TÉCNICO LISBOA

Flight Management System and Dead-Reckoning Navigation

Design of a Flight Management System and implementation of dead-reckoning navigation

Pedro Afonso **66277**
João Manito **73096**
Daniel de Schiffart **81479**

Instituto Superior Técnico
Integrated Master's Degree in Aerospace Engineering
Integrated Avionic Systems

2018/2019

Contents

I Route Distance **1**

1 Definition of Coordinates on a Spherical Earth **1**

2 Distance Determination **2**

II Dead-Reckoning Navigation **2**

3 Flight Path Definition **2**
3.1 Waypoint input . . . 3
3.2 Theoretical distance for the flight plan . . . 3

4 Flight Simulation **3**
4.1 Velocity and distance . . . 4
4.2 True heading . . . 4
4.3 Refreshing the segment position . . . 4

5 Modified Simulations **4**
5.1 Iteration velocity and distance . . . 5
5.2 Non-ideal Sensor . . . 6
5.3 Feedback Speed Controller . . . 6
5.4 Results Analysis . . . 6

Abstract

For the first laboratory of the course of Integrated Avionic Systems the objective was to design a simple version of a *Flight Management System* and use it to simulate a navigation across a series of pre-defined waypoints across a sphere-shaped earth. Further on, the development focused on the study of dead-reckoning navigation, its implementation within the Flight Management System and the comparison of a simulation with this feature against the original simulation. The final part of the laboratory shifted the focus to possible errors within the acquisition of flight velocity within the flight and ways to reduce these errors to obtain more accurate navigation. The entire work was to be implemented in C code and use a basic interface of both terminal and text-files for input and output of information.

Part I
Route Distance

The first part of this laboratory project was to develop the basic functions to allow for basic simulated navigation. With the final objective of this part being to determine the total distance of a path comprised of a series of waypoints, the work was split into different components to allow for streamlined development.

1 Definition of Coordinates on a Spherical Earth

On a spherical Earth, the coordinates are given in latitude and longitude in relation to a reference equator and meridian, while altitude is given as the difference between a certain point's distance to the center of the Earth and the spherical Earth's radius. The value of the Earth's radius was defined as being 6378000 meters.

With this environmental information, the obvious approach for saving the data of waypoints and positions in the code would be with a definition of a C structure. Our code implements a `struct` with the `typedef` name of `Coord`.

2 Distance Determination

To determine the distance between two coordinates on a spherical earth, the first objective is to determine the distance of the shortest great-circle path. As we assume that the altitude (and therefore the radius) of the path is constant for each segment of the flight, this can be done using the radial distance between the two points and multiplying it by the radius of the path. Using ϕ as latitude and λ as latitude, the radial distance θ between points 1 and 2 is given by equation 1.

$$\theta = \cos^{-1}\left(\cos\phi_2 \cos\left(\lambda_1 - \lambda_2\right)\cos\phi_1 + \sin\phi_2 \sin\phi_1\right) \tag{1}$$

Defining the Earth radius as R and the height of any coordinate as h, we can finally obtain the distance of the great-circle path between two coordinates as d, which is given by equation 2.

$$d = \theta \times (R + h_1) \tag{2}$$

These equations have been implemented as C functions in the file `lab1.c`.

Applying this to a sequencial set of waypoints is a matter of iteration and cumulative sums to determine the total distance. However, an issue is raised when the altitude is not constant between points, as we consider the altitude constant between any two waypoints using the value of the first point of any segment. This will lead to some error in the measurement of altitudes, as sudden jumps of altitude will distort the distance travelled from reality by a little bit. The focus of this discussion is illustrated by figure 1 in two dimensions.

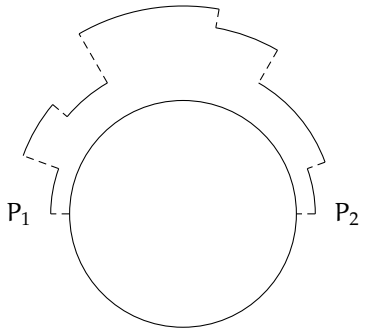

Figure 1: Simulation of the implemented distance calculation algorithm in 2D. Notice the separation of segments, the constant altitude in each of them and the unrealistic jumps of altitude between each of them.

Part II
Dead-Reckoning Navigation

The second part of this laboratory project developed on the work of the previous part to simulate a full flight from start to finish and to observe the effect of extra modifications and fixes on the accuracy of the final simulation. For that purpose, a flight plan comprised of waypoints was devised and a speed condition was applied to each segment.

3 Flight Path Definition

For a simulation to occur, a flight plan was necessary. According to the assignment, the flight plan was comprised of twelve cities, with the final waypoint being equal to the first waypoint. A value for the true airspeed was also selected for each segment, and stored in the waypoint that preceeded it. With this information, a set of coordinates was selected for each city using the city's airport as a reference. The list of waypoints can be found in table 1.

City	Latitude	Longitude	Altitude (ft)	TAS ($m\,s^{-1}$)
Lisbon	38.7812995911	−9.13591957092	374	200
Paris	49.0127983093	2.54999995232	392	300
Moscow	55.40879821777344	37.90629959106445	588	200
Oslo	60.193901062012	11.100399971008	681	500
Rome	41.8002778	12.2388889	13	250
Madrid	40.471926	−3.56264	1998	300
Funchal	32.697898864746	−16.774499893188	192	400
Ponta Delgada	37.7411994934	−25.6979007721	259	700
New York	40.63980103	−73.77890015	13	500
Halifax	44.8807983398	−63.5085983276	477	200
London	51.4706	−0.461941	83	100
Lisbon	38.7812995911	−9.13591957092	374	250

Table 1: Waypoints of the flight path used in this part of the laboratory project. Latitude and longitude are represented in decimal degrees, altitude is in feet and true airspeed is in meters per second.

3.1 Waypoint input

The input of these waypoints was done using a comma-separated value text file (henceforth referred to as `csv`). The data found in table 1 was left as-is when introduced into the file, using commas to separate each column in a line, removing all whitespace utilized (except in the header line).

For this information, we created a separate structure from the `Coord` structure used previously to allow for the storage of more information. We called this structure `Waypoint`, which was defined as seen in file `waypoints.h`.

The file is read by calling the function `read_file`, and the waypoints are returned in a `Waypoint` array whose pointer was provided as an argument to the function. We decided to use the function `fgets` in loop to get all the lines with the waypoints written. Inside the loop, longitude, latitude, altitude and true airspeed values are stored in a `Waypoint` structure. The function `csv_waypoint_parse(char line[])` receives a string containing a line of the waypoint file and returns the obtained values inside a `Waypoint` structure. The separation of each value inside the line string was done using the `strtok`, using the expected comma token.

3.2 Theoretical distance for the flight plan

To allow for the waypoint data stored in the received array to be used with the remaining functions, a `Coord` structure had to be extracted from every used waypoint. The function `waypoint_to_coord()` covers that functionality, receiving the stored data and making it usable with the necessary functions, by converting its latitude and longitude to radian format and converting the altitude to the corresponding value in SI units (in this case, meters).

The function `coord_dist()` receives the two waypoints and compute its distance. The formula is equation 2 and was explained in section 2.

4 Flight Simulation

Using the waypoints obtained in section 3, we can now start running a virtual simulation of a flight that runs a flight path comprised of the referred waypoints. This section covers the basic procedure of a simulation using constant V_{TAS} between each waypoint, for which the value is obtained from the waypoint file already referenced. The simulation herein described will therefore act as a frame of reference for further simulations.

The V_{TAS} used in a segment of the flight path corresponds to the value obtained from the segment's starting waypoint.

4.1 Velocity and distance

For the present work, we used two separate methods for computing the position. For the first lab question, we used constant V_{TAS} for each leg of the flight. The trajectory between the two waypoints was divided by segments that we called subpoints. The distance between two subpoints is defined by the true aispeed and time span relation. If we define D as the distance between two consecutive waypoints and d as the distance covered in each simulation step, with each step being a fixed Δt of time, at speed V_{TAS}, we can use the relations to find the number of subpoints n in a segment.

$$d = V_{TAS} \times \Delta t \Rightarrow n = \frac{D}{d}$$

On the other hand, the θ_{path} is given by

$$\theta_{path} = arcsen\left(\frac{\frac{dh}{dt}}{v_{TAS}}\right)$$

where the altitude rate dh/dt is given by

$$\frac{dh}{dt} = -\alpha h(t) - \alpha h_r(t)$$

at any given time, where $h(t)$ is the current altitude and $h_r(t)$ the current target altitude.

4.2 True heading

The true heading is obtained by a function `depheading(position_current, waypoint_next_coor)` which receives the current position, in a specific subpoint and the following waypoint. This is the formula to get ψ_T from the coordinates of two different points:

$$\psi_T = \tan^{-1}\left(\frac{-\cos\phi_2 \sin(\lambda_1 - \lambda_2)}{-\cos\phi_2 \cos(\lambda_1 - \lambda_2)\sin\phi_1 + \sin\phi_2 \cos\phi_1}\right)$$

4.3 Refreshing the segment position

For each time t, there is a true heading. So a loop was implemented with `coord_fromdist`. This new function gives us the next subpoint exact position, with the new coordinates. The input is the current position, the distance travelled, the heading and the climb. The two coordinates are given by the following formulas:

$$\phi_2 = \sin^{-1}\left(sin(\phi_1)\cos\left(\frac{d}{R}\right) + \cos\phi_1 \sin\left(\frac{d}{R}\right)\cos\psi\right)$$

$$\lambda_2 = \lambda_1 + \tan^{-1}\left(\frac{\sin\psi \sin\left(\frac{d}{R}\right)\cos\phi_1}{\cos\left(\frac{d}{R}\right) - \sin\phi_1 \sin\phi_2}\right)$$

To end the loop, the log file is updated with the new data. All the process is repeated again for each trajectory segment until arrive to the next waypoint.

The heading of the aircraft throughout the flight path can be seen in figure 2.

5 Modified Simulations

As part of the assignment, the simulation described in section 4 and the conditions of said simulation were to be modified and the results observed and compared.

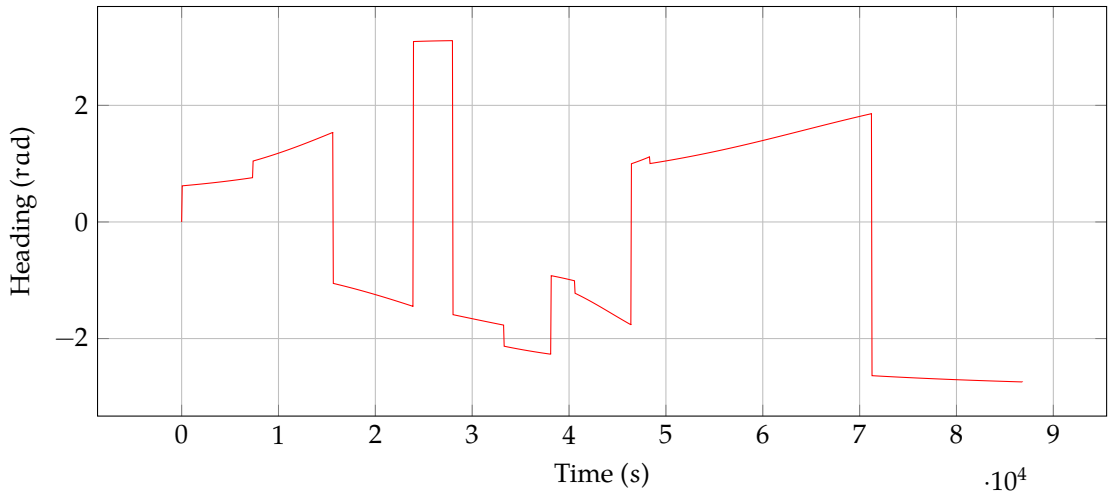

Figure 2: Aircraft heading through the flight path.

5.1 Iteration velocity and distance

Since a time-dependent simulation was required for the remaining simulations, a Flat-Earth Approximation was used. By using the coordinates of the first point as reference, and for short distances, we can define a local ENU referential. We can compute the V_{TAS} components in that referential

$$\begin{aligned} V_{East}(t) &= V_{TAS}(t)\cos(\theta_{path}(t))\sin(\phi_T(t)) \\ V_{North}(t) &= V_{TAS}(t)\cos(\theta_{path}(t))\cos(\phi_T(t)) \\ V_{Up}(t) &= V_{TAS}(t)\sin(\theta_{path}(t)) \end{aligned}$$

and then obtain the position change in ENU components as

$$\begin{aligned} \Delta E &= V_{East} \times \Delta t \\ \Delta N &= V_{North} \times \Delta t \\ \Delta U &= V_{Up} \times \Delta t \end{aligned}$$

Using the radius of the sphere where the airplane is located,

$$R = R_{Earth} + h$$

we can convert the changes in ENU coordinates to changes in the ECEF referential

$$\begin{aligned} \Delta\phi &= \frac{\Delta N}{R} \\ \Delta\lambda &= \frac{\Delta E}{R \times \cos\phi_1} \end{aligned}$$

The new coordinates are obtained by adding the computed changes to the initial point

$$\begin{aligned} \phi_2 &= \phi_1 + \Delta\phi \\ \lambda_2 &= \lambda_1 + \Delta\lambda \\ h_2 &= h_1 + \Delta h \end{aligned}$$

Since this approximation depends on the simulation's temporal resolution, the smaller the time increment the more realistic will be the result.

5.2 Non-ideal Sensor

The first test scenario was to include an error in the measured airspeed, which to this point was considered to be constant throughout any segment between waypoints. This sensor would cause a modification to the detected airspeed, which will inevitably cause differences in the final result.

The error measurements caused by the sensor with respect to time t, with $T = 20\ min$, are

$$V_m(t) = V_{TAS}\left(1 + 0.01 \times sen\left(\frac{2\pi t}{T}\right)\right)$$

Inside the loop, for each subpoint, we store now two different θ_{path} with the same function `thetapath`. The first one receives as input the True Air Speed and the position current true. However the second one, that we called `theta_sensor`, receives as input the position current relative to the sensor and the speed given by the sensor with a natural error, `Vm`.

So, there are also two different positions, named `position_current_true` and `position_current_sensor` which are calculated with the same heading. Using the function `update_position` refresh both the measured(virtual) and the real positions during the simulation. The first one receives `theta_true`, its previous position and True Air Speed, as the second one receives the speed read by the sensor, `theta_sensor` and its previous position.

Each t means 1 minute, the unit chosen for the simulation time scale. After running the simulation, the difference between the two subpoints obtained are printed in the logfile.

After reading the log file, it's clear the error is increased after each waypoint as we expected. By the other side, between two subpoints sometimes the error in the current subpoint is smaller than its previous subpoint. In a large number of subpoints, a major error can be observed when we compare the first one with the last one. The error is being accumulated after crossing each waypoint.

5.3 Feedback Speed Controller

To mitigate the noise generated by the airspeed sensor, a feedback speed controller was devised. The controller takes the desired (reference) airspeed, V_{ref}, and the current airspeed, computes the error and returns a control signal to the actuators. The implemented controller is a Proportional - Integrative (PI) controller. This type of controller was selected because it provides a good convergence speed as well as removing the static error. The controller (in its time-domain form) is defined by

$$u(t) = K_p e(t) + K_i \int_0^t e(t)$$

Where $u(t)$ is the control signal, $e(t)$ is the airspeed error relative to V_{ref} and K_i and K_p are, respectively, the integral and proportional control constants . This control signal is then fed to the actuators. For the present simulation, the simulated actuator is a simple delay of one simulation time step:

$$V_{TAS}(t) = u(t-1)$$

For this simulation, we chose K_i and K_p in a relatively loose way, since the determination of precise values for these constants was regarded as outside the main scope of this lab. We then obtained $K_i = 0.001$ and $K_p = 0.5$.

5.4 Results Analysis

The errors are decreased when a controller is used. There is an improvement on the reference following and the Velocity measured by the sensor. it was explained in subsection 5.3. The Proportional Gain improves the following reference and the Integrative Gain makes the static error 0.

From the analysis of the results, we can clearly see that the usage of the Dead-Reckoning method significantly decreases the positioning errors, due to its capability to reset the position error when a waypoint is reached. However, when using just a simple speed controller there were unexpected results. While in some cases the error using the controller is smaller than the error obtained using the noisy V_{TAS} signal as expected,

in other cases the error was bigger. This is almost certainly due to a incorrectly designed controller, where the parameters aren't good enough for this problem. There is also a possibility that the added actuator delay is also responsible for some of the error, since the delay is a full simulation time step, in this case 1 minute, which makes the response of the system lag considerably to a control signal.
There is also an interesting segment of the simulation, the Oslo-Rome leg, where the Dead Reckoning error is larger than the error given by both the noisy airspeed signal and the controller. Most likely this is also a consequence of the bad controller design.

Globally, we can consider the simulation successful, but requiring additional testing and calibration of the controller, as well as a smaller simulation step to avoid the high lag between control signal and actuator.

Table 2: Position error at the waypoints

Waypoint	Noise	Controller	DR + Controller
1	0.00	0.00	0.00
2	210.31	300.10	298.01
3	511.69	771.75	144.36
4	607.94	769.45	163.47
5	1294.72	1295.05	2009.74
6	1483.41	1436.92	764.30
7	1440.60	1556.41	134.72
8	1284.10	1383.57	470.25
9	466.55	1383.57	1212.79
10	1060.09	816.02	211.37
11	1503.28	1585.10	61.82
12	1729.36	1786.82	33.65

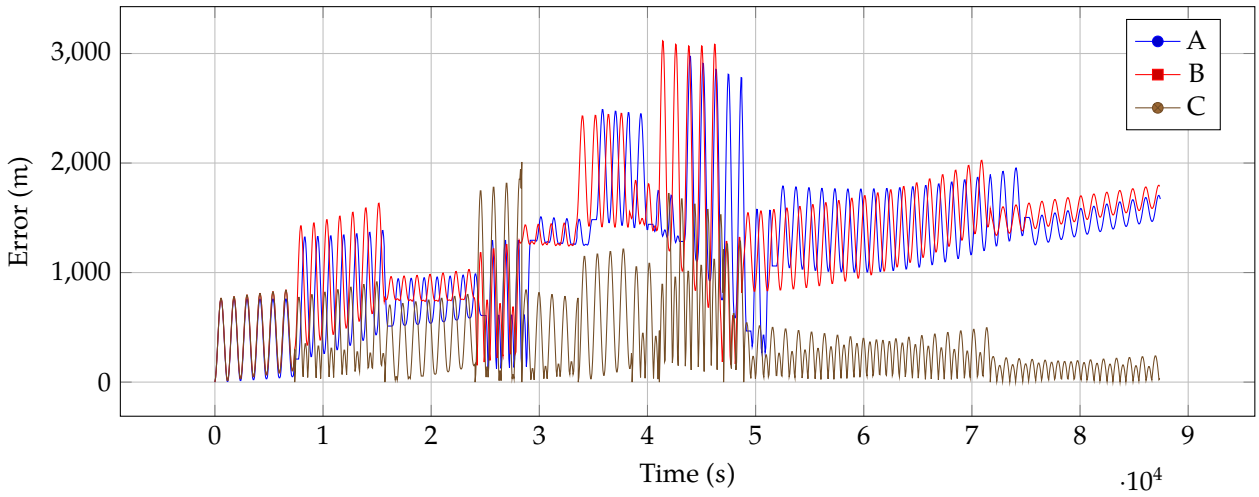

Figure 3: Subpoints distances error between the real position using TAS and the measured position using the velocity sensor – blue. Subpoints distances error between the real position using TAS and the measured position using the controller with the velocity sensor – red. Subpoints distances error between the real position using TAS and the measured position using the controller and sensor with correction of true position in the waypoints – brown.