Got angle and angular velocity (Kalman filter algorithm)

Got angle and angular velocity (Kalman filter algorithm)

- Kalman filtering algorithm
- Implementation ideas
- Implementation code
- Software code

The tutorial introduces the use of Kalman algorithm to calculate the attitude angle.

```
The MPU6050 module can detect the three-axis acceleration, the motion data of the
three-axis gyroscope, and the temperature data.
```

Kalman filtering algorithm

Kalman filtering is an algorithm that uses the linear system state equation to optimally estimate the system state through the system input and output observation data.

Since the observation data includes the influence of noise and interference in the system, the optimal estimate can also be regarded as a filtering process.

Implementation ideas

Use the acceleration of the X, Y, and Z axes collected by the three-axis accelerometer in the MPU6050 and the angular velocity of the X, Y, and Z axes collected by the three-axis gyroscope, perform Kalman filtering, and finally obtain the filtered angle data.

① Input data: Get the three-axis acceleration and three-axis angular velocity data collected by the MPU6050 sensor;

② Prior estimation calculation: Predict the state at the current moment based on the state estimation and system model at the previous moment;

③ Predict covariance matrix: Calculate the predicted state covariance matrix based on the system model and prior estimation;

④ Calculate Kalman gain: Calculate the Kalman gain based on the predicted state covariance matrix, the observation noise covariance matrix and the observation model;

⑤ Observation update: Use the data measured by the MPU6050 sensor to update the prior estimation and obtain the posterior estimation;

⑥ Update covariance matrix: Calculate the updated state covariance matrix based on the Kalman gain and the prior error covariance matrix;

⑦ Return the filtered data.

Implementation code

```
float KF_X(float acce_Y, float acce_Z, float gyro_X) // Input: Y-axis
acceleration, Z-axis acceleration, X-axis angular velocity.
```

```
{
static float x_hat[2][1] = {0}; // posterior estimate
static float x_hat_minus[2][1] = {0}; // a priori estimate
static float p_hat[2][2] = {{1, 0}, {0, 1}}; // posterior error covariance matrix
static float p_hat_minus[2][2] = {0}; // a priori error covariance matrix
static float K[2][1] = {0}; // Kalman gain
const float Ts = 0.005; // sampling interval (5ms)
const float I[2][2] = {{1, 0}, {0, 1}};
float u[1][1] = {{gyro_X}};
float A[2][2] = {{1, -Ts}, {0, 1}}; // A matrix
float B[2][1] = {{Ts}, {0}}; // B matrix
float C[1][2] = {{1, 0}}; // C matrix
float Q[2][2] = {{1e-10, 0}, {0, 1e-10}}; // process noise
float R[1][1] = {{1e-4}}; // measurement noise
float A_T[2][2] = {{1, 0}, {-Ts, 1}}; // transpose of A matrix
float C_T[2][1] = {{1}, {0}}; // transpose of C matrix
float temp_1[2][1] = {0}; // used to store intermediate calculation results
float temp_2[2][1] = {0}; // used to store intermediate calculation results
float temp_3[2][2] = {0}; // used to store intermediate calculation results
float temp_4[2][2] = {0}; // used to store intermediate calculation results
float temp_5[1][2] = {0}; // Used to store intermediate calculation results
float temp_6[1][1] = {0}; // Used to store intermediate calculation results
float y = atan2(-acce_Y, acce_Z); // Calculate angle using acceleration
// Prediction part
// Prior estimation formula
mul(2, 2, 2, 1, A, x_hat, temp_1);
mul(2, 1, 1, 1, B, u, temp_2);
x_hat_minus[0][0] = temp_1[0][0] + temp_2[0][0];
x_hat_minus[1][0] = temp_1[1][0] + temp_2[1][0];
// Prior error covariance formula
mul(2, 2, 2, 2, A, p_hat, temp_3);
mul(2, 2, 2, 2, temp_3, A_T, temp_4);
p_hat_minus[0][0] = temp_4[0][0] + Q[0][0];
p_hat_minus[0][1] = temp_4[0][1] + Q[0][1];
p_hat_minus[1][0] = temp_4[1][0] + Q[1][0];
p_hat_minus[1][1] = temp_4[1][1] + Q[1][1];
// Correction part
// Kalman gain formula
mul(1, 2, 2, 2, C, p_hat_minus, temp_5);
mul(1, 2, 2, 1, temp_5, C_T, temp_6);
temp_6[0][0] = 1.0f / (temp_6[0][0] + R[0][0]);
mul(2, 2, 2, 1, p_hat_minus, C_T, temp_1);
mul(2, 1, 1, 1, temp_1, temp_6, K);
// Posterior estimation formula
mul(1, 2, 2, 1, C, x_hat_minus, temp_6);
temp_6[0][0] = y - temp_6[0][0];
mul(2, 1, 1, 1, K, temp_6, temp_1);
x_hat[0][0] = x_hat_minus[0][0] + temp_1[0][0];
x_hat[1][0] = x_hat_minus[1][0] + temp_1[1][0];
// Update error covariance formula
mul(2, 1, 1, 2, K, C, temp_3);
temp_3[0][0] = I[0][0] - temp_3[0][0];
temp_3[0][1] = I[0][1] - temp_3[0][1];
temp_3[1][0] = I[1][0] - temp_3[1][0];
temp_3[1][1] = I[1][1] - temp_3[1][1];
mul(2, 2, 2, 2, temp_3, p_hat_minus, p_hat);
```

```
// Return value
return x_hat[0][0];
}

/**************************************************************************
Function: Kalman filter
Input: angular velocity, acceleration
Output: None
**************************************************************************/
float KF_Y(float acce_X, float acce_Z, float gyro_Y) // Input: X-axis
acceleration, Z-axis acceleration, Y-axis angular velocity.
{
static float x_hat[2][1] = {0}; // posterior estimate
static float x_hat_minus[2][1] = {0}; // a priori estimate
static float p_hat[2][2] = {{1, 0}, {0, 1}}; // posterior error covariance matrix
static float p_hat_minus[2][2] = {0}; // a priori error covariance matrix
static float K[2][1] = {0}; // Kalman gain
const float Ts = 0.005; // sampling interval (5ms)
const float I[2][2] = {{1, 0}, {0, 1}};
float u[1][1] = {{gyro_Y}};
float A[2][2] = {{1, -Ts}, {0, 1}}; // A matrix
float B[2][1] = {{Ts}, {0}}; // B matrix
float C[1][2] = {{1, 0}}; // C matrix
float Q[2][2] = {{1e-10, 0}, {0, 1e-10}}; // process noise
float R[1][1] = {{1e-4}}; // measurement noise
float A_T[2][2] = {{1, 0}, {-Ts, 1}}; // transpose of A matrix
float C_T[2][1] = {{1}, {0}}; // transpose of C matrix
float temp_1[2][1] = {0}; // used to store intermediate calculation results
float temp_2[2][1] = {0}; // used to store intermediate calculation results
float temp_3[2][2] = {0}; // used to store intermediate calculation results
float temp_4[2][2] = {0}; // used to store intermediate calculation results
float temp_5[1][2] = {0}; // Used to store intermediate calculation results
float temp_6[1][1] = {0}; // Used to store intermediate calculation results
float y = atan2(-acce_X, acce_Z); // Calculate angle using acceleration
// Prediction part
// Prior estimation formula
mul(2, 2, 2, 1, A, x_hat, temp_1);
mul(2, 1, 1, 1, B, u, temp_2);
x_hat_minus[0][0] = temp_1[0][0] + temp_2[0][0];
x_hat_minus[1][0] = temp_1[1][0] + temp_2[1][0];
// Prior error covariance formula
mul(2, 2, 2, 2, A, p_hat, temp_3);
mul(2, 2, 2, 2, temp_3, A_T, temp_4);
p_hat_minus[0][0] = temp_4[0][0] + Q[0][0];
p_hat_minus[0][1] = temp_4[0][1] + Q[0][1];
p_hat_minus[1][0] = temp_4[1][0] + Q[1][0];
p_hat_minus[1][1] = temp_4[1][1] + Q[1][1];
// Correction part
// Kalman gain formula
mul(1, 2, 2, 2, C, p_hat_minus, temp_5);
mul(1, 2, 2, 1, temp_5, C_T, temp_6);
temp_6[0][0] = 1.0f / (temp_6[0][0] + R[0][0]);
mul(2, 2, 2, 1, p_hat_minus, C_T, temp_1);
mul(2, 1, 1, 1, temp_1, temp_6, K);
// Posterior estimation formula
mul(1, 2, 2, 1, C, x_hat_minus, temp_6);
```

```
temp_6[0][0] = y - temp_6[0][0];
mul(2, 1, 1, 1, K, temp_6, temp_1);
x_hat[0][0] = x_hat_minus[0][0] + temp_1[0][0];
x_hat[1][0] = x_hat_minus[1][0] + temp_1[1][0];
// Update error covariance formula
mul(2, 1, 1, 2, K, C, temp_3);
temp_3[0][0] = I[0][0] - temp_3[0][0];
temp_3[0][1] = I[0][1] - temp_3[0][1]; temp_3[1][0] = I[1][0] - temp_3[1][0];
temp_3[1][1] = I[1][1] - temp_3[1][1]; mul(2, 2, 2, 2, temp_3, p_hat_minus,
p_hat); // Return value return x_hat[0][0]; }
```

Software code

Balanced Car PID Control Basics: 08-13 tutorial only provides one project file.

```
Product supporting materials source code path: Attachment → Source code summary →
3.PID_Course → 08-13.Balanced_Car_PID
```