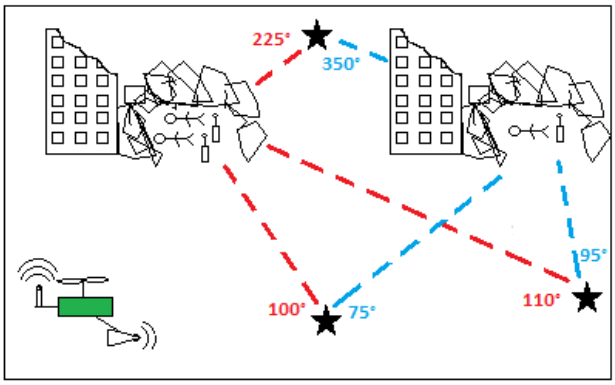
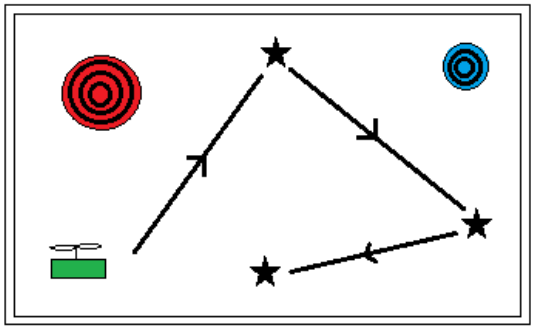
Drone Phone Home

GSM Localization in Disaster Scenarios

June 2015





DES906F15

Barbara Albertine Flindt

Jeppe Blicher Tarp

Aalborg University
Department of Computer Science
Software

AALBORG UNIVERSITY

STUDENT REPORT

Title:
Drone Phone Home - GSM Localization in Disaster Scenarios

Theme:
Embedded Systems

Project Period:
SW10, Spring Semester 2015

Project Group:
DES906F15

Authors:
Barbara Albertine Flindt

Jeppe Blicher Tarp

Supervisor:
René Rydhof Hansen, Mads Christian Olesen, Andrea Fabio Cattoni

Page Count:
68

Finished:
June 2015

Signatures:

Barbara Flindt: _____

Jeppe Tarp: _____

Resume

When natural disasters strike in urban areas hundreds or thousands of people die, many of them in the days following the disaster. Collapsed buildings bury people and finding and extracting them is a time-consuming and often dangerous undertaking. Given that 80% of people pulled out from debris have their cell phone on them, using GSM localization could be a faster, safer way to help locate buried victims.

This report details the development of a module for GSM localization which can be attached to a UAV to help USAR workers locate buried victims. This solution was first presented in the pre-specialization project. We first present preliminary knowledge required to understand our solution such as knowledge about GSM networks, OpenBTS and USAR as well as preliminary testing of the solution. We also describe our use case scenario. Then we present relevant tools and techniques, such as triangulation, directional antennae and more.

After detailing our revised hardware architecture, we describe the software architecture to support the module and the reasoning behind our route planning system, which is focused on maximising localization accuracy when performing angle of arrival measurements. We also describe our mathematical reasoning behind how triangulation with inaccurate angle of arrival measurements affects the accuracy.

After explaining the design and implementation of the software system we detail the development of a software simulation to test the system, seeing as we did not have all the hardware necessary to perform live testing.

Finally we reflect on scalability, limitations and future work for this project and conclude.

Preface

This report documents the development by the group des906F15 of a prototype of a module to be used on UAVs for GSM localization in disaster areas. The reader is expected to have technical knowledge equivalent to that of a 4th semester master's student of Software Engineering at AAU. As well, the reader is expected to have read the pre-specialization report from the previous semester and have a basic understanding of GSM and radio signals. The report focuses especially on a control system for the UAV, including route planning and localization using angle of arrival measurements. A list of acronyms used is provided at the start of the report and acronyms will also be defined in footnotes the first time they appear in the text.

Acknowledgements

The group would like to thank Anders Friis, Chief Technology Officer at SkyWatch A/S, for procuring hardware, and for providing information about UAVs and the initial idea for the project. The group would also like to thank René Rydhof Hansen, Mads Chr. Olesen and Andrea Fabio Cattoni for excellent supervision and guidance during the semester. Finally, the group would like to thank the OpenBTS community for their quick and helpful responses in regards to the use of OpenBTS.

Contents

List of Acronyms

AoA Angle of Arrival

API Application Programming Interface

BSC Base Station Controller

BTS Base Transceiver Station

CLI Command Line Interface

CPU Central Processing Unit

FDMA Frequency Division Multiple Access

GPS Global Positioning System

GSM Global System for Mobile Communications

GUI Graphical User Interface

IMSI International Mobile Subscriber Identity

IP Internet Protocol

I-LOV Intelligentes sicherndes Lokalisierungssystem für die Rettung und Bergung von Verschütteten

JSON JavaScript Object Notation

LoS Line of Sight

LUR Location Update Request

MS Mobile Station

MSC Mobile Switching Center

MTC Mobile Terminated Call

RSSP Received Signal Strength Power

SAR Search and Rescue

SIM Subscriber Identity Module

SMS Short Message Service

TDMA Time Division Multiple Access

TDoA Time Difference of Arrival

UAV Unmanned Aerial Vehicle

USAR Urban Search and Rescue

USRP Universal Software Radio Peripheral

VOIP Voice over IP

Chapter 1

Introduction

Natural disasters hitting urban areas often carry a high death toll. Many of these deaths do not occur at the time of the disaster but in the days following, when rescue workers race against time to locate and extract victims from the rubble. After 48 hours a buried victim's chances of survival are very slim. Statistics from the 1995 earthquake in Kobe, Japan showed that although the survival rate of people extracted was 80.5% on the first day, this number plummeted to 28.5% on the second day and 5.8% on the third day of the relief efforts [1]. As these numbers indicate, time is of the essence during USAR[1], but extraction of buried victims is time consuming work and thus it is vital to locate places with many buried victims as quickly as possible. Additionally, natural disasters wreak havoc on established infrastructure in urban areas and thus navigating a disaster area is slow and sometimes even dangerous for rescue workers. Already, UAVs[2] are being utilized to make the navigation of disaster areas safer and quicker so using these further in the USAR work is an option that needs to be explored. Seeing as a survey by the German Federal Agency for Technical Relief showed that 80% of buried victims had their phone on them when rescued this opens up the possibility of using GSM[3] localization to figure out the best places to start the rescue operations.

Problem Statement

Previously we have described the problem of finding people buried in debris after natural disasters [2]. This is an important issue for rescue workers and relatives of buried victims and currently locating people in debris is a cumbersome task, which costs human lives. In this report we will describe the development of a module that can be attached to a UAV and which will facilitate rough localization of cell phones in a disaster area.

[1] Urban Search and Rescue
[2] Unmanned Aerial Vehicles
[3] Global System for Mobile Communications

In this first prototype of the module the focus will been on:

- Prototype of our suggested architecture wrt. software and hardware.
- Control system and communications for the module.
- Mathematical reasoning about triangulation with inaccurate data, seeing as our research has not brought up any sources for this.
- A route planning algorithm to maximise localization accuracy.
- Optimization measures for scalability and efficiency.

Chapter 2

Preliminaries

This chapter covers preliminary testing of the solution's hardware architecture as well as theoretical knowledge needed to fully understand the system and its application. All subjects mentioned in this chapter, save architecture testing, are covered more in-depth in the previous semester report [2].

2.1 Urban Search and Rescue

This section provides an overview of what USAR entails. USAR is the term used for SAR[1] operations in urban environments, which usually means a large amount of rubble and debris and many people buried in the ruins of buildings.

USAR has 5 stages:

- Preparedness - time between disasters. USAR teams maintain preparedness by training crew and maintaining gear. Organizations should be able to respond to calls for assistance 24/7.
- Mobilization - time immediately following a disaster. Local teams assess damage and communicate needs for international assistance.
- Operations - rescue operations. The stage at which teams perform rescue operations.
- Demobilization - ceasing operations. Teams pack up and leave the disaster site.
- Post Mission - evaluation. Review of operations and discussion of possible improvements. Teams then return to preparedness stage.

During operations, using technological aids for localization of buried victims falls under what is known as technical SAR. The technology is useful for medium and heavy USAR teams, that perform extractions from collapsed buildings and other structures.

[1] Search and Rescue

2.2 GSM Networks

GSM is the most widely used standard for mobile communications. It facilitates voice calls, SMS[2] and basic data communication.

From "GSM Localization in Disaster Scenarios" [2]:

> GSM is a cellular network [3] where coverage is achieved by having a large quantity of cells. Each cell is given a globally unique identifier, and is served by a BTS[3]. Every cell is assigned to a number of channels which consists of two frequencies, one for downlink and one for uplink, within the same frequency band. Cells which are geographically close are assigned to different channels, such that handsets communicating with different BTS's create minimal interference. This is known as FDMA[4].
>
> Each BTS is controlled by a BSC[5] which has several base stations under its control. The BSC manages assigning channels to each BTS and handing of connected handsets from one BTS to another. Additionally, the BSC relays all communication from each BTS under its control to an MSC[6] and vice versa. The MSC is responsible for routing calls and other services to the recipient, and invoking other GSM elements, such as SIM[7] identification, which will not be described in this report, as they are not relevant to this project. An illustration of the network can be seen in Figure 2.1

From the same source on handset to BTS communication[2]:

> As mentioned, GSM handsets communicate with a given BTS through a number of channels. These channels are a pair of frequencies in a frequency band, for example, the GSM-1800 band uses frequencies 1710-1785 MHz for uplink and 1805-1880 MHz for downlink. The difference between the uplink and downlink frequency for a channel is always the same within a frequency band, so for channels in the GSM-1800, the downlink frequency will always be 95 MHz higher than uplink. This difference is called the duplex spacing of a band. A channel uses 200 KHz, so the GSM-1800 band contains more than 350 different channels, specifically the channels numbered 512 to 885 in the GSM specification[4]. To allow multiple handsets to communicate with a base station,

[2]Short Message Service
[3]Base Transceiver Station
[4]Frequency Division Multiple Access
[5]Base Station Controller
[6]Mobile Switching Center
[7]Subscriber Identity Module

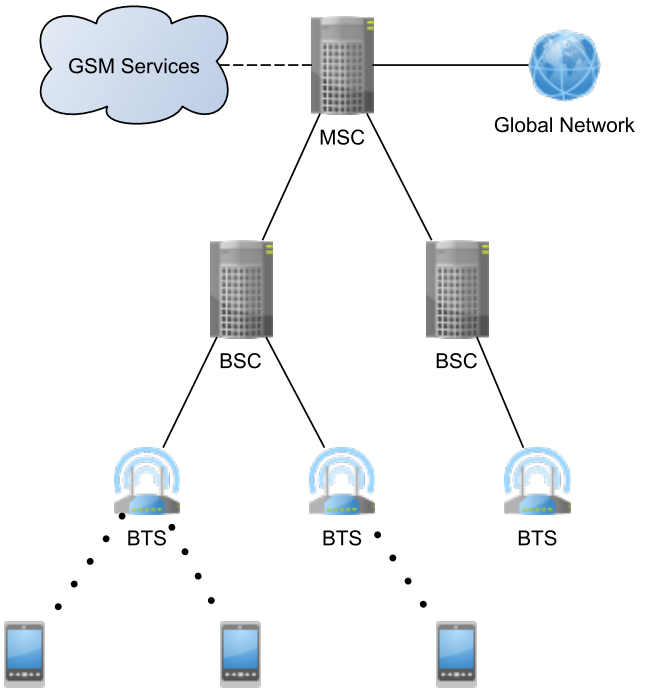


Figure 2.1: GSM Network

TDMA[8] is used. In TDMA time is divided into timeslots, which are clustered together in groups of eight called frames. Each device will be assigned to a time slot in which it is allowed to transmit to the BTS. This synchronization is achieved by enabling the BTS, when it starts receiving transmissions from the handset, to reply with how much earlier the handset needs to transmit next time to meet its assigned time slot [4].

2.3 Localization

Several GSM localization techniques exist, here AoA[9] will be explained. Localization by AoA is quite simple in theory: the angle of arrival of the signal from the MS[10] is determined from at least two BTS and the intersection of those angles is the location of the MS. An illustration of this can be found in Figure 2.2. The accuracy of this localization technique is dependent on

[8] Time Division Multiple Access
[9] Angle of Arrival
[10] Mobile Station

whether there is a clear LoS[11] or not. For further explanation of GSM localization techniques, see the previous project report[2].

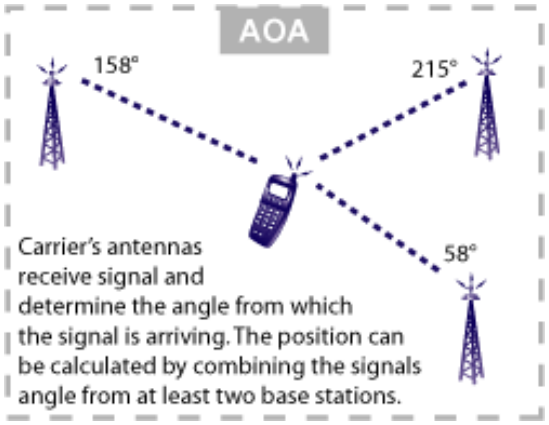


Figure 2.2: Positioning via Angle of Arrival [5]

2.4 Existing Solutions

Currently a few projects exist that utilize GSM localization in SAR operations. A quick overview of these will be given in this section.

2.4.1 I-LOV

The I-LOV[12] research project was a German research project endorsed by the Federal Ministry of Education and Research in Germany. Their solution to finding people trapped in debris was a five layer architecture, seen in Figure 2.3 and summarised below:

1. A mobile handset to be located and any existing GSM networks.
2. A special BTS equipped with a jammer.
3. Handheld scanners used to measure signal strength and angle of arrival.
4. Infrastructure layer handling communication, synchronization and organization.
5. A PC providing the needed computation powers for analysis.

[11] Line of Sight
[12] Intelligentes sicherndes Lokalisierungssystem für die Rettung und Bergung von Verschütteten

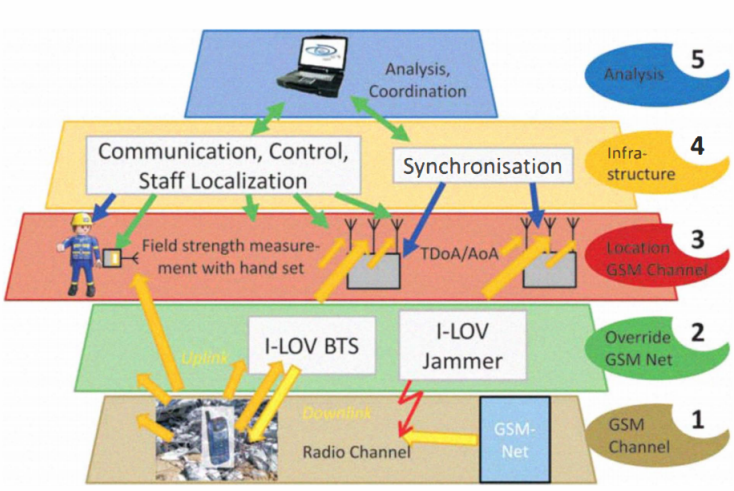


Figure 2.3: I-LOV Architecture [6]

The general idea is that the rescue team upon arrival will set up the I-LOV base station and turn on the jammer to ensure that all mobile handsets in the area connect to their system. The special BTS will then connect to handsets and initiate a modified MTC[13] which makes the handset transmit with full power. Rescue workers will then walk around with a field strength measuring device utilizing AoA and RSSP[14], which is the strength of the received signal, to locate the handsets.

2.4.2 RESCUECELL

RESCUECELL is a project funded by the European Union, finished in late 2014, which utilizes GSM to assist with SAR missions. The architecture includes three different types of devices:

- Static Nodes, which are a type of BTS that should be placed around the affected area.
- Mobile Nodes, which are handheld devices carried around the disaster area by SAR personnel.
- Command Centre, which is a single node handling data collection and system management of static and mobile nodes.

Information about the RESCUECELL system is, at the time of writing, still very sparse and as such not much more information can be given about it.

[13] Mobile Terminated Call
[14] Received Signal Strength Power

2.5 Architecture

The initial hardware architecture of this project is shown in Figure 2.4. A module with a USRP[15] and a small computer is attached to the UAV which relays information to a control station set up in the SAR headquarters.

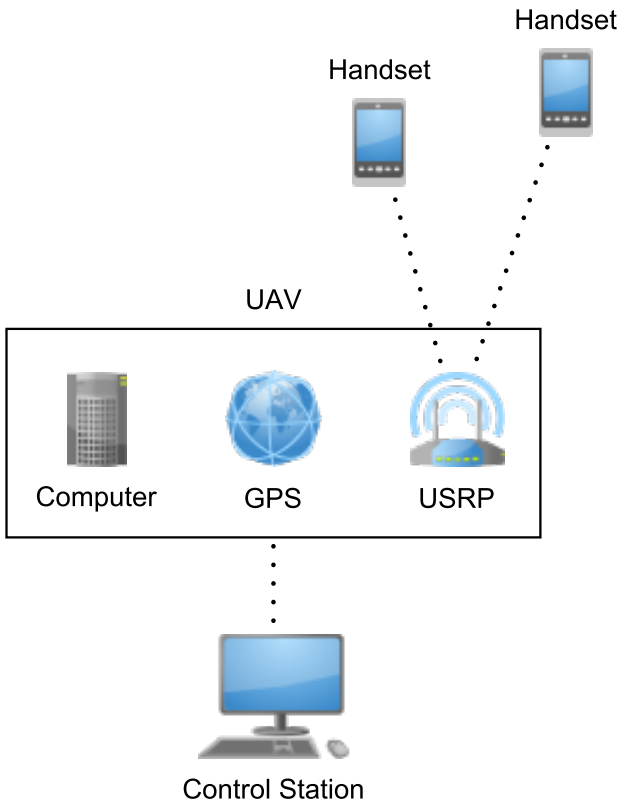


Figure 2.4: Overall Architecture[2]

The USRP acts as a BTS and allows handsets in the area to connect to it in order to start locating them. The USRP is equipped with two antennae: a directional receiving antenna and an omnidirectional transmission antenna, which will allow for AoA measurements. The on-board computer runs the BTS software and handles data management of incoming handset information as well as flight pattern calculations to facilitate localization of connected handsets. The UAV has a GPS[16] which provides location coordinates for the

[15]Universal Software Radio Peripheral
[16]Global Positioning System

localization algorithm. The USRP can possibly allow two-way communication between buried victims and rescue personnel.

The control station receives information from the UAV module and creates a visual representation of the located handsets in the area. As well, the control station facilitates manual operation of the UAV if need be.

2.5.1 Testing of Setup

To determine if the architecture was viable a small test was performed. A B200 USRP with a directional receiver and an omnidirectional transmitter, connected to a laptop which recorded the results, was placed on top of a 10 metres tall hill. 50 metres away, at the foot of the hill, a single MS, connected to the BTS, was left on the ground. The laptop runs the BTS software OpenBTS which has a Physical Channel API[17] providing RSSP measurements. OpenBTS will be explained further in Section 3.1. With the receiving, directional, antenna being turned clockwise in 45 deg steps two SMS were sent to the MS at every step and the RSSP was recorded. The test setup is illustrated in Figure 2.5, and the result can be seen in Figure 2.6.

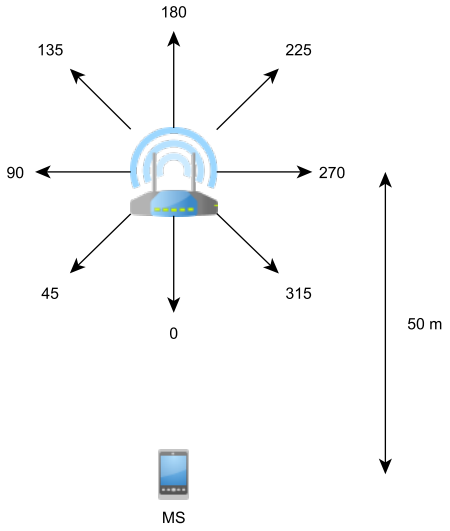


Figure 2.5: Test Setup

As can be seen in the result only the three RSSP measurements closest to pointing directly towards the handset resulted in a received signal. This confirms information that the tested receiving antenna should have a directionality of around 120°[7] and that there is a significant decrease in signal

[17] Application Programming Interface

Direction	Received RSSP ($dB_{fullscale}$)
0	-40
45 & 315	-45
90 through 270	-71 (Only noise)

Figure 2.6: Test Results

power as the antenna is faced away from the source. This has both positive and negative consequences; the positive consequence is that using an AoA localization method is indeed possible, the negative consequence is that this hardware setup is not enough to implement such a system, as any handsets not in front of the UAV would simply be disconnected from the network, in which case it would stop generating messages on the Physical Channel API. A revised architecture for the implementation of the system will be described in Section 4.1

2.6 Use Case

With the preliminaries covered, this section will describe the use case for this project. In the use case an earthquake has hit a major city. Local USAR teams have communicated their need for medium and heavy USAR teams to extract victims from the rubble. The teams arrive on the disaster site and set up their headquarters. Once they have set up, a technical SAR worker scans the area to see if any GSM networks are still functioning and if so, jams these signals in the area where USAR work is being done.

Once this has been achieved, one or several UAVs equipped with the Drone-PhoneHome localization module are switched on and given information about the size and location of the area they need to cover. The UAVs first connect with nearby handsets (rescue worker phones are excluded from the network at this point) and after a suitable number of handsets have connected, the UAVs plan their route and take off. The UAVs fly over the area that the team is supposed to cover and find buried handsets. These findings are relayed back to the headquarters where a technical SAR worker monitors the UAVs and their gathered results together with the person(s) responsible for choosing where to start extracting buried victims.

If the UAVs need manual control, the technical SAR workers will handle this so that operation of the module is not dependent on the UAVs being autonomous. Once the UAVs are running low on battery they return and a new UAV can either be deployed, or the UAVs can be called back and extraction work in the chosen site can begin. During extraction the UAVs can be utilized to help rescue workers navigate the rubble and find the safest and shortest route to their destination as well as facilitate communication via the

GSM network.

Figure 2.7 is an image to illustrate the use of the product as well as the primary concerns of the users.

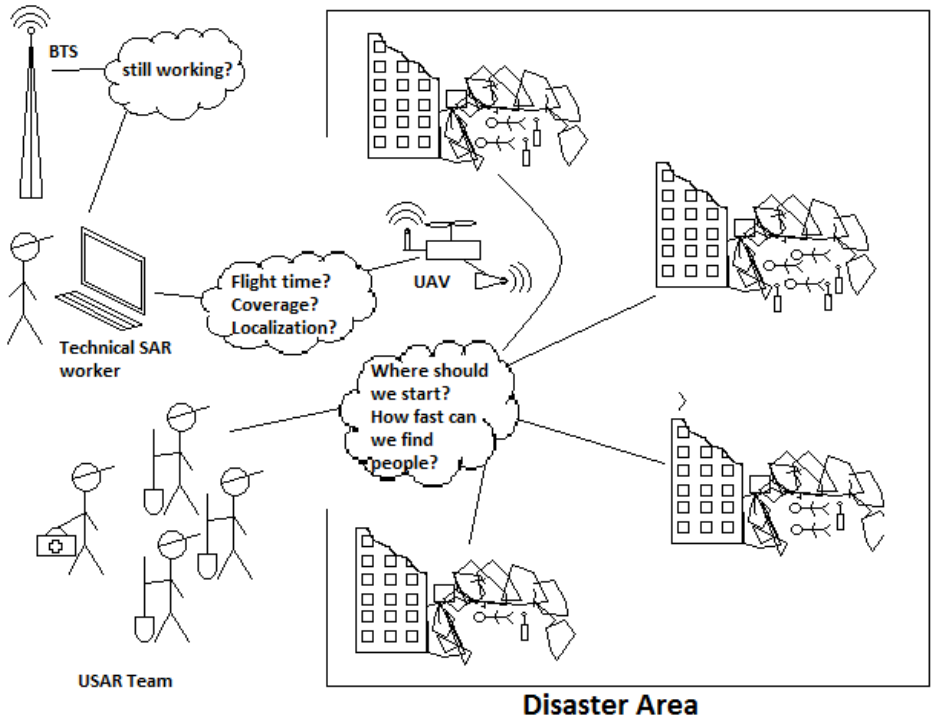


Figure 2.7: Illustration of Use Case

Localization of the buried handsets is done using AoA to triangulate the location. This is achieved using the OpenBTS Physical Channel API and turning the UAVs around to measure signal strength with the directional antennae. The GUI[18] shows the located handset in a way that allows the technical SAR worker to easily spot good sites to begin work. The UAVs fly following individual pre-planned routes which maximise the number of sites they can visit and the amount of readings they can perform. An illustration can be seen in Figure 2.8.

[18]Graphical User Interface

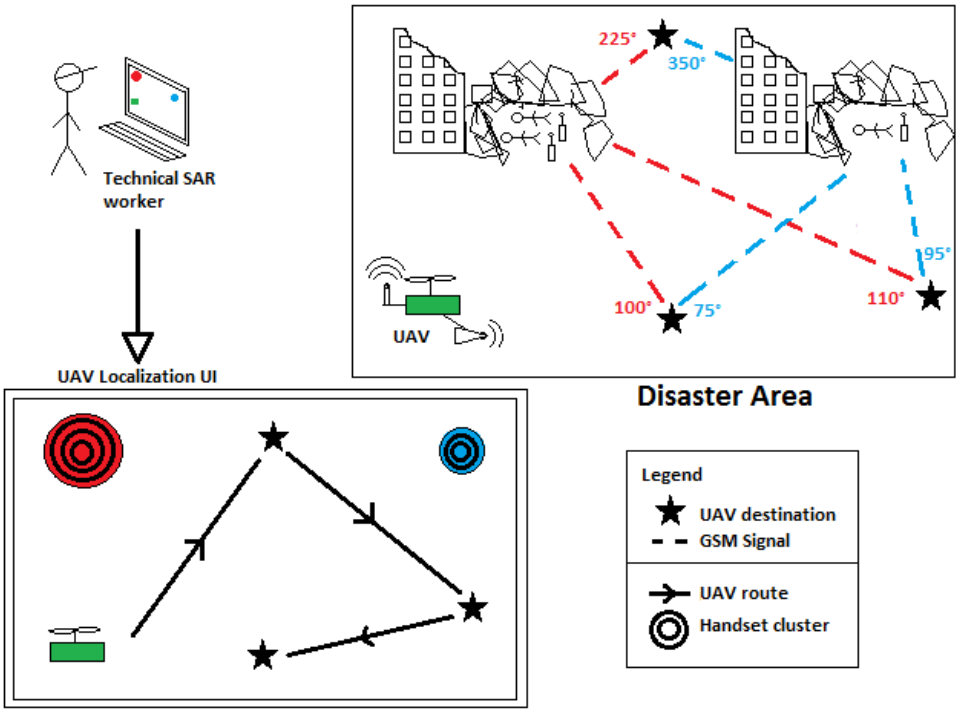


Figure 2.8: Illustration of Use Case

Chapter 3

Tools and Techniques

This chapter describes the tools and techniques used in the project, some of which are part of our contribution to the solution of the problem.

3.1 OpenBTS

OpenBTS is a free, open source, software based GSM access point. It utilizes IP[1] connectivity to deploy a GSM network that can allow connected handsets to send SMS and make voice calls over VOIP[2]. Additionally, through the Physical Channel API, OpenBTS provides information about connected handsets such as RSSP, connected channel, time delay and more. This information is relayed approximately every half second during data communication and voice calls as well as LUR[3]. This API in conjunction with a directional antenna enables AoA determination with OpenBTS. In Listing 3.1 an example of a message from the Physical Channel API can be seen.

```
1  {
2    "name" : "PhysicalStatus",
3    "timestamp" : "18446744072283447705",
4    "version" : "0.1",
5    "data" : {
6      "burst" : {
7        "RSSI" : -49.4808,
8        "RSSP" : -27.4808,
9        "actualMSPower" : 11,
10       "actualMSTimingAdvance" : 0,
11       "timingError" : 1.59709
12     },
13     "channel" : {
14       "ARFCN" : 153,
15       "IMSI" : "001010000000001",
16       "carrierNumber" : 0,
```

[1]Internet Protocol
[2]Voice over IP
[3]Location Update Request

```
17        "timeslotNumber" : 0,
18        "typeAndOffset" : "SDCCH/4-1",
19        "uplinkFrameErrorRate" :
20      },
21      "reports" : {
22        "neighboringCells" : [],
23        "servingCell" : {
24          "RXLEVEL_FULL_dBm" : -67,
25          "RXLEVEL_SUB_dBm" : -67,
26          "RXQUALITY_FULL_BER" : 0,
27          "RXQUALITY_SUB_BER" : 0
28        }
29      }
30    }
31  }
```

Listing 3.1: Physical Channel API Message Example

The information relevant to this project is the `IMSI`, the unique identifier for each SIM, `timestamp`, `RSSP`, `actualMSPower` and `actualMSTimingAdvance`. The `actualMSTimingAdvance` gives a very rough estimation of the distance between the MS and the BTS, within 550 metres. This is useful for filtering out handsets that are too far away to be considered for the current sweep performed by the UAV. `RSSP` is the raw signal strength reading which depends on the direction of the antenna. The `actualMSPower` is the transmission power of the MS during that burst and it is used to ensure that the `RSSP` readings are consistent. The reason for using OpenBTS is that it is free and open-source, unlike other GSM access point software.

3.2 USRP

The USRP setup is comprised of a board and up to two antennae: a transmitter and a receiver. The B200 USRP[8] is a software defined radio, i.e. a radio where signal generation and modulation is handled digitally. The B200 board has a frequency range of 70 MHz to 6 GHz, which means it can generate and receive radio signals within this range. The actual output and input ranges of the setup is dependent on the overlap between the frequency ranges of the board and the antennae. As the antennae have ranges of 850 MHz to 6.5 GHz for the receiver and 824 MHz to 960 MHz (or 1710 MHz to 1990 MHz as it is a dualband antenna) for the transmitter the frequency range is perfect for GSM signals. The B200 USRP can be seen in Figure 3.1. The weight of a single B200 is 94 grams.

3.2.1 Antennae

The receiving antenna is a so-called log-periodic directional antenna. The antenna is constructed as a series of dipole antennae spaced in a logarithmic

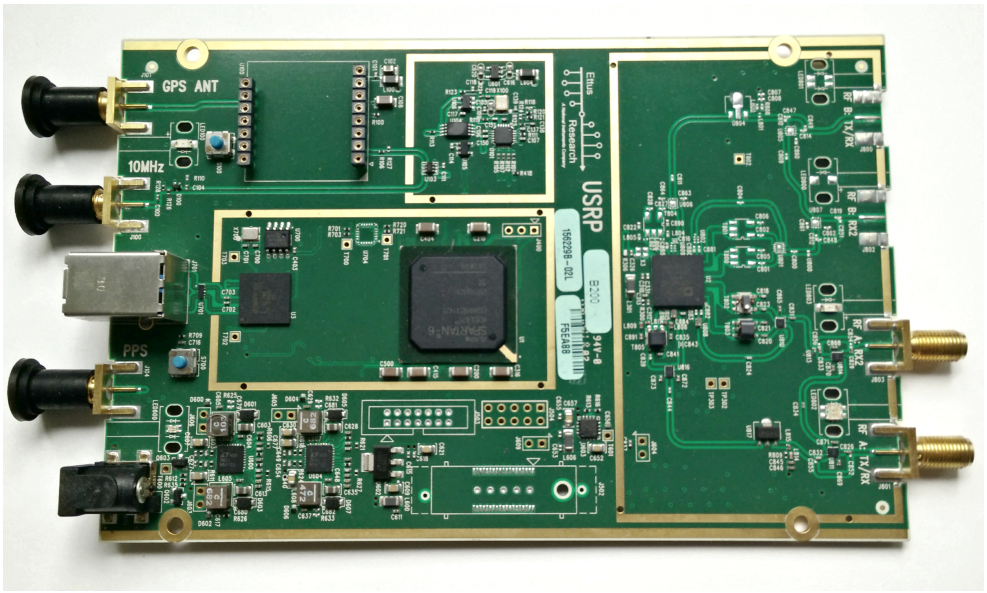


Figure 3.1: B200 USRP

function of the frequency, called σ, with the length of each element corresponding to resonance at different frequencies within the bandwidth of the antenna [9]. The resulting radiation pattern of this construction is shown in Figure 3.2. The radiation pattern of the omnidirectional antenna is a torus. The weight of the omnidirectional antenna is 21 grams and 34 grams for directional antenna.

Moving the antenna around and monitoring the drop and increases in signal strength gives an indication of the direction of the signal, as the signal will be received better when the tip of the antenna is pointed at the source. The antennae used are shown in Figure 3.3.

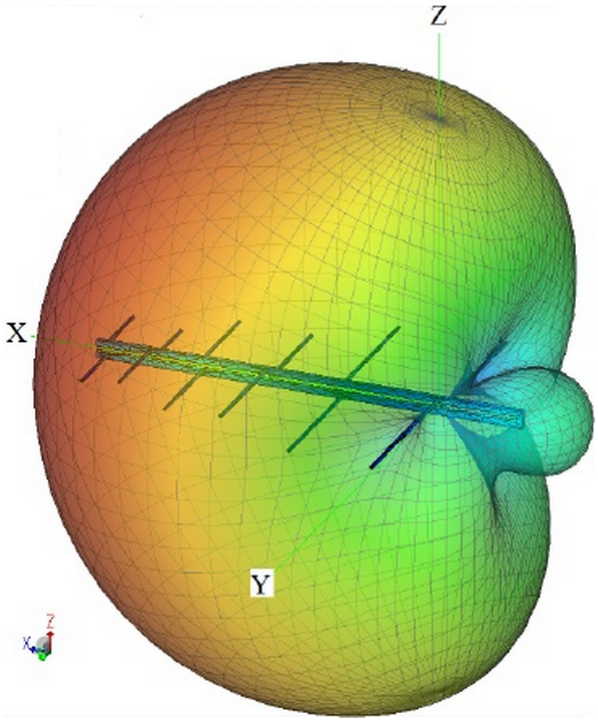


Figure 3.2: Radiation Pattern of a Log-Periodic Antenna [10]

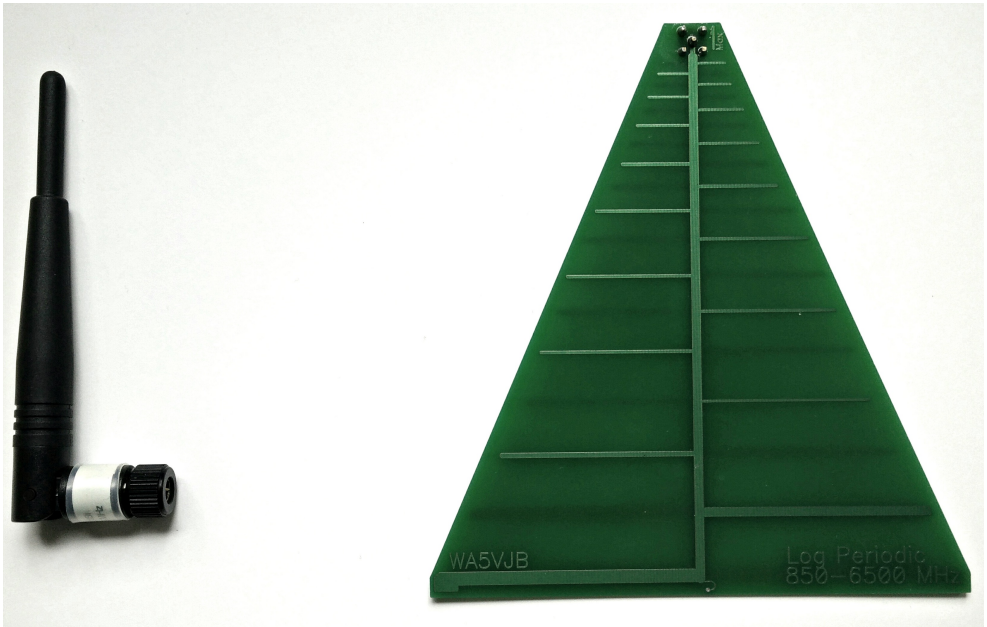


Figure 3.3: Omnidirectional and Directional Antennae

3.3 Computer

In order to perform localization, run OpenBTS and take readings from the Physical Channel API a computer is required. Seeing as the hardware is going to be placed on a UAV the computer needs to be very lightweight, yet still powerful enough to perform the needed computations. An option for this is the BeagleBone Black, a credit-card-sized computer with 512MB RAM, 4GB storage and 2x PRU 32-bit microcontrollers. The BeagleBone weighs only 42 grams and can be seen in Figure 3.4

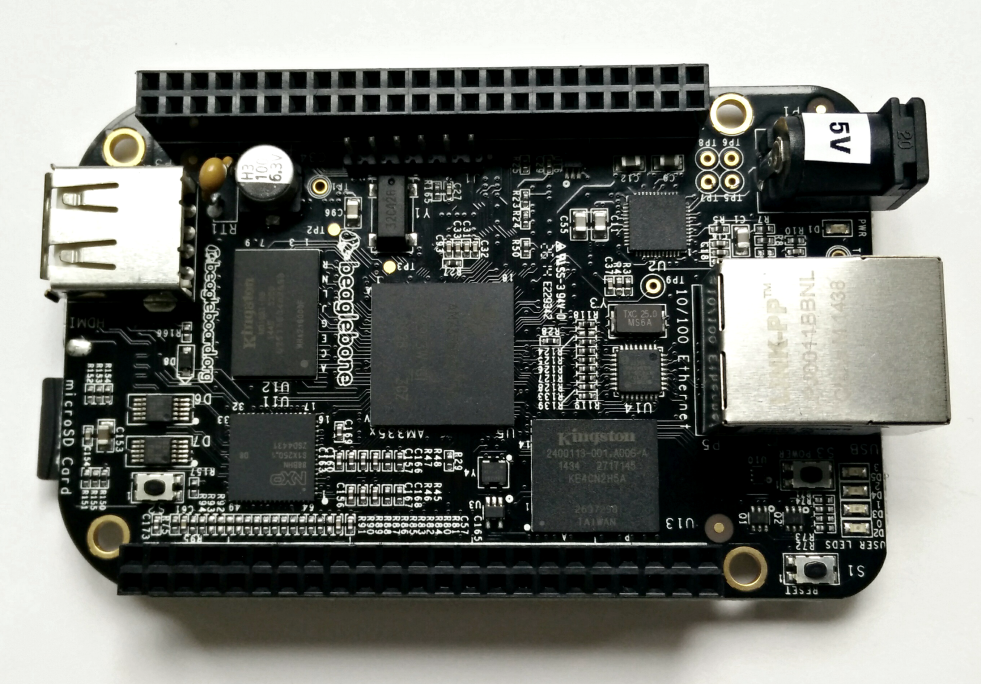


Figure 3.4: BeagleBone Black

3.4 Triangulation

Triangulation is a technique for locating an unknown point using measurements of angles. In Euclidean geometry, the location of the point C can be found by measuring the angle towards it from two known points A and B. These angles are relative to the angle of the line between the known points. We describe the angles measured at A and B as α and β respectively, and the distance between A and B as l. In Figure 3.5 a triangle has been created from lines going through the three points, with angles α and β. The distance between A and C can then be found using the law of sines:

$$\frac{|AC|}{sin\beta} = \frac{l}{sin(180 - (\alpha + \beta))} \Leftrightarrow |AC| = \frac{l}{sin(180 - (\alpha + \beta))} sin\beta \tag{3.1}$$

Knowing the distance and angle from A to C allows us to find the exact location.

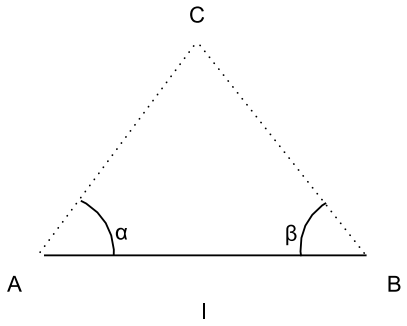


Figure 3.5: Triangulation

3.4.1 Effects of Inaccurate Angle Measurements

If triangulation is performed with inaccurate angle measurements, this will of course have a negative effect on the accuracy of the triangulation. To find out how much this inaccuracy affects the result, a model was created. In this model, we describe the maximum inaccuracy of a measured angle as E. Instead of using α and β as angles of lines, they are used as the facing of a flat cone (or an infinite height triangle) with an angle at the top of $2E$. Within this model, the target object can be anywhere within the area of overlap of these two cones. This leaves us with two other parameters affecting the accuracy of the triangulation: The distance from the target object, and the difference of α and β. This is shown in Figure 3.6.

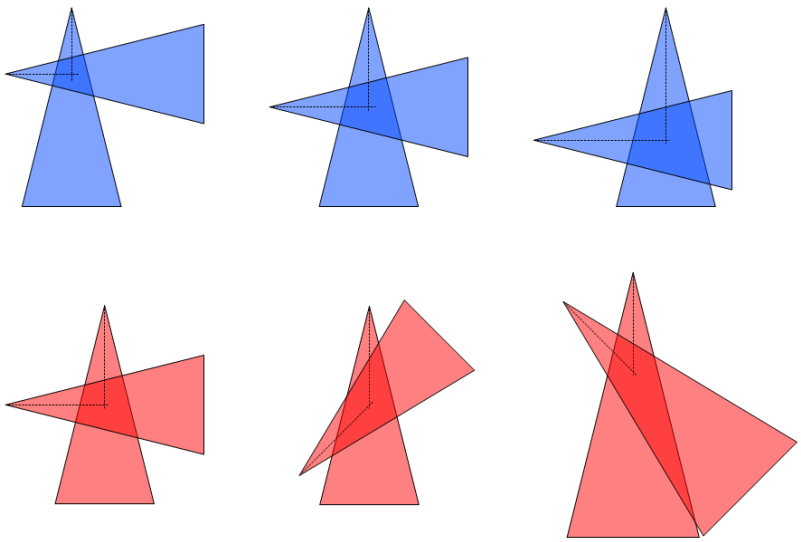


Figure 3.6: Triangulation Accuracy

To calculate the accuracy of a triangulation, we compute the values shown in Figure 3.7, where $triangulate(A, B, \alpha, \beta)$ is a function returning the point C using the method of triangulation described earlier in this section. C_1 through C_4 describes the intersections of the edges of the two cones. The maximum error of the location of C can then be described as $E_{loc} = max(|C_1C|, ..., |C_4C|)$. A test of the inaccuracy based on distance and a test based on directions was performed, and the results are presented in Figure 3.8 and Figure 3.9 where the distance from C to A and B are named d_a and d_b respectively. The difference of direction is described as $\Theta = (180 - (\alpha + \beta))$.

Looking at the results in Figure 3.8, we see that the inaccuracy is roughly proportional to the average distance between measuring sites and target, which is intuitively consistent, since the width of a cone is proportional to the distance from the top. The reason for the results only being *roughly* proportional, is that as the cone widens, the shape of the other cone has a greater effect on the results.

The results in Figure 3.9 are more interesting. We see that the optimal difference of angle is around 90°, and worsens as we get closer to 0° and 180°. The reason why $0 \leq \theta \leq 10°$ does not yield a result, is the fact that for any $\theta <= 2E$ the cones will overlap and two of the intersections will be undefined, in essence leading to a possible inaccuracy that is infinite. Similarly, when

Name	**Value**
α_1	$\alpha - E$
α_2	$\alpha + E$
β_1	$\beta - E$
β_2	$\beta + E$
C	$triangulate(A, B, \alpha, \beta)$
C_1	$triangulate(A, B, \alpha_1, \beta_1)$
C_2	$triangulate(A, B, \alpha_1, \beta_2)$
C_3	$triangulate(A, B, \alpha_2, \beta_1)$
C_4	$triangulate(A, B, \alpha_2, \beta_2)$

Figure 3.7: Values for Inaccuracy Computation

d_a	d_b	$(d_a + d_b)/2$	Θ	E	E_{loc}
10	10	10	90°	5°	1.36
20	10	15	90°	5°	2.11
30	10	20	90°	5°	2.94
40	10	25	90°	5°	3.80
50	10	30	90°	5°	4.66
50	20	35	90°	5°	5.04
50	30	40	90°	5°	5.54
50	40	45	90°	5°	6.13
50	50	50	90°	5°	6.78

Figure 3.8: Triangulation Localization Inaccuracies wrt. Distance

$\theta \geq 180° - 2E$ the cones face each other and contain the top of the other cone within their area. This also leads to undefined intersections, however in this case the inaccuracy is not infinite, but $l_a + l_b$ as the target will be somewhere between the two points.

We also see a slight bias towards larger θs rather than smaller. This happens because of the fact that when the two cones face slightly away from each other, distance is exaggerated by the error in angle, much more so than when they are facing slightly towards each other. This effect is visible from the last illustration in Figure 3.6.

d_a	d_b	Θ	E	E_{loc}
20	20	0°	5°	N/A
20	20	10°	5°	N/A
20	20	20°	5°	20.00
20	20	30°	5°	10.04
20	20	40°	5°	6.73
20	20	50°	5°	5.10
20	20	60°	5°	4.12
20	20	70°	5°	3.49
20	20	80°	5°	3.04
20	20	90°	5°	2.71
20	20	100°	5°	2.71
20	20	110°	5°	3.04
20	20	120°	5°	3.49
20	20	130°	5°	4.12
20	20	140°	5°	5.10
20	20	150°	5°	6.73
20	20	160°	5°	10.04
20	20	170°	5°	N/A
20	20	180°	5°	N/A

Figure 3.9: Triangulation Localization Inaccuracies wrt. Direction

3.5 Tangent Plane Projection

This section is based on the technical report on latitude and longitude tangent plane projection by Ivan S. Ashcraft[11].

When performing triangulation in a small area it is computationally and mathematically simpler to work with Cartesian (x, y) coordinates rather than latitude and longitude. In order to achieve this, the latitude and longitude of the desired area is projected onto a tangent plane centred on the point of tangency, illustrated in Figure 3.10. The tangent plane is oriented with the positive y-axis going North. To achieve the projection, the following equations are used:

$$R_\phi = R_E cos(\phi) \tag{3.2}$$

where R_E is the local radius of the Earth at the centre of the tangent plane and ϕ is the latitude.

$$A = R_E sin(\Delta\phi) \tag{3.3}$$

$$B = R_\phi(1 - cos(\Delta\theta))sin(\phi_0) \tag{3.4}$$

where θ is the longitude and ϕ_0 is the latitude at the point of tangency.

$$C = R_\phi sin(\Delta\theta) \tag{3.5}$$

$$x = C \tag{3.6}$$

$$y = A + B \tag{3.7}$$

It should be noted that x and y are in the same units as the radius of the Earth.

Conversion from tangent grid to latitude and longitude carries the complication that R_ϕ is needed, but is also an unknown in the calculation. Thus, $\phi \approx \phi_0$ is used, and accuracy can be improved by iteration with the result. For the conversion the following equations are used:

$$\Delta\theta = arcsin\left(\frac{x}{R_\phi}\right). \tag{3.8}$$

$$\Delta\phi = arcsin\left(\frac{y - (1 - cos(\Delta\theta))sin(\phi_0)R_\phi}{R_E}\right). \tag{3.9}$$

The latitude is given by $\phi = \phi_0 + \Delta\phi$ and longitude is given by $\theta = \theta_0 + \Delta\theta$, where ϕ_0 and θ_0 is the latitude and longitude at the point of tangency.

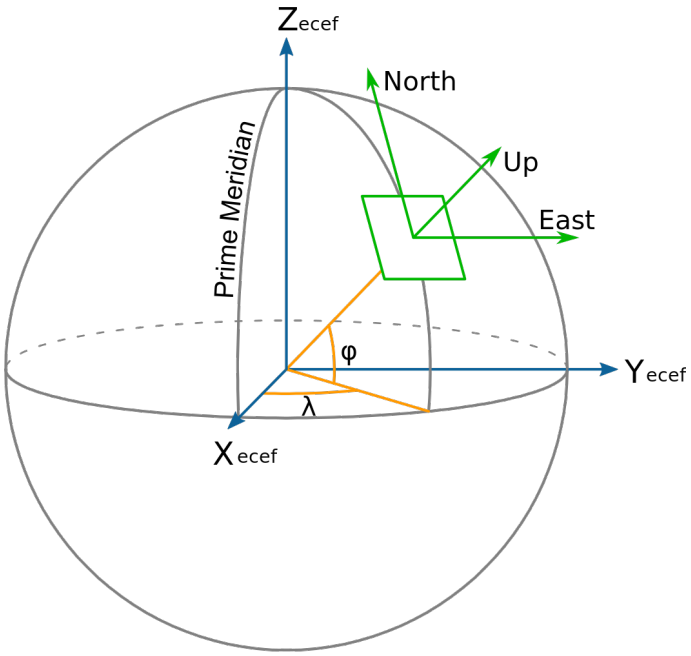


Figure 3.10: Tangent Plane Graphics: Wikipedia, public domain

3.6 ZeroMQ

ZeroMQ (also written ∅MQ) is a lightweight, scalable communications library supporting communication both between threads or processes on a single machine but also across different platforms and protocols, such as TCP. The messaging can be done with various patterns such as publisher-subscriber, explained below. The library is what allows OpenBTS to publish messages from the Physical Channel API and for consistency in this project all socket communication is done via ZeroMQ. The library is supported in every major language which means porting this project is easy in this regard [12].

3.6.1 Publisher-Subscriber Pattern

Publisher-subscriber is a messaging pattern where, instead of sending a specific message, characterized in classes, to a specific receiver, the publisher simply publishes the messages without knowing if there are any subscribers and what they want. Many subscribers can subscribe to the same publisher. The subscriber, much like the publisher, subscribes to a certain class of messages without knowing if there are any publishers of this message class.

Chapter 4

Solution

This chapter details analysis, design and implementation of the system prototype.

4.1 Revised Hardware Architecture

As described in Section 2.5, the previously proposed architecture would not be usable in an actual implementation of the system, meaning the architecture had to be revised. The changes to the proposed architecture are as follows:

- Two receiving channels are needed, one with the directional antenna for taking RSSP readings, and one omnidirectional antenna for communication between BTS and MS. However, as the USRP only has two antenna connections two USRPs with synchronized clocks should be connected to the BTS computer. Clock synchronizations can be achieved on a hardware level on the B200 USRPs by connecting them with a cable[8]. The desired hardware setup is listed in Figure 4.1.
- The RSSP from the OpenBTS Physical Channel API can no longer be used, instead the RSSP from the reading USRP at the time of the GSM burst should be used. To realize this, a feature should be added to OpenBTS such that it receives signal data from the second USRP, and replaces the RSSP published in the Physical Channel API with that of the directional antenna.

These changes mean that the architecture overview shown in Figure 2.4 from Section 2.5 no longer accurately illustrates the system, and a revised overview can be seen in Figure 4.2.

It should be mentioned that we did not have access to a second USRP at the time of this project, so development has been completed with the original architecture, using software simulations instead of real world testing where necessary.

USRP	Frontend	Antenna	Designation
USRP 1	RX	Omni	Uplink
	TX	Omni	Downlink
USRP 2	RX	Directional	Readings

Figure 4.1: Final USRP Setup

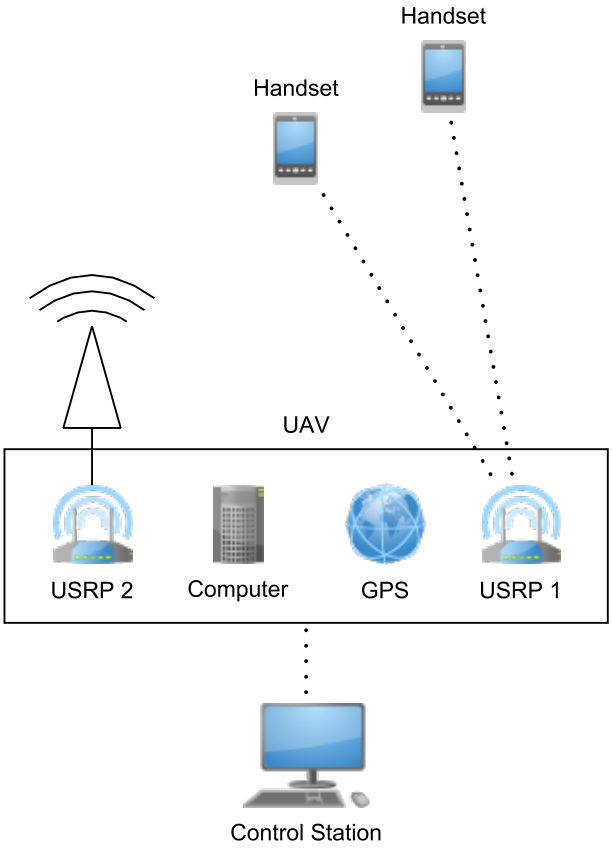


Figure 4.2: Revised Architecture

The weight of the required hardware components can be seen in Figure 4.3.

Item	Weight(g)	Quantity	Total (g)
BeagleBone	42	1	42
Omni Antenna	21	2	42
Dir Antenna	34	1	34
SMA-SMA Cable	35	2	70
B200	94	2	188
USB Cable	60	2	120
Total			496

Figure 4.3: Architecture Payload

4.2 Software Architecture

The software for the prototype implementation of the system has been written in Python, and split into five modules, as seen in Figure 4.4. This distribution of modules is based on the *type* of operations implemented in each module, rather than the subject matter of those operations, for example, there is functionality related to UAV communication in both the input and output module. An overview of each module will be given in this section, and in-depth explanation of the design and implementation of the most interesting parts will be presented later in the chapter.

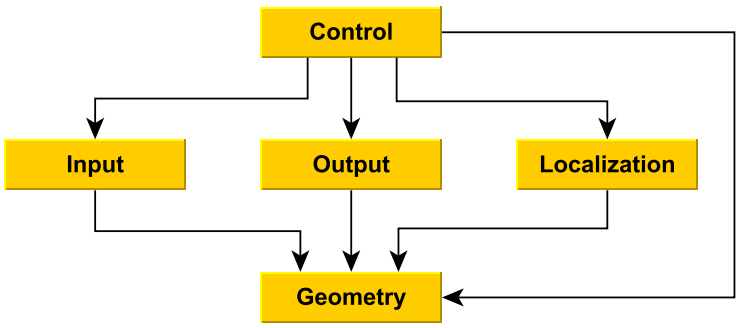


Figure 4.4: Software Modules

4.2.1 Geometry

The geometry module contains data structures and functionality pertaining to coordinates. It is split up in two main parts: world, which contains the `LatLng` class and calculations done on these latitude and longitude sets, and Cartesian, which is home to the `Vector` class, as well as calculations re-

lated to geometry on a two dimensional plane. Finally, the geometry module contains the `TangentPlane` class, which implements the math described in Section 3.5, to convert between `LatLng` instances and Cartesian coordinate sets.

The geometry module is the base for a lot of the functionality in this system, and is referenced by all other modules.

4.2.2 Input

The input module relates, as the name implies, to obtaining data from outside the system. Two main sources of information are present: The UAV itself, and the OpenBTS process running alongside the system. The UAV is simply expected to publish physical data about itself, such as its latitude, longitude, facing and battery level.

The communication with OpenBTS is more involved. The input module contains a system for receiving and sorting messages posted by the OpenBTS Physical Channel API, as well as functionality to run commands in the OpenBTS CLI[1] and parse the results of those. These commands include getting a list of connected handsets, reading the background noise level as well as sending SMS messages. Sending messages might have been more logically placed in the output module, but it is the only CLI command that is not used for retrieving data, so it was allowed to be grouped with the other CLI functionality.

4.2.3 Output

The output module handles communication from the system to other processes. The system needs to send data to two destinations: The UAV needs to be told about desired destinations and facings, so that the built-in software on the UAV can move to and maintain these. Additionally, the system needs to send data to the GUI being monitored by the USAR worker operating the system.

In this prototype GUI communication is simply handled by a websocket server where any number of GUI clients can connect. This requires a direct network connection from the UAV to the computer running the GUI, and in a final version, some other way of relaying this information should be created. A possibility could be transmitting the information through a mobile data connection using OpenBTS.

4.2.4 Localization

The localization module contains algorithms for three purposes:

[1] Command Line Interface

- Route planning for the UAV, to give the system the greatest possible chance to accurately locate as many handsets as possible, before running out of battery for flight.
- Calculations of AoA from data retrieved in the input module, and triangulation of handset locations using the calculated angles.
- A system for managing the calculated angles and location per handset, and determining whether the accuracy of a given handset's location can be improved.

We have tried to keep the localization module and the input module as decoupled as possible, to make either replaceable if a better option should be discovered.

4.2.5 Control

This module contains the control system. The control system is responsible for employing the other modules, figuring out when the UAV should move to the next point in the route, when OpenBTS should be told to send SMS messages to which handsets, how often to receive data from input systems etc. In essence, while the other modules determine the *functionality* of the system, the control module determines its *behaviour*.

4.3 Usage of OpenBTS

A central element of the solution is the usage of the OpenBTS Physical Channel API, as described in Section 3.1. A large number of messages are generated, and they need to be processed and stored. The system responsible for this is a part of the `input` module described in Section 4.2. We decided to create the system for managing these messages in three tiers:

1. The message tier - where each individual message is represented.
2. The group tier - where messages are grouped by handset and timestamp.
3. The manager tier - which is a single manager that can be queried for groups matching certain parameters.

Each of these tiers were implemented as a class, the functionality of which will be described in the following sections.

4.3.1 The Message Tier

The message tier was implemented in a class called `PhysicalAPIReading`. In hindsight, the naming of this class could have been better, as when we refer to a "reading" in the rest of the report, we actually refer to the data available in the group tier, not the message tier. With that said, the class can be seen in Listing 4.1.

```
1  class PhysicalAPIReading(object):
2    def __init__(self, message, location, noise):
3      self.location = location
4      self.noise = noise
5      self.process_message(message)
6
7    def process_message(self, message):
8      data = message['data']
9      burst = data['burst']
10
11     self.timestamp = int(message['timestamp'])
12     self.imsi = int(data['channel']['IMSI'])
13     self.ms_power = burst['actualMSPower']
14     self.rssp = float(burst['RSSP'])
```

Listing 4.1: The PhysicalAPIReading class

The constructor takes three parameters:

- `message`, a JSON[2] message from the API as described in Section 3.1.
- `location`, a structure containing latitude, longitude and facing of the UAV at the time the message was received.
- `noise`, the noise level at the time.

The `process_message` function, called from the constructor at line 5, extracts the data from the message that is needed for the system and the rest is discarded. The necessary data has been described in Section 3.1.

4.3.2 The Group Tier

The group tier is represented by a class called `PhysicalAPIGroup`. Instances of this class are what represents a "reading" when talking about the system as a whole, and contain a list of `PhysicalAPIReading` instances. These instances are added to the group by the manager, and are required to satisfy two conditions:

- Each message must originate from the same IMSI[3]

[2] JavaScript Object Notation
[3] International Mobile Subscriber Identity

- Each message must have been generated from the same data transaction with the handset.

It is up to the manager to make sure these conditions are satisfied. However, when this is the case, a `PhysicalAPIGroup` has a number of useful properties.

- It contains messages related to a single IMSI.
- It contains messages generated while the UAV was at a single location, which can be compared to the location of other groups.
- The messages in the group can be sorted by timestamp, giving the group a start time and end time.
- A full group can be generated by sending an SMS to a handset.

The group additionally exposes functionality to filter the messages contained in it. The implementation of this filtering can be seen in Listing 4.2. The purpose of the filtering is to remove messages which had a different UAV facing than the others, since these are likely to be messages that were received after the UAV started turning, and as such might have the wrong RSSP. Additionally, it removes readings where the handset transmitted with a power that was too different from the median power, to ensure that all RSSP values are on the same scale.

The `weighted_direction` method called in line 10, works by taking a list of angles and creating a unit vector with each angle. These vectors are then summed, and the angle of the resulting vector is returned. In lines 13-15 the list comprehension completes the actual filtering of the list, creating a copy with only the desired elements.

```
1  class PhysicalAPIGroup(object):
2    #...
3    def filtered_readings(self):
4      facings = []
5      powers = []
6      for r in self.readings:
7        facings.append(r.get_facing())
8        powers.append(r.get_ms_power())
9
10     facing = weighted_direction(*facings)
11     power = powers[int(len(powers) / 2)]
12
13     result = [r for r in self.readings if
14               (abs(r.get_facing() - facing) <= 10 and
15                abs(r.get_ms_power() - power) <= 2)]
16     return result
```

Listing 4.2: Filtering messages from a group

4.3.3 The Manager Tier

The manager is represented by the class `PhysicalAPIManager`. Only one instance of this class should be created, and it contains a list of groups. The manager has two important functions: It must be able to place a message into the correct group, and it should be possible to query the manager for groups satisfying certain conditions.

The function for placing messages, as seen in Listing 4.3, goes through each existing group and finds matches for the new reading. A group is a match for a message if their IMSI matches, and the message timestamp is within a threshold of the timespan between the start and end time of the group. Finding matches happens in the list comprehension in lines 8-11, and has three possible results:

- If no matches were found, a new group is created for the reading.
- If one match was found, the message is added to that group.
- If two matches were found, it means that the message would bridge the gap in those groups' timespans, so the groups are merged and the message is added to the resulting group.

```
1  class PhysicalAPIManager(object):
2    #...
3    def add_reading(self, reading):
4      time = float(reading.get_timestamp())
5      imsi = reading.get_imsi()
6      target = None
7      # Check if the reading fits in any current group(s)
8          matches = [group for group in self.groups if
9                      (imsi == group.imsi
10                   and time > group.start_time() - self.threshold
11                   and time < group.end_time() + self.threshold)]
12     # If no groups matched, create a new one
13     if len(matches) == 0:
14       target = PhysicalAPIGroup(imsi)
15       self.groups.append(target)
16     else:
17       target = matches[0]
18       # If multiple groups were matched, the new
19       # reading would make this groups too similar
20       # in timestamps, therefore they should be merged
21       for m in matches[1:]:
22         target.merge(m)
23         self.groups.remove(m)
24     target.add_reading(reading)
```

Listing 4.3: Placing a message in the correct group

The querying function can be seen in Listing 4.4. The primary use for this function is to find all groups referring to the same handset generated at

a specific point, as this is the set of groups necessary to compute the AoA for the handset at that location. The function is designed to work with keyword arguments, so a call to the function might look like:

`manager.filter_groups(imsi=12345678, near=LatLng(57.029308, 9.979909))`

Additional keyword arguments available are `distance`, which determines how far from the desired point a group is allowed to be, and `remove`, which can be set to true to have the manager remove the resulting groups from its list. The if statement at line 7-9 makes sure a keyword argument is actually present before using it for filtering, for example, if no IMSI was specified, groups with all IMSIs are returned.

```
1  class PhysicalAPIManager(object):
2    #...
3    def filter_groups(self, imsi = None, distance = 10,
4                            near = None, remove = False):
5      result = []
6      for group in self.groups:
7        if ((imsi is None or imsi == group.imsi)
8              and (distance is None or near is None or
9                   group.is_near(near, distance))
10       ):
11         result.append(group)
12
13     if remove:
14       for group in result:
15         self.groups.remove(group)
16
17     return result
```

Listing 4.4: Querying for specific groups

All in all, this structure allows for easy access to the necessary readings for whichever task they are needed.

4.4 Determining Angle of Arrival

To determine the AoA of a signal from a handset, several readings are needed at the same location, with the directional antenna facing in different directions. Optimally, you would turn the antenna smoothly in a 360° turn and consider the angle with the greatest RSSP the correct angle. However, as OpenBTS Physical Channel API is the source of the RSSP measurement, only discrete readings are available, whenever communication with a handset is in progress.

The solution to this is to pick a number of readings, n, to make at each site, initiate communication with each handset that is currently being tracked, for example by sending an SMS to each of them, and waiting for the API to publish channel information. Then, the UAV should turn the antenna $360°/n$ and repeat the procedure, until n readings have been obtained.

To calculate a direction from these discrete readings, take the readings made for each handset at each direction and define a vector with direction equal to the facing of the antenna at the time of that reading, and magnitude equal to some function of the RSSP of the reading. Then, each of these vectors are summed, and the direction of the resulting vector is considered the AoA. Examples of this approach can be seen in Figure 4.5. As illustrated, this approach allows us to have a better granularity in results than the angle turned between each reading.

All that is left to do is to define the function from RSSP to magnitude. Two considerations apply here: First, the RSSP is measured in dB relative to the maximum receiving power of the USRP, i.e. a negative number, where the noise level in the area is a lower number. Second, measuring in dB means that an increase of ten in RSSP is a signal one order of magnitude greater. Given these facts, we define the function as:

$$mag = 10^{(RSSP-NOISE)/10} \tag{4.1}$$

This approach means that noise level readings will produce unit vectors, and the relation between the magnitude of the other vectors will be proportional to the actual power received.

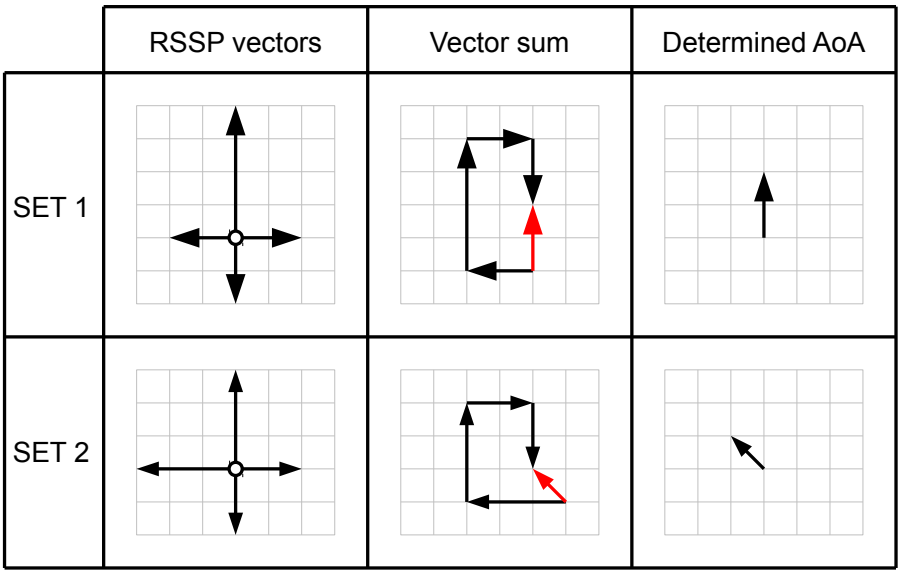


Figure 4.5: Example AoA calculations

4.4.1 Evaluation

To test the accuracy of this method, we created a model for RSSP at different directions, based on our testing, described in Section 2.5.1. In this model we assume that full power, ω is received at an angle, θ, of zero, compared to the direction towards the handset. We assume that the power drops linearly by $5dB$ every 45° up to 60° (the directionality of the antenna) where it drops to noise level, α, and that no signal less than noise level is received. The model is as follows:

$$RSSP = \begin{cases} MAX(\omega - 5/45 \times \theta, \alpha) & \text{if } \theta < 60 \\ \alpha & \text{otherwise} \end{cases} \tag{4.2}$$

This test was run with three different ω values, with θ at 5° intervals, and assuming eight readings to determine an angle (i.e. a reading every 45°). The results are shown in Figure 4.6. These results show that, assuming the model is consistent with reality, this method is able to detect the correct angle of arrival with maximum inaccuracy of 5°. The results are naturally best if a reading is made pointing directly at the handset, and nearly as good if the angle is directly between two readings.

ω / θ	$\alpha + 20$	$\alpha + 30$	$\alpha + 40$
0	0.00	0.00	0.00
5	2.51	2.47	2.47
10	5.45	5.36	5.35
15	17.93	18.00	18.01
20	20.96	20.98	20.98
25	24.04	24.01	24.02
30	27.07	27.00	27.00
35	39.55	39.64	39.65
40	42.49	42.52	42.53
45	45.00	45.00	45.00

Figure 4.6: AoA test using model

4.4.2 Implementation

The AoA calculations are implemented in the localization module mentioned in Section 4.2. The results of the calculations are stored in a class called `AngleOfArrival` which has the IMSI, location in latitude and longitude and, naturally, the calculated angle. The actual computation is performed in the function `compute_aoa`, seen in Listing 4.5

```
1  def compute_aoa(groups, imsi, point):
2    sets = [[r for r in group.filtered_readings()] for group in groups]
3    result = Vector()
4    for set in sets:
5      directions = []
6      power = 0
7      noise = 0
8      for reading in set:
9        directions.append(reading.get_facing())
10       power += reading.get_rssp()
11       noise += reading.get_noise()
12     power /= len(set)
13     noise /= len(set)
14     mag = 10 ** ((power - noise) / 10)
15     v = Vector.DM(weighted_direction(*directions), mag)
16     result += v
17
18   return AngleOfArrival(imsi, point, result.direction(),
19                         result.magnitude())
```

Listing 4.5: Determining Angle of Arrival

The parameter `groups` is a list of `PhysicalAPIGroup` instances, which is described in Section 4.3. The function is a direct implementation of the method illustrated in Figure 4.5 and the calculation described in Equation 4.1. This function is one of the only instances in the system where the modules are not decoupled, seeing as Listing 4.5 is dependent on the implementation of `PhysicalAPIGroup`, from the input module.

4.5 Triangulating Location

In Section 3.4 the math for triangulating the position of a handset based on the AoA calculations is explained. This is implemented in the localization module mentioned in Section 4.2. The triangulation is implemented in a class called `Triangulator`, which is instantiated with a tangent plane.

4.5.1 Triangulation Calculations

The triangulation and error calculation is split into three functions: `triangulate`, `triangulate_aoa` and `triangulate_eval`. The function `triangulate_aoa` can be seen in Listing 4.6.

```
1  class Triangulator(object):
2  #...
3    def triangulate_aoa(self, first, second):
4      a = self.plane.to_cartesian(first.point)
5      b = self.plane.to_cartesian(second.point)
6      theta = Vector.XY(b[0] - a[0], b[1] - a[1]).direction()
7      alpha = first.direction - theta
8      beta = 180 - (second.direction - theta)
9      c = self.triangulate(a, alpha, b, beta)
```

```
10      return c
```

Listing 4.6: Triangulation with `AngleOfArrival` Instances

As shown, `triangulate_aoa` takes as its parameters two instances of the `AngleOfArrival` class which, as explained in Section 4.4, has both direction and position (in latitude and longitude) as attributes. The positions are converted to Cartesian coordinates and in line 6 a variable, `theta` is calculated. This variable will be used to convert the angle measurements of the two points from being relative to the global 0 degrees (due East) to being relative to the line through the location of the two measurements. An illustration of this can be found in Figure 4.7. The two angle measurements are adjusted and the `triangulate` function is called and the resulting Cartesian coordinate returned.

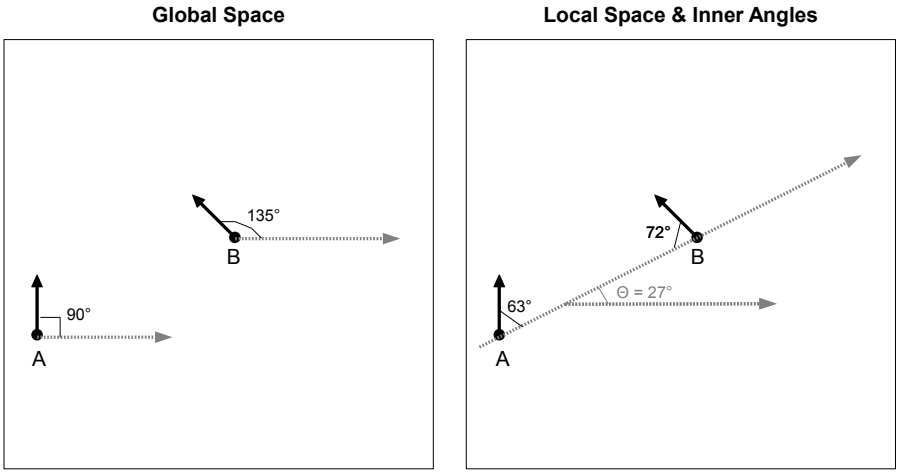


Figure 4.7: Illustration of conversion from global to relative angles

The `triangulate` function, seen in Listing 4.7, is a direct implementation of the math for basic triangulation described in Section 3.4.

```
1  class Triangulator(object):
2  #...
3    def triangulate(self, a, alpha, b, beta):
4      v = Vector.XY(b[0] - a[0], b[1] - a[1])
5      l = v.magnitude()
6      ac = l / sin(radians(180 - alpha - beta)) * sin(radians(beta))
```

```
7       d = v.direction() + alpha
8       return a[0] + cos(radians(d)) * ac, a[1] + sin(radians(d)) * ac
```

Listing 4.7: The Triangulation Function

4.5.2 Computing Inaccuracy

In order to calculate the eror of the measurement the `triangulate_eval` function has been implemented and can be seen in Listing 4.8.

```
1  class Triangulator(object):
2  #...
3    def triangulate_eval(self, first, second, error = 5):
4      try:
5          c = self.triangulate_aoa(first, second)
6      except:
7          return 0, 0, float('inf')
8      f1 = AngleOfArrival(None, first.point,
9                          first.direction - error, None)
10     f2 = AngleOfArrival(None, first.point,
11                         first.direction + error, None)
12     s1 = AngleOfArrival(None, second.point,
13                         second.direction - error, None)
14     s2 = AngleOfArrival(None, second.point,
15                         second.direction + error, None)
16     try:
17       Cx = [self.triangulate_aoa(f1, s1),
18             self.triangulate_aoa(f2, s1),
19             self.triangulate_aoa(f1, s2),
20             self.triangulate_aoa(f2, s2)]
21
22       Dx = [sqrt((c[0] - Ci[0]) ** 2 + (c[1] - Ci[1]) ** 2)
23             for Ci in Cx]
24     except:
25         Dx = [float('inf')]
26
27     return c[0], c[1], max(Dx)
```

Listing 4.8: Finding Error in Triangulation

This function implements the math for triangulation with inaccurate measurements, described in Section 3.4.1. The list `Cx` in line 17 contains all the triangulation measurements of the intersections of the cones resulting from the inaccuracy. The list `Dx` in line 22 contains the distances from these to the triangulated result. The return values are the Cartesian coordinates for the triangulation as well as the maximum error in metres.

4.6 Route Planning

When planning the flight route of the UAV, it is desirable to perform readings in as many sites as possible. Since we will be using a triangulation technique

with AoA information, the best accuracy will be achieved with readings from different sides of the target handsets, so the chosen sites additionally have to be spaced far enough apart that it is likely that each handset is surrounded, but still close enough that inaccuracies in the AoA have only minimal effect. An algorithm was designed to maximize the number of sites based on a number of parameters. The necessary parameters can be split into three categories, pertaining to the UAV hardware, the target area and algorithm preferences, and are listed in Figure 4.8.

Parameter	**Description**
UAV / Hardware	
t_{flight}	UAV flight time (s)
$t_{reading}$	Time needed to complete a directional reading (s)
t_{turn}	Time for the UAV to turn 360°(s)
v_{flight}	UAV flight velocity (m/s)
Target Area	
L_{target}	Center of the target area (lat,lng)
L_{start}	Location of the UAV before starting (lat,lng)
r_{target}	Desired radius around L_{target} (m)
Algorithm	
$n_{readings}$	Number of directional readings per site

Figure 4.8: Route Planning Parameters

Given these parameters, we want the algorithm to return a route as a series of n sites $\{L_1, ...L_n\}$. We define a route as valid, if the UAV can complete travel and readings within its flight time, described as $t_{route} \leq t_{flight}$. A number of calculations are necessary to find t_{route}. As described in Section 3.5, it is possible to convert latitude/longitude coordinates to Cartesian coordinates and back. In this case we use a tangent plane centred on L_{target}, such that any point can be described by L_i in latitude and longitude or P_i in x and y coordinates, and $P_{target} = (0, 0)$. This means we can use Euclidean geometry in calculations regarding routes. With this in place the following equations are used:

$$t_{site} = t_{turn} + t_{reading} \times n_{readings} \tag{4.3}$$

$$t_{route} = n \times t_{site} + \frac{|\overrightarrow{P_{start}P_1}| + \sum_{i=1}^{n-1} |\overrightarrow{P_iP_{i+1}}|}{v_{flight}} \tag{4.4}$$

Knowing how to validate a proposed route, it is also necessary to be able to compare the quality of a route. The quality of a route is defined as the ease of accurately triangulating an arbitrary point within the target area. As

described in Section 3.4, inaccuracies double when the average distance from the sites of measurement doubles. Additionally, angle differences, θ, close to 90° are optimal, while angle differences approaching 0° or 180° are nearly useless. This means we can approximate the accuracy to be proportional with $sin(\theta)$. Therefore we define the accuracy when locating an arbitrary point P_a as:

$$acc_{P_a} = max\left(\frac{sin(\theta)}{(|\overrightarrow{P_aP_i}| + |\overrightarrow{P_aP_j}|)/2}\right) \text{ for each } P_i \in route, P_j \in route \quad (4.5)$$

The quality of a route can then be described as the route which produces the highest average acc_P for all possible points within r_{target} meters of P_{target}. Since the number of points within the area of this circle is infinite, some method of choosing a representative set of points should be decided on. An example could be using Bridson's algorithm, to create a natural distribution of tightly packed points within the area[13][14].

Knowing how to compute both the validity and quality of a given route, an algorithm to find a good route should be described. Three different approaches to this were considered:

1. Find a way to generate the optimal route.
2. Generate a number of routes that are known to be valid and select the highest quality one.
3. Generate a number of routes where the relation between their quality is known, and select the best one that is valid.

The first option is obviously the most attractive. However, there are an infinite amount of possible routes, and no obvious way in which to calculate the best one. Even if the area is segmented into discrete points which are used as candidates for measuring sites within a route, the obvious solution is still to generate each possible route and check the quality and validity of each, turning this into a variation of the Travelling Salesman Problem, which is NP-complete, and therefore likely not an option in a time-critical circumstance such as SAR[15].

The concession that finding the optimal route is not feasible leaves the two heuristic approaches. Considering that computing the validity of a route is a lot simpler than the quality, we decided to focus on option three. This leaves us with the problem of generating a route that intuitively has a high quality. We decided to create the routes in the following manner:

1. Place a site in the center of the area. This has a few advantages: First, we limit the maximum distance to at least one measuring point to be the radius of the area. Second, if the UAV has to be recalled prematurely, having angle measurements in the center of the area at least gives a rough idea of where the most handsets can be found.

2. Place a number of sites on the circumference of the target area, having equal angles between each.

3. Validate the route. If it is valid, remember this route as the suggested route. If not, return the previous suggestion (if no previous suggestion, the size of target area should be decreased, or the UAV should be moved closer to the area.)

4. Start the process over, but place one more site during step 2.

Given that the parameters for the route planning algorithm comes from a variety of sources and are split across multiple functions, we decided to present pseudo-code for the implementation instead of the actual code. This can be seen in Algorithm 1. In Figure 4.9 an example of a route generated this way is shown, with a few handsets and their optimal triangulation sites. As can be seen from this, if the handset is on a line between the center and an edge site, it will have a useful angle with the center and a different edge site. Additionally, if it is on a line between two edge sites, it will have a useful angle between the center and one of those edges.

Input : A UAV, *uav*, which contains physical information, such as velocity, as well as its offset from the center of the area
Input : A number, *radius*, describing the size of the target area
Output: The resulting route, result

```
1  done ← false;
2  n ← 2;
3  result ← null;
4  while !done do
5  |  route ←emptyList;
6  |  append(route, (0, 0));
7  |  for i ← 0 to n do
8  |  |  dir ← i × (2π/n );
9  |  |  append(route, (cos (dir ) × radius, sin (dir ) × radius));
10 |  end
11 |  if isValid(uav, route) then
12 |  |  result ←route;
13 |  else
14 |  |  done ← true;
15 |  end
16 |  n ←n +1;
17 end
```

Algorithm 1: Route Planning Pseudocode

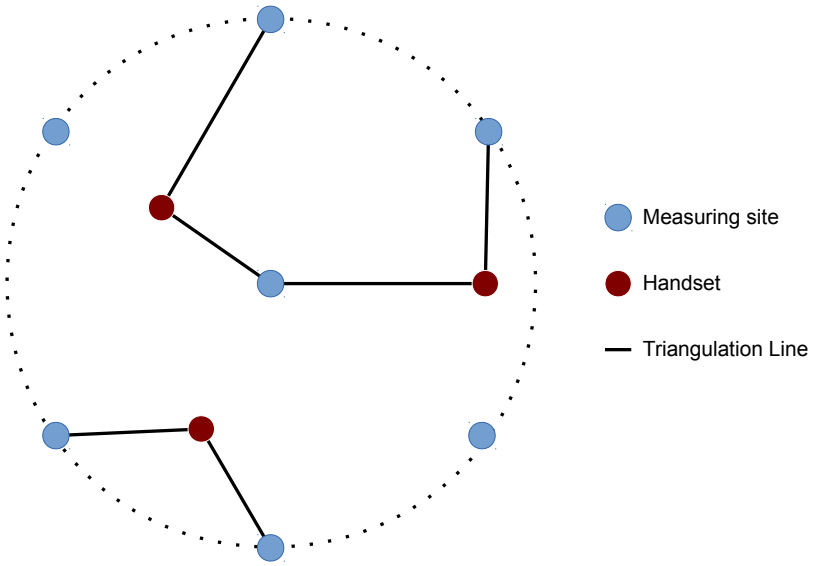


Figure 4.9: Example of Triangulation Angles in Suggested Route

4.6.1 Optimization and Evaluation

One of the biggest limitations of the proposed algorithm is the fact that the UAV has to start close to the center of the target area for the flight time to be spent optimally. To mitigate this, a simple optimization to the algorithm was made: If the UAV is more than two thirds of the radius away from the center, the first and second site in the route are swapped, and the route is rotated such that the new first site is in the same direction as the start point. This is illustrated in Figure 4.10.

When evaluating the route planning, we assumed a v_{flight} of six metres per second, t_{turn} and $t_{reading}$ both at six seconds, $n_{readings}$ at eight and a t_{flight} of 12 minutes. Given these parameters, it is possible to cover an area with a radius of 500 metres with the center site and four additional sites, assuming the UAV is launched from somewhere within the area. If t_{flight} is increased to 20 minutes, a radius of 1000 metres can be covered with the same number of sites.

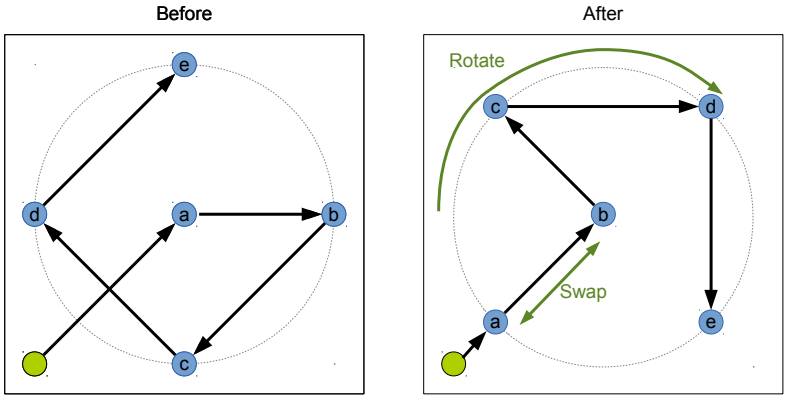


Figure 4.10: Route Optimization

4.7 Control System

The control system is, as mentioned in Section 4.2, responsible for the behaviour of the system. A rough description of the desired behaviour is as follows:

1. When starting up, make sure all dependencies are functional, i.e. make sure that the UAV communicates and that OpenBTS is running. Then the UAV system should wait for a target area, and plan a route.

2. Pop the next destination from the list of sites in the route, and tell the UAV to move to that location. While in transit, collect information about connected handsets from OpenBTS.

3. When a destination is reached, determine which handsets should be contacted and how many readings should be performed for each at this point.

4. Tell the UAV to turn to the desired facing for the next reading.

5. When the facing has been achieved, send SMS to the desired handsets, and collect API messages from OpenBTS.

6. If more readings are needed at this site, go to step 4, otherwise continue.

7. If there are more destinations in the route, go to step 2, otherwise shut the system down.

While this seems like a sensible behaviour for the system, looking at it from the outside, one important thing is missing: When to run the actual localization computations. One thing to consider is that the control system is sharing a processor with OpenBTS, a system that is very time critical, so it would

make sense to perform the computations at times where OpenBTS puts the least amount of strain on the CPU[4]. For this reason we decided to have the system compute and relay handset locations to the GUI during travel from one destination to the next. Additionally, when reaching the end of the route and "shutting down" the system should continue doing any pending computations and relaying data until the UAV is retrieved or battery is depleted. This desired behaviour is illustrated in Figure 4.11.

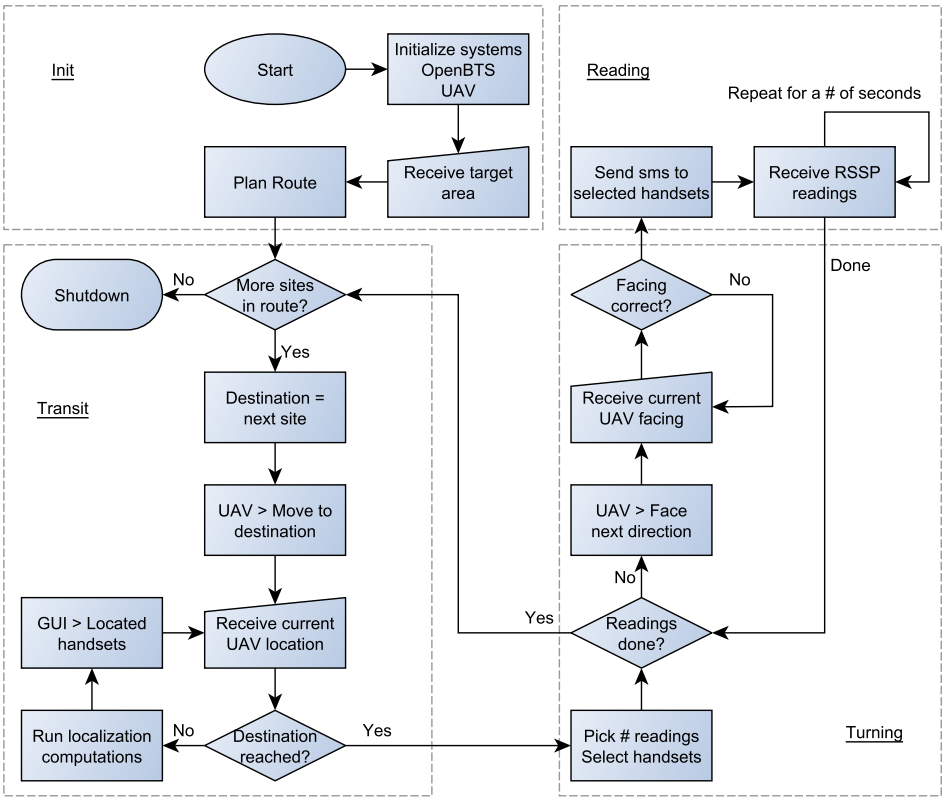


Figure 4.11: Flowchart of System Behaviour

An important observation of this system is that it is going to be performing radically different tasks at different points in time. These tasks are equivalent to the named areas in the flowchart, and are implemented in the system as an enumeration of behavioural states, shown in Listing 4.9, of which the system can be in one at a time.

```
1 States = type('Enum', (), dict(START = 0, TRANSIT = 1,
2                                 READING = 2, TURNING = 3))
```

Listing 4.9: System States

[4]Central Processing Unit

The implementation of these states is realized by using a variation of the command pattern [16]. The commands are a data structure, seen in Listing 4.10. Instead of naming the exact function to be called, it contains the next desired state of the system, as well as the parameters of that state, and a delay, which denotes how long the system should at most wait before running code related to that state. No target is specifically mentioned either, since the invoker and target are both the same instance of a central class called `Control`.

This class contains references to each other module, a number of handler functions for each state, and an invoker function, determining which handler should be called. In addition to having a handler for general operation, called the "step" handler, each state can optionally specify a handler for entering and exiting the state. Each handler must return a `Command` instance, which is passed to the invoker next. The functionality within each state is described in this section. However, as most of the code in the handlers revolve around calling functions from other modules, a minimal amount of the source code will be presented here. Before getting to the states, though, the invoker system will be presented.

```
1 class Command(object):
2   def __init__(self, nextstate, nextparam, delay):
3     self.nextstate = nextstate
4     self.nextparam = nextparam
5     self.delay = delay
```

Listing 4.10: The `Command` Structure

4.7.1 Delegation of Commands

Commands are invoked through two functions, seen in Listing 4.11. These functions assume a state structure which contains information about the previous state, current state as well as parameters to be passed to the next handler. The `delegate` function checks if the current state is different from the previous. If this is the case it calls the `run_delegation` function twice, with the *leave* and *enter* parameters, before updating the previous state. It always finishes by calling `run_delegation` with the *step* parameter.

The `run_delegation` function is what actually invokes the handlers. The first part of the function, lines 12-18 finds the correct handler in a lookup table, depending on previous and current state, as well as the parameter passed to `run_delegation` from the `delegate` function. If no handler was found, the function returns here. This check is done as the entry and leaving handlers for each state are optional. Then, in line 23, the chosen handler is called with the desired parameters, returning a `Command` instance. The parameter from the returned command is then saved into the state structure. However, as the *leave* and *enter* handlers are only called when a state transition is already in progress, they are not allowed to change which state is being transferred

to. The return value of both of these functions is the delay before the next delegation should take place.

The `delegate` function is called in a loop for as long as the control system is running, causing the control process to sleep for the desired delay between each call.

```
1  class Control(object):
2    #...
3    def delegate(self):
4      if self.state.current != self.state.previous:
5        self.run_delegation('leave')
6        self.run_delegation('enter')
7        self.state.previous = self.state.current
8
9      return self.run_delegation('step')
10
11   def run_delegation(self, type):
12     handler = None
13     if type == 'leave':
14       handler = self.handlers[self.state.previous]['leave']
15     elif type == 'enter':
16       handler = self.handlers[self.state.current]['enter']
17     else:
18       handler = self.handlers[self.state.current]['step']
19
20     if handler is None:
21       return 0
22
23     result = handler(self.state.param)
24     self.state.param = result.nextparam
25
26     if type == 'step':
27       self.state.current = result.nextstate
28       return result.delay
29     return 0
```

Listing 4.11: The `Command` Delegation Functions

4.7.2 The START State

In this state initialization happens. The system will enter this state upon start up, and once left, will never enter it again.

Entry

On entry, the system initializes all other modules, opening sockets and preparing singleton instances such as `PhysicalAPIManager` described in Section 4.3.

Step

In the step handler, three things are checked: Whether the UAV has started reporting information, including its location, whether any handsets have connected to OpenBTS, and whether a target area has been defined. If all these are true, the system enters the `TRANSIT` state. The step handler is invoked once per second until the state is changed.

Leaving

Upon leaving the state, the route planning algorithm described in Section 4.6 is run.

4.7.3 The TRANSIT State

The system remains in this state whenever the UAV is travelling from one place to another. This is where localization calculations are done. When the system is "shut down" it stays in the transit state without a destination, to keep running calculations and relaying data to the GUI. The `TRANSIT` step has no handler for leaving the state.

Entry

On entry, the next site from the planned route, if any, is chosen as the destination, and the UAV output module is told to move the UAV to this location. If no more destinations are planned, the UAV will hold position.

Step

In the step state the following actions happen in order:

1. If there are any groups pending in the OpenBTS input module, these are converted to angles of arrival, as described in Section 4.4 and stored in the localization module.

2. If there are any pairs of `AngleOfArrival` instances related to the same IMSI which have not been triangulated, the triangulation is done as described in Section 4.5.

    a) A maximum number of total AoA and triangulation computations to be done in each step can be defined, to limit the amount of time and processor power spent here. If that limit is reached, any pending calculations carry over to the next step.

3. Any located handsets are transmitted to the GUI through the output module.

4. If no destination is set, the system is in shutdown mode, and no further action will be taken in this step.
5. The location of the UAV is received through the input module, and relayed to the GUI
6. If the UAV has reached the destination, it enters the `TURNING` step with parameter $p = 0$.

While in this state, the step handler is run every second.

4.7.4 The TURNING State

This is the simplest state. The system is in this state when it needs to face a specific angle before a reading. It uses a parameter, p, the number of reading sets which has been completed at the current site. It has no leaving handler.

Entry

In this step, the desired facing is calculated as $360°/n_{readings} \times p$. This facing is relayed to the UAV through the output module.

Step

With each step the facing of the UAV is retrieved from the input module, and compared to the desired facing. If it has been reached, the system enters the `READING` state, with parameters $p = p$ and $t = 0$. As turning is relatively fast, this step is run five times per second until the target is reached.

4.7.5 The READING State

The reading state is when the system receives data from the OpenBTS Physical Channel API. It uses two parameters, p, which as in the turning state is the number of reading sets which have been completed at the current site, and t which is the duration in which the system has been in this state. No handler for leaving the state is specified.

Entry

On entry, the OpenBTS CLI is queried for connected handsets. A specified number of these are picked out for reading, since "thousands" of handsets can be connected at a time, but only "hundreds" can have ongoing communication (more testing is needed to find exact numbers) [17]. An SMS is sent to the selected handsets.

Step

This step is run two times per second, as that is the approximate frequency with which the OpenBTS API publishes messages. During the step, `PhysicalAPIReading` instances are generated from the input module, and added to the `PhysicalAPIManager` as described in Section 4.3. Then, the system checks if $t >= t_{reading}$, i.e. if the system has been in the reading state for the time designated per reading. If not, the system stays in this state, but increments t by 0.5 before the next step.

If the system has been in this state for long enough, two things can happen. If $p + 1 >= n_{readings}$, all readings at this site are done, and the system enters the `TRANSIT` step. If not, it returns to the `TURNING` step, with parameter $p = p + 1$.

4.8 User Interface

As described in Section 4.2 the GUI communication is handled by a websocket server. In the prototype, the interface is a website utilizing the Google Maps API to show the position of the UAV as well as the located handsets. Handsets are shown as semi-transparent blue circles with radius equal to the maximum error of the triangulation. If handsets overlap the colour intensifies within this overlap. An example of the interface in action can be seen in Figure 4.12. In the picture, the red pin is the UAV and the blue circles are located handsets. Below the map is a log showing which handsets have updated positions. Seeing as the interface is completely decoupled from the rest of the system, updating the look or changing which data is displayed is very easy, the interface in this report is mainly developed for demonstration purposes.

A simple message protocol was designed for the messages from the control system to the GUI. Each message is in the form `<id>:<type>:<payload>`, where `<id>` is a number unique to each instance of the control system if the message relates specifically to that instance, for example location updates. If the message concerns the system in general, for example if it is a list of located handsets, the `<id>` is left out. `<type>` is simply the textual representation of the message type, and `<payload>` varies with each type. Following is an explanation of the four types of messages which are implemented, and an example of each can be seen in Listing 4.12 as a Python string.

- `LOCATION` update messages, sent by a UAV in transit so that it can be tracked on the GUI. The payload of this message is `<lat>,<lng>`.
- `IMSIS` messages. The control system sends a list of each IMSI connected to that specific system. The payload is a comma separated list of IMSIs.

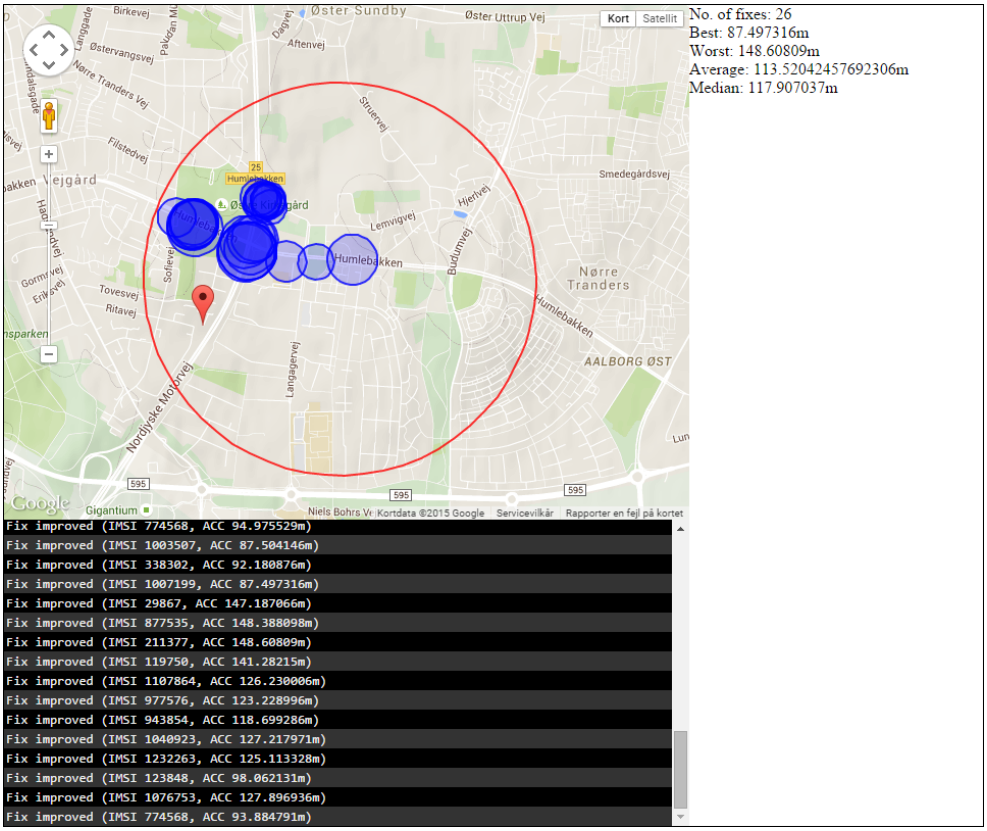


Figure 4.12: Graphical User Interface

- `TARGET` messages contain information about the target search area for a UAV. The payload of this message type is in the form `<center-lat>,<center-lng>,<radius>`.
- `FIXES` messages. These messages are the most important, as they relay information about located handsets. The payload of these messages is a comma separated list of fixes, where each fix is in the form `<imsi>;<lat>;<lng>;<accuracy>`. No `<id>` is sent with these messages, as it does not matter which UAV computed the fix. If it becomes important, an extension to the GUI could compare the IMSIs to those received from each system in an `IMSIS` message.

```
1 '1:LOCATION:57.029308,9.979909'
2 '1:IMSIS:348576348756,484567912832'
3 '1:TARGET:57.029308,9.979909,500'
4 ':FIXES:348576348756;57.029444;9.979301;45.1,' +
5         '484567912832;57.028000;9.970101;60.7'
```

Listing 4.12: GUI Message Examples

Chapter 5

Testing

In this chapter the testing of our solution is described along with results.

5.1 Simulation

Given that testing in real life with the current hardware is not doable, a software simulation was developed. The simulation consists of a testing environment representing the disaster area. This environment is populated with handset clusters, i.e. a random number of handsets (currently a maximum of 10) located at random coordinates within a certain distance (maximum 50m) from a randomly selected centre of the cluster. The testing environment also contains a UAV, of course.

The environment is constructed to act as everything outside of the UAV control system. It simulates OpenBTS by generating Physical Channel API messages for the handsets connected to the BTS and sending SMS and it moves the UAV around in the simulated world.

The class `World` handles the outer environment factors such as size of the site. Instances are initialized with the following attributes:

- Width and height, which are both set to the global constant `ENVIRONMENT_SIZE`.
- A base noise level for the area, set to the global constant `ENVIRONMENT_NOISE`.
- A list of handsets.
- A list of UAVs.
- A ZeroMQ publisher for information from OpenBTS.
- A ZeroMQ publisher for information about the UAV.
- A ZeroMQ subscriber for the messages from the UAV control system.

- A tangent plane centred on the area, for conversion between Cartesian coordinates and latitude and longitude.

The implementation of the handsets is done in a class called `MobileHandset`. The handsets have the following attributes:

- Location stored in Cartesian x and y coordinates
- Power level which is randomly set to between 0.5 and 1.5 upon instantiation
- An IMSI which is randomly generated.

The UAV is implemented in a class appropriately named `UAV` and has the following attributes:

- Location in the same way as the `MobileHandset` class.
- Battery in %.
- A facing in degrees.
- A speed in m/s.
- A destination in Cartesian coordinates, which is initially the same as the current location.
- A target facing which is initially the current facing.
- A list of connected handsets.
- A control system.

Finally, a small class called `SpoofCLI` simulates the OpenBTS CLI, it has functions to return the noise level of the environment, return an IMSI list and send an SMS. While no actual SMS is sent it tells the environment to generate readings for the target handset for a number of seconds corresponding to the time there would be traffic between the BTS and the handset in the real world if an SMS was sent.

5.1.1 Updating the Simulation

The simulation is run via the `update` function, which is called every 0.5 seconds. The function can be seen in Listing 5.1

```
1 class World(object):
2 #...
3   def update(self):
4     #get controller messages and set uav target destination/facing
5     while True:
6       message = self.uav_input.poll()
```

```
7         if message is None:
8           break
9
10        if message['type'] == 'move':
11          #convert to cartesian coordinates
12          latlng = LatLng(message['data']['lat'],
13                          message['data']['lng'])
14          self.uavs[0].destination_x, self.uavs[0].destination_y =
15                               self.tangent_plane.to_cartesian(latlng)
16        else:
17          self.uavs[0].target_facing = message['data']
18
19      #update drone locations/turn/generate readings
20      for uav in self.uavs:
21        distance = sqrt((uav.x - uav.destination_x)**2 +
22                        (uav.y - uav.destination_y)**2)
23        if distance > 1:
24          travel = min(distance, 0.5*uav.speed)
25          uav_travel_vector =(Vector.XY(uav.destination_x -
26                               uav.x, uav.destination_y - uav.y))
27          uav.x += cos(radians(uav_travel_vector.direction()))*travel
28          uav.y += sin(radians(uav_travel_vector.direction()))*travel
29          uav.facing = uav_travel_vector.direction()
30
31        #turn drone
32        else:
33          direction_offset = uav.target_facing - uav.facing
34          if direction_offset < -180:
35            direction_offset += 360.0
36          elif direction_offset > 180:
37            direction_offset -= 360.0
38          if abs(direction_offset) > 2.0:
39            turn = min(abs(direction_offset), UAV_TURNSPEED*0.5)
40            uav.facing += sign(direction_offset)*turn
41            uav.facing %= 360
42
43        #generate readings for phones the UAV is connected to
44        reading_next = []
45        for handset in uav.reading_from:
46          imsi, count = handset
47          reading = self.generate_handset_reading(uav,
48                                                  self.get_handset(imsi),
49                                                  int(time.clock()*1000))
50          if reading is not None:
51            self.openBTS.send_reading(reading)
52          if count > 0:
53            reading_next.append((imsi, count-1))
54        uav.reading_from = reading_next
55
56        #convert from cartesian coordinates to lat/lng
57        latlng = self.tangent_plane.from_cartesian(uav.x, uav.y, 2)
58        #publish updated UAV information
59        lat, lng = latlng.degrees()
```

```
60        self.uav_output.send_data(uav.battery, lat, lng, uav.facing)
```

Listing 5.1: The `update` Function

First, the incoming messages from the UAV's control system are handled in the loop starting on line 3. Only the most recent messages are used, the loop goes through the message queue until it is empty and updates either destination coordinates (in case of a `move` message) or target facing (in case of a `turn` message) for the UAV. Once the message queue is empty each UAV is updated with a new location, calculated from the speed of the UAV, if the UAV is in transit (i.e. destination and location are not the same, within a certain margin of inaccuracy). If the UAV is currently turning (i.e. target facing and current facing are not the same within a margin of error) the facing is updated. Starting on line 42 any OpenBTS readings needed are generated and finally the new location of the UAV is published to the control system of the UAV.

5.1.2 Generating Physical Channel API Messages

As mentioned in Section 4.3 readings from OpenBTS Physical API is not the same in the implementation as it is in the text of this report. This function generates a single OpenBTS Physical API message. This is done by calculating what the `RSSP` value would be for a given handset in relation to where the UAV is located and facing when taking the reading. The attenuation model used is described in Section 4.4. Seeing as this is a simulation only the important parts of the message are generated. The implementation can be seen in Listing 5.2.

```
1  class World(object):
2  #...
3    def generate_handset_reading(self, uav, handset, timestamp):
4      if handset is None:
5        return
6      #Vectors for UAV and handset
7      uav_handset_vector = Vector.XY((handset.x - uav.x),
8                                     (handset.y - uav.y))
9      uav_facing_vector = Vector.DM(uav.facing, 1)
10
11     #Find angle between UAV and handset vector
12     direction_offset = abs(uav_handset_vector.direction() -
13                            uav_facing_vector.direction())
14     if direction_offset > 180:
15         direction_offset -= 360
16     if direction_offset < -180:
17         direction_offset += 360
18
19     #Calculate RSSP based on dir. of antenna relative to handset
20     attenuation = generate_attenuation(direction_offset)
21     rssp = handset.base_rssp() - attenuation
22     rssp = max(rssp, self.noise)
23
```

```
24      #Generate Physical API reading data
25      return json.dumps(
26      {
27       "data":
28       {
29        "channel":
30        {
31         "IMSI": str(handset.imsi)
32        },
33         "burst":
34         {
35          "RSSP": rssp,
36          "actualMSTimingAdvance": uav_handset_vector.magnitude()/550,
37          "actualMSPower": ACTUAL_MS_POWER
38         }
39       },
40       "timestamp": str(timestamp),
41      })
```

Listing 5.2: Generating a Physical Channel API Message

First, the functions checks that there is an actual handset to generate a message for. Then, starting on line 5 a vector between the handset and the UAV as well as a unit vector representing the facing of the UAV is created. The angle between these two is then calculated in order to generate a proper attenuation for the signal, which is used in lines 16-18 where the `RSSP` value is calculated. Finally, the JSON object is created and returned.

5.1.3 Limitations

The simulation, of course, is not a perfect replacement for the real world. The scenario the environment describes is the absolute optimal scenario, which makes sense given that this is a first prototype and thus if it does not work under optimal conditions, testing under any other condition is pointless. The limitations of the simulation are as follows:

- The UAV accelerates and decelerates instantly in the simulation. This could be significant in relation to how fast we can reach a target location as well as impact when readings begin being taken.
- The readings generated in the environment are "perfect", i.e. they never deviate from the mathematical model of signal attenuation we have developed. The simulated UAV also picks up readings directly beneath it, where a real world UAV would not pick these up as the directionality of the receiving antenna does not allow it.
- There is no risk of signal loss for the handsets or the BTS. The environment presents absolutely optimal circumstances for the connection and thus the handling of lost connections cannot be tested in this simulation.

- There is no risk of multipath signals, which might affect the accuracy of the localization.
- OpenBTS does not run, it is spoofed in the simulation, which means that anything that is dependent on OpenBTS implementation is not tested.
- The environment is the perfect area for navigating the UAV, there are no physical obstacles of any kind, no wind etc.
- The noise level of the area is consistent in the entire area, there are no pockets of lower or higher background noise which could impact the signal strength and thus the readings.

5.2 Tests and Results

The accuracy of the system has been tested with the following setups:

1. Area radius of 500 metres, 5 measurement sites, 8 measurements per site (requires minimum flight time of 12 minutes)
2. Area radius of 1000 metres, 5 measurement sites, 8 measurements per site (requires minimum flight time of 20 minutes)

The simulation environments have each time been populated with 4 clusters of a random number (maximum 10) of handsets, placed randomly within environment. Only one UAV has been deployed. The simulation was run 5 times for each setup and the best, worst, average and median error recorded.

5.2.1 Simulation with 500 Metre Area Radius

The results of this simulation setup can be seen in Figure 5.1. The best, i.e. the lowest, error measurements are fairly consistent, all falling within 0.5 metres of each other. The worst cases are slightly more erratic falling within 8.74 metres of each other. This indicates that while some handset locations are quite bad for localization, all the good locations are equally good. The average measurements are also very consistent, at least with a higher number of handsets. The last simulation run is somewhat of an outlier with few connected handsets and a markedly higher worst, average and median error. However, again, the best case is consistent with the other simulations, which indicates that 44m is the best case accuracy for this simulation setup. It is also noteworthy that even in the last simulation where the average error is roughly 60m. the search radius has gone down from $500m^2$ to $60m^2$, an 88% decrease in search radius and a 98.56% decrease in search area for each handset.

	Simulation Index				
Measurement	1	2	3	4	5
Handsets #	20	22	33	20	13
Best (m)	43.88	43.79	44.26	43.74	44.08
Worst (m)	70.88	62.14	66.01	66.10	67.55
Average (m)	50.43	50.73	49.68	49.34	57.78
Median (m)	46.36	48.34	47.98	47.57	66.02

Figure 5.1: Results of Simulations with 500m Area Radius

	Simulation Index				
Measurement	1	2	3	4	5
Handsets #	19	23	20	15	29
Best (m)	87.52	87.53	88.56	90.02	87.73
Worst (m)	97.10	143.38	129.57	145.58	129.74
Average (m)	91.72	106.98	104.10	111.58	98.73
Median (m)	92.09	98.88	100.34	97.19	96.57

Figure 5.2: Results of Simulations with 1000m Area Radius

5.2.2 Simulation with 1000 Metre Area Radius

The results of this simulation setup can be seen in Figure 5.2. As can be expected, when the area is quadrupled but the number of measurement sites stay the same, the accuracy decreases. Compared to the results in Figure 5.1 the error has roughly doubled for all measurements. Again, the best measurements are fairly consistent, falling within 2.5 metres of each other while the worst are erratic (48.5 metres difference between lowest and highest worst measurement). This is consistent with the "good locations are all equally good" sentiment. For a search area with a radius of 1000m the average error of a handset location is roughly 105m. This is a decrease in search radius of 89.5% and a decrease in search area of 98.9% for each handset.

Chapter 6

Reflection

Here, we reflect on the final product, its limitations and detail what we consider the most important future improvements.

6.1 Scalability

With the base functionality implemented, this prototype is now working as intended, as seen in Section 5.2. With the functionality in place we now present our arguments for the scalability of our solution:

- To decrease the time spent on locating a single handset, bearing in mind that the directional antenna completely loses connection to the handset once the handset is outside of its directionality, the system could, once this happens, start generating "fake" Physical API messages with the RSSP set to the noise level of the area. This will allow the system to communicate with more handsets at a time, seeing as the faked readings do not take up time slots on the frequency channel.
- If there are too many handsets for a single BTS in a search area, multiple UAVs could be deployed and the system should then divide the handsets between the UAVs. This could be done by giving each instance of the system an ID as well as the total number of deployed system instances, n. Since each handset has a unique numerical IMSI each system would only handle handsets where $IMSI$ modulo $n = ID$.
- An observation made in Section 5.2, namely that the accuracy is proportional to the size of the area, means that while the system may be less accurate in large areas, the reduction in percentage of the search area remains the same. This means that, with the right UAV, this system could be deployed in very large areas and still be useful.
- To increase accuracy two systems could be deployed on two UAVs and produce readings for the same handsets. This would give the same effect

as having more reading sites for a single system, but would be possible with UAVs with shorter flight times. The systems could share the handsets by utilizing the handover procedure native to the GSM standard. However, this would require a route planning algorithm that takes the number of UAVs into account.

While none of the methods mentioned are tested as of yet, if one or more of these are feasible the system could easily scale to large areas as well as the collaboration of several UAVs.

6.2 Limitations

Given that this is a prototype project, several limitations to both the prototype implementation and the methods used exist. Below, we describe these and their impact on the system.

- A major limiting factor at this point is the flight time of present UAVs given the payload presented by the hardware. Flight time directly affects how many measuring sites the UAV can reach and this is a deciding factor in the accuracy of the measurements. Modern UAVs such as the Huginn X1, developed by Sky-Watch A/S, have a flight time of approximately 25 minutes without payload, but given the weight of the hardware (500g without outer casing), this time will be reduced by a significant amount [18]. As such reducing the weight of the payload would directly improve the accuracy of the system. Given that the cables supplied with the hardware are quite long this is definitely an option.

- In the simulation developed for the testing of the prototype, as well as in the preliminary tests of the hardware equipment there is a direct LoS between the handset and the antennae. Our sources indicate that this impacts the accuracy of the measurements greatly, however our research has not turned up any AoA applications where measurements are taken from high in the air. This means that the AoA measurements we have may not have realistic accuracy, however in order to determine this more testing is required.

- On the topic of AoA it should also be noted that the RSSP in the current simulation is based on a model developed without extensive tests. In the real world directional readings may not behave as the model suggests, which could impact AoA computations. In that case, it would especially affect those where no readings are made with the directional antenna pointing directly at the handset.

- As mentioned, we did not have the hardware necessary to implement the revised hardware architecture, thus the system has only been tested in simulation. For the revised hardware architecture the two USRPs need to be synchronized and a modification to OpenBTS has to be made to allow the system to generate Physical API readings with data from one USRP and RSSP from the other. Given that we did not have the hardware, we have not looked into the complexity of this modification though we do know that clock synchronization of the USRPs is possible on a hardware level.

- Currently the GUI requires an Internet connection in order to receive updates from the system. This communication could possibly be converted to GSM data traffic as suggested in Section 4.2. Another possibility is transmitting the data through the control channel that the UAV uses to communicate with its handler.

- The system is not tested on the limited computer power that will be available to it when deployed, however preliminary tests of our system suggests that it will not be a problem. As well, OpenBTS is already running on small computers in other applications. One important issue in this case is that the computer will handle signal processing for not one, but two USRPs which will present a significantly larger load on the processor. However, lightweight parallel computers exist that could possibly take the place of the BeagleBone if necessary. One such computer is the Parallella board [19].

- Naturally there is a limit to how many handsets can be connected and communicating via a single BTS at any one time, however we only have anecdotal evidence for what this upper limit is. Determining this is subject to further testing, but we have not had enough handsets available to perform this load test ourselves.

- The simulation developed is of course a very limited representation of the real world and the list of its limitations mentioned in Section 5.1.3 and in previous items in this section may impact the accuracy greatly. However, even with some loss of accuracy when applied to a real world scenario, the simulation has a 98% decrease in search area which is significant enough that we hold this as a feasible solution.

6.3 Future Work

The developed product is, of course, only a first prototype of the system. For future development we would perform the following improvements:

- The most important future task is to assemble the hardware architecture and make the required change to the OpenBTS system, such that the system is testable in a real world setting.
- Currently the system is only operating with a simulated UAV so the input and output modules are not consistent with how communication with a real UAV would be. As such the implementation of these modules should be changed to reflect an actual UAV API.
- Change the communication between the system and the GUI to allow data transfer while the UAV is in operation.
- Testing of the current implementation of the system without direct LoS to the handsets should be performed in order to determine if AoA is a feasible technique in the context of USAR.
- Implement and test the methods described in Section 6.1 in order to determine if this system is indeed scalable to the degree that we expect.
- Test, and if necessary refine, the model described for RSSP with relation to the directionality of the antenna to ensure that it is consistent with the real world.
- Perform load test of OpenBTS to determine how many handsets can be connected and communicating concurrently. This is important to determine how many instances of the system are needed for a disaster area.
- Decrease the weight of the system as much as possible in order to maximize flight time. As mentioned in Section 6.2 this is vital to the accuracy of the localization. This could be done by using lightweight cables and exploring alternative hardware options.
- Experiment with other localization techniques (e.g. TDoA[1]) to determine if AoA provides the best accuracy. The reason AoA is the only method tested during this project is that other techniques require a much greater familiarity with digital signal processing.
- To ensure that all handsets in the disaster area connect to the DronePhoneHome module a GSM jammer should be developed. The jammer should identify any GSM networks still operational in the area and prevent communication with these.
- The GUI should be implemented with functionality to allow the USAR worker to ban rescue workers' handsets from the DronePhoneHome network. This is done to ensure that localization is only performed on the relevant handsets in the area.

[1]Time Difference of Arrival

- The system should be modified to support manual control of the UAV and still be able to operate as intended.
- The connected handsets should be filtered by their timing advance so that localization is not attempted on the handsets that are far outside the search area of the UAV.
- For improved performance the system should be implemented in C++ or another compiled language.

6.4 Conclusion

We set out to develop a module for use with a UAV to aid USAR workers, by locating people buried in debris, through GSM localization of their cell phones. We designed a hardware architecture for such a module, and even though we did not have access to *all* the components necessary to assemble this architecture, we had access to enough that we were able to perform preliminary tests enabling us to reason about the effectiveness of this architecture.

For the system we designed and implemented several software modules necessary for its operation, namely:

- A geometry module allowing us to convert between Euclidean and Earth geometry.
- An input and an output module for communicating with other systems, such as the UAV, OpenBTS and a GUI.
- A localization module for calculating the locations of cell phones through the use of AoA and triangulation, as well as planning a route for the UAV to follow.

Finally we implemented a control system to tie these modules together and ensure correct behaviour of the system.

We made several theoretical contributions as well, namely mathematical reasoning about the effect of attempting triangulation with inaccurate angle measurements, as well as a heuristic route planning algorithm, including ways to compute the validity and quality of a given route.

As we did not have the hardware to test the system under realistic conditions, we developed a software simulation using models based on our testing, which emulates the environment in which the system should be deployed, as well as the input the system might receive during operations. Additionally we created a prototype of the GUI that an operator might be presented with.

In these simulated conditions, we achieved a best case accuracy of 44 metres in our localization. In the average case, the accuracy was one tenth of the radius of the area covered. This amounts to an area of around 1.5% of the total area.

Finally, we reflected upon the limitations and scalability of the developed system, and presented suggestions on how development can be continued, first to make the system usable in a real world scenario, as well as improve the accuracy.

All in all, the problem statement described in the introduction of this report was fulfilled.

Bibliography

[1] H. Rodriguez, W.A. Anderson, P.J. Kennedy, E.L. Quarantelli, E. Ressler, and R. Dynes. *Handbook of Disaster Research*. Handbooks of Sociology and Social Research. Springer, 2009.

[2] Barbara Flindt Jeppe Tarp. Gsm localization in diaster scenarios, 2015.

[3] Gsm architecture. `http://www.radio-electronics.com/info/cellulartelecomms/gsm_technical/gsm_architecture.php`, cited 2014.

[4] ETSI. Etsi ts 100 910. `http://www.etsi.org/deliver/etsi_ts/100900_100999/100910/08.20.00_60/ts_100910v082000p.pdf`, 2005.

[5] CartouCHe. Positioning. `http://www.e-cartouche.ch/content_reg/cartouche/LBStech/en/html/LBStechU2_poslabel1.html`, cited 2015.

[6] S. Zorn, R. Rose, A. Goetz, and R. Weigel. A novel technique for mobile phone localization for search and rescue applications. In *Indoor Positioning and Indoor Navigation (IPIN), 2010 International Conference on*, pages 1–4, Sept 2010.

[7] Ettus Research. Usrp-users mailing list. `http://lists.ettus.com/pipermail/usrp-users_lists.ettus.com/`, cited 2015.

[8] Ettus Research. Usrp b200-b210 spec sheet. `http://www.ettus.com/content/files/kb/b200-b210_spec_sheet.pdf`, cited 2015.

[9] Wikipedia. Log-periodic antenna. `http://en.wikipedia.org/wiki/Log-periodic_antenna`, cited 2015.

[10] Sukru B. Bilgin Halil I. Gok Tayfun Nesimoglu Volkan Turgul, Meltem Dirim. Broadband signal search and direction finding at uhf frequencies. `http://www.academia.edu/2448501/Broadband_signal_search_and_direction_finding_at_UHF_frequencies`, 2010.

[11] Ivan S. Ashcraft. Projecting an arbitrary latitude and longitude onto a tangent plane. Technical Report MERS 99-04, Brigham Young University, 459 Clyde Building, Provo, Utah 84502, January 1999.

[12] iMatix Corporation. http://zeromq.org/, cited 2015.

[13] Robert Bridson. Fast poisson disk sampling in arbitrary dimensions. http://www.cs.ubc.ca/~rbridson/docs/bridson-siggraph07-poissondisk.pdf, cited 2015.

[14] Jason Davies. Poisson-disc sampling. https://www.jasondavies.com/poisson-disc/, cited 2015.

[15] Wikipedia. Traveling salesman problem. http://en.wikipedia.org/wiki/Travelling_salesman_problem, cited 2015.

[16] Wikipedia. Command pattern. http://en.wikipedia.org/wiki/Command_pattern, cited 2015.

[17] Various authors. Openbts-discuss mailing list. http://sourceforge.net/p/openbts/mailman/openbts-discuss/, cited 2015.

[18] Anthea Technologies. Huginn x1. http://www.antheatechnologies.com/sky-watch-huginn-x1/huginn-x1-information/product-information.aspx, cited 2015.

[19] Parallella. Parallella - the board. https://www.parallella.org/board/, cited 2015.