UAS Position Estimation in GPS Degraded/Denied Skies via WiFi and Cell Network Data (WiCeNav)

Matthew Joseph Davis

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Aeronautics & Astronautics

University of Washington

2020

Committee:

Juris Vagners

Christopher Lum

Program Authorized to Offer Degree:
Aeronautics & Astronautics

University of Washington

Abstract

UAS Position Estimation in GPS Degraded/Denied Skies via WiFi and Cell Network Data (WiCeNav)

Matthew Joseph Davis

Chair of the Supervisory Committee:
Professor Emeritus Juris Vagners
William E. Boeing Department of Aeronautics and Astronautics

Smart phones tell us where we are. How do they accomplish this task? Moreover, how do they accomplish this task when a GPS signal is not present? What specific functionality exists in cellular networks that allows this? Once identified, can that functionality be used in a small payload for UAS? Are there commercial solutions that already do this? Why or why not? This thesis aims to answer these questions by delving into network specifications, investigating a variety of commercial off the shelf solutions as well as several custom ones, and will ultimately detail a mixed signal solution that provides a GPS independent position estimate. The proposed solution uses readily available hardware running a signal agnostic software package.

TABLE OF CONTENTS

Page

List of Figures iv

Glossary vii

Chapter 1: Introduction 1

Chapter 2: Mathematical Review 3
2.1 Linear Least Squares 3
2.1.1 Weighted Linear Least Squares 5
2.2 Non-Linear Least Squares (NLLS) 5
2.3 Discrete-Time Linear Kalman-Bucy Filter 7
2.3.1 Recursive Nature 8
2.3.2 Changing State 9
2.3.3 Table of Equations 10
2.4 Multilateration 11
2.4.1 Non-Linear Least Squares (NLLS) Solution 12
2.4.2 Analytic Solution 13

Chapter 3: 3GPP 15
3.1 Overview 15
3.2 Specification vs Implementation 17
3.2.1 Method Availability Roadblock 18
3.2.2 Hardware Roadblock 20
3.2.3 Communication Roadblock 20

Chapter 4: Prior Work 22
4.1 Prior AFSL Work 22

4.2 Zak(Zaher) M. Kassas . . . 23
4.3 Other Contributors . . . 24

Chapter 5: Payload Hardware . . . 25
5.1 Payload Computer . . . 26
5.2 Cellular Modem . . . 28

Chapter 6: Payload Software . . . 30
6.1 Overview . . . 30
6.2 Software Modules . . . 31
6.3 Important Libraries . . . 33
6.3.1 libserial . . . 34
6.3.2 Zero MQ . . . 34
6.3.3 nmealib . . . 34
6.4 Build Environment . . . 35

Chapter 7: Tested Solutions . . . 36
7.1 Introduction . . . 36
7.2 Solution 1: TelitHE910D, ATMONI . . . 39
7.2.1 Overview . . . 39
7.2.2 Available Data . . . 42
7.2.3 Cell Tower Locations . . . 44
7.2.4 Range Estimation . . . 46
7.2.5 About Data Reduction . . . 48
7.2.6 Sanity Check . . . 48
7.2.7 Flight Data . . . 51
7.2.8 Ground Data . . . 55
7.2.9 Method Conclusions . . . 62
7.3 Solution 2: FONA, ATCIPGSMLOC . . . 62
7.3.1 Overview . . . 62
7.3.2 Ground Data . . . 64
7.3.3 Flight Data . . . 68
7.3.4 Method Conclusions . . . 72
7.4 Solution 3: WiCe-Nav . . . 73

7.4.1 Overview . . . 73
7.4.2 Software Time Note . . . 76
7.4.3 Ground Data . . . 78
7.4.4 Method Conclusions . . . 83

Chapter 8: Future Work . . . 84
8.1 Overview . . . 84
8.2 Tested Solutions . . . 84
8.2.1 ATMONI . . . 84
8.2.2 FONA . . . 85
8.2.3 WiCe . . . 85
8.3 Plug and Play Solution . . . 86
8.3.1 NMEA Publisher . . . 87

Chapter 9: Conclusions . . . 89
9.1 Overview . . . 89
9.2 Data Availability . . . 89
9.3 Current and Past Research . . . 90
9.4 Current COTS Hardware . . . 90
9.5 Tested Solutions . . . 91

Bibliography . . . 93

Appendix A: Software Module I/O . . . 97
A.1 Overview . . . 97
A.2 Executables . . . 97
A.2.1 **KillSwitch** . . . 97
A.2.2 **CellModemInteractor** . . . 98
A.2.3 **GeoSol** . . . 102
A.2.4 **GPS Receiver** . . . 105
A.2.5 **Ozz** . . . 107
A.2.6 **Logger** . . . 109
A.2.7 **WAPS** . . . 111

LIST OF FIGURES

Figure Number Page

2.1 Non-Linear Least Squares Algorithm 6
2.2 Discrete-Time Linear Kalman-Bucy Filter 10
3.1 Observed Time Difference Of Arrival 17
5.1 WiCe-Nav Hardware and Signal Flow 25
6.1 WiCe-Nav Software Data Flow 30
6.2 Software Module Structure 32
6.3 Module Start 33
7.1 North Seattle Ground Test Route 36
7.2 South Seattle Ground Test Route 37
7.3 Flight Test Route 37
7.4 Flight Test Install 38
7.5 WiCe, Ground Test Install 39
7.6 ATMONI Solution Hardware 40
7.7 ATMONI Solution Software 41
7.8 Tower with 120° cells 42
7.9 Serving/Neighboring Cells 43
7.10 Sample Open CellID Data 44
7.11 Sample Updated Map Data 45
7.12 Sample AT#MONI Data 46
7.13 Example Sensor Model 47
7.14 R = 0, NLLS Position Estimation 49
7.15 R = 10, NLLS Position Estimation 50
7.16 R = 10, 3D Analytic Position Estimation 50
7.17 R = 10, Analytic 2D Position Estimation, Flight 51
7.18 Carnation Flight Estimate Data 52

7.19 Carnation Flight Cells Read 52
7.20 Flight Cell 1 Data 53
7.21 Flight Cell 2 Data 53
7.22 Flight Cell 3 Data 54
7.23 Flight Cell 4 Data 54
7.24 Flight Cell 5 Data 55
7.25 R = 10, Analytic 2D Position Estimation, Ground 56
7.26 Analytic 2D Position Estimation, Ground 56
7.27 2D Deltas, Ground 57
7.28 North Seattle Ground Cells Read 57
7.29 North Seattle Ground Range MAE 58
7.30 Ground Cell 6 Data 59
7.31 Ground Cell 12 Data 59
7.32 Ground Cell 50 Data 60
7.33 Ground Cell 2 Data 60
7.34 Ground Cell 13 Data 61
7.35 Ground Cell 31 Data 61
7.36 FONA Solution Hardware 63
7.37 FONA Solution Software 64
7.38 FONA Initial Results, Ground 65
7.39 FONA Deltas, Ground 65
7.40 Interpolated GPS vs Fona, Ground 66
7.41 Estimate with Cell Locations and Serving Radii, Ground, 20m Filter 67
7.42 Estimate with Cell Locations and Serving Radii, Ground, 75m Filter 67
7.43 Estimate with Cell Locations and Serving Radii, Ground, 200m Filter 68
7.44 FONA Initial Results, Flight 69
7.45 FONA Flight Deltas, Flight 69
7.46 Interpolated GPS vs Fona, Flight 70
7.47 Estimate with Cell Locations and Serving Radii, Flight, 250m Filter 71
7.48 Estimate with Cell Locations and Serving Radii, Flight, 500m Filter 71
7.49 Estimate with Cell Locations and Serving Radii, Flight, 2000m Filter 72
7.50 Geolocation JSON String Format 73
7.51 WiCe-Nav Hardware 74

7.52 WiCe-Nav Software 75
7.53 WiCe, Telit Tick and Publish Time 77
7.54 WiCe Estimate 78
7.55 WiCe, Position Estimate vs Interpolated Truth 79
7.56 WiFi Scraper Publish Time 80
7.57 Geo Sol Publish Time 80
7.58 Position Estimate Mean Delta vs Time Offset 81
7.59 WiCe, Position Estimate vs 11.25s Shifted Interpolated Truth 82

8.1 Abstracted Software Data Flow 86

GLOSSARY

3GPP: Third Generation Partnership Project

AFSL: Autonomous Flight Systems Laboratory

A-GNSS: Assisted Global Navigation Satellite System

API: Application Programming Interface

ARIB: The Association of Radio Industries and Bussinesses, Japan

ARM: Advanced RISC Machines

ASCII: American Standard Code for Information Interchange

ATIS: The Alliance for Telecommunications Industry Solutions, USA

BVLOS: Beyond Visual Line of Sight

C2: Command and Control

CCSA: China Communications Standards Association

COTS: Commercial Off The Shelf

DOP: Dilution of Precision

ETSI: The European Telecommunications Standards Institute

E-UTRA: Evolved Universal Terrestrial Radio Access

GPS: Global Positioning System

GSM: Global System for Mobile Communications

JSON: JavaScript Object Notation

LCS: Location Services

LGA: Land Grid Array

LPP: LTE Position Protocol

MAE: Mean Absolute Error

MSL: Mean Sea Level

NB-IOT: Narrow Band Internet of Things

NED: North East Down

NMEA: National Marine Electronics Association

ODTOA: Observed Time Difference Of Arrival

OMA: Open Mobile Alliance

OS: Operating System

PCIE: Peripheral Component Interconnect express

QMI: Qualcomm MSM Interface

RISC: Reduced Instruction Set Computer

RSSI: Received Signal Strength Indicator

RSTD: Reference Signal Time Differences

TSDSI: Telecommunications Standards Development Society, India

TTA: Telecommuncations Technology Association, Korea

TTC: Telecommunications Technology Committee, Japan

TTL: Transistor-Transistor Logic

UE: User Equipment

UTDOA: Uplink Time Difference Of Arrival

WLAN: Wireless Local Area Network

Chapter 1

INTRODUCTION

The most common way for a UAV to determine its location is to use a GPS receiver and GNSS signals. Generally if the goal is to fly from point A to point B, the autopilot needs a constant stream of GPS truth data to ensure it tracks along its planned route. This reliance on constant GPS data bounds the possible operational areas. Areas with weak signal, or areas subject to GPS jamming or spoofing become areas outside of those bounds. This project aimed to provide an alternate estimate using cellular networks to extend the bounds of the operational areas. The focus was to aim to use commercial off the shelf solutions and not to reinvent the wheel, especially when doing so would replace the wheel with a hexagon.

Early on an additional design criterion was chosen to restrict the solution to a plug and play payload so as to not have to augment autopilot firmware. This makes the solution more malleable and realizable as an actual product. Specifically this means that the output of the payload would be what a GPS receiver outputs and the input could be anything that can create a GPS estimate. This project's industry partner is T-Mobile so the input source has always been cellular networks but the software developed is actually signal agnostic. This feature will be discussed more later but is a relevant detail to point out, especially when it comes to future work.

As was stated earlier, the main focus since the inception of this project has been to use existing solutions to create something new. As anyone with a smart phone knows, if you have service, then you most likely have at least an approximate position estimate. As anyone who has watched a true crime documentary or drama knows, references to cell phone tracking are fairly abundant. Those two points together led this project's designers

to wonder if we could leverage any existing technology within cellular networks to gain a GPS independent position estimate. A surprising detail that emerged in the initial research into this problem space revealed that almost all the prior work in this area has been outside of the network looking in, i.e. using network specifications and low level signal structure to back out position estimates. One possible reason why as will be shown later is that simply accessing the tracking technology is not quite as straight forward and open as one may like. This doesn't necessarily mean it is impossible, only that it is firewalled for good reason.

The next few chapters are dedicated to developing the baseline mathematical and technical concepts needed to describe the solutions tested without having to circle back over and over. The relevant mathematics will be covered first, then some relevant applications, followed by a section covering 3GPP. Without prematurely going into too much detail 3GPP is a standards writing organization that defines cellular (2G/3G/4G/5G) network capabilities among other things [1]. Following that prior Autonomous Flight Systems Laboratory (AFSL) and other work in this problem space will be covered. The final section in the pre-solution portion of this document will cover the software architecture developed to service this and other solutions.

Once all the background work is done three tested solutions will be covered, the third being WiCeNav. The first will be a method that uses a cellular modem, a sensor model, and multi-lateration. The second uses a different cellular modem that has a built-in function that is documented as giving a GPS estimate. While this does prove to be true there are serious limitations that will be detailed. The final solution, WiCeNav, uses 2 cellular modems and a WiFi scraper to generate a GPS estimate. The first modem is to gather cell network data and the second is to communicate with Google's geolocation API(via the internet over the cell network) that consumes the cell and WiFi data to produce the estimate.

Following the solution details, future work to integrate this solution with aircraft will be covered. Finally conclusions will be drawn. An appendix covers interaction with the software. That appendix serves as the starting point for work extending this project.

Chapter 2
MATHEMATICAL REVIEW

This section covers the mathematics that will be referenced in later sections. It is assumed that the reader is familiar with theses concepts and this section is a review. This is essentially a high level overview of sections of chapters 1 & 3 in [2], so for a more complete overview see that reference.

2.1 Linear Least Squares

Note: This is an overview of [2, Chapter 1, Section 2].

Assume you have a set of measurements:

$$\tilde{y} = \{\tilde{y}_1, t_1; \tilde{y}_2, t_2; ...\tilde{y}_m, t_m\} \tag{2.1}$$

and that your proposed mathematical model, i.e. your estimated output is:

$$\hat{y}(t) = \sum_{i=1}^{n} x_i h_i(t), \quad m \geq n \tag{2.2}$$

where:

$$h_i \ \in \ \{h_1(t), h_2(t), ..., h_n(t)\} \tag{2.3}$$

are a set of independent specified basis functions. Also assume that your measurements have some error in them such that:

$$\tilde{y}(t_j) = \sum_{i=1}^{n} x_i h_i(t_j) + \nu(j), \quad j = 1, 2, ..., m \tag{2.4}$$

Where ν_j is the measurement error. Relating $\tilde{y}$ to $\hat{y}$:

$$\tilde{y} = \sum_{i=1}^{n} \hat{x}_i h_i(t_j) + e_j, \quad j = 1, 2, ..., m \tag{2.5}$$

Where e_j is the residual error:

$$e_j = \tilde{y}_j - \hat{y}_j \tag{2.6}$$

Writing everything is matrix form:

$$\tilde{y} = H\hat{x} + e \tag{2.7}$$

Where

$$\tilde{y} = [\tilde{y}_1, \tilde{y}_2, ..., \tilde{y}_m]^T = \textit{measured y-values}$$

$$H = \begin{bmatrix} h_1(t_1) & h_2(t_1) & \dots & h_n(t_1) \\ \vdots & \vdots & & \vdots \\ h_1(t_m) & h_2(t_m) & & h_n(t_m) \end{bmatrix}$$

$$\hat{x} = [\hat{x}_1, \hat{x}_2, ..., \hat{x}_n]^T = \textit{estimated x-values}$$

$$e = [e_1, e_2, ..., e_m]^T \ \textit{residual errors}$$

The point of all of this is to provide the optimal estimate of $\hat{x}$, specifically one that minimizes the sum square residual error e_j, given by:

$$J = \frac{1}{2} e^T e \tag{2.8}$$

In order to do this the definiton of e from eq. (1.5) is used and derivatives are taken. Setting the first derivative to 0 and checking the sign of the second derivative (these are matrix quantities, known as the Jacobian and the Hessian respectively) are the necessary and sufficient conditons that the estimate is indeed a minimum. See [2] for the full procedure. The resulting equation is:

$$\hat{x} = (H^T H)^{-1} H^T \tilde{y} \tag{2.9}$$

2.1.1 Weighted Linear Least Squares

The previous section assumes equal weight on all of the measurements. If you wanted to give significance or weight some measurements over others (which we do, and is a key portion of the later filtering sections), the new function to minimize is given as:

$$J = \frac{1}{2} e^T W e \tag{2.10}$$

Performing the same derivatives and applying the same conditions as above, the resulting estimate equation is almost identical:

$$\hat{x} = (H^T W H)^{-1} H^T W \tilde{y} \tag{2.11}$$

2.2 Non-Linear Least Squares (NLLS)

Note: This is an overview of [2, Chapter 1, Section 4] .

In the previous section it was assumed that the measurements were a linear function of the state variables. In practical applications this is not necessarily the case. In order to take that into account the measurement model is changed to:

$$\tilde{y} = f(x) + \nu \tag{2.12}$$

Where $f(x)$ is a column vector of independent functions of x. The functions are assumed to be single valued, continuous, and at least once differentiable.

Applying the same relations as earlier, eq. (1.6) can now be represented as:

$$\tilde{y} = f(\hat{x}) + e \tag{2.13}$$

We are still aiming to minimize eq. (1.9) but it is most likely not possible using the same mechanisms as before. In this case we use an algorithm to iterate on $\hat{x}$ until the change in eq. (1.9) is below a user defined tolerance i.e. where the derivative of $J \approx 0$. See below for a diagram of an algorithm(reprinted from [2, p.28]):

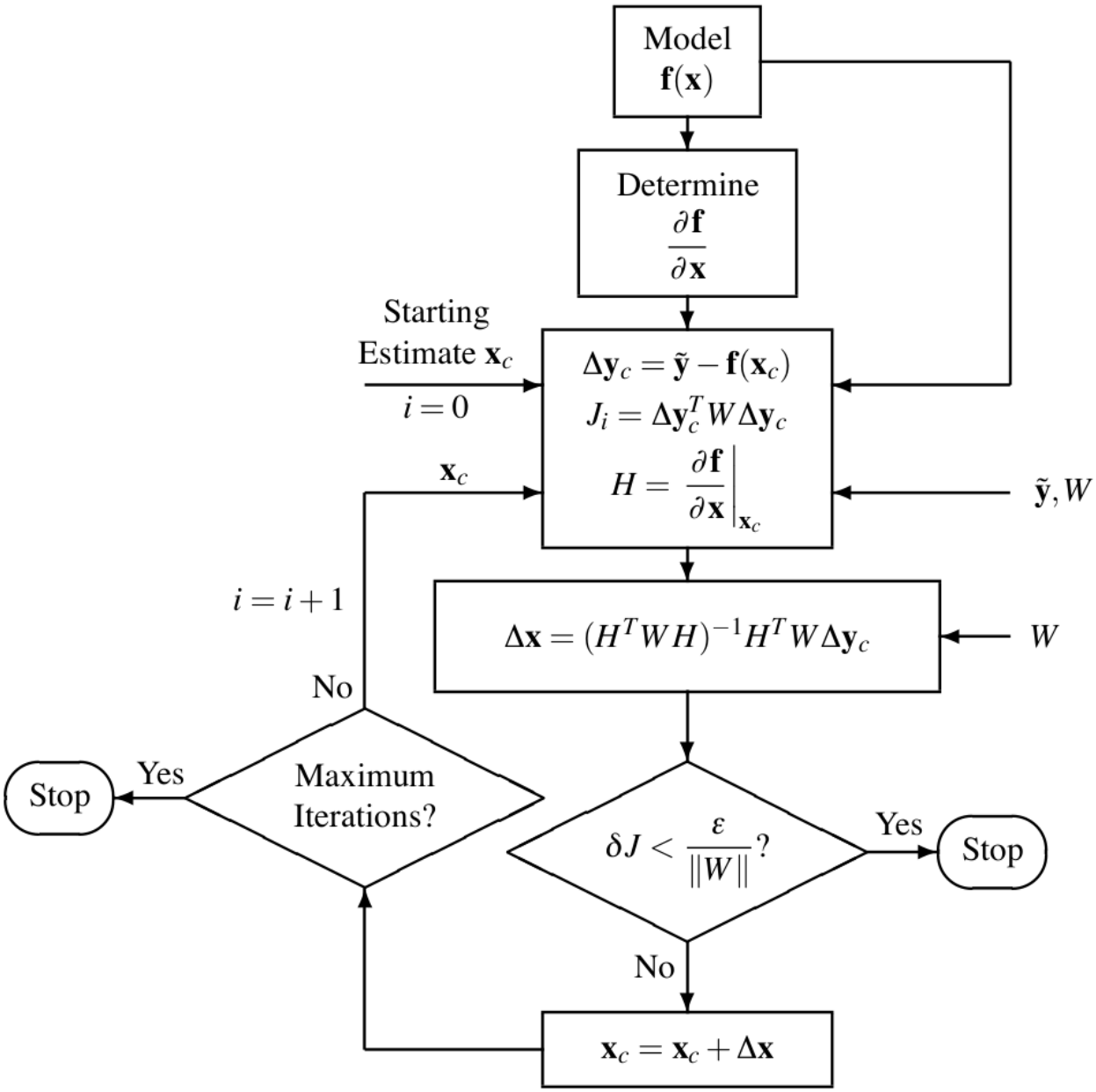


Figure 2.1: Non-Linear Least Squares Algorithm

Notes:

- This approach is defined using a first order Taylor series expansion about the current $\hat{x}_c$ guess. The assumption here is that:

- $f(\hat{x}) \approx f(x_c) + H\Delta x \leftarrow$ first order expansion
- $H = \left.\frac{\delta f}{\delta x}\right|_{x_c}$

- This approach assumes the initial guess of $\hat{x}_c$ is *sufficiently* close

2.3 Discrete-Time Linear Kalman-Bucy Filter

Note: This section is on overview [2, (Chapter 1, Section 3) and (Chapter 3, Sections 1 - 3)]

There is no quick and succinct way to cover the entire theory of the Kalman-Bucy filter. An attempt will be made here to hit the high points though. If you are not familiar with Kalman-Bucy filters it is highly recommended you read the sections highlighted in the note above.

Additionally it should be noted that the probability concepts used in Kalman-Bucy filters are not covered here *but* the assumption is that all noise is gaussian. It doesn't *have* to be, the version covered here is though. Also the weighting matrix from earlier examples is replaced by the inverse of the covariance of the measurement noise i.e. $W = R^{-1}$. The reader is pointed to Gauss-Markov theorem for a complete derivation but suffice it to say that to minimize the variance of an estimation the weighting matrix used should be as defined above.

A discrete time linear Kalman-Bucy filter is a set of equations that are recursive in nature, in that they use previous data to generate new data. The state they are estimating is allowed to change, which is a departure from the other forms of estimation described. In order to provide accurate results feedback is used. The final form of the Kalman-Bucy filter can be represented as a table of equations that provide the optimal pole locations for the estimator error dynamics. That is what a Kalman-Bucy filter ***actually*** does, it tunes the error dynamics such that the error tends to 0 and it does it quickly (think about tuning 2nd order systems for percent overshoot, response time, and the like). The following describes each of these characteristics.

2.3.1 Recursive Nature

The recursive nature of the Kalman-Bucy filter is easy to describe via linear sequential estimation where linear batch estimation (least squares) is extended to multiple 'batches'. If you estimated $\hat{x}$ every time a new batch came in, using all of the data, then you would have to invert a larger and larger matrix. If you only used the newest batch to estimate $\hat{x}$ then you would be discarding earlier data that would be useful in providing the optimal estimate. Linear sequential estimation and Kalman-Bucy filters use a couple of tricks that allow you to 'cache' earlier data so you don't have to invert massive matrices and you are still using all the available data. Assuming that the merged batches of measurements is defined as:

$$\tilde{y} = Hx + \nu \tag{2.14}$$

Where

$$\tilde{y} = \begin{bmatrix} \tilde{y}_1 \\ \dots \\ \tilde{y}_2 \end{bmatrix}, \; H = \begin{bmatrix} H_1 \\ \dots \\ H_2 \end{bmatrix}, \; \nu = \begin{bmatrix} \nu_1 \\ \dots \\ \nu_2 \end{bmatrix} \tag{2.15}$$

The critical assumption is that the associated weighting matrix is block diagonal, such that:

$$W = \begin{bmatrix} W_1 & \vdots & 0 \\ \dots & & \dots \\ 0 & \vdots & W_2 \end{bmatrix} \tag{2.16}$$

Under that assumption the equation:

$$\hat{x}_2 = (H^T W H)^{-1} H^T W \tilde{y} \tag{2.17}$$

can be expanded to:

$$\hat{x}_2 = [H_1^T W_1 H_1 + H_2^T W_2 H_2]^{-1} (H_1^T W_1 \tilde{y}_1 + H_2^T W_2 \tilde{y}_2) \tag{2.18}$$

From here two matrix variables P and K are defined that are functions of W and H values. K is known as the *Kalman Gain Matrix* and P is the *Information Matrix*. Using these two variables you can 'cache' previously calculated values and update your estimate.

2.3.2 Changing State

A key difference in the Kalman-Bucy filter vs the previous forms of estimation is that the estimate is allowed to changed. In order to provide accurate results, feedback is used. If the truth model used was:

$$\dot{x}(t) = Fx + Bu \tag{2.19}$$

$$y = Hx \tag{2.20}$$

The state equation with linear feedback looks like:

$$\dot{\hat{x}}(t) = F\hat{x} + Bu + K[\tilde{y} - H\hat{x}] \tag{2.21}$$

$$\hat{y} = H\hat{x} \tag{2.22}$$

and the measurement model is:

$$\tilde{y} = Hx + v(t) \tag{2.23}$$

Where $v(t)$ is a zero mean gaussian noise process.

In order to analyze the performance of the estimator, we want to see how the error in the estimator grows over time. The error is defined as:

$$\tilde{x} \equiv \hat{x} - x \tag{2.24}$$

and its time derivative is:

$$\dot{\tilde{x}} = (F - KH)\tilde{x} + Kv \tag{2.25}$$

Here eq. (2.25) shows the filter dynamics. In order for the error magnitude to decay to zero, $F - KH$ must be stable (real portion of eigenvalues < 0). So the work after this is to place the poles of the estimator dynamics such that the eigenvalues are fast while still filtering out noise.

2.3.3 Table of Equations

The following figure is a reprinted table from [2, p. 148] and is final result of the Kalman-Bucy filter approach to placing the poles of the estimator dynamics.

Model	$\mathbf{x}_{k+1} = \Phi_k \mathbf{x}_k + \Gamma_k \mathbf{u}_k + \Upsilon_k \mathbf{w}_k, \quad \mathbf{w}_k \sim N(\mathbf{0}, Q_k)$ $\tilde{\mathbf{y}}_k = H_k \mathbf{x}_k + \mathbf{v}_k, \quad \mathbf{v}_k \sim N(\mathbf{0}, R_k)$
Initialize	$\hat{\mathbf{x}}(t_0) = \hat{\mathbf{x}}_0$ $P_0 = E\left\{\tilde{\mathbf{x}}(t_0)\,\tilde{\mathbf{x}}^T(t_0)\right\}$
Gain	$K_k = P_k^- H_k^T \left[H_k P_k^- H_k^T + R_k\right]^{-1}$
Update	$\hat{\mathbf{x}}_k^+ = \hat{\mathbf{x}}_k^- + K_k[\tilde{\mathbf{y}}_k - H_k \hat{\mathbf{x}}_k^-]$ $P_k^+ = [I - K_k H_k] P_k^-$
Propagation	$\hat{\mathbf{x}}_{k+1}^- = \Phi_k \hat{\mathbf{x}}_k^+ + \Gamma_k \mathbf{u}_k$ $P_{k+1}^- = \Phi_k P_k^+ \Phi_k^T + \Upsilon_k Q_k \Upsilon_k^T$

Figure 2.2: Discrete-Time Linear Kalman-Bucy Filter

Where Φ_k = the state transition matrix, and

$$cov\{v\} \equiv R = E\{vv^T\} \tag{2.26}$$

$$cov\{w\} \equiv Q = E\{ww^T\} \tag{2.27}$$

Where $E\{\cdot\}$ is the expected value.

Note: assume measurement errors v and a priori errors w are uncorrelated $E\{wv^t\} = 0$

2.4 Multilateration

This section will cover multi-lateration. This technique is used in several applications relevant to this project. The first tested solution uses it to attempt to gain a position estimate. GPS receivers also use it. The basic idea is to use multiple ranges from known objects to back out the unknown location of an object. The main difference in different applications is how those ranges are gathered. They all have at least three unknowns to start with though, namely your x,y, and z coordinates in whatever frame you are using. GPS also solves for a time bias τ that exists on the GPS receiver, but that is outside of the scope of this discussion.

As stated above, what we are looking to do here is to take at least three range estimates and back out a location. There were two solutions used for this problem. The first is an application of non-linear least squares. The second is an analytic solution for this specific problem. Both solutions start off the same.

Assume you have two vectors, one that goes from the origin to a cell tower:

$$\overrightarrow{t}_i = [x_{t_i}, y_{t_i}, z_{t_i}] \tag{2.28}$$

and another that goes from the origin to your location:

$$\overrightarrow{r}_i = [x_r, y_r, z_r] \tag{2.29}$$

Then the range vector is:

$$\overrightarrow{\rho}_i = [(x_{t_i} - x_r), (y_{t_i} - y_r), (z_{t_i} - z_r)] \tag{2.30}$$

With a magnitude of:

$$||\rho_i||_2 = \sqrt{(x_{t_i} - x_r)^2 + (y_{t_i} - y_r)^2 + (z_{t_i} - z_r)^2} \tag{2.31}$$

The next portion of this process relies on knowledge of:

- Locations of cell towers
- A measured range
- A weighting matrix

How to actually get this information is covered later but for now assume you have it. The next steps are solution dependent.

2.4.1 Non-Linear Least Squares (NLLS) Solution

Referring back to figure 1.1, for this solution you need:

- $f(r)$ where $r_i = [x_r, y_r, z_r]$
- $H = \left.\frac{\delta f}{\delta r}\right|_{r_c}$

Those are readily available from the earlier defined eqs:

$$f(r)_i = \rho_i = \sqrt{(x_{t_i} - x_r)^2 + (y_{t_i} - y_r)^2 + (z_{t_i} - z_r)^2} \tag{2.32}$$

$$\frac{\delta \rho_i}{\delta x_r} = \frac{(x_r - x_{t_i})}{\sqrt{(x_{t_i} - x_r)^2 + (y_{t_i} - y_r)^2 + (z_{t_i} - z_r)^2}} \tag{2.33}$$

$$\frac{\delta\rho_i}{\delta y_r} = \frac{(y_r - y_{t_i})}{\sqrt{(x_{t_i} - x_r)^2 + (y_{t_i} - y_r)^2 + (z_{t_i} - z_r)^2}} \tag{2.34}$$

$$\frac{\delta\rho_i}{\delta z_r} = \frac{(z_r - z_{t_i})}{\sqrt{(x_{t_i} - x_r)^2 + (y_{t_i} - y_r)^2 + (z_{t_i} - z_r)^2}} \tag{2.35}$$

$$H = \begin{bmatrix} \frac{\delta\rho_1}{\delta x} & \frac{\delta\rho_1}{\delta y} & \frac{\delta\rho_1}{\delta z} \\ \frac{\delta\rho_2}{\delta x} & \frac{\delta\rho_2}{\delta y} & \frac{\delta\rho_2}{\delta z} \\ \vdots & \vdots & \vdots \\ \frac{\delta\rho_n}{\delta x} & \frac{\delta\rho_n}{\delta y} & \frac{\delta\rho_n}{\delta z} \end{bmatrix} \tag{2.36}$$

From here the method depicted in fig. 1.1 is followed until the change in J is less than some tolerance ε defined.

2.4.2 Analytic Solution

This solution uses the given ranges to create a pseudo-measurement that is then used to remove the nonlinear components of the unknowns. This works for any number of readings $\geq$ the number of unknowns (an important point if you have an altimeter and reduce the unknowns to 2) but for this example assume 3 unknowns and 3 readings.

Starting by squaring the range equations:

$$\begin{aligned} \rho_1^2 &= (x_{t_1} - x_r)^2 + (y_{t_1} - y_r)^2 + (z_{t_1} - z_r)^2 \\ \rho_2^2 &= (x_{t_2} - x_r)^2 + (y_{t_2} - y_r)^2 + (z_{t_2} - z_r)^2 \\ \rho_3^2 &= (x_{t_3} - x_r)^2 + (y_{t_3} - y_r)^2 + (z_{t_3} - z_r)^2 \end{aligned}$$

Then create a pseudo-measurement by averaging the readings:

$$\frac{\rho_1^2 + \rho_2^2 + \rho_3^2}{3} = \rho_4^2 \tag{2.37}$$

Where:

$$\rho_4^2 = (x_{t_4} - x_r)^2 + (y_{t_4} - y_r)^2 + (z_{t_4} - z_r)^2 \tag{2.38}$$

Note:

$$\begin{aligned} x_{t_4} &= \tfrac{x_{t_1}+x_{t_2}+x_{t_3}}{3} \\ y_{t_4} &= \tfrac{y_{t_1}+y_{t_2}+y_{t_3}}{3} \\ z_{t_4} &= \tfrac{z_{t_1}+z_{t_2}+z_{t_3}}{3} \end{aligned}$$

Next re-arrange the range equations into the form:

$$\rho_i^2 - (x_{t_i}^2 + y_{t_i}^2 + z_{t_i}^2) = x_r^2 + y_r^2 + z_r^2 - 2x_{t_i}x_r - 2y_{t_i}y_r - 2z_{t_i}z_r \tag{2.39}$$

Subtract the pseudo-measurement from the other equations to remove the $x_r^2 + y_r^2 + z_r^2$ terms, resulting in equations of the form:

$$\rho_i^2 - (x_{t_i}^2 + y_{t_i}^2 + z_{t_i}^2) - \rho_4^2 + (x_{t_4}^2 + y_{t_4}^2 + z_{t_4}^2) = 2[(x_{t_4} - x_{t_i})\ (y_{t_4} - y_{t_i})\ (z_{t_4} - z_{t_i})][x_r\ y_r\ z_r]^T \tag{2.40}$$

Note that all the $x_{t_i}, y_{t_i}, z_{t_i}$ are known quantities so what we actually have here is a matrix equation of the form:

$$b = Ax \tag{2.41}$$

and the general solution to this is:

$$x = (A^T A)^{-1} A^T b \tag{2.42}$$

Note: In the case of more readings than unknowns, instead of creating a pseudo measurement you can just use one of the extra measurements. The mathematics works out the same.

Chapter 3

3GPP

3.1 Overview

Taken directly from their website 'The 3rd Generation Partnership Project (3GPP) unites [Seven] telecommunications standard development organizations (ARIB, ATIS, CCSA, ETSI, TSDSI, TTA, TTC), known as "Organizational Partners" and provides their members with a stable environment to produce the Reports and Specifications that define 3GPP technologies.'[1]. Essentially 3GPP publishes specifications releases and those define what the network and the user equipment (UE, smart phone in most cases) should provide and how they should perform. As of the writing of this document the current 3GPP release is 15.2.0 (each 3GPP release is an update to all specifications, sub-releases update sets of specifications). The specified components this document will be concerned with are LCS, E-UTRA and LPP. LCS stands for Location Services [3]. This specification defines how location services should be implemented in any network (2G,3G,4G). E-UTRA is Evolved Universal Terrestrial Radio Access, it is the air interface between base stations and user equipment (i.e. towers and cellphones)[4]. E-UTRA is another 3GPP creation and for the sake of simplicity take E-UTRA to mean 4G-LTE. LPP stands for LTE Positioning Protocol[5]. This protocol was developed by 3GPP and has been extended by Open Mobile Alliance[6] [7].

Digging into the LCS spec from above what comes to light is that support is defined (almost exclusively by 3GPP) for a myriad of positioning methods in current 4G-LTE networks. Taken from the LCS specification[5] the listed methods supported in LTE are:

- Uplink and downlink cell coverage based positioning methods;
- (Downlink) Observed Time Difference Of Arrival (OTDOA) Positioning method;

- Assisted Global Navigation Satellite System (A-GNSS) based positioning methods;
- Uplink Time Difference Of Arrival (UTDOA) positioning methods;
- Barometric pressure sensor method;
- Wireless Local Area Network (WLAN) method;
- Bluetooth method;
- Terrestrial Beacon System methods.

Hybrid positioning using multiple methods from the list of positioning methods above is also supported. Describing all of these methods in sufficient detail is outside of the scope of this document, but they are all defined in a separate specification for positioning methods[8]. It is useful to note two methods that come up quite a bit, A-GNSS and OTDOA. A-GNSS is assisted GPS and comes in two flavors. The first is to provide faster Time To First Fix by providing the UE with satellite ephemeris data, and the second is to help in low signal areas by using a received data from the UE and cached data from the base station[9]. OTDOA is Observed Time Difference of Arrival and is reliant on multilateration(See mathematics review section for the equations of this problem). The UE measures time of arrival of signals from multiple base stations (at least three) and creates several Reference Signal Time Differences by subtracting two different TOAs[10]. This creates several hyperbolas and where they intersect is where the UE is. The figure below (Reprinted from [11, p. 12])is an illustration of this.

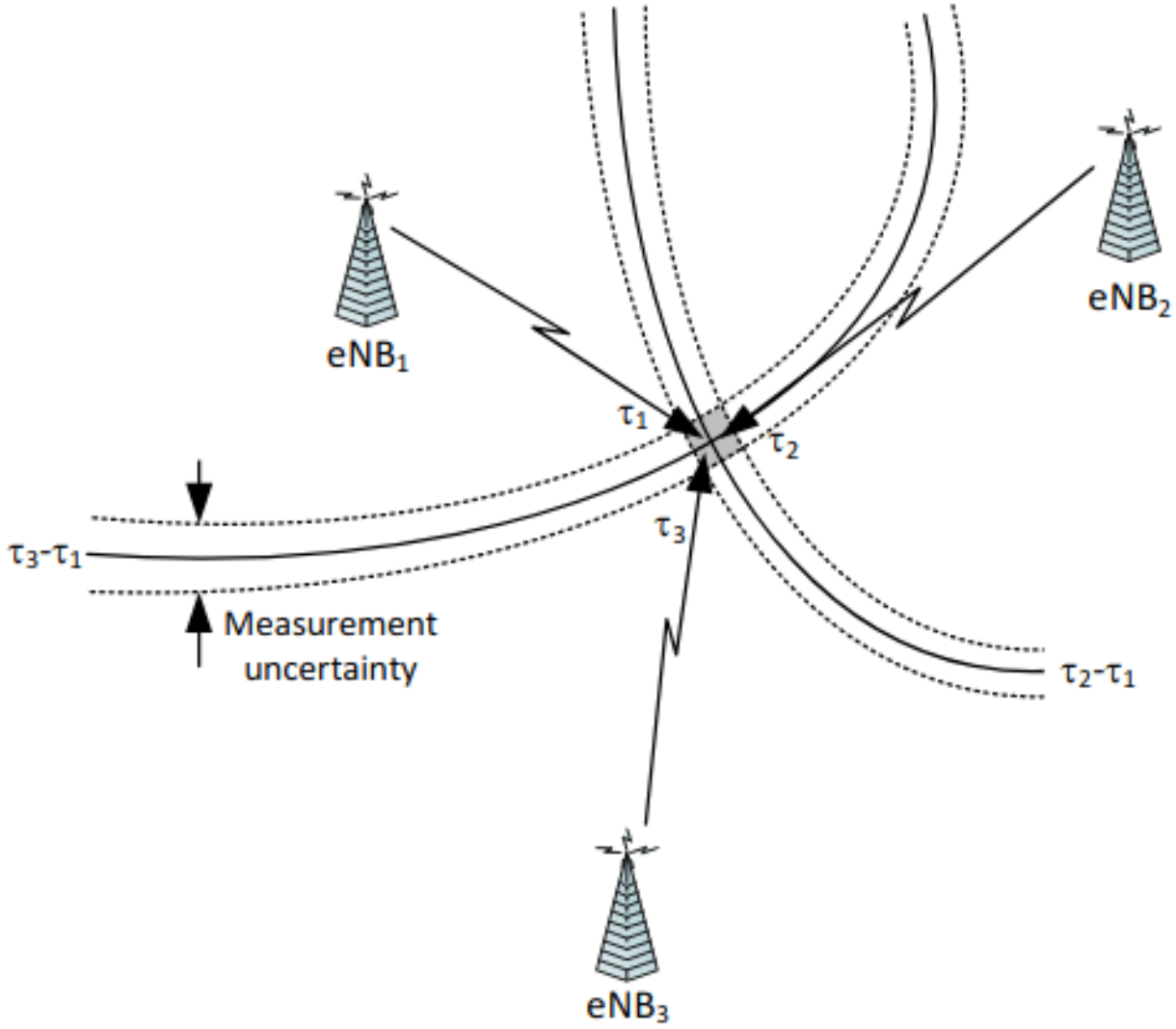


Figure 3.1: Observed Time Difference Of Arrival

3.2 Specification vs Implementation

While the specifications describe exactly what this project is looking for, there are three practical roadblocks that exist. The first and most important one is the actual availability of these methods in the network. The second is the hardware used to interact with the network. The third is the method that we use to communicate with that hardware. It should be noted

that some of this is speculative, given that this project only had dealings with T-Mobile and we focused on cellular modems that were not part of a smart phone. The significance of these two points will be elucidated in the following paragraphs.

3.2.1 Method Availability Roadblock

This specific issue has been the biggest roadblock for the entire project. The idea of using a COTS solution heavily relied on the flow being:

1. Set up a cell modem to talk to a payload computer

2. Ask the cell modem where it is

3. Manipulate that estimate to look like NMEA sentences for the autopilot

With the major portion of the work being step 2. The specifications published by 3GPP point towards the methods being available, along with several statements in the LCS specification [3, Section 4] like:

- The position information shall be reported in standard, i.e. geographical co-ordinates, together with the time-of-day and the estimated errors (uncertainty) of the location of the UE according to specification TS 23.032. The velocity of the UE may be optionally returned in a format specified in TS 23.032

- There are many different possible uses for the location information. The positioning feature may be used internally by the GSM/UMTS/EPS network (or attached networks), by value-added network services, by the UE itself or through the network, and by "third party" services. The positioning feature may also be used by an emergency service (which may be mandated or "value-added"), but the position service is not exclusively for emergencies.

This is however not the case in the networks we had access to. The 'ask where the UE is' methods are cordoned off to certain categories of users. The categories defined in the LCS [3, 4.2] are:

- The Commercial LCS (or Value Added Services) will typically be associated with an application that provides a value-added service to the subscriber of the service, through knowledge of the UE location (and optionally, velocity) and if available, and at the operator's discretion, the positioning method used to obtain the location estimate. This may be, for example, a directory of restaurants in the local area of the UE, together with directions for reaching them from the current UE location.

- The Internal LCS will typically be developed to make use of the location information of the UE for Access Network internal operations. This may include; for example, location assisted handover and traffic and coverage measurement. This may also include support certain O&M related tasks, supplementary services, IN related services and GSM bearer services and teleservices.

- The Emergency LCS will typically be part of a service provided to assist subscribers who place emergency calls. In this service, the location of the UE caller and, if available, the positioning method used to obtain the location estimate is provided to the emergency service provider to assist them in their response. This service may be mandatory in some jurisdictions. In the United States, for example, this service is mandated for all mobile voice subscribers.

- The Lawful Intercept LCS will use the location information to support various legally required or sanctioned services.

In our area of operation, only Emergency and Lawful Intercept have access to location services. That was something that we spent a large amount of time trying to get around.

3.2.2 Hardware Roadblock

The second roadblock is the hardware. As was stated earlier this project looked at hardware that could be added to a small payload computer. We specifically stayed away from just putting a smart phone into an aircraft. The reason we did that is that we want a general solution. Using a phone's OS, sensors, and API ties you to that architecture instantly and any problems with determinism, power, latency, and inaccuracy are now yours to solve. If instead you use an add on modem, you can write general, cross platform source code to manage how you interact with that modem and be able to swap out modems at will. This was an important feature used heavily during the hardware testing phase.

The problem however, is that since we did not use a smart phone, we didn't have access to an API that could possibly have talked to a Location Services Server and done things like Observed Time Difference of Arrival. This is speculation, the API documentation (Android, Apple) doesn't point to *how* location is provided, just that you can get updates[12][13]. In the case of anything using Google Maps the geolocation developer guide lists cell network and WiFi data and the possible inputs for location services [14]. This is exactly what the final solution of this project uses. So most likely the consequence of using an add-on modem instead of a smart phone is that you have to collect, format, send, and reduce the data yourself instead of your phone doing so. It is a relevant piece to point out though, given that the questions asked at the beginning of this document had to do with how your phone knows where you are.

3.2.3 Communication Roadblock

The third roadblock is what command set is available based on the piece of hardware you are using. It is helpful to realize that every cell modem has firmware that has a set of recognized commands. You send a command from your payload computer and that is recognized and translated into a network call sent out by the modem. The available map limits what you have access to within the network. So even if all the methods in the 3GPP spec *are* in fact

implemented in the network and there is a waiting location services server you may not have access to it if the firmware on the cell modem doesn't expose that functionality.

There is an interesting point to make with respect to the interface type in addition to the function mapping one. It starts with how to actually communicate with the chosen hardware. There are two options, AT commands or QMI. AT (or Hayes) commands are serial based ASCII commands supported by all cell modems[15]. QMI is a socket based protocol that is specific to Qualcomm chipsets[16]. AT commands were chosen as the specific interface type for this project for several reasons. The first is that the integration effort is almost zero. While both AT and QMI have manufacturer specific commands, QMI has may also have manufacturer specific drivers. Those drivers may or may not work with ARM systems (a concern for the hardware set used in this problem space) and there are also two widely used linux drivers that work with different support libraries and introduce additional integration considerations[17]. The second is the complexity of the interface. AT commands are just strings you send over a serial port. QMI are packets generated from C code. The learning curve on top of the potential for it not being cross-platform made QMI a non-starter from a practical point of view for this project. All that being said the actual problem here is that for the most part, the QMI and AT command map for each device is slightly different. So something that is exposed in AT may or may not be exposed in QMI and vice-versa.

Chapter 4
PRIOR WORK

This chapter will cover prior work specific to cellular networks and UAS. None of it works inside the network focusing on existing capabilities but it is worth mentioning because of how closely related it is. There is a vast amount of research that deals with signals of opportunity and exploiting available signals to estimate location.

4.1 Prior AFSL Work

The Autonomous Flight Systems Laboratory (AFSL) at University of Washington Seattle works almost exclusively in small UAV payloads and associated software. Prior and current work includes projects in:

- Relative position measurement of visually distinct objects for UAV guidance [18]
- Position estimation via ADS-B Transponder and Local Area Multilateration [19]
- Search and Locate algorithms and implementations [20],[21],[22]
- Risk in UAS over populated areas [23]
- Autonomous collision awareness [24]

The first two items are examples of GPS independent technology developed at least partially by AFSL. They do not deal with cellular or WiFi signals, so are not necessarily germane to the topic this document deals with. They are mentioned mostly for situational awareness with respect to the lab's history. They would be interesting extensions to this project though, specifically the relative position measurement project.

4.2 Zak(Zaher) M. Kassas

There is not a single name that comes up more than Zak Kassas when it comes to using cellular signals for navigation. His publications page [25] has an immense amout of information (papers, theses, patents) dealing with signals of opportunity and specifically with cellular networks. He and his lab have done work with 2G-5G signals, fusion of multiple GPS independent sources, and have used even generated maps in real time to be used by multiple systems [26]. To cover everything he has done just with cellular signals requires a chapter in a book(much longer than these chapters), which he has written[27]. There is one significant difference between his work and this project, he has done all of his work outside of a network subscription. His ASPIN lab[28] has created a software defined radio that monitors signals of opportunity and uses them for navigation. Taken from the APSIN page describing this radio, called MATRIX [29]

" Multichannel Adaptive TRansceiver Information eXtractor (MATRIX) is a state-of-the-art, specialized software-defined radio (SDR). MATRIX treats all ambient signals in the environment, most of which are not intended as positioning or navigation sources, as potential signals of opportunity. Examples of these signals include audio (e.g., AM, FM), television (e.g., HDTV), cellular (e.g., CDMA, GSM, LTE), and satellite communication (e.g., Iridium). MATRIX continuously searches for opportune signals from which it draws navigation and timing information, employing signal characterization on-the-fly as necessary. "

Basically his lab created a radio that can scan for signals, and based off of knowledge about the structure of those signals extract data that is used in ranging and position estimation. His work is much more technical than this project, and has successfully replaced the wheel with another wheel. This project specifically steered away from this focus, because the aim was to use what exists in the network. It is however extremely impressive and the majority of the work that shows up when you run a search in this problem space has his name on it.

4.3 Other Contributors

Other research contributions include Lund University (Improving OTDOA in NB-IoT Systems)[30] in Sweden and Tampere University of Technology(Survey of Trends in IoT Positioning) [31] in Finland. Industry members like Spirent[10] and Ericsson[32] have white papers describing the positioning methods with respect to operating in LTE networks. Outside of research there has been some work in the last several years by several companies on how to integrate UAVs into cellular networks [33] [34][35].The bulk of this work has been aimed at how to provide cellular coverage to other entities using UAVs (via flying base stations that act as relays) or relaying C2/payload data through networks to support Beyond Visual Line of Sight (BVLOS) operations. Currently the only company *possibly* doing anything with UAV localization via cell networks is Verizon, but the only information there is from 2015 and doesn't directly state that that is what they are doing [36].

Chapter 5

PAYLOAD HARDWARE

This section decribes the components of the physical system. Some of the design choices that led to the current configuration are discussed. This is not an exhaustive list but is a representative sample of available COTS solution hardware. Below is a figure of the hardware and signal flow:

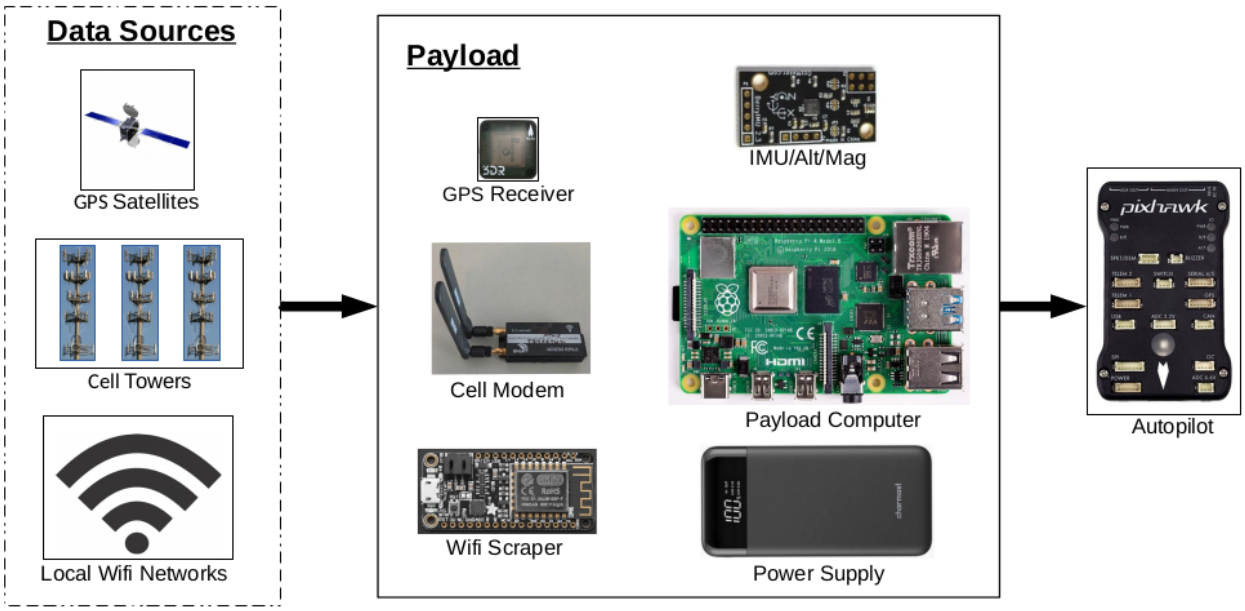


Figure 5.1: WiCe-Nav Hardware and Signal Flow

There are a few discrete pieces that make up this project:

1. Payload computer

2. Cellular modem

3. Wifi Data Scraper
4. IMU
5. GPS Receiver
6. Power Source

The pieces chosen to fit in those spots were:

1. Raspberry Pi 4
2. Telit HE910D, Adafruit FONA, Sierra MC7455
3. Adafruit Feather Huzzah
4. OzzMaker BerryIMU v2
5. VK-162
6. Charmast USB-C Power Bank

All of these choices required some research, but everything besides the Raspberry Pi and the cell modems were chosen almost entirely for ease of integration. They all accomplished the job they were needed for, were powered by 5 or 3.3V (the Pi outputs), were cheap, and had self explanatory interfaces (like a serial port) or had existing code bases with Pi integrations.

5.1 Payload Computer

The choice of payload computer did take some time, but again ease of integration was a driving factor. A google search of "single board computer" gives a good example of the available hardware. The Raspberry Pi 4, Nvidia Jetson TX2, and ASUS Tinker were the

front runners in the 'cheap and relatively easy to get up and running' category. The Pi was chosen because it met the needs of the project and it has a large base of integrations and support.

As an aside, the needs of the project that drove the payload computer choice were:

- The computer has I/O for serial, USB, and network.
- The CPU can handle the load of the software, with remaining overhead.
- The computer can run a popular linux distribution.
- The computer is not overly power hungry.
- The computer doesn't require cooling that can't be applied easily.
- Ideally there is a large base of example integrations (Ease of integration).

The first two points are really the only non-starters. If you can't talk to devices and you can't run your software then you have a flying brick. Determining if the CPU can handle the load of the software before you write the software is mostly speculative. The assumption used here was that all the code would be C or C++, and that everything would be as simple as possible. Checking that the CPU is relative new and that (available RAM) $\geq$ 1GB was used as the threshold 'Good' values.

The rest of the qualifications are mostly for ease of integration. Running a popular linux distribution will allow for you to easily pull in third party applications and libraries (via 'sudo apt install' or similar), and is free. Power considerations can be a non-starter but only if you don't do your due diligence up front (i.e. spec out the pieces you plan on pulling in). Cooling can be a challenge but also something that should be uncovered during the research phase when looking for example integrations. Example integrations are a huge help, this cannot be overstated. Having working examples that use the same pathways as planned can unstick a project quickly.

Ticking through the criteria the Pi runs Raspian, which is a derivative of Debian. The CPU is Cortex-A72 released in 2015, there is 4GB of RAM, and there is no shortage of I/O. The Pi is powered by a 5V 3A supply, which is a common supply type for USB-C power banks. Cooling was needed but a push-pull setup with two small 5V 30x30x10mm fans was sufficient to keep the CPU temperature stable under load. Raspberry Pi SBCs are massively popular and have a very large library of example integrations. There have been no issues as of yet with choosing the Pi as the payload computer.

All of that being said, the NVIDIA Jetson was a close second but it was decided that it was overkill for the application. It would be a good start for extending this project for more computationally intensive applications.

5.2 Cellular Modem

The cell modem only had two criteria:

- Have useful functions exposed in AT Commands
- Have a common interface (no soldering, easy to get data out)

There is a litany of companies that sell cellular modems. This project uses Sierra, Telit, and Adafruit products. The Sierra initially looked promising but did not turn out to be useful for anything other than internet access in the WiCe solution. The Telit was chosen because its ATMONI function was found to be one of the few exposed AT commands out there that directly reports signal characteristics that were needed for solution 1. The Adafruit Fona was used because it goes one further and has a method that reports latitude and longitude of the modem.

Cellular modems can come in a variety of form factors. The most common ones are:

- Land Grid Array (LGA)
- Peripheral Component Interconnect express (PCIe)

- M.2

LGA sockets are common on CPUs and require a specific mating socket for each manufacturer. This made those form factors non-starters. M.2 is the newest of the three (2013) and there are not a lot of small break-out boards commercially available to handle these types of boards i.e. M.2 cellular modems. Boards exist for a variety of other applications. PCIe is older (2002) and has been the industry standard for quite a while. There is a large set of cheap break-out boards available, there is a wealth of information in the form of previous integrations, and most companies still have a large selection of PCIe boards. Given that information, this project focused on PCIe boards with the exception of the FONA, which is an LGA module already integrated into a breakout board. M.2 is slowly replacing PCIe as the standard so extensions to this project should re-evaluate the availability of boards and modems regularly.

Chapter 6

PAYLOAD SOFTWARE

6.1 Overview

The WiCe-Nav software solution is a set of C++ executables that use a publish/subscribe pattern for inter-process communication. Each module only does what it needs to and they all use the same basic setup. This keeps things straight forward, easy to manage, and easy to refactor. This section will focus on the what and why of the software. Following this overview a high level module overview will be given. Following that there will be a description of the libraries chosen as helpers for this project. Finally the build environment will be discussed. Actual examples (the how) and interface definitions are given in Appendix A. Below is a diagram of the data flow in WiCe-Nav:

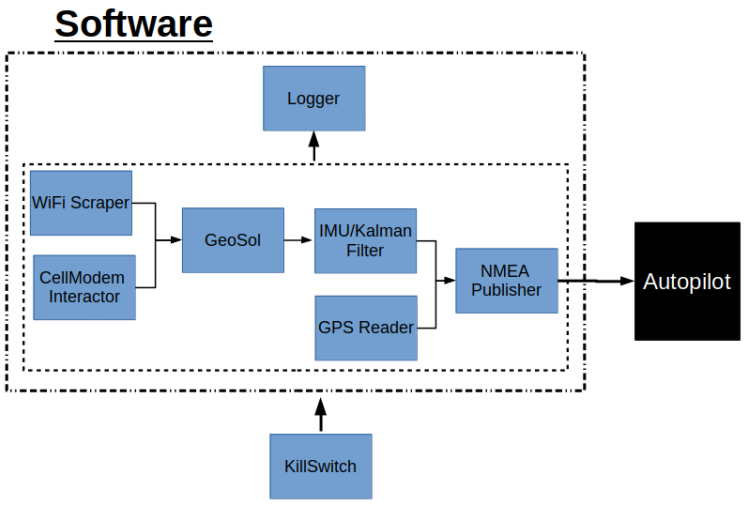


Figure 6.1: WiCe-Nav Software Data Flow

There are a couple things that should be clarified in fig. 6.1. The arrows represent the flow of data *out* of modules. The dashes boxes represent mass subscription, i.e. the Logger subscribes to everything in the box that points to it and everything in the box that the KillSwitch points to subscribes to it.

The attentive reader will note that this is only one of the solutions used when three are advertised. That is because this is the only one that has worked. The others are presented as a survey of available solutions and how they fall short. Design and development for this project took the following form:

1. Identify a possible solution.

2. Implement enough architecture to gather necessary flight data.

3. Post-process in Matlab, analyze whether or not the solution is promising.

4. If it is, develop the rest of the functionality in C++ and push it to the payload.

It should also be noted that not everything in the diagram has been implemented. The current state of the project does not fuse GPS and WiCe estimation data, the IMU was not pulled into the tested WiCe-Nav solution presented later, and GPS data is currently only used as truth data. Also the NMEA Publisher has not been written yet, which means the autopilot integration portion is still untested. Those two pieces are in the design phase and will be implemented in future work.

6.2 Software Modules

Each block in fig. 6.1 represents a stand-alone executable. Each module has command line parameters that configure the executable. They also all have the same basic structure. This is done for several reasons. First, it was important to the designers of the project that everything followed the single responsibility principle from the SOLID principles, i.e. each piece handles only the thing it needs to; it has a single responsibility. This makes finished

pieces stable and keeps everything separated appropriately. Secondly the structure abstracts away (or makes variable) the points and rates data flows in and out of the software. This means to change things like the serial port, the publish socket, or the tick time all you do so change a variable in a script. You don't have to rebuild anything, you just re-start the module. Finally, each module is essentially implementing a template. Everything is more or less in the same place in every module so a developer knows where to look and how all the pieces interact. The figure below shows the structure:

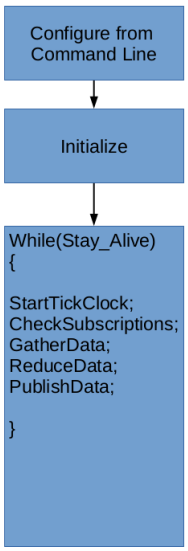


Figure 6.2: Software Module Structure

First the command line parameters are parsed. Those set things like publisher and subscription sockets, tick time (how long each step through the while loop takes), and other module specific parameters. Once the parameters are known, all of necessary variables are initialized and the module runs until the KillSwitch tells it to stop.

It should be noted that fig. 6.2 is only of the 'main' file. Each module also has places for utility static functions, socket handlers, and module specific classes. Those are not as easy to show in a diagram but the structure is there in the code. The point here is that the main files across the modules *should* look the same and be abstracted like fig. 6.2.

A call to start a module could look like the following:

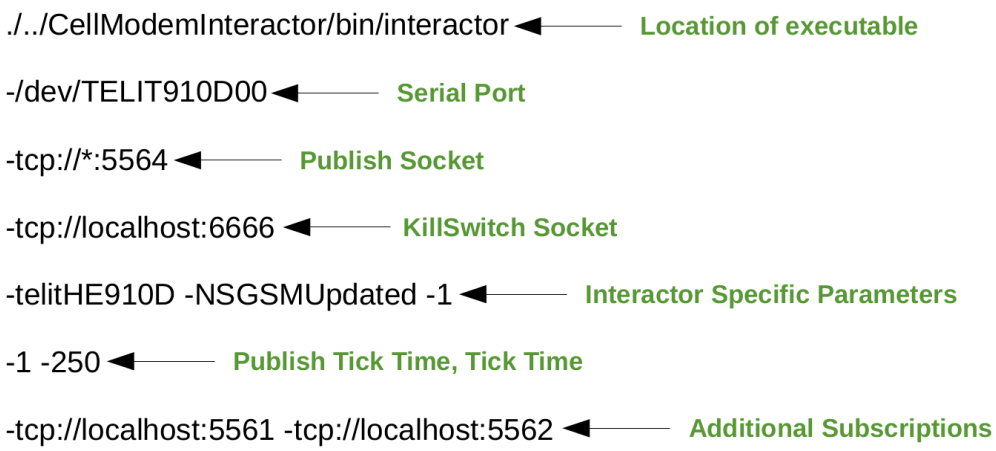


Figure 6.3: Module Start

It may be obvious but it should be noted that this call would typically be a single line in a script. It was broken up to show the different pieces. All of the modules have slightly different needs as far as configuration parameters go but the order of publish, kill, subscription is the same on all of them. You can see from fig. 6.3 that as was mentioned earlier, the data flows and rates are all command line parameters. To make the tick time 5 seconds and the serial port /dev/SomethingElse you simply change the script that starts the module. Some of the modules like the interactor also use a C++ version of the factory pattern. This returns pointers to the specific class that represent a specific configuration, i.e. a Sierra cellular modem has a different controller than a Telit modem and that is configurable via a command line parameter.

6.3 Important Libraries

The interfaces used throughout this project are not new. Like many things in software, very good solutions for using the interfaces already exist so they were leveraged here. There were three libraries pulled into this project that did a lot of the heavy lifting. They were:

- libserial
- ZeroMQ
- nmealib

6.3.1 libserial

Developing C++ software that interfaces with serial ports in a Linux environment is not as easy and straight forward as it is in Windows. The termios structure consists of a set of flags that are configured via bit maskings[37]. To that end, libserial was used to configure serial ports. The cellular modems, WiFi scraper, and GPS are all serial ports with default settings outside of the baud rate so this was an easy way to abstract a problem that had already been neatly solved elsewhere.

6.3.2 Zero MQ

The software developed for WiCe-Nav is multi-process. This was a design choice made because some of the hardware pieces run at different rates and some things need to wait or periodically check for new data. This means that at a minimum it needed to be multi-threaded. Setting up a multi-threaded application quickly is asking for trouble when it comes to debugging and re-factoring, so the choice to go multi-process (which by definition is multi-threaded) simplified the problem at the cost of latency. Zero-MQ was the result of a study of the inter-process communication solution space. It is low latency(15ms average measured, 26ms peak), simple to implement, and supports the publish/subscribe architecture that was decided on shortly after the multiprocess decision was made.

6.3.3 nmealib

The most common form of output data from a GPS receiver will be some set of NMEA sentences[38]. These are strings with various markers and values that represent items like

latitude, longitude, velocity, and DOP quantities. This has been the case for quite a while. So in keeping with the theme of the project, we elected to use the wheel. nmealib is a set of functions that parses nmea data and outputs the values encoded within the sentences. It also has the ability to generate NMEA sentences from data, which will make it very useful in the NMEAPublisher block.

6.4 Build Environment

The software was developed initially on Ubuntu 18.04 using Visual Studio Code and built for the Raspberry Pi on the Raspberry Pi using Visual Studio Code for Raspberry Pi developed by HeadMelted. It was compiled with g++ version 7.5.0.

Other important notes:

- libserial was built from source (v 1.0). The 'apt install' version will not work with this software. Also for Pi builds the following packages were installed (on the Pi, via apt install):
  - automake
  - libtool
  - sip-dev
  - libboost-all-dev
  - libgtest-dev
- ZeroMQ version 4.2.5-1 installed via: apt install libzmq3-dev
- nmealib version 0.5.3 downloaded from sourceforge and massaged to compile with the c++17 flag. All nmea files are in the build for the GPS receiver, there is no need to download them.

Chapter 7

TESTED SOLUTIONS

7.1 Introduction

There were two types of testing that were done to gather data, ground and flight. The ground routes were in North and South Seattle and the flight test area was in Carnation, WA. Flight testing was done in a Sabre 1900mm foam UAV. Ground testing was done in a Subaru Impreza. See figures below:

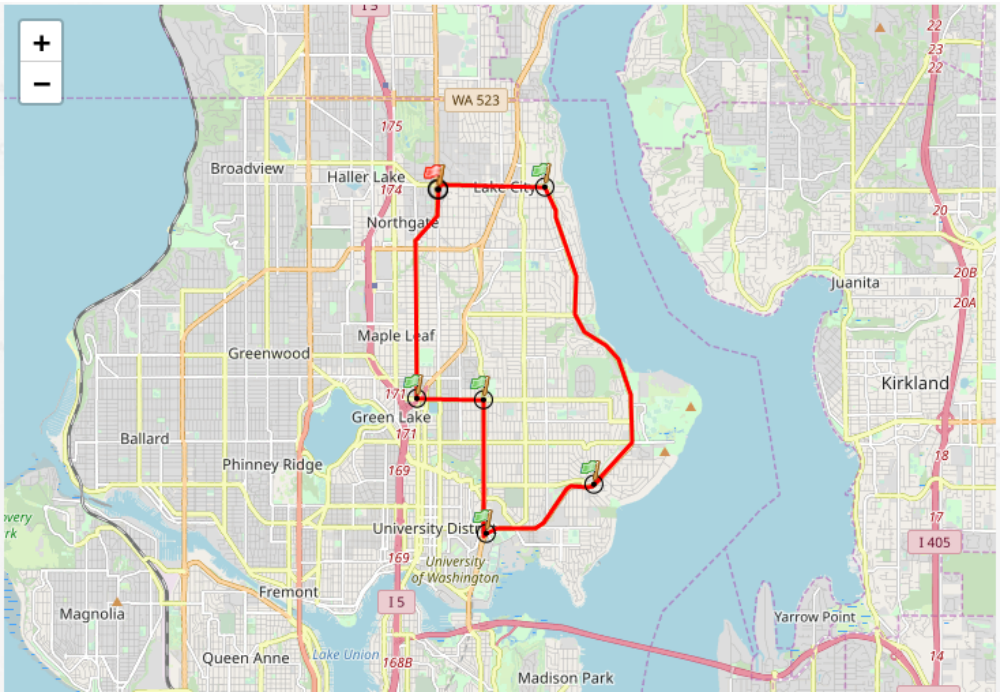


Figure 7.1: North Seattle Ground Test Route

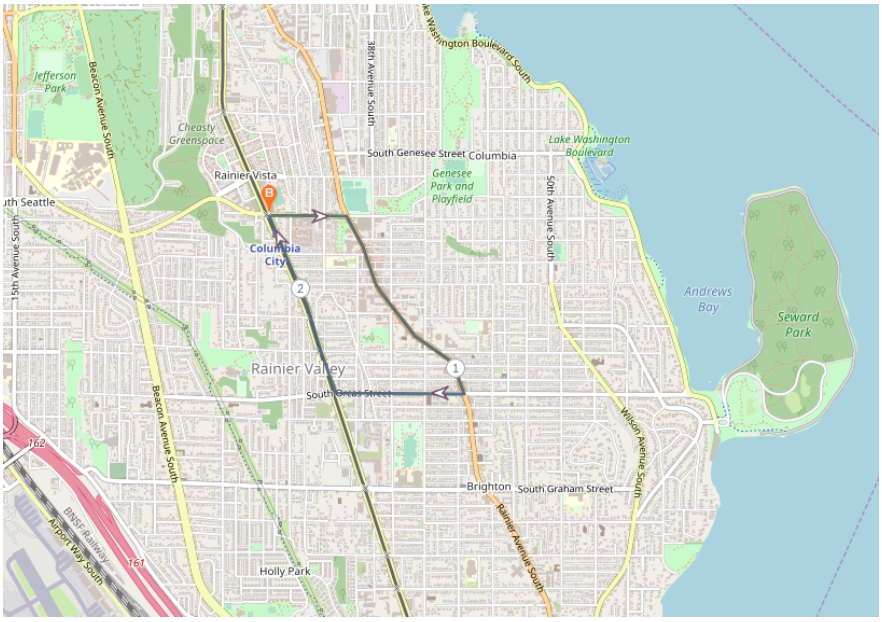


Figure 7.2: South Seattle Ground Test Route

Figure 7.3: Flight Test Route

Figure 7.4: Flight Test Install

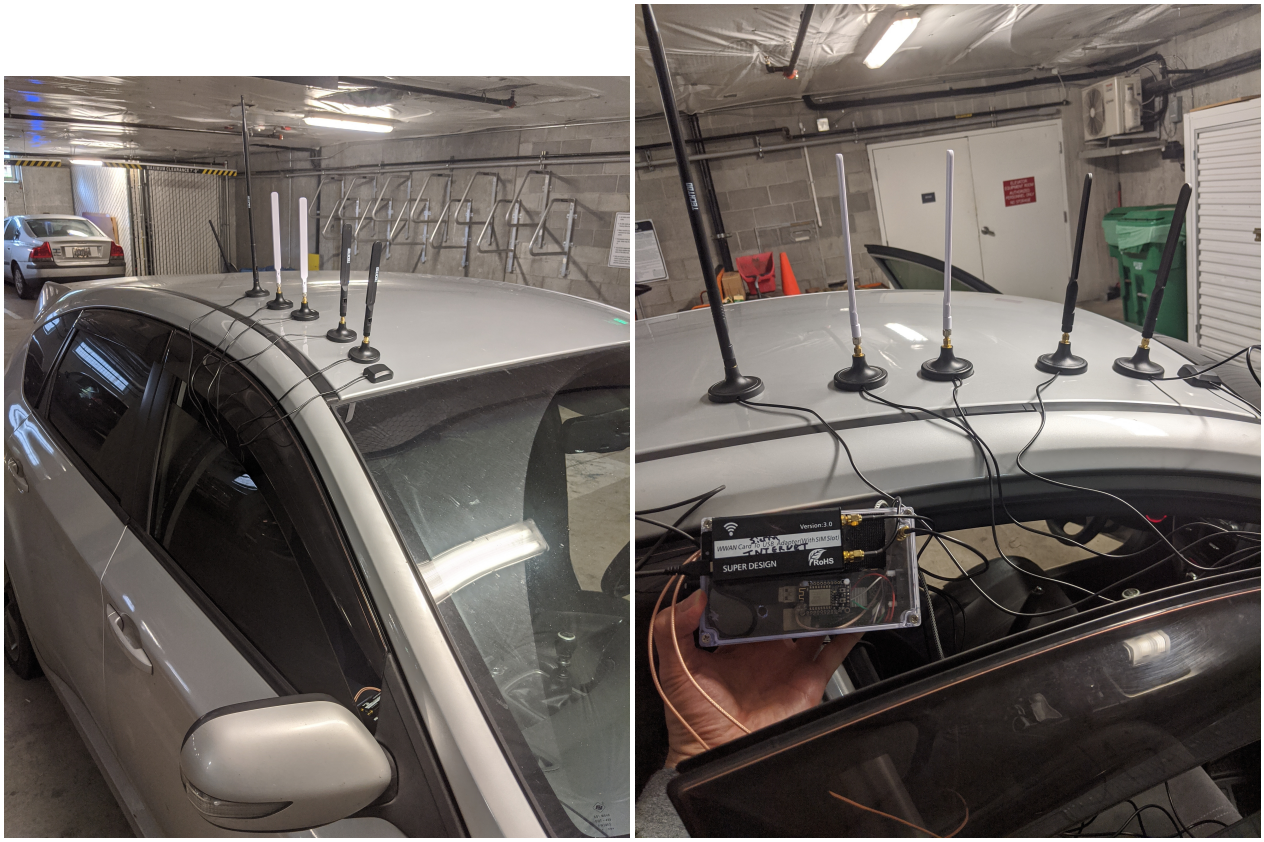


Figure 7.5: WiCe, Ground Test Install

7.2 Solution 1: TelitHE910D, ATMONI

7.2.1 Overview

This solution has a fair amount of problems with it. The further along in development it progressed, the more it became apparent that the estimate it would eventually return would not be very useful. It was however, a good way to accomplish a couple of tasks:

- Test the system in a dynamic environment, where it is passing and logging data.
- Actually test methods that are available, seeing if a simple approach is viable.

- Provide some insight into what cells signals look like on the ground and in the air.

This solution attempts to estimate location using multilateration(see section 2.4). The hardware used was:

- Raspberry Pi 4
- VK-162 (For truth)
- TelitHE910D
- Charmast USB-C Power Bank

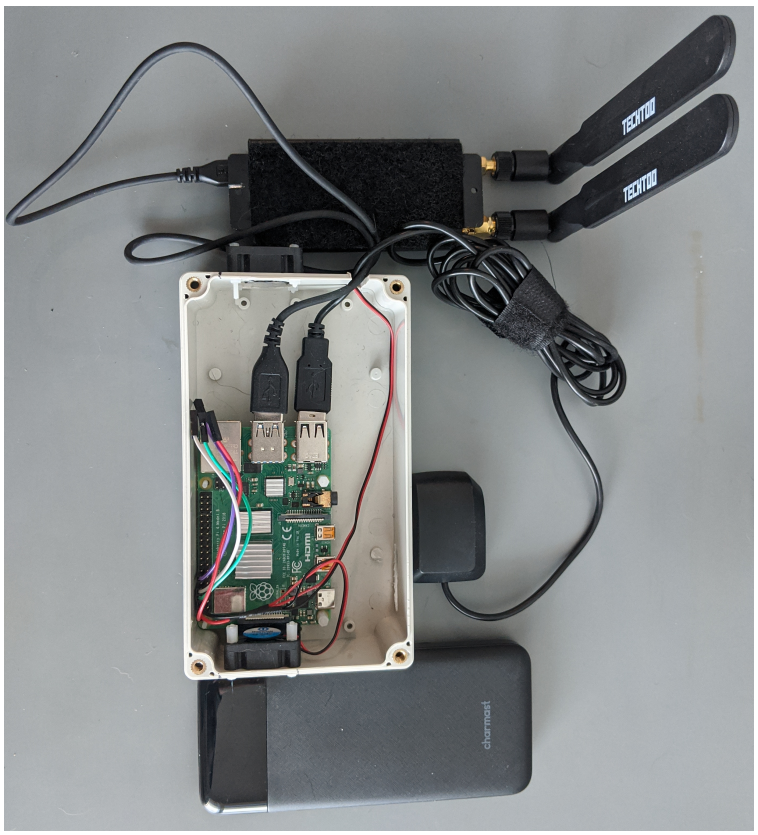


Figure 7.6: ATMONI Solution Hardware

The software modules used in this solution were:

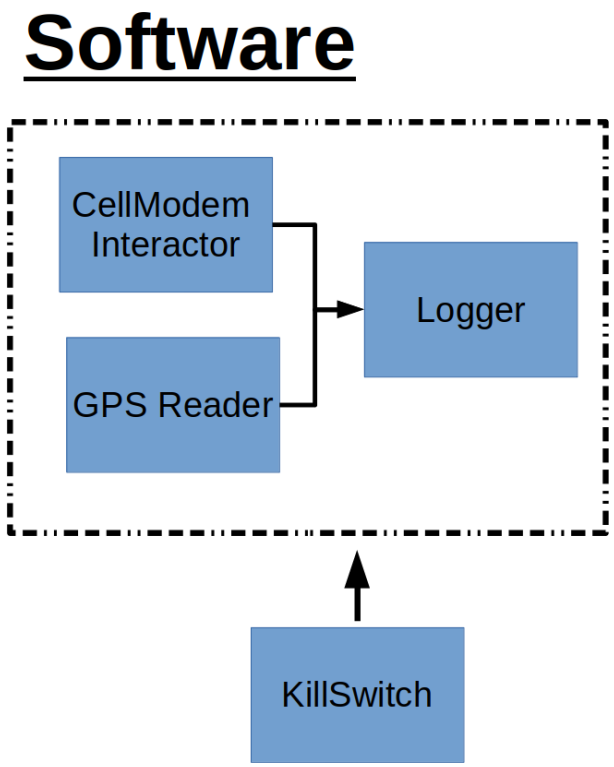


Figure 7.7: ATMONI Solution Software

The CellModem Interactor uses a Telit controller. It issues the AT#MONI command to the cell modem, captures the response, and processes the data. It is built to publish raw cell data, position estimate, or both. The GPS reader simply reads the data from the GPS receiver, parses it, and publishes it. The KillSwitch publishes a kill message that every module subscribes to. The message causes each module to exit its main loop, release resources, close connections, and quit execution. The logger subscribes to the modules and logs the data published.

The unknowns here are ranges and locations of cell towers. As is turns out these are not trivial problems to solve. The next three sections cover what the issues are and how this solution attempted to solve them.

7.2.2 Available Data

This has been the sticking point again and again in the project and its effect is seen here the most. The problem is that in order to solve the problem we need ranges from multiple known points. If there was a method that simply reported something like:

$$[r_1\ x_{t_1}\ y_{t_1}\ z_{t_1},\ r_2\ x_{t_2}\ y_{t_2}\ z_{t_2},\ r_3\ x_{t_3}\ y_{t_3}\ z_{t_3}]$$

Then applying the multilateration algorithms developed earlier would be straight forward. That method may exist, but as pointed out earlier has not been made available to us. Therefore what was needed was to figure out what we could get from available COTS modems, i.e. what kind of data is exposed in the AT command maps.

Some things to be aware of are some of the terminology and facts with respect to your phone and the towers it talks to. First, towers are not typically a single omnidirectional antenna. They are comprised of cells. See figure below (reprinted from OpenCellID FAQ):

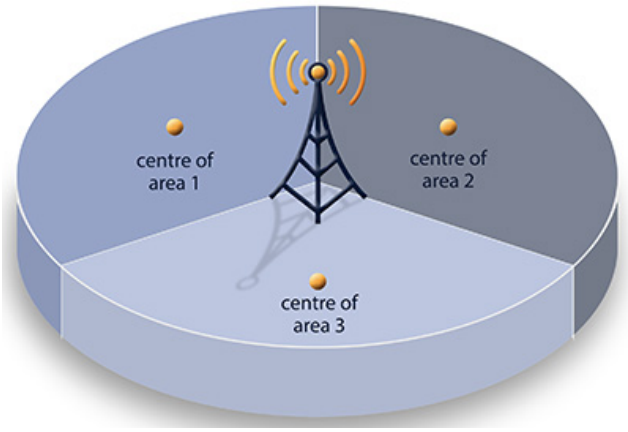


Figure 7.8: Tower with 120° cells

The cell on the tower your phone is actually connected to is called the serving cell, and the cells on the towers around that serving cell are called the neighboring cells[39]. The area a cell serves is represented by a circle and the radius of that circle is the serving radius. See figure below [39, reprinted from p.4]

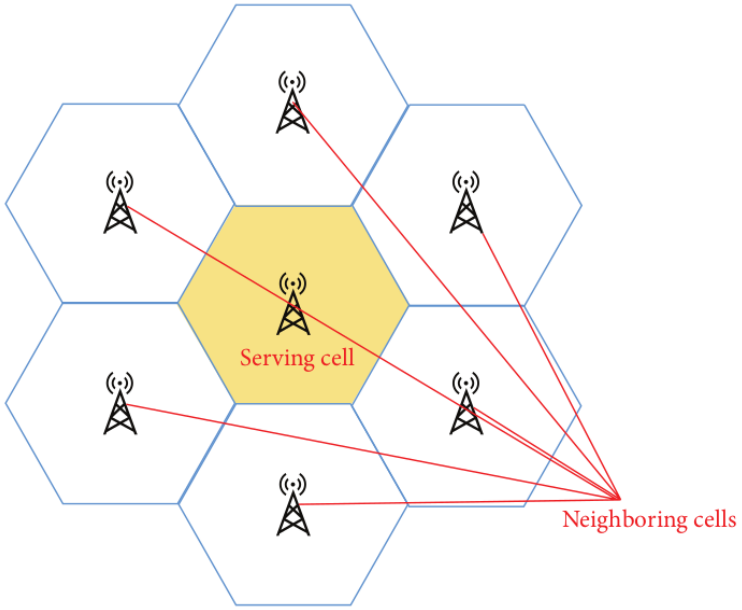


Figure 7.9: Serving/Neighboring Cells

*Note:*Serving radii can and do overlap.

So what is needed here is the distance of the cellular modem to its serving cell's tower and at least two of its neighboring cell's towers. In addition to that, we need to know where the towers are that those ranges correspond to. As was stated earlier, neither of those are reported by any method available in COTS modems. So what can we get? The answer with respect to ranging is RSSI. The answer with respect to cellular towers is that you can get approximate cell locations, and that depends on the type of network you are using. This is simplified a bit by the fact that there is really only one large database of cell towers available, and it uses data typically only reported via cellular modems in 2G networks. So that is what this solution used. 2G network tower data and RSSI. The general idea here was to:

- Get a map of local tower locations loaded in memory
- Query a sensor for ranges and tower identifiers

- Query the map for tower GPS locations using the tower identifiers
- Use a sensor model to estimate range
- Using ranges and tower locations, use one of the multilateration algorithms developed earlier to estimate location.

7.2.3 Cell Tower Locations

The only real option for downloading tower location data is OpenCellID. The data in the csv file available for download looks like the following figure:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	radio	mcc	net	area	cell	unit	lon	lat	range	samples	changeable	created	updated	averageSignal
2	GSM	302	720	42980	11334	0	-122.32933044434	47.706069946289	1000	1	1	1325201054	1325201054	0
3	GSM	310	410	42980	21532	0	-122.29225158691	47.726669311523	1000	1	1	1410220109	1410220109	0
4	GSM	310	410	42980	11578	0	-122.31422424316	47.733535766602	1000	1	1	1411484587	1411484587	0

Figure 7.10: Sample Open CellID Data

The database of tower locations comes with some caveats. First, the download is massive. It's nearly a 1GB .csv file with millions of lines. So if you are looking for a small database of local towers, you have to reduce the file. Second, OpenCellID works off of users contributing data to the database. The data reported is their GPS location + serving cell + RSSI (if available). The location in the database is the average GPS location of the users who have reported that cell. So it's not exact and it's a cell location, not a tower location. Finally all you get is latitude and longitude of the cell, no elevation.

All of these are not insurmountable problems, they just add to more tasks. The first one can be handled fairly easily just via a program that can read csv files and filter based on a grid and radio type, which is what was done here. Using the known bounds of the operational area (plus a small margin) and that the radios are GSM (2G) several mini-maps were created for use in the test areas. There is a Windows application that an OpenCellID

user developed for this same purpose, but given that this project was developed on and targeted at Linux, that was not much use here.

The second problem was is where Google Geolocation API enters the picture. Given the same identifiers that the OpenCellID database uses, the Geolocation API will return a lat,long, and serving radius of the cell[14]. The assumption here is that Google's database is more accurate. That is purely a common sense assumption given that they are a large company and that provides a paid service that utilizes their own database of locations.

The third problem is solved by another Google creation, the Elevation API. Given a latitude and longitude, the API will return an MSL elevation of the ground at that point on earth. Granted, the transmitter won't be on the ground, so this is another approximation.

So starting with the original OpenCellID download the following steps were taken:

1. Filter by operational area and radio type to create local map.

2. Run the local map through the GeoLocation API to update locations.

3. Run the updated locations through the Elevation API to get elevation.

* After ground/flight tests, these steps were repeated for cells that were read but not in the OpenCellID database.

The final form of the map looks like the figure below:

	A	B	C	D	E	F	G	H	I
1	radioType	mcc	mnc	lac	cellID	lat	long	servRadius	elevation
2	GSM	310	260	41391	292	47.7108636	-121.997101	2805	15.4104595184326
3	GSM	310	410	52049	54233	47.7062358	-121.9151229	1965	131.622161865234
4	GSM	310	410	52023	21423	47.6981473	-121.9182768	1934	87.1281433105469

Figure 7.11: Sample Updated Map Data

From an operational perspective, the cells were gathered in real-time via the Telit modem's AT#MONI command[40]. The data looked like the figure below:

```
#MONI: Cell BSIC LAC  CellId ARFCN  Power  C1 C2 TA RxQual PLMN
#MONI: S    64 A1AF 11F1   689    -61dbm 46 46 0  0      AT&T
#MONI: N1   34 A1AF 12F5   759    -63dbm 44 44
#MONI: N2   33 A1AF FFFF   756    -68dbm 39 39
#MONI: N3   12 A1AF E20F   760    -84dbm 15 15
#MONI: N4   37 A1AF 11FA   687    -85dbm 22 22
#MONI: N5   FF FFFF 0000   691    -111dbm -1 -1
#MONI: N6   FF FFFF 0000   757    -111dbm -1 -1
```

Figure 7.12: Sample AT#MONI Data

S = Serving, N1-6 = Neighboring Cells. Given this data the hex strings in the data were cast into doubles and then the map was queried to get GPS coordinates of the towers.

Note: The argument could be made here to just use the Geolocation API instead of having a map in memory, and that is a valid point. This was done in the WiCe solution, but comes at the cost at indeterminate latency(due to varying cell signal) and the need for an additional modem (due to Linux's Modem/Network Manager). That was something the designers of this project wanted to avoid (specifically the latency) and was only entertained when all the other options were exhausted.

7.2.4 Range Estimation

The ranges are estimated from received signal strength (RSSI). This was gathered via the Telit modem's AT#MONI command[40], represented as power in the data gathered (See fig. 7.12). The metric measured power lost due to distance, and this is where the sensor model comes into play. There are a large number of models that exist in literature, but one simple one that comes up fairly often is the modified Friis equation[41]:

$$P_r(D) = P_{r_1} - Klog_{10}(D) \tag{7.1}$$

Where:

$$P_r = \text{Power Received (RSSI)}$$
$$P_{r_1} = \text{Power Received at 1 meter}$$
$$K = \text{constant}$$
$$D = \text{Distance from transmitter(meters)}$$

Given RSSI and true distance, P_{r_1}, K can be experimentally determined using gathered data and linear least squares(see section 2.1) where:

$$y = \text{RSSI}$$
$$H = [1, \ -log_{10}(D)]$$
$$x = [P_{r_1}, K]^T$$

An example of the sensor model looks like the following (using P_{r_1}, K from [41]):

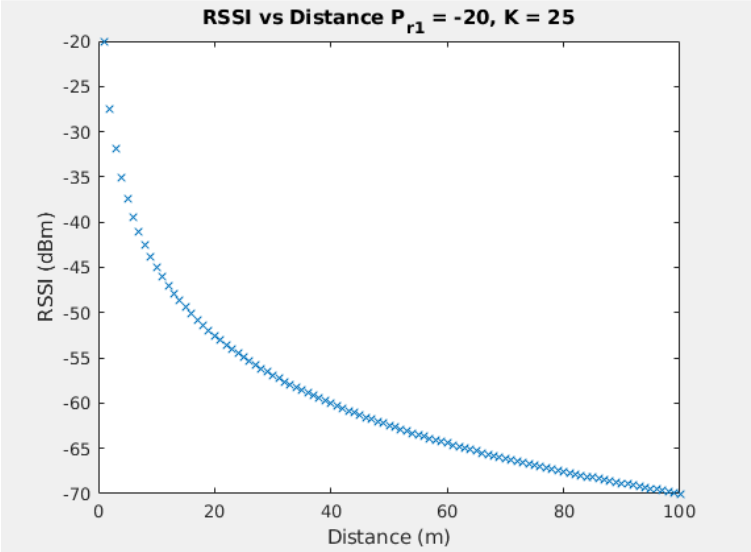


Figure 7.13: Example Sensor Model

Again, the astute reader might realize that this introduces another problem. If you have to experimentally determine these constants for each cell before you can estimate range, then you'd need to gather data *before* you lose GPS. That is correct. The plan was to

gather data in an area first, reduce the data to estimate the sensor model parameters and see how they varied between cells. If they were fairly close, the values could be averaged and then applied to all towers. This turned out to not be the case as will be shown shortly. The more important criteria is if the sensor model is accurate and the end result was a good position estimate. This also turns out to not be the case. Again, the further into development this method went, the more its purpose became to stress the system and see all the software/hardware pieces in action, in the air.

7.2.5 About Data Reduction

For this solution (and all the others) the GPS receiver published data at 1Hz. The cellular modem gathered data at 4Hz. Every piece of data gathered has a system time stamp attached to it. So using the time stamps as a marker, GPS locations were linearly interpolated for each of the cellular modem readings. This is what is used as truth data. In the subsequent sections, this truth is compared against and deltas (estimate - truth) are taken and shown. Also, all the data reduction is done in Matlab.

7.2.6 Sanity Check

To test the software built to reduce the data, a sanity check was done using 'perfect' data and then injecting gaussian noise $N(0, R)$ into that perfect data. The perfect data used the locations of the towers from the database and created the ranges from the interpolated GPS locations. So it assumes zero mean gaussian noise and that the locations of the cells = location of the towers. The very first test used $R = 0$, Non-Linear Least Squares Algorithm and is shown below:

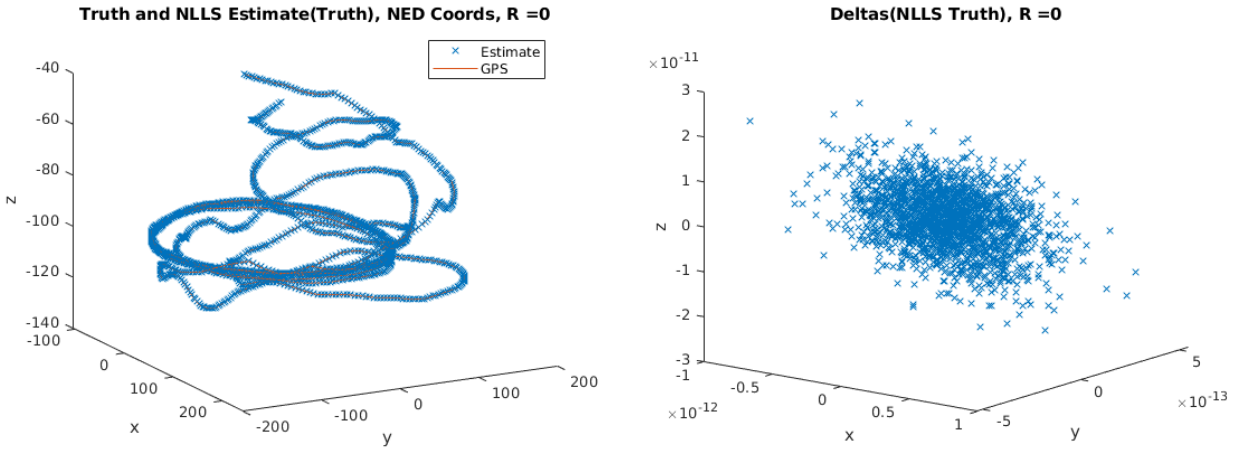


Figure 7.14: R = 0, NLLS Position Estimation

Note: GPS coords were changed to NED (North-East-Down, local flat earth). A reference altitude of 0 was used, and in this frame positive z is down. So a z coordinate of < 0 is above ground here.

Looking at the two graphs everything looks to be in order, the deltas are on the order of machine precision. Next the data was filtered to capture just the cleaner orbit data and noise (via Matlab's randn function multiplied by R and added to the perfect signal) was introduced. Below is $R = 10$:

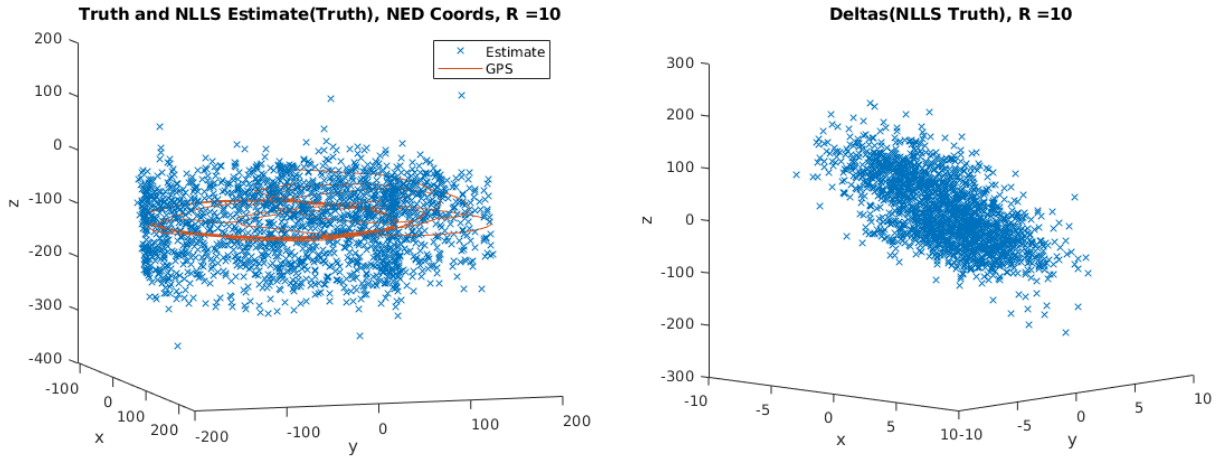


Figure 7.15: R = 10, NLLS Position Estimation

The deltas in the x and y direction are what one would expect, but as you can see the solution is sensitive to noise in the z direction. As a check the same data was run through the analytic solution. See figures below:

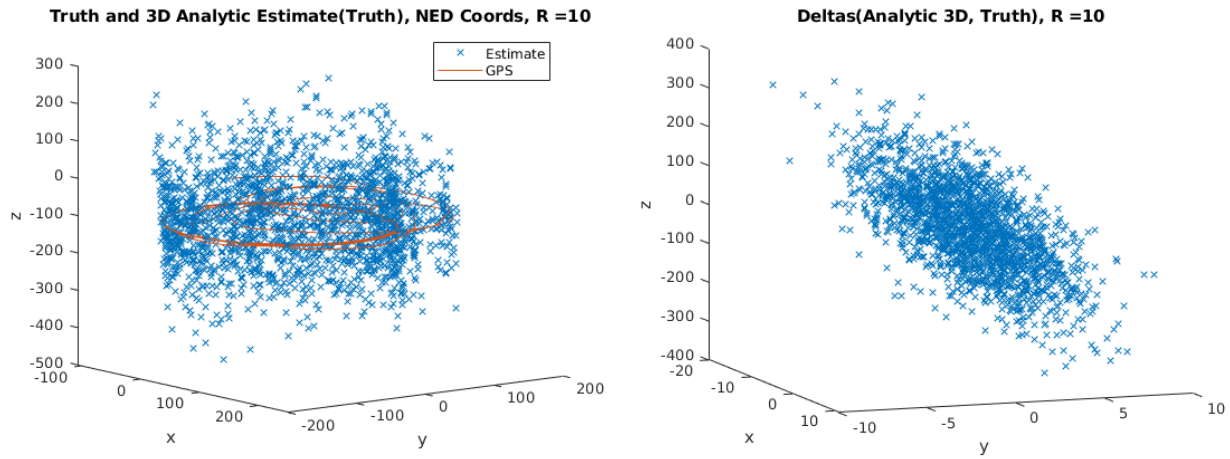


Figure 7.16: R = 10, 3D Analytic Position Estimation

The analytic solution shows the same z sensitivity. This isn't actually a problem, given that an altimeter was always part of the hardware plan. Using the z coordinate of NED as

a stand in for the altimeter and running the 2D analytic algorithm puts the deltas back to where they should be:

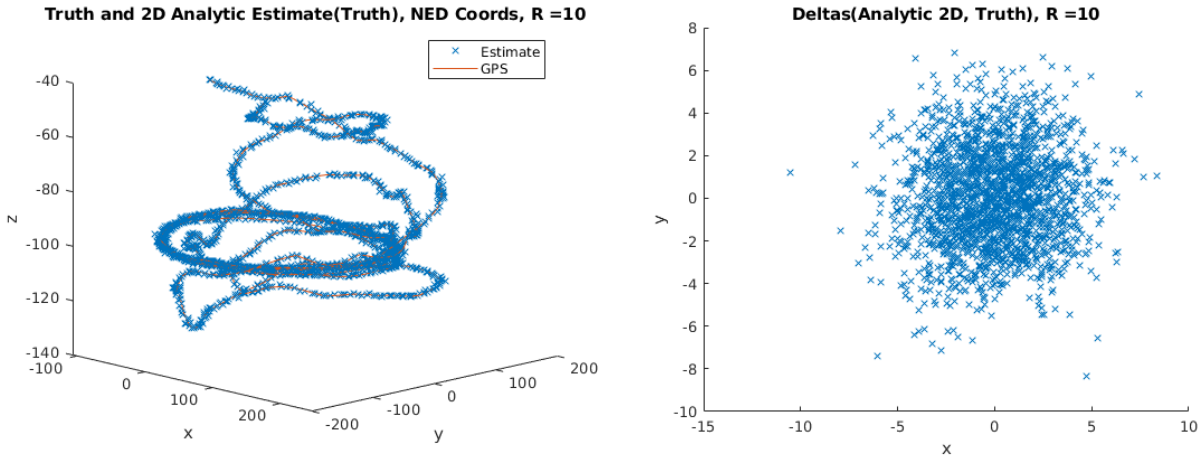
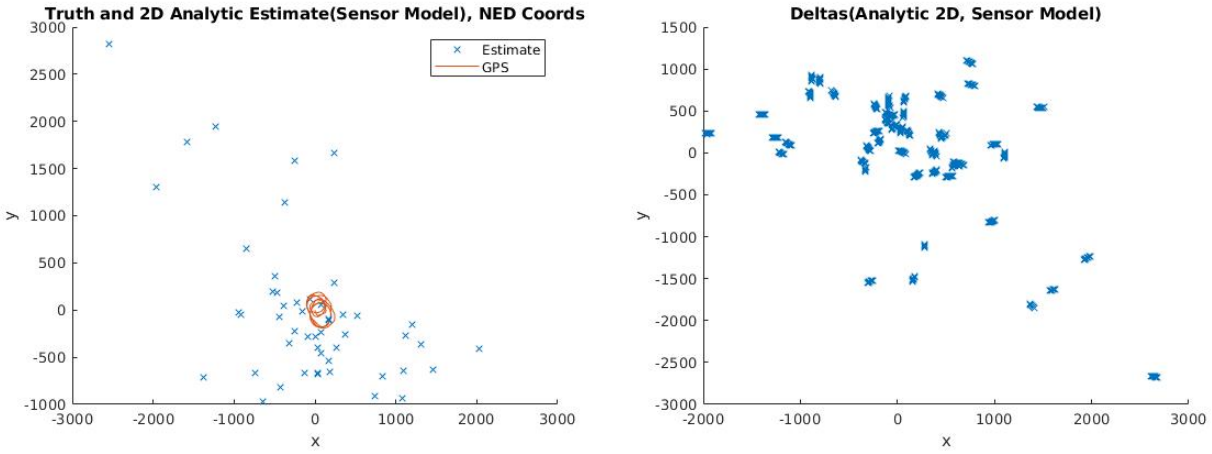


Figure 7.17: R = 10, Analytic 2D Position Estimation, Flight

7.2.7 Flight Data

Running the flight data through the 2D analytic algorithm and using the sensor model produces the following output:



(a) Carnation Flight 2D Position Estimate (b) Carnation Flight 2D Position Estimate Deltas

Figure 7.18: Carnation Flight Estimate Data

Not exactly ideal results. Further, the above filters out estimates that are more than 5km off the last estimate. This is where the sensor model and approximate locations come into play. There are a total of 5 cells read throughout the flight:

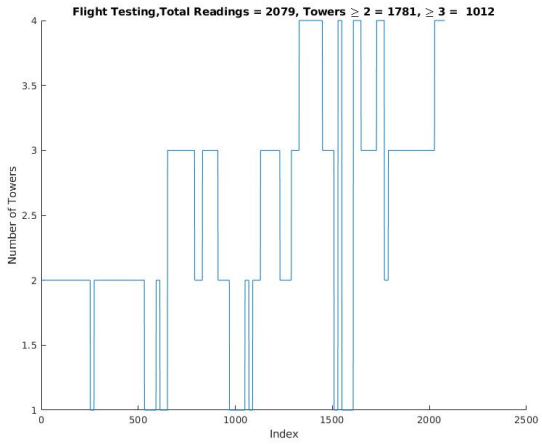


Figure 7.19: Carnation Flight Cells Read

The following plots show the sensor model fits, the deltas between fit_i and truth, and the mean absolute error (MAE) for both ranging and fit:

Note: $\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|error_i| \quad \text{or} \quad \frac{||error||_1}{n}$

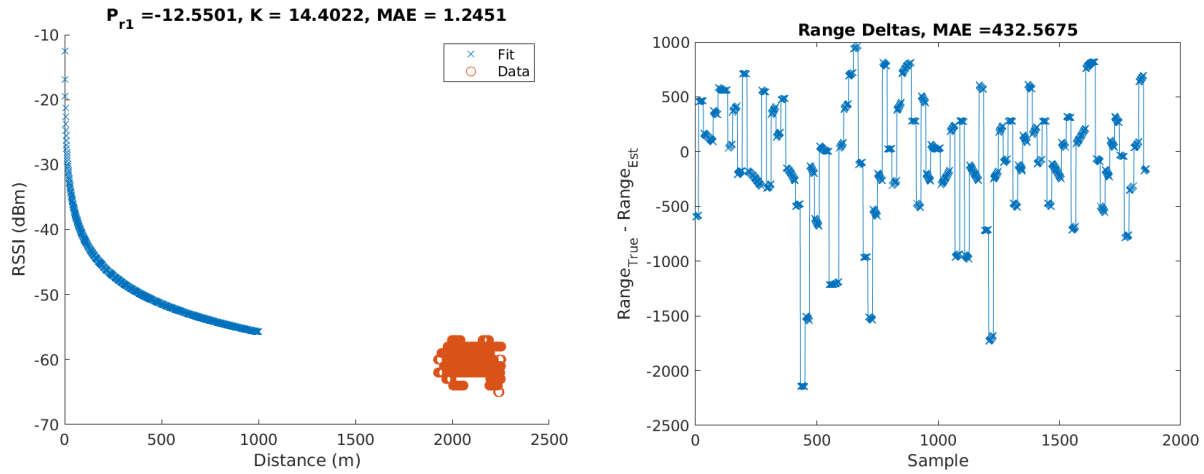


Figure 7.20: Flight Cell 1 Data

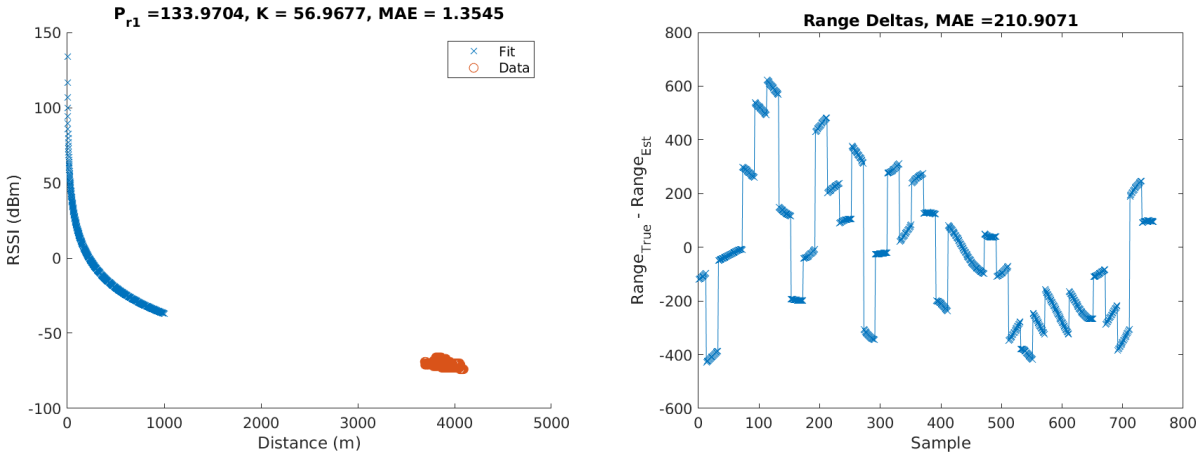


Figure 7.21: Flight Cell 2 Data

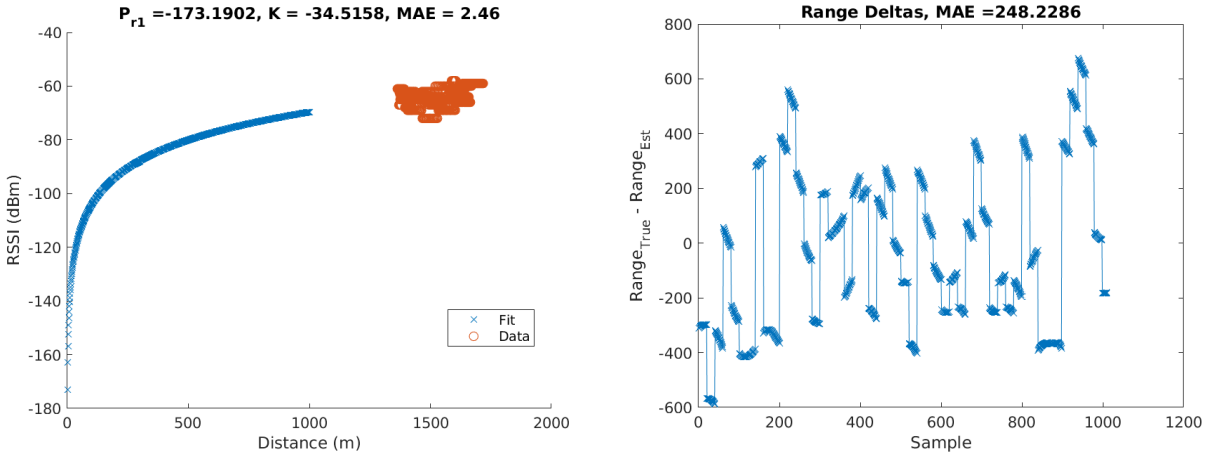


Figure 7.22: Flight Cell 3 Data

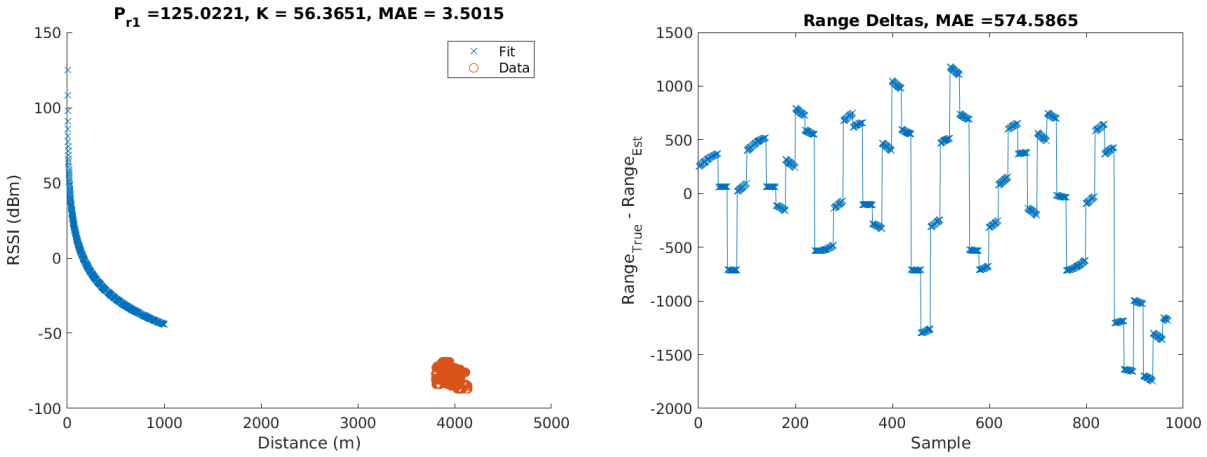


Figure 7.23: Flight Cell 4 Data

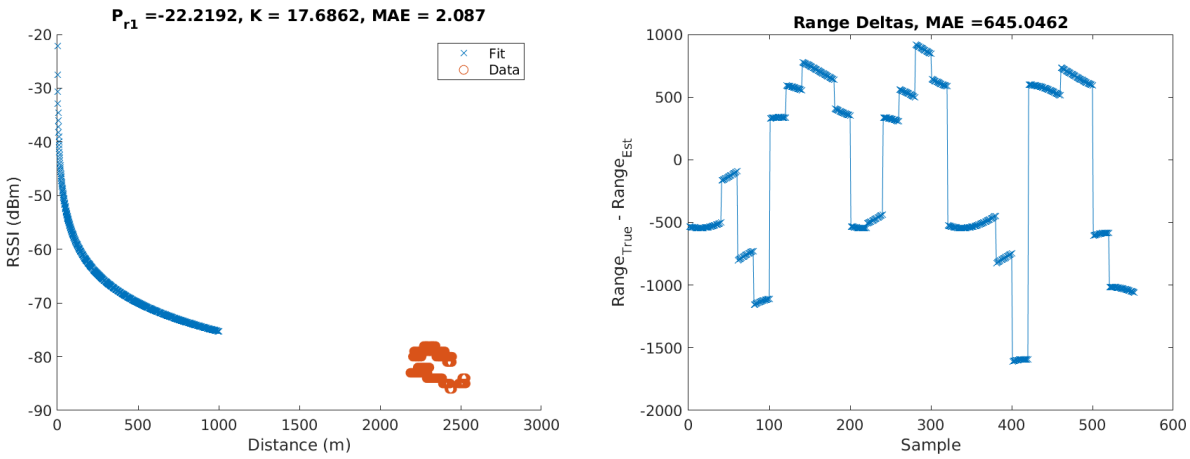


Figure 7.24: Flight Cell 5 Data

First off it's fairly obvious that the deltas show that something is not right here. Second, some of the fits look physically wrong. Cells 2 and 4 have positive P_{r_1} and cell 3 is gaining signal with distance. Remember that P_{r_1} represents signal lost at 1m, so it can't be positive. Also signal *should* be lost with distance, not gained. So cell 3's fit does not represent reality. There are a couple sources of fit error here, some of which have already been discussed:

1. The approximate cell locations, confounded by the fact that the serving radius of Carnation cells is on the order of km.

2. The data is grouped and far away from the cell 'location', where large changes in distance have small changes in signal (based on the model)

3. RSSI is reported in integers and based on the fit a change of 1dBm can represent 100m

4. The weighting matrix used in NLLS fit is identity.

7.2.8 Ground Data

This data was gathered on the north Seattle route (see fig. 7.1). Starting with the 2D analytic estimate with truth and R = 10 noise:

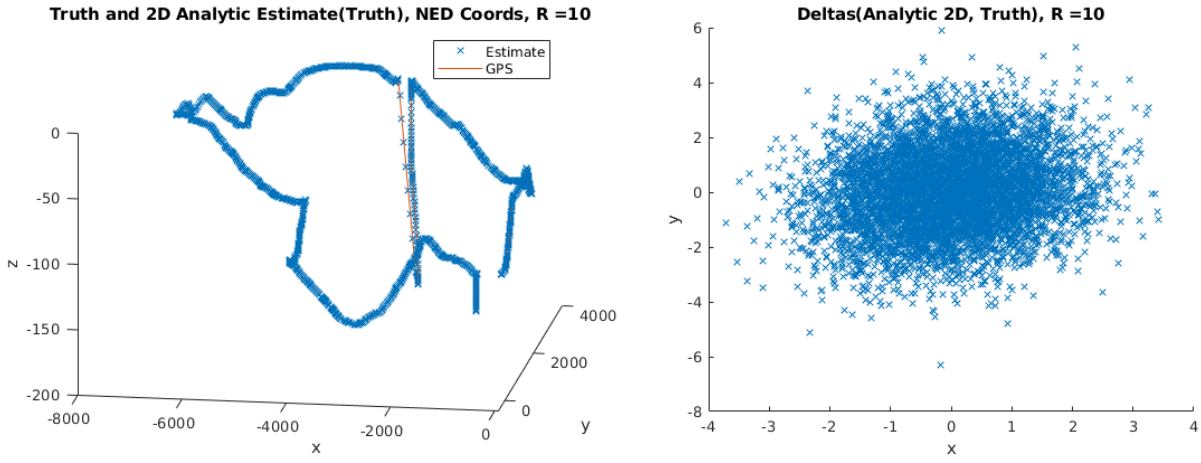


Figure 7.25: R = 10, Analytic 2D Position Estimation, Ground

The estimate from sensor model functions did not fair any better than the flight version:

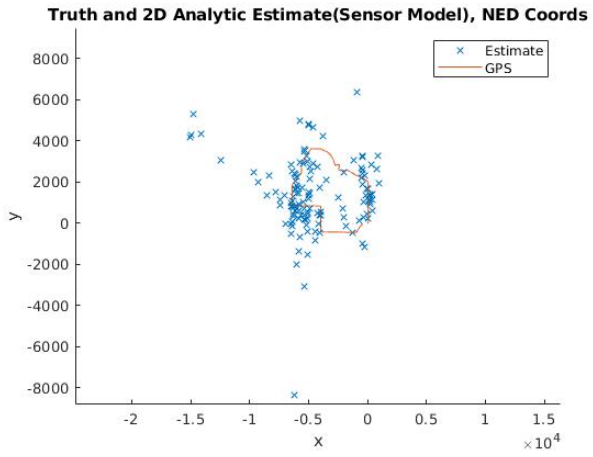
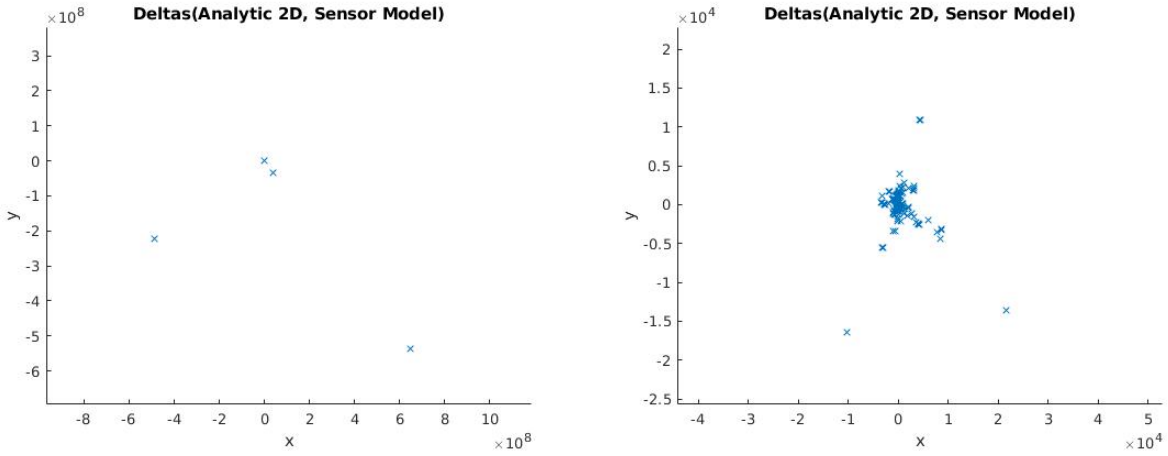


Figure 7.26: Analytic 2D Position Estimation, Ground

In fact some of the deltas were much further off (pay attention to the scale on the axis):



(a) Deltas Zoomed Out

(b) Deltas Zoomed In

Figure 7.27: 2D Deltas, Ground

This could be due to the fact that this data was gathered in an urban area, on the ground, and with a lot more cells read (50 Total). What this means is that there is a lot more noise, obstacles and variability in the cell output power.

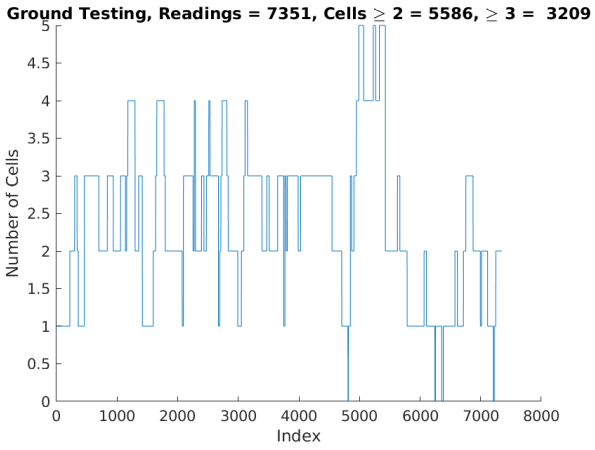


Figure 7.28: North Seattle Ground Cells Read

One interesting thing that came to light is that some of the cells had data that fit the model very well and had accurate ranges. On the other hand some were a lot worse than the flight cells. Below is a plot of the mean absolute error of the deltas per cell:

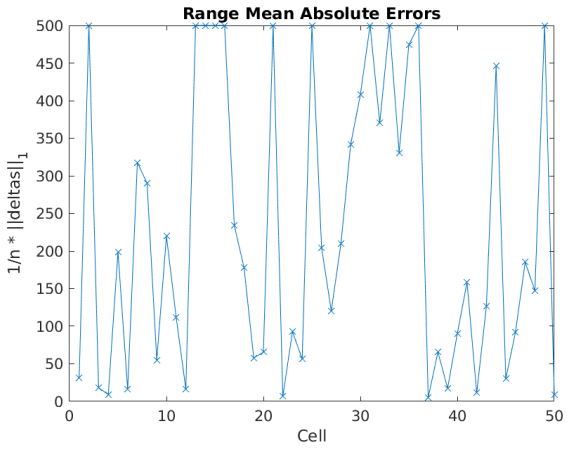


Figure 7.29: North Seattle Ground Range MAE

Note: The largest range MAE is not 500. Some of them are so far off that the shape of the plot is lost trying to capture them, so a cap was put at 500.

As far as cells that fit well, a total of 9 of the 50 had a range MAE below 20m. 11 had a range MAE over 500. So 20% were fairly good and 20% were really bad. In general the actual shape of the fit had little to do with the accuracy of the ranging. Some of the best cells had inverted fits or had $P_{r_1} > 0$. The thing that the good fits had in common is that the fit MAE was low (relative to axis span). Below are several of the good cells:

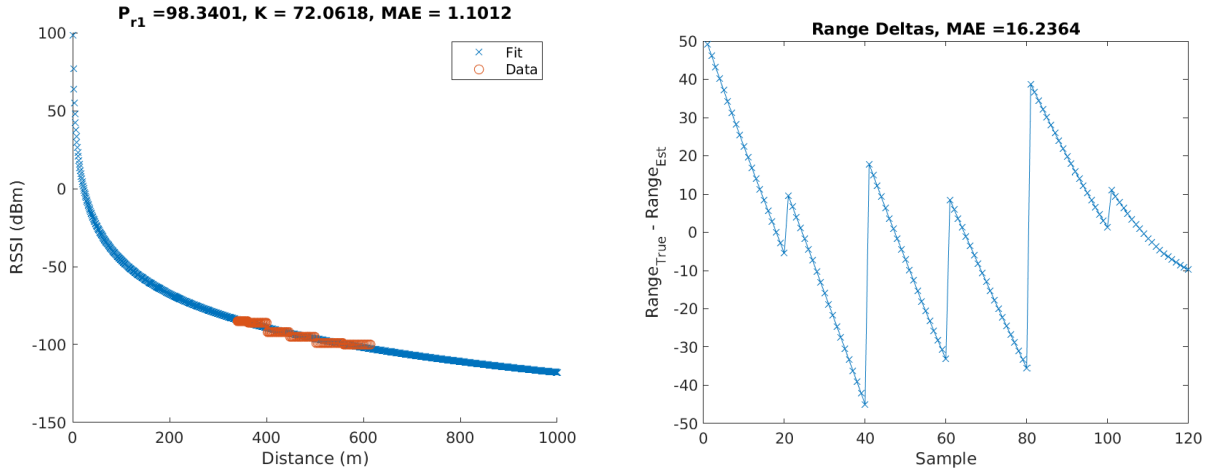


Figure 7.30: Ground Cell 6 Data

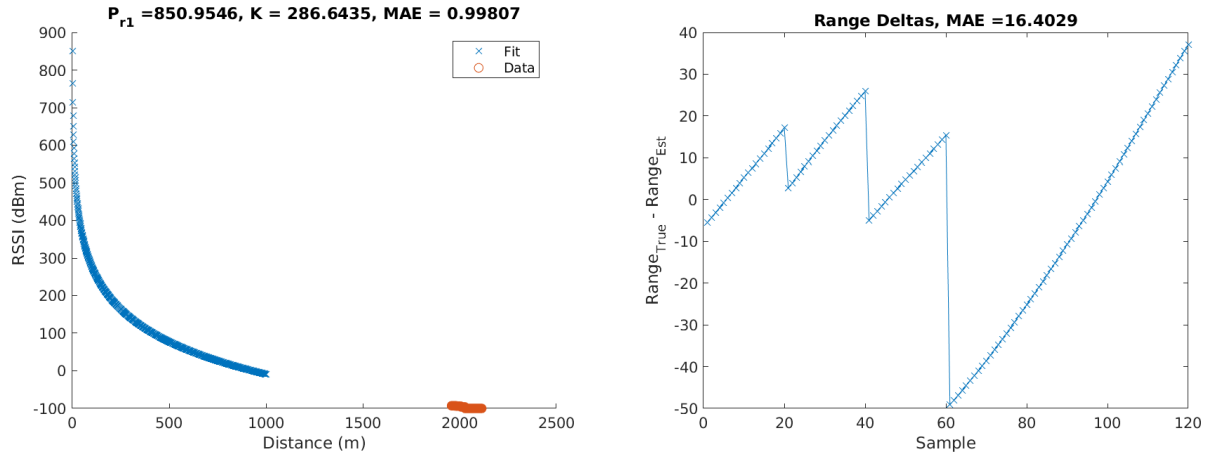


Figure 7.31: Ground Cell 12 Data

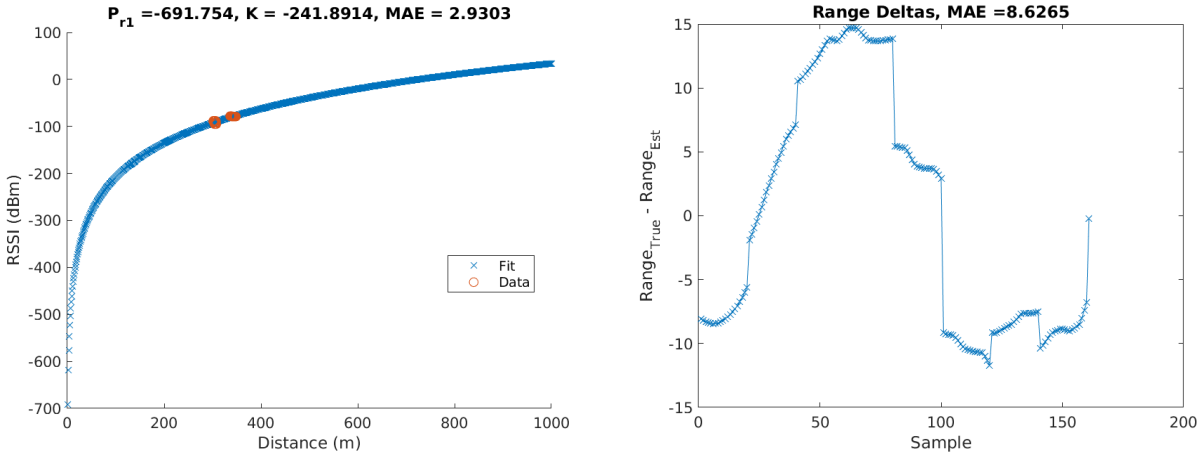


Figure 7.32: Ground Cell 50 Data

As one might guess, the cells that results in very bad estimates had a fit MAE that was fairly high:

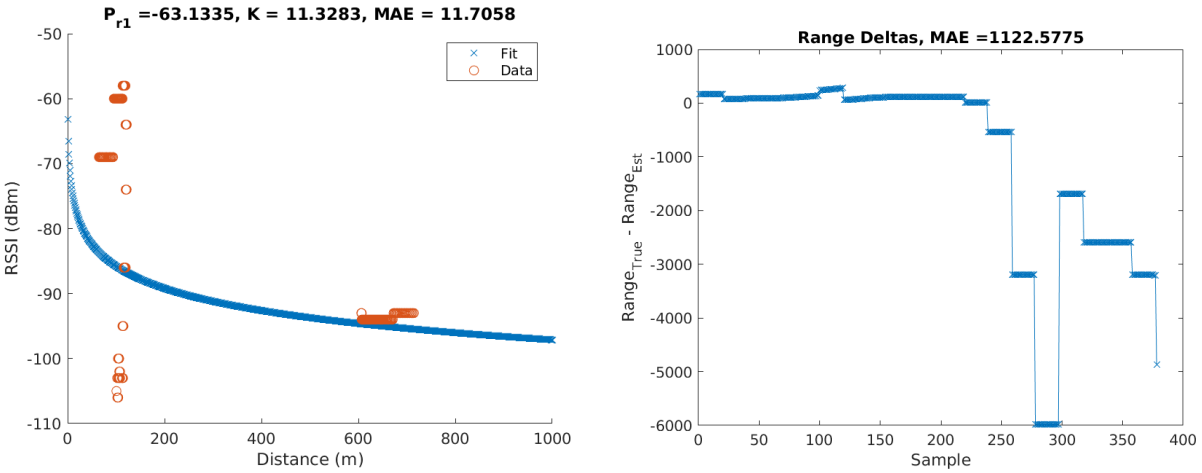


Figure 7.33: Ground Cell 2 Data

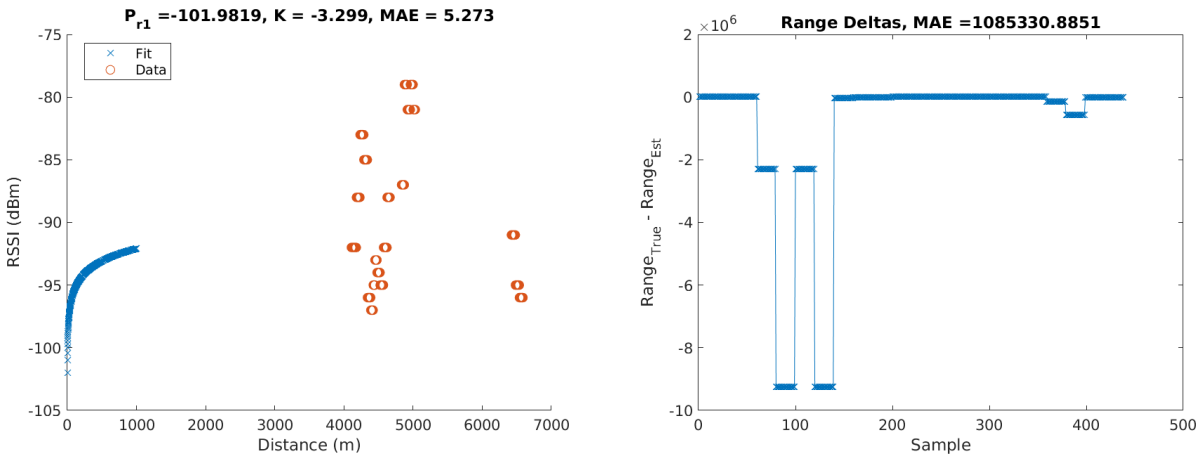


Figure 7.34: Ground Cell 13 Data

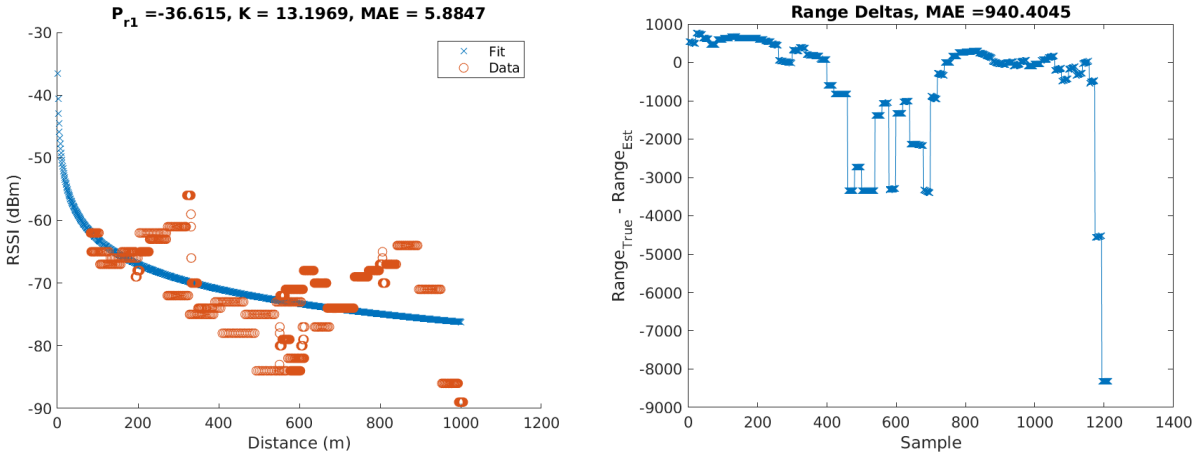


Figure 7.35: Ground Cell 31 Data

The sources of error here are the same as the flight data, with two differences:

1. The serving radius on the majority of the cells here is less than 1km. If the location given is the center of the serving radius of the cell (as is suspected), then the approximate location on any cell that has a smaller radius is better.

2. This data was gathered in an area with many sources of signal noise and obstacles. This can attenuate the signal received by the modem and introduce additional errors.

7.2.9 Method Conclusions

This method in is not a viable solution for GPS-Denied/Degraded Position Estimation. If more accurate methods become available the software and hardware structure is in place to support them. Currently though there are many sources of error and an a priori knowledge about cell signal characteristics requirement.

It did provide a valuable stress test to the system and some good data about cellular signals in the ground and air. In general:

- There are usually enough towers for multilateration.
- RSSI did not improve significantly while in the air (at the altitudes we flew at).
- Gathering data in urban areas where noise and obstacles exist has a large impact on signal quality.

7.3 Solution 2: FONA, ATCIPGSMLOC

7.3.1 Overview

This method seemed promising in that it is documented to be an AT command that returns a latitude and longitude of a cellular modem[42]. While it does do that, there are some severe limitations that will be shown shortly.

The hardware used was:

- Raspberry Pi 4
- VK-162 (For truth)
- Adafruit FONA (with SIM800 cellular modem and 1200mAh battery)

- Charmast USB-C Power Bank

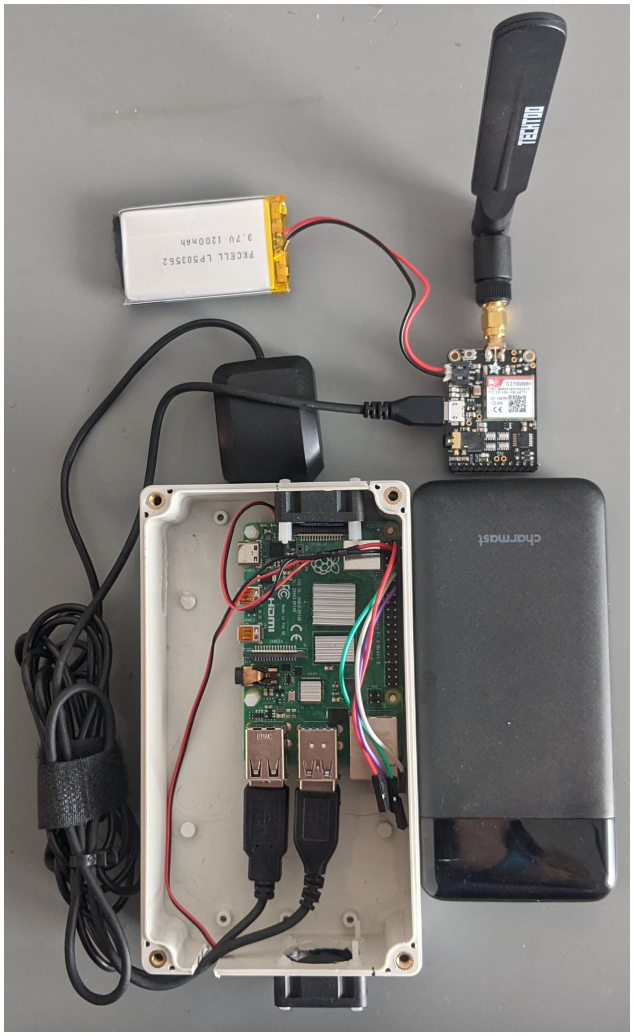


Figure 7.36: FONA Solution Hardware

The software modules used in this solution were:

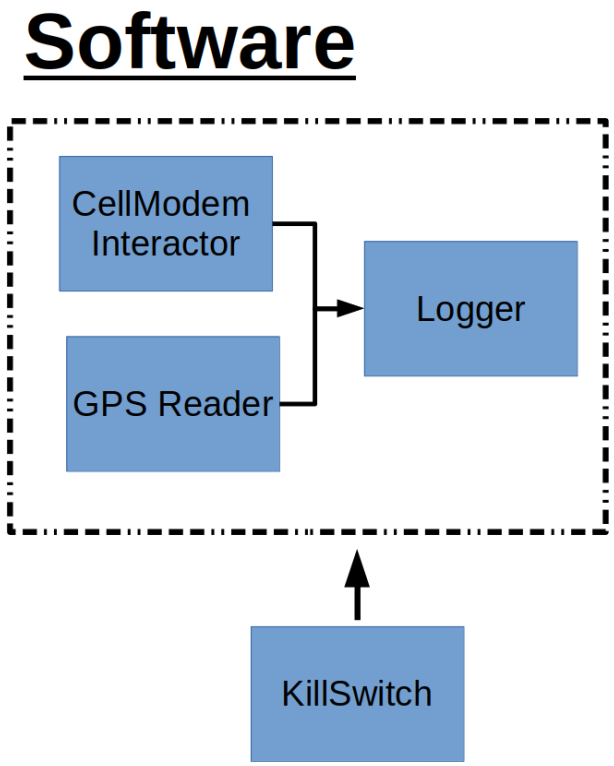


Figure 7.37: FONA Solution Software

These are the same modules used in the previous section, except for that the CellModem-Interactor is configured to use a FONA controller instead of a Telit controller. It issues the AT#CIPGSMLOC command and publishes the response. GPS reader, KillSwitch, and logger work the same as in the ATMONI solution.

There are no unknowns here to solve for, this solution is just a hardware/software integration problem. GPS and Fona data were gathered, logged, and reduced after ground and flight testing.

7.3.2 Ground Data

In order to show what the issue with the Fona is it is easier to start with the ground testing results. The data was gathered in the same north Seattle route as ATMONI (see fig. 7.1).

See the initial results below:

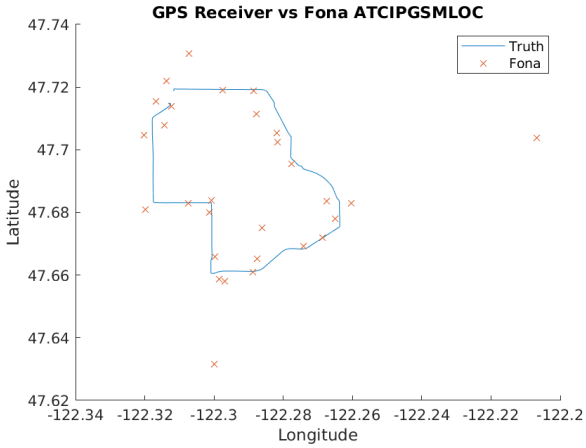


Figure 7.38: FONA Initial Results, Ground

Initially it doesn't look that bad. The points are mostly where they should be. The deltas (calculated via the Vincenty formula) do have several spikes but the they are nowhere near as far off as ATMONI:

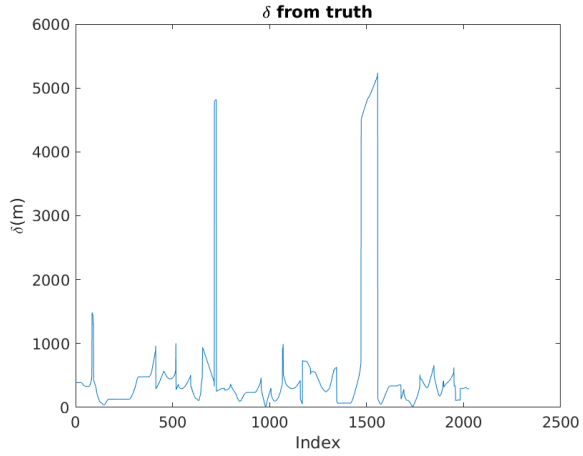


Figure 7.39: FONA Deltas, Ground

The *problem* is the number of points in the index of the fig. 7.39 vs the number of

points in fig. 7.38. For a large portion of the time the estimate is static, while the vehicle is actually moving. Below is fig. 7.38 with lines pointing from the estimate to the actual location (interpolated GPS):

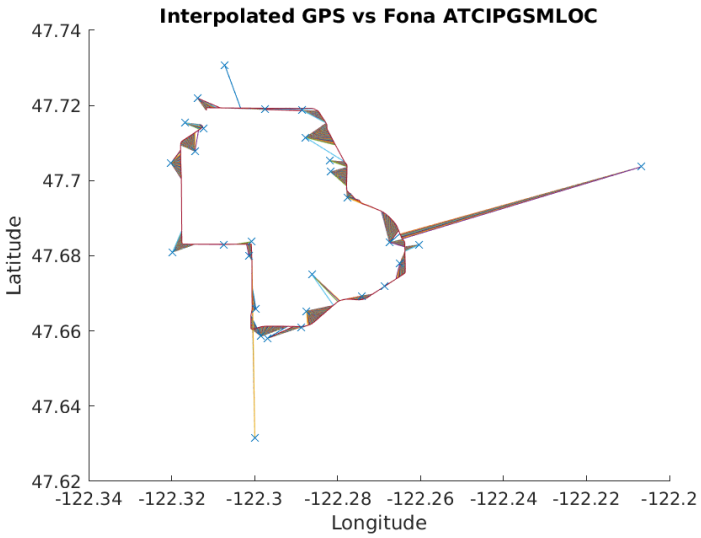


Figure 7.40: Interpolated GPS vs Fona, Ground

Now what it looks like is that this method returns the serving cell location. The step taken to check this was to look at the point reported and plot all the serving cell locations within some distance (called $cellFilter$ in the code) from the database created for ATMONI and overlay them on fig. 7.40. The serving cell locations are represented as small circles and serving radii are larger circles surrounding the cell locations. Setting $cellFilter = 20$ produces the following:

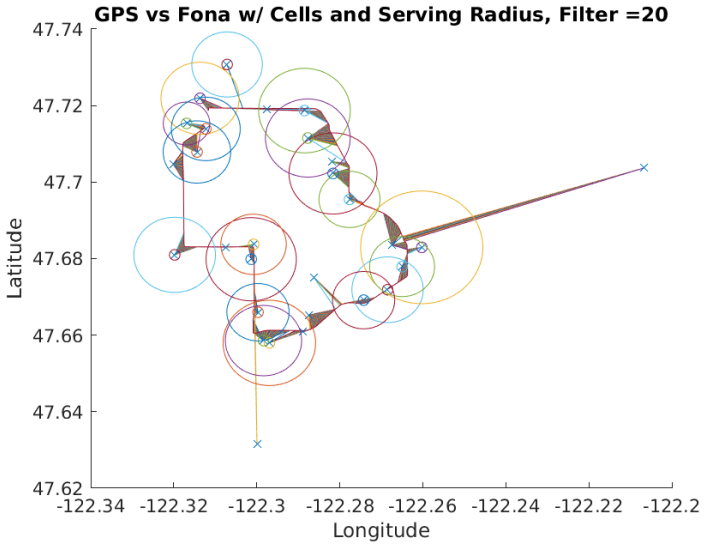


Figure 7.41: Estimate with Cell Locations and Serving Radii, Ground, 20m Filter

Setting it to 75 produces:

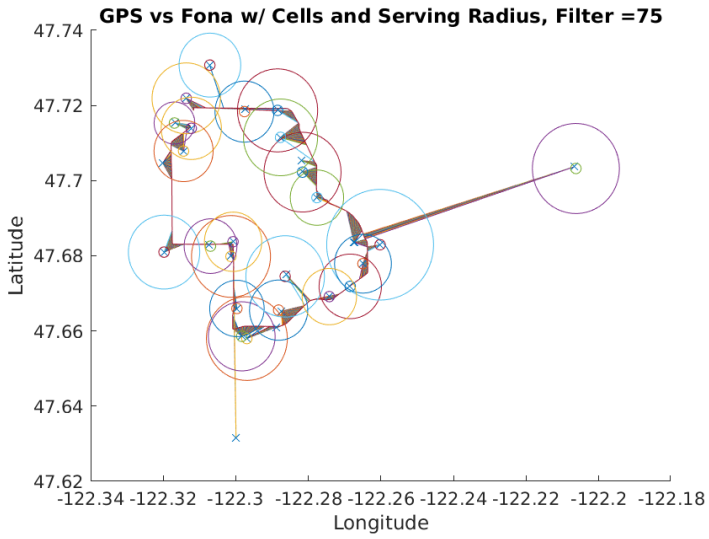


Figure 7.42: Estimate with Cell Locations and Serving Radii, Ground, 75m Filter

Setting it to 200 captures all but 2 of the cells:

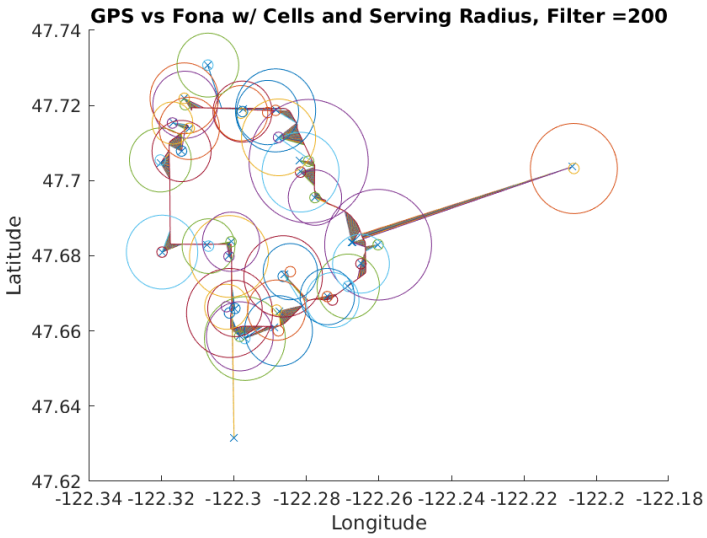


Figure 7.43: Estimate with Cell Locations and Serving Radii, Ground, 200m Filter

So it is a fairly reasonable conclusion to say that this method returns the approximate cell location and that location updates whenever the modem has a new serving cell. Most of 'x's are within 20m of a cell location from the earlier database, all but 5 are within 75m, and all but 2 are within 200m. An interesting point here is that this is an IP based method, in that you have to have internet service to use it (see [42, p.66] for an example call) which means that Adafruit either has their own database of cell locations or is offloading the estimate to a service like Google Geolocation API. The cell locations used in the ATMONI database are from Google Geolocation API so either the Fona uses old data or another service/database.

7.3.3 Flight Data

The flight data is much less clear as to what the problem is. There are large deltas between where the fona reports the cell location and where the database used in ATMONI does. Below are the initial results:

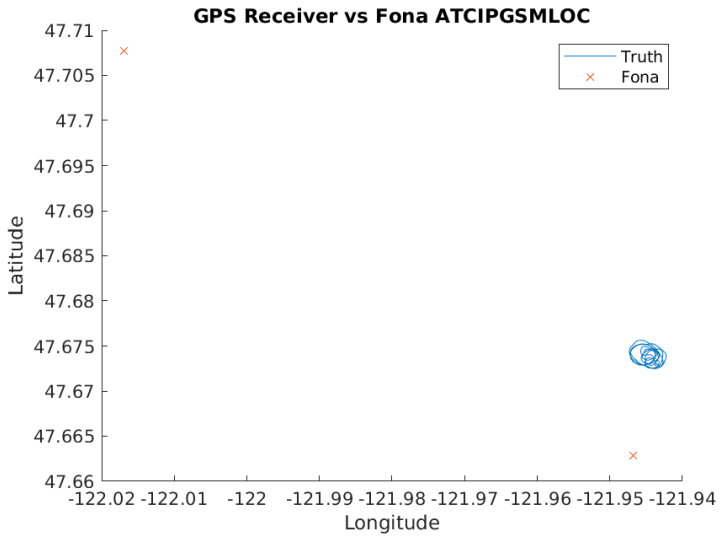


Figure 7.44: FONA Initial Results, Flight

And the deltas:

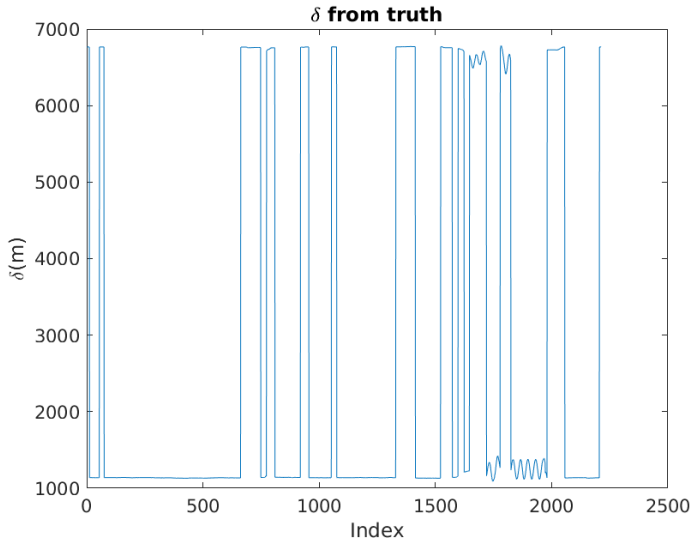


Figure 7.45: FONA Flight Deltas, Flight

One interesting thing here is you can see where the UAV is orbiting (points 1750 - 2000).

It has the same problem with a large number of points read vs small number of points plotted in the estimate. Now the interpolated data:

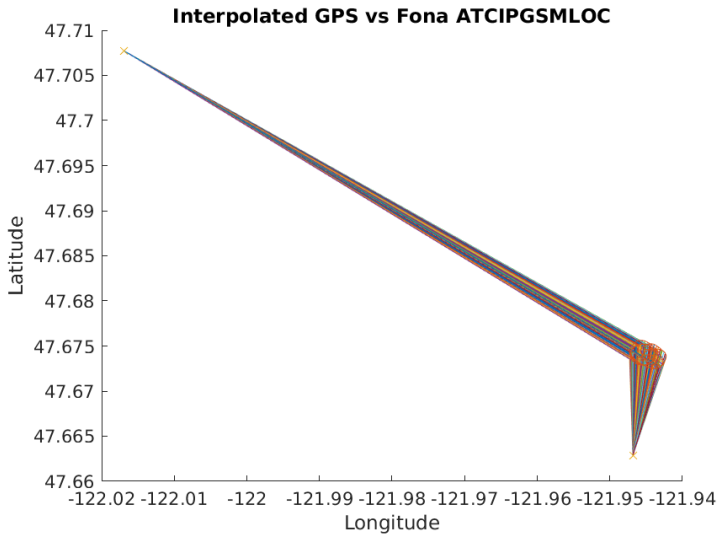


Figure 7.46: Interpolated GPS vs Fona, Flight

Cell locations don't start getting plotted until the filter is opened up to 500m, and the capturing both requires opening the filter to 2000:

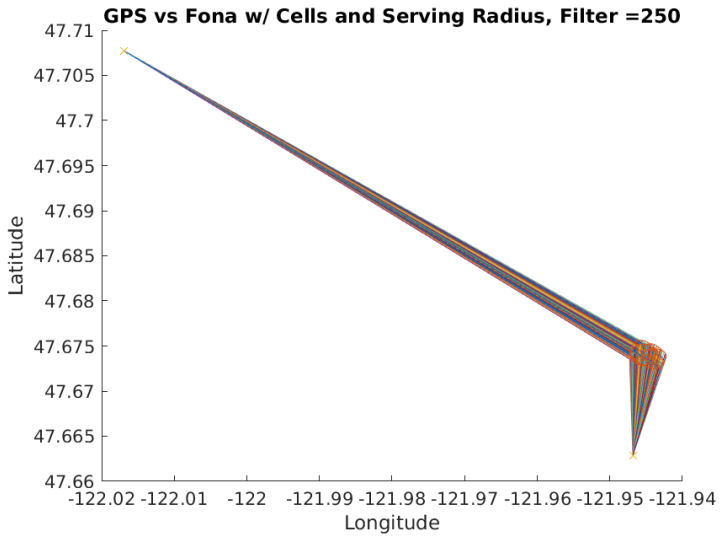


Figure 7.47: Estimate with Cell Locations and Serving Radii, Flight, 250m Filter

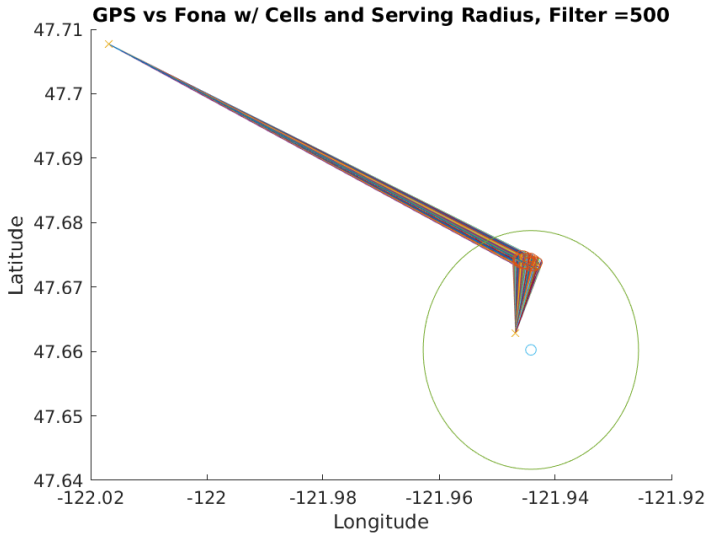


Figure 7.48: Estimate with Cell Locations and Serving Radii, Flight, 500m Filter

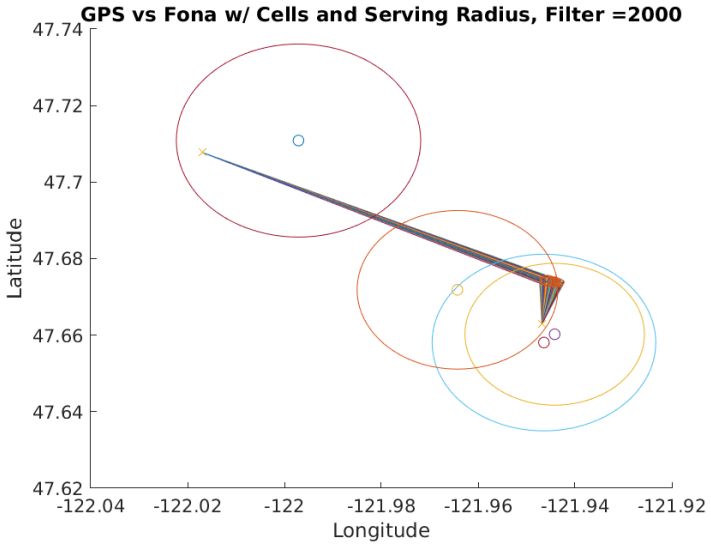


Figure 7.49: Estimate with Cell Locations and Serving Radii, Flight, 2000m Filter

Based on the ground data, the assumption here is that the two databases are just very far off. With the serving radii being as large as they are it would not be a surprising for the approximate locations here to be worse.

7.3.4 Method Conclusions

If this solution was used for a fast moving vehicle, moving through a cell dense area, where all the cells had fairly small serving radii, it might be useful. Outside of that very narrow band, this method is not a viable solution either. It is useful for rough approximations, gathered sporadically. The smaller the average cell radius the better the estimate. It was much simpler to implement and required nothing outside of an AT command, but those are basically the only high points.

7.4 Solution 3: WiCe-Nav

7.4.1 Overview

Out of the three solutions developed, WiCe-Nav is the most promising. It does have one issue, and it is latency. That will be covered in more detail shortly but overall it's the best bang for your buck when trying to use already built tools to estimate a position. WiCe-Nav is an extension of ATMONI in a sense. It gathers local cellular network signal data as well as local WiFi signal data, packages that data in a JSON string, and sends it off to Google's Geolocation API [14]. An example call (taken from the from the Geolocation Developer page) is below:

```
{
  "homeMobileCountryCode": 310,
  "homeMobileNetworkCode": 410,
  "radioType": "gsm",
  "carrier": "Vodafone",
  "considerIp": "true",
  "cellTowers": [
    // See the Cell Tower Objects section below.
  ],
  "wifiAccessPoints": [
    // See the WiFi Access Point Objects section below.
  ]
}
```

Figure 7.50: Geolocation JSON String Format

In order to accomplish this the following hardware was used:

- Raspberry Pi 4
- Telit HE910D (For local cell signals)
- Sierra MC7455 (For internet/Google call)
- Adafruit Feather Huzzah (WiFi scraper)

- VK-162
- Charmast USB-C Power Bank

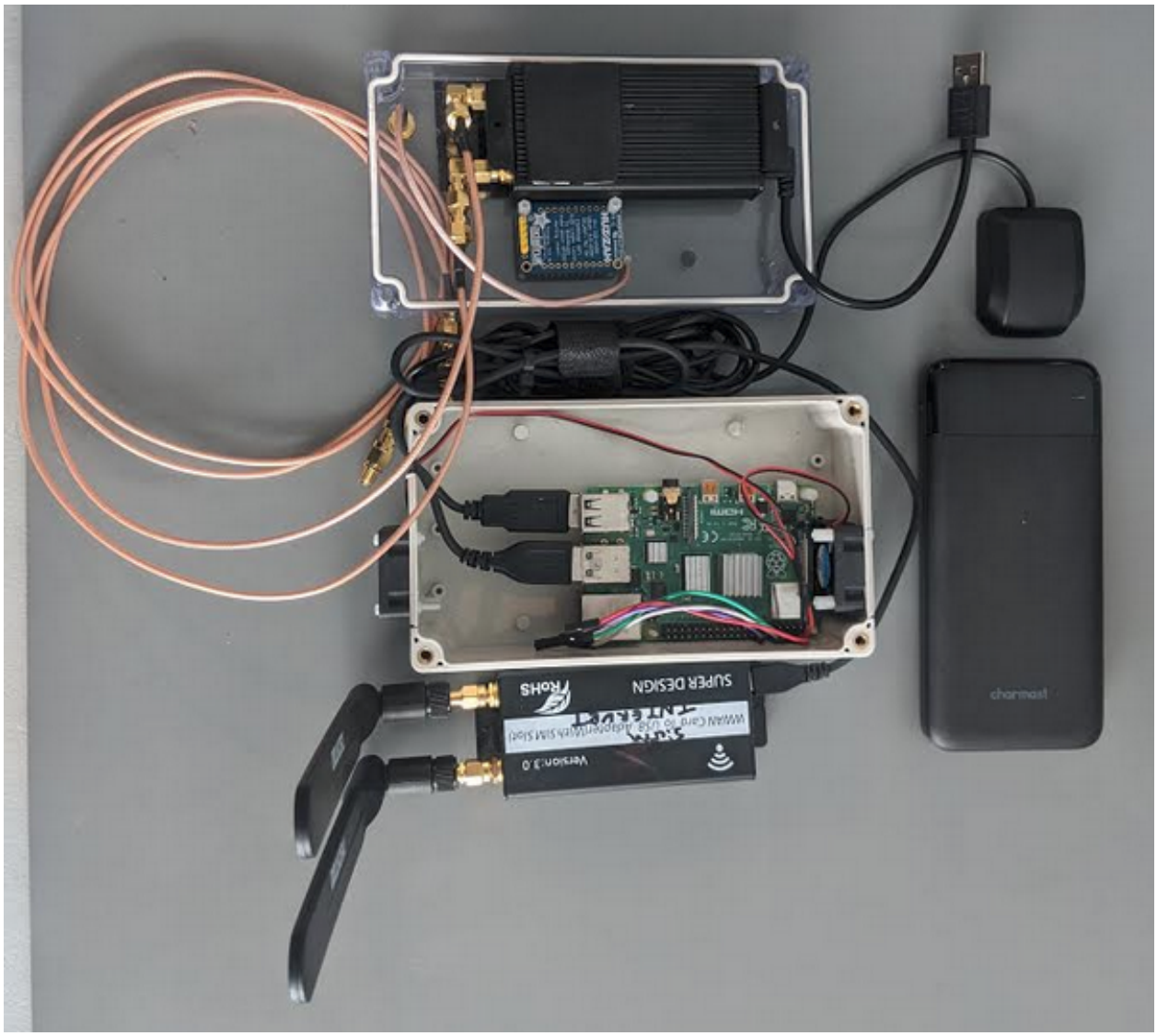


Figure 7.51: WiCe-Nav Hardware

The software modules used were:

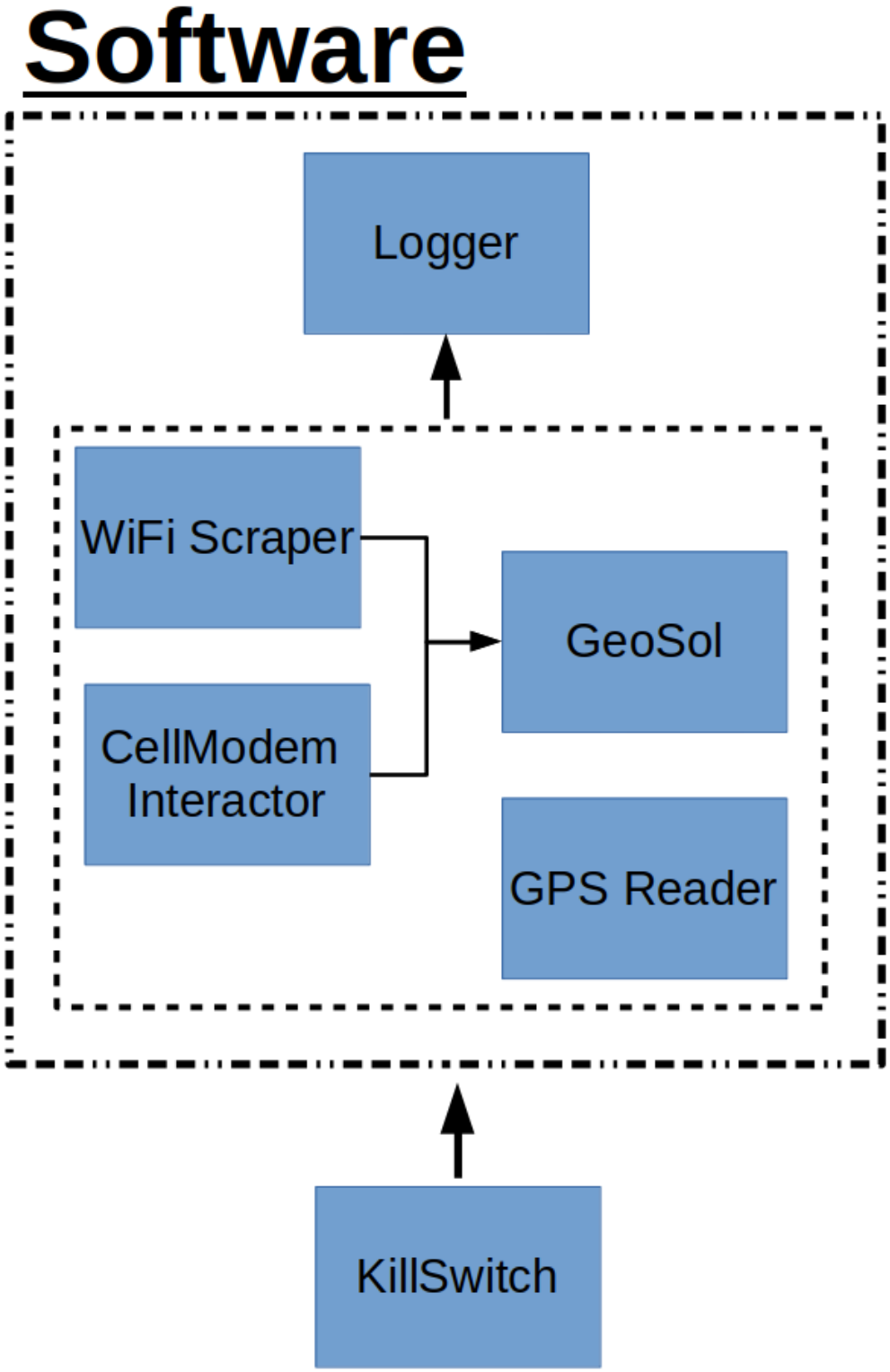


Figure 7.52: WiCe-Nav Software

The GeoSol and WiFi Scraper software modules are new here. The WiFi Scraper issues a command to the Huzzah to gather WiFi data, parses the response, and publishes the data. The GeoSol module subscribes to both the CellModem Interactor and the WiFi Scraper and sends the Google Geolocation API call when it has new WiFi data (*Note:* It is also set up to use only the newest data in the case of multiple updates). The reason it is triggered on WiFi data is two fold. The first is that the update rate is much lower. The cell modem publishes data at 1Hz while the WiFi Scraper is .3Hz. The second is range of WiFi networks is much smaller than cell towers, so the data is only good for a smaller window.

7.4.2 Software Time Note

Before moving on to the data, there should be some clarification about the ***parameter*** tick time, tick time ***data***, and ***publish*** time. In the software, the parameter tick time passed in via script is the lower bound for how long the main loop for each module can run. The top of the loop grabs a time stamp and compares that to the bottom of the loop. If $tickTime - (loopEnd - loopStart) = overHead > 0$, then the module sleeps for $overHead$. This is done for a couple of reasons. Generally the modules are simple, take very little time to run, and the rates don't need to be very high. So in order to reduce processor load you limit how much work each is doing by adding the lower bound. Also, constant update rates keep Δt constant and that makes any math with time stepping easier.

Tick time data refers to how low it takes for a module to do everything it is tasked to do. So where the parameter sets the lower bound, tick time data tells you how long it actually took. It is the quantity $(loopEnd - loopStart)$. This is a useful quantity to see how much overhead you have and is good for setting the lower bound to ensure that Δt *is* constant.

Publish time is gathered from the logs and is the measurement of Δt between actual published messages. Each message has a system time stamp so the publish time vector is gathered by taking the Δs between each of the message time stamps in the logs. This should match the tick time data if $overHead < 0$(lower bound exceeded) and should match the tick time parameter if $overHead > 0$(lower bound not exceeded). This is a way to verify outside

of the software that the lower bound is working and to see how well it works. *Note:* Tick time data is gathered via the logs if the module is configured to publish its tick time (via the command line parameters)

In summary:

- The tick time parameter sets the lower bound of the main loop.
- Tick time data shows how long the tick actually takes.
- Publish time shows Δt between published messages in the logs.

Also as an example, below is the tick time data and publish time of the Telit modem during ground testing of WiCe-Nav:

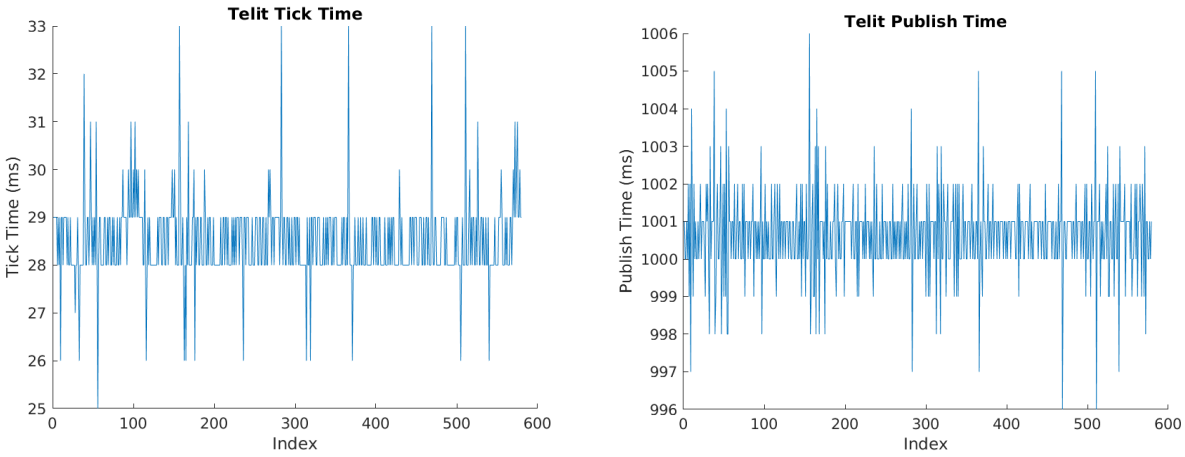


Figure 7.53: WiCe, Telit Tick and Publish Time

You can see that the tick time data shows an average tick time of ≈ 28.5 ms, the tick time parameter is set to 1 second, and the publish time matches the tick time parameter nicely.

7.4.3 Ground Data

The ground data was gathered in the south Seattle test area (see fig. 7.2). After processing the data (using GPS data as truth), it looks like the following:

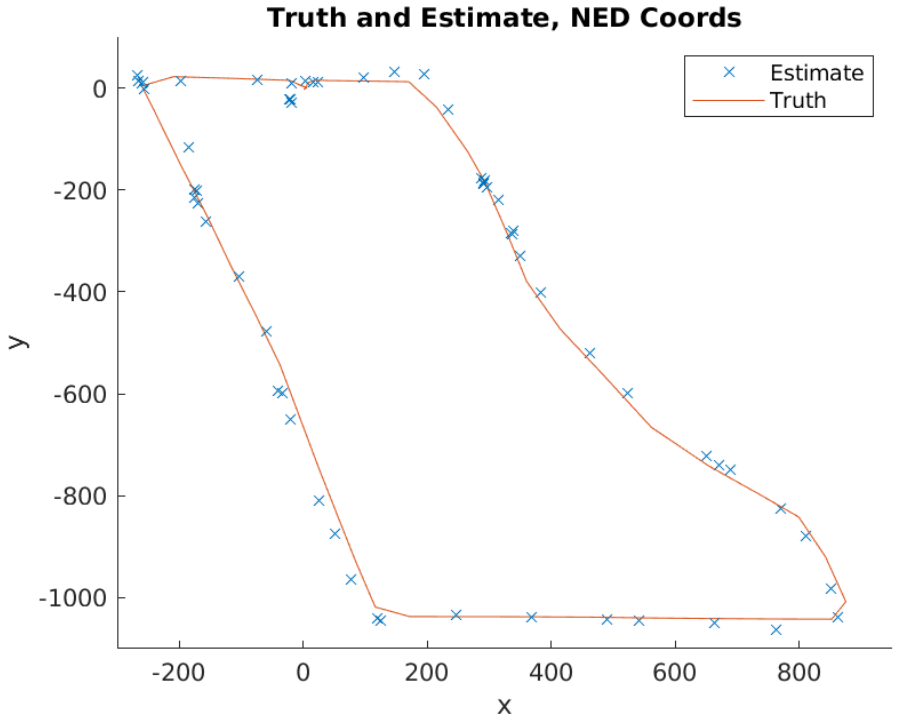


Figure 7.54: WiCe Estimate

Initially these are very promising results. The solution looks like it is updating and the deltas *look* small. The problem comes when you look at the interpolated truth data vs the estimate:

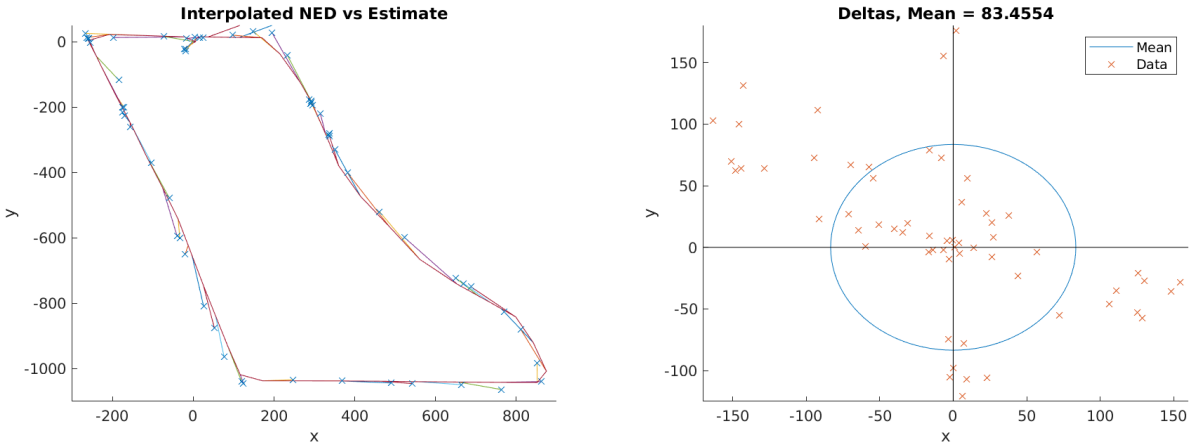


Figure 7.55: WiCe, Position Estimate vs Interpolated Truth

Note: The mean here is the vector version of MAE or average magnitude of the delta vector:

$$\frac{1}{n}\sum_{i=1}^{n}||delta_i||_2 \tag{7.2}$$

The direction of travel in this test is clockwise, so just based off the deltas there is some latency here. Now it is time to circle back to the new software modules and figure out where our problem(s) lie.

For testing the tick time parameter of both the WiFi Scraper and GeoSol was set to 1 second. In static testing both of these modules had a publish time greater than 1 second so this was a way to characterize what a reasonable tick time parameter *should* be.

Starting with the WiFi Scraper, it takes $\approx$ 3s to publish:

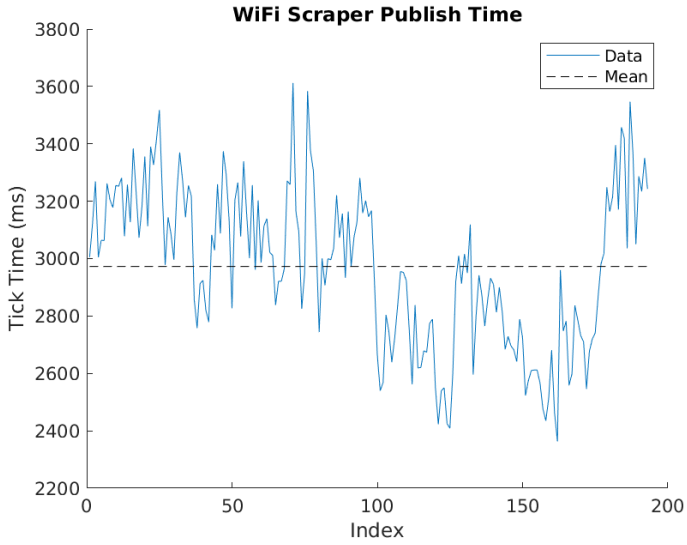


Figure 7.56: WiFi Scraper Publish Time

This is the first link in the latency chain. Now looking at GeoSol, the tick time is ≈ 8.35s:

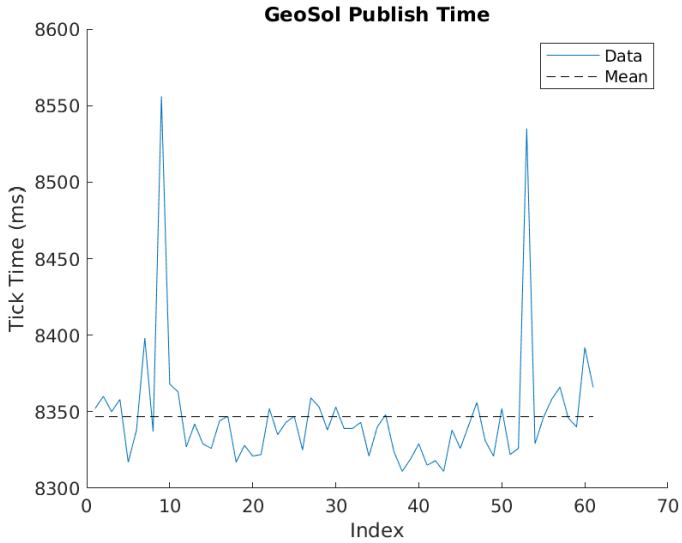


Figure 7.57: Geo Sol Publish Time

A couple of questions come up from the WiFi Scraper and GeoSol's publish time results.

First, what is the driver for these latencies? Second, what would the deltas be if they were reduced?

The latter is fairly easy to approximate. Given that the GeoSol only fires when it has new data, the Wifi data sitting at GeoSol is 0 to ≈ 3 seconds old. That WiFi data is between 0 and 3 seconds old when it is published. The GeoSol then takes ≈ 8.35 seconds to return a result. So the data is between ≈ 8 and 14 seconds old when it comes back to the system as a position estimate. To verify these bounds, the offset was swept from 8 to 14:

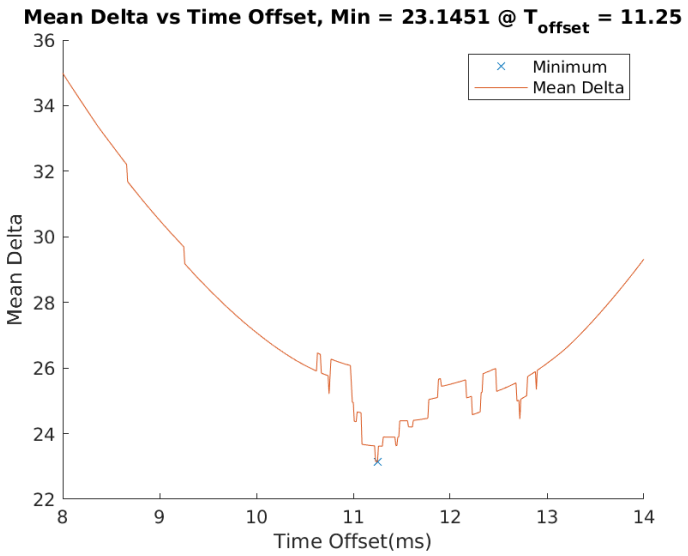


Figure 7.58: Position Estimate Mean Delta vs Time Offset

The minimum is at 11.25 seconds in the past, so it would be safe to say that on average the estimate is 11.25 seconds old. Looking at the data, it is probably also safe to say that most of the estimates are between 10.5 and 13 seconds old. Running the min offset through the data reduction produces the following:

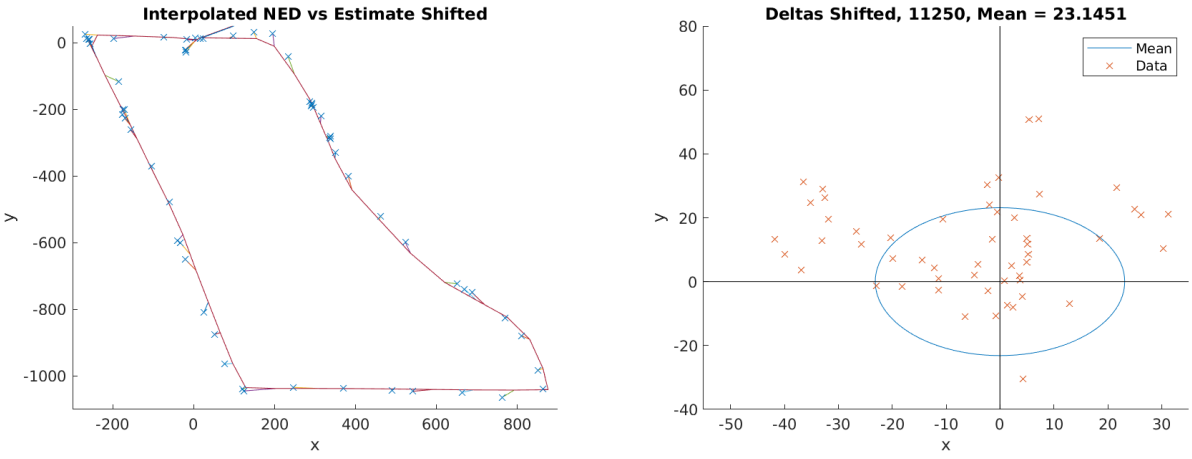


Figure 7.59: WiCe, Position Estimate vs 11.25s Shifted Interpolated Truth

That is not a bad solution. It updates and is fairly accurate. The next step is to figure out the drivers of these latencies. That is fairly straight forward to find, but less to fix.

In the case of the WiFi scraper, it's the scraper's firmware. 99% of the functionality lives in the firmware. Outside of opening a serial port and publishing data from the firmware call, this module does basically nothing. So to reduce the latency, a different call, different firmware (actually an option here), or a different module should be used.

The GeoSol is a little bit stickier. The source of the latency is the API call. The module main loop checks for new data and only fires if the WiFi data has been updated. Most of the main loop is waiting for the return of the API call. One interesting thing is that the latency is ***not*** variable. All of the publish time data points are within 10s of milliseconds of each other. So the question becomes: Is the source of the latency the time it takes to connect to the server that does the calculation? Or is it just how long that server takes to calculate the location? Without more data it is hard to tell one way or the other. It could be that test area is small enough that this is localized latency from poor reception. Testing in different areas with better (or worse) reception would provide more data and point to a

possible cause, and that could lead to a fix.

One thing that hasn't been mentioned is the update rate of this method. .125Hz (once every 8 seconds) is not something that will work with most autopilots. The usual update rate an autopilot expects for GPS data is ≥1Hz. So something needs to be done to upsample this estimate. Reducing the method latency would help but another way to output an updating sample is to incorporate an IMU. This will be further discussed in section 8.3.

7.4.4 Method Conclusions

Out of the three methods tested throughout this project, this one is the most promising. It is a stable, updating estimate. While the current version has limitations from latency and signal availability, the ideal version is fairly accurate. Reducing the tick time of the WiFi Scraper and GeoSol software modules would make this a viable solution for low altitude, WiFi dense areas.

Chapter 8
FUTURE WORK

8.1 Overview

There is a fair amount of work to still be done on this project for it to be a viable product/test bed for GPS degraded/denied solutions. The final aim for the project is for users to be able to unplug their GPS receiver and plug this payload into that connector. When they start the payload their autopilot thinks it's talking to a GPS receiver while the payload is creating GPS data from any source that can generate it. In order to do that there is still some work to do on the base functionality to support a general estimate input. At this point it is safe to say that some of that work is still unknown, just based on the fact that a full working example has not been produced. This section will cover the future work for the tested solutions and the project in general.

8.2 Tested Solutions

8.2.1 ATMONI

There are several things that could reinvigorate this method:

- Better cell location data
- A more accurate ranging method.
- Higher altitude data from an urban area.

The first two should help make the method more accurate in any condition, and the third would be an interesting way to see if the current version is useful in that type of area. Given its poor performance though, continued work here is not near the top of any list of priorities.

8.2.2 FONA

This method is an interesting case. It requires very little hardware and does provide an estimate. It would be useful as a seed for other methods and might be useful as a periodic truth update when used with an IMU based filter. That is purely speculative though, and again this method is probably finished where it is at.

8.2.3 WiCe

This method was found fairly late in this portion of the project life-cycle. Due to that this one is still a bit rough around the edges. One advantage is that this method still has several executable future work items. The first is to focus on reducing latency of the WiFi Scraper. That is something that can be implemented and tested right away. The Huzzah Overview page has several example calls and there are even options to create your own custom firmware.

Another possible option here is to gather WiFi data from another source. The Huzzah is not the only option, but it is one of the most documented. The Raspberry Pi has a WiFi module so it *should* be possible to get WiFi data directly from the payload computer. That option should be avoided though, as it ties the solution to the Pi. An external WiFi module is the most general option and should be prioritized. All the being said, getting the tick time data down below a second (ideally lower) is the goal here. Any method that produces that result is a step in the right direction.

The issue of the GeoLocation API call latency requires more testing in various areas of various cell signal coverage. Once the source of the latency can be identified with some certainty (signal quality or estimate source), then the path forward is more clear. Testing in different areas doesn't require any additional development though, so it is the easiest step to take with this method.

8.3 Plug and Play Solution

Referring to fig. 6.1 the majority of the blocks can be condensed into an 'Estimate Source' block:

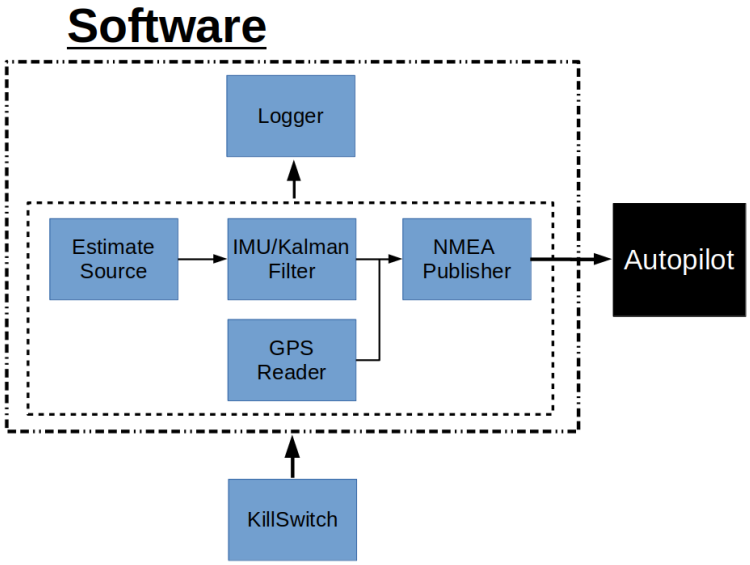


Figure 8.1: Abstracted Software Data Flow

This is the general idea. If GPS data exists, the NMEA publisher acts as a GPS bridge. Else the estimate source publishes data at some rate, the IMU/Kalman filter upsamples and filters the data, and the NMEA publisher outputs NMEA sentences to the autopilot. This is obviously easier said than done and is the main focus of the next section of the project lifecycle. It can be broken into two main sections though, namely the way it's already broken up in fig. 8.1:

- Choice of Filtering Algorithm
- Implementation of the NMEA Publisher

For this portion of the project an IMU was integrated to give access to that data for future work (see appendix A.2.5 for more on this). The filtering algorithm will most likely

be similar to the loosely coupled version discussed in [2, Chapter 7, Section 2.1], but that is beyond the scope of this document. A rough outline of the work to be done in the NMEA Publisher can be done here though.

8.3.1 NMEA Publisher

The software to take data and publish NMEA sentences is already integrated and available in the GPS Reader module. The methods and module to specifically do it needs to be designed and implemented. To that end there is some research that needs to be done on in a couple of key areas:

1. What is the latency of the GPS Bridge?

2. What is the minimum data set/rate that the autopilot firmware expects?

Note: There are other considerations from a hardware integration perspective but those are trivial in comparison to the two above.

The first question should be obvious why it's important. This payload is meant to add to, not subtract from, the operational area of the UAV. If introducing the GPS bridge creates undesirable effects, then those need to be addressed before moving forward. Fortunately this question can be answered by simply by doing it, and should be the first step in implementing the NMEA Publisher.

The second question will take some research into open source firmware. The first place should be the firmware on the test UAV. The things to be identified here are:

1. What NMEA sentences are expected?

2. What data do those contain?

3. What are the update rates/timeouts?

4. How can we calculate or approximate those values?

All four of these points define what the expected input to the NMEA publisher is. The first two are on the NMEA Publisher side and should define the members of that module. The third and fourth point feeds back to the Kalman filter and any estimate source planned to be used with this project. There will most likely be some re-factoring of the WiCe algorithm to accommodate the interface designed for the NMEA Publisher at this point. The NMEA Publisher *should not* do anything other than expect these quantities in a message once it picks the source i.e., it *should not* be addressing points three and four from above.

Finally, it should be noted that the NMEA Publisher needs to be able to switch between sources based on metrics that exist in the data. In order to not twist too many 'knobs' at once, this should be implemented last (but designed for early) i.e.:

1. Implement the GPS bridge and test latency and performance.

2. Implement the filtering algorithm and do the same.

3. Add the switch and compare deltas.

This is a way to bound possible sources of error and latency.

Chapter 9

CONCLUSIONS

9.1 Overview

The aim of this thesis was to lay out a mixed signal solution for UAV position estimation that used cellular and WiFi network data. Along the way data availability, current and past researchers, and the current state of COTS solutions of this type were covered. The software built to tackle this problem was covered and shown to be a useful starting point for multiple tested solutions as well as future applications in this problem space. The following sections will cover the highlights from each of these areas.

9.2 Data Availability

Initially, the specifications produced by 3GPP led us to believe that there would be easily available support for a multitude of location estimation methods. We assumed that with the right firmware on the right modem the estimate was a method call away. That isn't necessarily too far from the truth. The problem is that those methods are not exposed to the general public. So for now, simply being a subscriber to a network does not give you access to the methods described in the 3GPP specifications. Furthermore it is not entirely clear that the majority of them exist. In short, with respect to GPS independent location estimation methods:

- Some groups of people have to be able to access these methods because of FCC mandates.

- We as network subscribers are not in one of those groups.

- There is no clear documentation as to what those methods are and nothing was provided to us by our industry partner.

9.3 Current and Past Research

There are not a lot of researchers in this space. There aren't any trying to build solutions that require a network subscription. Zak (Zaher) Kassas is a very active contributor in the signals of opportunity space. His work is adjacent to this space and uses low level signal knowledge to back out ranging data.

Other contributors have done things in a couple of areas including:

- White papers on LTE positioning methods
- Improving positioning in NB-IoT systems
- Integrating UAVs into cellular networks

The white papers and the NB-IoT work have no concrete examples and mostly just restate what is in the 3GPP specifications while adding some of the mathematical foundations. The integration work is almost exclusively using cellular networks to extend the range of signals; either bouncing off towers to relay C2 and payload data or providing internet access to rural areas. So some of this work is in the space, but not necessarily useful.

9.4 Current COTS Hardware

This project requires several discrete pieces. Almost all of those pieces have very good options. There are several payload computer options that are powerful, light, have numerous IO options, and consume little power. Most types of sensors we would want to pull in have small, low power options. Supplying power via a small battery or power bank is very easy. The only thing that is lacking is a modem firmware with methods specifically targeted at this area. As far as we have found, it does not currently exist. There are methods available in smart phones through Android or IOS, but that ties the solution to those systems (unless

it's fed to this system as a remote estimate). There is also no clear indication of what data those methods use, and if they use GPS we would have another problem of trying to kill the GPS signal to test the 'GPS-independent' focus here.

9.5 Tested Solutions

The method conclusions sections for each method have more detail (section 7.2.9, section 7.3.4, section 7.4.4) but here are the high points:

ATMONI

- This method showed that in both testing areas there were enough cell towers for multilateration.
- At the altitudes flown there was no significant increase in cell signals or towers seen.
- This method in its current form is not a viable solution, but future work can be done using the framework created.

FONA

- This method returns the approximate cell location of the serving cell of the cellular modem.
- This method in its current form is not a viable solution, but could be used to seed other methods that require sporadic approximate locations.

WiCe

- This method returns a fairly accurate($\approx$ 23 meters MAE at low latency) updating estimate.

- This method has some serious latency issues (8 - 14 seconds) when gathering WiFi data and using the Google GeoLocation API.

- If the latency can be reduced for both the WiFi Scraper and the GeoSol modules, this method could be a viable GPS-independent solution (assuming WiFi data availability).

Overall, a COTS solution for using only cellular data does not currently exist. The hypothesis that being 'inside' the network would provide additional information was not accurate, at least for regular users (users not in the group that have access to location data, like emergency services). Using a mixed signal approach and Google's API can provide a GPS independent position estimate. This assumes the existence of WiFi data, and that is not guaranteed in rural areas or in higher altitudes than flown in testing. The software used to facilitate this project is the most promising development and should be extended for use in other scenarios/signal sets. For more information about the software see the following appendix appendix A.1 and the software gitlab page.

BIBLIOGRAPHY

[1] 3GPP, "The mobile broadband standard," Apr 2020. Available at https://www.3gpp.org/about-3gpp/about-3gpp.

[2] J. L. Crassidis and J. L. Junkins, *Optimal estimation of dynamic systems*. Chapman and HallCRC, 2012.

[3] 3GPP, "Functional stage 2 description of location services," Dec 2019. Available at https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=834.

[4] 3GPP, "Feasibility study for evolved universal terrestrial radio access (utra) and universal terrestrial radio access network," Jul 2018. Available at https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=1341.

[5] 3GPP, "Evolved universal terrestrial radio access (e-utra) lte positioning protocol (lpp)," Dec 2019. Available at https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=2441.

[6] O. M. Alliance, "Oma specworks," Mar 2020. Available at https://www.omaspecworks.org/.

[7] L. Wirola, "Expert advice: Positioning protocol for next-gen cell phones," Mar 2015. Available at https://www.gpsworld.com/wirelessexpert-advice-positioning-protocol-next-gen-cell-phones-11125/.

[8] 3GPP, "Stage 2 functional specification of user equipment (ue) positioning in e-utran," Jun 2019. Available at https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=2433.

[9] Cisco, "Assisted gps (a-gps) overview," Feb 2018. Available at https://www.cisco.com/c/en/us/td/docs/routers/access/800/829/software/gps/Assisted-gps.html.

[10] Spirent, "An overview of lte positioning," Feb 2012. Available at https://www.spirent.com/~/media/whitepapers/mobile/lte_lbs_white_paper_2012.pdf.

[11] S. Fischer, "Observed time difference of arrival (odtoa) positioning in 3gpp lte," Jun 2014. Available at https://www.qualcomm.com/media/documents/files/otdoa-positioning-in-3gpp-lte.pdf.

[12] "Request location updates:android developers," Apr 2020. Available at https://developer.android.com/training/location/request-updates.

[13] "Cllocationmanager," Apr 2020. Available at https://developer.apple.com/documentation/corelocation/cllocationmanager.

[14] "Developer guide: Geolocation api," Apr 2020. Available at https://developers.google.com/maps/documentation/geolocation/intro.

[15] E. Team, "All you wanted to know about at and gsm at commands," Jan 2019. Available at https://www.electronicsforu.com/resources/cool-stuff-misc/gsm-at-commands.

[16] "An introduction to libqmi," Apr 2014. Available at https://sigquit.wordpress.com/2012/08/20/an-introduction-to-libqmi/.

[17] "Qmi/gobi management in the kernel: qmi_wwan or gobinet?," Aug 2017. Available at https://sigquit.wordpress.com/2014/06/11/qmiwwan-or-gobinet/.

[18] R. Svitelskyi, "A gimbal-supported, mono camera, relative position measurement system of a visually distinct object for uav guidance," Master's thesis, University of Washington, Seattle, WA, June 2019.

[19] C. W. Lum, H. Rotta, R. Patel, H. Kuni, T. Patana-anake, J. Longhurst, and K. Chen, "Uas operation and navigation in gps-denied environments using multilateration of aviation transponders," in *Proceedings of the AIAA SciTech 2019 Forum*, (San Diego, CA), January 2019.

[20] A. C. Arbeit and C. W. Lum, "Unmanned vehicle searches," August 2014. US Patent 14/453,406.

[21] C. W. Lum, J. Vagners, and R. T. Rysdyk, "Search algorithm for teams of heterogeneous agents with coverage guarantees," *AIAA Journal of Aerospace Computing, Information, and Communication*, vol. 7, pp. 1–31, January 2010.

[22] C. W. Lum, J. Vagners, J. S. Jang, and J. Vian, "Partitioned searching and deconfliction: Analysis and flight tests," in *Proceedings of the 2010 American Control Conference*, (Baltimore, MD), June 2010.

[23] C. W. Lum, K. Gauksheim, T. Kosel, and T. McGeer, "Assessing and estimating risk of operating unmanned aerial systems in populated areas," in *Proceedings of the 2011 AIAA Aviation Technology, Integration, and Operations Conference*, (Virginia Beach, VA), September 2011.

[24] K. Ueunten, "Modeling aircraft position and conservatively calculating airspace violations for an autonomous collision awareness system for unmanned aerial systems," Master's thesis, University of Washington, Seattle, WA, March 2014.

[25] Z. M. Kassas, "Zaher kassas publications," Apr 2020. Available at http://kassas.eng.uci.edu/publications.html.

[26] "Aspin research," Apr 2020. http://aspin.eng.uci.edu/research.html.

[27] "Navigation with cellular signals," Apr 2020. Available at http://kassas.eng.uci.edu/papers/Kassas_Navigation_with_Cellular_Signals.pdf.

[28] "Aspin lab homepage," Apr 2020. Available at http://aspin.eng.uci.edu/index.html.

[29] "Matrix radio," Apr 2020. Available at http://aspin.eng.uci.edu/MATRIX.html.

[30] S. Hu, A. Berg, X. Li, and F. Rusek, "Improving the performance of otdoa based positioning in nb-iot systems," Sep 2017. Available at https://arxiv.org/pdf/1704.05350.pdf.

[31] P. Silva, V. Kaseva, and E. Lohan, "Wireless positioning in iot: A look at current and future trends," Jul 2018. Available at https://www.mdpi.com/1424-8220/18/8/2470/pdf.

[32] "Positioning with lte," Sep 2011. Available at https://www.sharetechnote.com/Docs/WP-LTE-positioning.pdf.

[33] A. Fotouhi, H. Qiang, M. Ding, M. Hassan, L. Giordano, A. Rodriguez, and J. Yuan, "Survey on uav cellular communications:practical aspects, standardization advancements,regulation, and security challenges," Mar 2019. Available at https://arxiv.org/pdf/1809.01752.pdf.

[34] M. Mozaffari, W. Saad, M. Bennis, Y.-H. Nam, and M. Debbah, "A tutorial on uavs for wireless networks:applications, challenges, and open problems," 2019. Available at https://ieeexplore.ieee.org/document/8660516.

[35] "Lte unmanned aircraft systems," May 2017. Available at https://www.qualcomm.com/media/documents/files/lte-unmanned-aircraft-systems-trial-report.pdf.

[36] M. Harris, "Nasa and verizon plan to monitor us drone network from phone towers," Jun 2015. Available at https://www.theguardian.com/technology/2015/jun/03/verizon-nasa-drones-cellphone-towers.

[37] "Linux serial ports using c/c++," Dec 2018. Available at https://blog.mbedded.ninja/programming/operating-systems/linux/linux-serial-ports-using-c-cpp/.

[38] "National marine electronics association," Dec 2018. Available at https://www.nmea.org/content/STANDARDS/NMEA_0183_Standard.

[39] H. Liu, Y. Zhang, X. Su, X. Li, and N. Xu, "Mobile localization based on received signal strength and pearsons correlation coefficient," *International Journal of Distributed Sensor Networks*, vol. 2015, pp. 1–10, 08 2015.

[40] "Telit 3g modules at commands reference guide," Oct 2016. Available at https://www.telit.com/wp-content/uploads/2017/09/Telit_3G_Modules_AT_Commands_Reference_Guide_r11.pdf.

[41] K. Heurtefeux and F. Valois, "Is rssi a good choice for localization in wireless sensor network?," in *2012 IEEE 26th International Conference on Advanced Information Networking and Applications*, pp. 732–739, 2012.

[42] "Sim800 series at command manual," Aug 2015. Available at https://www.elecrow.com/wiki/images/2/20/SIM800_Series_AT_Command_Manual_V1.09.pdf.

Appendix A

SOFTWARE MODULE I/O

A.1 Overview

This section will present the current state of the software modules used in this project. This software is built for Linux, it will not work on Windows. It is assumed that the reader knows how to create and run scripts in Linux, knows what udev rules are, and where to the default locations for lib and include are. A generic overview of the main loop will be presented where necessary. For the most part it will be assumed that users are interacting with the executable and just need to interface with the software, not develop it further. This section will be available on the software gitlab page and will be updated when appropriate.

Note: The only way to cleanly kill **any** of the modules here is to use the KillSwitch module. Pass close attention to the input arguments there.

A.2 Executables

A.2.1 ***KillSwitch***

Current Version = 0.1

Input

- None

Output

- Module shutdown message

Expected Form

- N/A

The KillSwitch sends out a message to turn off modules. During the main loop of every module (save this one), the kill message is checked for. Currently it is the only way to shut down a module cleanly. It turns on and pushes out the kill message 20 times, sleeping for 250ms in between each message. So roughly it comes up and runs for 5 seconds. Every module has an input parameter for the KillSwitch socket that needs to match the publish socket passed to the KillSwitch. Every module outputs a message when it shuts down so that is how you verify that the KillSwitch worked. It is meant to be run from a script, so if you don't see the shutdown message check your parameters and run it again. It is possible to set each module to have a different KillSwitch socket and have a separate KillSwitch for each module, or they all have the same KillSwitch socket and one script kills them all. Dealer's choice there.

- Example Start Call: ./../killswitch -tcp://*:6666;
- Input Parameters(Parameters **MUST** be prepended with '-' in the call):
  - pubAddress
    - Publish address for the kill message
    - Example: -tcp://*:5564.

      Note: local **publish** sockets start with *, local **subscribe** sockets start with localhost. This will be repeated, but you'll probably make this mistake.

A.2.2 ***CellModemInteractor***

Current Version = 0.1

Input

- None

Output

- Position estimate **or** raw tower data. See Data Mode below for more.

Expected Form

- Talks to a cellular modem connected via USB or serial port.

Overview

This module interacts with cellular modems, much like the name would suggest. AT commands are assumed which *may* take some additional steps in setup. All modems are different but they typically have a 'USB Setup' function documented somewhere in the AT command map document. Also most modems enumerate several ports when plugged in. If you have multiple devices you **must** have a udev rule to keep the port constant. See the software gitlab page for the rules used in this project. There are currently three supported modems:

- Adafruit Fona
- TelitHE910D
- SierraMC7455 (Will start, won't publish anything)

Note: You need an active SIM card for these to work.

There is also a special spoofer mode for the telit that uses the Logger output file to output the data again. The interactor has different output for each modem and the modem has three possible output modes that will be covered in the input parameters section below.

- Example start call: ./../interactor -/dev/TELIT910D00 -tcp://*:5564 -tcp://localhost:6666 -telitHE910D -../logs/Telit_Car_NSLongRoute.csv -1 -1 -1000 -tcp://localhost:5560 -tcp://localhost:5561
- Input Parameters(in order, **MUST** be prepended with '-' in the call):

1. portPath
   - Path to the serial port.
   - Example = -/dev/TELIT910D00
2. dataPubAddress
   - Socket to publish data to, 1024 to 65535 see this for more on that. This just needs to be declared and different from other modules.
   - Example = -tcp://*:5564.

     Note: local **publish** sockets start with *, local **subscribe** sockets start with localhost.
3. killSubAddress
   - Socket that the Killswitch sends out the Kill message.
   - Example = -tcp://localhost:6666.

     Note: local **publish** sockets start with *, local **subscribe** sockets start with localhost.
4. configType
   - Four possible options (exactly as typed):
     * telitHE910D
     * THE910DSpoof
     * sierraMC7455
     * fona800
   - Example = -telitHE910D

   Note: See logfile and data mode notes for each of these configs, they do not all publish/use the same data
5. logFile

- File used to replay a log. Populates a vector of data entries and publishes one each step through the main loop. If data runs out before the KillSwitch is used, the last entry is repeated indefinitely.
- Example = -../logs/Telit_Car_NSLongRoute.csv

Note: Only does something when THE910DSpoof config is used. All other configs it is just a base class member.

6. tickTimePub
   - Tells the module to publish the tick time data. 0 or 1. See section 7.4.2 for definition.
   - Example = -0
7. dataMode
   - Tells the interactor what data to publish. 0, 1, or 2.
     * 0: Position estimate only
     * 1: Raw data only
     * 2: Both position estimate and raw data

     Configuration Specific Outputs:
     * telitHE910D
       · 0: Publishes nothing.
       · 1: Publishes AT#MONI data.
       · 2: Publishes AT#MONI data.
     * THE910DSpoof
       · 0: Publishes nothing.
       · 1: Publishes AT#MONI data from log file.
       · 2: Publishes AT#MONI data from log file.
     * sierraMC7455

        - 0: Publishes nothing.
        - 1: Publishes nothing.
        - 2: Publishes nothing.
      - fona800
        - 0: Publishes AT+CIPGSMLOC data.
        - 1: Publishes nothing.
        - 2: Publishes AT+CIPGSMLOC data.

8. ticklength
   - Tells the module how long each tick should be. Expects an integer, milliseconds assumed.
   - Example = -1000
9. (Optional) subaddress_1, subaddress_2, ... subaddress_n
   - Additional subscription addresses. Currently none required, left in for future work.
   - Example = -tcp://localhost:5560 -tcp://localhost:5561
     Note: local **publish** sockets start with *, local **subscribe** sockets start with localhost.

A.2.3 ***GeoSol***

Current Version = 0.1

Input

- CellModemInteractor, WAPS

Output

- Position Estimate in the form of:

- Latitude
- Longitude
- Accuracy

See [14] for more.

Expected Form

- N/A

Overview

GeoSol is what combines the cell and WiFi data, turns it into a JSON string, and sends it off to the Google GeoLocation API. It requires a google geolocation developer key to use. It will only send out the API call when new WiFi data is present and if the cell data has been updated at least once.

- Example Start Call: ./../geosol -tcp://*:5571 -tcp://localhost:6666 -1 -1000 -310 -260 -AsgdhFT7eye72Swuw -tcp://localhost:5564 -DATA_PUB_TELITHE910D -tcp://localhost:5570 -DATA_PUB_WAPS

- Input Parameters(in order, parameters **MUST** be prepended with '-' in the call):

  1. dataPubAddress
     - Socket to publish data to, 1024 to 65535 see this for more on that. This just needs to be declared and different from other modules.
     - Example = -tcp://*:5571.

       Note: local **publish** sockets start with *, local **subscribe** sockets start with localhost.
  2. killSubAddress

- Socket that the Killswitch sends out the Kill message.
- Example = -tcp://localhost:6666.

  Note: local **publish** sockets start with *, local **subscribe** sockets start with localhost.

3. tickTimePub
   - Tells the module to publish the tick time data. 0 or 1. See section 7.4.2 for definition.
   - Example = -0
4. ticklength
   - Tells the module how long each tick should be. Expects an integer, milliseconds assumed.
   - Example = -1000
5. mcc
   - Mobile Country Code. Dependent on the network provider (of SIM card used in cell modem). Plenty of databases on Google for this.
   - Example = -310
6. mnc
   - Mobile Network Code. Should be local area code. Dependent on the network provider (of SIM card used in cell modem). Plenty of databases on Google for this.
   - Example = -260
7. APIKey
   - Developer API key from Google. There is a free level and as long ass you don't exceed the number of calls in the free zone this is a free service. This number was never exceeded during development and testing.

- Example = -Apthery7aerftgobhnnl8df

8. subAdd$_1$ subFlag$_1$ subAdd$_2$ subFlag$_2$ ... subAdd$_n$ subFlag$_n$
   - Subscribe Address/Data Flag pair. Current Data Flags are:
     * DATA_PUB_TELITHE910D(*)
     * DATA_PUB_SIERAMC7455
     * DATA_PUB_FONA800
     * DATA_PUB_WAPS(*)
     * DATA_PUB_GEOSOL
     * DATA_PUB_GPS

     Note: Only the starred ones are supported in the current GeoSol version. Only these two, and you need both.
   - Example = -tcp://localhost:5564 -DATA_PUB_TELITHE910D -tcp://localhost:5570 -DATA_PUB_WAPS

   Note: These **must** come in pairs. Not passing in pairs will cause the module to not start correctly.

A.2.4 ***GPS Receiver***

Current Version = 0.1

Input

- None

Output

- GPS Data

Expected Form

- Talks to a GPS receiver connected via USB or Serial Port.

Overview

This module reads data from a GPS receiver and parses the NMEA sentences. Outputs GPS Data that is self explanatory (i.e. each variable is labeled).

- Example Start Call: ./../gpsreceiver -/dev/GPS_UART -tcp://*:5565 -tcp://localhost:6666 -1 -10

- Input Parameters(in order, parameters **MUST** be prepended with '-' in the call):

  1. portPath
     - Path to the serial port.
     - Example = -/dev/GPS_UART
  2. dataPubAddress
     - Socket to publish data to, 1024 to 65535 see this for more on that. This just needs to be declared and different from other modules.
     - Example = -tcp://*:5565.
       Note: local **publish** sockets start with *, local **subscribe** sockets start with localhost.
  3. killSubAddress
     - Socket that the Killswitch sends out the Kill message.
     - Example = -tcp://localhost:6666.
       Note: local **publish** sockets start with *, local **subscribe** sockets start with localhost.
  4. tickTimePub
     - Tells the module to publish the tick time data. 0 or 1. See section 7.4.2 for definition.
     - Example = -0

5. ticklength
   - Tells the module how long each tick should be. Expects an integer, milliseconds assumed.
   - Example = -1000

A.2.5 ***Ozz***

Current Version = 0.1

Input

- None

Output

- Raw accelerometer(milli g), gryo(milli degrees/second), magnetometer(milli Gauss), pressure(Pa), and temperature(C) values

Expected Form

- Talks to an Ozzmaker BerryIMU v2 connected via i2c

Overview

This module gives high rate local sensor data. It provides x,y,z acceleration, gyro, and magnetometer data as well as temperature and pressure.

Note:The update rates of all of the sensors are different (on the actual hardware, not in the executable. This is a note to help you when setting your tick time via script). See below:

- Accelerometer: 476Hz
- Gyro: 476Hz

- Mag: 80Hz
- Temp: 83.33Hz
- Press: 83.33Hz
- Example start call: ./../ozz -/dev/i2c-1 -tcp://*:5569 -tcp://localhost:6666 -1 -0 -100 -tcp://localhost:5575
- Input Parameters(in order, **MUST** be prepended with '-' in the call):
  1. portPath
     - Path to the device.
     - Example = -/dev/i2c-1
  2. dataPubAddress
     - Socket to publish data to, 1024 to 65535 see this for more on that. This just needs to be declared and different from other modules.
     - Example = -tcp://*:5569.

       Note: local **publish** sockets start with *, local **subscribe** sockets start with localhost.
  3. killSubAddress
     - Socket that the Killswitch sends out the Kill message.
     - Example = -tcp://localhost:6666.

       Note: local **publish** sockets start with *, local **subscribe** sockets start with localhost.
  4. tickTimePub
     - Tells the module to publish the tick time data. 0 or 1. See section 7.4.2 for definition.

- Example = -0

5. dataMode
   - Tells the interactor what data to publish. 0, 1, or 2.
     * 0: Accel, Gyro, Mag, Temp, and Press published
     * 1: Accel, Gyro, and Mag published
     * 2: Temp and Press published
     * Example = -0
6. ticklength
   - Tells the module how long each tick should be. Expects an integer, milliseconds assumed.
   - Example = -100
7. (Optional) subaddress_1, subaddress_2, ... subaddress_n
   - Additional subscription addresses. Currently none required, left in for future work.
   - Example = -tcp://localhost:5560 -tcp://localhost:5561
     Note: local **publish** sockets start with *, local **subscribe** sockets start with localhost.

A.2.6 ***Logger***

Current Version = 0.1

Input

- None

Output

- None

Expected Form

- N/A

Overview

The logger logs data published by whatever modules its subscriber list has in it. This is a zero effort way to see what modules publish. Just add their socket to the subscription list. One difference in this module is that it has no tick length. The loop constantly runs and checks for new data.

- Example Start Call: ./../logger -DataLog -tcp://*:5561 -tcp://localhost:6666 -0 -tcp://localhost:5564 -tcp://localhost:5565 -tcp://localhost:5570 -tcp://localhost:5571

- Input Parameters(in order, parameters **MUST** be prepended with '-' in the call):

  1. logFileName
     - Name for the log file. Date and time the log started gets added to the end. File gets saved where you call the logger from.
     - Example = -DataLog
  2. dataPubAddress
     - Socket to publish data to, 1024 to 65535 see this for more on that. This just needs to be declared and different from other modules.
     - Example = -tcp://*:5561.

       Note: local **publish** sockets start with *, local **subscribe** sockets start with localhost.

     Note: This module does not publish anything. Pub socket left in for future work.
  3. killSubAddress
     - Socket that the Killswitch sends out the Kill message.

- Example = -tcp://localhost:6666.

  Note: local **publish** sockets start with *, local **subscribe** sockets start with localhost.

4. tickTimePub
   - Tells the module to publish the tick time data. 0 or 1. See section 7.4.2 for definition.
   - Example = -1
5. subaddress_1, subaddress_2, ... subaddress_n
   - Subscription addresses. If you want to see what a module publishes, this is the easiest way to do that.
   - Example = -tcp://localhost:5575 -tcp://localhost:5576

     Note: local **publish** sockets start with *, local **subscribe** sockets start with localhost.

A.2.7 ***WAPS***

Current Version = 0.1

Input

- None

Output

- WiFi Access Point Data

Expected Form

- Talks to a AdaFruit Huzzah with NodeMCU Lua firmware, connected via a USB or serial port.

Overview

The WiFi Access Point Scraper (WAPS) module scrapes local WiFi data. That is all it does, the main loop just queries WiFi data and publishes it every tick.

- Example Start Call: ./../waps -/dev/WAPS_UART -tcp://*:5570 -tcp://localhost:6666 -1 -1000 -tcp://localhost:5575 -tcp://localhost:5576

- Input Parameters(in order, parameters **MUST** be prepended with '-' in the call):

  1. portPath
     - Path to the serial port.
     - Example = -/dev/WAPS_UART
  2. dataPubAddress
     - Socket to publish data to, 1024 to 65535 see this for more on that. This just needs to be declared and different from other modules.
     - Example = -tcp://*:5570.
       Note: local **publish** sockets start with *, local **subscribe** sockets start with localhost.
  3. killSubAddress
     - Socket that the Killswitch sends out the Kill message.
     - Example = -tcp://localhost:6666.
       Note: local **publish** sockets start with *, local **subscribe** sockets start with localhost.
  4. tickTimePub
     - Tells the module to publish the tick time data. 0 or 1. See section 7.4.2 for definition.
     - Example = -1

5. ticklength
   - Tells the module how long each tick should be. Expects an integer, milliseconds assumed.
   - Example = -1000
6. (Optional) subaddress_1, subaddress_2, ... subaddress_n
   - Additional subscription addresses. Currently none required, left in for future work.
   - Example = -tcp://localhost:5575 -tcp://localhost:5576
     Note: local **publish** sockets start with *, local **subscribe** sockets start with localhost.