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

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`, as represented in the file `waypoints.h`.

```
9  typedef struct {
10     double latitude;
11     double longitude;
12     double altitude;
13 } 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` as visible below.

```
double coord_dist(Coord coord1, Coord coord2){
    /* Accepts a coord struct.
     * Returns the great circle distance between coordinates in meters. */

    double ortho = acos(cos(coord2.latitude) * cos(coord1.longitude - coord2.longitude)
    ↪  * cos(coord1.latitude) + sin(coord2.latitude) * sin(coord1.latitude));

    return (ortho * (EARTH_RADIUS + coord1.altitude));
}
```

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.

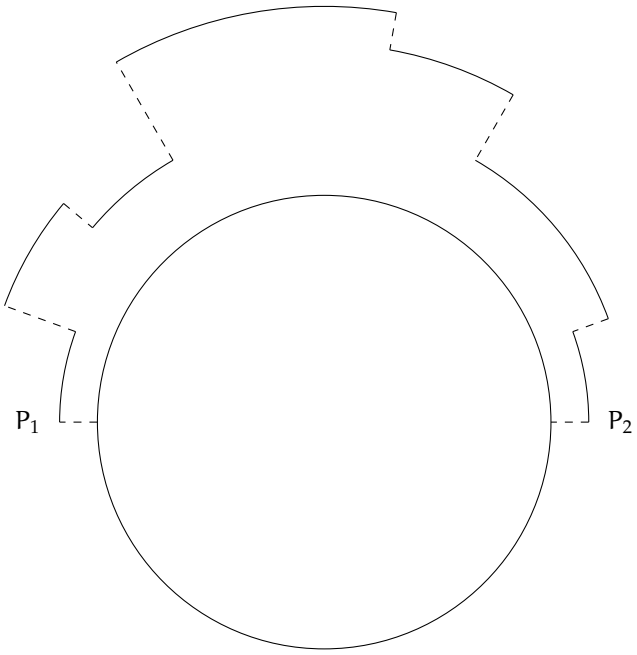

Figure 1: Simulation of the implemented distance calculation algorithm in 2D. Notice the separation of segments, the constant altitude in each of them and the unrealistic jumps of altitude between each of them.

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	0
Paris	49.0127983093	2.54999995232	392	0
Moscow	55.40879821777344	37.90629959106445	588	0
Oslo	60.193901062012	11.100399971008	681	0
Rome	41.8002778	12.2388889	13	0
Madrid	40.471926	−3.56264	1998	0
Funchal	32.697898864746	−16.774499893188	192	0
Ponta Delgada	37.7411994934	−25.6979007721	259	0
New York	40.63980103	−73.77890015	13	0
Halifax	44.8807983398	−63.5085983276	477	0
London	51.4706	−0.461941	83	0
Lisbon	38.7812995911	−9.13591957092	374	0

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). The content of the waypoint file, which was titled `waypoints.csv`, can be found below.

```
City,Latitude,Longitude,Altitude[ft],TAS
Lisbon,38.7812995911,-9.13591957092,374,200
Paris,49.0127983093,2.54999995232,392,200
Moscow,55.40879821777344,37.90629959106445,588,200
Oslo,60.193901062012,11.100399971008,681,200
Rome,41.8002778,12.2388889,13,200
Madrid,40.471926,-3.56264,1998,200
Funchal,32.697898864746,-16.774499893188,192,200
Ponta Delgada,37.7411994934,-25.6979007721,259,200
New York,40.63980103,-73.77890015,13,200
Halifax,44.8807983398,-63.5085983276,477,200
London,51.4706,-0.461941,83,200
Lisbon,38.7812995911,-9.13591957092,374,200
```

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`.

```
16 typedef struct Waypoint{
17     double latitude;
18     double longitude;
19     double altitude;
```

```
    double tas;
    char location[BUFFER];
} Waypoint ;
```

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.

```
int read_file(Waypoint* waypointlist){
    /* Accepts a Waypoint array to fill with waypoints from a file.
     * Returns 0 in case of success, 1 in case of file opening failure. */

    char* filename = "waypoints.csv";
    char line[100];
    int i = 0;

    FILE* fid;

    if ((fid = fopen(filename,"r")) == NULL){
        printf("Error opening waypoints.csv file.\n");
        return 1;
    }

    fgets(line,100,fid); /* Skip first line */

    while(!feof(fid)){
        line[0] = '\0';
        fgets(line,100,fid);

        if(strlen(line) > 30){
            waypointlist[i] = csv_waypoint_parse(line);
            i++;
        }
    }

    fclose(fid);

    return 0;
}
```

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.

```
Waypoint csv_waypoint_parse(char line[]){
    /* Accepts a string formatted according to a line from the provided CSV file */
    /* Returns the line's corresponding waypoint in a Waypoint struct */

    int i = 0;
    char* field;
    Waypoint waypoint;

    field = strtok(line, ",");
```

```
110
111     while(field != NULL){
112         switch(i){
113             case 0:
114                 strcpy(waypoint.location, field);
115             case 1:
116                 sscanf(field, "%lf", &waypoint.latitude);
117             case 2:
118                 sscanf(field, "%lf", &waypoint.longitude);
119             case 3:
120                 sscanf(field, "%lf", &waypoint.altitude);
121             case 4:
122                 sscanf(field, "%lf", &waypoint.tas);
123         }
124
125         field = strtok(NULL, ",");
126         i++;
127     }
128
129     return waypoint;
130 }
```

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).

```
1  Coord waypoint_to_coord (Waypoint waypoint){
2      /* Accepts a waypoint and returns a coordinate struct with its position. */
3      Coord coordinate;
4
5      coordinate.latitude = waypoint.latitude * M_PI/180;
6      coordinate.longitude = waypoint.longitude * M_PI/180;
7      coordinate.altitude = waypoint.altitude * FT2METER;
8
9      return coordinate;
10 }
```

The next way point from the structure vector `waypoint_list` is also converted to compute the distance between the first.

```
1  printf("Started in city %s\n", waypoint_list[0].location);
2      printf("The waypoint count is %d\n", waypoint_count);
3      for(i = 0; i < waypoint_count; i++){
4          waypoint_prev_coor = waypoint_to_coord(waypoint_list[i]);
5          waypoint_next_coor = waypoint_to_coord(waypoint_list[i+1]);
6
7          theodist += coord_dist(waypoint_prev_coor, waypoint_next_coor);
8      }
```

```
9
10      printf("Total theoretical distance is %fm\n", theodist);
```

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

3.3 True heading

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 get

$$d = \frac{D}{v_{TAS} \times \Delta t}$$

```
subpoint_count = coord_dist(waypoint_prev_coor, waypoint_next_coor) / (v_tas *
↪  TIMESPAN);
```

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.

```
1  double theta_path(Coord coor1, Coord coor2, double v_tas){
2      /* Accepts two coordinates and returns the climb angle from the first to
3       * the second based on the altitude rate equation. */
4      double altituderate = -ALTRATE_MOD * coor1.altitude + ALTRATE_MOD * coor2.altitude;
5      return asin(altituderate / v_tas);
6  }
```

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:

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

```
1  double depheading(Coord coord1, Coord coord2){
2      /* Accepts two coord structs.
3       * Returns departure heading (from north) in radians using Great Circles
↪  (Orthodrome). */
4
5      double depheading = atan2(-cos(coord2.latitude) * sin(coord1.longitude -
↪  coord2.longitude), -cos(coord2.latitude) * cos(coord1.longitude -
↪  coord2.longitude) * sin(coord1.latitude) + sin(coord2.latitude) *
↪  cos(coord1.latitude));
```

```
    return depheading;
}
```

For each time t, there is a true heading. So a loop was implemented with the functions described above and `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 three coordinates are given by the following formulas:

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

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

```
Coord coord_fromdist(Coord coord1, double dist, double heading, double climb){
    /* Accepts a coord struct, a distance in meters, a heading in radians, and a climb distance in meters.
     * Returns a coord struct. */

    Coord coord2;

    coord2.latitude = asin(sin(coord1.latitude) * cos(dist / EARTH_RADIUS) + cos(coord1.latitude) * sin(dist / EARTH_RADIUS) * cos(heading));
    coord2.longitude = coord1.longitude + atan2(sin(heading) * sin(dist / EARTH_RADIUS) * cos(coord1.latitude), cos(dist / EARTH_RADIUS) - sin(coord1.latitude) * sin(coord2.latitude));
    coord2.altitude = coord1.altitude + climb;

    return coord2;
}
```

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.

```
        for(j = 0; j < floor(subpoint_count); j++){
            theta = theta_path(position_current, waypoint_next_coor, v_tas);
            heading = depheading(position_current, waypoint_next_coor);

            /* position_current = iter(position_current, v_tas, TIMESPAN, theta, heading); */
            position_current = coord_fromdist(position_current, v_tas * TIMESPAN, heading, v_tas * sin(theta));
            /*fprintf(f, "Longitude = %f, Latitude = %f, Altitude = %fm, Distance left = %fm, Subpoint %d\n", position_current.latitude * 180/M_PI, position_current.longitude * 180/M_PI, position_current.altitude, coord_dist(position_current, waypoint_next_coor), j);*/
            /* totaldist += v_tas * TIMESPAN; */
            runtime += TIMESPAN;
            update_log(fid_log, runtime, theta, v_tas, v_tas, heading, position_current, position_current);

        }
```

3.4 Sensor included

The error measurements caused by the sensor, with $T = 20min$, are:

$$V_m(t) = V_{TAS}(1 + 0.01 * sen(\frac{2\pi t}{T})) \tag{3}$$