|  |  |
| --- | --- |
| http://robotc.net/forums/images/spacer.gif | **Code:**  //////////////////////////////////////////////////////////////////////////////////////////// // //                                  Self-Balancing Robot // // Original code downloaded from http://popak.org/robot/gyro\_rk4\_v4.c. // and created by Ramin // // ///////////////////////////////////////////////////////////////////////////////////////////// // Robot on two wheels // Change 'bias' to make the robot turn faster or slower (zero for stationary) //  k1, k2, k3, k4 feedback gains are specified below  const tSensors GyroSensor = (tSensors) S1;   //gyro sensor// const tSensors sonar = S3;  const int GyroScale = 4; #define half\_h 2     // Increment used in Runge-Kutta integration const int t\_scale = 500; const int kMinAvoidDistance = 30;  // distance before robot turns  float angleRateChangeBias = 0; float pwrWithFiltering = 0; long nGyroValue; int pwr = 0; float kBatteryAdjustGain;  // This is the gain that is computed adaptively based on the battery voltage (as the battery is drained, the gain is increased)   float angleRateChangeCurr; bool bAvoidingObject = false; int turnOffsetCurr = 0; int turnOffsetPrev; int GyroBias;  // This is the bias applied to my gyro sensor. int nAvoidAdjustment; int nDriveStraightAdjustment;  #define getGyroValue() ((SensorValue(GyroSensor) - GyroBias) / GyroScale)  void calculateGyroBias(); void calculateDriveStraightAndAvoidAdjustments();  task main () {   const bool bUseBatteryAdjust = false;    bFloatDuringInactiveMotorPWM = false;// This causes the motors to stop when they are set to zero    SetSensorType(sonar, sensorSONAR);                // Use the sonar sensor for collisoin avoidance   SetSensorType(GyroSensor, sensorAnalogInactive);    calculateGyroBias();    if (bUseBatteryAdjust)   {     // Measure the battery voltage and compensate for it by adjusting the gain (kBatteryAdjustGain)      int batt;     const int kMaxBattery = 8816;     const int kMinBattery = 8196;      batt = nAvgBatteryLevel;     kBatteryAdjustGain = 0.7 + ((1.1 - 0.7) / (kMaxBattery - kMinBattery)) \* (kMaxBattery - batt);   }    nMotorEncoder[motorC] = 0;   nMotorEncoder[motorA] = 0;     while(true)   {       long f2;      static long f3 = 0;      //     // Runge-Kutta integration (http://en.wikipedia.org/wiki/Runge-kutta)     //     {       static float angleRateChangePrev = 0;       static int timePrev = nPgmTime;       int timeCurr;       long f1;        f1 = f3;             // f(T(n))        wait1Msec(half\_h);        f2 = getGyroValue(); // f(T(n) + h/2)        wait1Msec(half\_h);        f3 = getGyroValue(); // f(T(n+1))        timeCurr = nPgmTime;        angleRateChangeCurr = angleRateChangePrev + (f1 + 2 \* f2) \* (timeCurr - timePrev)/t\_scale;        angleRateChangePrev = angleRateChangeCurr;        timePrev = timeCurr;      }      //     // compute the linear velocity     //      int nBaseSpeed;     int xVelocity;        int xPositionCurr;     {          static int xPositionPrev = 0;        xPositionCurr = nMotorEncoder[motorC];        xVelocity = xPositionCurr - xPositionPrev;        xPositionPrev = xPositionCurr;      }      // Compute the long-term average of tilt change     const float gyroFilterTimeConstant = 1;   // To take care of the gyro drift (range 0..100) [[originally was 0.001 and not 0.01     angleRateChangeBias = (angleRateChangeBias \* (100 -  gyroFilterTimeConstant) + angleRateChangeCurr \* gyroFilterTimeConstant) / 100;      const float kTargetVelocity = 0.5;    // This is the desired velocity of the robot      // These are the feedback loop weighting ("gain") factors       const int k1 = 0;                     // position feedback gain      const int k2 = 90;                    // velocity feedback gain      const int k3 = 8;                     // tilt feedback gain      const int k4 = 10;                    // angular velocity feedback gain      pwr =   k1 \* xPositionCurr                                // position feedback gain           + k2 \* (xVelocity - kTargetVelocity)                // velocity feedback gain           + k3 \* (angleRateChangeCurr - angleRateChangeBias)  // tilt feedback gain           + k4 \* f3;                                          // angular velocity feedback gain      const int a\_pwr = 100;                                                        // power filter constant (range is 0..100)     pwrWithFiltering = ((100 - a\_pwr) \* pwrWithFiltering + (a\_pwr \* pwr)) / 100;  // NOTE: 'a\_pwr' is currently 100!      if (bUseBatteryAdjust)        nBaseSpeed = kBatteryAdjustGain \* pwrWithFiltering;      else        nBaseSpeed = 0.7 \* pwrWithFiltering;      calculateDriveStraightAndAvoidAdjustments();      motor[motorA] = nBaseSpeed - nAvoidAdjustment - nDriveStraightAdjustment;     motor[motorC] = nBaseSpeed + nAvoidAdjustment;   } }   void calculateDriveStraightAndAvoidAdjustments() {   int nDriveStraightErr;   const int k\_d = 16;           // how fast react to obstacles   const float k\_e = 6.5;    // feedback to keeps the two wheels in-sync (i.e. drive straight)    turnOffsetCurr = nMotorEncoder[motorA] - nMotorEncoder[motorC];   nDriveStraightErr = turnOffsetCurr - turnOffsetPrev;   turnOffsetPrev = turnOffsetCurr;    if (bAvoidingObject)   {     nDriveStraightAdjustment =  0;     nAvoidAdjustment         =  k\_d;      if (time1(T3) > 2500)        bAvoidingObject = false;    }    else   {     nDriveStraightAdjustment =  k\_e \* nDriveStraightErr;     nAvoidAdjustment         =  0;      if (SensorRaw[sonar] < kMinAvoidDistance)     {       // are we close to an obstacle?       bAvoidingObject = true;       ClearTimer(T3);   // If yes, turn for 2.5 second     }   }    return; }   void calculateGyroBias() {   // Find the gyro bias associated w/ the balanced position   // Hold the robot in the balanced position for 3 sec to find the gyro bias    const bool bCalculateDynamically = false;    if (bCalculateDynamically)   {     GyroBias = 0;      ClearTimer(T1);      while (time1[T1] < 3000)      {        // filter the sensor output         const int kGyroFilter = 2;   // 0.2 with divisor of 10 to avoid floating point        nGyroValue = SensorValue(GyroSensor);        wait1Msec(100);        GyroBias = ((10 - kGyroFilter) \* GyroBias + kGyroFilter \* nGyroValue) / 10;      }    }    else    {      // I ended up hard-coding the bias after measuring it a few times.      // comment out the following line if you want the Gyro bias to be measured adaptively      GyroBias = 600.5;    }    PlaySound(soundBlip);   // play a sound when the training is over   return; } |

**Code:**

////////////////////////////////////////////////////////////////////////////////////////////  
//  
//                                  Self-Balancing Robot  
//  
// Original code downloaded from http://popak.org/robot/gyro\_rk4\_v4.c.  
// and created by Ramin  
//  
//  
/////////////////////////////////////////////////////////////////////////////////////////////  
// Robot on two wheels  
// Change 'bias' to make the robot turn faster or slower (zero for stationary)  
//  k1, k2, k3, k4 feedback gains are specified below  
  
const tSensors GyroSensor = (tSensors) S1;   //gyro sensor//  
const tSensors sonar = S3;  
  
const int GyroScale = 4;  
#define half\_h 2     // Increment used in Runge-Kutta integration  
const int t\_scale = 500;  
const int kMinAvoidDistance = 30;  // distance before robot turns  
  
float angleRateChangeBias = 0;  
float pwrWithFiltering = 0;  
long nGyroValue;  
int pwr = 0;  
float kBatteryAdjustGain;  // This is the gain that is computed adaptively based on the battery voltage (as the battery is drained, the gain is increased)  
  
  
float angleRateChangeCurr;  
bool bAvoidingObject = false;  
int turnOffsetCurr = 0;  
int turnOffsetPrev;  
int GyroBias;  // This is the bias applied to my gyro sensor.  
int nAvoidAdjustment;  
int nDriveStraightAdjustment;  
  
#define getGyroValue() ((SensorValue(GyroSensor) - GyroBias) / GyroScale)  
  
void calculateGyroBias();  
void calculateDriveStraightAndAvoidAdjustments();  
  
task main ()  
{  
  const bool bUseBatteryAdjust = false;  
  
  bFloatDuringInactiveMotorPWM = false;// This causes the motors to stop when they are set to zero  
  
  SetSensorType(sonar, sensorSONAR);                // Use the sonar sensor for collisoin avoidance  
  SetSensorType(GyroSensor, sensorAnalogInactive);  
  
  calculateGyroBias();  
  
  if (bUseBatteryAdjust)  
  {  
    // Measure the battery voltage and compensate for it by adjusting the gain (kBatteryAdjustGain)  
  
    int batt;  
    const int kMaxBattery = 8816;  
    const int kMinBattery = 8196;  
  
    batt = nAvgBatteryLevel;  
    kBatteryAdjustGain = 0.7 + ((1.1 - 0.7) / (kMaxBattery - kMinBattery)) \* (kMaxBattery - batt);  
  }  
  
  nMotorEncoder[motorC] = 0;  
  nMotorEncoder[motorA] = 0;  
  
  
  while(true)  
  {  
      long f2;  
     static long f3 = 0;  
  
    //  
    // Runge-Kutta integration (http://en.wikipedia.org/wiki/Runge-kutta)  
    //  
    {  
      static float angleRateChangePrev = 0;  
      static int timePrev = nPgmTime;  
      int timeCurr;  
      long f1;  
  
      f1 = f3;             // f(T(n))  
       wait1Msec(half\_h);  
       f2 = getGyroValue(); // f(T(n) + h/2)  
       wait1Msec(half\_h);  
       f3 = getGyroValue(); // f(T(n+1))  
       timeCurr = nPgmTime;  
       angleRateChangeCurr = angleRateChangePrev + (f1 + 2 \* f2) \* (timeCurr - timePrev)/t\_scale;  
       angleRateChangePrev = angleRateChangeCurr;  
       timePrev = timeCurr;  
     }  
  
    //  
    // compute the linear velocity  
    //  
     int nBaseSpeed;  
    int xVelocity;  
       int xPositionCurr;  
  
   {  
         static int xPositionPrev = 0;  
  
      xPositionCurr = nMotorEncoder[motorC];  
       xVelocity = xPositionCurr - xPositionPrev;  
       xPositionPrev = xPositionCurr;  
     }  
  
    // Compute the long-term average of tilt change  
    const float gyroFilterTimeConstant = 1;   // To take care of the gyro drift (range 0..100) [[originally was 0.001 and not 0.01  
    angleRateChangeBias = (angleRateChangeBias \* (100 -  gyroFilterTimeConstant) + angleRateChangeCurr \* gyroFilterTimeConstant) / 100;  
  
    const float kTargetVelocity = 0.5;    // This is the desired velocity of the robot  
  
    // These are the feedback loop weighting ("gain") factors  
  
     const int k1 = 0;                     // position feedback gain  
     const int k2 = 90;                    // velocity feedback gain  
     const int k3 = 8;                     // tilt feedback gain  
     const int k4 = 10;                    // angular velocity feedback gain  
  
    pwr =   k1 \* xPositionCurr                                // position feedback gain  
          + k2 \* (xVelocity - kTargetVelocity)                // velocity feedback gain  
          + k3 \* (angleRateChangeCurr - angleRateChangeBias)  // tilt feedback gain  
          + k4 \* f3;                                          // angular velocity feedback gain  
  
    const int a\_pwr = 100;                                                        // power filter constant (range is 0..100)  
    pwrWithFiltering = ((100 - a\_pwr) \* pwrWithFiltering + (a\_pwr \* pwr)) / 100;  // NOTE: 'a\_pwr' is currently 100!  
  
    if (bUseBatteryAdjust)  
       nBaseSpeed = kBatteryAdjustGain \* pwrWithFiltering;  
     else  
       nBaseSpeed = 0.7 \* pwrWithFiltering;  
  
    calculateDriveStraightAndAvoidAdjustments();  
  
    motor[motorA] = nBaseSpeed - nAvoidAdjustment - nDriveStraightAdjustment;  
    motor[motorC] = nBaseSpeed + nAvoidAdjustment;  
  }  
}  
  
  
void calculateDriveStraightAndAvoidAdjustments()  
{  
  int nDriveStraightErr;  
  const int k\_d = 16;           // how fast react to obstacles  
  const float k\_e = 6.5;    // feedback to keeps the two wheels in-sync (i.e. drive straight)  
  
  turnOffsetCurr = nMotorEncoder[motorA] - nMotorEncoder[motorC];  
  nDriveStraightErr = turnOffsetCurr - turnOffsetPrev;  
  turnOffsetPrev = turnOffsetCurr;  
  
  if (bAvoidingObject)  
  {  
    nDriveStraightAdjustment =  0;  
    nAvoidAdjustment         =  k\_d;  
  
    if (time1(T3) > 2500)  
       bAvoidingObject = false;  
   }  
   else  
  {  
    nDriveStraightAdjustment =  k\_e \* nDriveStraightErr;  
    nAvoidAdjustment         =  0;  
  
    if (SensorRaw[sonar] < kMinAvoidDistance)  
    {  
      // are we close to an obstacle?  
      bAvoidingObject = true;  
      ClearTimer(T3);   // If yes, turn for 2.5 second  
    }  
  }  
  
  return;  
}

**Code:**

task main()  
{  
   while(1)  
   {  
      bmotorflippedmode[port1] = true  
      bmotorflippedmode[port10] = true  
                  
      //Both my motors are upsidedown :D  
      motor[port1] = vexRT[AccelY] + vexRT[AccelX];  
      motor[port10] = vexRT[AccelY] - vexRT[AccelX];  
   }  
}