Quaternion2Euler.h Quaternion2Euler.cpp

Miscellaneous Files - No Configurations (Global Scope)

```
#pragma once
#include <math.h>
#define max(a,b) ((a<b)?b:a)
#define min(a,b) ((a>b)?b:a)
void Quaternion2Euler(
    double* roll,
    double* pitch,
    double* yaw,
    const double q[4]);
```

Quaternion2Euler.cpp | Quaternion2Euler.h

Miscellaneous Files - No Configurations | (Global Scope)

```cpp

#include "Quaternion2Euler.h"

void Quaternion2Euler(
    double* roll,
    double* pitch,
    double* yaw,
    const double q[4]) {
    double e0 = q[0];
    double e1 = q[1];
    double e2 = q[2];
    double e3 = q[3];
    if (e0 < 0.0) {
        e0 = -e0;
        e1 = -e1;
        e2 = -e2;
        e3 = -e3;
    }
    *roll = atan2(2.0 * (e0 * e1 + e2 * e3), (e0 * e0 + e3 * e3 - e1 * e1 - e2 * e2));
    *pitch = asin(max(-1.0, min(1.0, 2.0 * (e0 * e2 - e1 * e3))));
    *yaw = atan2(2.0 * (e0 * e3 + e1 * e2), (e0 * e0 + e1 * e1 - e2 * e2 - e3 * e3));
}

```

function_CppK...onEstimator.h | function_CppK...Estimator.cpp

Miscellaneous Files - No Configurations | (Global Scope)

```
#pragma once

#include <math.h>
#include <cmath>

#define PI (3.14159265359f)
#define max(a,b) ((a<b)?b:a)
#define min(a,b) ((a>b)?b:a)


//Create a cross-product matrix
extern void CrossMatrix(
    double CrossMat[3][3], //OUTPUT: Cross product matrix
    const double vec3[3] //INPUT: vector to be converted to a cross product matrix
);


//Inertial to body rotation matrix
extern void Rotation_I2b(
    double Ri2b[3][3],  //OUTPUT: Rotation matrix from the inertial to body frame (Rows first, then columns)
    const double e0, //INPUT: scalar part of the quaternion orientation in the inertial frame
    const double e1, //INPUT: x-axis part of the quaternion orientation in the inertial frame
    const double e2, //INPUT: y-axis part of the quaternion orientation in the inertial frame
    const double e3  //INPUT: z-axis part of the quaternion orientation in the inertial frame
);

//Rotate a vec3 from the inertial frame to the body frame
extern void inertial_2_body_Rotation(
    double* x_b, //OUTPUT: x-position in the body frame
    double* y_b, //OUTPUT: y-position in the body frame
    double* z_b, //OUTPUT: z-position in the body frame
    const double xI, //INPUT: x-position in the inertial frame
    const double yI, //INPUT: y-position in the inertial frame
    const double zI, //INPUT: z-position in the inertial frame
    const double e0, //INPUT: scalar part of the quaternion orientation in the inertial frame
    const double e1, //INPUT: x-axis part of the quaternion orientation in the inertial frame
    const double e2, //INPUT: y-axis part of the quaternion orientation in the inertial frame
    const double e3  //INPUT: z-axis part of the quaternion orientation in the inertial frame
);
```

```c
//Normalize a 4d array
extern void vec4_norm(
    double C[4], //OUTPUT: normalized 4d array
    const double a[4] //INPUT: a 4d array to be normalized
);

//Normalize a 3d array
extern void vec3_norm(
    double C[3], //OUTPUT: normalized 3d array
    const double a[3] //INPUT: a 3d array to be normalized
);

//Matrix vector multiplication c = A*b
extern void Matrix3_Vec3_Multiplication(
    double c[3], //OUTPUT: output vector c = A*b
    const double A[3][3], //INPUT: input matrix c = A*b
    const double b[3] //INPUT: input vector c = A*b
);

//Multiply a vector by a scalar c = b * a
extern void Vec3_Scalar_Multiplication(
    double c[3], //OUTPUT: output vector c = b * a
    const double b[3], //INPUT: input vector c = b*a
    const double a //INPUT: scalar multiplier c = b*a
);

//Calculate the cross product between two 3d arrays
extern void vec3_cross(
    double C[3], //The cross product C = aXb
    const double a[3], //The 3D vector a in C=aXb
    const double b[3] //The 3D vector b in C=aXb
);

//Add two 3d vectors c=a+b
extern void vec3_addition(
    double c[3], //OUTPUT: output vector c = a + b
    const double a[3], //INPUT: input vector c = a + b
    const double b[3] //INPUT: input vector c = a + b
);

//subtract one 3d vector from another c=a-b
extern void vec3_subtraction(
    double c[3], //OUTPUT: output vector c = a - b
    const double a[3], //INPUT: input vector c = a - b
    const double b[3] //INPUT: input vector c = a - b
);

//Perform the quaternion derivative dq = 1/2*[S]*omega
extern void QuaternionTimeDerivative(
    double dq[4], //OUTPUT: the quaternion time derivative
    const double q[4], //INPUT: the quaternion
    const double omega[3] //INPUT: [rad/s] angular velocity
);

//Calculate the gradient of the acceleration and magnetometer cost function V
//see Manon Kok and Thomas B. Schon, ``A Fast and Robust Algorithm for Orientation Estimation
// using Inertial Sensors'', IEEE Signal Processing Letters, 2019. DOI: 10.1109 / LSP.2019.2943995
extern void KokSchonGradient(
    double gradient[3], //OUTPUT: the gradient of the magnetometer cost function
    const double NormalizedAccelerometer[3], //INPUT: [-] x-front, y-right, z-down normalized accelerometer signal
    const double NormalizedMagnetometer[3], //INPUT: [-] x-front, y-right, z-down normalized magnetometer signal
    const double quaternion[4], //INPUT: Normalized quaternion orientation of the body in the inertial frame
    const double inclinationAngleDegrees, //INPUT: [degrees] the geomagnetic inclination angle
    const double alpha //INPUT: [1,2] tuning parameter for magnetometer trust weighting
);

extern void KokSchonUpdate(
    double quaternion_new[4], //OUTPUT: updated quaternion orientation
    const double quaternion[4], //INPUT: previous quaternion orientation
    const double gradient[3], //INPUT: Kok Schon Gradient vector
    const double gyrometer[3], //INPUT: [rad/s] 3-axis gyro data with bias removed
    const double dt, //INPUT: [s] time-step
    const double beta //INPUT: [<< 1] small-valued tuning parameter
);

//The Kok Schon Quaternion Estimator Algorithm
```

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

function_CppK...ionEstimator.h | function_CppK...Estimator.cpp

Miscellaneous Files - No Configurations | (Global Scope)

```cpp
#include "function_CppKokSchonQuaternionEstimator.h"

//Create a cross-product matrix
extern void CrossMatrix(
    double CrossMat[3][3], //OUTPUT: Cross product matrix
    const double vec3[3] //INPUT: vector to be converted to a cross product matrix
) {

    CrossMat[0][0] = 0.0f;
    CrossMat[0][1] = -vec3[2];
    CrossMat[0][2] = vec3[1];
    CrossMat[1][0] = vec3[2];
    CrossMat[1][1] = 0.0f;
    CrossMat[1][2] = -vec3[0];
    CrossMat[2][0] = -vec3[1];
    CrossMat[2][1] = vec3[0];
    CrossMat[2][2] = 0.0f;
}


//Inertial to body rotation matrix
extern void Rotation_I2b(
    double Ri2b[3][3],  //OUTPUT: Rotation matrix from the inertial to body frame (Rows first, then columns)
    const double e0, //INPUT: scalar part of the quaternion orientation in the inertial frame
    const double e1, //INPUT: x-axis part of the quaternion orientation in the inertial frame
    const double e2, //INPUT: y-axis part of the quaternion orientation in the inertial frame
    const double e3  //INPUT: z-axis part of the quaternion orientation in the inertial frame
) {

    /*local variables*/
    double e0_2 = e0 * e0;
    double e1_2 = e1 * e1;
    double e2_2 = e2 * e2;
    double e3_2 = e3 * e3;
    double e0e1 = e0 * e1;
    double e0e2 = e0 * e2;
    double e0e3 = e0 * e3;
    double e1e2 = e1 * e2;
    double e1e3 = e1 * e3;
    double e2e3 = e2 * e3;

    Ri2b[0][0] = (e0_2 + e1_2 - e2_2 - e3_2);
    Ri2b[0][1] = 2.0f * (e0e3 + e1e2);
    Ri2b[0][2] = 2.0f * (e1e3 - e0e2);
    Ri2b[1][0] = 2.0f * (e1e2 - e0e3);
    Ri2b[1][1] = (e0_2 - e1_2 + e2_2 - e3_2);
    Ri2b[1][2] = 2.0f * (e0e1 + e2e3);
    Ri2b[2][0] = 2.0f * (e0e2 + e1e3);
    Ri2b[2][1] = 2.0f * (e2e3 - e0e1);
    Ri2b[2][2] = (e0_2 - e1_2 - e2_2 + e3_2);

}

//Rotate a vec3 from the inertial frame to the body frame
extern void inertial_2_body_Rotation(
    double* x_b, //OUTPUT: x-position in the body frame
    double* y_b, //OUTPUT: y-position in the body frame
    double* z_b, //OUTPUT: z-position in the body frame
    const double xI, //INPUT: x-position in the inertial frame
    const double yI, //INPUT: y-position in the inertial frame
    const double zI, //INPUT: z-position in the inertial frame
    const double e0, //INPUT: scalar part of the quaternion orientation in the inertial frame
    const double e1, //INPUT: x-axis part of the quaternion orientation in the inertial frame
    const double e2, //INPUT: y-axis part of the quaternion orientation in the inertial frame
    const double e3  //INPUT: z-axis part of the quaternion orientation in the inertial frame
) {

    /*local variables*/
    double e0_2 = e0 * e0;
    double e1_2 = e1 * e1;
    double e2_2 = e2 * e2;
    double e3_2 = e3 * e3;
    double e0e1 = e0 * e1;
    double e0e2 = e0 * e2;
```

```
    double e0e3 = e0 * e3;
    double e1e2 = e1 * e2;
    double e1e3 = e1 * e3;
    double e2e3 = e2 * e3;

    *x_b = (e0_2 + e1_2 - e2_2 - e3_2) * xI +
        2.0f * (e0e3 + e1e2) * yI +
        2.0f * (e1e3 - e0e2) * zI;
    *y_b = 2.0f * (e1e2 - e0e3) * xI +
        (e0_2 - e1_2 + e2_2 - e3_2) * yI +
        2.0f * (e0e1 + e2e3) * zI;
    *z_b = 2.0f * (e0e2 + e1e3) * xI +
        2.0f * (e2e3 - e0e1) * yI +
        (e0_2 - e1_2 - e2_2 + e3_2) * zI;
}


//Rotate a vec3 from the body frame to the inertial frame
extern void body_2_Inertial_Rotation(
    double* xI, //OUTPUT: x-position in the inertial frame
    double* yI, //OUTPUT: y-position in the inertial frame
    double* zI, //OUTPUT: z-position in the inertial frame
    const double xb, //INPUT: x-position in the body frame
    const double yb, //INPUT: y-position in the body frame
    const double zb, //INPUT: z-position in the body frame
    const double e0, //INPUT: scalar part of the quaternion orientation in the inertial frame
    const double e1, //INPUT: x-axis part of the quaternion orientation in the inertial frame
    const double e2, //INPUT: y-axis part of the quaternion orientation in the inertial frame
    const double e3  //INPUT: z-axis part of the quaternion orientation in the inertial frame
) {

    /*local variables*/
    double e0_2 = e0 * e0;
    double e1_2 = e1 * e1;
    double e2_2 = e2 * e2;
    double e3_2 = e3 * e3;
    double e0e1 = e0 * e1;
    double e0e2 = e0 * e2;
    double e0e3 = e0 * e3;
    double e1e2 = e1 * e2;
    double e1e3 = e1 * e3;
    double e2e3 = e2 * e3;

    *xI = (e0_2 + e1_2 - e2_2 - e3_2) * xb +
        2.0f * (-e0e3 + e1e2) * yb +
        2.0f * (e1e3 + e0e2) * zb;
    *yI = 2.0f * (e1e2 + e0e3) * xb +
        (e0_2 - e1_2 + e2_2 - e3_2) * yb +
        2.0f * (-e0e1 + e2e3) * zb;
    *zI = 2.0f * (-e0e2 + e1e3) * xb +
        2.0f * (e2e3 + e0e1) * yb +
        (e0_2 - e1_2 - e2_2 + e3_2) * zb;
}

//Normalize a 4d array
extern void vec4_norm(
    double C[4], //OUTPUT: normalized 4d array
    const double a[4] //INPUT: a 4d array to be normalized
)
{
    //Calculate the denominator
    double den = sqrt(
        a[0] * a[0] +
        a[1] * a[1] +
        a[2] * a[2] +
        a[3] * a[3]);

    //Avoid dividing by zero
    if (den > 0.000001f)
    {
        //Normalize the array
        C[0] = a[0] / den;
        C[1] = a[1] / den;
        C[2] = a
        C[3] = a
    }
    else
    {
```

(local variable) double e1_2
Search Online

```
        //The array is numerically zero and cannot be normalized
        C[0] = a[0];
        C[1] = a[1];
        C[2] = a[2];
        C[3] = a[3];
    }
}


//Normalize a 3d array
extern void vec3_norm(
    double C[3], //OUTPUT: normalized 3d array
    const double a[3] //INPUT: a 3d array to be normalized
)
{
    //Calculate the denominator
    double den = sqrt(
        a[0] * a[0] +
        a[1] * a[1] +
        a[2] * a[2]);

    //Avoid dividing by zero
    if (den > 0.000001)
    {
        //Normalize the array
        C[0] = a[0] / den;
        C[1] = a[1] / den;
        C[2] = a[2] / den;
    }
    else
    {
        //The array is numerically zero and cannot be normalized
        C[0] = a[0];
        C[1] = a[1];
        C[2] = a[2];
    }
}

double norm3(double x[3]) {
    return sqrt(x[0] * x[0] + x[1] * x[1] + x[2] * x[2]);
}

//Matrix vector multiplication c = A*b
extern void Matrix3_Vec3_Multiplication(
    double c[3], //OUTPUT: output vector c = A*b
    const double A[3][3], //INPUT: input matrix c = A*b
    const double b[3] //INPUT: input vector c = A*b
) {
    for (int ii = 0; ii < 3; ii++) {
        c[ii] = 0.0f;
        for (int jj = 0; jj < 3; jj++) {
            c[ii] += A[ii][jj] * b[jj];
        }
    }
}

//Multiply a vector by a scalar c = b * a
extern void Vec3_Scalar_Multiplication(
    double c[3], //OUTPUT: output vector c = b * a
    const double b[3], //INPUT: input vector c = b*a
    const double a //INPUT: scalar multiplier c = b*a
) {
    for (int ii = 0; ii < 3; ii++) {
        c[ii] = b[ii] * a;
    }
}

//Calculate the cross product between two 3d arrays
extern void vec3_cross(
    double C[3], //The cross product C = aXb
    const double a[3], //The 3D vector a in C=aXb
    const double b[3] //The 3D vector b in C=aXb
)
{
    //Calculate the cross product C = aXb
    C[0] = a[1] * b[2] - b[1] * a[2];
    C[1] = a[2] * b[0] - b[2] * a[0];
    C[2] = a[0] * b[1] - b[0] * a[1];
```

```c
}

//Add two 3d vectors c=a+b
extern void vec3_addition(
    double c[3], //OUTPUT: output vector c = a + b
    const double a[3], //INPUT: input vector c = a + b
    const double b[3] //INPUT: input vector c = a + b
) {
    for (int ii = 0; ii < 3; ii++) {
        c[ii] = a[ii] + b[ii];
    }
}

//subtract one 3d vector from another c=a-b
extern void vec3_subtraction(
    double c[3], //OUTPUT: output vector c = a - b
    const double a[3], //INPUT: input vector c = a - b
    const double b[3] //INPUT: input vector c = a - b
) {
    for (int ii = 0; ii < 3; ii++) {
        c[ii] = a[ii] - b[ii];
    }
}

//Perform the quaternion derivative dq = 1/2*[S]*omega
extern void QuaternionTimeDerivative(
    double dq[4], //OUTPUT: the quaternion time derivative
    const double q[4], //INPUT: the quaternion
    const double omega[3] //INPUT: [rad/s] angular velocity
) {
    dq[0] = -q[1] * omega[0] - q[2] * omega[1] - q[3] * omega[2];
    dq[1] = q[0] * omega[0] - q[3] * omega[1] + q[2] * omega[2];
    dq[2] = q[3] * omega[0] + q[0] * omega[1] - q[1] * omega[2];
    dq[3] = -q[2] * omega[0] + q[1] * omega[1] + q[0] * omega[2];
}

//Calculate the gradient of the acceleration and magnetometer cost function V
//see Manon Kok and Thomas B. Schon, ``A Fast and Robust Algorithm for Orientation Estimation
// using Inertial Sensors'', IEEE Signal Processing Letters, 2019. DOI: 10.1109 / LSP.2019.2943995
extern void KokSchonGradient(
    double gradient[3], //OUTPUT: the gradient of the magnetometer cost function
    const double NormalizedAccelerometer[3], //INPUT: [-] x-front, y-right, z-down normalized accelerometer signal
    const double NormalizedMagnetometer[3], //INPUT: [-] x-front, y-right, z-down normalized magnetometer signal
    const double quaternion[4], //INPUT: Normalized quaternion orientation of the body in the inertial frame
    const double inclinationAngleDegrees, //INPUT: [degrees] the geomagnetic inclination angle
    const double alpha //INPUT: [1,2] tuning parameter for magnetometer trust weighting
) {
    double Ri2b[3][3];
    double gravityInBodyFrame[3];
    double MagInBodyFrame[3];
    double inclinationAngleRadians = inclinationAngleDegrees * PI / 180.0f;
    double cos_incAngle = cos(inclinationAngleRadians);
    double sin_incAngle = sin(inclinationAngleRadians);
    double accelMinusGravity[3];
    double MagMinusMag[3];
    double GravityCrossGravityError[3];
    double MagCrossMagError[3];

    //Get the rotation matrix from the inertial frame to the body frame
    Rotation_I2b(
        Ri2b,  //OUTPUT: Rotation matrix from the inertial to body frame (Rows first, then columns)
        quaternion[0], //INPUT: scalar part of the quaternion orientation in the inertial frame
        quaternion[1], //INPUT: x-axis part of the quaternion orientation in the inertial frame
        quaternion[2], //INPUT: y-axis part of the quaternion orientation in the inertial frame
        quaternion[3]  //INPUT: z-axis part of the quaternion orientation in the inertial frame
    );

    //Get the gravity unit vector in the body frame
    for (int ii = 0; ii < 3; ii++)
        gravityInBodyFrame[ii] = Ri2b[ii][2];

    //Get the geomagnetic unit vector in the body frame
    for (int ii = 0; ii < 3; ii++)
        MagInBodyFrame[ii] = Ri2b[ii][0] * cos_incAngle + Ri2b[ii][2] * sin_incAngle;

    //subtract gravity from the accelerometer signal (i.e. gravity error)
    for (int ii = 0; ii < 3; ii++)
        accelMinusGravity[ii] = NormalizedAccelerometer[ii] - gravityInBodyFrame[ii];
```

```c

    //get the magnetometer error
    for (int ii = 0; ii < 3; ii++)
        MagMinusMag[ii] = NormalizedMagnetometer[ii] - MagInBodyFrame[ii];

    //Get the cross-product between the gravity and the gravity error
    vec3_cross(GravityCrossGravityError, gravityInBodyFrame, accelMinusGravity);

    //Get the cross-product between the body geomagnetic vector and its error
    vec3_cross(MagCrossMagError, MagInBodyFrame, MagMinusMag);

    //The Kok Schon gradient is the sum of the cross products
    for (int ii = 0; ii < 3; ii++)
        gradient[ii] = GravityCrossGravityError[ii] + alpha * MagCrossMagError[ii];
}

extern void KokSchonUpdate(
    double quaternion_new[4], //OUTPUT: updated quaternion orientation
    const double quaternion[4], //INPUT: previous quaternion orientation
    const double gradient[3], //INPUT: Kok Schon Gradient vector
    const double gyrometer[3], //INPUT: [rad/s] 3-axis gyro data with bias removed
    const double dt, //INPUT: [s] time-step
    const double beta //INPUT: [<< 1] small-valued tuning parameter
) {
    double q0 = quaternion[0];
    double q1 = quaternion[1];
    double q2 = quaternion[2];
    double q3 = quaternion[3];

    //get the magnitude of the gradient
    double normGrad = sqrt(max(0.0f, gradient[0] * gradient[0] +
        gradient[1] * gradient[1] +
        gradient[2] * gradient[2]));


    double omega[3]; //Corrected angular velocity (rad/s)
    //Avoid the possibility of dividing by zero
    if (normGrad > 0.0001f) {
        //Normal case
        //Get the corrected angular velocity (rad/s)
        for (int ii = 0; ii < 3; ii++)
            omega[ii] = gyrometer[ii] - beta * gradient[ii] / normGrad;
    }
    else {
        //should never get here
        //Set the corrected angular velocity to the measured value (rad/s)
        for (int ii = 0; ii < 3; ii++)
            omega[ii] = gyrometer[ii];
    }

    //Update the quaternion estimate
    double qu[4]; //updated (but not normalized) quaternion
    qu[0] = q0 + 0.5f * dt * (-q1 * omega[0] - q2 * omega[1] - q3 * omega[2]);
    qu[1] = q1 + 0.5f * dt * (q0 * omega[0] - q3 * omega[1] + q2 * omega[2]);
    qu[2] = q2 + 0.5f * dt * (q3 * omega[0] + q0 * omega[1] - q1 * omega[2]);
    qu[3] = q3 + 0.5f * dt * (-q2 * omega[0] + q1 * omega[1] + q0 * omega[2]);

    //Get the magnitude of the quaternion
    double norm_qu = sqrt(max(0.0f, qu[0] * qu[0] +
        qu[1] * qu[1] +
        qu[2] * qu[2] +
        qu[3] * qu[3]));

    //Avoid division by zero
    if (norm_qu > 0.001)
        //Update the quaternion
        for (int ii = 0; ii < 4; ii++)
            quaternion_new[ii] = qu[ii] / norm_qu;
    else {
        //should never get to this case, reset the quaternion
        quaternion_new[0] = 1.0f;
        quaternion_new[1] = 0.0f;
        quaternion_new[2] = 0.0f;
        quaternion_new[3] = 0.0f;
    }
}

//The Kok Schon Quaternion Estimator Algorithm
```

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

    //extract the quaternion components
    double e0 = quaternion[0];
    double e1 = quaternion[1];
    double e2 = quaternion[2];
    double e3 = quaternion[3];

    //Ensure that the quaternion is a unit quaternion
    double norm_q = sqrt(e0 * e0 + e1 * e1 + e2 * e2 + e3 * e3);
    if (fabs(norm_q - 1.0f) > 0.00001f) {
        //The quaternion is not a valid unit quaternion, normalize it
        if (norm_q > 0.0001f) {
            e0 = e0 / norm_q;
            e1 = e1 / norm_q;
            e2 = e2 / norm_q;
            e3 = e3 / norm_q;
        }
        else
        {
            //Avoid dividing by zero
            e0 = 1.0f;
            e1 = 0.0f;
            e2 = 0.0f;
            e3 = 0.0f;
        }
    }


    //Get the inertial-frame to body-frame rotation matrix
    double RI2b[3][3];
    Rotation_I2b(RI2b, e0, e1, e2, e3);

    //get a gravity unit vector [unitless]
    double unitGravity[3];
    vec3_norm(unitGravity, gravity);

    //Normalize the magnetometer data
    double unitMag[3];
    vec3_norm(unitMag, magnetometer); //[unitless]

    //Get the Kok Schon gradient
    double gradient[3];
    //get the Kok Schon Gradient
    KokSchonGradient(
        gradient, //OUTPUT: the gradient of the magnetometer cost function
        unitGravity, //INPUT: [-] x-front, y-right, z-down normalized accelerometer signal
        unitMag, //INPUT: [-] x-front, y-right, z-down normalized magnetometer signal
        quaternion, //INPUT: Normalized quaternion orientation of the body in the inertial frame
        inclinationAngleDegrees, //INPUT: [degrees] the geomagnetic inclination angle
        alpha //INPUT: [1,2] tuning parameter for magnetometer trust weighting
    );

    //Get the Kok Schon estimate of the quaternion
    KokSchonUpdate(
        quaternion_new, //OUTPUT: updated quaternion orientation
        quaternion, //INPUT: previous quaternion orientation
        gradient, //INPUT: Kok Schon Gradient vector
        gyrometer, //INPUT: [rad/s] 3-axis gyro data with bias removed
        dt, //INPUT: [s] time-step
        beta //INPUT: [<< 1] small-valued tuning parameter
    );


    //If any output is infinite or NaN, reset everything
    if (isinf(quaternion_new[0]) || isnan(quaternion_new[0])
        || isinf(quaternion_new[1]) || isnan(quaternion_new[1])
        || isinf(quaternion_new[2]) || isnan(quaternion_new[2])
```

```
        || isinf(quaternion_new[3]) || isnan(quaternion_new[3])
        )
    {
        //Some proplem happened, reset everything
        //Reset quaternions
        quaternion_new[0] = 1.0f;
        for (unsigned int ii = 1; ii < 4; ii++)
            quaternion_new[ii] = 0.0f;

        return 1;  //indicate that something failed
    }
    else {
        return 0; //normal case
    }
}
```

function_Auto...ointTracker.h function_Auton...intTracker.cpp

Miscellaneous Files - No Configurations (Global Scope)

```
#pragma once

#include <math.h>
#define pi (3.14159265359f)
#define max(a,b) ((a<b)?b:a)
#define min(a,b) ((a>b)?b:a)
#define sign(a) ((a<0)?-1.0:1.0)



void function_Autonomous_GPS_WaypointTracker(
    double* delta_t, //OUTPUT: [0,1] throttle command
    double* delta_e, //OUTPUT: [-1,1] elevator command
    double* delta_a, //OUTPUT: [-1,1] aileron command
    double* delta_r, //OUTPUT: [-1,1] rudder command
    const double q[4], //INPUT: [e0;e1;e2;e3] (-1 to 1) quaternion orientation
    const double gyro[3], //INPUT: [p;q;r] (rad/s) gyrometer signals
    const double lat_target, //INPUT: [lat](deg) target latitude angle
    const double lon_target, //INPUT: [lon](deg) target longitude angle
    const double alt_target, //INPUT: [-zI](m) target altitude
    const double latitude, //INPUT: [lat](deg) current latitude angle
    const double longitude, //INPUT: [lon](deg) current longitude angle
    const double altitude,  //INPUT: [-zI](m) current altitude
    const double kp_roll,  //INPUT: [PID gain] proportional gain for roll control
    const double kd_roll,  //INPUT: [PID gain] derivative gain for roll control
    const double kp_pitch,  //INPUT: [PID gain] proportional gain for pitch control
    const double kp_yaw,  //INPUT: [PID gain] proportional gain for yaw control
    const double kp_rudder,  //INPUT: [PID gain] proportional gain for yaw control
    const double max_roll,  //INPUT: [limit](rad) max roll angle limit
    const double max_pitch  //INPUT: [limit](rad) max pitch angle limit
);
```

function_Auto...ntTracker.cpp | function_Auto...pointTracker.h

Miscellaneous Files - No Configurations | (Global Scope) | ConvertGPS(double dXI[3], const double LatTarget, c

```cpp
#include "function_Autonomous_GPS_WaypointTracker.h"

static double cosd(double angleDegrees) {
    return cos(pi / 180.0 * angleDegrees);
}

static double norm3(double x[3]) {
    return sqrt(x[0] * x[0] + x[1] * x[1] + x[2] * x[2]);
}

static void ConvertGPS(
    double dXI[3],
    const double LatTarget,
    const double LonTarget,
    const double AltTarget,
    const double GPS_lat,
    const double GPS_lon,
    const double alt
)
{
    const double rEarth = 6371000.0; //(m) earth's radius
    dXI[0] = (rEarth * pi / 180.0) * (LatTarget - GPS_lat); //xI displacement
    double dLon = LonTarget - GPS_lon;
    if (fabs(dLon) > 180.0) {
        if (LonTarget > GPS_lon)
            dLon = LonTarget - (GPS_lon + 360.0);
        else
            dLon = (360.0 + LonTarget) - GPS_lon;
    }
    //yI displacement
    dXI[1] = sign(dLon) * rEarth * acos((cosd(dLon) - 1.0) * (cosd(GPS_lat) * cosd(GPS_lat)) + 1.0);
    //zI displacement
    dXI[2] = alt - AltTarget;
}

static void processWaypoints(
    double* pitchDes,
    double* yawDes,
    const double target_lat,
    const double target_lon,
    const double target_alt,
    const double GPS_lat,
    const double GPS_lon,
    const double alt
)
{
    double dXI[3] = { 0.0 };
    //Get the distance to the next waypoint
    ConvertGPS(dXI,
        target_lat,
        target_lon,
        target_alt,
        GPS_lat,
        GPS_lon,
        alt
    );

    //find the absolute distance
    double distance = norm3(dXI);

    //get the desired pitch angle (rad)
    *pitchDes = atan(-dXI[2] / max(1.0, distance));
    //get the desired yaw angle (rad)
    *yawDes = atan2(dXI[1], dXI[0]);
}


static void Quaternion2Euler(
    double* roll,
    double* pitch,
    double* yaw,
    const double q[4]) {
    double e0 = q[0];
```

```
    double e1 = q[1];
    double e2 = q[2];
    double e3 = q[3];
    *roll = atan2(2.0 * (e0 * e1 + e2 * e3), (e0 * e0 + e3 * e3 - e1 * e1 - e2 * e2));
    *pitch = asin(max(-1.0, min(1.0, 2.0 * (e0 * e2 - e1 * e3))));
    *yaw = atan2(2.0 * (e0 * e3 + e1 * e2), (e0 * e0 + e1 * e1 - e2 * e2 - e3 * e3));
}

//Get the shortest angle difference
static double getAngleError(
    double xd, //(rad) desired angle
    double x //(rad) actual angle
) {
    //get the difference between the angles
    double diff = xd - x; //(rad)

    //Get the shortest absolute angle between the two angles
    double d = acos(cos(xd) * cos(x) + sin(xd) * sin(x));

    double dx; //shortest angle
    //Check whether they are different
    if (fabs(diff) - d > 0.1) {
        //they are different, use the shortest absolute angle
        dx = -sign(diff) * d;
    }
    else {
        //they are the same
        dx = diff;
    }
    return dx;
}

static void getCommands(
    double* delta_t,
    double* delta_e,
    double* delta_a,
    double* delta_r,
    const double roll,
    const double pitch,
    const double yaw,
    const double pitchDes,
    const double yawDes,
    const double gyro[3],
    const double kp_roll,  //INPUT: [PID gain] proportional gain for roll control
    const double kd_roll,  //INPUT: [PID gain] derivative gain for roll control
    const double kp_pitch,  //INPUT: [PID gain] proportional gain for pitch control
    const double kp_yaw,  //INPUT: [PID gain] proportional gain for yaw control
    const double kp_rudder,  //INPUT: [PID gain] proportional gain for yaw control
    const double max_roll,  //INPUT: [limit](rad) max roll angle limit
    const double max_pitch  //INPUT: [limit](rad) max pitch angle limit
)
{
    double yaw_err = getAngleError(yawDes, yaw);
    double roll_des = kp_yaw * yaw_err;
    roll_des = max(-max_roll, min(roll_des, max_roll));

    double roll_err = getAngleError(roll_des, roll);

    double pitch_set = max(-max_pitch, min(pitchDes, max_pitch));
    double pitch_err = getAngleError(pitch_set, pitch);

    *delta_a = kp_roll * roll_err - kd_roll * gyro[0];

    *delta_r = max(-1.0, min(kp_rudder * yaw_err, 1.0));
    *delta_e = kp_pitch * pitch_err;
    *delta_t = 0.8;

    *delta_a = max(-1.0, min(*delta_a, 1.0));
    *delta_e = max(-1.0, min(*delta_e, 1.0));
}


void function_Autonomous_GPS_WaypointTracker(
    double* delta_t, //OUTPUT: [0,1] throttle command
    double* delta_e, //OUTPUT: [-1,1] elevator command
    double* delta_a, //OUTPUT: [-1,1] aileron command
    double* delta_r, //OUTPUT: [-1,1] rudder command
    const double q[4], //INPUT: [e0;e1;e2;e3] (-1 to 1) quaternion orientation
```

```
    const double gyro[3], //INPUT: [p;q;r] (rad/s) gyrometer signals
    const double lat_target, //INPUT: [lat](deg) target latitude angle
    const double lon_target, //INPUT: [lon](deg) target longitude angle
    const double alt_target, //INPUT: [-zI](m) target altitude
    const double latitude, //INPUT: [lat](deg) current latitude angle
    const double longitude, //INPUT: [lon](deg) current longitude angle
    const double altitude,  //INPUT: [-zI](m) current altitude
    const double kp_roll,  //INPUT: [PID gain] proportional gain for roll control
    const double kd_roll,  //INPUT: [PID gain] derivative gain for roll control
    const double kp_pitch,  //INPUT: [PID gain] proportional gain for pitch control
    const double kp_yaw,  //INPUT: [PID gain] proportional gain for yaw control
    const double kp_rudder,  //INPUT: [PID gain] proportional gain for yaw control
    const double max_roll,  //INPUT: [limit](rad) max roll angle limit
    const double max_pitch  //INPUT: [limit](rad) max pitch angle limit
) {
    //get the estimated roll pitch and yaw angles
    double roll, pitch, yaw;
    Quaternion2Euler(&roll, &pitch, &yaw, q);

    //Process the waypoints
    double pitchDes, yawDes;
    processWaypoints(&pitchDes, &yawDes,
        lat_target, lon_target, alt_target,
        latitude, longitude, altitude);

    //Get the control commands
    getCommands(delta_t, delta_e, delta_a, delta_r,
        roll, pitch, yaw, pitchDes, yawDes, gyro,
        kp_roll, kd_roll, kp_pitch, kp_yaw, kp_rudder, max_roll, max_pitch);
}
```

AutonomousGPSFlight.ino

```
#include <SPI.h>
#include <SD.h>
#include <MPU9250_WE.h>                              //Include the MPU9250 library for Accel, Gyro, and Magnetom
#include <BMP280_DEV.h>                              //Include the BMP280_DEV.h library for altitude, pressure,
#include <function_CppKokSchonQuaternionEstimator.h>  //Include the Kok Schon Quaternion Estimator
#include <Quaternion2Euler.h>                        //Include the library to convert quaternion to euler orient
#include <SoftwareSerial.h>                          //Include the SofwareSerial library for GPS communication
#include <TinyGPS.h>                                 //Include the GPS library
#include <Servo.h>                                   //Include the Servo library
#include <function_Autonomous_GPS_WaypointTracker.h>  //Include the GPS waypoint tracking library

//name the file for the SD card
String filename = "FlightData001.csv";
//Say whether to collect GPS data or not
bool useGPS = true;

//####################Variables for GPS#########################//
TinyGPS gps;
const int rxPin = 1;
const int txPin = 0;
SoftwareSerial ss(rxPin, txPin);
static void readGPS(unsigned long ms);
float GPSlat = 0.0;  //declare GPS latitude and longitude variables
float GPSlon = 0.0;
float GPSspeed = 0.0;  //declare GPS speed (m/s)
double target_lat = 83.820934;
double target_lon = -111.785286;
double target_alt = 854.0;
bool GPSisValid = false;
//##################End ofVariables for GPS######################//

//####################Variables for SD Card Reader###############//
#if !defined(ARDUINO_ARCH_RP2040)
#error For RP2040 only
#endif
#if defined(ARDUINO_ARCH_MBED)
#define PIN_SD_MOSI PIN_SPI_MOSI
#define PIN_SD_MISO PIN_SPI_MISO
#define PIN_SD_SCK PIN_SPI_SCK
#define PIN_SD_SS PIN_SPI_SS
#else
#define PIN_SD_MOSI PIN_SPI0_MOSI
#define PIN_SD_MISO PIN_SPI0_MISO
#define PIN_SD_SCK PIN_SPI0_SCK
#define PIN_SD_SS PIN_SPI0_SS
#endif
#define _RP2040_SD_LOGLEVEL_ 0
//###############End of Variables for SD Card Reader#############//

//#############Variables for 10 DOF MPU9250 Sensor##############//
#define MPU9250_ADDR 0x68  //define the address for the MPU 9250
//Create the MPU9250 object and name it myMPU9250
MPU9250_WE myMPU9250 = MPU9250_WE(MPU9250_ADDR);
//Create the BMP280_DEV object, name it bmp280. I2C address is 0x77
BMP280_DEV bmp280;
//declare temperature, pressure, and altitude
float temperature, pressure, altitude;
//##########End of Variables for 10 DOF MPU9250 Sensor###########//


//###################Variables for the Servo Motors##############//
Servo AileronServo;
Servo ElevatorServo;
```

```
Servo RudderServo;
const int AileronPin = 12;
const int ElevatorPin = 11;
const int RudderPin = 10;
const double kp_roll = 2.0;
const double kd_roll = 0.0;
const double kp_pitch = -2.0;
const double kp_yaw = 0.5;
const double kp_rudder = 0.2;
const double max_roll = 3.14 / 4.0;
const double max_pitch = 3.14 / 4.0;
//#################End of Variables for the Servo Motors##########//

//Additional variables
bool USBconnected = true;
double quaternion[4] = { 1.0, 0.0, 0.0, 0.0 };
double t_last = 0.0;

void setup() {

  // Start serial communication
  Serial.begin(9600);
  delay(3000);
  if (!Serial) {
    USBconnected = false;
  }
  if (useGPS) {
    //Start communication with the GPS module
    ss.begin(9600);
    unsigned long GPSdelay = 1000;  //Time delay in ms
    readGPS(GPSdelay);
  }
  Wire.begin();  //Begin I2C
  delay(2000);
  if (!myMPU9250.init()) {  //Start the MPU, if it fails, report an error
    if (USBconnected) {
      Serial.println("MPU9250 does not respond");
    }
  }
  if (!myMPU9250.initMagnetometer()) {  //Start the magnetometer, if it fails, report
    if (USBconnected) {
      Serial.println("Magnetometer does not respond");
    }
  }
  bmp280.begin();  // Default initialization, place the BMP280 into SLEEP_MODE
  //bmp280.setPresOversampling(OVERSAMPLING_X4); // Set the pressure oversampling to X4
  //bmp280.setTempOversampling(OVERSAMPLING_X1); // Set the temperature oversampling to X1
  //bmp280.setIIRFilter(IIR_FILTER_4); // Set the IIR filter to setting 4
  bmp280.setTimeStandby(TIME_STANDBY_2000MS);  // Set the standby time to 2 seconds
  bmp280.startNormalConversion();              // Start BMP280 continuous conversion in NORMAL_MODE

  if (USBconnected) {
    Serial.println("Position your MPU9250 flat and don't move it - calibrating...");
  }
  delay(1000);
  myMPU9250.autoOffsets();  //Callibrate the accelerometer and gyro offsets
  if (USBconnected) {
    Serial.println("Done!");
  }

#if defined(ARDUINO_ARCH_MBED)
  if (USBconnected) {
    Serial.print("Starting SD Card ReadWrite on MBED ");
  }
```

```
#else
  if (USBconnected) {
    Serial.print("Starting SD Card ReadWrite on ");
  }
#endif

  if (USBconnected) {
    Serial.println(BOARD_NAME);
    Serial.print("Initializing SD card with SS = ");
    Serial.println(PIN_SD_SS);
    Serial.print("SCK = ");
    Serial.println(PIN_SD_SCK);
    Serial.print("MOSI = ");
    Serial.println(PIN_SD_MOSI);
    Serial.print("MISO = ");
    Serial.println(PIN_SD_MISO);
  }

  if (!SD.begin(PIN_SD_SS)) {
    if (USBconnected) {
      Serial.println("Initialization failed!");
    }
  } else {
    for (int ii = 0; ii < 10; ii++) {
      if (SD.exists(filename)) {
        filename = "FlightData";
        filename += String(ii);
        filename += ".csv";
      }
    }
  }
  if (USBconnected) {
    Serial.println("Initialization done.");
  }

  //create the SD card file and open it for writing
  File dataFile = SD.open(filename, FILE_WRITE);

  //If it opened correctly, write the header to it
  if (dataFile) {
    String headerString = "time,";
    headerString += "gFx,gFy,gFz,wx,wy,wz,p,Bx,By,Bz,Azimuth,Pitch,Roll,Latitude,
      Longitude,Speed (m/s),delta_e,delta_a,delta_r";
    dataFile.println(headerString);
    dataFile.close();
  } else {
    if (USBconnected) {
      Serial.println("Initialization failed to open the file");
    }
  }

  //###############Set up the Servos########################//
  AileronServo.attach(AileronPin);
  ElevatorServo.attach(ElevatorPin);
  RudderServo.attach(RudderPin);

  t_last = 0.0;
}

void loop() {
  //Get the values from the accelerometer
  xyzFloat accel = myMPU9250.getGValues();
  //Get the values from the Gyrometer
```

```
  xyzFloat gyro = myMPU9250.getGyrValues();
  //Get the values from the Magnetometer
  xyzFloat Mag = myMPU9250.getMagValues();

  //put the data into arrays
  double gyrometer[3];
  double gravity[3];
  double magnetometer[3];
  gyrometer[0] = gyro.x * 3.14159 / 180.0;
  gyrometer[1] = -gyro.y * 3.14159 / 180.0;
  gyrometer[2] = -gyro.z * 3.14159 / 180.0;

  gravity[0] = -accel.x;
  gravity[1] = accel.y;
  gravity[2] = accel.z;

  magnetometer[0] = (Mag.y - 40.3058);
  magnetometer[1] = -(Mag.x + 11.444);
  magnetometer[2] = (Mag.z + 33.2877);

  //get the time
  double t = (double)millis() / 1000.0;
  double dt = t - t_last;
  t_last = t;

  //Get the quaternion orientation
  function_KokSchonQuaternionEstimator(
    quaternion,
    quaternion,
    gyrometer,
    gravity,
    magnetometer,
    dt,
    67.0,
    1.0,
    0.3);

  double roll, pitch, yaw;
  Quaternion2Euler(&roll, &pitch, &yaw, quaternion);

  //Get the bmp280 measurements
  //Temperature in Celsius, pressure in hectopascals, altitude in meters
  bmp280.getMeasurements(temperature, pressure, altitude);

  //Get GPS measurements
  if (useGPS) {
    unsigned long GPSdelay = 1;  //Time delay in ms
    readGPS(GPSdelay);
    unsigned long age;
    //Read the latitude and longitude
    gps.f_get_position(&GPSlat, &GPSlon, &age);

    //check that the GPS signal is valid
    if (((GPSlat - 43.0) < 2.0) && ((GPSlon + 111.0) < 2) && useGPS) {
      GPSisValid = true;
    } else {
      GPSisValid = false;
    }
  }

  //get the control commands
  double delta_t = 0.0;
  double delta_e = 0.0;
  double delta_a = 0.0;
```

```
double delta_r = 0.0;
if (GPSisValid) {
  function_Autonomous_GPS_WaypointTracker(
    &delta_t, &delta_e, &delta_a, &delta_r,
    quaternion, gyrometer, target_lat, target_lon, target_alt + 2.0,
    GPSlat, GPSlon, target_alt,
    kp_roll, kd_roll, kp_pitch, kp_yaw, kp_rudder,
    max_roll, max_pitch);
} else {
  delta_e = -kp_pitch * pitch;
  delta_a = -kp_roll * roll;
  delta_r = 0.0;
}


//set the servo positions
const double delta_e_trim = -0.13;
ElevatorServo.write(mapServoCmd(delta_e - delta_e_trim));
AileronServo.write(mapServoCmd(delta_a));
RudderServo.write(mapServoCmd(delta_r));

String dataString = "";
dataString += t;  //time
dataString += ",";
dataString += gravity[0];  //gFx
dataString += ",";
dataString += gravity[1];  //gFy
dataString += ",";
dataString += gravity[2];  //gFz
dataString += ",";
dataString += gyrometer[0];  //wx
dataString += ",";
dataString += gyrometer[1];  //wy
dataString += ",";
dataString += gyrometer[2];  //wz
dataString += ",";
dataString += pressure;  //p
dataString += ",";
dataString += magnetometer[0];  //Bx
dataString += ",";
dataString += magnetometer[1];  //By
dataString += ",";
dataString += magnetometer[2];  //Bz
dataString += ",";
dataString += yaw;  //Azimuth
dataString += ",";
dataString += pitch;  //Pitch
dataString += ",";
dataString += roll;  //Roll
dataString += ",";
if (useGPS) {
  char charArray[12];
  dtostrf(GPSlat, 1, 6, charArray);  //Latitude
  dataString += charArray;
  dataString += ",";
  dtostrf(GPSlon, 1, 6, charArray);  //Longitude
  dataString += charArray;
  dataString += ",";
  dataString += GPSspeed;  //Speed (m/s)
} else {
  dataString += "0";  //Latitude
  dataString += ",";
  dataString += "0";  //Longitude
  dataString += ",";
```

```
    dataString += "0";  //Speed (m/s)
  }
  dataString += ",";
  dataString += delta_e;
  dataString += ",";
  dataString += delta_a;
  dataString += ",";
  dataString += delta_r;

  writeToSDcard(dataString);
}

void writeToSDcard(String dataString) {
  //create the SD card file and open it for writing
  File dataFile = SD.open(filename, FILE_WRITE);

  //If it opened correctly, write the header to it
  if (dataFile) {
    dataFile.println(dataString);
    dataFile.close();
  } else {
    if (USBconnected) {
      Serial.println("Failed to open the file");
    }
  }
}


static void readGPS(unsigned long ms) {
  unsigned long start = millis();
  do {
    while (ss.available())
      gps.encode(ss.read());
  } while (millis() - start < ms);
}

static int mapServoCmd(double delta) {
  double slope = (135.0 - 90.0) * delta;
  return (((int)slope) + 90.0);
}
```