

```
1  //-----
2  //      Title:MASTER DEGREE THESIS by ANTONIO SCAZZI
3  //
4  //  Description:header file with all additional functions
5  //-----
6  #pragma once
7  #include "globalvar.h"
8
9  void IsOnGround();
10
11 void GearCheck();
12
13 std::vector<double> RelativePos(double C2_latitude, double C2_longitude, double C2_altitude, double DRONE_latitude, ↗
    double DRONE_longitude, double DRONE_altitude, double C2_pitch, double C2_roll, double C2_heading);
14
15 //TRANSFORM the info on latitude longitude and altitude to the NED reference frame centered in C2 and rotated by ↗
    heding,pitch and roll
16 std::vector<double> EcefToBody(double C2_latitude, double C2_longitude, double C2_altitude, double DRONE_latitude, ↗
    double DRONE_longitude, double DRONE_altitude, double C2_pitch, double C2_roll, double C2_heading);
17
18 //TRANSFORM the info on latitude longitude and altitude to the NED reference frame centered in C2
19 std::vector<double> EcefToNEDPhi(double C2_latitude, double C2_longitude, double C2_altitude, double ↗
    DRONE_latitude, double DRONE_longitude, double DRONE_altitude, double C2_heading);
20
21 void IsOnGround()
22 {
23     if (UserPlane.velocity < 50 && flag_initialgroundcheck == 0 && UserPlane.simtime>0.1)
24     {
25         //correct the heading on the take off runway
26         initial.Heading = initial_heading;
27         hr = SimConnect_SetDataOnSimObject(hSimConnect, DEFINITION_2, SIMCONNECT_OBJECT_ID_USER, 0, 0, sizeof ↗
            (initial), &initial);
28         flag_initialgroundcheck = 1;
```

```
29     }
30     else if (UserPlane.velocity > 50 && flag_initialgroundcheck == 0 && UserPlane.simtime > 0.1)
31     {
32         flag_initialgroundcheck = 1;
33         flag_decollo = 4;
34     }
35 }
36
37 void GearCheck()
38 {
39     if (flag_landgear == 0 && UserPlane.velocity > 350)
40     {
41         //retrazione del carrello
42         SimConnect_TransmitClientEvent(hSimConnect, SIMCONNECT_OBJECT_ID_USER, EVENT_LANDING_GEAR, 0,
43                                         SIMCONNECT_GROUP_PRIORITY_HIGHEST, SIMCONNECT_EVENT_FLAG_GROUPID_IS_PRIORITY);
44         flag_landgear = 1;
45     }
46
47
48     // Funzione per trasformare le differenze di coordinate ECEF nel sistema body
49     std::vector<double> EcefToBody(double C2_latitude, double C2_longitude, double C2_altitude, double DRONE_latitude,
50                                   double DRONE_longitude, double DRONE_altitude, double C2_pitch, double C2_roll, double C2_heading)
51     {
52         // Converti gli angoli dell'origine in radianti
53         double phi0 = C2_latitude * PI / 180.0;
54         double lambda0 = C2_longitude * PI / 180.0;
55         double h0 = C2_altitude * .3048;
56         double phi = DRONE_latitude * PI / 180.0;
57         double lambda = DRONE_longitude * PI / 180.0;
58         double h = DRONE_altitude * .3048;
59         double pitch0 = C2_pitch * PI / 180.0;
```

```
60     double roll0 = C2_roll * PI / 180.0;
61     double heading0 = C2_heading * PI / 180.0;
62
63     //calcolo l'eccentricità al quadrato
64     double e2 = 1 - (WGS84_B * WGS84_B) / (WGS84_A * WGS84_A);
65
66     //trasformo in sistema ECEF C2Plane
67     double N_C2_ECEF = WGS84_A / sqrt(1 - e2 * sin(phi0) * sin(phi0));
68     double X_C2_ECEF = (N_C2_ECEF + h0) * cos(phi0) * cos(lambda0);
69     double Y_C2_ECEF = (N_C2_ECEF + h0) * cos(phi0) * sin(lambda0);
70     double Z_C2_ECEF = (N_C2_ECEF * (pow(WGS84_B, 2) / pow(WGS84_A, 2)) + h0) * sin(phi0);
71     //trasformo in sistema ECEF DRONE
72     double N_DRONE_ECEF = WGS84_A / sqrt(1 - e2 * sin(phi) * sin(phi));
73     double X_DRONE_ECEF = (N_DRONE_ECEF + h) * cos(phi) * cos(lambda);
74     double Y_DRONE_ECEF = (N_DRONE_ECEF + h) * cos(phi) * sin(lambda);
75     double Z_DRONE_ECEF = (N_DRONE_ECEF * (pow(WGS84_B, 2) / pow(WGS84_A, 2)) + h) * sin(phi);
76     //calcolo le differenze in ecef
77     double DeltaX = X_DRONE_ECEF - X_C2_ECEF;
78     double DeltaY = Y_DRONE_ECEF - Y_C2_ECEF;
79     double DeltaZ = Z_DRONE_ECEF - Z_C2_ECEF;
80
81
82
83
84     // Matrice di rotazione ECEF a NED
85     double R[3][3] = {
86         { -sin(phi0) * cos(lambda0), -sin(phi0) * sin(lambda0), cos(phi0) },
87         { -sin(lambda0), cos(lambda0), 0 },
88         { -cos(phi0) * cos(lambda0), -cos(phi0) * sin(lambda0), -sin(phi0) }
89     };
90
91     // Vettore delle differenze di coordinate ECEF
92     double dECEF[3] = { DeltaX, DeltaY, DeltaZ };
```

```
93
94     // Vettore delle coordinate NED risultanti
95     std::vector<double> NED(3, 0.0);
96
97     // Calcola le coordinate NED
98     for (int i = 0; i < 3; ++i) {
99         for (int j = 0; j < 3; ++j) {
100             NED[i] += R[i][j] * dECEF[j];
101         }
102     }
103
104
105     //ruotare attorno ai vari assi
106     // Matrice di rotazione per heading
107     double R_heading[3][3] = {
108         { cos(-heading0), -sin(-heading0), 0 },
109         { sin(-heading0), cos(-heading0), 0 },
110         { 0, 0, 1 }
111     };
112
113     // Matrice di rotazione per pitch
114     double R_pitch[3][3] = {
115         { cos(pitch0), 0, sin(pitch0) },
116         { 0, 1, 0 },
117         { -sin(pitch0), 0, cos(pitch0) }
118     };
119
120     // Matrice di rotazione per roll
121     double R_roll[3][3] = {
122         { 1, 0, 0 },
123         { 0, cos(roll0), -sin(roll0) },
124         { 0, sin(roll0), cos(roll0) }
125     };
```

```
126
127
128     // Applicare le rotazioni al sistema NED
129
130     // Rotazione per heading
131     std::vector<double> bodyAxes(3, 0.0);
132     for (int i = 0; i < 3; ++i) {
133         for (int j = 0; j < 3; ++j) {
134             bodyAxes[i] += R_heading[i][j] * NED[j];
135         }
136     }
137
138     // Rotazione per pitch
139     std::vector<double> bodyAxes_roll(3, 0.0);
140     for (int i = 0; i < 3; ++i) {
141         for (int j = 0; j < 3; ++j) {
142             bodyAxes_roll[i] += R_pitch[i][j] * bodyAxes[j];
143         }
144     }
145
146     // Rotazione per roll
147     std::vector<double> bodyAxes_final(3, 0.0);
148     for (int i = 0; i < 3; ++i) {
149         for (int j = 0; j < 3; ++j) {
150             bodyAxes_final[i] += R_roll[i][j] * bodyAxes_roll[j];
151         }
152     }
153     return bodyAxes_final;
154 }
155
156 // Funzione per trasformare le differenze di coordinate ECEF nel sistema body
157 std::vector<double> EcefToNEDPhi(double C2_latitude, double C2_longitude, double C2_altitude, double
    DRONE_latitude, double DRONE_longitude, double DRONE_altitude, double C2_heading)
```

```
158 {
159
160     // Converti gli angoli dell'origine in radianti
161     double phi0 = C2_latitude * PI / 180.0;
162     double lambda0 = C2_longitude * PI / 180.0;
163     double h0 = C2_altitude * .3048;
164     double phi = DRONE_latitude * PI / 180.0;
165     double lambda = DRONE_longitude * PI / 180.0;
166     double h = DRONE_altitude * .3048;
167     double heading0 = C2_heading * PI / 180.0;
168
169     //calcolo l'eccentricità al quadrato
170     double e2 = 1 - (WGS84_B * WGS84_B) / (WGS84_A * WGS84_A);
171
172     //trasformo in sistema ECEF C2Plane
173     double N_C2_ECEF = WGS84_A / sqrt(1 - e2 * sin(phi0) * sin(phi0));
174     double X_C2_ECEF = (N_C2_ECEF + h0) * cos(phi0) * cos(lambda0);
175     double Y_C2_ECEF = (N_C2_ECEF + h0) * cos(phi0) * sin(lambda0);
176     double Z_C2_ECEF = (N_C2_ECEF * (pow(WGS84_B, 2) / pow(WGS84_A, 2)) + h0) * sin(phi0);
177     //trasformo in sistema ECEF DRONE
178     double N_DRONE_ECEF = WGS84_A / sqrt(1 - e2 * sin(phi) * sin(phi));
179     double X_DRONE_ECEF = (N_DRONE_ECEF + h) * cos(phi) * cos(lambda);
180     double Y_DRONE_ECEF = (N_DRONE_ECEF + h) * cos(phi) * sin(lambda);
181     double Z_DRONE_ECEF = (N_DRONE_ECEF * (pow(WGS84_B, 2) / pow(WGS84_A, 2)) + h) * sin(phi);
182     //calcolo le differenze in ecef
183     double DeltaX = X_DRONE_ECEF - X_C2_ECEF;
184     double DeltaY = Y_DRONE_ECEF - Y_C2_ECEF;
185     double DeltaZ = Z_DRONE_ECEF - Z_C2_ECEF;
186
187
188
189
190     // Matrice di rotazione ECEF a NED
```

```
191     double R[3][3] = {
192         { -sin(phi0) * cos(lambda0), -sin(phi0) * sin(lambda0), cos(phi0) },
193         { -sin(lambda0), cos(lambda0), 0 },
194         { -cos(phi0) * cos(lambda0), -cos(phi0) * sin(lambda0), -sin(phi0) }
195     };
196
197     // Vettore delle differenze di coordinate ECEF
198     double dECEF[3] = { DeltaX, DeltaY, DeltaZ };
199
200     // Vettore delle coordinate NED risultanti
201     std::vector<double> NED(3, 0.0);
202
203     // Calcola le coordinate NED
204     for (int i = 0; i < 3; ++i) {
205         for (int j = 0; j < 3; ++j) {
206             NED[i] += R[i][j] * dECEF[j];
207         }
208     }
209
210
211     //ruotare attorno ai vari assi
212     // Matrice di rotazione per heading
213     double R_heading[3][3] = {
214         { cos(-heading0), -sin(-heading0), 0 },
215         { sin(-heading0), cos(-heading0), 0 },
216         { 0, 0, 1 }
217     };
218
219
220     // Applicare le rotazioni al sistema NED
221
222     // Rotazione per heading
223     std::vector<double> bodyAxes(3, 0.0);
```

```
224     for (int i = 0; i < 3; ++i) {
225         for (int j = 0; j < 3; ++j) {
226             bodyAxes[i] += R_heading[i][j] * NED[j];
227         }
228     }
229     return bodyAxes;
230 }
231
232 // Funzione per trasformare l
233 std::vector<double> RelativePos(double C2_latitude, double C2_longitude, double C2_altitude, double DRONE_latitude,
234                                double DRONE_longitude, double DRONE_altitude, double C2_pitch, double C2_roll, double C2_heading)
235 {
236     // Converti gli angoli dell'origine in radianti
237     double phi0 = C2_latitude * PI / 180.0;
238     double lambda0 = C2_longitude * PI / 180.0;
239     double pitch0 = C2_pitch * PI / 180.0;
240     double roll0 = C2_roll * PI / 180.0;
241     //double heading0 = C2_heading * PI / 180.0;
242     //asse z rivolto verso l'alto e x avanti, y sinistra
243     double heading0 = -phi0 * PI / 180.0;
244
245     std::vector<double> NED(3, 0.0);
246     NED[0] = DRONE_latitude - C2_latitude;
247     NED[1] = DRONE_longitude - C2_longitude;
248     NED[2] = DRONE_altitude - C2_altitude;
249
250     //ruotare attorno ai vari assi
251     // Matrice di rotazione per heading
252     double R_heading[3][3] = {
253         { cos(heading0), -sin(heading0), 0 },
254         { sin(heading0), cos(heading0), 0 },
255         { 0, 0, 1 }
256     };
257 }
```



```
256
257 // Matrice di rotazione per pitch
258 double R_pitch[3][3] = {
259     { cos(pitch0), 0, sin(pitch0) },
260     { 0, 1, 0 },
261     { -sin(pitch0), 0, cos(pitch0) }
262 };
263
264 // Matrice di rotazione per roll
265 double R_roll[3][3] = {
266     { 1, 0, 0 },
267     { 0, cos(roll0), -sin(roll0) },
268     { 0, sin(roll0), cos(roll0) }
269 };
270
271
272 // Applicare le rotazioni al sistema NED
273
274 // Rotazione per heading
275 std::vector<double> bodyAxes(3, 0.0);
276 for (int i = 0; i < 3; ++i) {
277     for (int j = 0; j < 3; ++j) {
278         bodyAxes[i] += R_heading[i][j] * NED[j];
279     }
280 }
281
282 // Rotazione per pitch
283 std::vector<double> bodyAxes_roll(3, 0.0);
284 for (int i = 0; i < 3; ++i) {
285     for (int j = 0; j < 3; ++j) {
286         bodyAxes_roll[i] += R_pitch[i][j] * bodyAxes[j];
287     }
288 }
```

```
289
290     // Rotazione per roll
291     std::vector<double> bodyAxes_final(3, 0.0);
292     for (int i = 0; i < 3; ++i) {
293         for (int j = 0; j < 3; ++j) {
294             bodyAxes_final[i] += R_roll[i][j] * bodyAxes_roll[j];
295         }
296     }
297     return bodyAxes;
298 }
299
```