf{f_1}, \mathbf{f_2}) = |\mathbf{f_1} - \mathbf{f_2}| \tag{3.12}$$

Let us next define the *miss distance* is then the distance between the true point ($\mathbf{x_p}$) and the resulting point $\mathbf{r}$:

$$m = |\mathbf{x_p} - \mathbf{r}| \tag{3.13}$$

. This metric will become useful later when we need to evaluate the accuracy of different numerical algorithms.

3.3 Determine Orientation

Throughout this chapter, we have implicitly simplified the positioning problem by assuming that each conductor is always parallel to the relative Z-axis of the sensor(s). This had the effect of reducing the dimensionality of the problem to the two dimensional case.

In reality, the drone (or craft) carrying the sensor exists in three dimensions and should continuously align itself according to the changing direction of the wires, or we run the risk of violating our assumptions. This is particularly important if we need to use multiple sensors located at a relative offset to each other, as any deviations in the angle relative to the wire will alter their true position in the x–y plane.

3.3.1 Rotation of a Rigid Body

Let us begin by defining the three angles of rotation that uniquely define an orientation of a rigid body (our drone) in $\mathbb{R}^3$, after Euler [28] and the xyz (pitch-roll-yaw) convention that is ubiquitous in aerospace engineering.

This convention decomposes an arbitrary rotation into the three elemental rotations about each of the axes in the coordinate system, represented as a transformation matrix:

$$\mathbf{R} = \mathbf{R_z}(\alpha)\mathbf{R_y}(\beta)\mathbf{R_x}(\gamma) \tag{3.14}$$

Where α, β and γ is the yaw, pitch and roll Euler angles respectively, as illustrated in figure 3.7. The elementary transformation matrices are given by:

$$\mathbf{R_x}(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \tag{3.15}$$

$$\mathbf{R_y}(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \tag{3.16}$$

$$\mathbf{R_z}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.17}$$

To rotate a point $\mathbf{p}$, we pre-multiply it with the transformation matrix: $\hat{\mathbf{p}} = \mathbf{R}\mathbf{p}$.

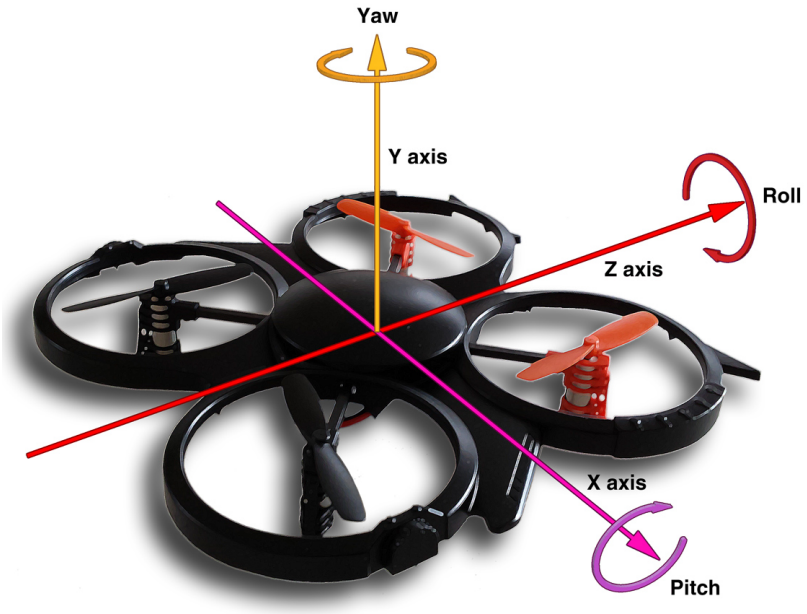


Figure 3.7: The pitch, yaw and roll that define an arbitrary rotation of a drone.

3.3.2 Drone Rotation

We can eliminate the roll angle right from the bat, given the fact that a power line is nearly always level with the horizon looking perpendicular to the wire direction (x axis). Then, assuming the drone is equipped with an IMU (Intertial Measurment Unit), which tracks the current velocity, acceleration and orientation of the craft, we can instruct the navigational system to keep level with the horizon as well.

Unfortunately, the effect of the other two angles are not quite so trivial. For instance, a conductor will have a certain sag over a given distance, and two transmission towers may even be constructed at different elevations, usually in mountainous regions where conductors follow the steep gradient of the local terrain. This has the effect of shifting the position of sensors that are offset in the y axis. We will need to apply a minor adjustment of the pitch to compensate, though only in the range $0° - 80°$ and $280° - 360°$. Exceeding this range does not make any sense, as a power line upside down is indistinguishable to the drone.

The last Euler angle, the yaw, is needed to orient the drone according to the cardinal direction of the power line, which can change whenever it passes a transmission tower. Assume a drone is carrying multiple magnetic field sensors and is rotated 45° clockwise about the Y-axis (yaw). Then a sensor located at point $\begin{bmatrix}1 & 0 & 0\end{bmatrix}^T$ relative to another will sample the following relative point instead:

$$\mathbf{p}' = \begin{bmatrix}\frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}}\end{bmatrix}^T$$

3.3.3 GPS Assisted Orientation

To synchronize the orientation of the drone and the power line, we might defer to a pre-recorded geographical path of the power line in a database (containing both the height gradient and the geographic coordinates), found using the current location estimated by GPS.

Granted, this does rely on the accuracy and completeness of the power line map, but this is exactly the kind of information that should be readably available to the power utility companies that inspect power line, assuming it exists. However, if the positioning technology would prove useful to applications beyond power line inspection, it may not necessarily be legal or practical to distribute this kind of high-precision information.

3.3.4 Magnetic Field Orientation

We know, based on equation 2.1 and figure 2.2, that every magnetic field vector is orthogonal to the z axis relative to the power line. This implies that a measured magnetic field vector $\mathbf{v}$ satisfies $\mathbf{v}.\hat{\mathbf{z}} = 0$, where $\hat{\mathbf{z}}$ is the unit vector representing the direction of the power line. In $\mathbb{R}^3$, this equation has an infinite number of vector solutions in the same plane, which is illustrated by figure 3.8. To find an orthogonal vector $\mathbf{w}$ to an arbitrary vector $\mathbf{v}$, we will use one of the following vectors ($\mathbf{w_a}$ or $\mathbf{w_b}$, whichever have a non-zero magnitude:

$$\mathbf{v} = \begin{bmatrix}x\\y\\z\end{bmatrix} \quad \mathbf{w_a} = \begin{bmatrix}-y\\x\\0\end{bmatrix} \quad \mathbf{w_b} = \begin{bmatrix}0\\-z\\y\end{bmatrix}$$

Determine Pitch Only While we cannot uniquely determine the $\hat{\mathbf{z}}$ vector with single non-zero sample of the magnetic field, we can nevertheless compute the plane that

contains it and adjust the pitch angle of the drone such that it is orthogonal to said plane. Now every possible yaw angle ($0°-360°$) essentially represents the infinite number of vectors that could be $\hat{\mathbf{z}}$.

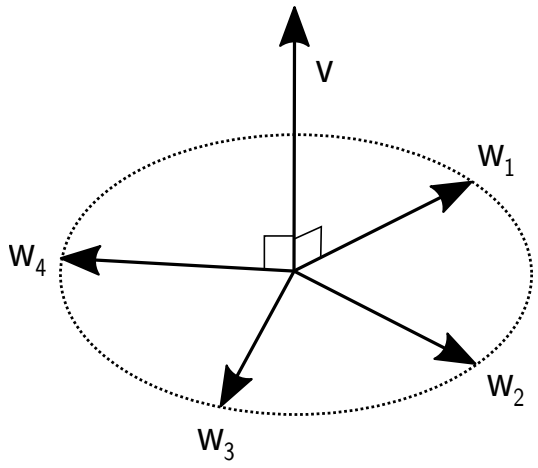


Figure 3.8: A selection of orthogonal vectors of $\mathbf{v}$ in R^3: $\mathbf{w_1}, \mathbf{w_2}, \cdots, \mathbf{w_4}$.

Determine Pitch and Yaw To determine the yaw angle as well, we need a second vector sample that that been rotated by the same angles. Assuming the angular velocity of the drone (and thus sensor) is zero, we know that the same sensor will measure magnetic field vectors rotated by the same amount relative to the power line, even over time. Similarly, the sensor is stationary, it should also measure different vectors given that the magnetic field is constantly changing direction (within an AC period).

The solution is therefore to sample a sensor at different time instants in order to read at least two different vectors ($\mathbf{v}$ and $\mathbf{u}$). We then know that the cross product of these two vectors ($\mathbf{v} \times \mathbf{u}$) is by definition orthogonal to both, and must be parallel to the $\hat{\mathbf{z}}$ vector we are seeking. That gives us the direction of the power line, which allows us to adjust correctly adjust the pitch, yaw and roll angles.

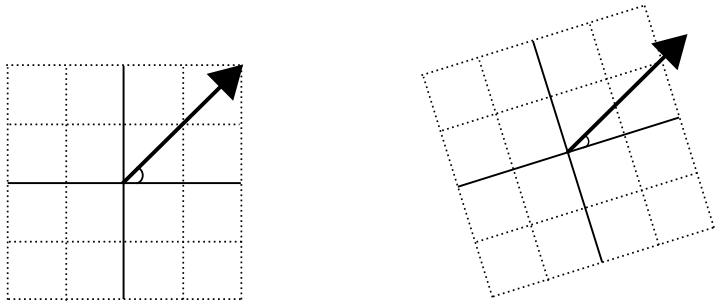
If the drone is subject to a non-zero angular velocity (tracked by a gyroscope or IMU), then we have to to compensate by undoing the rotation of the second vector given the amount the drone has rotated in the elapsed time between the two samples.

Adjusting the Rotation Correcting the drone rotation given $\hat{\mathbf{z}}$ is equivalent to the problem if rotating the current reference frame such that a vector $\mathbf{v}$ appears as vector $\mathbf{t}$.

To understand this problem, let's first consider the two dimensional case and vector $\mathbf{a} = \begin{bmatrix} 0 & 1 \end{bmatrix}^T$. If we rotate the coordinate counter-clockwise α degrees, without altering or touching vector $\mathbf{a}$, we see that the vector now appears to have rotated anti-clockwise α degrees in the new coordinate system, as illustrated in figure 3.9.

To undo the rotation, we look at the angle of $\mathbf{a}$ before (a_α) and after (a_γ), and compute the difference:

$$\begin{aligned} \hat{\alpha} &= a_\gamma - a_\alpha \\ &= (a_\alpha - \alpha) - a_\alpha = -\alpha \end{aligned}$$



(a) Non-rotated reference frame (b) Rotated reference frame

Figure 3.9: The effect of rotating a reference frame around a static environment. Note that the apparent rotation of the vector is in the opposite direction.

Clearly, we need to subtract the new angle from the old desired angle to reverse the rotation.

To summarize, in order to rotate our drone such that a vector $\mathbf{v}$ appears to be parallel to $\mathbf{t}$, we compute their elementary rotations compared to $\begin{bmatrix}1 & 1 & 1\end{bmatrix}^T$ and rotate according to these equations:

$$\Delta_\alpha = t_\alpha - v_\alpha \tag{3.18}$$
$$\Delta_\beta = t_\beta - v_\beta \tag{3.19}$$
$$\Delta_\gamma = t_\gamma - v_\gamma \tag{3.20}$$

3.4 Phase Sequence

As previously established in subsection 2.3.1, the AC current in in a three-phase system must be sequentially offset by 120°. We have also seen that an initial starting degree, Ψ, is irrelevant to our field invariants. Thus, we only have to focus on phase offsets 0°, 120° and 240°, which we will henceforth call δ_1, δ_2 and δ_3, respectively.

In table 2.1, we arbitrarily assigned phase offset δ_1 to the current of wire A, then δ_2 to wire B, and lastly δ_3 to wire C. However, this is only one out of $3! = 6$ possible combinations, and we have no reason to believe any one in particular is universally correct.

In fact, because the space between nearby conductors (and the ground) exhibit a non-zero capacitance as function of distance, and the middle wire is closer to the other two wires (in our example) than they are to their counterparts, the total capacitance is not the same for all three wires. However, it is desirable that the current in the three-phase system is identical in order for the neutral wires to carry as little current as possible. So, when this effect would be is too pronounced (typically when the length exceed 140 km), we can negate it by swapping the positions of the wires (transposition) $\frac{1}{3}$ and $\frac{2}{3}$ into the total distance, such that each wire is the middle wire exactly $\frac{1}{3}$ of the total length. This is all illustrated by figure 3.10.

Unfortunately, when this scheme is used, we can no longer assume each phase is permanently assigned to a specific wire position. But does this actually pose a problem? Let us investigate the different possible ways in which we can arrange the different phase offsets.

Table 3.1 enumerates every possible phase sequence in a three-phase system. Next, if we add the angle 120° or 240° to every phase offset using Ψ, we can see that we are only

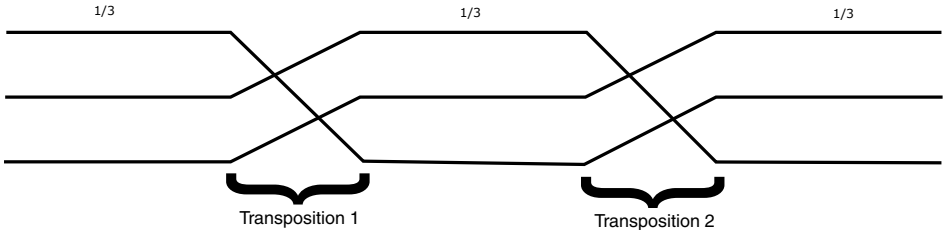


Figure 3.10: A transposition scheme applied to a power line.

Wire A	Wire B	Wire C	$\Psi = 120°$	$\Psi = 240°$	Category
1	**2**	**3**	231	312	Positive
1	3	2	213	**321**	Negative
2	1	3	**321**	132	Negative
2	3	1	312	**123**	Positive
3	1	2	**123**	231	Positive
3	**2**	**1**	132	132	Negative

Table 3.1: Every possible sequence of phase offsets that we can arrange with three conductors. Note that they are only two real categories, if we reduce the common offset Ψ.

really dealing with to types of sequences: $\delta_1\delta_2\delta_3$ (or 123) — the positive sequence — or $\delta_3\delta_2\delta_1$ (321), the negative sequence. Every other sequence is merely a shifted version of those two fundamental sequences, and as we've seen in our field invariants, shifting by a common constant can be completely discounted.

3.4.1 Effect on Field Invariants

The phase sequence used in table 2.1 is positive, but what happens to our field invariants if we substitute a negative phase sequence?

Clearly, it will have no effect on the angle of the amplitude (equation 3.5), as it does not reference the phase offsets in any manner. This is in contrast to the phase difference (equation 3.9), which is very much a function of the AC phase offsets.

Figure 3.11 displays the phase difference in the x–y plane when the phase sequence is negative. It is evident that it is merely a version of rotated 180° (both the x and y axes are flipped). It is perhaps not surprising that the x axis is flipped, as we can go from a positive phase sequence to a negative sequence, simply by rotating the drone 180° about the y axis (pitch).

In order to determine if the phase sequence is negative or positive, we might exploit the fact that the phase invariant is flipped vertically, and measure whether or not moving upwards produces an increases in the estimated y position by our algorithm. If it does not, we know we should use the opposite phase sequence category.

3.5 Real-World Considerations

We have yet to take into account a number of real-world sources of inaccuracy, such as the normal operating range of the utility frequency, or the fact that the three conductors may not individually carry the same amount of current.

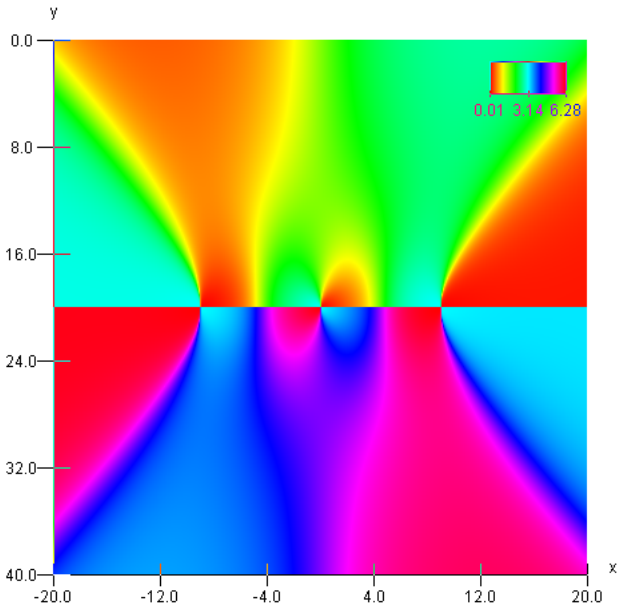


Figure 3.11: The phase difference when the phase sequence is negative. The color scale goes from 0 to 2π radians.

3.5.1 Frequency Deviations

The target frequency of the AC current flowing in electrical grid of Norway is 50 Hz, just like most other European countries. The frequency is regulated to be within a range of 0.2 Hz (49.9 Hz - 50.1 Hz) in most circumstances, but in recent years it has begun to deviate more often from this norm (up to 4 hours a day) [4], likely spurred on by the growth of electricity trade within the European marked.

Looking purely at the deviation itself, and not the change over time, the biggest concern is the change in phase offset of the observed sinusoids [1]. With that in mind, we will first numerically investigate the effects of significantly changing the frequency to the field invariants we have discovered.

Using the conductors described in table 2.1, we look at the phase of the magnetic field sinusoids B_{nx} and B_{ny} (equation 2.28 and 2.31 respectively) when the utility frequency is 47 Hz, 50 Hz and 53 Hz, and then the resulting phase difference. This is well beyond the acceptable range of frequency deviation, but we should exaggerate the effect to be sure we have observed its implications.

The effect of changing the frequency can be seen in figure 3.12. It does appear that even the phase difference is coupled with the frequency deviation, though a bit less than either δ_x or δ_y by themselves. We will determine just how this translates into inaccuracies in the positional algorithm later in chapter 6.

3.5.2 Current Imbalance

The last and final practical consideration, is the slight current imbalance of the three conductors in a three-phase system.

We will use a minute-by-minute measurement of the average imbalance over the three conductors in a power line at a 300 kV power line in Frogner, Oslo, over 30 minutes, presented as figure 3.13. There is a similar downward trend for all three conductors, and

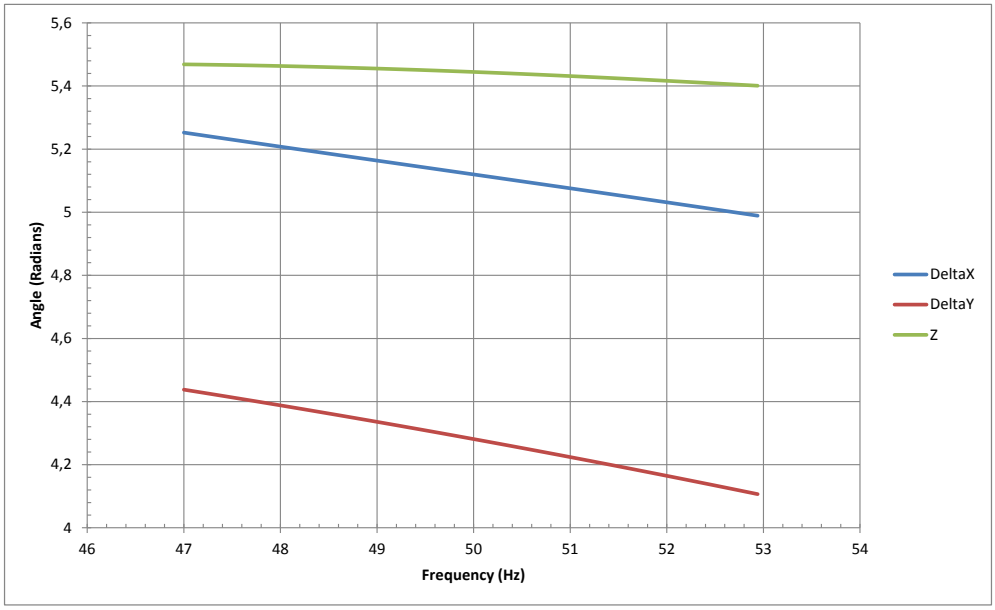


Figure 3.12: The effect of changing the frequency on point (-20, 0), expressed in δ_x, δ_y and Z_θ.

they seem to be approximately equidistant (subject to random variation). The global average and SD (Standard Deviation) of each conductor is listed in table 3.2.

To use this information in our simulation, let us consider the local SD of the the conductors at a specific time, and select the times when this SD is at the lowest and the highest value (see table 3.3). We can then use these values in our simulation directly, given that the invariants we are using are insensitive to the absolute current.

	L1	L2	L3	Average
Average	154.357	156.288	160.785	157.144
SD	3.888	4.016	4.071	3.992

Table 3.2: Average and SD (Standard Deviation) of the current in the three conductors (L1, L2 and L3).

Extreme	Time	Local SD	L1	L2	L3
Lowest Local SD	15	1.502	151.270	154.871	153.727
Highest Local SD	21	5.130	154.521	150.847	163.09

Table 3.3: Times when the SD of the conductor currents at a instant is lowest or highest.

Balance Parameters If the current balance do turn out to matter, how will be then encode it along with the field invariants? These invariants are indifferent to the absolute value of the currents, but they are affected by the relative current difference between the wires.

One way to represent this, is to normalize the currents against the current of the middle wire (assuming three wires), and define the two resulting non-trivial fractions as

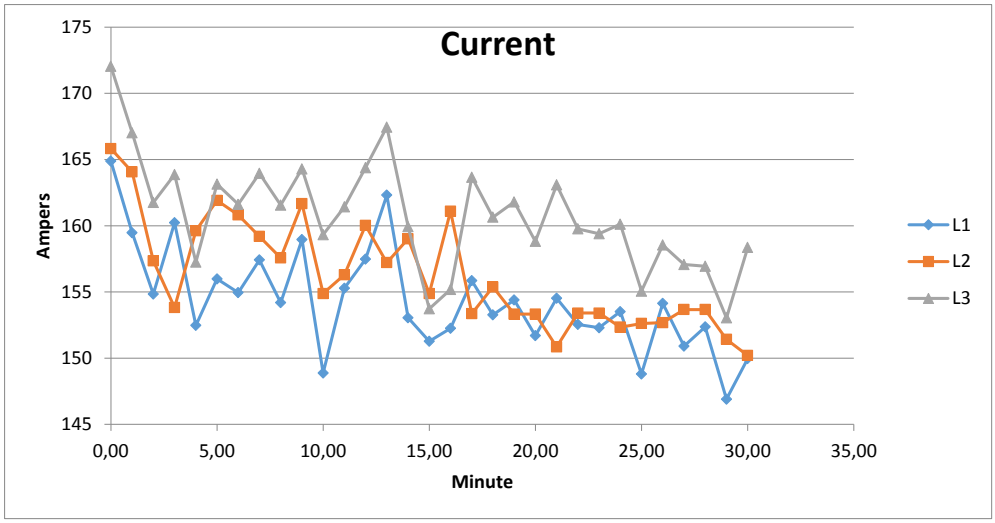


Figure 3.13: The current in three conductors (L1, L2 and L3) of a 300 kV power line during a 30 minute window.

a and b:

$$I_1' = \frac{I_1}{I_2} = a \tag{3.21}$$

$$I_2' = \frac{I_2}{I_2} = 1 \tag{3.22}$$

$$I_3' = \frac{I_3}{I_2} = b \tag{3.23}$$

Next, in the experimental data, a and b never exceed the interval 0.94 - 1.08, but we will limit them to 0.90 - 1.1 for good measure.

$$0.9 \leq a \leq 1.1 \tag{3.24}$$

$$0.9 \leq b \leq 1.1 \tag{3.25}$$

Any further imbalance is very unlikely, given that it would completely negate the benefits of using a three-phase system in the first place, and thus quickly be corrected as a faulty state.

3.6 Estimating Conductor Currents

Assuming we can accurately determine our current position, we are able to calculate the current I in each conductor.

Let A_x be the amplitude of the measured sinusoid in the x component, and N_{ix} be the normalized amplitude (see equation 3.3) of conductor i carrying 1 A of current. Then we can estimate the true current I_p by rearranging equation 3.2:

$$I_p = \sqrt{\frac{A_x^2}{N_{1x}^2 + N_{2x}^2 + N_{3x}^2 - N_{1x}N_{2x} - N_{1x}N_{3x} - N_{2x}N_{3x}}} \tag{3.26}$$

If the conductor currents differ by the relative fractions a and b, as defined in 3.23, we can nevertheless find I_2 as follows:

$$I_2 = \sqrt{\frac{A_x^2}{a^2 N_{1x}^2 + N_{2x}^2 + b^2 N_{3x}^2 - a N_{1x} N_{2x} - ab N_{1x} N_{3x} - b N_{2x} N_{3x}}} \tag{3.27}$$

Then I_1 and I_2 follows directly.

Note that we can substitute A_y and N_{iy} into these equations directly, without making any other changes.

Chapter 4

Computer Simulation

It is fairly trivial to simulate the magnetic field, and the associated amplitude angle (3.5) and phase difference (3.9) in Mathematica, or any equivalent CAS such as Maple or MATLAB.

Code listing A.1 contains a fairly straight-forward implementation in under 48 lines, with reasonable performance on a desktop computer. Despite the succinctness of such a high-level implementation, we will nevertheless aim to target a more general-purpose language platform, such as Java or C. These languages can offer more control over every performance trade-off associated with the algorithm, and not to mention enable extensive micro-optimizations. There is also the matter of a more restrictive license, particularly with regards to Mathematica, and the issue of compiling the solution to a stand-alone program.

The potential performance boost of using a more low-level language could be crucial in getting the algorithm to run at an acceptable speed on an embedded device, like on the Raspberry Pi [10]. With that in mind, we will select Java for our final implementation. It is a fairly good compromise between low-level complexity (C) and high-level succinctness, while retaining most of the performance benefits of C (on a modern JVM), though other languages (such as Go) might have been an even better choice [19].

4.1 Implementation in Mathematica

The Mathematica implementation (A.1) is structured using a layered definition of the two field invariants of interest, with separate definitions for both the x and y component. Starting with the magnetic field strengths (*fieldx* and *fieldy*) emanating from a single wire, we compute the the sinusoidal magnetic field sum in B_{nx} and B_{ny} (see 2.28 and 2.31), both in terms of its phase δ (*phasex* and *phasey*) and its amplitude A (*amplitudex* and *amplitudey*). Finally we compute the amplitude angle (3.5) in *AmplitudeAngle*.

With this as a basic foundation, one can map a given phase difference and amplitude angle to a near correct x and y coordinate using numeric optimization:

Listing 4.1: Finding a point by the amplitude and phase.

```
FindPoint[A_, B_] :=
 Module[{x, y},
  NMinimize[{(AmplitudeAngle[x, y] - A)^2 +
    (angleDifference[phasey[x, y, 0], phasex[x, y, 0]] - B)^2,
    -20 <= x <= 20, 0 <= y <= 40}, {x, y},
    Method -> "SimulatedAnnealing"]]
```

We are looking for the global minimum, so *NMinimize* is an appropriate tool for the job. To test and plot the accuracy of this position algorithm, we compute the values of our two invariants given a x and y, and then return the point distance to the proposed answer:

Listing 4.2: Plotting the accuracy of the Mathematica solution.

```
AccuracyTest[x_, y_] := Module[{R, V, W},
  R = Test[x, y];
  V = R[[3, 2, 1, 2]];
  W = R[[3, 2, 2, 2]];
  Sqrt[(x - V)^2 + (y - W)^2]]

DiscretePlot3D[AccuracyTest[x, y],
  {x, -20, 20, 1}, {y, 0, 40, 1}]
```

It's clear from the output plot 4.1 that certain areas are extremely inaccurate, deviating over 40 meters from the correct position. It would be possible to filter out results that extreme, using either GPS or the previous position and speed of the drone (INS), but this still does not address the milder inaccuracies near the center.

It is particularly troublesome that the hotspots are concentrated around the diagonals, as the drone is likely to fly over the power line through these diagonals, coupled with the fact that the upper part seems to be more inaccurate than the bottom part.

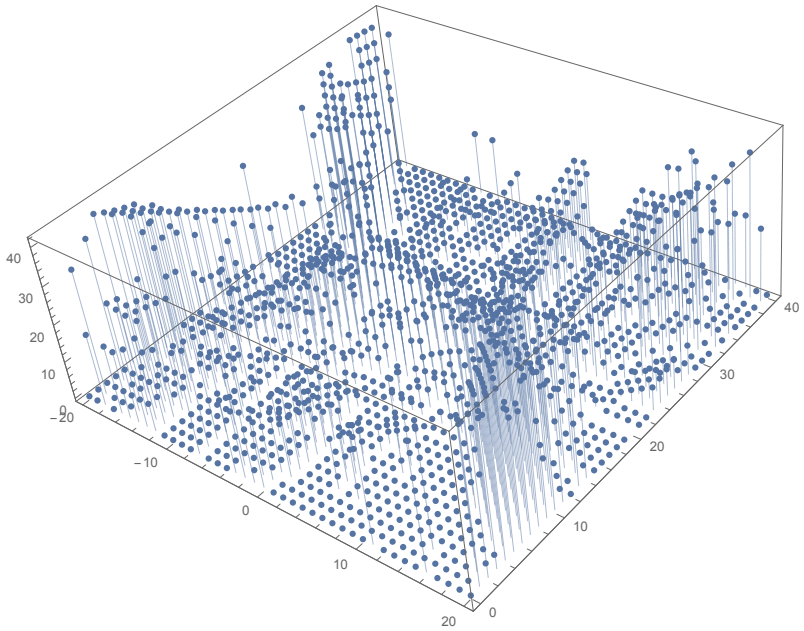


Figure 4.1: The inaccuracy of listing 4.1, in meters.

Still, the accuracy might improve if we add additional sensors at a different relative position. But before we investigate the possibility, we present the Java implementation of our position algorithm.

4.2 Overview of Java Implementation

The Java program was initially developed to plot the magnetic field based on a very large data set (4000 rows x 4000 columns) generated by Statnett, given that very few open source plotting programs (such GnuPlot) could even handle the sheer amount of data. It then seemed natural to compute the magnetic field ourselves, and use the same program for plotting the result, making it feasible to visually verify our calculation.

It was extended further to find invariants for our position algorithm, allowing us, among other things, to plot and better understand the amplitude and phase of the sinusoidal magnetic field. Finally, after evaluating the performance and accuracy of a number of different algorithms, we settled on using CMA-ES.

We will use Java 8 and Apache Maven [11] in our project. But before we get to the results, let us discuss all the basic aspects of the implementation.

4.2.1 Build Tool

A good build tool ensures the build process is consistently able to produce the same compiled output, regardless of the current system environment. Maven is common build tool for Java projects, and is capable of both installing and upgrading library dependencies from any external sources, as well as manage more complex build steps than the standard tooling (such as embedding dependencies in a single JAR-file).

Given these clear benefits, and the fact that common build are now well integrated in modern IDEs, we felt justified in choosing Maven. We will use it to include the following Java libraries:

- **PngEncoder from ObjectPlanet,** Benchmarks on Oracle's JDK on Windows 7 have shown that this library is many times faster than the standard method for saving PNG images - *ImageIO*.
- **UniVocity CSV/TSV Parsers,** A library optimized for fast and correct parsing.
- **Google Guava,** Generic utility-classes and methods that simplify common operations, such as I/O, collections and implementing object methods. We also use its Stopwatch class to time method calls.
- **Apache Commons Bean Utils,** This library is used to translate command line arguments to property assignments that initialize certain objects.
- **Apache Commons Math,** Provides an implementation of the FFT, as well as the CMA-ES algorithm.

We also instruct Maven to merge all the dependencies and the program itself into a single JAR-file, using the *maven-shade-plugin*. Superfluous dependency code is filtered with *ProGuard*, reducing the final JAR file by 4.3 MiB (83%).

4.2.2 Command Line Interface

The syntax of our command line interface developed organically over time, primarily to avoid having to recompile every time we needed to change some parameter or the appearance of the density plots. Nevertheless, it is fairly conventional, where each option or flag is delimited by space characters, and options are prefixed with the hyphen character.

The exact syntax can be seen in listing 4.3. Flags are divided three categories; firstly, general flags that apply to both plotting CSV/TSV and generated data (see table 4.1), secondly and thirdly, required and optional flags that is only valid when we compute the magnetic field from a model (see table 4.2).

Listing 4.3: Syntax of our Java program.

```
PlotCSV [-d file] [-m file] [-c colorscheme] [-ms source]
        [-s scheme] [-x start:end:count] [y start:end:count]
        [z start:end] [-t start:end:count]
        [-v m|a|x|y|y-x|ydivx] [-l] [-a] [-d x:y]
        [-dp count] [-dr radius] [-dm] [-solver name]
        [-f bool] [-ag name1,name2,name3] [-of format]
        [-ax label:tickCount:width]
        [-ay label:tickCount:height]
        [-gx start:end] [-gz start:end] output
```

Flag	Description
-d[ata]	Path to a CSV-file containing the magnitude data to plot.
-m[odel]	Path to an XML-file containing the model that will be simulated.
-s[cheme]	The color scheme used in the output image. Rainbow is default. Other schemes include Temperature, HSV, HSV2 and HSV10.
-z	The minimum and maximum z value to display using a color scheme. This is computed automatically if left out. The count is ignored.
-l[og]	Apply the common logarithm to the values before they are visualized.
-a[bslog]	Apply a modified form of the log-modulus transformation (equation 4.1) to each value.
-gy	Display a graph of the maximum value in a given column, using the,given range in the Y axis.
-gz	Range of values in the displayed graph. If not specified, the z range will be used instead.
-ax	Display the x axis with a label, a given number of ticks and a width.
-ay	Display the y axis with a label, a given number of ticks and a height.
output	Format of the output file(s), where %d is the current frame index.

Table 4.1: Table of common flags in our CLI.

Incidentally, every density plot in this thesis is generated using our PlotCSV Java program. Instructions for reproducing every figure can be found in section A.3.

4.2.3 Logarithm of Both Positive and Negative Values

Due to equation 2.15, the magnetic flux density is inversely proportional to the distance $|\mathbf{d}|$, and difficult to plot on a linear scale given widely different orders of magnitude. We can improve the visualization of this kind of data by plotting the values on a logarithmic scale, like in figure 2.9, but this approach breaks down entirely if we have to deal with both negative and positive values, such as the individual magnetic field components x and y.

One key observation is that at a certain point, values extremely close to zero might as well be zero due to limitations in the sensitivity of the sensor hardware. Therefore,

Flag	Required	Description
-v[alue]	✓	The value that will actually be visualized. Can be either one of m[agnitude], a[ngle], x, y, y-x or ydivx [y / x].
-x	✓	The minimum and maximum x coordinate of each pixel in model space, and the final width of the output image. Default is -1:1:640.
-y	✓	The minimum and maximum y coordinate of each pixel in model space, and the final height of the output image. Default is -1:1:480.
-t	✓	The starting and ending time in milliseconds, along with the number of frames to generate. Each frame is incremented by a regular time step.
-d[rone]		Test the CMA-ES algorithm once using a given drone location (x, y).
-dp		Test the CMA-ES algorithm count times and print the resulting probability distribution of the error distance.
-dm		Test the CMA-ES algortihm for every point specified by the x and y options, and write the resulting error distance map to an image.
-dr		Optional argument to the previous two CMAES tests - if set, limits the search area around the drone to radius meters.
-solver		The algorithm used in the previous tests. Default is CMAES. The other alternative is BOBYGA.
-f		If TRUE, ensure the animation or image is written to the file system. Otherwise, disable it.
-ag		A list of aggregate functions that should be applied to every frame in the animation. These functions include AVERAGE, MAX, XYD [phase difference of the peak angle in x and y], XYF [phase difference by computing the sinusoid in x and y] and AMPF [amplitude difference by computing the sinusoid in x and y].
-of		The output format of the generated image(s). Default is PNG, but it can also be CSV.

Table 4.2: Table of flags for customizing the magnetic field generator.

a lot of the negative end of the logarithmic scale is wasted on values that we cannot measure in the first place.

We can avoid this problem by applying the log-modulus transformation [15] instead, causing extreme outliers in both directions to fit in the plot using a logarithmic scale, whereas very small values close to zero become linear. The definition of this transformation is as follows:

$$L(x) = sgn(x) \log(|x| + 1)$$

We compare this function against the natural logarithm in figure 4.2. Unlike the natural logarithm, $L(x)$ is entirely defined in $\mathbb{R}$ for negative numbers, and when $-1 \leq x \leq 1$, the function is approximately linear.

The linear range is a bit too large for our data set, given that the magnetic field density is entirely within -1 to 1 T, so we'll scale it down by multiplying x by a large constant:

$$\hat{L}(x) = sgn(x) \log(|1000x| + 1) \tag{4.1}$$

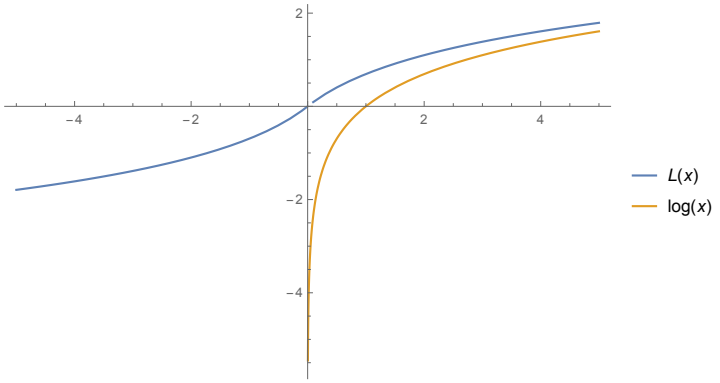


Figure 4.2: Comparison of the log-modulus transformation and the ordinary natural logarithm.

4.2.4 Plotting TSV data

We were able to obtain a raw TSV output of a simulated magnetic field from a source at Statnett. This 91.6 MiB file contains exactly 16 million data points, ranging from point (-20, 0) to (20, 40) using a 1 cm increment in both directions. The virtual model simulated is almost identical to figure 2.1, with the exception of the actual wire coordinates. Every data point represent the average field strength of the corresponding point in microtesla, as depicted in figure 4.3.

The overhead ground wires are also visible in the figure, though their effect on the magnetic field appears to be minimal and local in scope. Thus, we can just ignore them in our simulation.

A snippet of the Statnett data set can be seen in listing 4.4. This format is loaded into memory by our *CvsFieldParser* class, which in turn uses the Java library *uniVocity-parsers* to perform the actual TSV parsing.

Listing 4.4: Illustration of the TSV data set provided by Statnett.

x\z [m]						
B [mT]	0.000	0.010	0.020	(... 3996 ...)		40.000
−20.000	0.059	0.059	0.059	0.059	0.059	0.036
−19.990	0.059	0.059	0.059	0.059	0.059	0.036
... (3997) ...						
20.000	0.057	0.057	0.057	0.057	0.057	0.036

The core of *CsvFieldParser* is its *parse* method that accepts a *java.io.Reader* object, and returns a *ScalarArrayField* that represents the TSV data in memory. This is a concrete class that implements the interface using a primitive double array, and in turn **DiscreteField**.

4.3 Code Abstractions

In this section we will explore the motivation behind the object-oriented abstractions within the implementation, especially with regards to the balance between performance and good design.

We will also cover parts of the implementation that powers the plotting feature, and not just the final position algorithm.

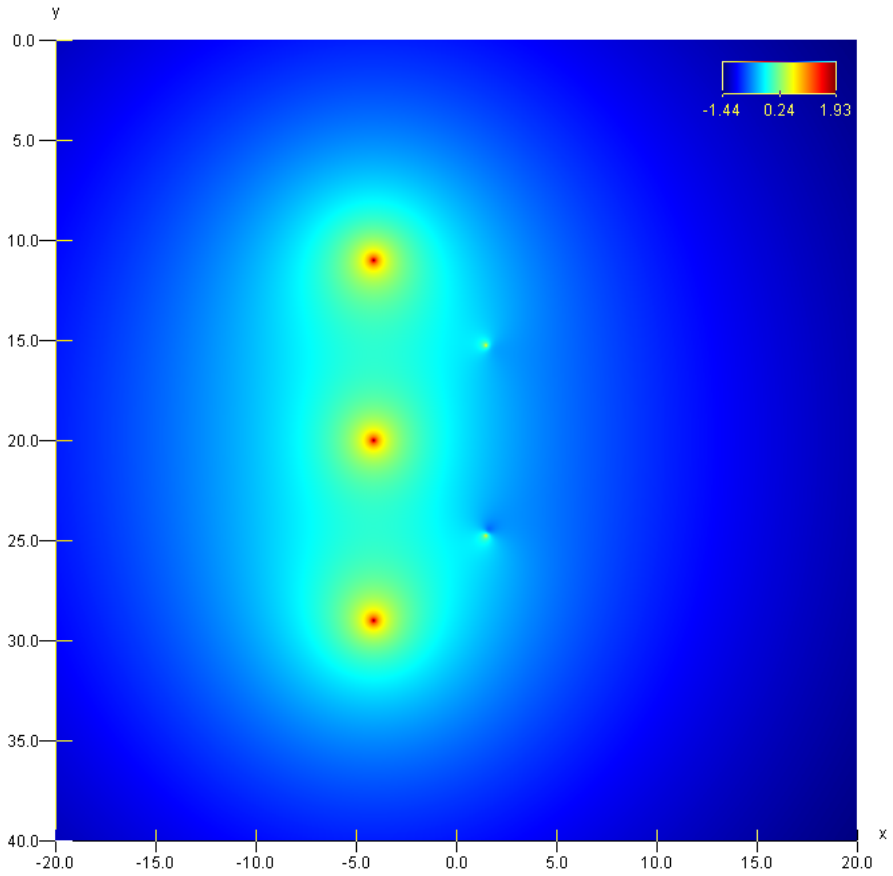


Figure 4.3: Log-plot of the data set provided by Statnett. Note that the field strength is measured in microtesla, and that the x and y coordinates of each wire have been accidentally flipped.

4.3.1 Scalar Field Abstractions

The interface **ScalarField**, (see listing 4.6) a specialization of the more general interface **DiscreteField** (see listing 4.5), is an abstraction over a discrete grid of cells of floating point values, where each column or row is associated with a corresponding x or y coordinate, respectively.

This abstraction allows us to treat pre-computed data, such as data loaded from a TSV file, exactly the same as data generated on-the-fly from a virtual model. In addition, the **DiscreteField** interface naturally associates meta-data such as the x and y coordinate of each cell, making it easier to supply this information to algorithms and the graphical plotting. This approach also has the advantage of expose algorithms directly on the object (such as *ScalarField.getRange*), with the ability to specialize each algorithm with a more optimized implementation in descendant concrete classes.

Scalar Field Methods

Note the use of default methods in listing 4.6. This is a new feature introduced in Java 8 that let interfaces specify a default behaviour in the absence of a concrete implementation, originally intended to enable interface evolution without sacrificing binary compatibility. We can use it in place of abstract classes, and write a sensible default

Listing 4.5: Essential methods in the discrete field interface. See appendix listing A.2 for more details.

```
public interface DiscreteField {
   double getColumn(int column);
   double getRow(int row);
   int getColumns();
   int getRows();
```

Listing 4.6: Interface of a scalar field. See listing A.3 for more.

```
public interface ScalarField extends DiscreteField {
   double getValue(int column, int row);

   default void fillValues(int startColumn, int endColumn,
         int startRow, int endRow, double[] outputValues) {
      /* Use getValue() to fill the array */
   }
```

algorithm for each method, making it easier to implement the interface. The comments in the code listings are summaries of the actual implementation in code.

We've also left out all of the less relevant methods in the ScalarField interface that deal with conversion to other representations (*stream*, *rows*, *toDoubleArray* and *toArrayField*), or perform aggregate computation (such as *getRange*). They can all be found in the more complete code listing A.3 in the appendix.

The two important methods of the interface are *getValue* and *fillValues*. Still, only *getValue*, which simply returns the value of a cell at a given row and column, is actually required, as *fillValues* can be implemented in terms of *getValue*.

Performance Considerations

The primary motivation for adding *fillValues* is performance - the computation in some implementations may depend on intermediate values that are shared amongst a subset of every, but critically not every cell. In that case, the fill method may be able to exploit this fact and reuse these intermediate values, unlike the direct method.

Additionally, on modern CPUs, the time needed to process an instruction have significantly outpaced the time it takes to look up a single section of data in RAM; as a result, data must be preloaded in much faster (but smaller) caches to avoid CPU stalls, where the CPU must waste significant amount of time waiting for the main memory. It is thus imperative to keep the amount of memory needed to be loaded in cache to a minimum, or we can suffer at worst a hundred fold reduction in performance as CPU stalls begin to dominate. This fact led us to the design of *fillValues* - by allowing an object to do a significant chunk of processing before passing on the torch, we can avoid the need to access the memory of multiple objects too often when we construct a chain of scalar fields, each processing the value of the next. And it is exactly this of construction that is useful in OOP, as it divides up the computational responsibility (and complexity) among multiple objects.

However, one should also consider the impact of over using *fillValues*. It could lead to the overhead of copying data back and forth from main memory overshadowing any savings, especially given that a simple sequence of *getValue* calls would primarily use

the CPU registers instead. With that in mind, we should always turn to benchmarks - first to determine if there even is an issue, and secondly to choose between different performance optimizations [14].

4.3.2 Vector Field Abstractions

Continuous Vector Field

The discrete field abstractions can be useful, they are not always appropriate. Many numerical optimization algorithms, such as BOBYQA [22], cannot be applied to a discrete grid of values if they assume very close points have similar value, but are nevertheless different. This expectation is clearly violated on a grid when we compare points on the same cell.

Listing 4.7: Interface of a continuous vector field. See listing A.4 for more

```
1 public interface ContinuousField {
2    int getComponents();
3
4    void addComponents(double x, double y, int startComponent,
5          int endComponent, double[] destination,
6          int destinationOffset);
```

It is also necessary to prove that our algorithms work on a continuous field, as this is better representation of magnetic fields in reality. Finally, we need the ability to represent a vector field, where each point in space is associated with a vector instead of a scalar value.

Arguably the most elegant design would be a straight forward getter method that accepts a coordinate to a point, and return the corresponding vector value, as seen in listing 4.8.

Listing 4.8: Simple definition of a vector field.

```
1 interface SimpleVectorField {
2  /* RealVector is an object that represents a
3     n-dimensional vector */
4    RealVector getVector(double x, double y);
5 }
```

Unfortunately, we cannot represent an n-dimensional vector in a primitive type (such as double), so the return type must be a reference type. Here it is a RealVector from Apache Math, which can cause additional object allocations for each method call. Admittedly, this is not necessarily an issue on a modern JVMs - garbage collectors are typically optimized to handle a large number of short-lived objects, and the JIT compiler can even move allocations off the heap through escape analysis [14]. But we are potentially dealing with millions of objects, all of which may be temporarily stored in an array and finally transformed to a scalar field. The flyweight pattern might be a better fit here, or we can simply flatten the vectors into a single primitive array, as they're all of the same length.

We chose the latter in **ContinuousField** by declaring a method named *addComponents* (see listing 4.7), as well as exposing the simple getter method above as *getVector)*.

Instead of returning an object, we add the components of the vector to a given double array starting at a specified index. Specifying the starting index allows us to store the vector components of multiple points in a single flattened double array.

There is also an overloaded method that includes the parameters *startComponent* (a) and *endComponent* (b). This instructs the method to write every component with an index i within the interval $[a, b)$, where i is a zero-based.

Four Dimensional Continuous Field

If the balance of current in the three conductors turns out to be crucial, we may need to represent the balance (or imbalance) as two unknown dimensions a and b (defined in subsection 3.5.2.

In order to compute the magnetic field of a point, subject to a different a and b, we add these dimensions to *addComponent* in an extended version of the interface **ContinuousField**, called **FourDimensionalField**. This interface can be seen in code listing 4.9.

Listing 4.9: Interface of a four dimensional vector field.

```
1 interface FourDimensionalField extends ContinuousField {
2    void addComponents(double x, double y, double z, double t,
3    int startComponent, int endComponent, double[] destination, int
        destinationOffset);
4 }
```

Continuous Field of a Wire

With these basic building blocks, we can begin translating the mathematical modelling of the magnetic field to Java. The magnetic field surrounding a single wire is computed by *WireContinuousField*, a concrete class based on **ContinuousField**, and is based on equation 2.9 and 2.10 when $|d| \geq R$, and equation 2.17 otherwise. The result can be seen in listing 4.10.

Note that both components of the magnetic field vector reuse the same three intermediate variables - dX, dY and $r2$. Incidentally, this is also why we discarded a simpler possibility: a method that returned a given component of a vector in point (x, y). In that case, we would have been forced to recompute the intermediate values for each component in the vector.

Discrete Vector Fields

We can also define an analogous vector field interface for the discrete case by combining **ContinuousField** and **ScalarField**, resulting in the **VectorField** interface in listing 4.11.

Callers of method *fillComponents* specify the region of vector points of interest, using an interval of columns and rows, coupled with a interval of components, that should be flatten into an array of floating points and copied one-by-one to the destination array starting at the given destination index.

The main purpose of permitting a component interval, is to simplify the process of extracting a single component off every vector in the field. This is, among other things, needed when specifying x or y as a value flag to our CLI.

Individual vector components are copied first, then vectors by column and then finally by row. This is all illustrated in figure 4.4.

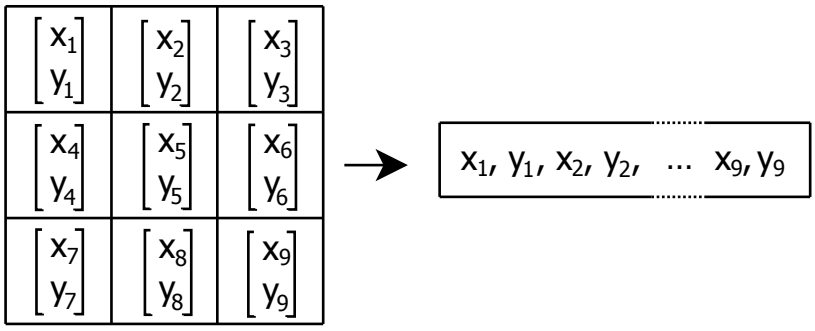


Figure 4.4: Illustration of the distinct way data is structured in the destination array of *fillComponents*.

Discrete Magnetic Field for Multiple Wires

In order to simulate the magnetic field of multiple wires, we create a concrete class *VectorSimulatedField* that implements the previously mentioned **VectorField** interface. As established in section 2.3, the aggregate magnetic field is simply equal to the summation of the magnetic field contributions from each wire. We can replicate this by accepting a collection of *WireContinuousField* objects, and then add their components in *fillCompnents*, as depicted in code listing 4.12.

Using the log-modulus transformation (selected using the flag *abslog*), we can plot the individual components of the magnetic field in this class on a logarithmic scale. The result is given in figure 4.5.

4.3.3 Positioning Abstractions

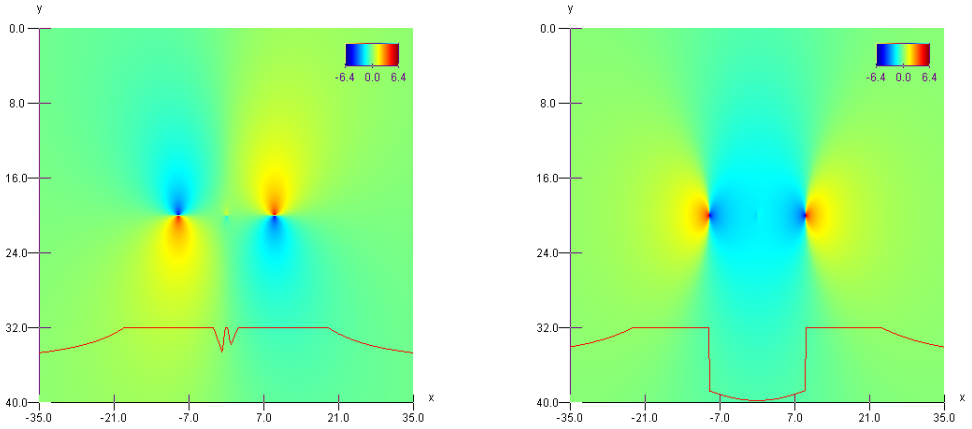
The final abstraction is a vector field that represents the sinusoidal magnetic field strengths in both components, which will be essential to compute the magnetic field invariants found in chapter 3.

Sinusoidal Vector Field

Equation 2.28 and 2.31 established that the magnetic field strengths is a sinusoid, which can represent in a vector field consisting of these four quantities as components: A_x, δ_x, A_y and δ_y. The angular frequency ω can safely be disregarded, as it is approximately constant and independent of the current position.

To model this, we introduce the class **HarmonicSumField** that implements the interface *ContinuousField*, and, similar to **VectorSimulatedField**, uses a collection of *WireContinuousField* objects to compute A_{ix} (given by equation 2.44 and 2.45).

Then it is simply a matter of translating equation 2.35 and 2.36 into *addComponents*. The only real issue is correctly handling the interval of components to extract (*startComponent* and *endComponent*), and recognize that we can cache the cosines in equation 2.35 through a lookup table. The very core of this implementation can be seen in listing 4.13.



(a) The x component.

(b) The y component.

Figure 4.5: The modified log-modulus transformation plot of the x and y components of the magnetic field in our power line example, 1/3 into the cycle.

We can become more confident in the the correctness of our implementation, by extracting and comparing the exact same components from *VectorSimulatedField* using the FFT instead (see section 5.1). This was actually done in figure 3.6, so all that remains is to plot the phase difference of the second and fourth component in the **HarmonicSumField**, which can be seen in figure 4.6.

If we compare the two figures, we see that the phase difference is the same whether or not we use the FFT or the Harmonic Addition Theorem, as implemented in our program. Thus, we can be reasonably certain in our implementation.

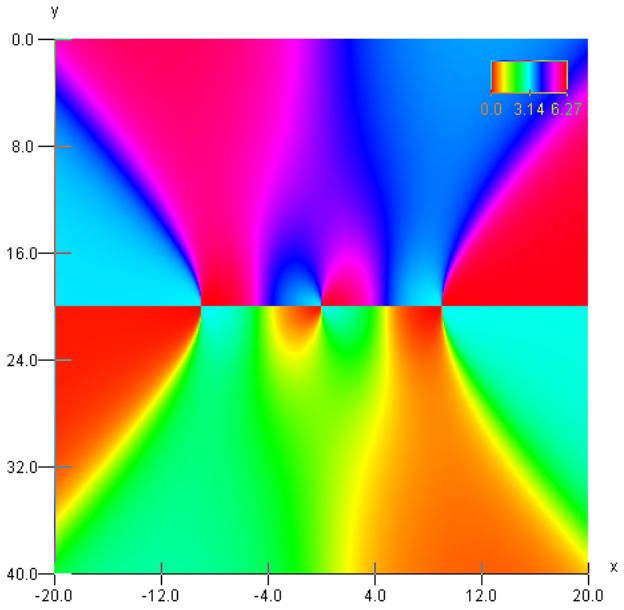


Figure 4.6: Plot of the angle difference of the second and fourth component in **HarmonicSumField**. Should be identical to figure 3.6.

Drone Vector Field

We can transform the sinusoidal parameters in **HarmonicSumField** to yet another vector field - the **DroneVectorField**. Here, each component in a point vector represent the input to the position algorithm, which will mainly be the amplitude angle 3.5 and/or the phase difference 3.9 of one or more points in **HarmonicSumField**.

We will determine the optimal definition of the input components later, particularly with regards to the number of points (and their relative offset) to extract from the harmonic sum field.

4.3.4 Power Line Model

One missing ingredient is the details of the power line model, such as the position and current of each of the conductors. This is a necessary input to every *WireContinuousField* object that in turn constitute the basis for every higher level vector field.

We can supply these missing values using a simple XML configuration file - the file in listings 4.14 have been used to produce every density plot up until this point. It reflects the position and currents of table 2.1 by specifying the number of conductors in the x–y plane plane, their individual position (x and y), wire thickness (*radius*) and the state of the alternating current.

This current is provided as a sinusoid, where *amp* is the amplitude, in Ampers, *freq* is the angular frequency in degrees per second, and *phase* is the is the phase displacement in degrees. There is also a DC offset (*dc*), which is simply a constant current added to the sinusoid, yielding the following equation for the current in a wire:

$$I = amp \cdot \sin\left(freq \cdot t + phase\right) + dc$$

4.3.5 Plotting System

As mentioned previously, we had to construct a custom system for plotting the density of x–y planescalar fields to get decent performance when generating high-resolution animations. This also informed the design of the vector field abstractions, as explained in section 4.3.1.

The plotting itself is controlled by the *FieldGraph* class. It is cable of taking a **ScalarField**, a region in XYZ space, a value mapping function (such as $\ln x$) and whether or not to include XYZ axes, and produce a **BufferedImage** as a result.

These classes and interfaces are used to support the graphing operation:

- **Gradient,** An interface representing a mapping of normalized values in the interval $0 \leq x \leq 1$ to an RGB color. This is used to colourize density plots.
- **RainbowGradient,** This is the main gradient used throughout the paper, mapping a value to the red, green and blue color channel using trapezoids as seen in figure 4.7.
- **GraphAxis,** Represents the thickness, number of ticks and label of an x or y axis.
- **LineGraph,** A class that can draw a simple curve, such as the red curve in figure 2.9.

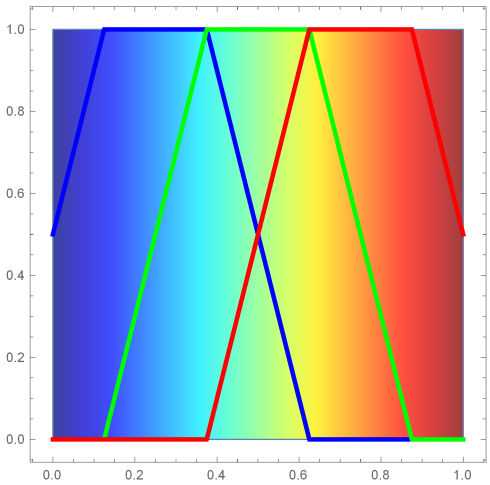


Figure 4.7: Mapping of the red, green and blue color channel in *RainbowGradient*

Listing 4.10: A magnetic vector field around a wire.

```
public class WireContinuousField implements ContinuousField {
   /* ... */

   public WireContinuousField(double centerX, double centerY,
         double wireRadius, double wireCurrent, double permeability)
            {
         /* ... */
   }

   public int getComponents() {
      return 2;
   }

   public void addComponents(double x, double y,
         int startComponent, int endComponent,
         double[] destination, int destinationOffset) {
      double dX = x - centerX;
      double dY = y - centerY;
      double r2 = dX * dX + dY * dY; // Always positive

      if (r2 < wireRadiusSquared) {
         // This factor is negative when the current is negative,
         // rotating the resulting vector 180 degrees
         if (startComponent == 0) {
            destination[destinationOffset++] += dY * factorWire;
         }
         if (endComponent == 2) {
            destination[destinationOffset] += -dX * factorWire;
         }
      } else {
         if (startComponent == 0) {
            destination[destinationOffset++] += dY * numeratorSpace
               / r2;
         }
         if (endComponent == 2) {
            destination[destinationOffset] += -dX * numeratorSpace /
               r2;
         }
      }
   }
}
```

Listing 4.11: Interface of a vector field. See listing A.5 for more

```
public interface VectorField extends DiscreteField {
   int getComponents();

   void fillComponents(int startColumn, int endColumn,
         int startRow, int endRow,
         int startComponent, int endComponent, double[] destination,
         int destinationOffset, boolean zeroDestination);
```

Listing 4.12: Computing the total magnetic field by bridging *WireContinuousField* and **VectorField**.

```
public class VectorSimulatedField implements VectorField {
   public VectorSimulatedField(Collection<ContinuousField> sources,
         double[] xCoordinates, double[] yCoordinates) {
      /* ... */
   }

   public void fillComponents(int startColumn, int endColumn,
         int startRow, int endRow,
         int startComponent, int endComponent, double[] destination,
         int destinationOffset, boolean zeroDestination) {

      /* ... */

      for (int i = 0; i < sources.size(); i++) {
         ContinuousField source = sources.get(i);
         int position = destinationOffset;

         for (int row = startRow; row < endRow; row++) {
            for (int col = startColumn; col < endColumn; col++) {
               source.addComponents(xCoordinates[col],
                  yCoordinates[row], startComponent,
                  endComponent, destination, position);
               position += endColumn - startColumn;
            }
         }
      }
   }
}
```

Listing 4.13: Computing the parameters of the sinusoids in the magnetic field.

```
double amplitude = 0;
double phaseNumerator = 0;
double phaseDenominator = 0;

for (int i = 0; i < conductorList.size(); i++) {
  // Amplitude of conductor at index i
  double iConductor = conductors[i * condNumComponents + component];

  // Prepare phase
  phaseNumerator += iConductor * sineTable[i];
  phaseDenominator += iConductor * cosineTable[i];

  // Self-contribution
  amplitude += iConductor * iConductor;

  for (int j = i + 1; j < conductorList.size(); j++) {
    amplitude += iConductor * conductors[j * condNumComponents +
          component] * differenceTable.getCosine(i, j);
  }
}
```

Listing 4.14: A simple model of a power line.

```
<?xml version="1.0"?>
<model>
    <conductors>
        <conductor x="-9.0" y="20.0" radius="0.04">
            <current amp="12000" freq="18000"
                     phase="0" dc="0" />
        </conductor>
        <conductor x="0" y="20.0" radius="0.04">
            <current amp="12000" freq="18000"
                     phase="120" dc="0" />
        </conductor>
        <conductor x="9.0" y="20.0" radius="0.04">
            <current amp="12000" freq="18000"
                     phase="240" dc="0" />
        </conductor>
    </conductors>
</model>
```

Chapter 5

Algorithms

Now that we have constructed an accurate model of the magnetic field, along with feature invariants that only depend on the current position, we can start evaluating different optimization algorithms that can reverse the process and correlate a feature with a position.

We will mostly bypass all the minute details of the implementations underlying each algorithm in this chapter, as they are provided by the open-source project Apache Commons Math, and instead focus on their theoretical foundations.

5.1 Fast-Fourier Transform

This essential signal-processing algorithm is so ubiquitous it almost needs no introduction. Up until this point, we have also only used it to verify our work on **HarmonicSumField**—by comparing its output to the sinusoid in **VectorSimulatedField**. But it will become useful later when we need to extract the amplitude and phase of the magnetic field through a magnetometer.

The FFT is an efficient algorithm for computing the Discrete Fourier Transform, which is considered a special case of the continuous Fourier transform theory, defined by the this Fourier integral [6]:

$$H(f) = \int_{-\infty}^{\infty} h(t) e^{-j2\pi ft}\, dt \tag{5.1}$$

Here, $H(f)$ is a complex-valued function that encode the amplitude and phase of a every wave that constitute the frequency domain of $h(t)$. It is also possible to recover the time-domain $h(t)$ from $H(t)$ through the inverse Fourier Transform:

$$h(t) = \int_{-\infty}^{\infty} H(f) e^{j2\pi ft}\, df \tag{5.2}$$

Keep in mind that $h(t)$ can be complex-valued as well.

In the discrete case, we extract N samples from a periodic function $h(t)$ at a fixed interval T over a duration T_0, ensuring that $1/T$ is at least twice the highest frequency in $h(t)$ (sampling theorem) to avoid time aliasing.

$$\hat{H}\left(\frac{n}{NT}\right) = X_k = \sum_{k=0}^{N-1} h(kT) e^{-j2\pi n \frac{k}{N}} \quad n = 0, 1, \cdots, N-1 \tag{5.3}$$

This relies on the connection between the Fourier Transform and the Fourier Series, which also allows us to extract the amplitude A and phase ϕ of the sinusoid we were looking for in section 4.3.3:

$$A = \frac{|X_k|}{N} = \frac{\sqrt{\Re(X_k)^2 + \Im(X_k)^2}}{N} \tag{5.4}$$

$$\phi = Atan2(\Im(X_k), \Re(X_k)) \tag{5.5}$$

5.2 Covariance Matrix Adaptation Evolution Strategy

The CMA-ES is an optimization method that can find the global minimum of a real-valued cost n-dimensional function f, without requiring a gradient or a Jacobian. It is centred around an iterative process where a subset of λ random candidate solutions sampled from a distribution are selected based on the corresponding function value to influence the sampling of the generation, akin to the process of natural selection by evolution. [13]

It belongs to the family of randomized search algorithms, and is ideal to optimize "noisy" functions that exhibit discontinuities or ridges, and many local optima. This includes function with a very high and erratic gradient (ill-conditioned) that cannot be separated into n 1-dimensional problems (which is significantly easier to solve).

Figure 5.1 illustrates how the algorithm is able to converge on a global optimum over a short number of generations.

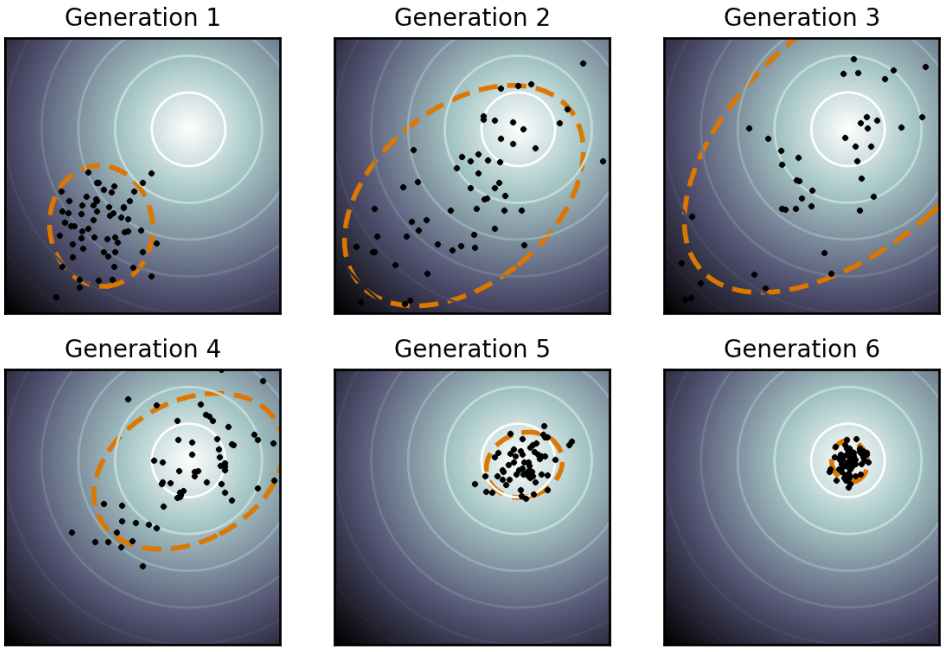


Figure 5.1: Figure illustrating how the CMA-ES algorithm converges on the global optimum on a spherical two-dimensional function. Note how the 3-sigma ellipsoid of the multivariate normal distribution (dotted lines) changes over the generations. [23]

5.2.1 Initial Parameters

Before introducing the main iterative loop, we need to define every parameter that tune the evolutionary algorithm, like the population count (λ) and step-size (ω) (which control the mutation rate), and their default values that have been experimentally determined. However, the specific problem domain may call for values that differ with these defaults.

The population count, and the number of selected candidates (μ), depend on the dimensionality of the fitness function (n):

$$\lambda = 4 + \lfloor 3 \ln n \rfloor \tag{5.6}$$

$$\mu = \left\lfloor \mu' \right\rfloor, \quad \mu' = \frac{\lambda}{2} \tag{5.7}$$

Next, the contributions of each candidate that survives the selection phase is weighted according to ω_i, where i is their individual ranking given the function values; the point associated with the lowest function value has the first rank ($i = 1$), and so on:

$$\omega_i = \frac{\omega_i'}{\sum_{j=1}^{\mu} \omega_j'}, \quad w_i' = \ln(\mu' + 0.5) - \ln i \quad \text{for } i = 1, \cdots, \mu \tag{5.8}$$

Where $\sum_{i=1}^{\mu} \omega_i = 1$. We also dynamically alter the rate the distribution adapts to new information, in order to maximize performance. The core idea is to keep track of the length of the evolution path, loosely stated as the total distance the mean of the stochastic distribution has translated due to evolution (discounting ω), and compare it against the expected path length as if the distribution was static and not subject to evolution.

If the evolution path is shorter than expected, then we must be close to the global minimum and we are very likely walking back and forth in a circle, cancelling steps. As a result, the step size should be decreased. We have the opposite case when the evolution path is longer than expected — then most steps will be pointing in the same direction, and we ought to increase the step size in order to get to the minimum faster:

$$c_\omega = \frac{\mu_\theta + 2}{n + \mu_\theta + 5}, \quad d_\omega = 1 + 2 \max\left(0, \sqrt{\frac{\mu_\theta - 1}{n+1}} - 1\right) + c_\omega \tag{5.9}$$

Where u_θ is the variance effective selection mass:

$$u_\theta = \frac{1}{\sum_{i=1}^{\mu} \omega_i^2} \geq 1 \tag{5.10}$$

We also need to specify the learning rate of both the rank-one update and the rank-μ-update to ensure old information is not discarded immediately:

$$c_c = \frac{4 + \mu_\theta / n}{n + 4 + 2u_\theta / n} \tag{5.11}$$

$$c_1 = \frac{2}{(n + 1.3)^2 + \mu_\theta} \tag{5.12}$$

$$c_\mu = \min\left(1 - c_1, \alpha_\mu \frac{\mu_\theta - 2 + 1/\mu_\theta}{(n+2)^2 + \alpha \mu_\theta / 2}\right) \quad \text{where } \alpha_\mu = 2 \tag{5.13}$$

5.2.2 Main Iteration

Once all the external parameters have been set to their appropriate values, we can begin iterating through the generations until we have reached some specific termination goal, generally when additional CPU time would not have yield any further results.

Initialization Before the first generation, let the evolution path be $p_\omega^{(g=0)} = 0$ and the conjugate evolution path be $p_c^{(g=0)} = 0$. The covariance matrix of the multivariate normal distribution X, $\mathbf{C}^{(g=0)}$, should be set to the identity matrix $\mathbf{I}$. The generation index starts at zero, and is incremented by one.

It is also necessary to supply an initial starting point $m^{(g=0)} \in \mathbb{R}^n$ that will serve as the mean of the random distribution, and a initial step-size $\omega^{(g=0)} \in \mathbb{R}$. Note the subscript $x^{(g=0)}$ — this limits the scope to the initial generation, and suggest the variable will change in the next generation.

Sampling The first step in generation g is to sample λ points, with index $k = 1, \cdots, \lambda$, from the stochastic distribution X:

$$\mathbf{z_k^{(g)}} \sim \mathcal{N}(0, \mathbf{I}) \tag{5.14}$$

$$\mathbf{y_k^{(g)}} = \mathbf{B^{(g)}D^{(g)}}z_k \sim \mathcal{N}(0, \mathbf{C^{(g)}}) \tag{5.15}$$

$$\mathbf{x_k^{(g)}} = \mathbf{m^{(g)}} + \sigma^{(g)}\mathbf{y_k^{(g)}} \sim \mathcal{N}(m^{(g)}, (\sigma^{(g)})^2\mathbf{C^{(g)}}) \tag{5.16}$$

Here, $\mathbf{B^{(g)}}$ and $\mathbf{D^{(g)}}$ are based on an eigen-decomposition of the covariance matrix $\mathbf{C^{(g)}}$:

$$\mathbf{C} = \mathbf{BD}^2\mathbf{B}^T = \mathbf{BDDB}^T \tag{5.17}$$

To be precise, $\mathbf{B^{(g)}}$ is an orthonormal basis of the eigenvectors, and $\mathbf{D^{(g)}}$ is a diagonal matrix consisting of the square roots of the corresponding eigenvalues.

Candidate Selection Then we order every point $\mathbf{x_k^{(g)}}$ such that $\mathbf{x_{i:\lambda}^{(g)}}$ is the i-th smallest point according to $\mathbf{f}(\mathbf{x_{i:\lambda}^{(g)}})$, and select a subset μ of the result:

$$\langle \mathbf{y} \rangle_\mathbf{w} = \sum_{i=1}^{\mu} \omega_i \mathbf{y_{i:\lambda}^{(g)}} \tag{5.18}$$

$$\mathbf{m^{(g+1)}} = \mathbf{m^{(g)}} + \sigma^{(g)} + \langle \mathbf{y} \rangle_\mathbf{w} = \sum_{i=1}^{\mu} \omega_i \mathbf{x_{i:\lambda}^{(g)}} \tag{5.19}$$

Here, $\mathbf{y_{i:\lambda}^{(g)}}$ is simply:

$$\mathbf{y_{i:\lambda}^{(g)}} = \frac{\mathbf{x_{i:\lambda}^{(g)}} - \mathbf{m^{(g)}}}{\sigma^{(g)}} \tag{5.20}$$

Dynamic Step-Size Further, we ensure the step-size σ is optimal:

$$\mathbf{p_\sigma^{(g+1)}} = (1 - c_\sigma)\mathbf{p_\sigma^{(g)}} + \sqrt{c_\sigma(2 - c_\sigma)\mu_\theta}(\mathbf{C^{(g)}})^{-\frac{1}{2}}\langle \mathbf{y} \rangle_\mathbf{w} \tag{5.21}$$

$$\sigma^{(g+1)} = \sigma^{(g)} \times \exp\left(\frac{c_\sigma}{d_\sigma}\left(\frac{||\mathbf{p_\sigma^{(g+1)}}||}{E||\mathcal{N}(0, \mathbf{I})||} - 1\right)\right) \tag{5.22}$$

Where $(\mathbf{C^{(g)}})^{-\frac{1}{2}}$ is deduced from equation 5.17:

$$(\mathbf{C^{(g)}})^{-\frac{1}{2}} = \mathbf{BD}^{-1}\mathbf{B}^T \tag{5.23}$$

And the expectation of the Euclidian norm of a $\mathcal{N}(0, \mathbf{I})$ distribution is as follows:

$$E||\mathcal{N}(0, \mathbf{I})|| = \sqrt{2}\Gamma(\frac{n+1}{2})/\Gamma(\frac{n}{2}) \tag{5.24}$$

Covariance Matrix Adaptation And finally, we update the covariance matrix of the normal distribution X:

$$\mathbf{p_c^{(g+1)}} = (1 - c_c)\mathbf{p_c^{(g)}} + h_\sigma \sqrt{c_c(2 - c_c)\mu_\theta}\langle \mathbf{y} \rangle_{\mathbf{w}} \tag{5.25}$$

$$\begin{aligned}\mathbf{C^{(g+1)}} = (1 - c_1 - c_\mu)\mathbf{C^{(g)}} + c_1 \left(\mathbf{p_c^{(g+1)}} (\mathbf{p_c^{(g+1)}})^T + \delta(h_\sigma)\mathbf{C^{(g)}} \right) \\ + c_\mu \sum_{i=1}^{\mu} \omega_i \mathbf{y_{i:\lambda}^{(g)}} (\mathbf{y_{i:\lambda}^{(g)}})^T\end{aligned} \tag{5.26}$$

Where h is the Heaviside step function:

$$h_\sigma = \begin{cases} 1 & \text{if } \frac{||\mathbf{p}_\sigma^{(\mathbf{g+1})}||}{\sqrt{1-(1-c_\sigma)^{2\times(g+1)}}} < (1.4 + \frac{2}{n+1})E||\mathcal{N}(0, \mathbf{I})|| \\ 0 & \text{otherwise} \end{cases} \tag{5.27}$$

And δ is:

$$\delta(h_\sigma) = (1 - h_\sigma)c_c(2 - c_c) \leq 1 \tag{5.28}$$

Then we increment the generation index and repeat from candidate selection, until our end conditions have been satisfied.

Chapter 6

Results

We are now ready to test the feasibility of the magnetic field invariants we found in chapter 3.

6.1 Single Sensor

Let us evaluate the accuracy of a single sensor, located at (0, 0) relative to the drone position. We will extract the amplitude and phase of the sinusoids in both the x and y axis, using the FFT algorithm, and use them to look up the corresponding position in the model based on the field invariants phase difference (equation 3.9 and figure 3.6) and amplitude angle (equation 3.5 and figure 3.1).

We can visualize these two invariants by normalizing them (using the scale given by the figures), and taking the magnitude as a vector, as seen in figure 6.1:

$$\sqrt{\left(\frac{Z_\theta}{2\pi}\right)^2 + \left(\frac{2A_\theta}{\pi}\right)^2}$$

6.1.1 Evaluating BOBYQA and CMA-ES

To confirm the need for a stochastic algorithm such as CMA-ES, we will also attempt to solve this problem with the much simpler BOBYQA method.

BOBYQA Settings

For BOBYQA, we will select the largest number of interpolation points possible (6 in our case), and a stopping radius of 10^{-13}[meters]. We'll also set the maximum number of iterations to 200 000 and maximum number of function evaluations to 100 000, as a start. Code listing 6.1 shows how the algorithm is initialized from Apache Math.

CMA-ES Settings

To give the CMA-ES algorithm the best possible chance of success, we will initially set all the parameters to a very high level, such as the population count and convergence threshold. Later they may be decreased to improve performance, assuming the algorithm remains accurate. Specifically, the algorithm is set to a population count λ of 1000, a maximum generation count to 4000, and the absolute convergence threshold to $\Delta < 10^{-12}$. This can all be seen in code listing 6.2

Listing 6.1: Initialization of the BOBYQA algorithm.

```
1  private int interpolationPoints = 6;
2  public double stoppingRadius = 1E-13;
3
4  public PointValuePair findMinimum(ContinuousField field, double[]
       target, double[] guessPoint, BoundingBox boundingBox) {
5      BOBYQAOptimizer optimizer = new
           BOBYQAOptimizer(interpolationPoints,
           boundingBox.getExtent().getMaxValue(), stoppingRadius);
6
7      // Real solution
8      return optimizer.optimize(
9            new SimpleBounds(boundingBox.getLowerBound().toArray(),
                 boundingBox.getUpperBound().toArray()),
10           new InitialGuess(guessPoint),
11           new MaxIter(200000),
12           new MaxEval(100000),
13           new
                 ObjectiveFunction(ContinuousField.withSquareDistance(field,
                 target)));
14 }
```

Random Sampling

To test both approaches, we will take 10 000 random points from the region starting from point [-35, 0] to [34, 40], extract the corresponding field invariants, and then find the point with the closest invariants in the model using either BOBYQA and CMA-ES. We will let the wires be configured according to table 2.1.

Algorithm	A_θ	Z_θ	Avg Elapsed [ms]	Avg Miss [m]	Miss >1	Miss >10	Failures
BOBYQA		✓	2.97	30.38	8060	7499	1936
BOBYQA	✓		**2.18**	23,52	9595	8727	393
BOBYQA	✓	✓	2.78	32.16	9868	9274	129
CMAES		✓	76.77	8.28	8894	2818	**0**
CMAES	✓		209.00	26.67	9751	7803	**0**
CMAES	✓	✓	95.93	**3.61**	**2623**	**1602**	**0**

Table 6.1: Testing BOBYQA and CMAES with 10000 random points. The columns A_θ and Z_θ indicate whether or not we used the amplitude angle and phase difference in our search. "Miss" is the distance from the correct point to the point found by the algorithm.

Figure 6.1 displays the result of this test. We also test the effect of using either the amplitude angle (A_θ), the phase difference (Z_θ), or both at the same time.

It is evident that BOBYQA is not suitable for this problem. In fact, out of just 10 000 samples, it was unable to find more than 4 points with a miss distance of less than 1 meters. Not to mention the 1% - 19% error rate, caused by an inability to reduce the trust region.

In contrast, CMA-ES is tremendously more effective, and is able to accurately find the correct point within 1 meters in more than 73.7% of the time when using both A_θ and Z_θ. Not surprisingly, its accuracy drops heavily (0.02%) when it is only relying

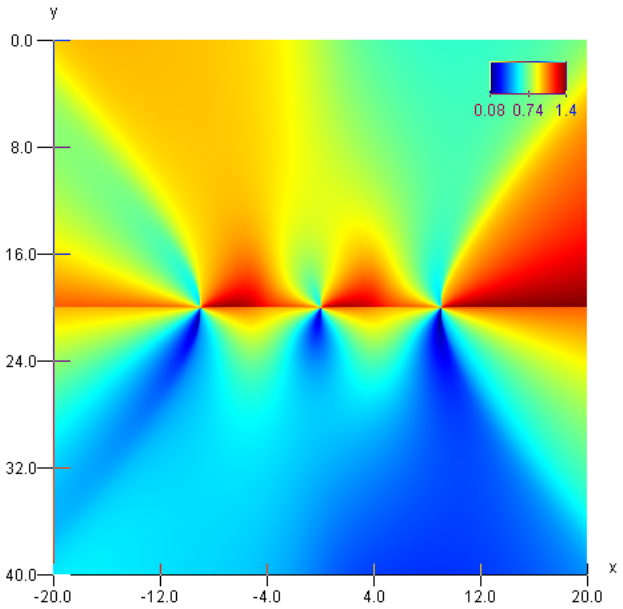


Figure 6.1: Visualization of both the phase difference and the amplitude angle. This normalizes both as components in a vector, and displays the magnitude of said vector.

on the amplitude angle, as this invariant does not differ significantly from quadrant to quadrant. Phase difference Z_θ does a lot better on its own, but it is still less accurate combining the two invariants.

We can also plot the miss distances in a coordinate system, as seen in figure 6.2. The problematic points seems to be concentrated along the diagonals, and in an ellipsoid around the center.

6.1.2 Points of High Inaccuracy

In order to improve the accuracy of the CMA-ES algorithm, we must determine why the result was more than 10 meters off the correct point in 16% of the cases. If we look at the top 10 most inaccurate results (table 6.2), we see that the field invariants between the original point and the incorrect result point are almost identical. We have therefore likely exhausted the potential of algorithmic improvement, and we must turn elsewhere to get a better result.

6.2 Multiple Sensors

It appears we need additional information to be able to improve the accuracy of our method. One approach is to install multiple sensors on the drone, with known relative offsets from each other.

Like in section 3.2, we will use the convention $(a,b)(c,d)\cdots(x,y)$ to indicate a set of sensors with relative positions, and after trying out a couple of different configurations (see table 6.3) where we assume the maximum relative offset is about 1 meters (a decent size for a drone), we seem to hit diminishing returns after 3 or 4 sensors.

Beside the average miss distance, we also should focus on minimizing the number of miss distances that exceed 1 meter.

Listing 6.2: Initialization of the CMA-ES algorithm.

```
private int iterations = 2000;
private int populationSize = 4000;
private double threshold = 0.000000000001;

public PointValuePair findMinimum(ContinuousField field,
  double[] target, double[] guessPoint, BoundingBox boundingBox) {
    CMAESOptimizer optimizer = new CMAESOptimizer(iterations, 0,
        true, 0, 0, new JDKRandomGenerator(), false, new
        SimpleValueChecker(-1, threshold));

    // Real solution
    return optimizer.optimize(
          new SimpleBounds(boundingBox.getLowerBound().toArray(),
              boundingBox.getUpperBound().toArray()),
          new InitialGuess(guessPoint),
          new CMAESOptimizer.Sigma(boundingBox.getExtent().toArray()),
          new CMAESOptimizer.PopulationSize(populationSize),
              GoalType.MINIMIZE,
          new MaxIter(2000000),
          new MaxEval(10000000),
          new ObjectiveFunction(ContinuousField.withSquareDistance(
                field, target)));
}
```

6.2.1 Problematic Points

Even five sensors (S_5) is unable to completely eliminate the error rate, but is this due to insufficient information or stochastic errors? We can attempt to answer this by repeatedly applying the CMA-ES algorithm to the point with the highest miss distance in run S_5, which happened to be (7.1909021317, 21.6243214568), and plot the miss distance against the distance between the target field invariants and the field invariants of the resulting point.

This plotting can be seen in figure 6.3, and it does appear to solidify our intuition that two sensors or more are necessary. Specifically, the correlation between a lower invariant distance and a low miss distance is hampered by the two blue points near 10 meters in miss distance, which also have a fairly low invariant distance. The cluster of incorrect points for S_2 are, in contrast, much higher up. There's also only a single cluster, unlike S_1, which have an additional cluster near 40 meters.

6.3 Improving Accuracy

Now that we have established the need for multiple sensors, we can begin to improve its accuracy further. We present two general approaches.

6.3.1 Select the Lowest Invariant Distance

If the correlation between accuracy and invariant distance visible in figure 6.3 is true, we could simply repeat the CMA-ES algorithm N times and select the result with the lowest invariant distance.

This does actually improve the accuracy, as seen in table 6.4, but at the cost of a

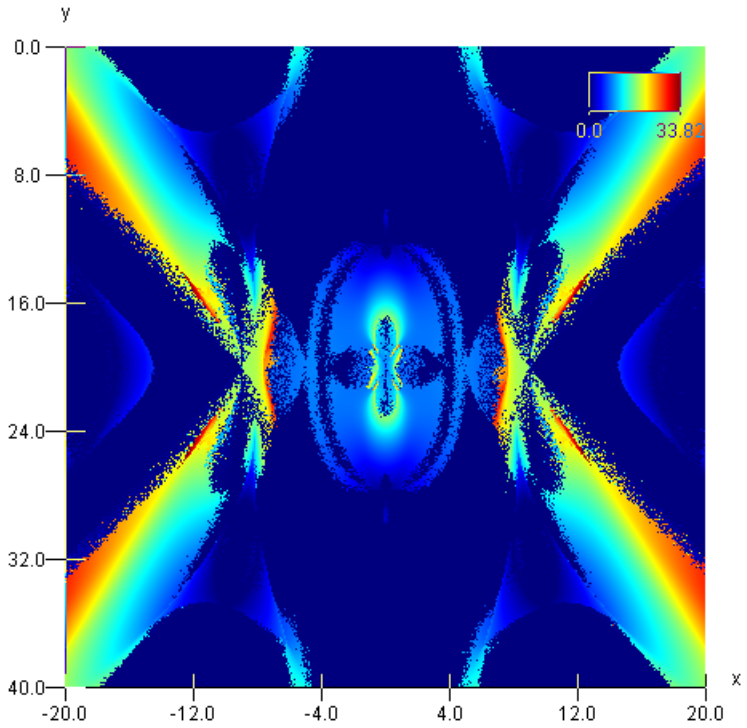


Figure 6.2: Plot of miss distances when using CMAES and the two field invariants.

significant increase in CPU time. Though, the most valuable improvement is the lower number of misses that exceed 10 meters.

One possible optimization is find some acceptable level of invariant distance (say, 10^{-5} m), and only repeat the computation if the distance is above this limit.

6.3.2 Restricting the Search Area

Up until now, we have allowed the algorithm to scan the entire area of interest (-35, 0) to (35, 40), as we cannot necessarily guarantee that the GPS device will produce a more accurate fix on our location (especially in the vertical axis).

However, once we have begun to pin down our position more precisely due to the relative offset calculated by our method, we could use a smaller search area in our algorithm using the previous known approximate position and a INS (coupled with GPS) to estimate the maximum distance we could have travelled. If we also account for the maximum drift of the INS device, and use the bearing of the drone, we can determine just how far the drone could have travelled in the x–y plane, and shrink the search area accordingly.

Table 6.5 suggests that this is a particularly successful method of getting a high accuracy, so we will attempt to integrate it in our final position algorithm.

6.3.3 Virtual Sensors

A third option is to try to sample the magnetic field invariants from a larger area than is ordinary possible with multiple on-board sensors, specifically by travelling a set distance and measure the field at regular intervals.

X_0	Y_0	Z_0	A_0		X_r	Y_r	Z_r	A_r
7.8221	19.3169	3.2425	1.0851		-12.1178	18.4412	3.2441	1.0853
7.9134	19.5518	3.2225	1.2047		-35.0000	9.0976	3.2277	1.2054
7.7232	20.7658	0.1140	1.0764		-33.7411	34.8873	0.1140	1.0764
8.0429	20.2188	0.0624	1.3578		-35.0000	26.0632	0.0725	1.3583
6.9213	17.8390	3.4551	0.9220	⇛	-28.9125	0.2126	3.4299	0.8705
6.8622	22.3903	0.3431	0.9032		6.7668	22.5136	0.3684	0.9097
6.6792	17.2851	3.5424	0.9053		6.7547	17.4480	3.5154	0.9076
7.8488	19.9612	3.2176	1.5393		-29.6948	19.3075	3.2176	1.5393
-6.4891	23.7175	2.6202	0.8523		26.0178	40.0000	2.6296	0.8031

Table 6.2: The ten most inaccurate results in the CMAES run of two invariants and one sensor. Here, X_0 and Y_0 are the coordinates of the test point, and Z_0 with A_0 are the two field invariants. The resulting point (X_r and Y_r) have the invariants Z_r and A_r.

Sensor Configuration	Avg Elapsed (ms)	Avg Miss (m)	Miss >1	Miss >10
S_1 = (0,0)	95.9302047	3.614360296	2623	1602
S_2 = (0,0)(1,0)	115.8416916	0.431728231	508	118
S_3 = (0,0)(1,0)($\frac{1}{2}$, $\frac{1}{2}$)	137.8660531	0.280135445	502	53
S_4 = (0,0)(1,0)(1,1)(0,1)	146.9185581	0.205809262	366	41
S_5 = (0,0)(1,0)(1,1)(0,1)($\frac{1}{2}$,$\frac{1}{2}$)	152.9552156	0.226309288	432	37

Table 6.3: The effect of adding additional sensor inputs with known relative offsets.

If we first orient the drone such that is it parallel to the conductor wires, and then ascend vertically to a height well below the conductors, taking measurements every 3 meters. This would effectively create a large number of virtual sensors by sacrificing temporal accuracy.

Note that this method requires some knowledge of the speed and acceleration of the drone over the period of measurements, or we would not be able to specify the relative offset of each sample. It also cannot handle changing conductor positions in the x–y plane, which we pretty much expect when the drone is at high speed (due to the wire hang).

We perform three different tests, with either zero or five repeats per point each. The first test requires a single sensor at (0, 0), and that the drone moves 9 meters up or down, sampling the magnetic field every 3 meters. The second test uses two sensors (0, 0) and (1, 0) over the same distance, resulting in 8 overall virtual sensors. In the last test we have the drone move horizontally instead, again sampling at every 3 meters. The result can be seen in table 6.6.

Interestingly enough, the horizontal sensors are an order of magnitude more accurate than the virtual sensors. This approach will be useful if we are already hovering above the power line at a constant height, and enter from the left or right in the x–y plane. Then we only need to fly 9 meters to accurately pin-point our location.

However, if we happen to launch the drone underneath or near the power line, it might be prudent to use the vertical virtual sensors to get an more approximate fix on a position. That might theoretically allow us to ascend from the ground automatically, without having to secure a separate launch site with a free line-of-sight of the sky, and instead depend on the fact that power lines will be clear of any structures or foliage.

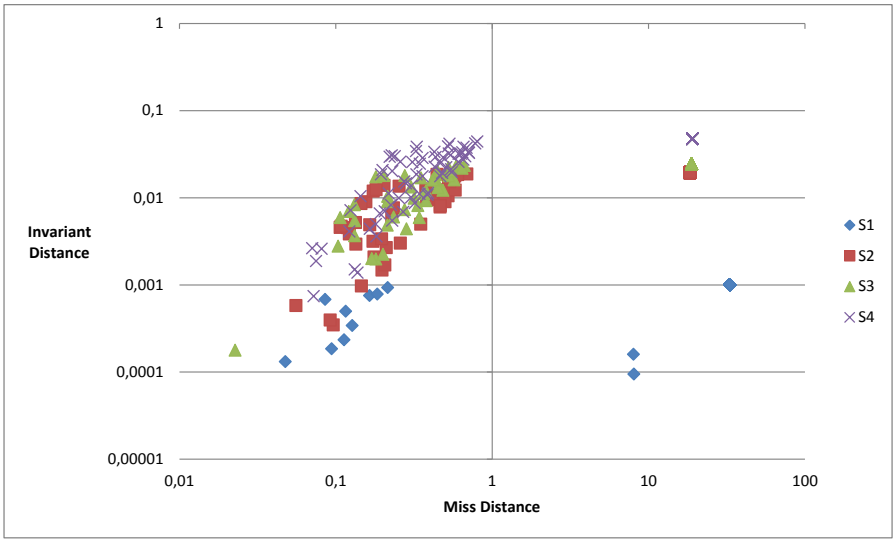


Figure 6.3: The miss distance contra the invariant distance when looking up the invariants of point $p = (7.19, 21.62)$ against S_1, S_2, S_3 and S_4. S_5 is not included as it is almost indistinguishable from S_4. Note that we only display the first 100 points for each run.

Repeat Count	Avg Elapsed (ms)	Avg Miss (m)	Miss >0.1	Miss >1	Miss >10
0	156.2293226	0.298607616	119	55	6
3	603.4885498	0.138980042	95	24	6
6	1023.382991	0.06902604	86	16	2
10	1625.373811	0.04376276	73	12	0

Table 6.4: The effect of selecting the lowest invariant distance of 1 + Repeat Count runs of the CMA-ES algortihm on the same point, using 1000 test points. We assume the drone is carrying three sensors (S_3).

Note that there's very little to gain from using two sensors (1 meter apart) in the vertical case.

6.4 Simulating Navigation

We now made all the necessary preparations to use our algorithm in a simple drone navigation simulation. As always, we will use the model in table 2.1 as our reference.

The drone will have a maximum speed of 1 m/s and be restricted to a maximum acceleration of 1 m/s^2. It will be instructed to perform a simple task—starting from a point unknown to itself, it must ascend to point (20,40), and then cruise sideways to point (0,40). The route should be as direct as possible. In our test, the starting point will be (30, 0). For this particular run, we will assume the magnetic field sensors are ideal with no loss of precision, and the conductors carry a perfect AC current with no frequency variation or amplitude variation between each other.

Sensors	Radius	Avg Elapsed (ms)	Avg Miss (m)	Miss >0.1	Miss >1	Miss >10
3	1	94.76944309	3.28E-06	0	0	0
3	2	109.4988193	2.59E-06	0	0	0
3	4	107.3388309	2.83E-06	0	0	0
3	8	116.5809785	3.74E-04	1	0	0
3	12	122.0760691	0.002862963	14	1	0
3	16	131.4016354	0.04825665	28	5	0

Table 6.5: The accuracy of the CMA-ES algorithm after incrementally increasing the radius of the search area, using 1000 test points.

Virtual Sensors	RC	Avg Elapsed (ms)	Avg Miss (m)	Miss >0.1	Miss >1	Miss >10
4V. 3 meters	0	158.5018123	0.011278116	234	7	1
4V. 3 meters	5	225.4582485	0.002968629	126	0	0
4V. 2R. 3 meters	0	199.8190648	0.011107173	218	4	2
4V. 2R. 3 meters	5	261.8589657	0.002598686	109	0	0
4H. 3 meters	0	156.4985323	0.001459634	18	6	0
4H. 3 meters	5	159.0923529	1.21E-04	4	0	0

Table 6.6: Testing the accuracy of virtual sensors over 10000 points, coupled with a repeat count (RC) of 5. The first test (4V) measures the effect of 4 virtual horizontal sensors. In the second test (4V. 2R), we have two rows 1 meter apart, with 4 virtual horizontal sensors each. The third test is a repeat of the first, except using 4 horizontal sensors (4H).

6.4.1 Proposed Algorithm

1. Ascend 9 meters and sample the magnetic field using a single sensor every 3 meters. This step may require an INS if the drone cannot guarantee a known constant ascension speed. Using these four samples, we use the virtual sensor approach as detailed in subsection 6.3.3, which yields our first estimate of the current position. If, according to the algorithm, we have moved downwards instead of upwards as expected, we know the assumed phase sequence is wrong and we have to use the opposite phase category (see 3.4.1).

2. If we now need to estimate the orientation, we use the methods described in 3.3.4—either use GPS, or compute the cross-product of two different magnetic field vectors from the same sensor.

3. We can now use this position estimate to restrict the search area for any subsequent position lookups. Using the maximum speed of the drone and the time the position estimate was calculated, we deduce the maximum extent of the search area. This area can then be extended, just in case.

4. Given a previous position and conservative search area, we use the method in subsection 6.3.2 to more accurately find the current position. Repeat as often as sensible.

Let us put this algorithm to the test. It has been implemented by the *DroneSimulation* class, and it can be started by our PlotCSV program using the commend seen in code listing 6.3.

Listing 6.3: Computing the amplitude angle and phase difference in Mathematica.

```
java -jar PlotCSV.jar -m Model.xml -modelsource d3 -x
-35:35:1920 -y 0:40:1000 -t 0:0:1 -v m -f FALSE
-drone_simulation -dr 6 -solver cmaes
-solver.iterations 4000 -solver.populationSize 1000
-solver.threshold 0.000000000000001 drone2\output.png
```

The drone position during the test is illustrated by figure 6.4, and the distance from the estimated position and the true1 position is plotted by figure 6.5. The miss distance on our estimated position over the course of the flight averages out to about 0.31 mm (SD: 6.09 mm).

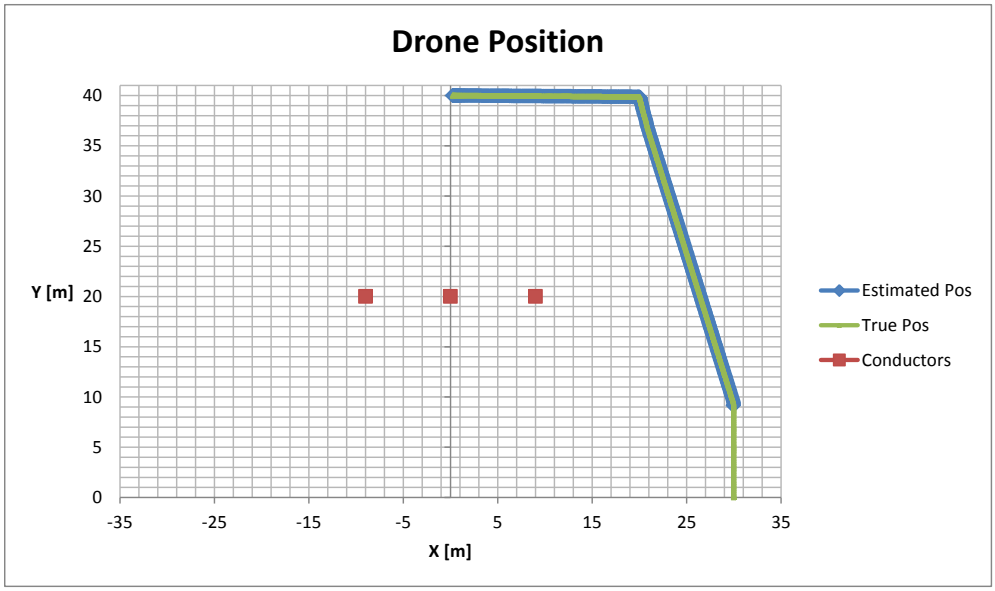


Figure 6.4: Position of the drone, and the estimated position.

6.4.2 Effect of Frequency Deviation

In subsection 3.5.1, we theorized that small deviations in the system frequency (ideally 50 Hz) might negatively impact the accuracy of our algorithm. We are now ready to put this idea to the test, simply by computing the field invariants at point p using a slightly lower or higher frequency, and then try to locate the original point p in a comparison model that is running at the ideal frequency.

If we use the S_3 configuration, along with restricting the radius to 6 meters and setting the repeat count to 5, we end up with the results in table 6.7. The frequency was altered in the model XML-file by changing the *freq* attribute to $47 * 360 = 16920$ and $53 * 360 = 19080$, respectively. Note that the frequency is typically within a range of 49.1 to 50.1, so these constitute extreme deviations from the norm. Nevertheless, the accuracy of the algorithm remained unchanged, so it is clear that random deviations in the frequency will have essentially no effect.

We can thus safely ignore the frequency deviation.

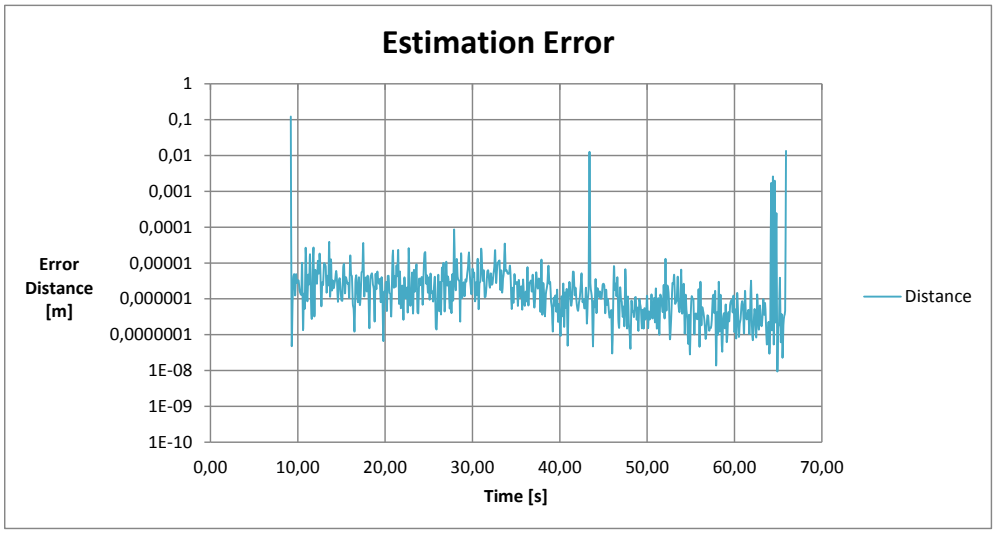


Figure 6.5: The distance between the estimated position and the true position of the drone, over time. This assumes the currents are perfectly balanced.

Frequency	Avg Elapsed (ms)	Avg Miss (m)	Miss Std (m)	Miss >0.1
50 (Ideal)	130.166	1.864E-05	0.004	0
47	109.894	3.783E-05	0.006	0
53	117.831	1.493E-05	0.004	0

Table 6.7: The effect of changing the frequency of the sample model, against a comparison model (always at 50 Hz). This is done on the same 1000 random points.

6.4.3 Effect of Current Imbalance

Continuing from subsection 3.5.2, we test the effect of using the currents from the lowest local SD and the highest local SD (see table 3.3), on the algorithm accuracy. This is done both using the configuration in S_3 and 4 vertical virtual sensors (4V), an presented in table 6.8. Sadly, it looks like the current imbalance is having a massive negative effect on the overall accuracy of our algorithm, where the average miss distance is 2.368 meters in the worst case (HCD, or the highest local SD).

To correct this, we need to include the current imbalance as a dimension in our search space, using a and b as defined in 3.23. The CMA-ES algorithm will then simply use four dimensions in the search points - the x and y coordinate of the point, along with the a and b parameters that adjust the global current imbalance. The imbalance itself is restricted by the interval in 3.25.

The result of adding these two new dimensions is displayed in table 6.9. Here, $C_(a, i)$ is a shorthand for configuration A, which specifies a test using a repeat count of 5, a restricted radius of 6 meters and sensors S_i. While in $C_{b,i}$, we use 4 vertical virtual sensors in i number of rows.

Finally, we also limit the maximum invariant distance (3.12) to 0.3, and fail the search otherwise (see the failure count in the table). That way, we avoid the very problematic spots, and instead encourage the drone to move to another location and try

Setup	Avg Elapsed	Avg Miss	Miss Std (m)	Miss >0.1	Miss >1
S_3. LCD	771.002 ms	2.903 m	0.948 m	928	689
S_3. HCD	946.791 ms	4.368 m	2.090 m	971	857
4V. LCD	932.899 ms	1.081 m	1.040 m	805	399
4V. HCD	1059.090 ms	4.319 m	2.078 m	911	667

Table 6.8: The effect of two different current imbalances LCD (Low Current Deviation) and HCD (High Current Deviation), using three sensors S_3 and four vertical sensors 4V. We use 1000 random points in each test.

again. Estimated points that miss by over 10 meters could also be filtered out by a GPS and known power line path, if the invariant distance does not do it already.

Table 6.9: If we a and b as dimensions in the search space, we get the following results. The same 1000 random points is used here as well. The number of misses that exceeded 10 meters is recorded in the *Failures* column in parenthesis.

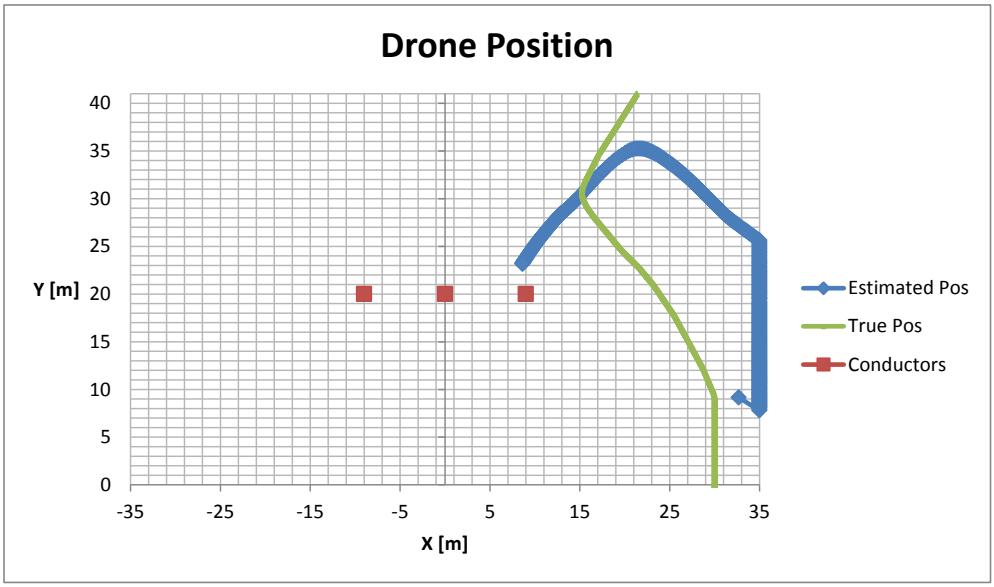
Setup	Avg Elapsed	Avg Miss:	Std:	Miss >0.1	Miss >1	Failures
$C_{a,1}$ + HCD	246,086 ms	1,781 m	1,335 m	975	755	0
$C_{a,2}$ + HCD	310,184 ms	0,018 m	0,018 m	7	0	0
$C_{a,3}$ + LCD	376.067 ms	0.018 m	0.134 m	4	3	0
$C_{a,3}$ + HCD	424.571 ms	0.008 m	0.092 m	3	2	0
$C_{b,1}$ + HCD	1355.476 ms	0.100 m	0.316 m	25	5	11 (2)
$C_{b,2}$ + HCD	1584.745 ms	0.013 m	0.114 m	22	4	14

It seems that the miss distance is adequately low the $C_{a,2}$ and $C_{a,3}$ case, provided that we have a good initial location. In contrast, the single drone sensor case, $C_{a,2}$, is practically unusable. The initial location is supplied by $C_{b,1}$ and $C_{b,2}$, which unfortunately is not quite as good — in particular, there is a large number of cases where the algorithm fails to find a point with a low enough invariant distance (0.3 or less). It also appears to be necessary to use two rows of 4 virtual sensors, to avoid misses over 10 meters that are undetectable by high invariant distances.

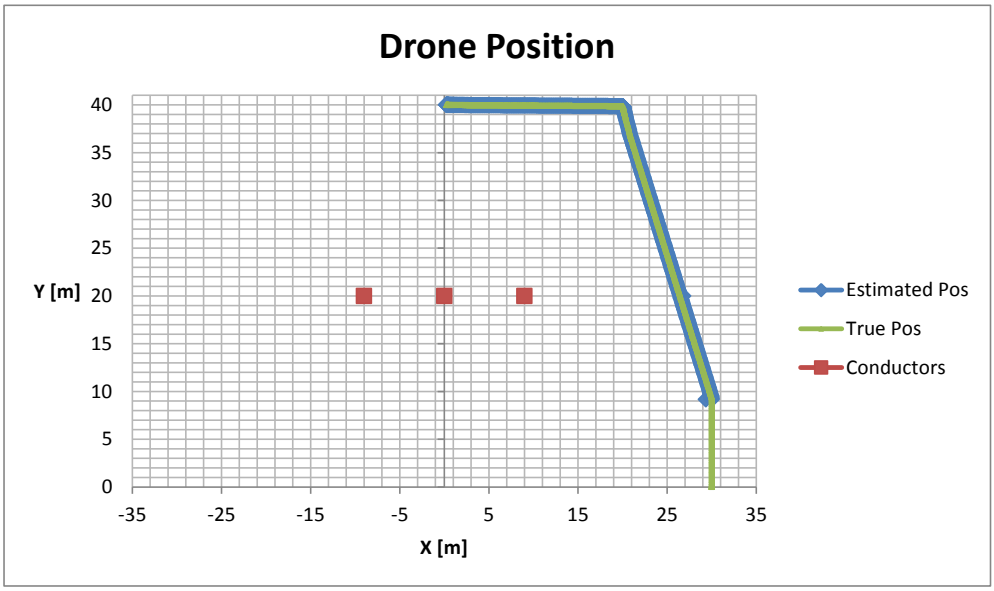
Summary To sum up, in order to account for current imbalance, we should amend our proposed algorithm in subsection 6.4.1 by adding two additional dimensions a and b, and also use at least two rows of virtual columns in the initial search phase. If that first initial step fails, we simply repeat until we get a low field invariant distance.

Finally, let us rerun the drone simulation using $C_{b,1}$ initially, and $C_{a,3}$ after we have our initial fix, all under the most difficult condition (HCD). This is displayed in figure 6.6.

The average miss distance after turning on a and b estimation, was about 1.80 mm (SD: 30.88 mm). This is just over six times less accurate than when the current is perfectly balanced (0.31 mm), yet it is still perfectly acceptable.



(a) Navigating without estimating a and b, which fails completely.



(b) Navigating while also estimating a and b.

Figure 6.6: Comparison of attempting to navigate in a field with a high current imbalance (HCD), using $a = b = 1$, or by allowing CMA-ES to estimate a and b.

Chapter 7

Conclusion

We demonstrated that it is possible to extract very accurate positional information from the magnetic field generated by a power line, using either multiple measurements over time or at least **two sensors** and a previous position. By aligning the drone with the plane of the magnetic field lines, we could follow the path of the power line at an average accuracy of **0.31 mm**.

After taking into account that the current in each wire deviates from the average current at most 10%, we nevertheless achieve the very respectable accuracy of **1.80 mm**.

The only remaining issue is the computational cost, where a single evaluation requires approximately 350 ms on a modern desktop computer.

Chapter 8

Future Work

- Investigate ways of improving the performance of the algorithm, either by switching to a faster alternative to CMA-ES or possibly by designing an equivalent Kalman filter [18]. Also ensure that performance is competitive on an embedded computer, such as the Raspberry Pi.
- Currently, most of our work is theoretical. It is necessary to eventually test the algorithm in practice, using real hardware near a power line or something equivalent. That would reveal any real-world considerations we did not consider, or hardware limitations that we failed to take into account.
- Determine if this method works on power lines consisting of multiple three-phase circuits, not just a single circuit. Can it be also be extended to ground wires?
- Our work assumes we know the exact type of power line, and the spacing between each conductor. Can these parameters be estimated with no prior knowledge, using only samples from the magnetic field?

Appendix A

Code Listings

A.1 Mathematica

Mathematica was primarily used to plot every 3D figure, though it turned out to be more than capable of computing the solution to the position problem.

The amplitude angle and phase difference (3.5 and 3.9) can be computed as follows. This assumes that each wire is positioned according to table 2.1:

Listing A.1: Computing the amplitude angle and phase difference in Mathematica.

```
fieldx[x_,y_,cX_, cY_, r_, I_,p_] := Module[{dX,dY,r2},
  dX=x-cX;dY=y-cY;r2=dX^2+dY^2;
  dY*((p*I)/(2*Pi))/r2]

fieldy[x_,y_,cX_, cY_, r_, I_,p_] := Module[{dX,dY,r2},
  dX=x-cX;dY=y-cY;r2=dX^2+dY^2;
  -dX*((p*I)/(2*Pi))/r2]

phasex[x_,y_,o_] :=Module[{p,c1,c2,c3},
  p=4.0*Pi*10^(-7);
  c1=fieldx[x,y,-9,20,0.04,12000,p];
  c2=fieldx[x,y,0,20,0.04,12000,p];
  c3=fieldx[x,y,9,20,0.04,12000,p];
  ArcTan[c1 * Sin[(0+o) Degree]+c2*Sin[(120+o) Degree]+
      c3*Sin [(240+o) Degree],
    c1 * Cos[(0+o) Degree]+c2*Cos[(120+o) Degree]+
      c3*Cos [(240+o)Degree]]]

phasey[x_,y_,o_] :=Module[{p,c1,c2,c3},
  p=4.0*Pi*10^(-7);
  c1=fieldy[x,y,-9,20,0.04,12000,p];
  c2=fieldy[x,y,0,20,0.04,12000,p];
  c3=fieldy[x,y,9,20,0.04,12000,p];
  ArcTan[c1 * Sin[(0+o) Degree]+c2*Sin[(120+o) Degree]+
      c3*Sin [(240+o) Degree],
    c1 * Cos[(0+o) Degree]+c2*Cos[(120+o) Degree]+
      c3*Cos [(240+o) Degree]]]
```

```
angleDifference[a_,b_]:=-Mod[b-a,2 Pi,-Pi]

amplitudex[x_,y_] := Module[{p,c1,c2,c3},
  p=4.0*Pi*10^(-7);
  c1=fieldx[x,y,-9,20,0.04,12000,p];
  c2=fieldx[x,y,0,20,0.04,12000,p];
  c3=fieldx[x,y,9,20,0.04,12000,p];
  Sqrt[c1^2+c2^2+c3^2-c1*c2-c1*c3-c2*c3]]

amplitudey[x_,y_] := Module[{p,c1,c2,c3},
  p=4.0*Pi*10^(-7);
  c1=fieldy[x,y,-9,20,0.04,12000,p];
  c2=fieldy[x,y,0,20,0.04,12000,p];
  c3=fieldy[x,y,9,20,0.04,12000,p];
  Sqrt[c1^2+c2^2+c3^2-c1*c2-c1*c3-c2*c3]]

AmplitudeAngle[x_, y_] := Module[{ax, ay},
  ax = amplitudex[x, y];
  ay = amplitudey[x, y];
  If[ax != 0, ArcTan[ay/ax], Sign[ay]*(Pi/2)]]
```

A.2 Java

A.2.1 Field Interfaces

This section contains unabridged versions of all the field interfaces presented in chapter 4, baring the actual implementation of every default method. Please consult the attached source code to view the interfaces in full, complete with the default implementation of every method.

Listing A.2: Every method in the **DiscreteField** interface.

```
1  import org.apache.commons.math3.geometry.**;
2
3  public interface DiscreteField {
4     double getColumn(int column);
5     double getRow(int row);
6     int getColumns();
7     int getRows();
8
9     default Vector2D getNearestPoint(double x, double y) {
10       /* Return the cell that is nearest
11          the given point. */
12    }
13    default double[] toColumnArray() {
14       /* Return current column X coordinates
15          as an array. */
16    }
17    default double[] toRowArray() {
18       /* Return the current row Y coordinates
19          as an array. */
20    }
21 }
```

Listing A.3: Every method in the **ScalarField** interface.

```
import com.google.common.collect.Range;

public interface ScalarField extends DiscreteField {
   double getValue(int column, int row);

   default void fillValues(int startColumn, int endColumn,
         int startRow, int endRow, double[] outputValues) {
      /* Use getValue() to fill the array */
   }
   default DoubleStream stream() {
      /* View this field as a stream of cell values. */
   }
   default Range<Double> getRange() {
      /* Return the minimum and maximum cell
        value expressed as a range. */
   }
   default Stream<double[]> rows() {
      /* Return a stream of all the rows of
        values in the field. */
   }
   default double[][] toDoubleArray() {
      /* Convert current scalar field to a jagged array. */
   }
   default ScalarArrayField toArrayField() {
      /* Convert the current scalar field to an
        array scalar field */
   }
}
```

Listing A.4: Every method in the **ContinuousField** interface.

```
import org.apache.commons.math3.linear.RealVector;

public interface ContinuousField {
   int getComponents();

   void addComponents(double x, double y, int startComponent,
         int endComponent, double[] destination,
         int destinationOffset);

   default void addComponents(double x, double y,
         double[] destination, int destinationOffset) {
      /* Default parameters */
   }

   default VectorField toVectorField(double[] xCoordinates,
         double[] yCoordinates) {
      /* Convert this continuous field to a discrete
        vector field. */
   }
   default double getMagnitude(double x, double y) {
      /* Retrieve the magnitude of the given cell. */
   }
   default RealVector getAngle(double x, double y) {
      /* Retrieve the polar angles of the given cell. */
   }
```

```
26     default RealVector getVector(double x, double y) {
27        /* Retrieve the value of the given cell. */
28     }
```

Listing A.5: Every method in the **VectorField** interface.

```
1  public interface VectorField extends DiscreteField {
2     int getComponents();
3
4     void fillComponents(int startColumn, int endColumn,
5           int startRow, int endRow,
6           int startComponent, int endComponent, double[] destination,
7           int destinationOffset, boolean zeroDestination);
8
9     default RealVector getVector(int column, int row) {
10       /* Retrieve the value of the given cell. */
11    }
12
13    default double getMagnitude(int column, int row) {
14       /* Retrieve the magnitude of the given cell. */
15    }
16    default RealVector getAngle(int column, int row) {
17       /* Retrieve the polar angles of the given cell. */
18    }
19    default RealVector getNearestValue(double x, double y) {
20       /* Retrieve the value of the cell nearest the given point. */
21    }
22
23    default double[] getComponentsArray(int startColumn, int
          endColumn,
24          int startRow, int endRow) {
25       /* Retrieve an array of all the components in the given
             rectangle. */
26    }
27    default double[] getComponentsArray(int startColumn, int
          endColumn,
28          int startRow, int endRow, int startComponent, int
                endComponent) {
29       /* Retrieve an array of the given components in the given
             rectangle. */
30    }
31
32    default void fillComponents(int startColumn, int endColumn, int
          startRow, int endRow,
33          double[] destination, int destinationOffset) {
34       /* Fill the destination array with the respective values of
             the components in the cell vectors. */
35    }
36
37    default ScalarField magnitudes() {
38       /* View the magnitude of each cell in this vector field. */
39    }
40    default ScalarField x() {
41       /* View the magnitude of the x component in this vector field.
             */
42    }
43    default ScalarField y() {
```

```
44      /* View the magnitude of the y component in this vector field.
           */
45   }
46   default DiscreteField angles() {
47      /* View the angle of each cell in this vector field. */
48   }
49   default ScalarField map(DoubleBinaryOperator operator) {
50      /* Create a scalar field from a two dimensional vector field.
           */
51   }
52   default ScalarField map(ScalarMapping operator) {
53      /* Create a scalar field from the X and Y values of this
           vector field. */
54   }
```

A.3 PlotCSV Commands

We also list the PlotCSV command lines needed to produce a number if figures in this paper in table A.1.

A.4 Attached Files

The following files of the source code and executable is contained within the digital PDF-version of this paper:

- Source Code: 
- Compiled Binary: 

Table A.1: Assorted Commands

Name	Command
Magnitude	-m Model.xml -x -35:35:1920 -y 0:40:1000 -z 0.001:0.3 -t 0:0.02:1000 -v m -l -gy 40:32 -gz -5:1 -ax x:6:32 -ay y:6:48 magnitude3/output-%04d.png
Maximum MF	-m Model.xml -x -35:35:1920 -y 0:40:1000 -z 0.001:0.3 -t 0:0.02:1000 -l -f False -ag max aggregate/output-%04d.png
Value of Component X	-m Model.xml -x -35:35:400 -y 0:40:400 -z -0.6:0.6 -t 0:0.02:100 -v x -gy 40:32 -gz -0.6:0.6 -ax x:6:32 -ay y:6:48 -of csv component-x2/output-%04d.csv
Value of Component Y	-m Model.xml -x -35:35:400 -y 0:40:400 -z -0.6:0.6 -t 0:0.02:100 -v y -abslog -gy 40:32 -gz -0.6:0.6 -ax x:6:32 -ay y:6:48 component-y2/output-%04d.png
Angle (HSV)	-m Model.xml -x -35:35:1920 -y 0:40:1000 -z -3.14:3.14 -t 0:0.02:1000 -s hsv -v a -ax x:6:32 -ay y:6:48 hsv/output-%04d.png
Sinusoid Field (X)	-m Model.xml -s hsv -modelsource d1 -x -20:20:400 -y 0:40:400 -t -0:1:1 -v x -ax x:6:32 -ay y:6:48 drone2/output-%04d.png
Drone PDF	-m Model.xml -m2 ModelHighFreqMaxCurrentDeviation.xml -modelsource d2 -x -35:35:1920 -y 0:40:1000 -t 0:0:1 -v m -drone_pdf 1000 -dc 5 -dr 6 -duc -solver cmaes -solver.iterations 4000 -solver.populationSize 1000 -solver.threshold 0.000000000000001 drone2/output-%04d.png
Drone LocTest	-m Model.xml -m2 ModelHighFreqMinCurrentDeviation.xml -modelsource d3 -x -35:35:1920 -y 0:40:1000 -t 0:0:1 -v m -dr 6 -duc -drone 26.3260053811 28.4330865133 -solver cmaes -solver.iterations 4000 -solver.populationSize 2000 -solver.threshold 0.000000000000001 drone2/output-%04d.png
Drone Simulation	-m Model.xml -m2 ModelHighFreqMaxCurrentDeviation.xml -modelsource d3 -duc -x -35:35:1920 -y 0:40:1000 -t 0:0:1 -v m -f FALSE -drone_simulation -dr 6 -solver cmaes -solver.iterations 4000 -solver.populationSize 1000 -solver.threshold 0.000000000000001 drone2/output-%04d.png
FFT Drone Field	-m Model.xml -x -35:35:1920 -y 0:40:1000 -t 0:0.02:8 -f false -ag drone -v m drone2/output-%04d.png
Phase Difference	-m Model.xml -x -20:20:400 -y 0:40:400 -z -3.14:3.14 -t 0:0.02:8 -f false -ag xyf -v anglediff -ax x:6:32 -ay y:6:48 -s hsv xyphase5/output-%04d.png
Amplitude Angle	-m Model.xml -x -20:20:400 -y 0:40:400 -z -3.14:3.14 -t 0:0.02:8 -ax x:6:32 -ay y:6:48 -f false -ag ampf -v a -s hsv xyamplitude4/output-%04d.png

Glossary

BOBYQA Bound Optimization BY Quadratic Approximation. 46, 60, 61

CAS Computer Algebra System. 27, 38

CLI Command Line Interface. 6, 41, 47

CMA-ES Covariance Matrix Adaptation Evolution Strategy. 1, 5, 28, 40, 42, 56, 60–63, 66, 67, 69, 71, 73

CSV Comma-Separated Values. 41

FFT Fast-Fourier Transform. 1, 22, 24, 40, 49, 55, 60

IDE Integrated Development Environment. 40

IMU Inertial Measurement Unit. 30, 31

INS Inertial Navigation System. 1, 39, 64, 67

OOP Object Oriented Programming. 45

TSV Tab-Separated values. 41, 43

Appendix B

Bibliography

[1] Estimation of system frequency objectives. http://www.nptel.ac.in/courses/108101039/download/Lecture-37.pdf. Accessed: 2015-06-08.

[2] Gps accuracy. http://web.archive.org/web/20150420125257/http://www.gps.gov/systems/gps/performance/accuracy/. Accessed: 2015-04-28.

[3] Indooratlas faq. https://indooratlas.uservoice.com/knowledgebase/articles/242897-what-is-the-positioning-accuracy-of-indooratlas. Accessed: 2015-06-11.

[4] Rapport fra systemansvarlig - om kraftsystemet i norge 2013. http://www.statnett.no/Global/Dokumenter/Kraftsystemet/Rapport%20fra%20Systemansvarlig%20om%20Kraftsyst.pdf. Accessed: 2015-06-08.

[5] Lisa Lorentzen Arne Hole and Tom Lindstrøm. *Kalkulus*. Universitetsforlaget, 2003.

[6] E. Oran Brigham. *The Fast Fourier Transform And Its Applications*. Prentice Hall, 1988.

[7] P. Debenest, M. Guarnieri, K. Takita, E.F. Fukushima, S. Hirose, K. Tamura, A. Kimura, H. Kubokawa, N. Iwama, and F. Shiga. Expliner - robot for inspection of transmission lines. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pages 3978–3984, May 2008.

[8] T. og B. Halvorsen Ericson. Kortsiktige sving- ninger i strømforbruket i alminnelig forsyning. forbruks- kurver basert på timesmålte data fra skagerak nett. Statistisk sentralbyrå, 2008.

[9] Paul Fahlstrom and Thomas Gleason. *Introduction to UAV Systems*. John Wiley & Sons, 2012.

[10] Raspberry Pi Foundation. Raspberry pi - teach, learn, and make with raspberry pi. https://www.raspberrypi.org/. Last visited on 2015-06-13.

[11] The Apache Software Foundation. Maven - welcome to apache maven. https://maven.apache.org/. Last visited on 2015-06-13.

[12] Ulf Kaintzyk Joao Felix Nolasco Friedrich Kiessling, Peter Nefzger. *Overhead Power Lines: Planning, Design, Construction*. Springer Science & Business Media, 2003.

[13] Nikolaus Hansen. The cma evolution strategy: A tutorial, 2005.

[14] Charlie Hunt and Binu John. *Java Performance*. Addison-Wesley, 2011.

[15] J. A. John and N. R. Draper. An alternative family of transformations. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 29(2):pp. 190–197, 1980.

[16] Lawrence S Lerner. *Physics: for Scientists and Engineers*. Jones and Bartlett Publishers, Inc, 1997.

[17] M. Büchel C. Dold M. Bütikofer M. feuerstein W. Fischer C. Bermes R. Siegwart M. Bühringer, J. Berchtold. Inspection of high-voltage power lines - a new approach. 2009.

[18] Joseph Moore and R. Tedrake. Magnetic localization for perching uavs on powerlines. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 2700–2707, Sept 2011.

[19] Sebastian Nanz and Carlo A. Furia. A comparative study of programming languages in rosetta code. *CoRR*, abs/1409.0252, 2014.

[20] R.G. Olsen, D. Deno, R. Conti, M. Frazier, J.R. Stewart, R. Wong, and R.M. Zavadil. Magnetic fields from electric power lines: theory and comparison to measurements. *Power Delivery, IEEE Transactions on*, 3(4):2127–2136, Oct 1988.

[21] Sergio L. Netto Paulo S. R. Diniz, Eduardo A. B. da Silva. *Digital Signal Processing: System Analysis and Design*. 2010.

[22] M. J. D. Powell. The bobyqa algorithm for bound constrained optimization without derivatives. *DAMTP 2009/NA06*, 2009.

[23] Sentewolf. Concept of directional optimization in cma-es algorithm. `http://en.wikipedia.org/wiki/File:Concept_of_directional_optimization_in_CMA-ES_algorithm.png`. Last visited on 2015-05-27.

[24] Kristian S. Stangeland. The direction of the magnetic field. `https://www.youtube.com/watch?v=ZSAhM2UO_f4`.

[25] Kristian S. Stangeland. Log-plot of the magnitude of the magnetic field. `https://www.youtube.com/watch?v=bsKawx2DhpU`.

[26] Gilbert Strang. *Introduction to Linear Algebra*. Wellesley-Cambridge Press; 4 edition, 2009.

[27] J.W. Waite, T. Gudmundsson, and D. Gargov. Uav power line position and load parameter estimation, December 29 2011. WO Patent App. PCT/US2011/041,112.

[28] Eric W. Weisstein. Euler angles. From MathWorld—A Wolfram Web Resource. `http://mathworld.wolfram.com/EulerAngles.html`. Last visited on 2015-06-04.

[29] Eric W. Weisstein. Harmonic addition theorem. From MathWorld—A Wolfram Web Resource. `http://mathworld.wolfram.com/HarmonicAdditionTheorem.html`. Last visited on 2015-05-06.