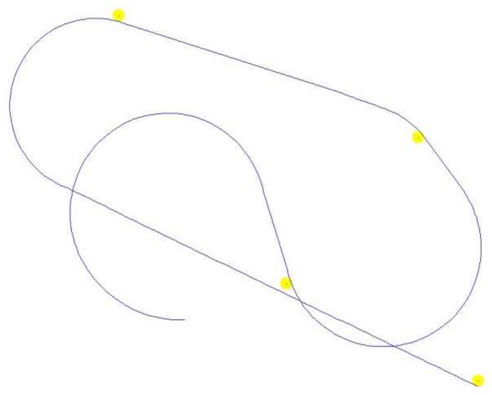
Thesis Report, November 2009

Navigation Control of an Unmanned Aerial Vehicle (UAV)

Bachelor Thesis in

Computer and Electrical Engineering

Tae Hyun Kim & Denny Toazza



School of Information Science, Computer and Electrical Engineering

Halmstad University

Navigation Control of an Unmanned Aerial Vehicle (UAV)

School of Information Science, Computer and Electrical Engineering
Halmstad University
Box 823, S-301 18 Halmstad, Sweden

November 2009

Acknowledgement

Success is not final, failure is not fatal:
it is the courage to continue that counts.
Winston Churchill

This thesis was developed in the context of a project that aimed at the assembling of an Unmanned Aerial Vehicle (UAV). This project was divided in two parts; hardware and software. One group took the responsibility for the development of the autopilot, stabilization

and assembling the plane, which includes the entire software-hardware interface. The other group is responsible for navigation software system to let UAV fly autonomously, taking GPS coordinates and waypoints destinations as input data. This thesis presents the work and the results related to the second part of the project.

First of all, we would like to thank our supervisors, Edison Pignaton de Freitas and Kristoffer Lidström for their technical advices and feedback during this thesis. We also want to thank the other group, Ugur Bozkurt and Mustafa Aslan for their work and collaboration. Special thanks to Juan Antonio Brena Moral for assistance in research and great support

In addition, special thanks to our family.

Details

Authors:	Tae Hyun Kim & Denny Toazza
University:	Halmstad University, Sweden
Degree Program:	Computer Science and Engineering
Title of Thesis:	Navigation Control of an Unmanned Aerial Vehicle (UAV)
Supervisor:	MSc Edison Pignaton de Freitas,
	Lic D Kristoffer Lidström

Abstract

The thesis covers a new navigation algorithm for UAV to fly through several given GPS coordinates without any human interference. The UAV first gets its current position from GPS receiver via Bluetooth connection with the navigator computer. With this GPS point, it draws an optimal trajectory to next destination. During the flight, the navigator computer issues the information about which direction to turn and how much to turn. This information will be used to steer the airplane servos.

The algorithm is programmed in Java LeJOS. It uses built-in Java classes about GPS and Bluetooth. The main computer, where the navigation program runs, is a LEGO Mindstorms NXT and it is used a GPSlim240 from HOLUX as a GPS receiver.

Contents

1 Introduction

1.1 Background

A long time ago, when the humans started to travel around the world in order to find food, water and other goods, people needed to navigate in order to find their path. Previously guided by the stars and nowadays by GPS, the goal is the same, find the way to a given destination point.

In the last past years, the requirements to find new ways to transport and to navigate increased with the demands from the end users. Technology advances in electronics and computation help the creation of new transports facilities, which besides addressing the mentioned demands, are making possible the development of autonomous systems that are capable to drive themselves without human intervention.

Taking the above mentioned context, this thesis focuses in the navigation control of a Unmanned Aerail Vehicle system. The main idea of the thesis is provide the navigation capability to an automated aircraft platform, which may allow a fully autonomous level flight from an original point to a destination, passing by a given number of waypoints.

The GPS data is virtually accessible from anywhere around the world. The basic idea of this thesis is to implement a navigation mechanism to the UAV based on received GPS coordinates. The GPS data, which is collected from a GPS device Holux GPSlim240, is separated in different NMEA packages and is sent by Bluetooth interface to be used by the navigation algorithm implemented in LEJOS and running on a Lego Mindstorms.

The work is basically divided in parts:

- Explanation of the used algorithm;
- Computer-based simulations; and
- Experiments performed using Lego Mindstorms.

1.2 Related Work

The main field of UAV's started to grow in the middle of nineties, when the used components started to be smaller, which made it possible to build a plane light and small with almost all the flight capabilities of a normal airplane. After this evolution, in a short space of time, governments, companies and universities started to develop several projects in this area. However, it is noteworthy to highlight that in the larger sense, the idea of UAVs come from the middle of the eighteenth century, when the Austrians used unmanned balloons load with explosives to attack the Italian city of Venice. Another remarkable step in the evolution of the UAVs was during the Second World War, when first “intelligent” rockets were used by the Nazi and Allies forces. Other important steps in the history of the evolution of the UAVs followed the history of the wars until nowadays, as the main usage of this technology has been in the military area. [1]

Some related works were selected presenting some sophisticated UAVs' project. The list of these projects is presented as follows:

- RC Car project has a point in common with this work in the sense that it also uses GPS coordinates to arrive at the destination points and also uses the same GPS device and the LEGO Mindstorms used in this project. [2]
- The project LEGO UAV 2 uses LEGO and is the first UAV built with LEGO Mindstorms.[3]

- The MIT/Draper Autonomous Helicopter Project is a project based in a helicopter, which uses a DGPS device to get an almost perfect position. This project is one of the first to implement a UAV.[4]

1.3 Limitation

This project deals with GPS coordinates. As well known, GPS information is transmitted from several satellites. If the GPS receiver is situated around several high buildings or if the weather is cloudy or rainy, the information is likely to have noise. This is checked by the experiment described in the Section 3.2.1

1.4 Outline

This thesis is organized as follows:

Chapter 2 (Methods): The methods to achieve the goals of this thesis are presented. Additionally, possible errors conditions are described, as well as the proposed solutions for them.

Chapter 3 (Simulation and Experimentation): With the methods described in Chapter 2, simulations performed in PC and experimentation with real GPS coordinates are explained in detail. Moreover, experiments with the final implementation in the Lego Mindstorms NXT are also presented.

Chapter4 (Conclusion and Future Work): This chapter summarizes what have being done in this project and discusses about future works that can be derived from this thesis.

2 Methods

2.1 Introduction

2.1.1 LeJOS NXT Project

LeJOS project is an open source application created to facilitate the process to build new robots with Lego Mindstorms using Java technology.

In this project, the release 1.8 is used. With this release, LeJOS provides means to use Bluetooth interface, Math classes, Java types including float, long and string, preemptive threads and arrays. All these facilities, which come in the LeJOS API, provide a program environment that makes it easier to implement the software solution for the problem that this thesis has the purpose to solve, the autonomous navigation of a UAV.

2.1.2 Candidate Algorithms

Before starting programming the navigation software, there were several algorithm candidates that were considered. Two of them were judged more suitable for the purposes of this thesis:

- Read all the destination coordinates. Draw the best trajectory passing by all the points, and then follow this path; and
- Check the next waypoint. Depending on UAV's position, fly straight or turn with constant angle. When it gets there, perform the same process toward the next destination.

The second option was taken first because it is more flexible than the first in various subjected to errors, providing a more robust solution, and also because it is less complicated to implement than the first one.

2.1.3 Intelligent UAV Algorithm

The main algorithm in the project is composed by the following steps:

A. Read current GPS position from the GPS receiver

B. If UAV passes the proximate circle, continue to the next point and go to **A**. If not, go to **C**.

C. If UAV is on the direction to the next point, go straight and go to **A**. If not, Turn left or right a given degree and go to **A**.

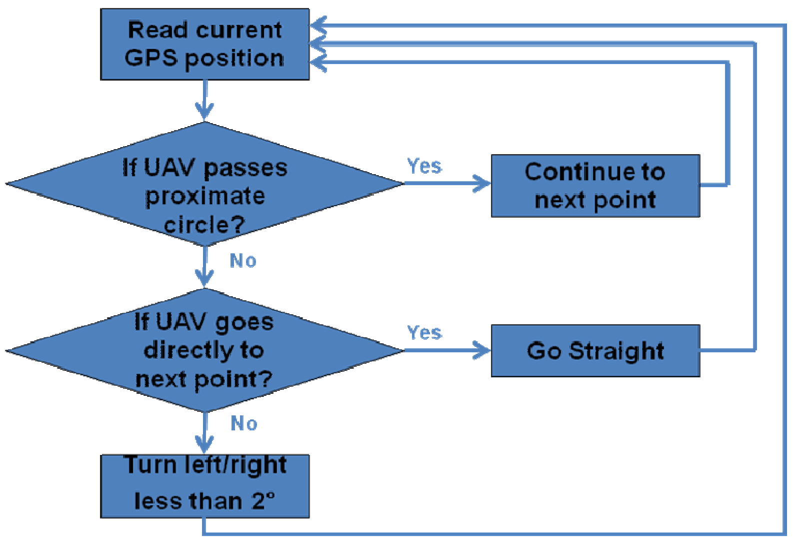

Read current GPS position
If UAV passes proximate circle?
Yes
Continue to next point
No
If UAV goes directly to next point?
Yes
Go Straight
No
Turn left/right less than 2°

Figure 2.1: Algorithm flow chart

The mentioned proximate circle is a virtual circle around the destination with assigned radius. The reason why this concept is necessary is that the UAV hardly passes exactly by the next point, even if it flies with a very accurate algorithm and GPS. Therefore, if the UAV gets through the proximate circle, the program considers the plane meets the goal and proceeds to the next step.

2.2 Methods

2.2.1 One Waypoint Passing Verification

First of all, to check the UAV’s position in relation to the next coordinate, the program calculates the distance between the current point and the next destination. Every second, this distance is measured and if it is shorter than the radius of the proximate circle, that is, if the UAV is in the proximate circle, it flies to another next point.

2.2.2 Virtual Future Waypoint

The next step is to check whether UAV is flying directly to the next coordinate or not. The reason why this process is needed is that if it is not the case, the plane should turn left or right in order to adjust its movement towards the next waypoint. Virtual future point helps to find out where the UAV is heading.

At this step, the following information is known: 1) current coordinate; 2) destination coordinate; and 3) the absolute angle of the UAV’s direction (Figure 2.2). From the two points, the distance can be inferred. When this distance is calculated with the absolute angle, virtual future point is made. If it is not in the proximate circle, UAV has to turn.

2.2.3 Turning Direction

In order to know which direction to turn, absolute angle of the current direction and destination is needed. This is illustrated in Figure 2.2.

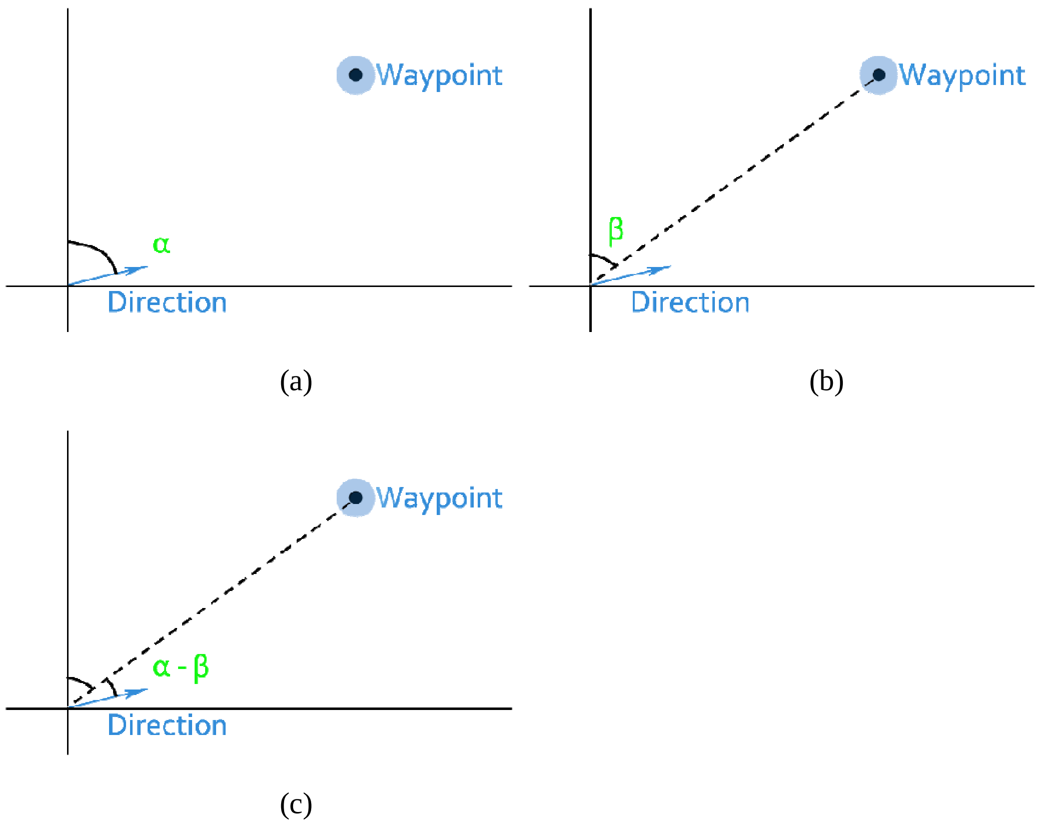
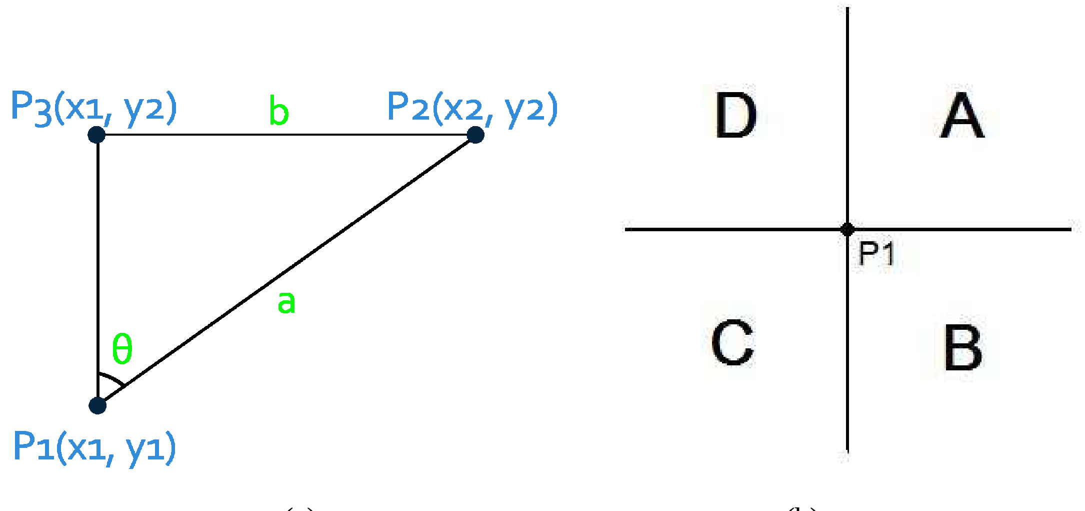


Figure 2.2: (a) Absolute angle of the current direction. (b) Absolute angle of the destination. (c) Difference between (a) and (b).

To get the absolute angle, first of all, the relative angle should be calculated and then, absolute angle can be derived from it. Supposing that the previous point is P1 (x1, y1) and current point is P2 (x2, y2). Considering Figure 2.3(a), there is another point P3 (x2, y1). It is supposed to be (x1, y2), but because points are GPS coordinates, which means x is latitude and y is longitude, P3 should be (x2, y1). From this triangle, with known cathetus *“b”* and hypotenuse *“a”*, and sine function, the relative angle can be found.



(a) (b)

Figure 2.3: (a) θ is relative angle of the direction from P1 to P2. (b) Quadrant for converting relative angle to absolute one.

This is how to change relative angle to absolute angle.

- If P2 is in the area A in Figure 2.3 (b), absolute angle is same as θ.
- If P2 is in the area B in Figure 2.3 (b), absolute angle is $\pi - \theta$.
- If P2 is in the area C in Figure 2.3 (b), absolute angle is $\pi + \theta$.
- If P2 is in the area D in Figure 2.3 (b), absolute angle is $2 * \pi - \theta$.

Now we know the absolute angle of the difference between the angle of the current direction and the destination.

- If $0 < \alpha - \beta \leq \pi$ or $\alpha - \beta < -\pi$, turn <u>left</u>.
- If $-\pi < \alpha - \beta \leq 0$ or $\alpha - \beta > \pi$, turn <u>right</u>.

2.3 Software Robustness to Undesired Limit Error Conditions

2.3.1 Rotation

The first error is the rotation of the UAV. In some cases, the plane cannot get to the destination and just make a circle around the point. From the simulation, it was found out that the UAV rotates when the next point is inside of the virtual circle, which is made by the plane with constant speed and turning angle. However, when the destination is outside of the circle, UAV does not turn around. This is illustrated in Figure 2.4

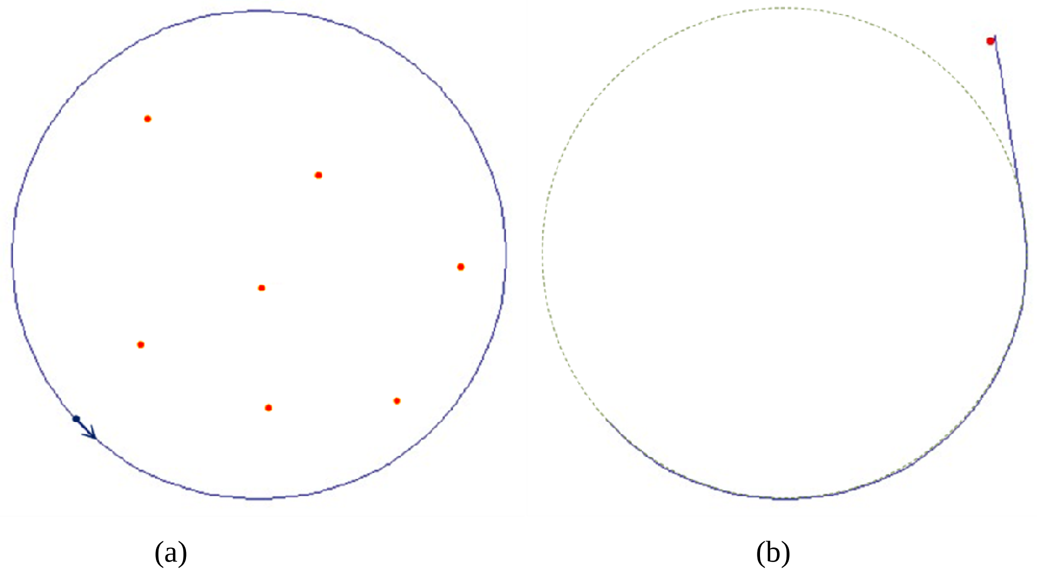
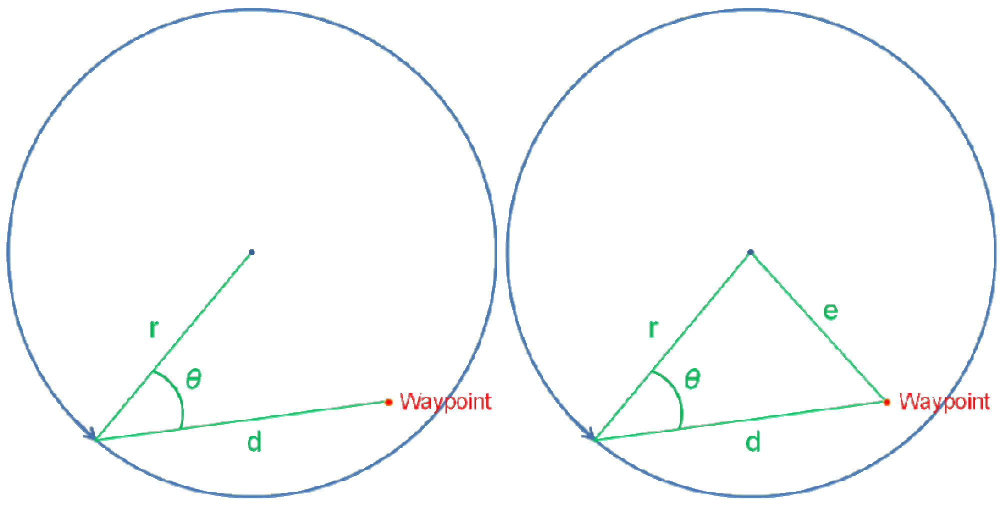


Figure 2.4: In (a), the red points, the destination waypoints, are inside of the virtual circle made by UAV. In (b), the destination is outside of the circle and the plane does not rotate.

In order to see if the waypoint is inside of the circle or not, the length of two sides, *r* and *d*, and the included angle between these sides (Figure 2.5) are needed. *r* is radius, *e* is the distance between the waypoint and the center of the circle, and *d* is the distance between the current position and the waypoint.



(a) (b)

Figure 2.5: In (a), *r* is radius, *d* is the distance between the current position and the waypoint, and *θ* is the included angle between them. In (b), *e* is the distance between the waypoint and the center of the circle.

The radius is derived from this equation:

$$v \times \frac{360°}{\alpha} = 2 \times \pi \times r \tag{2.1}$$

where "*v*" is velocity of the plane, "*α*" is current turning angle, and "*r*" is the radius. Therefore, the radius, "*r*", is,

$$r = \frac{v}{\alpha} \tag{2.2}$$

In order to derive "*θ*" in Figure 2.5(a), the absolute angle of the current direction, "*α*", and the absolute one from the current position to the destination, "β", are needed. If they are bigger than π, what is done is changing them by to the module of their subtraction, subtracted from 2π. Then, the "*θ*" is,

$$\theta = \left| \frac{\pi}{2} - |\alpha - \beta| \right| \tag{2.3}$$

Now “*r*”, “*d*”, and “*θ*” are known. From these three values and the law of cosine, “*e*” can be calculated.

$$e^2 = r^2 + d^2 - 2rd\cos\theta \tag{2.4}$$

If “*e*” is bigger than “*r*”, it means the destination is outside of the virtual circle, and the program works fine. However, if “*e*” is smaller than “*r*”, the UAV should turn more. The solution for this problem is to increase turning angle until the waypoint is outside of virtual circle.

2.3.2 Maximum turn

After improving the rotation error, another problem came out from the simulation. In some cases, to make the destination be inside of circle, the program keep increasing turning angle of UAV. The worst case is shown in Figure 2.6(a).

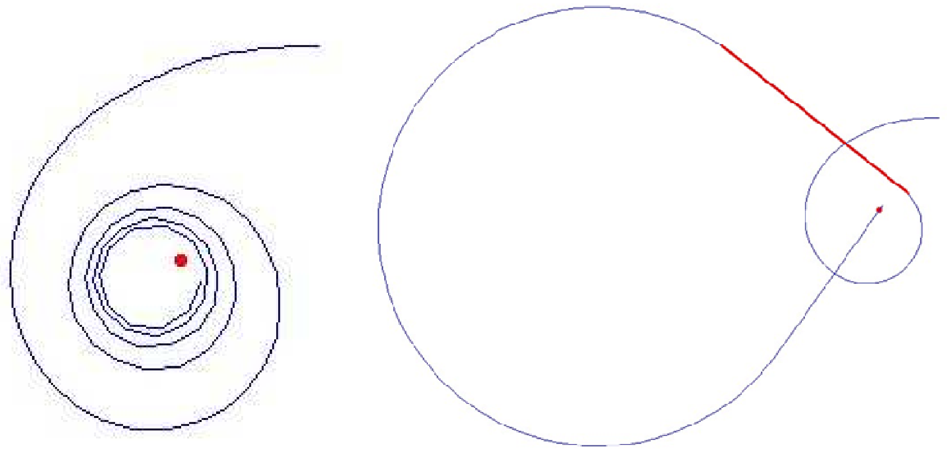


Figure 2.6: (a) One of the worst case. UAV drew a swirl toward the waypoint and the final turning angle was 36°. (b) Small red spot is destination and red line is the section where UAV flies straight.

The final angle in this case was 36°. The plane might fall or crash in the end because of high turning angle. Therefore, it needs a limited value for turning angle. We set the maximum angle as 15° and when the turning angle goes over this threshold, the UAV flies straight forward for 10 seconds and start to turn again with initial value of turning angle. This is illustrated in Figure 2.6 (b).

2.3.3 Zigzag Flight

In the algorithm, when the destination is on the left side of the current direction, the UAV turns to the left and to the right if the destination point is on the right. However, there is some situation that the plane draws a zigzag route around the destination point. This is illustrated in Figure 2.7 (a).

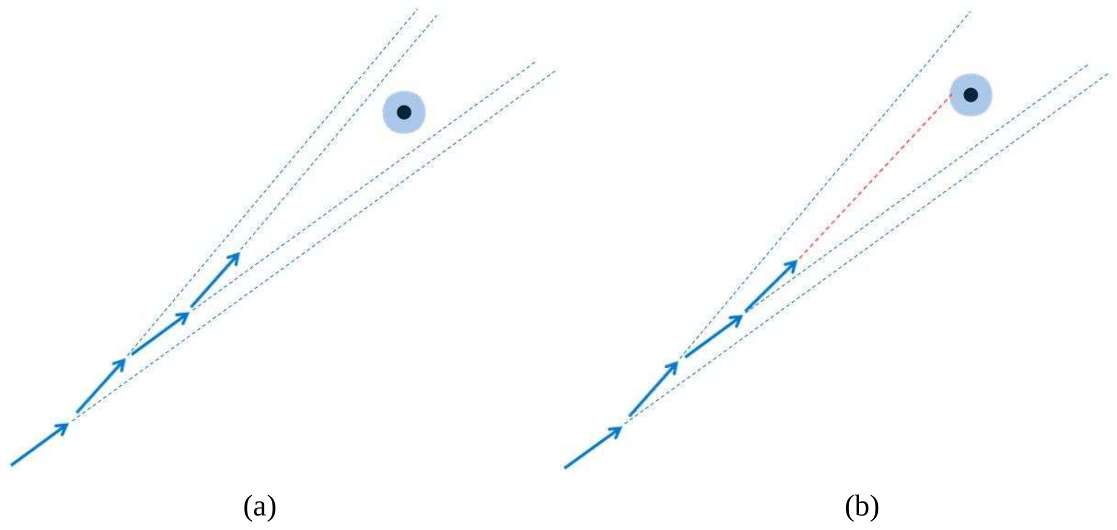


Figure 2.7: (a) Zigzag flight. (b) Solution of this error

The solution for this error is to compare current absolute turning angle with previous one. If they are same, the turning angle should be reduced. In the simulations preformed, it was reduced by 0.5 degree. Figure 2.7 (b) shows an example of a situation in which this problem is corrected. The red dotted line shows the line when the UAV manage to escape from the zigzag problem.

3

Simulation and Experiment

3.1 Simulation in PC with Integer

3.1.1 Basic Configuration

In this simulation, coordination data is not from GPS. Instead of GPS coordinates, integer coordinates are used in ‘double’ type. After running this program, these points are displayed in “0.00” type to check its track.

To show the UAV’s trajectory, a frame is used. It has the resolution of 1280 X 770, of which zero point is moved to (300, 200). After each movement, the program draws the line from previous point to current point every second. This makes the plane’s trajectory a continuous line.

There are 4 destinations; dest1 (280, 60), dest2 (500, 300), dest3 (0, 500), dest4 (600, -100). Around each waypoint, yellow circle, whose radius is 10, is displayed. It means the proximate circle (as explained in Section 2.1.3).

The speed of UAV is set as 10 m/s. Values about turning angle are the same as real flight. The initial turning angle is 3°.

In order to arrive at the maximum turning angle, the formula for a banked turn calculation [5] is used.

$$\frac{mv^2}{r} = mg\tan\theta \tag{3.1}$$

where "*m*" is the mass of the plane, "*v*" is the true airspeed, "*r*" is the radius of the turn, "*g*" is acceleration of gravity, and "*θ*" is current rolling angle. This represents the centrifugal force is the same as the horizontal component of the lift force (shown in Figure 3.1). After simplifying Formula (3.1) with Formula (2.2),

$$v \times \alpha = g\ \tan\theta \tag{3.2}$$

comes out. Here, "*α*" is turning angle.

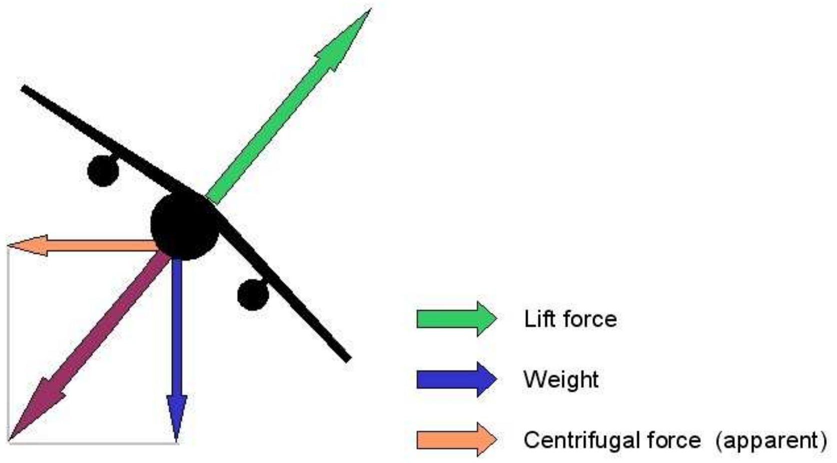


Figure 3.1: Vector diagram showing lift, weight and centripetal force acting on a fixed-wing aircraft during a banked turn. Blue arrow implies the gravity of the plane, *mg*. [6]

In order to find the maximum speed, "*v*", for the plane, Formula (3.3) is applied

Speed = (RPM x Pitch) ÷ 1056 [7] (3.3)

where RPM is 1030 RPM/V[8], battery is 11.1V, and pitch is 4 inches. Therefore, speed is approximately 43.3mph or 19.3m/s. However, this velocity is theoretical value, which means that the propeller is 100% efficient and that there is no loss due to aerodynamic drag. The determination of the real values is remained for the future work.

The rolling angle, "θ", is theoretically from -100° to +100° [9], but with the information provided in [10] and the discussion with the other group drove to a conclusion to use the maximum rolling angle as 30° to avoid crash.

From Formula (3.2), "v" is 19.3m/s, "g" is 9.8m/s^2, and "θ" is 30°. Thus, the maximum turning angle, "α", is 16.8 degree.

3.1.2 Difference from Real Flight

The big difference between real flight and this simulation is the type of data and the method *checkTrack*. When the program calculates distance with these data in 2D, it is much simpler than with GPS coordinates in 3D. It only needs the Pythagorean Theorem (Formula 3.5).

$$a^2 + b^2 = c^2 \tag{3.5}$$

Furthermore, in real flight, the algorithm works with GPS data, which is updated every second. However, in simulation, the simulator creates next point every second and gives it back to the program. These data is generated by,

$$\text{next\_x} = \text{curr\_x} + \text{speed} \times \sin(\text{ang}) \tag{3.6}$$

$$\text{next\_y} = \text{curr\_y} + \text{speed} \times \cos(\text{ang}) \tag{3.7}$$

where (*next_x*, *next_y*) is next point, (*curr_x*, *curr_y*) is current point, and *ang* is current absolute angle.

Another difference is the method, *checkTrack*, which finds out the future virtual point according to the current direction in order to check if the UAV is flying directly to next destination. In real flight, this method becomes more complicated. It is further discussed in

Section 3.2.3 of this thesis. However, the simulator performs it in a different way. The code from the method is:

```
double track_x = curr_x + distance(curr_x, curr_y, dest_x, dest_y) * Math.sin(ang),
         track_y = curr_y + distance(curr_x, curr_y, dest_x, dest_y) * Math.cos(ang);
```

Listing 3.1: The first part of the source code of the *checkTrack algorithm*

In Figure 2.3, *"a"* is the returning value of *distance* method, θ is *angle*, *P1* is (*curr_x*, *curr_y*), and *P2* is (*track_x*, *track_y*) which is the future virtual coordinates. After getting this future point, the method measures the distance from the destination to check if the UAV is inside of the proximate circle or not. This process is described in section 2.2.2.

3.1.3 Simulation Results

This is the result in the console window.

```
110.00 , 0.00
100.00 , 0.00
90.01 , 0.52   273.00   72.62   200.38   3.00
80.07 , 1.57   276.00   73.71   202.29   3.00
```

Listing 3.2: The first 4 rows of the result.

The first and second rows are two initial points of the plane. From the third row, there are log data of the UAV's route. The First and second column tell the current position of the airplane. The third one starts with the value of the absolute angle of the present direction. After the third row, the value changes according to the decision to turn right or left, or go straight. It adds or subtracts the value of turning angle if the UAV turns right or left. When the plane goes straight, it does not change.

The fourth column is the absolute angle of the current point and the next waypoint. The fifth one shows the difference of the fourth value and the absolute angle of the current point and the previous one. The last column stands for the turning angle. It varies because of the rotation problem (Section 2.3.1)

After several rows of log data, it is said that the UAV has arrived in next destination.

```
278.08 , 94.82   163.00   176.85   -13.85   3.50
281.01 , 85.26   163.00   182.28   -19.28   3.50
283.93 , 75.70   163.00   194.05   -31.05   3.50
286.85 , 66.14   163.00   228.17   -65.17   3.50
```

Listing 3.3: The result list when the UAV arrives in the first destination.

From three underlined values in Listing 3.2, the first one is the sum of the second and third one, which means the program did not have any error.

The final result is shown in Figure 3.2

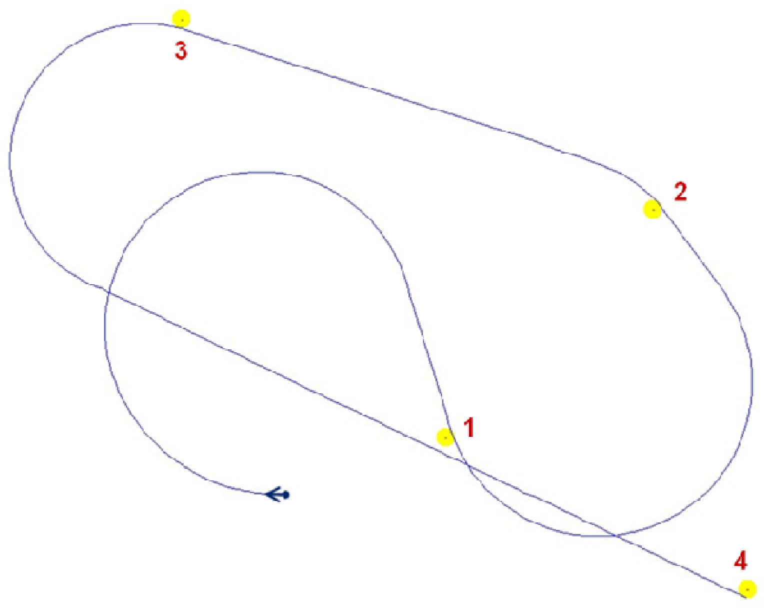


Figure 3.2: Numbers mean the order of the destination. Yellow circle around them is proximate circle (Section 2.1.3).

3.2 Simulation in PC with GPS Coordinates

3.2.1 Difference from the Prior Simulation

GPS coordinates consists of two values, latitude and longitude. In two dimensional systems, it is expressed as (x, y). However, latitude, which represents the position from the North Pole to South Pole and also means y axis in 2D systems, comes first in GPS data. Therefore, some codes had to be changed before the experiments with the real data from GPS.

During prior simulation, all the coordinates are stored in an *ArrayList*. It was used for drawing the track and checking the zigzag flight error (Section 2.3.3). However, it was found out that the *ArrayList* cannot be implemented in the LEGO NXT. Therefore, this function to avoid zigzag flight was disabled in the simulation performed in the LEGO NXT.

Real experiment deals with real GPS coordinates. Algorithm needs some other way to calculate the distance between points instead of Pythagorean Theorem (Formula 3.1). The earth is a sphere, so the formula in this case should consider the curvature on its surface.

```
public static double distance(double lat1, double lon1, double lat2,
                                          double lon2){
        double theta = lon1 - lon2;
        double dist = Math.sin(deg2rad(lat1)) * Math.sin(deg2rad(lat2))
                       + Math.cos(deg2rad(lat1)) * Math.cos(deg2rad(lat2))
                      * Math.cos(deg2rad(theta));
        dist = Math.acos(dist);
        dist = rad2deg(dist);      //change radian to degree
        dist = dist * 60 * 1.1515; //Mile
        dist = dist * 1.609344;    //Kilometer
        dist = dist * 1000;        //Meter
        return (dist);
```

Listing 3.4: The method, *distance*, calculates the distance between two GPS points.

In Listing 3.4, it is shown the Java code of the method that computes the distance between two GPS points, which considers the earth’s curvature.

Another difference is that the way for checking if the UAV flies directly to the destination or not is changed. First, the method *checkTrack* (illustrated in the section 3.1.2) was removed. Instead, a simple code was added.

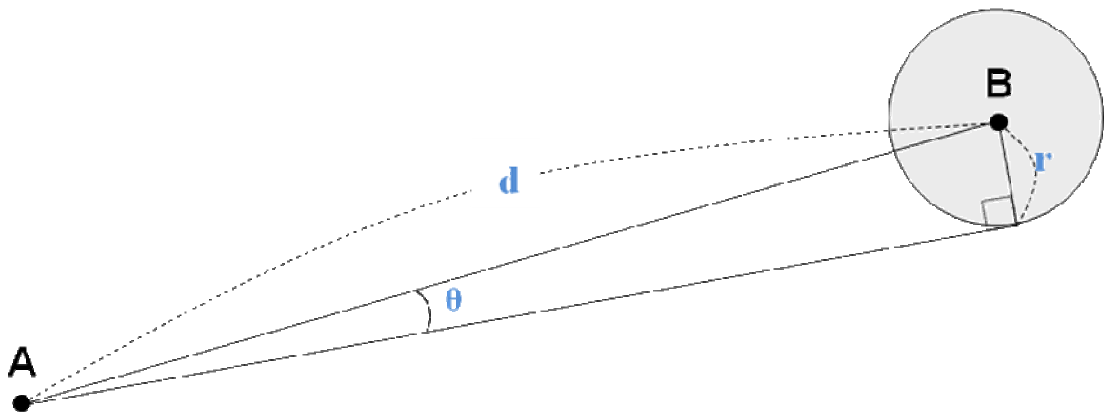


Figure 3.3: “A” is current point, “B” is destination, and “r” is the radius of proximate circle

In the Figure 3.3, “A” is current point, “B” is destination, and “r” is the radius of the proximate circle. Here, $\theta = \sin^{-1}\frac{r}{d}$ is the threshold angle for the plane to fly straight, so the value, *nav*, (For convenience, the value “α – β” presented in the Figure 2.2 is recalled in this part of the text as “*nav*”, so *nav* = α – β) is -θ < *nav* < θ, the UAV flies straight.

In the section 2.2.3, the navigation system has the algorithm.

- If $0 < \alpha - \beta \leq \pi$ or $\alpha - \beta < -\pi$, turn left.
- If $-\pi < \alpha - \beta \leq 0$ or $\alpha - \beta > \pi$, turn right.

This is replaced by

- If $\boldsymbol{\theta} < \alpha - \beta \leq \pi$ or $\alpha - \beta < -\pi$, turn left.
- If $-\pi < \alpha - \beta \leq \boldsymbol{-\theta}$ or $\alpha - \beta > \pi$, turn right.
- Else, fly straight

In the prior simulation, the program calculates the next points in integer type and uses it for the next calculation. However, this new simulation is quite different. It has a series of GPS points, which are called one by one. Figure 3.4 shows those GPS points.

The main class, *NavigationTestGPS*, gets GPS coordinates from the class, *TrackPoints*, as the data was coming from the GPS receiver. There might be some error message from the GPS receiver in real test, so here the point 10 and 24 is considered to be an error signal.

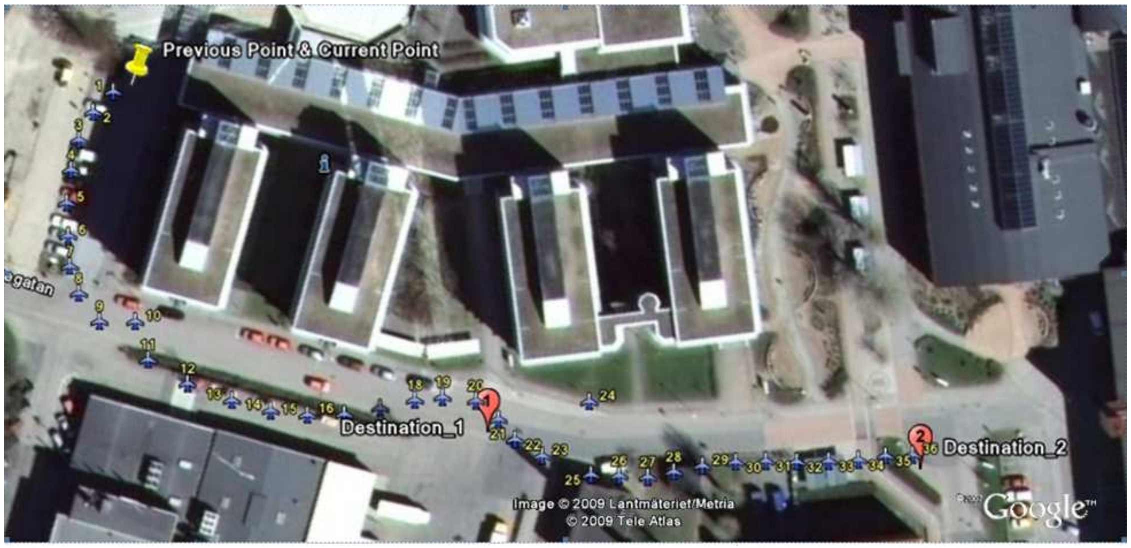

Previous Point & Current Point
Destination_1
Destination_2
Image © 2009 Lantmäteriet/Metria
© 2009 Tele Atlas
Google

Figure 3.4: pre-assigned GPS points

3.2.2 Simulation Results

The simulation presented a successful result, which is shown in the Listing 3.5. In every loop, the program gets GPS coordinates without any problem and it shows the direction to turn left or right, or to fly straight depending on the judgment described in the section 3.2.1, even in the noise condition, such as it occurs with the points 10 and 24. Additionally, it shows the turning angle, which is changed to prevent the rotation problem (section 2.3.1).

After, the UAV arrives in the first destination, the algorithm re-calculates the direction to fly corresponding to the next destination and the turning angle is set to the initial value, 3.5, as seen in line 21.

```
1: 56.664729  12.877706  Turn left   3.5 degrees
2: 56.664729  12.877618  Turn left   4.0 degrees
3: 56.664713  12.877528  Turn left   4.5 degrees
4: 56.664687  12.877445  Turn left   5.0 degrees
5: 56.664653  12.877363  Turn left   5.0 degrees
6: 56.664612  12.877295  Turn left   5.0 degrees
7: 56.664574  12.877231  Turn left   5.0 degrees
8: 56.664531  12.877191  Turn left   5.0 degrees
9: 56.664474  12.877175  Turn left   5.0 degrees
10: 56.664431  12.877251  Turn right  5.0 degrees
11: 56.664371  12.877194  Turn left   5.0 degrees
12: 56.664295  12.877227  Turn left   5.0 degrees
13: 56.664222  12.877284  Turn left   5.0 degrees
14: 56.664164  12.877344  Fly straight
15: 56.664115  12.877413  Fly straight
16: 56.664072  12.877492  Fly straight
17: 56.664037  12.877583  Fly straight
18: 56.664005  12.877673  Fly straight
19: 56.663976  12.877738  Fly straight
Arrive in the Destination 1 !!

20: 56.663932  12.877801  Fly straight
21: 56.663883  12.877807  Turn left   3.5 degrees
22: 56.663841  12.877799  Turn left   4.0 degrees
23: 56.663789  12.877819  Turn left   4.5 degrees
24: 56.663797  12.878037  Turn right  5.0 degrees
25: 56.663709  12.877882  Turn left   5.0 degrees
26: 56.663674  12.877939  Fly straight
27: 56.663639  12.877996  Fly straight
28: 56.663614  12.878062  Fly straight
29: 56.663587  12.878136  Fly straight
30: 56.663552  12.878214  Fly straight
31: 56.663518  12.878279  Fly straight
32: 56.663479  12.878344  Fly straight
33: 56.663443  12.878413  Fly straight
34: 56.663409  12.878478  Fly straight
35: 56.663381  12.878544  Fly straight
Arrive in the Destination 2 !!
```

Listing 3.5: Simulation results

3.3 Experiments with LEGO Mindstorms

3.3.1 Limitation of GPS Receiver

Before starting outdoor experiment, some limitations in the use of the GPS were found. In some measurements made in different days with different weather conditions, several errors related to imprecision have been found.

Figure 3.5 demonstrates the difference of collected points even though the GPS receiver did not move at all. The measurements were done during a cloudy day. These GPS provided data that had more than 20 meter gap.

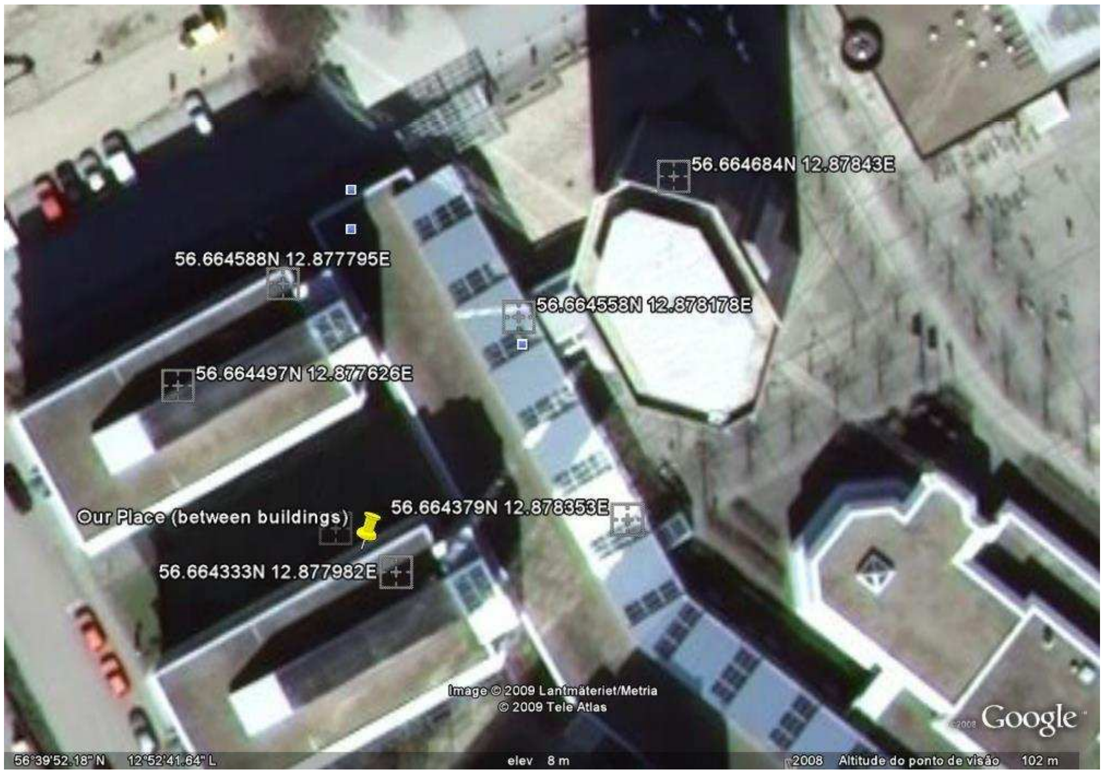


Figure 3.5: shows the results obtained between buildings.

During a sunny day, in an opened area, other measurements were done. They are shown in the Figure 3.6. In this case, the maximum difference between real position and error was smaller than 2 meters.

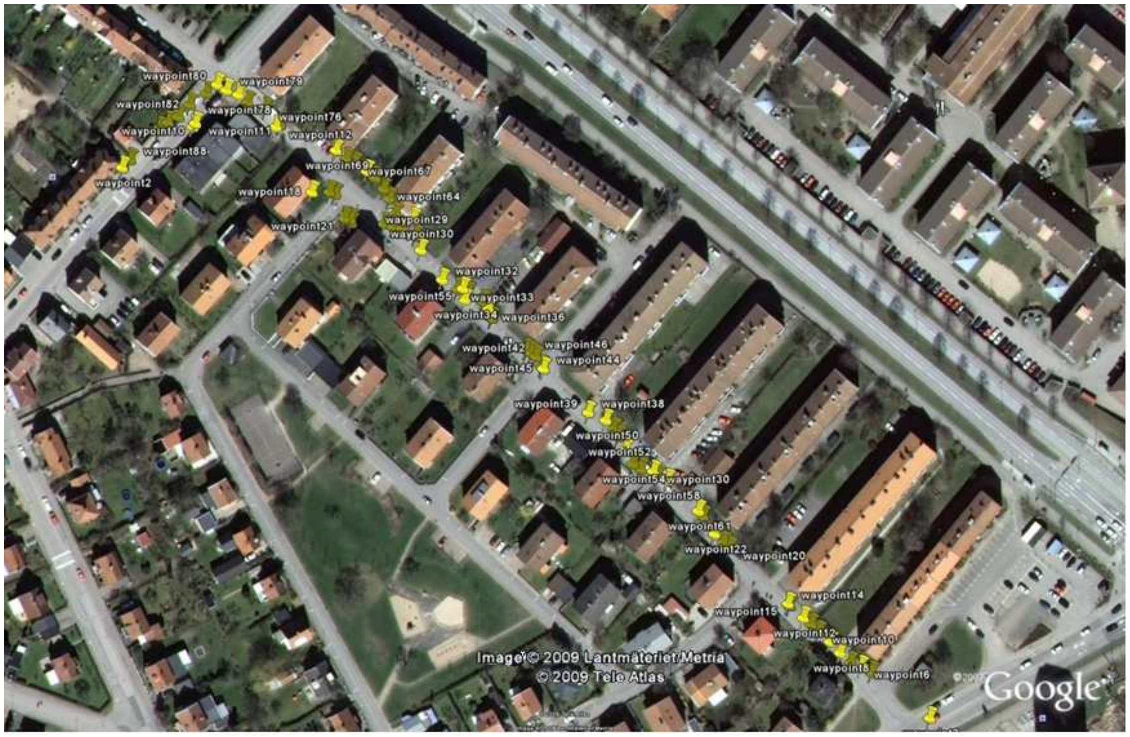


Figure 3.6: Results on sunny day while walking in constant speed.

In order to avoid errors, the following experiments were conducted in opened area during sunny days.

3.3.2 Basic Configuration

The Lego Mindstorms NXT gets GPS data and runs the navigation program with this data as input, providing the direction and the turning angle to reach the next destination waypoint. However, in the first turn of the loop, there is no previous point, but only current position just got from the GPS receiver. Therefore, the initial values for both points are given. In this experiment, the position in front of the Trade Center in Halmstad, Sweden is assigned as the starting point.

As can be observed in Figure 3.7, two initial points are at the same position. It does not mean a problem because the current position will change as soon as the program starts. Two destination points are set in Halmstad University campus, in an opened area.

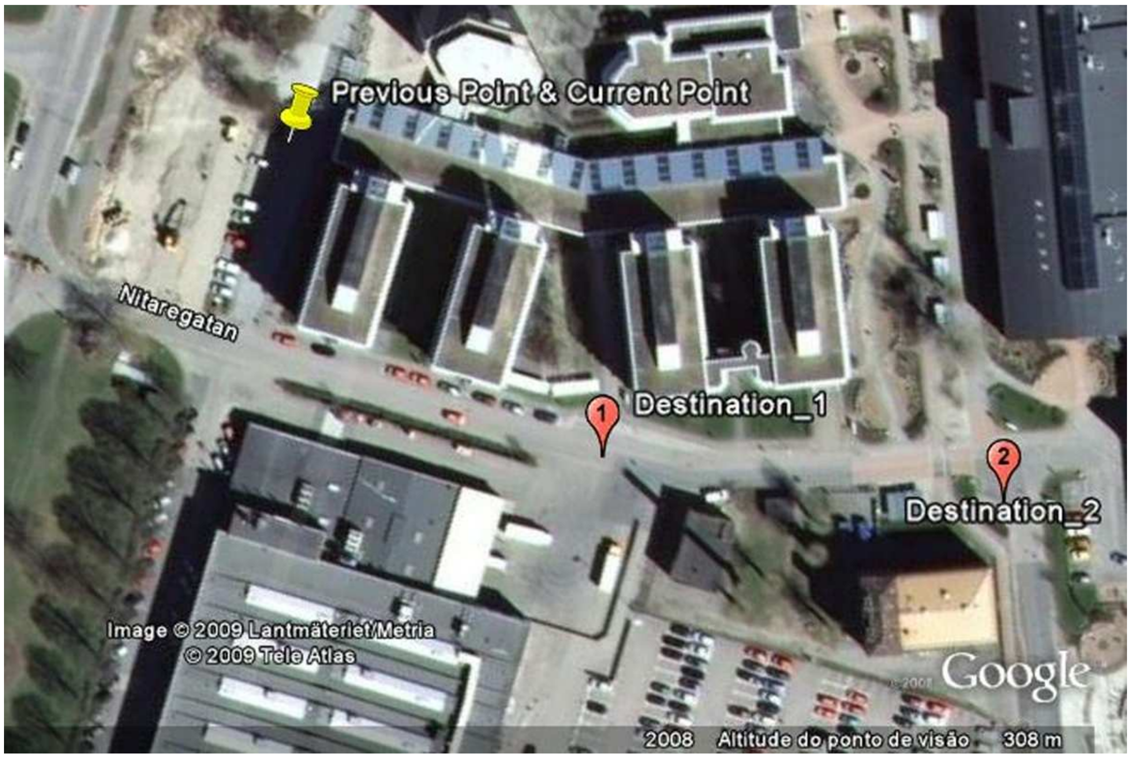

Previous Point & Current Point
Nitaregatan
Destination_1
Destination_2
Image © 2009 Lantmäteriet/Metria
© 2009 Tele Atlas
Google
2008 Altitude do ponto de visão 308 m

Figure 3.7: Previous Point, Current Point (56°39'53.00"N 12°52'40.00"E), Destination_1 (56°39'50.00"N 12°52'40.00"E), and Destination_2 (56°39'48.00"N 12°52'43.00"E).

The first experiment was performed by a person carrying the Mindstorms and the GPS, not by the UAV. A person holds the Mindstorms and GPS receiver and walk down the street in the university, following the instructions displayed on the Mindstorms screen. Thus, the speed in the program is set as 1 m/s, which is average speed when people walk.

3.3.3 MultiThreading

3.3.3.1 How does it Work?

A short explanation about the function is that a thread controls the GPS connection and shares the information with the navigation part through a data exchange class. Basically, threads [11] allow the processor to use two or more current tasks; in this case *GPSThread* (Listing 3.6) and *NavigationThread* (Listing 3.9) are used.

The main class *UAVLejos* (lines 35 and 38 of the Listing 3.7) calls the classes *GPSThread* and *NavigationThread* to be initiated. The class *GPSThread* is used to establish the Bluetooth connection between Lego Mindstorms and Holux GPSlim 240 receiver. Also in this class the GPS information is received, treated and sent to *NavigationThread.*

The figure 3.8 shows the flowchart of the behavior of the threads used to implement the simulations. The main program *UAVLejos* starts initially the classes *NavigationThread* and *GPSThread*. The class *NavigationThread* will be waiting until the first cycle is ended.

```
...
Vector devList = Bluetooth.getKnownDevicesList();
       ...
static boolean discoverBTDevices(){
       ...
return GPSDetected;
}
       ...
static int connectGPS(){
       …
try{
in = btGPS.openInputStream();
gps = new GPS(in);
gps.updateValues(true);//Update values always
       …
private void readGPS(){
//Store the initial coordinate and the moment
boolean firstMomentFlag = false;
while(true){
       …
GPSDetected = discoverBTDevices();
if(GPSDetected){
       …
connectionStatus = connectGPS();
if(connectionStatus == 2){
db.setGPSEnabled(true);
       …
db.setBTGPSMSG(MESSAGE_CONNECTED);
       …
current = new
Coordinates(gps.getLatitude(),gps.getLongitude());
db.setCurrent(current);
db.setSatellitesTracked(gps.getSatellitesTracked());
…
```

Listing 3.6: Key methods of the class *GPSThread*.

```
    public static void main(String[] args){
                TLBDB = new LRDataBridge();
                …
                TrackPoints gt = new TrackPoints(TLBDB);
                …
                NavigationThread nAvg = new NavigationThread(TLBDB);
                …
                gt.start();
                …
                nAvg.start();
```

Listing 3.7: Key method of the main class *UAVLejos*.

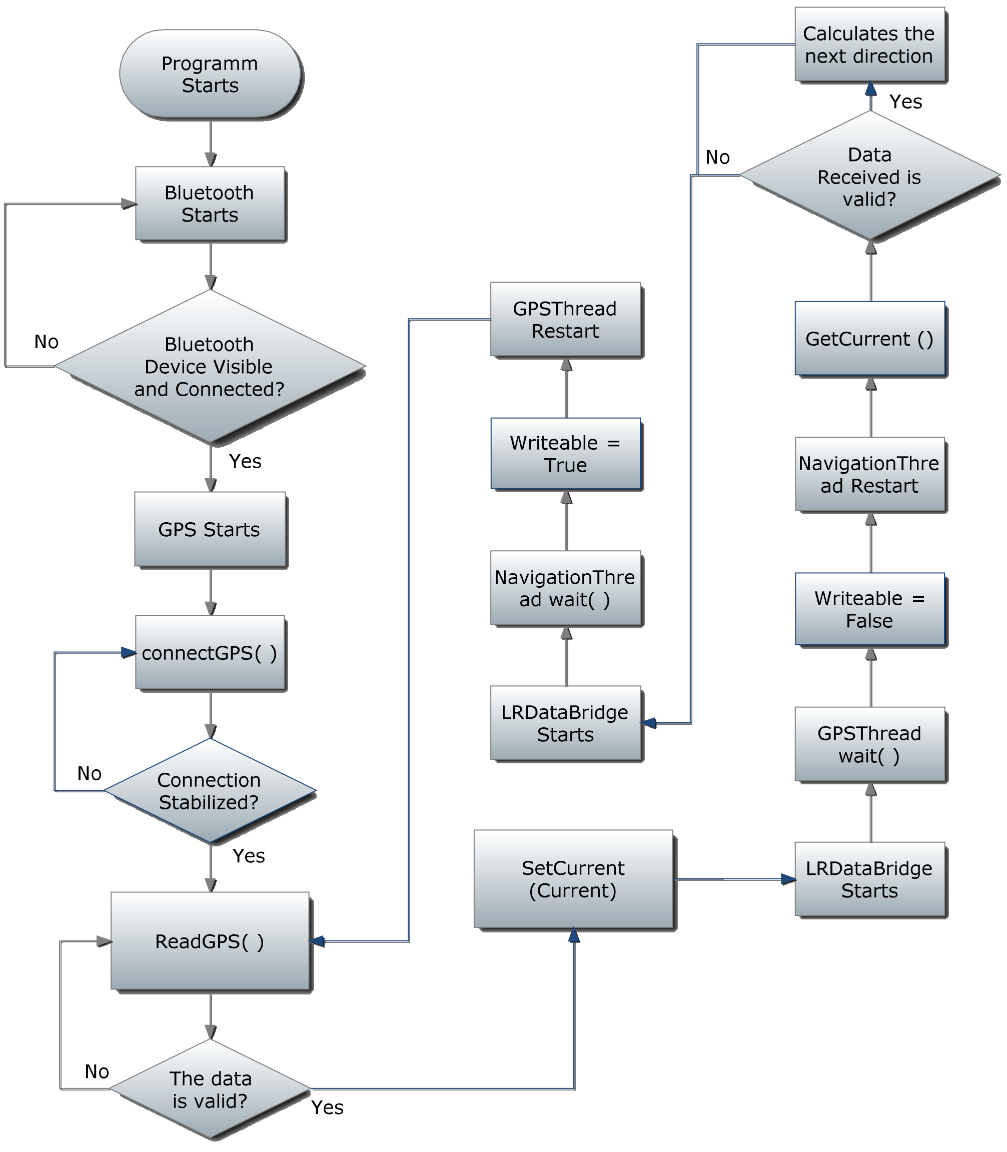


Figure 3.8: The flow chart of the program used in this experiment.

3.3.3.2 Bluetooth and GPS connection

The first step is to stabilize the Bluetooth connection. Then class *GPSThread* starts to find the GPS devices which are close to Lego Mindstorms. The method *Bluetooth.getKnownDevices()* (Line 113 of the Listing 3.6) shows all the Bluetooth devices nearby on the screen of Lego Mindstorms and “GPSlim 240” should be chosen.

If the Bluetooth connection is successful, the program will check for new GPS coordinates using the method *readGPS()* (Line 215 of the Listing 3.6). This method returns new coordinates using the method *db.setcurrent(current)* (Line 274 of the Listing 3.6), where *current* contains the latitude and longitude necessary for the navigation.

3.3.3.3 Synchronization

After the GPS coordinates set, the class *LRDataBridge* (the Listing 3.8 shows the principal methods used in this class) starts to work. As said before, this is the class that exchanges the data and controls the synchronization [12] between the threads.

The methods *setcurrent*() (Line 76 of the Listing 3.6) and *getcurrent*() (Line 88 of the Listing 3.6) are the methods used to establish the synchronization between the classes *NavigationThread* and *GPSThread*. The threads in the program are working in a way that the data produced from the class *GPSThread* is consumed by the class *NavigationThread*. Otherwise, the class *NavigationThread* would be always running and making the calculations several times using only one point, which would make the system believe that the GPS is not moving.

The variable *writeable* (Line 32 of the Listing 3.6) is a *boolean* used by the methods *setCurrent()* and *getCurrent()* to check which one will run and which one will call the method *wait()* (Lines 79 and 91 of the Listing 3.6).

Focusing on the previous steps, when class *GPSThread* sends a package containing GPS coordinates by *setCurrent()* (Line 274 of the Listing 3.6). *Writeable* is turned to *false* (Line

84 of the Listing 3.8) and the method *notify()* (Line 85 of the Listing 3.6) is called. In this moment the class *NavigationThread* restart to run and uses the method *getCurrent()* (Line 63 of the Listing 3.9) to receive the data from *LRDataBridge*. As the variable *writeable* is *false* and the class *GPSThread* will invoke the method *wait()* (Line 79 of the Listing 3.8).

The opposite case is done when the class *NavigationThread* completes the calculations or do not accept the coordinates sent by *GPSThread*. In these cases the request will arrive from *getCurrent()*. The variable *writeable* is turned to *true* (Line 95 of the Listing 3.8) and method *notify()* (Line 96 of the Listing 3.8) is called. At this time, the class *GPSThread* starts to run and the class *NavigationThread* will be waiting until be called again.

```
32 private boolean writeable = true;
…
76 public synchronized void setCurrent(Coordinates o) {
77     while (!writeable) {
78             try {
79                     wait();
80             } catch (Exception e) {
81     }
82 }
83     current = o;
84     writeable = false;
85     notify();
86 }
87
88 public synchronized Coordinates getCurrent() {
89     while (writeable) {
90              try {
91                     wait();
92     } catch (Exception e) {
93 }
94 }
95     writeable = true;
96     notify();
97
98     return current;
99 }
```

Listing 3.8: Method used to make the synchronization inside the class *LRDataBridge*.

3.3.3.4 Navigation

The coordinates that arrive at *NavigationThread* need to be tested. This test (Line 59 of the Listing 3.9) checks if the coordinates generated by *GPSThread* are trustworthy and if it is not the same one previously sent.

```
59  while  (distance(curr_x,  curr_y,  dest_x,
dest_y) >= circle) {
60      prev_x = curr_x;
61      prev_y = curr_y;
62 while (curr_x == prev_x) {
63      current = db.getCurrent();
64      curr_x = current.getLatitude();
65      x++;
66 }
67 curr_x = current.getLatitude();
68 curr_y = current.getLongitude();
```

Listing 3.9: Loop inside the Class *NavigationThread* used to check the new received GPS coordinates.

During several tests an external program, which records the GPS points of in the Lego Mindstorms, has been used. It was checked that some received data was the same as the data received in previous cycles. The same GPS coordinates were received for the period of 3 or 4 times. In most cases this problem occurred close to high buildings (explained in the section 3.3.1).

To solve the problem of multiple equal coordinates, one test is made with a *while* (Line 59 of the Listing 3.9) structure. This test compares the received data with the previously stored one. If the coordinates are the same as previous, the class *NavigationThread* will ask for new coordinates until the class *GPSThread* sends updated ones.

3.3.4 Experiment and results

The most difficult part of the project was to perform the practical experiments using the Lego Mindstorms. Without an emulator for this device, all the debugging were made directly on the hardware.

The expected results in External Tests Using Lego Mindstorms and Holux GPSlim 240 were: connect the Holux GPSlim 240 and Lego Mindstorms by Bluetooth interface; get the GPS coordinates and use the coordinates to manage the route to arrive in the destination waypoint.

A movie with results is found at: http://www.youtube.com/watch?v=ip23fQ-0-sg

3.3.4.1 Bluetooth Connection

The Bluetooth interface provides a suitable communication between the devices used in this project. The connection between the Lego Mindstorms and the Holux GPS is easy to be established. On the screen of the Lego Mindstorms appear all Bluetooth devices that are close. The connection is made by choosing “Holux GPSlim 240” item on the menu.

3.3.4.2 GPS Connection and Data Exchange

The devices take a short time to start to communicate after the Bluetooth set the communication. In the beginning the Lego Mindstorms shows the coordinates stored in the class *NavigationThread.* The connection is then successful.

The coordinates start to change after Holux GPSlim sends the first valid coordinates to the class *GPSThread* in Lego Mindstorms. The coordinates are updated in *NavigationThread* depending on the place where the tests were performed and the weather conditions. With bad weather the update stops for a short time. Depending on how long is this stop, it may disturb the desired results. The tests made close to high buildings have results comparable with the tests made with bad weather conditions.

3.3.4.3 Navigation and Check Points

The class *NavigationThread* commands all the information showed on the screen of Lego Mindstorms. It is possible to check in real time the update of the GPS coordinates sent from *GPSThread* to the *NavigationThread*, showing that the communication between threads is working correctly.

The behaviors of navigation in External Tests are questionable. The results show an unidentified bug. This bug makes all the directions to be turned to “Straight” always and without an apparent reason.

All code was divided into small parts, revised and tested separately for several times. The simulations performed in the computer that error has never appeared. Several changes were tested in the code, but they were not successful in solving this problem.

3.4 Computer Simulation Using GPS Coordinates and MultiThreading

The Simulations in Computer using Threads were made to obtain results in the order to be the most similar to the results obtained in Experiments with Lego Mindstorms.

3.4.1 Differences between Experiments with Lego Mindstorms

The figure 3.9 shows the flowchart of the Simulation in Computer using Threads. The software operates in the same order classes as the experiments made using Lego Mindstorms.

To simulate the *GPSThread* we take advantage of a class called *TrackPoints* (Listing 3.10). This class contains the method *db.setCurrent(current)* (Line 110 of the Listing 3.10) the same one as was used to exchange the GPS coordinates with the *NavigationThread* in experiments using Lego Mindstorms.

```
107   public void run(){
108          while (true){
110   current = new Coordinates(getLatitude(),getLongitude());
111   db.setCurrent(current);
113   n++;
115   }
117   private double getLatitude() {
118          add();
119          Point2D.Double gps = track.get(n);
120          return gps.getX();
121    }
122
123   private double getLongitude() {
124          add();
125          Point2D.Double gps = track.get(n);
126          return gps.getY();
```

Listing 3.10: Principal method of the class *TrackPoints.*

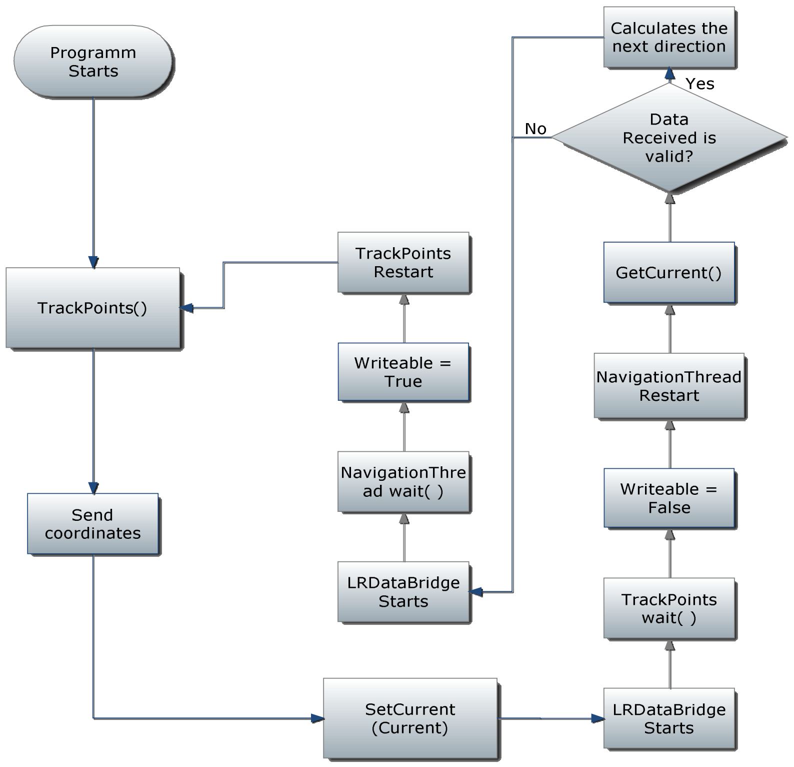


Figure 3.9: The flow chart of the program used in this experiment.

3.4.2 Basic Configurations

The main program starts initially the classes *NavigationThread* and *TrackPoints*. The class *NavigationThread* will be waiting until the first cycle is ended.

The class *TrackPoints* uses the same methods used in the session 3.2.1 to set the GPS data. These coordinates are exchanged between threads using *db.setCurrent(current)* (Line 110 of the Listing 3.10) inside the method *run()*.

How the class *LRDataBridge* uses the synchronization and how it exchanges the data and how the class *NavigationThread* works is explained in the section 3.3.3.

3.4.3 Results using GPS coordinates obtained in simulation 3.2

The GPS coordinates used in this simulation are the same as used in the Experiment 3.2.This is the result obtained on the console window using GPS coordinates:

Dir	X	Dir	X	Dir	X	Dir	X
Left	1	Left	10	**Arrive Dest 1**		Strai	27
Left	2	Left	11	Strai	19	Strai	28
Left	3	Left	12	Left	20	Strai	29
Left	4	Strai	13	Left	21	Strai	30
Left	5	Strai	14	Left	22	Strai	31
Left	6	Strai	15	Righ	23	Strai	32
Left	7	Strai	16	Left	24	Strai	33
Left	8	Strai	17	Strai	25	Strai	34
Left	9	Strai	18	Strai	26	**Arrive Dest 2**	

Table 3.1: Results obtained using real GPS coordinates.

The field “Dir” shows the direction, which the airplane needs to turn in order to arrive in the waypoint. The field “X” is updated each time when *NavigationThread* gets a new coordinate from the thread *TrackPoints.*

In the row of X = 1 is showed one initial point and one GPS point. After X = 2 at first line the calculation is made by the use of GPS points and continues changing according to the decisions made by the *NavigationThread.*

The first change in the direction appears in X = 10 and the direction turns from Left to Right. This change was applied in the way to show an error in the GPS data. When X = 11 the direction is back to turn left.

In X = 13 we can see the plane going Straight until arriving in the first Destination. The system shows the result and starts to calculate the route to the second waypoint.

From X = 19 until X = 24 the plane makes a curve to the left side with one turn to right X = 23 and from X = 25 until arriving in the last destination the plane follows a straight way until arriving at the last waypoint, as desired route.

In the last row and last line it is possible to see that the navigation made from the plane is correct. The plane arrives at the Destination 2. The *NavigationThread* shows the result and the system is turned off.

3.4.4 Results Using Real GPS Coordinates

3.4.4.1 Obtaining the GPS Coordinates

The data was obtained with a program running in Lego Mindstorms recording the GPS coordinates. These Latitudes and Longitudes were obtained every 2 seconds by Holux GPSlim 240 device. In order to have better results the simulation was held on a sunny day without clouds.

Once received this data, the Lego Mindstorms device was plugged to computer and the information was sent in a *.klm* file. This format is used in Google Earth and the Latitudes and Longitudes are extracted by opening this file with notepad.

A total of 36 points are used to make this simulation. A table with these points is found in the Appendix II.

3.4.4.2 Results

The table 3.2 shows the results in the console window. Like in the previous simulation “Dir” is the direction to turn and “X” updates when the *NavigationThread* obtains a new GPS coordinate from the thread *TrackPoints.*

In this simulation it is shown the worst case of a flight that occurs when the plane goes to the opposite direction of the waypoints. In this simulation the plane is not supposed to pass by any intermediate waypoint, the idea is to fly direct from the origin to the destination. It is used to illustrate how the navigation acts to turn when necessary.

In the moment X = 1, it works with one initial point previously updated by *NavigationThread* and one GPS point received from the thread *TrackPoints*. When X = 2 the program does not make the update because the coordinates are the same as the previous coordinates received from *TrackPoints*. These coordinates are excluded from the results and a next point is called from *TrackPoints*.

Dir	X	Dir	X	Dir	X	Dir	X
Left	1	Left	9	Left	20	Left	28
Left	3	Left	10	Left	21	Left	29
Left	4	Left	12	Left	22	Left	30
Left	5	Left	13	Left	24	Left	31
Left	6	Left	14	Right	25	Left	32
Left	7	Left	17	Right	26	Left	34
Left	8	Left	19	Right	27	Left	35

Table 3.2: Results obtained using real GPS coordinates.

The program issues the result to turn Left from X = 3 until X = 24. When X = 2, 11, 15, 16, 18, 23 and 33 the coordinates are not updated because of the while test (line 59 of the Listing 3.9) in *NavigationThread* and the program asks for new coordinates. In X = 15 and 16 has occurred that the coordinates are repeated for three consecutive iterations.

When X = 25, 26 and 27 it is possible to see in the figure 3.10 that the plane is starting a curve. These points are highlighted in the image. After this curve is completed the direction is set one more time to the Left until arrive in the final destination point.

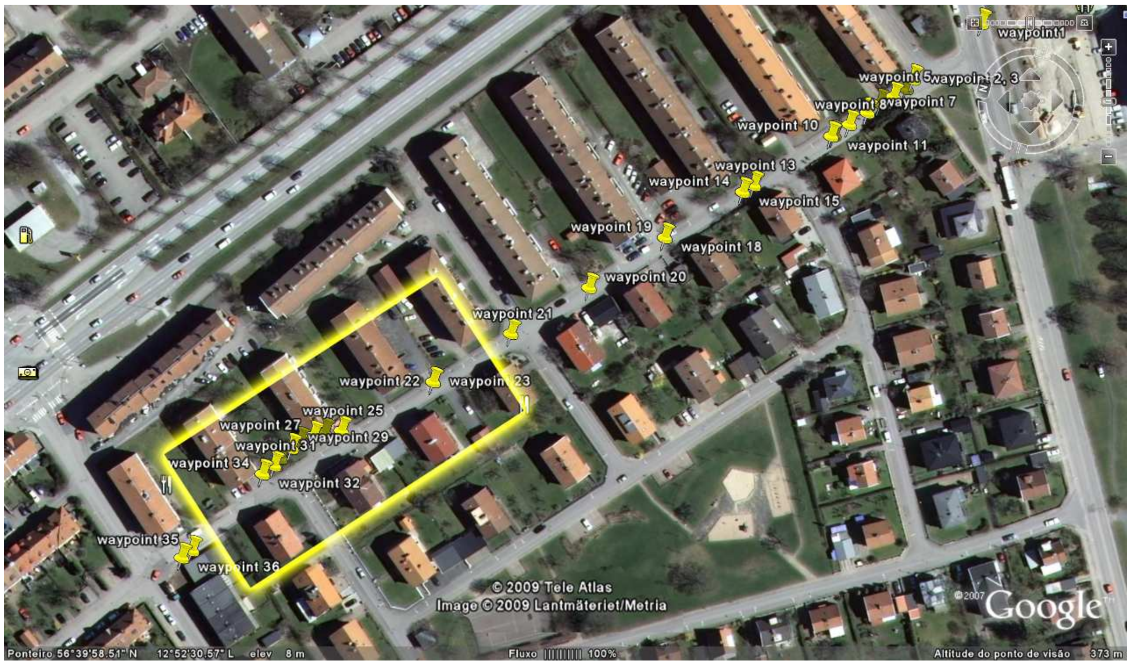


Figure 3.10: Image showing the GPS coordinates utilized in the simulation 3.4.4.

4

Conclusion and Future Work

In this project, a new algorithm to control the autonomous navigation of a UAV is designed. It helps the plane to travel through the waypoints without human's control and to minimize the time and fuel during its flight.

The proposed algorithm is not optimum. There are some issues to be considered in designing an optimal navigation system that are out of the scope of the purposes of this thesis. However, good results were achieved, meeting the major goal that was to provide a navigation controller that could provide a safe autonomous flight. One of the limitations of the work done in this thesis is that the destinations are 2 dimensional points, but as a future work, the software can take the altitude of waypoints into account. Moreover, additional improvements to the program can be made adding functions, as automatic returning after being too far away from a given position or searching optimal route considering next two waypoints.

Bibliography

[1] History of Unmanned Aerial Vehicles
http://en.wikipedia.org/wiki/History_of_unmanned_aerial_vehicles

[2] The RC Car project
URL: http://www.juanantonio.info/jab_cms.php?id=261

[3] The LEGO UAV project
URL: http://diydrones.com/profiles/blog/show?id=705844%3ABlogPost%3A32733

[4] The MIT/Draper Autonomous Helicopter Project
URL: http://web.mit.edu/whall/www/heli/

[5] The formula for a banked turn
URL: http://en.wikipedia.org/wiki/Banked_turn

[6] Accelerated_stall.gif by Giuliopp
URL: http://en.wikipedia.org/wiki/File:Accelerated_stall.gif

This file is licensed under the Creative Commons Attribution ShareAlike 3.0, Attribution ShareAlike 2.5, Attribution ShareAlike 2.0 and Attribution ShareAlike 1.0

[7] The formula for a maximum speed of an aircraft

URL:
http://www.airfieldmodels.com/information_source/math_and_science_of_model_aircraft/formulas/speed_and_propeller_efficiency.htm

[8] The manual of the propeller, Nippy 1210/103
URL: http://www.uberallmodel.cz/getfile.php?id=52

[9] The range of the rolling angle of a plane (page 11)
URL: http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA427480&Location=U2&doc=GetTRDoc.pdf

[10] FAA(2007) *Airplane flying handbook*, Skyhorse Publishing, 9-1 – 9-2

[11] Multithreading
URL: http://en.wikipedia.org/wiki/Multithreading

[12] H. E. Deitel et all. Java: *Como Programar.* Prentice Hall, 2000, p. 693

Appendix I: Hardware Specification

Technical Information:

32-bit AT91SAM7S256 main processor @ 4 MHz;

8-bit ATmega48 microcontroller @ 4MHz;

CSR BlueCore 4 bluetooth controller @ 23 MHz;

LCD display (100x64 pixels);

USB 1.1 interface working at full speed;

Bluetooth connectivity;

4 input ports and 3 output ports;

Digital wire interface

Power source: 6 AA batteries

Figure 1**:** Technical information of LeJOS Mindstorms NJT .

Specification:

Tracks up to 20 satellites

Update rate: 1HZ (max)

Receiver: L1, C/A code

Minimum signal tracked: -159dBm

Dimension: 64 x 22 x 15 mm

Weight : <35g

Lithium-ion battery lasts for more than 8 hours of use

Operation Humidity: 5% to 95% No condensing

Acquisition Time:

Reacquisition 0.1 se. averaged

Hot start 1 sec. averaged

Warm Start 38 sec. averaged

Cold start 42 sec. averaged

Interface & Protocol:

Bluetooth compatible

Frequency: 2.400 to 2.480 GHz

NMEA protocol output: V 2.2

Baud rate: 38400 bps

Data bit: 8

Parity: N

Stop bit 1

Output format: GGA (1), GSA (5), GSV (5), RMC (1), VTG(1)

Figure 2**:** List of the most important components of the GPSlim 240

Appendix II: GPS Coordinates Used in Simulation 3.4.4

Real GPS Data Used in the Simulation with Real GPS Coordinates:

track01 = (56.665145906858082, 12.877563484024409),

track02 = (56.665306124143331, 12.877189643692162),

track03 = (56.665306124143331, 12.877189643692162),

track04 = (56.665325197629670, 12.877121932815658),

track05 = (56.665340456418741, 12.877055175613473),

track06 = (56.665374788694152, 12.876988418411283),

track07 = (56.665412935666831, 12.876852996658275),

track08 = (56.665473970823115, 12.876803405593793),

track09 = (56.665496859006722, 12.876753814529311),

track10 = (56.665538820676669, 12.876667030166468),

track11 = (56.665546450071204, 12.876656539748982),

track12 = (56.665546450071204, 12.876656539748982),

track13 = (56.665740999631864, 12.876283653091051),

track14 = (56.665760073118203, 12.876243598769738),

track15 = (56.665790590696346, 12.876186378310721),

track16 = (56.665790590696346, 12.876186378310721),

track17 = (56.665790590696346, 12.876186378310721),

track18 = (56.666008028440612, 12.875795371840768),

track19 = (56.666008028440612, 12.875795371840768),

track20 = (56.666214022093076, 12.875381477187208),

track21 = (56.666431459837342, 12.874988563368621),

track22 = (56.666629824095275, 12.874613769362056),
track23 = (56.666645082884341, 12.874583251783913),
track24 = (56.666645082884341, 12.874583251783913),
track25 = (56.666854891234072, 12.874195106336911),
track26 = (56.666904482298554, 12.874153144666965),
track27 = (56.666957888060303, 12.874114997694286),
track28 = (56.666992220335714, 12.874078758070242),
track29 = (56.667030367308392, 12.874023444959858),
track30 = (56.667053255491999, 12.873972900221059),
track31 = (56.667079958372874, 12.873896606275703),
track32 = (56.667095217161946, 12.873843200513953),
track33 = (56.667125734740088, 12.873776443311766),
track34 = (56.667125734740088, 12.873776443311766),
track35 = (56.667266878538998, 12.873332031080063),
track36 = (56.667289766722605, 12.873218543836345);

Table 1**:** GPS Coordinates Used in Simulation 3.4.4 .

Appendix III: Navigation Software Source Code

```
import java.io.BufferedWriter;
import java.util.ArrayList;
import javax.microedition.location.Coordinates;
import lejos.gps.*;
import lejos.nxt.*;
import lejos.util.Stopwatch;

public class NavigationThread extends Thread{
	//Exchange Data Object
	private static LRDataBridge db;
	private static Coordinates current;
	private int setChange = 0;
	private int x = 0;


	public NavigationThread(LRDataBridge dObj){
		db = dObj;
	}

	static double  dest1_x = 56.663895,  dest1_y = 12.877778,
	               dest2_x = 56.663339,  dest2_y = 12.878611;

	static double [] vDest = new double [4];
static int circle = 10;
static double speed = 1,
                        originalTurn = deg2rad(3),
                        max_turn =  deg2rad(15);

double prev_x = 56.664728, prev_y = 12.877782;
double curr_x = 56.664728, curr_y = 12.877782;

int k = 0;     //counter how long UAV flies still after maximum angle
int n = 1;     //counter of destination

public void run(){
       while(true){
                      //-----add destination -----//
               vDest[0] = dest1_x;   vDest[1] = dest1_y;
               vDest[2] = dest2_x;   vDest[3] = dest2_y;
               double turn = originalTurn;
               for(int i = 0; i<4; i+=2){
                      double dest_x = vDest[i],
                                 dest_y = vDest[i+1];
               double ang = angle(prev_x, prev_y, curr_x, curr_y);
               while(distance(curr_x, curr_y, dest_x, dest_y)>=circle){
                                     prev_x = curr_x;
                              prev_y = curr_y;
                              while(curr_x==prev_x){
                                     current = db.getCurrent();
                                     curr_x=current.getLatitude();
                                     x++;
                                     }      //LCD.clearDisplay();
                              curr_y = current.getLongitude();
                      double a = angle(prev_x, prev_y,curr_x, curr_y),
                              b = angle(curr_x, curr_y, dest_x, dest_y),
                                l = a, m = b;
                              if(a>Math.PI) l = 2*Math.PI-a;
                              if(b>Math.PI) m = 2*Math.PI-b;
               //----- prevent UAV from rotating -----//
                              double r = speed/turn,

                                 c = Math.abs(Math.PI/2 - Math.abs(l-m)),
                                 d=distance(curr_x, curr_y, dest_x, dest_y),
                                 e=Math.sqrt(r*r + d*d - 2*r*d*Math.cos(c));

                              if(e<r){
```

```
        if(turn >= max_turn){          //if UAV turns too much
                               turn =  (0.01 * Math.PI/180);
                       }
               else if(turn == (0.01 * Math.PI/180) && k<10){
                               k++;
                       }
                       else {
                               if(turn <= originalTurn)
                                       turn = originalTurn;
                               turn+=Math.PI/360;
                               k = 0;
                       }
               }
                       //----- main control to turn -----//
               double nav = a-b,
                  angForStraight = Math.asin(circle/d);
        if((nav<=Math.PI && nav >angForStraight) || nav <= -Math.PI){
               ang-=turn;                   //turn left
               LCD.drawString("LE " + turn , 0, 3);
               LCD.drawString("Fly " + "(" + curr_x , 0, 4);
               LCD.drawString("Fly " + "(" + curr_y , 0, 5);
               LCD.drawString("Fly " + "(" + prev_x , 0, 6);
               LCD.drawString("Fly " + "(" + prev_y , 0, 7);

       }
      else if((nav>-Math.PI&&nav <=-angForStraight)||nav > Math.PI ){
               ang+=turn;                   //or   right
               LCD.drawString("RI "  , 0, 3);
               LCD.drawString("Fly " + "(" + curr_x , 0, 4);
               LCD.drawString("Fly " + "(" + curr_y , 0, 5);
               LCD.drawString("Fly " + "(" + prev_x , 0, 6);
               LCD.drawString("Fly " + "(" + prev_y , 0, 7);

       }
       else{
               LCD.refresh();
               LCD.drawString("ST " + x , 0, 3);
               LCD.drawString("Fly " + "(" + curr_x , 0, 4);
               LCD.drawString("Fly " + "(" + curr_y , 0, 5);
               LCD.drawString("Fly " + "(" + prev_x , 0, 6);
               LCD.drawString("Fly " + "(" + prev_y , 0, 7);

       }
       //}
```

```
                        }
                }
        }
} // end of run()

        public static double distance(double lat1, double lon1, double lat2, double
lon2){
                double theta = lon1 - lon2;
                double dist = Math.sin(deg2rad(lat1)) * Math.sin(deg2rad(lat2))
                                        +       Math.cos(deg2rad(lat1))        *
Math.cos(deg2rad(lat2)) * Math.cos(deg2rad(theta));
                dist = Math.acos(dist);
                dist = rad2deg(dist);
                dist = dist * 60 * 1.1515;
                dist = dist * 1.609344;
                dist = dist * 1000;
                return (dist);
        }
        public static final double angle(double lat1, double lon1, double lat2,
double lon2){
                double lat3 = lat2, lon3 = lon1;
                double   relAng   =      Math.asin(distance(lat2,   lon2,   lat3,
lon3)/distance(lat1, lon1, lat2, lon2));
                if(lon2>lon1 && lat2>=lat1){
                        return relAng;
                }
                else if(lon2>=lon1 && lat2<lat1){
                        return (Math.PI - relAng);
                }
                else if(lon2<lon1 && lat2<=lat1){
                        return (Math.PI + relAng);
                }
                else return (Math.PI*2 - relAng);
        }

        public static double deg2rad(double deg) {
          return (deg * Math.PI / 180.0);
        }

        public static double rad2deg(double rad) {
          return (rad * 180 / Math.PI);}}
```

Appendix IV: GPS Data Acquisition Software Source Code

```
import lejos.nxt.*;
public class UAVLeJOSNC {
	//GPS Receiver
	static private String GPSName = "HOLUX GPSlim240";
	static private byte[] pinCode = {(byte) '0', (byte) '0', (byte) '0', (byte)
'0'};//GPS Pin
	//Exchange Data Object
	private static LRDataBridge TLBDB;
	private static GPSThread gt;
	private static NavigationThread nAvg;
	public static void main(String[] args){
		try{
			TLBDB = new LRDataBridge();
			BatteryThread bt = new BatteryThread(TLBDB);
			GPSThread gt = new GPSThread(GPSName,pinCode,TLBDB);
			NavigationThread nAvg = new NavigationThread(TLBDB);
			gt.start();
			gt.setCommand(2);
			nAvg.start();
		}catch(Exception e){
			LCD.drawString("Problem starting",0,0);
```

```
                        LCD.refresh();
                }
                while(!Button.ESCAPE.isPressed()){
                }
                System.exit(0);
        }
}


import java.io.IOException;
import java.io.InputStream;
import java.util.Date;
import java.util.Vector;

import javax.bluetooth.RemoteDevice;
import javax.microedition.location.Coordinates;

import lejos.nxt.comm.*;
import lejos.nxt.*;
import lejos.util.Stopwatch;

/**
 * @author Denny Antonio Toazza
 * @autor Juan Antonio Brenha Moral
 *
 */
public class GPSThread extends Thread{
        private String GPSPattern = "";
        static private byte[] pin = {};

        private static byte[] cod = {0,0,0,0}; // 0,0,0,0 picks up every Bluetooth
//device regardless of Class of Device (cod).

        //Bluetooth Connection messages
        private String MESSAGE_SEARCHING = "Searching";
        private String MESSAGE_CONNECTING = "Connecting";
        private String MESSAGE_CONNECTED = "Connected";
        private String MESSAGE_RECONNECTING = "Reconnecting";
        private String MESSAGE_CONNECTION_FAILED = "Connection failed";
        private String MESSAGE_NO_GPS = "No detected GPS";

        private String MESSAGE_CLOSING = "Closing...";

        //Bluetooth
        static private RemoteDevice GPSDevice = null;
        static private BTConnection btGPS = null;
        static private InputStream in = null;
        static protected GPS gps = null;

        //Detect GPS Device
        private boolean GPSDetected = false;
        private int connectionStatus = 0;

        //GPS Data
        private Date init;
        private Date now;
        protected Coordinates current;

        //Command to process
        private int CMD = 0;

        //Connection counter
        private int connectionCounter = 0;

        //Exchange Data Object
        private LRDataBridge db;

        //LCD
        private int LCDRow = 6;

        /* CONSTRUCTOR */

        public GPSThread(String gpsp, byte[] p){
               GPSPattern = gpsp;
               pin = p;
        }
        public GPSThread(String gpsp, byte[] p, LRDataBridge dObj){
               GPSPattern = gpsp;
               pin = p;
               db = dObj;
        }

        /* Setters & Getters */

        public void setCommand(int c){
               CMD = c;
        }

        public void run(){
        		while(true){
        			if(CMD == 1){
        				//fetchData();
        				CMD = 0;
        			}else if(CMD == 2){
        				readGPS();
        			}
        		}
        	}

        	/**
        	 * Methods used to discover a predefined GPS receiver and you need
        	 * to connect with it directly.
        	 *
        	 * @param BTPatternName
        	 * @return
        	 */
        	private boolean discoverBTDevice(String BTPatternName){
        		boolean GPSDetected = false;
        		RemoteDevice btrd = null;
        		String BTDeviceName;

        		db.setBTGPSMSG(MESSAGE_SEARCHING);
        		Vector devList = Bluetooth.getKnownDevicesList();

        		if(devList.size() > 0){
        			for (int i = 0; i < devList.size(); i++) {
        				btrd = ((RemoteDevice) devList.elementAt(i));

        				BTDeviceName = btrd.getFriendlyName(true);
        				if(BTDeviceName.indexOf(BTPatternName) != -1){
        					GPSDevice = btrd;
        					GPSDetected = true;
        					break;
        				}
        			}
        		}
        		return GPSDetected;
        	}
        	static boolean discoverBTDevices(){
        		boolean GPSDetected = false;
        		LCD.clear();
        		LCD.drawString("Searching...", 0, 0);
        		LCD.refresh();
        		Vector devList = Bluetooth.inquire(5, 10,cod);
```

```
                if (devList.size() > 0){
                        String[] names = new String[devList.size()];
                        for (int i = 0; i < devList.size(); i++) {
                        RemoteDevice btrd = ((RemoteDevice) devList.elementAt(i));
                                names[i] = btrd.getFriendlyName(true);
                        }

                        TextMenu searchMenu = new TextMenu(names,1);
                        String[] subItems = {"Connect"};
                        TextMenu subMenu = new TextMenu(subItems,4);

                        int selected;
                        do {
                                LCD.clear();
                                LCD.drawString("Found",6,0);
                                LCD.refresh();
                                //Menu 1: Show all BT Devices
                                selected = searchMenu.select();
                                if (selected >=0){
                RemoteDevice btrd = ((RemoteDevice) devList.elementAt(selected));
                                        LCD.clear();
                                        LCD.drawString("Found",6,0);
                                        LCD.drawString(names[selected],0,1);
                                        LCD.drawString(btrd.getBluetoothAddress(),   0,
2
                                        int subSelection = subMenu.select();
                                        if (subSelection == 0){
                                                GPSDetected = true;
                                                GPSDevice = btrd;
                                                break;
                                        }
                                }
                        } while (selected >= 0);
                }else{
                        GPSDetected = false;
                }

                return GPSDetected;
        }

        /**
         * This method connect with a RemoteDevice.
         * If the connection has success then the method create an instance of
         * the class GPS which manages an InputStream
         *
```

```
 * @return
 */
static int connectGPS(){
int result;
btGPS = Bluetooth.connect(GPSDevice.getDeviceAddr(), NXTConnection.RAW,pin);
if(btGPS == null){
        result  = -1;//No connection
}else{
        result = 1;//Connection Sucessful
        }

        try{
                in = btGPS.openInputStream();
                gps = new GPS(in);
                gps.updateValues(true);//Update values always

                result = 2;
        }catch(Exception e){
                result = -2;
        }

        return result;
}

private void readGPS(){
        boolean firstMomentFlag = false;
        while(true){
                GPSDetected = discoverBTDevices();
                if(GPSDetected){
                        db.setBTGPSMSG(MESSAGE_CONNECTING);
                        connectionStatus = connectGPS();

                        if(connectionStatus == 2){
                                db.setGPSEnabled(true);
                                db.setBTGPSMSG(MESSAGE_CONNECTED);
                                Sound.beep();

                                Stopwatch sw = new Stopwatch();
                                while(!gps.existInternalError()){
                                        if(!firstMomentFlag){
                                        Date tempDate = gps.getDate();
                                        int hours = tempDate.getHours();
                                        int minutes = tempDate.getMinutes();
                                        int seconds = tempDate.getSeconds();
                                        int year = tempDate.getYear();
```

```
                                                int month = tempDate.getMonth();
                                                        int day = tempDate.getDay();
                                                        init = new Date();
                                                        init.setHours(hours);
                                                        init.setMinutes(minutes);
                                                        init.setSeconds(seconds);
                                                        init.setYear(year - 2000);
                                                        init.setMonth(month);

                                                        if(
                                                        (init.getSeconds() != 0) &&
                                                        (gps.getLatitude() != 0) &&
                                                        (gps.getGPSStatus())
                                                        ){



                                                                firstMomentFlag = true;
                                                        }
                                                }
                        current=new Coordinates(gps.getLatitude(),gps.getLongitude());
                                                db.setCurrent(current);
                                                db.setNow(gps.getDate());

        db.setSatellitesTracked(gps.getSatellitesTracked());
                                        }
                                        db.setBTGPSMSG(MESSAGE_RECONNECTING);
                                        db.setGPSEnabled(false);
                                }else{
                                        db.setBTGPSMSG(MESSAGE_CONNECTION_FAILED);
                                        db.setGPSEnabled(false);
                                }
                        }else{
                                db.setBTGPSMSG(MESSAGE_NO_GPS);
                                db.setGPSEnabled(false);
                        }
                }
        }
}
import javax.microedition.location.Coordinates;
import java.util.Date;

/**
 * This class has been designed to exchange data between the threads.
 *
 *
 * @author Juan Antonio Breña Moral
 * @author Denny Toazza
 */
public class LRDataBridge {

        //GPS Thread
        private boolean GPSEnabled = false;
        private String BT_GPS_MSG = "";
        private Coordinates current = new Coordinates(0,0,0);
        private Date init = new Date();
        private Date now = new Date();
        private boolean gpsInternalProblem = false;
        private int satellitesTracked = 0;

        //Waypoint Recorder Thread
        private int waypointCounter = 0;

        //System
        private int battery = 0;
        private int memory = 0;
        private int Change = 0;


        //Para a sincronizacão
        private boolean writeable = true;
        private double turn = 0;

        /**
         * Constructor
         */
        public LRDataBridge(){

        }


        /* GPS Methods */

        public void setBTGPSMSG(String MSG){
                BT_GPS_MSG = MSG;
        }

        public String getBTGPSMSG(){
                return BT_GPS_MSG;
        }
```

```
public void setInit(Date d){
        init = d;
}

public Date getInit(){
        return init;
}

public void setNow(Date d){
        now = d;
}

public Date getNow(){
        return now;
}

public void setSatellitesTracked(int st){
        satellitesTracked = st;
}

public int getSatellitesTracked(){
        return satellitesTracked;
}


public synchronized void setCurrent(Coordinates o){
        while ( !writeable ){
                try{
                        wait();
                }
                catch (Exception e) {
                        }
        }
        current = o;
        writeable = false;
        notify();
}

public synchronized Coordinates getCurrent(){
        while ( writeable ){
                try{
                        wait();
                }
                catch (Exception e) {
                        }
        }
        writeable = true;
        notify();

        return current;
}


public void setturn(double t){
        turn = t;
}

public double getturn(){
        return turn;

}




public void setGPSInternalStatus(boolean status){
        gpsInternalProblem = status;
}

public boolean getGPSStatus(){
        return gpsInternalProblem;
}

public boolean getGPSEnabled(){
        return GPSEnabled;
}

public void setGPSEnabled(boolean status){
        GPSEnabled = status;
}



/* Waypoint Recorder */

public void setWaypointCounter(int counter){
        waypointCounter = counter;
}

public int getWaypointCounter(){
        return waypointCounter;
        }
        }
}
```