March 21, 2012, 8:43 pm

In [part one](http://www.kerrywong.com/2012/03/08/a-self-balancing-robot-i/), I went over the theory behind the self-balancing robot and in [part two](http://www.kerrywong.com/2012/03/14/a-self-balancing-robot-ii/), I showed you my build using a modified toy truck. In this post, I will walk you through the code and show you the finished balancing robot. If you are impatient, you can scroll to the end for the demostration video and the source code.

The code uses the [Arduino](http://arduino.cc/) framework and is built using the [Netbeans IDE](http://netbeans.org/). For tutorials on how to develop Arduino applications using Netbeans, please take a look at my previous posting [here](http://www.kerrywong.com/2011/12/17/makefile-for-arduino-1-0/). If you prefer the standard Arduino IDE, you may need to change the include statements to reflect your Arduino library path setup. The entire code listing can be downloaded [here](http://www.kerrywong.com/blog/wp-content/uploads/2012/03/balancingbot.tar.gz). Now let’s move on to the code.

**Gyroscope calibration**

[LPY450AL](http://www.st.com/internet/com/TECHNICAL_RESOURCES/TECHNICAL_LITERATURE/DATASHEET/CD00254143.pdf) is an analog gyroscope and it is fairly easy to interface with the MCU using ATMega328’s analog inputs. Before performing any measurements, the gyroscope needs to be calibrated so we know where the zero point is (i.e. when there is no rotation). To do this, the following *CalGyro* routine is called right upon powering up within *setup()*.

In order to ensure accuracy, the platform must be stationary as any movement will likely affect the zero position readings of the gyroscope. As you can see, the code basically reads and averages the axis’ outputs while the robot is stationary and thus the raw ADC voltage value obtained represents the gyroscope output value when there is no rotation. The readings are averaged over 25 iterations to minimize errors. An LED controlled via PIN\_MODE\_LED is used to indicate whether the robot is undergoing the calibration process. So it is important for the robot to be stationary when the mode LED is lit, otherwise the calibrated value would be inaccurate.

[?](http://www.kerrywong.com/2012/03/21/a-self-balancing-robot-iii/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | float CalGyro() {      float val = 0.0;      digitalWrite(PIN\_MODE\_LED, HIGH);      delay(500);      for (int i = 0; i < 25; i++) {          val += analogRead(PIN\_GYRO\_X);          delay(10);      }        val /= 25.0;        Serial.print("GyroX Stationary=");      Serial.println(val);      digitalWrite(PIN\_MODE\_LED, LOW);        return val;  } |

**Inclination angle estimation**

The following routine reads the current angle measurement from the accelerometer. I discussed the code for [MMA8453Q](http://www.freescale.com/files/sensors/doc/data_sheet/MMA8453Q.pdf) in [one of my previous post](http://www.kerrywong.com/2012/01/09/interfacing-mma8453q-with-arduino/). So if you wanted to learn a bit more about this accelerometer, you can take a look there.

If you recall from our discussion in [the first part](http://www.kerrywong.com/2012/03/08/a-self-balancing-robot-i/), the inclination angle measured by the accelerometer is given by:

http://www.kerrywong.com/blog/jsMath/fonts/cmmi10/alpha/100/char12.pnghttp://www.kerrywong.com/blog/jsMath/fonts/cmsy10/alpha/100/char19.png*sin*(http://www.kerrywong.com/blog/jsMath/fonts/cmmi10/alpha/100/char12.png)=*gAx*… (1)

Where *Ax* is the accelerometer *x* axis reading. In the implementation below, the axis used was slightly different due to the placement and the orientation of the accelerometer, but the calculation is the same:

[?](http://www.kerrywong.com/2012/03/21/a-self-balancing-robot-iii/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | float GetAccAngle() {      getAccXYZ(&VxAcc, &VzAcc, &VyAcc);        float r = sqrt((float) VzAcc \* (float) VzAcc + (float) VyAcc \* (float) VyAcc);      accAngle = (float) VzAcc / r; //approximates sine        return accAngle;  } |

The angle estimation routine using the [IMU](http://en.wikipedia.org/wiki/Inertial_measurement_unit) (both accelerometer and gyroscope) is pretty straight forward as well. If you recall from our[previous discussion](http://www.kerrywong.com/2012/03/08/a-self-balancing-robot-i/), we used the following equations to calculate the estimated inclination angle in radians using the measurements from both the accelerometer and the gyroscpe:

http://www.kerrywong.com/blog/jsMath/fonts/cmmi10/alpha/100/char12.png(*t*)http://www.kerrywong.com/blog/jsMath/fonts/cmsy10/alpha/100/char19.pnghttp://www.kerrywong.com/blog/jsMath/fonts/cmmi10/alpha/100/char12.png(*t*1)+*G*(*t*)(*t*2–*t*1)=http://www.kerrywong.com/blog/jsMath/fonts/cmmi10/alpha/100/char12.png(*t*1)+*G*(*t*)http://www.kerrywong.com/blog/jsMath/fonts/cmr10/alpha/100/char01.png*t*… (2)

http://www.kerrywong.com/blog/jsMath/fonts/cmex10/alpha/100/char28.pnghttp://www.kerrywong.com/blog/jsMath/fonts/cmmi10/alpha/100/char12.png*est*http://www.kerrywong.com/blog/jsMath/fonts/cmsy10/alpha/100/char6A.png*t*2=http://www.kerrywong.com/blog/jsMath/fonts/cmmi10/alpha/100/char0B.png(http://www.kerrywong.com/blog/jsMath/fonts/cmmi10/alpha/100/char12.png*est*http://www.kerrywong.com/blog/jsMath/fonts/cmsy10/alpha/100/char6A.png*t*1+*G*http://www.kerrywong.com/blog/jsMath/fonts/cmr10/alpha/100/char01.png*t*)+http://www.kerrywong.com/blog/jsMath/fonts/cmmi10/alpha/100/char0C.pnghttp://www.kerrywong.com/blog/jsMath/fonts/cmmi10/alpha/100/char12.png*Ax* http://www.kerrywong.com/blog/jsMath/fonts/cmmi10/alpha/100/char0B.png+http://www.kerrywong.com/blog/jsMath/fonts/cmmi10/alpha/100/char0C.png=1 … (3) … (4)

And as we discussed previously, the above method is essentially [a special case of the more generic Kalman filter](http://proyectos.ciii.frc.utn.edu.ar/cuadricoptero/browser/articulo_case_filtro_comp/referencias/A%20comparison%20of%20complementary%20and%20kalman%20filtering.pdf?rev=7a5787cfd2f0d9b174a451d421f6dcfc5dfd4467). And here is the implementation of the equations above:

[?](http://www.kerrywong.com/2012/03/21/a-self-balancing-robot-iii/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | float EstimateAngle() {      currTimeStamp = micros();      elapsedTime = currTimeStamp - prevTimeStamp;      prevTimeStamp = currTimeStamp;        elapsedTimeSec = (float) elapsedTime / 1e6f;        float angle = GetAccAngle();      if (isFirstTime) {          currAngle = angle;          isFirstTime = false;      } else {          VxGyro = analogRead(PIN\_GYRO\_X);          degX = (float) (VxGyro - GYRO\_X\_INIT\_VAL) \* 3300.0f / 512.0f / 180.0f \* (float) elapsedTimeSec;          currAngle = 0.95 \* (prevAngle + degX) + 0.05 \* angle;      }        prevAngle = currAngle;        //Debugging code omitted here, see attached source for the full code listings.        return currAngle;  } |
|  |

The elapsed time is calculated by measuring the time spent between two consecutive function calls to *EstimateAngle()*. Alternatively, you can use hardware timer interrupt in your code so that the time interval is precisely known beforehand. Because the readings from the gyroscope come from the ADC directly, they will need to be converted to the actual angles being measured.

In the code above, *degX* records the rotated angle within the measurement time interval. Since the ADC resolution is 10 bit on ATmega328, the supply voltage is 3.3V (3300 mV) and the 4X gyroscope output sensitivity is 2mV/degree, the degree measured in radians can be calculated as:

http://www.kerrywong.com/blog/jsMath/fonts/cmmi10/alpha/100/char12.png=210http://www.kerrywong.com/blog/jsMath/fonts/cmsy10/alpha/100/char02.png180*Vgyro*http://www.kerrywong.com/blog/jsMath/fonts/cmsy10/alpha/100/char02.png3300http://www.kerrywong.com/blog/jsMath/fonts/cmsy10/alpha/100/char02.png2http://www.kerrywong.com/blog/jsMath/fonts/cmr10/alpha/100/char01.png*t*=512http://www.kerrywong.com/blog/jsMath/fonts/cmsy10/alpha/100/char02.png180*Vgyro*http://www.kerrywong.com/blog/jsMath/fonts/cmsy10/alpha/100/char02.png3300http://www.kerrywong.com/blog/jsMath/fonts/cmr10/alpha/100/char01.png*t*

We could’ve just used the above formula as the final estimate for the inclination angle as the above calculation matches the filtering algorithm we discussed earlier.

In my particular implementation, I added another low-pass filter for the combined sensor output in order to achieve better stability. The reason for adding this low-pass filter is as the following:

* The control loop is relatively fast (e.g. 10ms per iteration). Since the time constants for the motors tend to be much longer, a 50ms control loop time interval is more than adequate. In this case, we could just add some delay or use a hardware timer to make the interval longer. But since the angular acceleration is not a constant during the measurement interval, we will loose some accuracy in the inclination angle calculation if the time interval is too long.
* The wheels of the toy truck I used are roughly 3 inches in diameter, which are pretty small compared to the size of the robot. The small wheel size makes it more sensitive to the surface conditions, which translates into larger variations in sensor readings. By averaging over several readings, we can decrease the sensor variations.

That said, this last step may not be necessary in your case and you can simply set the ANGLE\_BUFFER\_LENGTH to 1 to bypass the buffer. The code for calculating the average is shown below. The variable *angleBuffer* is a [circular buffer](http://en.wikipedia.org/wiki/Circular_buffer) and the buffer values are updated within the main loop.

[?](http://www.kerrywong.com/2012/03/21/a-self-balancing-robot-iii/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | float GetAvgAngle() {      int count = 0;      float a = 0;        for (int i = 0; i < ANGLE\_BUFFER\_LENGTH; i++) {          if (angleBuffer[i] != 0) {              count++;              a += angleBuffer[i];          }      }        if (count > 0)          return a / float(count);      else          return 0.0;  } |

**The PID control loop**

Motors can be controlled via the classic [PID control](http://en.wikipedia.org/wiki/PID_controller). Microchip’s application notes [AN964](http://ww1.microchip.com/downloads/en/AppNotes/00964A.pdf) has some excellent explanation on the physics and math behind the PID control, and the notions in my implementation mirrors those in the application notes.

Here is the code for the main loop:

|  |  |
| --- | --- |
| 2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | float GetAvgAngle() {      int count = 0;      float a = 0;        for (int i = 0; i < ANGLE\_BUFFER\_LENGTH; i++) {          if (angleBuffer[i] != 0) {              count++;              a += angleBuffer[i];          }      }        if (count > 0)          return a / float(count);      else          return 0.0;  } |

**The PID control loop**

Motors can be controlled via the classic [PID control](http://en.wikipedia.org/wiki/PID_controller). Microchip’s application notes [AN964](http://ww1.microchip.com/downloads/en/AppNotes/00964A.pdf) has some excellent explanation on the physics and math behind the PID control, and the notions in my implementation mirrors those in the application notes.

Here is the code for the main loop:

[?](http://www.kerrywong.com/2012/03/21/a-self-balancing-robot-iii/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | void loop() {      float a = EstimateAngle() - INIT\_ANGLE;        angleBuffer[angleBufferIndex] = a;      angleBufferIndex = (angleBufferIndex + 1) % ANGLE\_BUFFER\_LENGTH;      float ang = GetAvgAngle();        curErr = ang - INIT\_ANGLE; //error      SumErr += curErr;        if (SumErr > SumErrMax) SumErr = SumErrMax;      else if (SumErr < SumErrMin) SumErr = SumErrMin;        //Ki\*SumE/(Kp\*Fs\*X)      integralTerm = SumErr \* elapsedTimeSec \* Ki / Kp \* 10.0;      derivativeTerm = curErr - prevErr;        if(derivativeTerm > 0.1) derivativeTerm = 0.1;      else if (derivativeTerm < -0.1) derivativeTerm = -0.1;        // Kd(curErr-prevErr)\*Ts/(Kp\*X)      derivativeTerm = derivativeTerm \* Kd \* elapsedTimeSec / Kp;        if(derivativeTerm > 120) derivativeTerm = 120;      else if (derivativeTerm < -120) derivativeTerm = -120;        Cn = (curErr + integralTerm + derivativeTerm) \* Kp / 10.0;      Serial.println(Cn);      WheelDirection dir;        if (Cn > 0) dir = DIR\_FORWARD;      else if (Cn < -0) dir = DIR\_BACKWARD;      else dir = DIR\_STOP;        throttle = abs(Cn);        if (abs(ang) > 0.7)  MoveWheels(DIR\_STOP, 0); //if angle too large to correct, stop motor      else MoveWheels(dir, throttle);        prevErr = curErr;  } |

In plain English, the control loop works like this:

* The current inclination angle is estimated and stored in the circular buffer. The averaged inclination angle measurement is fed into the PID controller. As mentioned earlier, if no averaging is needed, the current inclination estimate can be used directly by setting the buffer length to 1.
* Parameters of the PID controller is calculated using the current error measurement (*curErr*) and the cumulative error (*SumErr*).
* We use the output of the PID controller (*Cn*) as the “throttle” to control the motors. The sign of the *Cn* corresponds to the motor rotation direction.
* If the inclination angle is too large (e.g. greater than roughly 40 degree), there is no way for the robot to correct its position anyway so we will simply stop the motor. This condition is needed to prevent the wheels from moving when the robot is positioned horizontally.
* The current error term is assigned to *prevErr* and now we are ready for the next loop.

|  |
| --- |
|  |

**Tuning and trouble-shooting**

Now it’s time to power on the robot. Don’t expect it will work the very first time even if you did the programming correctly since there are many parameters involved in the whole process. Here are a few simple steps that you could use to ensure that the robot is functioning correctly and making parameter adjustments so that it could balance.

1. Set the mode switch so that the motor is turned off and the debug information can be viewed via a serial console.
2. Power up the robot and lie it flat, the angle reported from the accelerometer should be close to ±1 depending on which side is facing up (if you are using sine instead of approximating it the value read should be close to ±Pi/2) and the gyroscope reading should be around 0 since the robot is stationary.
3. Hold the robot up straight to its balanced position, the accelerometer angle reading should be close to 0 and the gyroscope reading should remain around 0 when there is no movement. Pay attention to the angle reading when the robot is balanced, it may be slightly off zero due to the actual weight distribution of the robot. And if this value is none-zero, you will need to set the INIT\_ANGLE to the observed value when the robot is in-balance so that this offset value can be subtracted from the actual measurements.
4. Now try move the robot back and forth about its balanced position, you should see both the accelerometer and the gyroscope readings oscillate around 0. Pay special attention to the sign of the gyroscope readings. It should match the sign of the accelerometer reading when the the robot is moving away from its balanced position.
5. Now set the mode switch to normal operation mode. Upon powering up, hold the robot in the air and move it gently around its balanced position. You should see the wheels move towards the direction of the inclination. And the motor should spin faster when the inclination angle is larger.

If the above test steps are successful, we are already well on our way to a successful build. Now it’s time to tune the PID controller. The easiest way to tune a PID controller is to tune the *P*, *I* and *D* parameters one at a time. You can refer to Microchip’s application notes [AN964](http://ww1.microchip.com/downloads/en/AppNotes/00964A.pdf) for more details. Basically, you will need to:

1. Set *I* and *D* term to 0, and adjust *P* so that the robot starts to oscillate (move back and forth) about the balance position. *P*should be large enough for the robot to move but not too large otherwise the movement would not be smooth.
2. With *P* set, increase *I* so that the robot accelerates faster when off balance. With *P* and *I* properly tuned, the robot should be able to self-balance for at least a few seconds.
3. Finally, increase *D* so that the robot would move about its balanced position more gentle, and there shouldn’t be any significant overshoots.

There are also a couple of constants in the *I* and *D* term calculations, they will need to be adjusted by experiments to determine the optimal values.

If you are unable to balance the robot with the PID parameter adjustments mentioned above, chances are that the accelerometer or the gyroscope values are not calculated correctly and you will need to go back to the basic calibration steps mentioned earlier and see whether the readings from gyroscope and the accelerometer are handled properly. One common mistake is the gyroscope angle calculation, you will need to use the correct sensitivity according to the datasheet of the particular gyroscope you are using along with the formula we discussed earlier to come up with the correct angle.

The choice of the values of *a* and *b* in equation (4) affects the time constants of the control and can be fine tuned with the control loop interval and ANGLE\_BUFFER\_LENGTH parameter to achieve the best result. In my case the weight I used is 0.95 for the gyroscope readings and 0.05 for the accelerometer readings. The ANGLE\_BUFFER\_LENGTH setting and the value for *b* affect the time constant (or low pass filter cutoff frequency) in opposite manner. When you increase the buffer length, the time constant becomes higher (cutoff frequency is lower). But when *b* is incremented, the accelerometer’s time constant actually becomes lower (cutoff frequency is higher).

Here are a couple of pictures of the robot in its balanced positions:

|  |  |
| --- | --- |
| [Self-Balancing Robot in Action 1](http://www.kerrywong.com/blog/wp-content/uploads/2012/03/BalancingRobot1.jpg)  Self-Balancing Robot in Action 1 | [Self-Balancing Robot in Action 2](http://www.kerrywong.com/blog/wp-content/uploads/2012/03/BalancingRobot2.jpg)  Self-Balancing Robot in Action 2 |

And the following is a short video of the balancing robot in action. The stability of the robot can be dramatically improved if you use a properly designed gearbox that has negligible gear backlash. The backlash in a typical toy gearbox is usually quite significant. In my case, the wheels can be turned by a few degrees in either direction before the gears are engaged, and this makes it harder to control the robot accurately. Also as mentioned earlier, larger wheels can also improve stability as the robot will be less affected by the surface condition.