

Quaternion2Euler.h X Quaternion2Euler.cpp

+ Miscellaneous Files - No Configurations (Global Scope)

```
1  #pragma once
2  #include <math.h>
3  #define max(a,b) ((a<b)?b:a)
4  #define min(a,b) ((a>b)?b:a)
5  void Quaternion2Euler(
6      double* roll,
7      double* pitch,
8      double* yaw,
9      const double q[4]);
10
```

```
1
2   #include "Quaternion2Euler.h"
3
4   void Quaternion2Euler(
5       double* roll,
6       double* pitch,
7       double* yaw,
8       const double q[4]) {
9       double e0 = q[0];
10      double e1 = q[1];
11      double e2 = q[2];
12      double e3 = q[3];
13      if (e0 < 0.0) {
14          e0 = -e0;
15          e1 = -e1;
16          e2 = -e2;
17          e3 = -e3;
18      }
19      *roll = atan2(2.0 * (e0 * e1 + e2 * e3), (e0 * e0 + e3 * e3 - e1 * e1 - e2 * e2));
20      *pitch = asin(max(-1.0, min(1.0, 2.0 * (e0 * e2 - e1 * e3))));
21      *yaw = atan2(2.0 * (e0 * e3 + e1 * e2), (e0 * e0 + e1 * e1 - e2 * e2 - e3 * e3));
22  }
23
```

```
1  #pragma once
2
3  #include <math.h>
4  #include <cmath>
5
6  #define PI (3.14159265359f)
7  #define max(a,b) ((a<b)?b:a)
8  #define min(a,b) ((a>b)?b:a)
9
10
11 //Create a cross-product matrix
12 extern void CrossMatrix(
13     double CrossMat[3][3], //OUTPUT: Cross product matrix
14     const double vec3[3] //INPUT: vector to be converted to a cross product matrix
15 );
16
17
18 //Inertial to body rotation matrix
19 extern void Rotation_I2b(
20     double Ri2b[3][3], //OUTPUT: Rotation matrix from the inertial to body frame (Rows first, then columns)
21     const double e0, //INPUT: scalar part of the quaternion orientation in the inertial frame
22     const double e1, //INPUT: x-axis part of the quaternion orientation in the inertial frame
23     const double e2, //INPUT: y-axis part of the quaternion orientation in the inertial frame
24     const double e3 //INPUT: z-axis part of the quaternion orientation in the inertial frame
25 );
26
27 //Rotate a vec3 from the inertial frame to the body frame
28 extern void inertial_2_body_Rotation(
29     double* x_b, //OUTPUT: x-position in the body frame
30     double* y_b, //OUTPUT: y-position in the body frame
31     double* z_b, //OUTPUT: z-position in the body frame
32     const double xI, //INPUT: x-position in the inertial frame
33     const double yI, //INPUT: y-position in the inertial frame
34     const double zI, //INPUT: z-position in the inertial frame
35     const double e0, //INPUT: scalar part of the quaternion orientation in the inertial frame
36     const double e1, //INPUT: x-axis part of the quaternion orientation in the inertial frame
37     const double e2, //INPUT: y-axis part of the quaternion orientation in the inertial frame
38     const double e3 //INPUT: z-axis part of the quaternion orientation in the inertial frame
39 );
```

```

40
41
42 //Normalize a 4d array
43 extern void vec4_norm(
44     double C[4], //OUTPUT: normalized 4d array
45     const double a[4] //INPUT: a 4d array to be normalized
46 );
47
48 //Normalize a 3d array
49 extern void vec3_norm(
50     double C[3], //OUTPUT: normalized 3d array
51     const double a[3] //INPUT: a 3d array to be normalized
52 );
53
54 //Matrix vector multiplication c = A*b
55 extern void Matrix3_Vec3_Multiplication(
56     double c[3], //OUTPUT: output vector c = A*b
57     const double A[3][3], //INPUT: input matrix c = A*b
58     const double b[3] //INPUT: input vector c = A*b
59 );
60
61 //Multiply a vector by a scalar c = b * a
62 extern void Vec3_Scalar_Multiplication(
63     double c[3], //OUTPUT: output vector c = b * a
64     const double b[3], //INPUT: input vector c = b*a
65     const double a //INPUT: scalar multiplier c = b*a
66 );
67
68 //Calculate the cross product between two 3d arrays
69 extern void vec3_cross(
70     double C[3], //The cross product C = aXb
71     const double a[3], //The 3D vector a in C=aXb
72     const double b[3] //The 3D vector b in C=aXb
73 );
74
75 //Add two 3d vectors c=a+b
76 extern void vec3_addition(
77     double c[3], //OUTPUT: output vector c = a + b
78     const double a[3], //INPUT: input vector c = a + b
79     const double b[3] //INPUT: input vector c = a + b
80 );
81
82 //subtract one 3d vector from another c=a-b
83 extern void vec3_subtraction(
84     double c[3], //OUTPUT: output vector c = a - b
85     const double a[3], //INPUT: input vector c = a - b
86     const double b[3] //INPUT: input vector c = a - b
87 );
88
89 //Perform the quaternion derivative dq = 1/2*[S]*omega
90 extern void QuaternionTimeDerivative(
91     double dq[4], //OUTPUT: the quaternion time derivative
92     const double q[4], //INPUT: the quaternion
93     const double omega[3] //INPUT: [rad/s] angular velocity
94 );
95
96 //Calculate the gradient of the acceleration and magnetometer cost function V
97 //see Manon Kok and Thomas B. Schon, "A Fast and Robust Algorithm for Orientation Estimation
98 // using Inertial Sensors", IEEE Signal Processing Letters, 2019. DOI: 10.1109 / LSP.2019.2943995
99 extern void KokSchonGradient(
100     double gradient[3], //OUTPUT: the gradient of the magnetometer cost function
101     const double NormalizedAccelerometer[3], //INPUT: [-] x-front, y-right, z-down normalized accelerometer signal
102     const double NormalizedMagnetometer[3], //INPUT: [-] x-front, y-right, z-down normalized magnetometer signal
103     const double quaternion[4], //INPUT: Normalized quaternion orientation of the body in the inertial frame
104     const double inclinationAngleDegrees, //INPUT: [degrees] the geomagnetic inclination angle
105     const double alpha //INPUT: [1,2] tuning parameter for magnetometer trust weighting
106 );
107
108 extern void KokSchonUpdate(
109     double quaternion_new[4], //OUTPUT: updated quaternion orientation
110     const double quaternion[4], //INPUT: previous quaternion orientation
111     const double gradient[3], //INPUT: Kok Schon Gradient vector
112     const double gyrometer[3], //INPUT: [rad/s] 3-axis gyro data with bias removed
113     const double dt, //INPUT: [s] time-step
114     const double beta //INPUT: [< 1] small-valued tuning parameter
115 );
116
117 //The Kok Schon Quaternion Estimator Algorithm

```

```

118     extern int function_KokSchonQuaternionEstimator(
119         double quaternion_new[4], //OUTPUT: updated quaternion orientation
120         const double quaternion[4], //INPUT: previous quaternion orientation
121         const double gyrometer[3], //INPUT: [rad/s] 3-axis gyro data with bias removed
122         const double gravity[3], //INPUT: [any] x-front, y-right, z-down body-frame gravity vector
123         const double magnetometer[3], //INPUT: [any] x-front, y-right, z-down calibrated magnetometer signal any units,
124         const double dt, //INPUT: [s] time-step
125         const double inclinationAngleDegrees, //INPUT: [degrees] the geomagnetic inclination angle
126         const double alpha, //INPUT: [1,2] tuning parameter for magnetometer trust weighting
127         const double beta //INPUT: [< 1] small-valued positive tuning parameter
128     );
129

```

```

1  #include "function_CppKokSchonQuaternionEstimator.h"
2
3  //Create a cross-product matrix
4  extern void CrossMatrix(
5      double CrossMat[3][3], //OUTPUT: Cross product matrix
6      const double vec3[3] //INPUT: vector to be converted to a cross product matrix
7  ) {
8
9      CrossMat[0][0] = 0.0f;
10     CrossMat[0][1] = -vec3[2];
11     CrossMat[0][2] = vec3[1];
12     CrossMat[1][0] = vec3[2];
13     CrossMat[1][1] = 0.0f;
14     CrossMat[1][2] = -vec3[0];
15     CrossMat[2][0] = -vec3[1];
16     CrossMat[2][1] = vec3[0];
17     CrossMat[2][2] = 0.0f;
18 }
19
20
21 //Inertial to body rotation matrix
22 extern void Rotation_I2b(
23     double Ri2b[3][3], //OUTPUT: Rotation matrix from the inertial to body frame (Rows first, then columns)
24     const double e0, //INPUT: scalar part of the quaternion orientation in the inertial frame
25     const double e1, //INPUT: x-axis part of the quaternion orientation in the inertial frame
26     const double e2, //INPUT: y-axis part of the quaternion orientation in the inertial frame
27     const double e3 //INPUT: z-axis part of the quaternion orientation in the inertial frame
28 ) {
29
30     /*local variables*/
31     double e0_2 = e0 * e0;
32     double e1_2 = e1 * e1;
33     double e2_2 = e2 * e2;
34     double e3_2 = e3 * e3;
35     double e0e1 = e0 * e1;
36     double e0e2 = e0 * e2;
37     double e0e3 = e0 * e3;
38     double e1e2 = e1 * e2;
39     double e1e3 = e1 * e3;
40     double e2e3 = e2 * e3;
41
42     Ri2b[0][0] = (e0_2 + e1_2 - e2_2 - e3_2);
43     Ri2b[0][1] = 2.0f * (e0e3 + e1e2);
44     Ri2b[0][2] = 2.0f * (e1e3 - e0e2);
45     Ri2b[1][0] = 2.0f * (e1e2 - e0e3);
46     Ri2b[1][1] = (e0_2 - e1_2 + e2_2 - e3_2);
47     Ri2b[1][2] = 2.0f * (e0e1 + e2e3);
48     Ri2b[2][0] = 2.0f * (e0e2 + e1e3);
49     Ri2b[2][1] = 2.0f * (e2e3 - e0e1);
50     Ri2b[2][2] = (e0_2 - e1_2 - e2_2 + e3_2);
51
52 }
53
54 //Rotate a vec3 from the inertial frame to the body frame
55 extern void inertial_2_body_Rotation(
56     double* x_b, //OUTPUT: x-position in the body frame
57     double* y_b, //OUTPUT: y-position in the body frame
58     double* z_b, //OUTPUT: z-position in the body frame
59     const double xI, //INPUT: x-position in the inertial frame
60     const double yI, //INPUT: y-position in the inertial frame
61     const double zI, //INPUT: z-position in the inertial frame
62     const double e0, //INPUT: scalar part of the quaternion orientation in the inertial frame
63     const double e1, //INPUT: x-axis part of the quaternion orientation in the inertial frame
64     const double e2, //INPUT: y-axis part of the quaternion orientation in the inertial frame
65     const double e3 //INPUT: z-axis part of the quaternion orientation in the inertial frame
66 ) {
67
68     /*local variables*/
69     double e0_2 = e0 * e0;
70     double e1_2 = e1 * e1;
71     double e2_2 = e2 * e2;
72     double e3_2 = e3 * e3;
73     double e0e1 = e0 * e1;
74     double e0e2 = e0 * e2;

```

```

75     double e0e3 = e0 * e3;
76     double e1e2 = e1 * e2;
77     double e1e3 = e1 * e3;
78     double e2e3 = e2 * e3;
79
80     *x_b = (e0_2 + e1_2 - e2_2 - e3_2) * xI +
81           2.0f * (e0e3 + e1e2) * yI +
82           2.0f * (e1e3 - e0e2) * zI;
83     *y_b = 2.0f * (e1e2 - e0e3) * xI +
84           (e0_2 - e1_2 + e2_2 - e3_2) * yI +
85           2.0f * (e0e1 + e2e3) * zI;
86     *z_b = 2.0f * (e0e2 + e1e3) * xI +
87           2.0f * (e2e3 - e0e1) * yI +
88           (e0_2 - e1_2 - e2_2 + e3_2) * zI;
89 }
90
91
92 //Rotate a vec3 from the body frame to the inertial frame
93 extern void body_2_Inertial_Rotation(
94     double* xI, //OUTPUT: x-position in the inertial frame
95     double* yI, //OUTPUT: y-position in the inertial frame
96     double* zI, //OUTPUT: z-position in the inertial frame
97     const double xb, //INPUT: x-position in the body frame
98     const double yb, //INPUT: y-position in the body frame
99     const double zb, //INPUT: z-position in the body frame
100     const double e0, //INPUT: scalar part of the quaternion orientation in the inertial frame
101     const double e1, //INPUT: x-axis part of the quaternion orientation in the inertial frame
102     const double e2, //INPUT: y-axis part of the quaternion orientation in the inertial frame
103     const double e3 //INPUT: z-axis part of the quaternion orientation in the inertial frame
104 ) {
105     /*local variables*/
106     double e0_2 = e0 * e0;
107     double e1_2 = e1 * e1;
108     double e2_2 = e2 * e2;
109     double e3_2 = e3 * e3;
110     double e0e1 = e0 * e1;
111     double e0e2 = e0 * e2;
112     double e0e3 = e0 * e3;
113     double e1e2 = e1 * e2;
114     double e1e3 = e1 * e3;
115     double e2e3 = e2 * e3;
116
117     *xI = (e0_2 + e1_2 - e2_2 - e3_2) * xb +
118           2.0f * (-e0e3 + e1e2) * yb +
119           2.0f * (e1e3 + e0e2) * zb;
120     *yI = 2.0f * (e1e2 + e0e3) * xb +
121           (e0_2 - e1_2 + e2_2 - e3_2) * yb +
122           2.0f * (-e0e1 + e2e3) * zb;
123     *zI = 2.0f * (-e0e2 + e1e3) * xb +
124           2.0f * (e2e3 + e0e1) * yb +
125           (e0_2 - e1_2 - e2_2 + e3_2) * zb;
126 }
127
128
129 //Normalize a 4d array
130 extern void vec4_norm(
131     double C[4], //OUTPUT: normalized 4d array
132     const double a[4] //INPUT: a 4d array to be normalized
133 ) {
134     //Calculate the denominator
135     double den = sqrt(
136         a[0] * a[0] +
137         a[1] * a[1] +
138         a[2] * a[2] +
139         a[3] * a[3]);
140
141     //Avoid dividing by zero
142     if (den > 0.000001f)
143     {
144         //Normalize the array
145         C[0] = a[0] / den;
146         C[1] = a[1] / den;
147         C[2] = a[2] / den;
148         C[3] = a[3] / den;
149     }
150     else
151     {
152

```

```

153         //The array is numerically zero and cannot be normalized
154         C[0] = a[0];
155         C[1] = a[1];
156         C[2] = a[2];
157         C[3] = a[3];
158     }
159 }
160
161
162 //Normalize a 3d array
163 extern void vec3_norm(
164     double C[3], //OUTPUT: normalized 3d array
165     const double a[3] //INPUT: a 3d array to be normalized
166 )
167 {
168     //Calculate the denominator
169     double den = sqrt(
170         a[0] * a[0] +
171         a[1] * a[1] +
172         a[2] * a[2]);
173
174     //Avoid dividing by zero
175     if (den > 0.000001)
176     {
177         //Normalize the array
178         C[0] = a[0] / den;
179         C[1] = a[1] / den;
180         C[2] = a[2] / den;
181     }
182     else
183     {
184         //The array is numerically zero and cannot be normalized
185         C[0] = a[0];
186         C[1] = a[1];
187         C[2] = a[2];
188     }
189 }
190
191 double norm3(double x[3]) {
192     return sqrt(x[0] * x[0] + x[1] * x[1] + x[2] * x[2]);
193 }
194
195 //Matrix vector multiplication c = A*b
196 extern void Matrix3_Vec3_Multiplication(
197     double c[3], //OUTPUT: output vector c = A*b
198     const double A[3][3], //INPUT: input matrix c = A*b
199     const double b[3] //INPUT: input vector c = A*b
200 ) {
201     for (int ii = 0; ii < 3; ii++) {
202         c[ii] = 0.0f;
203         for (int jj = 0; jj < 3; jj++) {
204             c[ii] += A[ii][jj] * b[jj];
205         }
206     }
207 }
208
209 //Multiply a vector by a scalar c = b * a
210 extern void Vec3_Scalar_Multiplication(
211     double c[3], //OUTPUT: output vector c = b * a
212     const double b[3], //INPUT: input vector c = b*a
213     const double a //INPUT: scalar multiplier c = b*a
214 ) {
215     for (int ii = 0; ii < 3; ii++) {
216         c[ii] = b[ii] * a;
217     }
218 }
219
220 //Calculate the cross product between two 3d arrays
221 extern void vec3_cross(
222     double C[3], //The cross product C = aXb
223     const double a[3], //The 3D vector a in C=aXb
224     const double b[3] //The 3D vector b in C=aXb
225 )
226 {
227     //Calculate the cross product C = aXb
228     C[0] = a[1] * b[2] - b[1] * a[2];
229     C[1] = a[2] * b[0] - b[2] * a[0];
230     C[2] = a[0] * b[1] - b[0] * a[1];

```



```

231 }
232
233 //Add two 3d vectors c=a+b
234 extern void vec3_addition(
235     double c[3], //OUTPUT: output vector c = a + b
236     const double a[3], //INPUT: input vector c = a + b
237     const double b[3] //INPUT: input vector c = a + b
238 ) {
239     for (int ii = 0; ii < 3; ii++) {
240         c[ii] = a[ii] + b[ii];
241     }
242 }
243
244 //subtract one 3d vector from another c=a-b
245 extern void vec3_subtraction(
246     double c[3], //OUTPUT: output vector c = a - b
247     const double a[3], //INPUT: input vector c = a - b
248     const double b[3] //INPUT: input vector c = a - b
249 ) {
250     for (int ii = 0; ii < 3; ii++) {
251         c[ii] = a[ii] - b[ii];
252     }
253 }
254
255 //Perform the quaternion derivative dq = 1/2*[S]*omega
256 extern void QuaternionTimeDerivative(
257     double dq[4], //OUTPUT: the quaternion time derivative
258     const double q[4], //INPUT: the quaternion
259     const double omega[3] //INPUT: [rad/s] angular velocity
260 ) {
261     dq[0] = -q[1] * omega[0] - q[2] * omega[1] - q[3] * omega[2];
262     dq[1] = q[0] * omega[0] - q[3] * omega[1] + q[2] * omega[2];
263     dq[2] = q[3] * omega[0] + q[0] * omega[1] - q[1] * omega[2];
264     dq[3] = -q[2] * omega[0] + q[1] * omega[1] + q[0] * omega[2];
265 }
266
267 //Calculate the gradient of the acceleration and magnetometer cost function V
268 //see Manon Kok and Thomas B. Schon, "A Fast and Robust Algorithm for Orientation Estimation
269 //using Inertial Sensors", IEEE Signal Processing Letters, 2019. DOI: 10.1109 / LSP.2019.2943995
270 extern void KokSchonGradient(
271     double gradient[3], //OUTPUT: the gradient of the magnetometer cost function
272     const double NormalizedAccelerometer[3], //INPUT: [-] x-front, y-right, z-down normalized accelerometer signal
273     const double NormalizedMagnetometer[3], //INPUT: [-] x-front, y-right, z-down normalized magnetometer signal
274     const double quaternion[4], //INPUT: Normalized quaternion orientation of the body in the inertial frame
275     const double inclinationAngleDegrees, //INPUT: [degrees] the geomagnetic inclination angle
276     const double alpha //INPUT: [1,2] tuning parameter for magnetometer trust weighting
277 ) {
278     double Ri2b[3][3];
279     double gravityInBodyFrame[3];
280     double MagInBodyFrame[3];
281     double inclinationAngleRadians = inclinationAngleDegrees * PI / 180.0f;
282     double cos_incAngle = cos(inclinationAngleRadians);
283     double sin_incAngle = sin(inclinationAngleRadians);
284     double accelMinusGravity[3];
285     double MagMinusMag[3];
286     double GravityCrossGravityError[3];
287     double MagCrossMagError[3];
288
289     //Get the rotation matrix from the inertial frame to the body frame
290     Rotation_I2b(
291         Ri2b, //OUTPUT: Rotation matrix from the inertial to body frame (Rows first, then columns)
292         quaternion[0], //INPUT: scalar part of the quaternion orientation in the inertial frame
293         quaternion[1], //INPUT: x-axis part of the quaternion orientation in the inertial frame
294         quaternion[2], //INPUT: y-axis part of the quaternion orientation in the inertial frame
295         quaternion[3] //INPUT: z-axis part of the quaternion orientation in the inertial frame
296     );
297
298     //Get the gravity unit vector in the body frame
299     for (int ii = 0; ii < 3; ii++)
300         gravityInBodyFrame[ii] = Ri2b[ii][2];
301
302     //Get the geomagnetic unit vector in the body frame
303     for (int ii = 0; ii < 3; ii++)
304         MagInBodyFrame[ii] = Ri2b[ii][0] * cos_incAngle + Ri2b[ii][2] * sin_incAngle;
305
306     //subtract gravity from the accelerometer signal (i.e. gravity error)
307     for (int ii = 0; ii < 3; ii++)
308         accelMinusGravity[ii] = NormalizedAccelerometer[ii] - gravityInBodyFrame[ii];

```

```

309
310 //get the magnetometer error
311 for (int ii = 0; ii < 3; ii++)
312     MagMinusMag[ii] = NormalizedMagnetometer[ii] - MagInBodyFrame[ii];
313
314 //Get the cross-product between the gravity and the gravity error
315 vec3_cross(GravityCrossGravityError, gravityInBodyFrame, accelMinusGravity);
316
317 //Get the cross-product between the body geomagnetic vector and its error
318 vec3_cross(MagCrossMagError, MagInBodyFrame, MagMinusMag);
319
320 //The Kok Schon gradient is the sum of the cross products
321 for (int ii = 0; ii < 3; ii++)
322     gradient[ii] = GravityCrossGravityError[ii] + alpha * MagCrossMagError[ii];
323 }
324
325 extern void KokSchonUpdate(
326     double quaternion_new[4], //OUTPUT: updated quaternion orientation
327     const double quaternion[4], //INPUT: previous quaternion orientation
328     const double gradient[3], //INPUT: Kok Schon Gradient vector
329     const double gyrometer[3], //INPUT: [rad/s] 3-axis gyro data with bias removed
330     const double dt, //INPUT: [s] time-step
331     const double beta //INPUT: [< 1] small-valued tuning parameter
332 ) {
333     double q0 = quaternion[0];
334     double q1 = quaternion[1];
335     double q2 = quaternion[2];
336     double q3 = quaternion[3];
337
338     //get the magnitude of the gradient
339     double normGrad = sqrt(max(0.0f, gradient[0] * gradient[0] +
340         gradient[1] * gradient[1] +
341         gradient[2] * gradient[2]));
342
343     double omega[3]; //Corrected angular velocity (rad/s)
344     //Avoid the possibility of dividing by zero
345     if (normGrad > 0.0001f) {
346         //Normal case
347         //Get the corrected angular velocity (rad/s)
348         for (int ii = 0; ii < 3; ii++)
349             omega[ii] = gyrometer[ii] - beta * gradient[ii] / normGrad;
350     }
351     else {
352         //should never get here
353         //Set the corrected angular velocity to the measured value (rad/s)
354         for (int ii = 0; ii < 3; ii++)
355             omega[ii] = gyrometer[ii];
356     }
357
358     //Update the quaternion estimate
359     double qu[4]; //updated (but not normalized) quaternion
360     qu[0] = q0 + 0.5f * dt * (-q1 * omega[0] - q2 * omega[1] - q3 * omega[2]);
361     qu[1] = q1 + 0.5f * dt * (q0 * omega[0] - q3 * omega[1] + q2 * omega[2]);
362     qu[2] = q2 + 0.5f * dt * (q3 * omega[0] + q0 * omega[1] - q1 * omega[2]);
363     qu[3] = q3 + 0.5f * dt * (-q2 * omega[0] + q1 * omega[1] + q0 * omega[2]);
364
365     //Get the magnitude of the quaternion
366     double norm_qu = sqrt(max(0.0f, qu[0] * qu[0] +
367         qu[1] * qu[1] +
368         qu[2] * qu[2] +
369         qu[3] * qu[3]));
370
371     //Avoid division by zero
372     if (norm_qu > 0.001)
373         //Update the quaternion
374         for (int ii = 0; ii < 4; ii++)
375             quaternion_new[ii] = qu[ii] / norm_qu;
376     else {
377         //should never get to this case, reset the quaternion
378         quaternion_new[0] = 1.0f;
379         quaternion_new[1] = 0.0f;
380         quaternion_new[2] = 0.0f;
381         quaternion_new[3] = 0.0f;
382     }
383 }
384
385 //The Kok Schon Quaternion Estimator Algorithm
386

```

```

387 extern int function_KokSchonQuaternionEstimator(
388     double quaternion_new[4], //OUTPUT: updated quaternion orientation
389     const double quaternion[4], //INPUT: previous quaternion orientation
390     const double gyrometer[3], //INPUT: [rad/s] 3-axis gyro data with bias removed
391     const double gravity[3], //INPUT: [any] x-front, y-right, z-down body-frame gravity vector
392     const double magnetometer[3], //INPUT: [any] x-front, y-right, z-down calibrated magnetometer signal any units,
393     const double dt, //INPUT: [s] time-step
394     const double inclinationAngleDegrees, //INPUT: [degrees] the geomagnetic inclination angle
395     const double alpha, //INPUT: [1,2] tuning parameter for magnetometer trust weighting
396     const double beta //INPUT: [< 1] small-valued positive tuning parameter
397 ) {
398
399     //extract the quaternion components
400     double e0 = quaternion[0];
401     double e1 = quaternion[1];
402     double e2 = quaternion[2];
403     double e3 = quaternion[3];
404
405     //Ensure that the quaternion is a unit quaternion
406     double norm_q = sqrt(e0 * e0 + e1 * e1 + e2 * e2 + e3 * e3);
407     if (fabs(norm_q - 1.0f) > 0.0001f) {
408         //The quaternion is not a valid unit quaternion, normalize it
409         if (norm_q > 0.0001f) {
410             e0 = e0 / norm_q;
411             e1 = e1 / norm_q;
412             e2 = e2 / norm_q;
413             e3 = e3 / norm_q;
414         }
415         else
416         {
417             //Avoid dividing by zero
418             e0 = 1.0f;
419             e1 = 0.0f;
420             e2 = 0.0f;
421             e3 = 0.0f;
422         }
423     }
424
425     //Get the inertial-frame to body-frame rotation matrix
426     double RI2b[3][3];
427     Rotation_I2b(RI2b, e0, e1, e2, e3);
428
429     //get a gravity unit vector [unitless]
430     double unitGravity[3];
431     vec3_norm(unitGravity, gravity);
432
433     //Normalize the magnetometer data
434     double unitMag[3];
435     vec3_norm(unitMag, magnetometer); //unitless
436
437     //Get the Kok Schon gradient
438     double gradient[3];
439     //get the Kok Schon Gradient
440     KokSchonGradient(
441         gradient, //OUTPUT: the gradient of the magnetometer cost function
442         unitGravity, //INPUT: [-] x-front, y-right, z-down normalized accelerometer signal
443         unitMag, //INPUT: [-] x-front, y-right, z-down normalized magnetometer signal
444         quaternion, //INPUT: Normalized quaternion orientation of the body in the inertial frame
445         inclinationAngleDegrees, //INPUT: [degrees] the geomagnetic inclination angle
446         alpha //INPUT: [1,2] tuning parameter for magnetometer trust weighting
447     );
448
449     //Get the Kok Schon estimate of the quaternion
450     KokSchonUpdate(
451         quaternion_new, //OUTPUT: updated quaternion orientation
452         quaternion, //INPUT: previous quaternion orientation
453         gradient, //INPUT: Kok Schon Gradient vector
454         gyrometer, //INPUT: [rad/s] 3-axis gyro data with bias removed
455         dt, //INPUT: [s] time-step
456         beta //INPUT: [< 1] small-valued tuning parameter
457     );
458
459
460     //If any output is infinite or NaN, reset everything
461     if (isinf(quaternion_new[0]) || isnan(quaternion_new[0])
462         || isinf(quaternion_new[1]) || isnan(quaternion_new[1])
463         || isinf(quaternion_new[2]) || isnan(quaternion_new[2])
464         || isinf(quaternion_new[3]) || isnan(quaternion_new[3]))

```

```
465     || isnf(quaternion_new[3]) || isnan(quaternion_new[3])
466     )
467     {
468         //Some problem happened, reset everything
469         //Reset quaternions
470         quaternion_new[0] = 1.0f;
471         for (unsigned int ii = 1; ii < 4; ii++)
472             quaternion_new[ii] = 0.0f;
473
474         return 1; //indicate that something failed
475     }
476     else {
477         return 0; //normal case
478     }
479 }
480
```

```

function_Auto...ointTracker.h  x  function_Auton...intTracker.cpp
Miscellaneous Files - No Configurations  (Global Scope)
1  #pragma once
2
3  #include <math.h>
4  #define pi (3.14159265359f)
5  #define max(a,b) ((a<b)?b:a)
6  #define min(a,b) ((a>b)?b:a)
7  #define sign(a) ((a<0)?-1.0:1.0)
8
9
10
11 void function_Autonomous_GPS_WaypointTracker(
12     double* delta_t, //OUTPUT: [0,1] throttle command
13     double* delta_e, //OUTPUT: [-1,1] elevator command
14     double* delta_a, //OUTPUT: [-1,1] aileron command
15     double* delta_r, //OUTPUT: [-1,1] rudder command
16     const double q[4], //INPUT: [e0;e1;e2;e3] (-1 to 1) quaternion orientation
17     const double gyro[3], //INPUT: [p;q;r] (rad/s) gyrometer signals
18     const double lat_target, //INPUT: [lat](deg) target latitude angle
19     const double lon_target, //INPUT: [lon](deg) target longitude angle
20     const double alt_target, //INPUT: [-zI](m) target altitude
21     const double latitude, //INPUT: [lat](deg) current latitude angle
22     const double longitude, //INPUT: [lon](deg) current longitude angle
23     const double altitude, //INPUT: [-zI](m) current altitude
24     const double kp_roll, //INPUT: [PID gain] proportional gain for roll control
25     const double kd_roll, //INPUT: [PID gain] derivative gain for roll control
26     const double kp_pitch, //INPUT: [PID gain] proportional gain for pitch control
27     const double kp_yaw, //INPUT: [PID gain] proportional gain for yaw control
28     const double kp_rudder, //INPUT: [PID gain] proportional gain for yaw control
29     const double max_roll, //INPUT: [limit](rad) max roll angle limit
30     const double max_pitch //INPUT: [limit](rad) max pitch angle limit
31 );

```

```
1  #include "function_Autonomous_GPS_WaypointTracker.h"
2
3  static double cosd(double angleDegrees) {
4      return cos(pi / 180.0 * angleDegrees);
5  }
6
7  static double norm3(double x[3]) {
8      return sqrt(x[0] * x[0] + x[1] * x[1] + x[2] * x[2]);
9  }
10
11  static void ConvertGPS(
12      double dXI[3],
13      const double LatTarget,
14      const double LonTarget,
15      const double AltTarget,
16      const double GPS_lat,
17      const double GPS_lon,
18      const double alt
19  )
20  {
21      const double rEarth = 6371000.0; //(m) earth's radius
22      dXI[0] = (rEarth * pi / 180.0) * (LatTarget - GPS_lat); //xI displacement
23      double dLon = LonTarget - GPS_lon;
24      if (fabs(dLon) > 180.0) {
25          if (LonTarget > GPS_lon)
26              dLon = LonTarget - (GPS_lon + 360.0);
27          else
28              dLon = (360.0 + LonTarget) - GPS_lon;
29      }
30      //yI displacement
31      dXI[1] = sign(dLon) * rEarth * acos((cosd(dLon) - 1.0) * (cosd(GPS_lat) * cosd(GPS_lat)) + 1.0);
32      //zI displacement
33      dXI[2] = alt - AltTarget;
34  }
35
36  static void processWaypoints(
37      double* pitchDes,
38      double* yawDes,
39      const double target_lat,
40      const double target_lon,
41      const double target_alt,
42      const double GPS_lat,
43      const double GPS_lon,
44      const double alt
45  )
46  {
47      double dXI[3] = { 0.0 };
48      //Get the distance to the next waypoint
49      ConvertGPS(dXI,
50          target_lat,
51          target_lon,
52          target_alt,
53          GPS_lat,
54          GPS_lon,
55          alt
56      );
57
58      //find the absolute distance
59      double distance = norm3(dXI);
60
61      //get the desired pitch angle (rad)
62      *pitchDes = atan(-dXI[2] / max(1.0, distance));
63      //get the desired yaw angle (rad)
64      *yawDes = atan2(dXI[1], dXI[0]);
65  }
66
67
68  static void Quaternion2Euler(
69      double* roll,
70      double* pitch,
71      double* yaw,
72      const double q[4]) {
73      double e0 = q[0];
```



```

74     double e1 = q[1];
75     double e2 = q[2];
76     double e3 = q[3];
77     *roll = atan2(2.0 * (e0 * e1 + e2 * e3), (e0 * e0 + e3 * e3 - e1 * e1 - e2 * e2));
78     *pitch = asin(max(-1.0, min(1.0, 2.0 * (e0 * e2 - e1 * e3))));
79     *yaw = atan2(2.0 * (e0 * e3 + e1 * e2), (e0 * e0 + e1 * e1 - e2 * e2 - e3 * e3));
80 }
81
82 //Get the shortest angle difference
83 static double getAngleError(
84     double xd, //(rad) desired angle
85     double x //(rad) actual angle
86 ) {
87     //get the difference between the angles
88     double diff = xd - x; //(rad)
89
90     //Get the shortest absolute angle between the two angles
91     double d = acos(cos(xd) * cos(x) + sin(xd) * sin(x));
92
93     double dx; //shortest angle
94     //Check whether they are different
95     if (fabs(diff) - d > 0.1) {
96         //they are different, use the shortest absolute angle
97         dx = -sign(diff) * d;
98     }
99     else {
100         //they are the same
101         dx = diff;
102     }
103     return dx;
104 }
105
106 static void getCommands(
107     double* delta_t,
108     double* delta_e,
109     double* delta_a,
110     double* delta_r,
111     const double roll,
112     const double pitch,
113     const double yaw,
114     const double pitchDes,
115     const double yawDes,
116     const double gyro[3],
117     const double kp_roll, //INPUT: [PID gain] proportional gain for roll control
118     const double kd_roll, //INPUT: [PID gain] derivative gain for roll control
119     const double kp_pitch, //INPUT: [PID gain] proportional gain for pitch control
120     const double kp_yaw, //INPUT: [PID gain] proportional gain for yaw control
121     const double kp_rudder, //INPUT: [PID gain] proportional gain for yaw control
122     const double max_roll, //INPUT: [limit](rad) max roll angle limit
123     const double max_pitch //INPUT: [limit](rad) max pitch angle limit
124 ) {
125
126     double yaw_err = getAngleError(yawDes, yaw);
127     double roll_des = kp_yaw * yaw_err;
128     roll_des = max(-max_roll, min(roll_des, max_roll));
129
130     double roll_err = getAngleError(roll_des, roll);
131
132     double pitch_set = max(-max_pitch, min(pitchDes, max_pitch));
133     double pitch_err = getAngleError(pitch_set, pitch);
134
135     *delta_a = kp_roll * roll_err - kd_roll * gyro[0];
136
137     *delta_r = max(-1.0, min(kp_rudder * yaw_err, 1.0));
138     *delta_e = kp_pitch * pitch_err;
139     *delta_t = 0.8;
140
141     *delta_a = max(-1.0, min(*delta_a, 1.0));
142     *delta_e = max(-1.0, min(*delta_e, 1.0));
143 }
144
145 void function_Autonomous_GPS_waypointTracker(
146     double* delta_t, //OUTPUT: [0,1] throttle command
147     double* delta_e, //OUTPUT: [-1,1] elevator command
148     double* delta_a, //OUTPUT: [-1,1] aileron command
149     double* delta_r, //OUTPUT: [-1,1] rudder command
150     const double q[4], //INPUT: [e0;e1;e2;e3] (-1 to 1) quaternion orientation

```

```

152 const double gyro[3], //INPUT: [p;q;r] (rad/s) gyrometer signals
153 const double lat_target, //INPUT: [lat](deg) target latitude angle
154 const double lon_target, //INPUT: [lon](deg) target longitude angle
155 const double alt_target, //INPUT: [-zI](m) target altitude
156 const double latitude, //INPUT: [lat](deg) current latitude angle
157 const double longitude, //INPUT: [lon](deg) current longitude angle
158 const double altitude, //INPUT: [-zI](m) current altitude
159 const double kp_roll, //INPUT: [PID gain] proportional gain for roll control
160 const double kd_roll, //INPUT: [PID gain] derivative gain for roll control
161 const double kp_pitch, //INPUT: [PID gain] proportional gain for pitch control
162 const double kp_yaw, //INPUT: [PID gain] proportional gain for yaw control
163 const double kp_rudder, //INPUT: [PID gain] proportional gain for yaw control
164 const double max_roll, //INPUT: [limit](rad) max roll angle limit
165 const double max_pitch //INPUT: [limit](rad) max pitch angle limit
166 } {
167     //get the estimated roll pitch and yaw angles
168     double roll, pitch, yaw;
169     Quaternion2Euler(&roll, &pitch, &yaw, q);
170
171     //Process the waypoints
172     double pitchDes, yawDes;
173     processWaypoints(&pitchDes, &yawDes,
174         lat_target, lon_target, alt_target,
175         latitude, longitude, altitude);
176
177     //Get the control commands
178     getCommands(delta_t, delta_e, delta_a, delta_r,
179         roll, pitch, yaw, pitchDes, yawDes, gyro,
180         kp_roll, kd_roll, kp_pitch, kp_yaw, kp_rudder, max_roll, max_pitch);
181 }
182

```



```

1  #include <SPI.h>
2  #include <SD.h>
3  #include <MPU9250_WE.h> //Include the MPU9250 library for Accel, Gyro, and Magnetom
4  #include <BMP280_DEV.h> //Include the BMP280_DEV.h library for altitude, pressure,
5  #include <function_CppKokSchonQuaternionEstimator.h> //Include the Kok Schon Quaternion Estimator
6  #include <Quaternion2Euler.h> //Include the library to convert quaternion to euler orient
7  #include <SoftwareSerial.h> //Include the SoftwareSerial library for GPS communication
8  #include <TinyGPS.h> //Include the GPS library
9  #include <Servo.h> //Include the Servo library
10 #include <function_Autonomous_GPS_WaypointTracker.h> //Include the GPS waypoint tracking library
11
12 //name the file for the SD card
13 String filename = "FlightData001.csv";
14 //Say whether to collect GPS data or not
15 bool useGPS = true;
16
17 //#####Variables for GPS#####//
18 TinyGPS gps;
19 const int rxPin = 1;
20 const int txPin = 0;
21 SoftwareSerial ss(rxPin, txPin);
22 static void readGPS(unsigned long ms);
23 float GPSSlat = 0.0; //declare GPS latitude and longitude variables
24 float GPSSlon = 0.0;
25 float GPSSspeed = 0.0; //declare GPS speed (m/s)
26 double target_lat = 83.820934;
27 double target_lon = -111.785286;
28 double target_alt = 854.0;
29 bool GPSisValid = false;
30 //#####End of Variables for GPS#####//
31
32 //#####Variables for SD Card Reader#####//
33 #if !defined(ARDUINO_ARCH_RP2040)
34 #error For RP2040 only
35 #endif
36 #if defined(ARDUINO_ARCH_MBED)
37 #define PIN_SD_MOSI PIN_SPI_MOSI
38 #define PIN_SD_MISO PIN_SPI_MISO
39 #define PIN_SD_SCK PIN_SPI_SCK
40 #define PIN_SD_SS PIN_SPI_SS
41 #else
42 #define PIN_SD_MOSI PIN_SPI0_MOSI
43 #define PIN_SD_MISO PIN_SPI0_MISO
44 #define PIN_SD_SCK PIN_SPI0_SCK
45 #define PIN_SD_SS PIN_SPI0_SS
46 #endif
47 #define _RP2040_SD_LOGLEVEL_ 0
48 //#####End of Variables for SD Card Reader#####//
49
50 //#####Variables for 10 DOF MPU9250 Sensor#####//
51 #define MPU9250_ADDR 0x68 //define the address for the MPU 9250
52 //Create the MPU9250 object and name it myMPU9250
53 MPU9250_WE myMPU9250 = MPU9250_WE(MPU9250_ADDR);
54 //Create the BMP280_DEV object, name it bmp280. I2C address is 0x77
55 BMP280_DEV bmp280;
56 //declare temperature, pressure, and altitude
57 float temperature, pressure, altitude;
58 //#####End of Variables for 10 DOF MPU9250 Sensor#####//
59
60
61 //#####Variables for the Servo Motors#####//
62 Servo AileronServo;
63 Servo ElevatorServo;

```

```

64 Servo RudderServo;
65 const int AileronPin = 12;
66 const int ElevatorPin = 11;
67 const int RudderPin = 10;
68 const double kp_roll = 2.0;
69 const double kd_roll = 0.0;
70 const double kp_pitch = -2.0;
71 const double kp_yaw = 0.5;
72 const double kp_rudder = 0.2;
73 const double max_roll = 3.14 / 4.0;
74 const double max_pitch = 3.14 / 4.0;
75 //#####End of Variables for the Servo Motors#####//
76
77 //Additional variables
78 bool USBconnected = true;
79 double quaternion[4] = { 1.0, 0.0, 0.0, 0.0 };
80 double t_last = 0.0;
81
82 void setup() {
83
84     // Start serial communication
85     Serial.begin(9600);
86     delay(3000);
87     if (!Serial) {
88         USBconnected = false;
89     }
90     if (useGPS) {
91         //Start communication with the GPS module
92         ss.begin(9600);
93         unsigned long GPSdelay = 1000; //Time delay in ms
94         readGPS(GPSdelay);
95     }
96     Wire.begin(); //Begin I2C
97     delay(2000);
98     if (!myMPU9250.init()) { //Start the MPU, if it fails, report an error
99         if (USBconnected) {
100             Serial.println("MPU9250 does not respond");
101         }
102     }
103     if (!myMPU9250.initMagnetometer()) { //Start the magnetometer, if it fails, report
104         if (USBconnected) {
105             Serial.println("Magnetometer does not respond");
106         }
107     }
108     bmp280.begin(); // Default initialization, place the BMP280 into SLEEP_MODE
109     //bmp280.setPresOversampling(OVERSAMPLING_X4); // Set the pressure oversampling to X4
110     //bmp280.setTempOversampling(OVERSAMPLING_X1); // Set the temperature oversampling to X1
111     //bmp280.setIIRFilter(IIR_FILTER_4); // Set the IIR filter to setting 4
112     bmp280.setTimeStandby(TIME_STANDBY_2000MS); // Set the standby time to 2 seconds
113     bmp280.startNormalConversion(); // Start BMP280 continuous conversion in NORMAL_MODE
114
115     if (USBconnected) {
116         Serial.println("Position your MPU9250 flat and don't move it - calibrating...");
117     }
118     delay(1000);
119     myMPU9250.autoOffsets(); //Calibrate the accelerometer and gyro offsets
120     if (USBconnected) {
121         Serial.println("Done!");
122     }
123
124     #if defined(ARDUINO_ARCH_MBED)
125     if (USBconnected) {
126         Serial.print("Starting SD Card ReadWrite on MBED ");
127     }

```

```

128 #else
129     if (USBconnected) {
130         Serial.print("Starting SD Card ReadWrite on ");
131     }
132 #endif
133
134     if (USBconnected) {
135         Serial.println(BOARD_NAME);
136         Serial.print("Initializing SD card with SS = ");
137         Serial.println(PIN_SD_SS);
138         Serial.print("SCK = ");
139         Serial.println(PIN_SD_SCK);
140         Serial.print("MOSI = ");
141         Serial.println(PIN_SD_MOSI);
142         Serial.print("MISO = ");
143         Serial.println(PIN_SD_MISO);
144     }
145
146     if (!SD.begin(PIN_SD_SS)) {
147         if (USBconnected) {
148             Serial.println("Initialization failed!");
149         }
150     } else {
151         for (int ii = 0; ii < 10; ii++) {
152             if (SD.exists(filename)) {
153                 filename = "FlightData";
154                 filename += String(ii);
155                 filename += ".csv";
156             }
157         }
158     }
159     if (USBconnected) {
160         Serial.println("Initialization done.");
161     }
162
163     //create the SD card file and open it for writing
164     File dataFile = SD.open(filename, FILE_WRITE);
165
166     //If it opened correctly, write the header to it
167     if (dataFile) {
168         String headerString = "time,";
169         headerString += "gFx,gFy,gFz,wX,wY,wZ,p,Bx,By,Bz,Azimuth,Pitch,Roll,Latitude,
170         | Longitude,Speed (m/s),delta_e,delta_a,delta_r";
171         dataFile.println(headerString);
172         dataFile.close();
173     } else {
174         if (USBconnected) {
175             Serial.println("Initialization failed to open the file");
176         }
177     }
178
179     //#####Set up the Servos#####
180     AileronServo.attach(AileronPin);
181     ElevatorServo.attach(ElevatorPin);
182     RudderServo.attach(RudderPin);
183
184     t_last = 0.0;
185 }
186
187 void loop() {
188     //Get the values from the accelerometer
189     xyzFloat accel = myMPU9250.getGValues();
190     //Get the values from the Gyrometer

```

```

191 xyzFloat gyro = myMPU9250.getGyrValues();
192 //Get the values from the Magnetometer
193 xyzFloat Mag = myMPU9250.getMagValues();
194
195 //put the data into arrays
196 double gyrometer[3];
197 double gravity[3];
198 double magnetometer[3];
199 gyrometer[0] = gyro.x * 3.14159 / 180.0;
200 gyrometer[1] = -gyro.y * 3.14159 / 180.0;
201 gyrometer[2] = -gyro.z * 3.14159 / 180.0;
202
203 gravity[0] = -accel.x;
204 gravity[1] = accel.y;
205 gravity[2] = accel.z;
206
207 magnetometer[0] = (Mag.y - 40.3058);
208 magnetometer[1] = -(Mag.x + 11.444);
209 magnetometer[2] = (Mag.z + 33.2877);
210
211 //get the time
212 double t = (double)millis() / 1000.0;
213 double dt = t - t_last;
214 t_last = t;
215
216 //Get the quaternion orientation
217 function_KokSchonQuaternionEstimator(
218     quaternion,
219     quaternion,
220     gyrometer,
221     gravity,
222     magnetometer,
223
224     dt,
225     67.0,
226     1.0,
227     0.3);
228
229 double roll, pitch, yaw;
230 Quaternion2Euler(&roll, &pitch, &yaw, quaternion);
231
232 //Get the bmp280 measurements
233 //Temperature in Celsius, pressure in hectopascals, altitude in meters
234 bmp280.getMeasurements(temperature, pressure, altitude);
235
236 //Get GPS measurements
237 if (useGPS) {
238     unsigned long GPSdelay = 1; //Time delay in ms
239     readGPS(GPSdelay);
240     unsigned long age;
241     //Read the latitude and longitude
242     gps.f_get_position(&GPSlat, &GPSlon, &age);
243
244     //check that the GPS signal is valid
245     if (((GPSlat - 43.0) < 2.0) && ((GPSlon + 111.0) < 2) && useGPS) {
246         GPSisValid = true;
247     } else {
248         GPSisValid = false;
249     }
250 }
251
252 //get the control commands
253 double delta_t = 0.0;
254 double delta_e = 0.0;
255 double delta_a = 0.0;

```

```

255 double delta_r = 0.0;
256 if (GPSisValid) {
257     function_Autonomous_GPS_WaypointTracker(
258         &delta_t, &delta_e, &delta_a, &delta_r,
259         quaternion, gyrometer, target_lat, target_lon, target_alt + 2.0,
260         GPSlat, GPSlon, target_alt,
261         kp_roll, kd_roll, kp_pitch, kp_yaw, kp_rudder,
262         max_roll, max_pitch);
263 } else {
264     delta_e = -kp_pitch * pitch;
265     delta_a = -kp_roll * roll;
266     delta_r = 0.0;
267 }
268
269
270 //set the servo positions
271 const double delta_e_trim = -0.13;
272 ElevatorServo.write(mapServoCmd(delta_e - delta_e_trim));
273 AileronServo.write(mapServoCmd(delta_a));
274 RudderServo.write(mapServoCmd(delta_r));
275
276 String dataString = "";
277 dataString += t; //time
278 dataString += ",";
279 dataString += gravity[0]; //gFx
280 dataString += ",";
281 dataString += gravity[1]; //gFy
282 dataString += ",";
283 dataString += gravity[2]; //gFz
284 dataString += ",";
285 dataString += gyrometer[0]; //wx
286 dataString += ",";
287 dataString += gyrometer[1]; //wy
288 dataString += ",";
289 dataString += gyrometer[2]; //wz
290 dataString += ",";
291 dataString += pressure; //p
292 dataString += ",";
293 dataString += magnetometer[0]; //Bx
294 dataString += ",";
295 dataString += magnetometer[1]; //By
296 dataString += ",";
297 dataString += magnetometer[2]; //Bz
298 dataString += ",";
299 dataString += yaw; //Azimuth
300 dataString += ",";
301 dataString += pitch; //Pitch
302 dataString += ",";
303 dataString += roll; //Roll
304 dataString += ",";
305 if (useGPS) {
306     char charArray[12];
307     dtostrf(GPSlat, 1, 6, charArray); //Latitude
308     dataString += charArray;
309     dataString += ",";
310     dtostrf(GPSlon, 1, 6, charArray); //Longitude
311     dataString += charArray;
312     dataString += ",";
313     dataString += GPSSpeed; //Speed (m/s)
314 } else {
315     dataString += "0"; //Latitude
316     dataString += ",";
317     dataString += "0"; //Longitude
318     dataString += ",";

```

```

319 |   dataString += "0"; //Speed (m/s)
320 | }
321 | dataString += ",";
322 | dataString += delta_e;
323 | dataString += ",";
324 | dataString += delta_a;
325 | dataString += ",";
326 | dataString += delta_r;
327 |
328 |   writeToSDcard(dataString);
329 | }
330 |
331 | void writeToSDcard(String dataString) {
332 |   //create the SD card file and open it for writing
333 |   File dataFile = SD.open(filename, FILE_WRITE);
334 |
335 |   //If it opened correctly, write the header to it
336 |   if (dataFile) {
337 |     dataFile.println(dataString);
338 |     dataFile.close();
339 |   } else {
340 |     if (USBconnected) {
341 |       Serial.println("Failed to open the file");
342 |     }
343 |   }
344 | }
345 |
346 |
347 | static void readGPS(unsigned long ms) {
348 |   unsigned long start = millis();
349 |   do {
350 |     while (ss.available())
351 |
352 |       |   gps.encode(ss.read());
353 |   } while (millis() - start < ms);
354 | }
355 |
356 | static int mapServoCmd(double delta) {
357 |   double slope = (135.0 - 90.0) * delta;
358 |   return (((int)slope) + 90.0);
359 | }

```